Coursera Programming Languages Course
Section 5 Summary

Standard Description: This summary covers **roughly** *the same material as the lecture videos and the materials (slides, code) posted with each video. It can help to read about the material in a narrative style and to have the material for an entire section of the course in a single document, especially when reviewing the material later. Please report errors in these notes on the discussion board.*

Contents

Switching from ML to Racket

For Part B of Programming Languages, we will use the Racket programming language (instead of ML) and the DrRacket 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.

The first few topics cover basic Racket programming since we need to introduce Racket before we start using it to study more advanced concepts. We will do this quickly because (a) we have already seen a similar language and (b) The Racket Guide, `http://docs.racket-lang.org/guide/index.html`, and other documentation at `http://racket-lang.org/` are excellent and free.

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 pairs built by cons 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.

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 x e)` where `x` is a variable and `e` is an expression, we evaluate `e` to a value and change the environment so that `x` 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)))
```

You can consult the Racket documentation to learn how to define your 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. It 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 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 xs)
  (cond [(null? xs) 0]
        [(number? xs) xs]
        [(list? xs) (+ (sum (car xs)) (sum (cdr xs)))]
        [#t 0]))
```

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

For all the usual reasons, we need to be able to define local variables inside functions. Like ML, there are expression forms that we can use anywhere to do this. Unlike ML, instead of one construct for local bindings, there are four. 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 x))]
           [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 and is in fact the preferred style over let-expressions. In this course, 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 `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 raise an error when `bad-letrec` is called. (Prior to Racket Version 6.1, it would instead bind `y` to a special "undefined" object, which almost always just had the effect of hiding a bug.)

For this class, you can decide whether to use local defines or not. The lecture materials generally will not, choosing instead whichever of `let`, `let*`, or `letrec` is most convenient and communicates best. But you are welcome to use local defines, with those "next to each other" behaving like `letrec` bindings.

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 like an implicit `let*`. In Racket, it is basically like an implicit `letrec`. This is convenient because it lets you 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, there are some new "gotchas" to be aware of:

- 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 of using an uninitialized value causes an error when you use the module (e.g., when you click "Run" for the file in DrRacket).
- So *within* a module/file, there is no top-level shadowing (you can still shadow within a definition or let-expressions), but one module can shadow a binding in another file, such as the files implicitly included from Racket's standard library. For example, although it would be bad style, we could shadow the built-in `list` function with our own. Our own function could even be recursive and call itself like any other recursive function. *However*, the behavior in the REPL is different, so do not shadow a function with your own recursive function definition in the REPL. Defining the recursive function in the Definitions Window and using it in the REPL still works as expected.

Bindings are Generally Mutable: `set!` Exists

While Racket encourages a functional-programming style with liberal use of closures and avoiding side effects, the truth is it has assignment statements. If `x` is in your environment, then `(set! x 13)` will *mutate* the

binding so that `x` now maps to the value 13. Doing so affects all code that has this `x` in its environment. Pronounced "set-bang," the exclamation point is a convention to alert readers of your code that side effects are occurring that may affect other code. Here is an example:

```
(define b 3)
(define f (lambda (x) (* 1 (+ x b))))
(define c (+ b 4))
(set! b 5)
(define z (f 4))
(define w c)
```

After evaluating this program, `z` is bound to 9 because the body of the function bound to `f` will, when evaluated, look up `b` and find `5`. However, `w` is bound to `7` because when we evaluated `(define c (+ b 4))`, we found `b` was `3` and, as usual, the result is to bind `c` to `7` regardless of how we got the `7`. So when we evaluate `(define w c)`, we get `7`; it is irrelevant that `b` has changed.

You can also use `set!` for local variables and the same sort of reasoning applies: you have to think about *when* you look up a variable to determine what value you get. But programmers used to languages with assignment statements are all too used to that.

Mutating top-level bindings is particularly worrisome because we may not know all the code that is using the definition. For example, our function `f` above uses `b` and could act strangely, even fail mysteriously, if `b` is mutated to hold an unexpected value. If `f` needed to defend against this possibility it would need to avoid using `b` after `b` might change. There is a general technique in software development you should know: *If something might get mutated and you need the old value, make a copy before the mutation can occur.* In Racket, we could code this up easily enough:

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

This code makes the `b` in the function body refer to a local `b` that is initialized to the global `b`.

But is this as defensive as we need to be? Since `*` and `+` are just variables bound to functions, we might want to defend against them being mutated later as well:

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

Matters would get worse if `f` used other helper functions: Making local copies of variables bound to the functions would not be enough unless those functions made copies of all their helper functions as well.

Fortunately, none of this is necessary in Racket due to a reasonable compromise: A top-level binding is not mutable unless the module that defined it contains a `set!` for it. So if the file containing `(define b 4)` did not have a `set!` that changed it, then we can rest assured that no other file will be allowed to use `set!` on that binding (it will cause an error). And all the predefined functions like `+` and `*` are in a module that does not use `set!` for them, so they also cannot be mutated. (In Scheme, all top-level bindings really are mutable, but programmers typically just assume they won't be mutated since it is too painful to assume otherwise.)

So the previous discussion is *not* something that will affect most of your Racket programming, but it is useful to understand what `set!` means and how to defend against mutation by making a copy. The point is that the possibility of mutation, which Racket often avoids, makes it very difficult to write correct code.

The Truth about `cons`

So far, we have used `cons`, `null`, `car`, `cdr`, and `null?` to create and access lists. For example, `(cons 14 (cons #t null))` makes the list `'(14 #t)` where the quote-character shows this is printing a list value, not indicating an (erroneous) function call to 14.

But the truth is *cons just makes a pair* where you get the first part with `car` and the second part with `cdr`. Such pairs are often called *cons cells* in languages like Racket. So we can write `(cons (+ 7 7) #t)` to produce the pair `'(14 . #t)` where the period shows that this is *not* a list. A list is, by convention and according to the `list?` predefined function, either `null` or a pair where the cdr (i.e., second component) is a list. A cons cell that is not a list is often called an *improper list*, especially if it has nested cons cells in the second position, e.g., `(cons 1 (cons 2 (cons 3 4)))` where the result prints as `'(1 2 3 . 4)`.

Most list functions like `length` will give a run-time error if passed an improper list. On the other hand, the built-in `pair?` primitive returns true for anything built with `cons`, i.e., any improper or proper list *except* the empty list.

What are improper lists good for? The real point is that pairs are a generally useful way to build an each-of type, i.e., something with multiple pieces. And in a dynamically typed language, all you need for lists are pairs and some way to recognize the end of the list, which by convention Racket uses the `null` constant (which prints as `'()`) for. As a matter of style, you should use proper lists and not improper lists for collections that could have any number of elements.

Cons cells are immutable, but there is `mcons`

Cons cells are immutable: When you create a cons cell, its two fields are initialized and will never change. (This is a major difference between Racket and Scheme.) Hence we can continue to enjoy the benefits of knowing that cons cells cannot be mutated by other code in our program. It has another somewhat subtle advantage: The Racket implementation can be clever enough to make `list?` a constant-time operation since it can store with every cons cell whether or not it is a proper list when the cons cell is created. This cannot work if cons cells are mutable because a mutation far down the list could turn it into an improper list.

It is a bit subtle to realize that cons cells really are immutable even though we have `set!`. Consider this code:

```
(define x (cons 14 null))
(define y x)
(set! x (cons 42 null))
(define fourteen (car y))
```

The `set!` of `x` changes the contents of the binding of `x` to be a different pair; it does not alter the contents of the old pair that `x` referred to. You might try to do something like `(set! (car x) 27)`, but this is a syntax error: `set!` requires a variable to assign to, not some other kind of location.

If we want mutable pairs, though, Racket is happy to oblige with a different set of primitives:

- `mcons` makes a mutable pair

- `mcar` returns the first component of a mutable pair
- `mcdr` returns the second component of a mutable pair
- `mpair?` returns `#t` if given a mutable pair
- `set-mcar!` takes a mutable pair and an expression and changes the first component to be the result of the expression
- `set-mcdr!` takes a mutable pair and an expression and changes the second component to be the result of the expression

Since some of the powerful idioms we will study next use mutation to store previously computed results, we will find mutable pairs useful.

Introduction to Delayed Evaluation and Thunks

A key semantic issue for a language construct is *when are its subexpressions evaluated*. For example, in Racket (and similarly in ML and most but not all programming languages), given `(e1 e2 ... en)` we evaluate the function arguments `e2`, ..., `en` once before we execute the function body and given a function `(lambda (...) ...)` we do not evaluate the body until the function is called. We can contrast this rule ("evaluate arguments in advance") with how `(if e1 e2 e3)` works: we do *not* evaluate both `e2` and `e3`. This is why:

```
(define (my-if-bad x y z) (if x y z))
```

is a function that *cannot* be used wherever you use an if-expression; the rules for evaluating subexpressions are fundamentally different. For example, this function would never terminate since every call makes a recursive call:

```
(define (factorial-wrong x)
  (my-if-bad (= x 0)
             1
             (* x (factorial-wrong (- x 1)))))
```

However, we can use the fact that function bodies are not evaluated until the function gets called to make a more useful version of an "if function":

```
(define (my-if x y z) (if x (y) (z)))
```

Now wherever we would write `(if e1 e2 e3)` we could instead write `(my-if e1 (lambda () e2) (lambda () e3))`. The body of `my-if` either calls the zero-argument function bound to `y` or the zero-argument function bound to `z`. So this function is correct (for non-negative arguments):

```
(define (factorial x)
  (my-if (= x 0)
         (lambda () 1)
         (lambda () (* x (factorial (- x 1))))))
```

Though there is certainly no reason to wrap Racket's "if" in this way, the general idiom of using a zero-argument function to *delay evaluation* (do not evaluate the expression now, do it later when/if the zero-argument function is called) is very powerful. As convenient terminology/jargon, when we use a zero-argument function to delay evaluation we call the function a *thunk*. You can even say, "thunk the argument" to mean "use `(lambda () e)` instead of `e`".

Using thunks is a powerful programming idiom. It is not specific to Racket — we could have studied such programming just as well in ML.

Lazy Evaluation with Delay and Force

Suppose we have a large computation that we know how to perform but we do not know if we need to perform it. Other parts of the program know where the result of the computation is needed and there may be 0, 1, or more different places. If we thunk, then we may repeat the large computation many times. But if we do not thunk, then we will perform the large computation even if we do not need to. To get the "best of both worlds," we can use a programming idiom known by a few different (and perhaps technically slightly different) names: lazy-evaluation, call-by-need, promises. The idea is to use mutation to remember the result from the first time we use the thunk so that we do not need to use the thunk again.

One simple implementation in Racket would be:

```
(define (my-delay f)
  (mcons #f f))

(define (my-force th)
  (if (mcar th)
      (mcdr th)
      (begin (set-mcar! th #t)
             (set-mcdr! th ((mcdr th)))
             (mcdr th))))
```

We can create a thunk `f` and pass it to `my-delay`. This returns a pair where the first field indicates we have not used the thunk yet. Then `my-force`, if it sees the thunk has not been used yet, uses it and then uses mutation to change the pair to hold the result of using the thunk. That way, any future calls to `my-force` with the same pair will not repeat the computation. Ironically, while we are using mutation in our *implementation*, this idiom is quite error-prone if the thunk passed to `my-delay` has side effects or relies on mutable data, since those effects will occur at most once and it may be difficult to determine when the first call to `my-force` will occur.

Consider this silly example where we want to multiply the result of two expressions `e1` and `e2` using a recursive algorithm (of course you would really just use `*` and this algorithm does not work if `e1` produces a negative number):

```
(define (my-mult x y)
 (cond [(= x 0) 0]
       [(= x 1) y]
       [#t (+ y (my-mult (- x 1) y))]))
```

Now calling `(my-mult e1 e2)` evaluates `e1` and `e2` once each and then does 0 or more additions. But what if `e1` evaluates to 0 and `e2` takes a long time to compute? Then evaluating `e2` was wasteful. So we could thunk it:

```
(define (my-mult x y-thunk)
 (cond [(= x 0) 0]
       [(= x 1) (y-thunk)]
       [#t (+ (y-thunk) (my-mult (- x 1) y-thunk))]))
```

Now we would call `(my-mult e1 (lambda () e2))`. This works great if `e1` evaluates to 0, fine if `e1` evaluates to 1, and terribly if `e1` evaluates to a large number. After all, now we evaluate `e2` on every recursive call. So let's use `my-delay` and `my-force` to get the best of both worlds:

```
(my-mult e1 (let ([x (my-delay (lambda () e2))]) (lambda () (my-force x))))
```

Notice we create the delayed computation once before calling `my-mult`, then the first time the thunk passed to `my-mult` is called, `my-force` will evaluate `e2` and remember the result for future calls to `my-force x`. An alternate approach that might look simpler is to rewrite `my-mult` to expect a result from `my-delay` rather than an arbitrary thunk:

```
(define (my-mult x y-promise)
 (cond [(= x 0) 0]
       [(= x 1) (my-force y-promise)]
       [#t (+ (my-force y-promise) (my-mult (- x 1) y-promise))]))

(my-mult e1 (my-delay (lambda () e2)))
```

Some languages, most notably Haskell, use this approach for all function calls, i.e., the semantics for function calls is different in these languages: If an argument is never used it is never evaluated, else it is evaluated only once. This is called *call-by-need* whereas all the languages we will use are *call-by-value* (arguments are fully evaluated before the call is made).

Streams

A stream is an infinite sequence of values. We obviously cannot create such a sequence explicitly (it would literally take forever), but we can create code that knows how to produce the infinite sequence and other code that knows how to ask for however much of the sequence it needs.

Streams are very common in computer science. You can view the sequence of bits produced by a synchronous circuit as a stream, one value for each clock cycle. The circuit does not know how long it should run, but it can produce new values forever. The UNIX pipe (`cmd1 | cmd2`) is a stream; it causes `cmd1` to produce only as much output as `cmd2` needs for input. Web programs that react to things users click on web pages can treat the user's activities as a stream — not knowing when the next will arrive or how many there are, but ready to respond appropriately. More generally, streams can be a convenient division of labor: one part of the software knows how to produce successive values in the infinite sequence but does not know how many will be needed and/or what to do with them. Another part can determine how many are needed but does not know how to generate them.

There are many ways to code up streams; we will take the simple approach of representing a stream as a thunk that when called produces a pair of (1) the first element in the sequence and (2) a thunk that represents the stream for the second-through-infinity elements. Defining such thunks typically uses recursion. Here are three examples:

```
(define ones (lambda () (cons 1 ones)))
```

```
(define nats
  (letrec ([f (lambda (x) (cons x (lambda () (f (+ x 1)))))])
    (lambda () (f 1))))
(define powers-of-two
  (letrec ([f (lambda (x) (cons x (lambda () (f (* x 2)))))])
    (lambda () (f 2))))
```

Given this encoding of streams and a stream `s`, we would get the first element via `(car (s))`, the second element via `(car ((cdr (s))))`, the third element via `(car ((cdr ((cdr (s))))))`, etc. Remember parentheses matter: `(e)` calls the thunk `e`.

We could write a higher-order function that takes a stream and a predicate-function and returns how many stream elements are produced before the predicate-function returns true:

```
(define (number-until stream tester)
  (letrec ([f (lambda (stream ans)
                (let ([pr (stream)])
                  (if (tester (car pr))
                      ans
                      (f (cdr pr) (+ ans 1)))))])
    (f stream 1)))
```

As an example, `(number-until powers-of-two (lambda (x) (= x 16)))` evaluates to 4.

As a side-note, all the streams above can produce their next element given at most their previous element. So we could use a higher-order function to abstract out the common aspects of these functions, which lets us put the stream-creation logic in one place and the details for the particular streams in another. This is just another example of using higher-order functions to reuse common functionality:

```
(define (stream-maker fn arg)
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (fn x arg)))))])
    (lambda () (f arg))))
(define ones (stream-maker (lambda (x y) 1) 1))
(define nats  (stream-maker + 1))
(define powers-of-two (stream-maker * 2))
```

Memoization

An idiom related to lazy evaluation that does not actually use thunks is *memoization*. If a function does not have side-effects, then if we call it multiple times with the same argument(s), we do not actually have to do the call more than once. Instead, we can look up what the answer was the first time we called the function with the argument(s).

Whether this is a good idea or not depends on trade-offs. Keeping old answers in a table takes space and table lookups do take some time, but compared to reperforming expensive computations, it can be a big win. Again, for this technique to even be *correct* requires that given the same arguments a function will always return the same result and have no side-effects. So being able to use this *memo table* (i.e., do memoization) is yet another advantage of avoiding mutation.

To implement memoization we do use mutation: Whenever the function is called with an argument we have not seen before, we compute the answer and then add the result to the table (via mutation).

As an example, let's consider 3 versions of a function that takes an `x` and returns fibonacci(x). (A Fibonacci number is a well-known definition that is useful in modeling populations and such.) A simple recursive definition is:

```
(define (fibonacci x)
  (if (or (= x 1) (= x 2))
      1
      (+ (fibonacci (- x 1))
         (fibonacci (- x 2)))))
```

Unfortunately, this function takes exponential time to run. We might start noticing a pause for `(fibonaccci 30)`, and `(fibonacci 40)` takes a thousand times longer than that, and `(fibonacci 10000)` would take more seconds than there are particles in the universe. Now, we could fix this by taking a "count up" approach that remembers previous answers:

```
(define (fibonacci x)
  (letrec ([f (lambda (acc1 acc2 y)
                (if (= y x)
                    (+ acc1 acc2)
                    (f (+ acc1 acc2) acc1 (+ y 1))))])
    (if (or (= x 1) (= x 2))
        1
        (f 1 1 3))))
```

This takes linear time, so `(fibonacci 10000)` returns almost immediately (and with a very large number), but it required a quite different approach to the problem. With memoization we can turn `fibonacci` into an efficient algorithm with a technique that works for lots of algorithms. It is closely related to "dynamic programming," which you often learn about in advanced algorithms courses. Here is the version that does this memoization (the `assoc` library function is described below):

```
(define fibonacci
  (letrec([memo null]
          [f (lambda (x)
               (let ([ans (assoc x memo)])
                 (if ans
                     (cdr ans)
                     (let ([new-ans (if (or (= x 1) (= x 2))
                                        1
                                        (+ (f (- x 1))
                                           (f (- x 2))))])
                       (begin
                         (set! memo (cons (cons x new-ans) memo))
                         new-ans)))))])
    f))
```

It is essential that different calls to `f` use the *same* mutable memo-table: if we create the table inside the call to `f`, then each call will use a new empty table, which is pointless. But we do not put the table at top-level just because that would be bad style since its existence should be known only to the implementation of `fibonacci`.

Why does this technique work to make `(fibonacci 10000)` complete quickly? Because when we evaluate `(f (- x 2))` on any recursive calls, the result is already in the table, so there is no longer an exponential

number of recursive calls. This is much more important than the fact that calling `(fibonacci 10000)` a second time will complete even more quickly since the answer will be in the memo-table.

For a large table, using a list and Racket's `assoc` function may be a poor choice, but it is fine for demonstrating the concept of memoization. `assoc` is just a library function in Racket that takes a value and a list of pairs and returns the first pair in the list where the car of the pair equal to the value. It returns `#f` if no pair has such a car. (The reason `assoc` returns the pair rather than the cdr of the pair is so you can distinguish the case where no pair matches from the case where the pair that matches has `#f` in its cdr. This is the sort of thing we would use an option for in ML.)

Macros: The Key Points

The last topic in this module is *macros*, which add to the *syntax* of a language by letting programmers define their own syntactic sugar. To preserve time for other topics, most of the macro material will be optional, but you are encouraged to work through it nonetheless. This section contains the key ideas that are *not* optional: Though you do not need macros for the homework assignment associated with this module, we need the *idea* for part of our study in the next module.

A *macro definition* introduces some new syntax into the language. It describes how to transform the new syntax into different syntax in the language itself. A *macro system* is a language (or part of a larger languages) for defining macros. A *macro use* is just using one of the macros previously defined. The semantics of a macro use is to replace the macro use with the appropriate syntax as defined by the macro definition. This process is often called *macro expansion* because it is common but not required that the syntactic transformation produces a larger amount of code.

The key point is that macro expansion happens *before* anything else we have learned about: before type-checking, before compiling, before evaluation. Think of "expanding all the macros" as a pre-pass over your entire program before anything else occurs. So macros get expanded everywhere, such as in function bodies, both branches of conditionals, etc.

Here are 3 examples of macros one might define in Racket:

- A macro so that programmers can write `(my-if e1 then e2 else e3)` where `my-if`, `then`, and `else` are keywords and this macro-expands to `(if e1 e2 e3)`.
- A macro so that programmers can write `(comment-out e1 e2)` and have it transform to `e2`, i.e., it is a convenient way to take an expression `e1` out of the program (replacing it with `e2`) without actually deleting anything.
- A macro so that programmers can write `(my-delay e)` and have it transform to `(mcons #f (lambda () e))`. This is different from the `my-delay` *function* we defined earlier because the function required the caller to pass in a thunk. Here the macro expansion does the thunk creation and the macro user should *not* include an explicit thunk.

Racket has an excellent and sophisticated macro system. For precise, technical reasons, its macro system is superior to many of the better known macro systems, notably the preprocessor in C or C++. So we can use Racket to learn some of the pitfalls of macros in general. The rest of this module (optional) will discuss:

- How macro systems must handle issues of tokenization, parenthesization, and scope — and how Racket handles parenthesization and scope better than C/C++
- How to define macros in Racket, such as the ones described above

- How macro definitions should be careful about the order expressions are evaluated and how many times they are evaluated
- The key issue of variable bindings in macros and the notion of *hygiene*

Optional: Tokenization, Parenthesization, and Scope

The definition of macros and macro expansion is more structured and subtle than "find-and-replace" like one might do in a text editor or with a script you write manually to perform some string substitution in your program. Macro expansion is different in roughly three ways.

First, consider a macro that, "replaces every use of `head` with `car`." In macro systems, that does *not* mean some variable `headt` would be rewritten as `cart`. So the implementation of macros has to at least understand how a programming language's text is broken into *tokens* (i.e., words). This notion of tokens is different in different languages. For example, `a-b` would be three tokens in most languages (a variable, a subtraction, and another variable), but is one token in Racket.

Second, we can 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 Racket, macro expansion preserves the code structure so this issue is not a problem. A Racket macro use always looks like `(x ...)` where `x` is the name of a macro and the result of the expansion "stays in the same parentheses" (e.g., `(my-if x then y else z)` might expand to `(if x y z)`). This is an advantage of Racket's minimal and consistent syntax.

Third, we can ask if macro expansion happens even when creating variable bindings. 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
(let* ([hd 0] [car 1]) hd) ; evaluates to 0
```

If we replace `car` with `hd`, then the first expression is an error (trying to bind `hd` twice) and the second expression now evaluates to 1. In Racket, 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.

Optional: Defining Macros with `define-syntax`

Let's now walk through the syntax we will use to define macros in Racket. (There have been many variations in Racket's predecessor Scheme over the years; this is one modern approach we will use.) 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 `my-if` to the environment so that expressions of the form `(my-if ...)` will be macro-expanded 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` in the body of `my-if` is just syntax whereas anything not in this list (and not `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 is 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 results. 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`.

Here is a second simple example where we use a macro to "comment out" an expression. We use `(comment-out e1 e2)` to be rewritten as `e2`, meaning `e1` will never be evaluated. This might be more convenient when debugging code than actually using comments.

```
(define-syntax comment-out
  (syntax-rules ()
    [(comment-out e1 e2) e2]))
```

Our third example is a macro `my-delay` so that, unlike the `my-delay` function defined earlier, users would write `(my-delay e)` to create a promise such that `my-force` would evaluate `e` and remember the result, rather than users writing `(my-delay (lambda () e))`. Only a macro, not a function, can "delay evaluation by adding a thunk" like this because function calls always evaluate their arguments.

```
(define-syntax my-delay
  (syntax-rules ()
    [(my-delay e)
     (mcons #f (lambda () e))]))
```

We should *not* create a macro version of `my-force` because our function version from earlier is just what we want. Give `(my-force e)` we *do* want to evaluate `e` to a value, which should be an mcons-cell created by `my-delay` and then perform the computation in the `my-force` function. Defining a macro provides no benefit and can be error prone. Consider this awful attempt:

```
(define-syntax my-force
  (syntax-rules ()
     [(my-force e)
      (if (mcar e)
```

```
          (mcdr e)
          (begin (set-mcar! e #t)
                 (set-mcdr! e ((mcdr e)))
                 (mcdr e)))]))
```

Due to macro expansion, uses of this macro will end up evaluating their argument *multiple times*, which can have strange behavior if `e` has side effects. Macro users will not expect this. 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 (begin (print "hi") (my-delay some-complicated-expression)))
```

we end up printing multiple times. Remember that macro expansion copies the entire argument `e` everywhere it appears in the macro definition, but we often want it to be evaluated only once. This version of the macro does the right thing in this regard:

```
(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))))]))
```

But, again, there is *no reason* to define a macro like this since a function does exactly what we need. Just stick with:

```
(define (my-force th)
  (if (mcar th)
      (mcdr th)
      (begin (set-mcar! th #t)
             (set-mcdr! th ((mcdr th)))
             (mcdr th))))
```

Optional: Variables, Macros, and Hygiene

Let's consider a macro that doubles its argument. Note 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 equivalent to each other. But this short example will let us investigate when macro arguments are evaluated and in what environment, so we will use it just as a poor-style example.

These two macros 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 should use a local variable:

```
(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 Racket, 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 Racket, 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 Racket 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 Racket 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. Racket's macro system has a way to do this, which involves explicitly violating hygiene, but we will not demonstrate it here.

Optional: More macro examples

Finally, let's consider a few more useful macro definitions, including ones 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 Racket's `let*` entirely in terms of `let`. We need some way to talk about "the rest of a list of syntax" and Racket'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.

Finally, here is a macro for a limited form of for-loop that executes its body $hi - lo$ times. (It is limited because the body is not given the current iteration number.) Notice the use of a let expression to ensure we evaluate `lo` and `hi` exactly once but we evaluate `body` the correct number of times.

```
(define-syntax for
  (syntax-rules (to do)
    [(for lo to hi do body)
     (let ([l lo]
           [h hi])
       (letrec ([loop (lambda (it)
                        (if (> it h)
                            #t
                            (begin body (loop (+ it 1)))))])
         (loop l)))]))
```