CSE341, Fall 2011, Lecture 14 Summary

Standard Disclaimer: This lecture summary is not necessarily a complete substitute for attending class, reading the associated code, etc. It is designed to be a useful resource for students who attended class and are later reviewing the material.

Switching from ML to Racket

For the next couple weeks, we will use the Racket programming language (instead of SML) and the Dr-Racket programming environment (instead of SML/NJ and emacs). Notes on installation and basic usage instructions are on the course website in a different document than this one.

Our focus will remain largely on key programming language constructs. We will "switch" to Racket because some of these concepts shine better in Racket. That said, Racket and ML share many similarities: They are both mostly functional languages (i.e., mutation exists but is discouraged) with closures, anonymous functions, convenient support for lists, no return statements, etc. Seeing these features in a second language should help re-enforce the underlying ideas. One moderate difference is that we will not use pattern matching in Racket.

For us, the most important differences between Racket and ML are:

- Racket does not use a static type system. So it accepts more programs and programmers do not need to define new types all the time, but most errors do not occur until run time.
- Racket has a very minimalist and uniform syntax.

Racket has many advanced language features, including macros, a module system quite different from ML, quoting/eval, first-class continuations, contracts, and much more. We will have time to cover only a couple of these topics.

This lecture is a bit more basic since we need to introduce Racket before we start using it to study more advanced concepts. After seeing some functional programming and list-processing in Racket, we will discuss the syntax and how parentheses matter. The most interesting topic in this lecture is Racket's different rules for creating local environments using `let` or `let*` or `letrec` or local definitions. Finally we discuss how a Racket file (which is by default a module) allows forward and backward references much like `letrec` and local definitions not at top-level.

Racket vs. Scheme

Racket is derived from Scheme, a well-known programming language that has evolved since 1975. (Scheme in turn is derived from LISP, which has evolved since 1958 or so.) The designers of Racket decided in 2010 that they wanted to make enough changes and additions to Scheme that it made more sense to give the result a new name than just consider it a dialect of Scheme. The two languages remain *very* similar with a short list of key differences (how the empty list is written, whether cons cells are mutable, how modules work), a longer list of minor differences, and a longer list of additions that Racket provides.

Overall, Racket is a modern language under active development that has been used to build several "real" (whatever that means) systems. The improvements over Scheme make it a good choice for this course and for real-world programming. However, it is more of a "moving target" — the designers do not feel as bound to historical precedent as they try to make the language and the accompanying DrRacket system better. So details in the course materials are more likely to become outdated.

We will not use a textbook for this portion of the course, but "The Racket Guide" on the Racket website is a useful and well-written resource for programmers new to Racket but not new to programming.

Getting Started: Definitions, Functions, Lists (and `if`)

The first line of a Racket file (which is also a Racket module) should be

```
#lang racket
```

This is discussed in the installation/usage instructions for the course. These lecture notes will focus instead on the content of the file after this line. A Racket file contains a collection of definitions.

A definition like

```
(define a 3)
```

extends the top-level environment so that `a` is bound to 3. Racket has very lenient rules on what characters can appear in a variable name, and a common convention is hyphens to separate words like `my-favorite-identifier`.

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, although at the end of the lecture we will discuss that, unlike ML, bindings can refer to later bindings in the file. In Racket, *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 Racket, 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 can discuss in another lecture how to define our own variable-number-of-arguments functions.

Unlike ML, you can use recursion with anonymous functions because the definition itself is in scope in the function body:

```
(define pow
  (lambda (x y)
    (if (= y 0)
        1
        (* x (pow x (- y 1))))))
```

The above example also used an if-expression, which has the general syntax `(if e1 e2 e3)`. It evaluates `e1`. If the result is `#f` (Racket's constant for false), it evaluates `e3` for the result. If the result is *anything else*, including `#t` (Racket's constant for true), it evaluates `e2` for the result. Notice how this is much more flexible type-wise than anything in ML.

There is a very common form of syntactic sugar you should use for defining functions that does not use the word `lambda` explicitly:

```
(define (cube3 x)
  (* x x x))
(define (pow x y)
    (if (= y 0)
        1
        (* x (pow x (- y 1)))))
```

This is more like ML's `fun` binding, but in ML `fun` is not just syntactic sugar since it is necessary for recursion.

We can use currying in Racket. After all, Racket's first-class functions are closures like in ML and currying is just a programming idiom.

```
(define pow
  (lambda (x)
    (lambda (y)
      (if (= y 0)
          1
          (* x ((pow x) (- y 1)))))))

(define three-to-the (pow 3))
(define eightyone (three-to-the 4))
(define sixteen ((pow 2) 4))
```

Because Racket's multi-argument functions really are multi-argument functions (not sugar for something else), currying is not as common. There is no syntactic sugar for calling a curried function: we have to write `((pow 2) 4)` because `(pow 2 4)` calls the one-argument function bound to `pow` with two arguments, which is a run-time error. Racket has added sugar for *defining* a curried function. We could have written:

```
(define ((pow x) y)
   (if (= y 0)
       1
       (* x (pow x (- y 1)))))
```

This is a fairly new feature and may not be widely known.

Racket has built-in lists, much like ML, and Racket programs probably use lists even more often in practice than ML programs. We will use built-in functions for building lists, extracting parts, and seeing if lists are empty. The function names `car` and `cdr` are a historical accident.

Primitive	Description	Example
`null`	The empty list	`null`
`cons`	Construct a list	`(cons 2 (cons 3 null))`
`car`	Get first element of a list	`(car some-list)`
`cdr`	Get tail of a list	`(cdr some-list)`
`null?`	Return `#t` for the empty-list and `#f` otherwise	`(null? some-value)`

Unlike Scheme, you cannot write `()` for the empty list. You can write `'()`, but we will prefer `null`.

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 null)))`. Lists need not hold elements of the same type, so you can create `(list #t "hi" 14)` without error.

Here are three examples of list-processing functions. `map` and `append` are actually provided by default, so we would not write our own.

```
(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs) (sum (cdr xs)))))

(define (append xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (append (cdr xs) ys))))

(define (map f xs)
  (if (null? xs)
      null
      (cons (f (car xs)) (map f (cdr xs)))))
```

Syntax and Parentheses

Ignoring a few bells and whistles, Racket has an amazingly simple syntax. Everything in the language is either:

- Some form of *atom*, such as `#t`, `#f`, `34`, `"hi"`, `null`, etc. A particularly important form of atom is an identifier, which can either be a variable (e.g., `x` or `something-like-this!`) or a *special form* such as `define`, `lambda`, `if`, and many more.
- A sequence of things in parentheses `(t1 t2 ... tn)`.

The first thing in a sequence affects what the rest of the sequence means. For example, `(define ...)` means we have a definition and the next thing can be a variable to be defined or a sequence for the sugared version of function definitions.

If the first thing in a sequence is not a special form and the sequence is part of an expression, then we have a function call. Many things in Racket are just functions, such as `+` and `>`.

As a minor note, Racket also allows `[` and `]` in place of `(` and `)` anywhere. As a matter of style, there are a few places we will show where `[...]` is the common preferred option. Racket does *not* allow mismatched parenthesis forms: `(` must be matched by `)` and `[` by `]`. DrRacket makes this easy because if you type `)` to match `[`, it will enter `]` instead.

By "parenthesizing everything" Racket has a syntax that is *unambiguous*. There are never any rules to learn about whether `1+2*3` is `1+(2*3)` or `(1+2)*3` and whether `f x y` is `(f x) y` or `f (x y)`. It makes *parsing*, converting the program text into a tree representing the program structure, trivial. Notice that XML-based languages like HTML take the same approach. In HTML, an "open parenthesis" looks like `<foo>` and the matching close-parenthesis looks like `</foo>`.

For some reason, HTML is only rarely criticized for being littered with parentheses but it is a common complaint leveled against LISP, Scheme, and Racket. If you stop a programmer on the street and ask him

or her about these languages, they may well say something about "all those parentheses." This is a bizarre obsession: people who use these languages quickly get used to it and find the uniform syntax pleasant. For example, it makes it very easy for the editor to indent your code properly.

From the standpoint of learning about programming languages and fundamental programming constructs, you should recognize a strong opinion about parentheses (either for or against) as a syntactic prejudice. While everyone is entitled to a personal opinion on syntax, one should not allow it to keep you from learning advanced ideas that Racket does well, like hygienic macros or abstract datatypes in a dynamically typed language or first-class continuations. An analogy would be if a student of European history did not want to learn about the French Revolution because he or she was not attracted to people with french accents.

All that said, practical programming in Racket does require you to get your parentheses correct and Racket differs from ML, Java, C, etc. in an important regard: *Parentheses change the meaning of your program. You cannot add or remove them because you feel like it. They are never optional or meaningless.*

In expressions, `(e)` means evaluate `e` and then call the resulting function with 0 arguments. So `(42)` will be a run-time error: you are treating the number 42 as a function. Similarly, `((+ 20 22))` is an error for the same reason.

Programmers new to Racket sometimes struggle with remembering that parentheses matter and determining why programs fail, often at run-time, when they are misparenthesized. As an example consider these seven 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
(define (fact n) (if (= n 0) 1 (n * (fact (- n 1)))))   ; 7
```

Line	Error
2	calls `1` as a function taking no arguments
3	uses `if` with 5 subexpressions instead of 3
4	bad definition syntax: `(n)` looks like an expression followed by more stuff
5	calls `*` with a function as one of the arguments
6	calls `fact` with 0 arguments
7	treats `n` as a function and calls it with `*`

Dynamic Typing (and `cond`)

Racket does not use a static type system to reject programs before they are run. As an extreme example, the function `(lambda () (1 2))` is a perfectly fine zero-argument function that will cause an error if you ever call it. We will spend significant time in a later lecture comparing dynamic and static typing and their relative benefits, but for now we want to get used to dynamic typing.

As an example, suppose we want to have lists of numbers but where some of the elements can actually be other lists that themselves contain numbers or other lists and so on, any number of levels deep. Racket allows this directly, e.g., `(list 2 (list 4 5) (list (list 1 2) (list 6)) 19 (list 14 0))`. In ML, such an expression would not type-check; we would need to create our own datatype binding and use the correct constructors in the correct places.

Now in Racket suppose we wanted to compute something over such lists. Again this is no problem. For example, here we define a function to sum all the numbers anywhere in such a data structure:

```
(define (sum xs)
  (if (null? xs)
      0
      (if (number? (car xs))
          (+ (car xs) (sum (cdr xs)))
          (+ (sum (car xs)) (sum (cdr xs))))))
```

This code simply uses the built-in *predicates* for empty-lists (`null?`) and numbers (`number?`). The last line assumes `(car xs)` is a list; if it is not, then the function is being misused and we will get a run-time error.

We now digress to introduce the `cond` special form, which is better style for nested conditionals than actually using multiple if-expressions. We can rewrite the previous function as:

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

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 us take dynamic typing one step further and change the specification for our `sum` function. Suppose we even want to allow non-numbers and non-lists in our lists (or as the argument to `sum` itself) in which case we just want to "ignore" such elements by adding 0 to the sum. If this is what you want (and it may not be — it could silently hide mistakes in your program), then we can do that in Racket. This code will never raise an error:

```
(define (sum arg)
  (cond [(null? arg) 0]
        [(number? arg) arg]
        [(list? arg) (+ (sum (car arg)) (sum (cdr arg)))]
        [#t 0]))
```

Local bindings: `let`, `let*`, `letrec`, local `define`

For all the usual reasons, we need to be able to define local variables inside of functions. Like ML, there are expression forms that we can use anywhere to do this. Unlike ML, instead of one form of let-expression there are three. This variety is good: Different ones are convenient in different situations and using the most natural one communicates to anyone reading your code something useful about how the local bindings are related to each other. This variety will also help us learn about scope and environments rather than just accepting that there can only be one kind of let-expression with one semantics. How variables are looked up in an environment is a fundamental feature of a programming language.

First, there is the expression of the form

```
(let ([x1 e1]
      [x2 e2]
      ...
```

```
      [xn en])
  e)
```

As you might expect, this creates local variables `x1`, `x2`, ... `xn`, bound to the results of evaluating `e1`, `e2`, ..., `en`. and then the body `e` can use these variables (i.e., they are in the environment) and the result of `e` is the overall result. Syntactically, notice the "extra" parentheses around the collection of bindings and the common style of where we use square parentheses.

But the description above left one thing out: What environment do we use to evaluate `e1`, `e2`, ..., `en`? It turns out we use the environment from "*before*" the let-expression. That is, later variables do *not* have earlier ones in their environment. If `e3` uses `x1` or `x2`, that would either be an error or would mean some *outer* variable of the same name. This is *not* how ML let-expressions work. As a silly example, this function doubles its argument:

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

This behavior is sometimes useful. For example, to swap the meaning of `x` and `y` in some local scope you can write `(let ([x y][y x]) ...)`. More often, one uses `let` where this semantics versus "each binding has the previous ones in its environment" does not matter: it communicates that the expressions are independent of each other.

If we write `let*` in place of let, then the semantics *does* evaluate each binding's expression in the environment produced from the previous ones. This *is* how ML let-expressions work. It is often convenient: If we only had "regular" `let`, we would have to nest let-expressions inside each other so that each later binding was in the body of the outer let-expressions. (We would have use n nested `let` expressions each with 1 binding instead of 1 `let*` with n bindings.) Here is an example using `let*`:

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

As indicated above, it is common style to use `let` instead of `let*` when this difference in semantics is irrelevant.

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 (triple 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, which is very common in real Racket code. You can use it if you like but do not have to. There are restrictions on where local defines can appear; at the beginning of a function body is one common place where they are allowed.

```
(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))
```

We need to be careful with `letrec` and local definitions: They allow code to refer to variables that are initialized *later*, but the expressions for each binding are still evaluated in order.

For mutually recursive functions, this is never a problem: In the examples above, the definition of `even?` refers to the definition of of `odd?` even though the expression bound to `odd?` has not yet been evaluated. This is okay because the use in `even?` is in a function body, so it will not be *used* until after `odd?` has been initialized. In contrast, this use of `letrec` is bad:

```
(define (bad-letrec x)
  (letrec ([y z]
           [z 13])
    (if x y z)))
```

The semantics for `letrec` requires that the use of `z` for initializing `y` refers to the `z` in the `letrec`, but the expression for `z` (the `13`) has not been evaluated yet. In this situation, Racket will bind `y` to a special "undefined" object. It is bad style and surely a bug to use undefined objects in your program.

Top-Level Definitions

A Racket file is a module with a sequence of definitions. Just as with let-expressions, it matters greatly to the semantics what environment is used for what definitions. In ML, a file was was like an implicit `let*`. In Racket, it is basically like an implicit `letrec`. This is convenient because it lets your order your functions however you like in a module. For example, you do not need to place mutually recursive functions next to each other or use special syntax. On the other hand, the same "gotchas" we saw with `letrec` apply to the definitions in a module:

- You cannot have two bindings use the same variable. This makes no sense: which one would a use of the variable use? With `letrec`-like semantics, we do *not* have one variable shadow another one if they are defined in the same collection of mutually-recursive bindings.
- If an earlier binding uses a later one, it needs to do so in a function body so that the later binding is initialized by the time of the use. In Racket, the "bad" situation is handled slightly differently at the top-level of a module: instead of getting an "undefined" object, you get an error when you use the module (e.g., when you click "Run" for the file in DrRacket).