Racket

Notes written by Ben Wood, Fall 2015

Contents

First Case Study: Racket and LISP

The Racket language (in the context of its ancestors LISP and Scheme) is the first programming language we will study in this course.

This document introduces core pieces of the Racket language as in lecture. For additional reference, see the Racket Guide. As we introduce features, we will sometimes not tell the "whole story" at first or we will start with simpler, more restrictive definitions.

Racket falls into a broad category of programming languages often called *functional programming languages*. *Functional* refers to a focus on functions in the mathematical sense. Unfortunately, the term can be a bit confusing, since many other languages (such as the widely-known C language) use constructs called *functions*, even though they do not follow a "functional" style. Functional languages are built around the *evaluation of expressions* and the application of functions for their results, rather than the execution of sequences of commands for their effects on stored data. In some sense, functional languages are a little closer to expressing

what to compute, while imperative (command-, statement-, and assignment-focused) languages are closer to expressing *how* to compute it. *Functional* languages are closer to math; *imperative* languages are closer to the model presented by most modern computer hardware systems.

Our goals in studying Racket are several, including:

- Exposure to ***functional programming***, which emphasizes the evaluation of expressions and functions on immutable data rather than the execution of commands for their effect on mutable storage. This experience (probably new to many of you) not only introduces you to a powerful, expressive, and elegant approach to programming (there are those words again), but it also helps you learn to think about problems in different ways.
- Exposure to a ***syntax*** that is likely very unfamiliar but also elegantly minimalist and predictable.
- Informal study of the ***semantics*** of a functional programming language.
- Consideration of how a ***motivating problem*** can shape the design of a programming language built to help solve that problem.
- Consideration of how the available computing ***hardware*** can influence the design of even a "high-level" programming language.
- Consideration of how the design of a programming language may introduce interesting problems in that language's ***implementation***.

Many of these goals should be familiar from our initial attempts to define the term *programming language* and our discussions of their purpose and design.

You may be surprised that "learn the Racket language" is not listed as an explicit goal. The discussion above should make clear that we are aiming for skills that are more transferable than intimate knowledge of a single programming language. (Though admittedly, learning a few new languages is a nice side effect of this course!)

We will revisit the "why" question later in the course when we have more shared perspective. But let's get to this promised study of a programming language already!

Racket and LISP, really

Our case study comes in two parts.

1. We will study the basics of the Racket language, some idioms of functional programming, and how well these things fit together.

   To facilitate our study, we will examine some programs that are so small they seem silly. Keep in mind that we can write similarly small and silly programs in any language. It is useful to explore some fundamental language concepts in the small before progressing to bigger programs. Pay attention to the words we use to describe the simple code we examine to start. Start with a blank slate – do not try to relate these ideas back to what you are familiar with from Java or other languages. We will get there, but doing so now can cause unnecessary confusion.

2. Along the way, we will consider how the motivating problems and available technology in place at the design and implementation of the original LISP language (Racket's origin ancestor, with significant resemblance) affected the language that resulted, and how this language introduced interesting new problems and ideas in programming language implementation.

Aside: Tools vs. Languages

While we will do all of our Racket programming in the DrRacket environment, it is important to remember that the Racket Language is distinct from the tools we use to write or run Racket programs. (The same holds for any well-defined programming language and the related tools and implementations.) In fact, later this semester, we will build our own implementation of a small programming language that looks very much like the core of Racket.

Racket is part of a family of closely related languages supported by DrRacket. To indicate which language we are using, write:

```
#lang racket
```

More notes on DrRacket are here.

Aside: As we explore the language…

While we are building our understanding of the Racket language, we will sometimes start with narrower definitions of language constructs than are true in reality. As we continue, we will see (and take) some (but not all) opportunities to generalize our definitions. Since our aim in studying Racket is a quick, but careful, introduction, we will not explore the whole Racket language, we will also stop short of "the whole story" for some language features. Our methods will also prime us for more careful study of the Standard ML language, where we will get closer to "the whole story," starting in a couple weeks.

Expressions, Values, Bindings, and Environments

A Racket program consists of *definitions* and *expressions*, *evaluated* in order. The result of evaluating a definition or an expression depends on a *dynamic environment*, which is roughly the values of the definitions earlier in the file. Before considering definitions, we will consider expressions.

Expressions and Values

An ***expression*** `e` may or may not have a ***value*** `v`. An expression is evaluated to a value according to a set of ***evaluation rules***. Each kind of expression has its own evaluation rule.

Values

The simplest kind of expression is a *value*. Consider numbers. The number 251 is written in Racket as the series of digits `251`. The evaluation rule for values is that a value evaluates to itself. For example, the expression `251` evaluates to the number value `251`, because it already is that value. All values are expressions, but not all expressions are values. In other words, values are expressions that cannot be evaluated any further.

Trivial as number values may seem, the way we defined them is important. We specified two things about number values:

1. **syntax:** how to write a number value
2. **semantics:** how to evaluate a number value (*i.e.*, evaluation rules)

As we build up the core of the Racket language we will continue in this manner, rapidly adding more constructs to the language by describing their syntax and semantics (evaluation rules).

Other Values

Besides number values, Racket has several other types of values, including boolean values, with syntax `#t` (true) and `#f` (false). We will consider other kinds of values later. The syntax for different types of values differs, but their evaluation rules remain the same: a value evaluates to itself.

Arithmetic Expressions

As an example of a (slightly) more interesting expression, let us consider addition. The **syntax** of a Racket addition expression is:

```
(+ e1 e2)
```

Here, `e1` and `e2` are expressions. For example, we could write `(+ 251 251)` and we could also write `(+ (+ 240 251) (+ 235 301))` or even `(+ #t 251)`. The parentheses are mandatory. Racket uses *prefix* notation, where operators in expressions like addition are written before their operands rather than between. The latter approach, familiar from many other

programming languages, is called *infix* notation.

The **evaluation rule** for an addition expression is:

1. Evaluate expression `e1` to value `v1` and then evaluate expression `e2` to value `v2` in the same dynamic environment; then
2. If both `v1` and `v2` are numbers, then the result is a number value that is the arithmetic sum of `v1` and `v2`, otherwise a type error occurs.

There are two interesting considerations arising from the syntax and evaluation rules for Racket addition:

1. **Dynamic Type-Checking:** Every value has a type. For example, `251` is a *number* value, while `#t` is a *boolean* value. The evaluation rule for Racket `+` makes it clear that Racket does *dynamic type-checking*, meaning it checks specific types of values when evaluation reaches an operation that actually requires a particular type of value. For example, Racket `+` requires that its operands be numbers. This is in contrast to compile-time *static type-checking* that you know from the Java language or others. We will return to discuss dynamic vs. static type-checking later.

2. **Recursive Structure and Evaluation:** Expressions that are not values contain other expressions, meaning the structure of an expression is recursive in nature. This is visible in the syntax. Since the structure of an expression may be arbitrarily *deep*, its evaluation must proceed recursively. To evaluate an addition expression (which is clearly not a value), we must first evaluate its subexpressions. The requirement for the subexpressions to evaluated first is called *eager evaluation* or *call-by-value*. Later in the course, we will discuss the implications of this choice and explore alternative orders of evaluation, but the recursive structure will remain.

3. **Environments:** Evaluation involves a *dynamic environment*, that, so far, has no influence on the result of evaluation. We will begin using environments shortly.

Other Arithmetic

Racket has nearly identical syntax and evaluation rules for other arithmetic operations, including `-` (difference), `*` (product), `/` (division, with fraction results), `quotient` (truncated integer division as familiar from other programming languages). For non-commutative arithmetic operations, the left-to-right order of operands is the same in prefix notation as in postfix notation. For example, `(- 251 240)` in Racket is equivalent to *(251 - 240)* in arithmetic. While `+`, `-`, and `*` always succeed, `/` and `quotient` cause errors if attempting to divide by `0`.

Number Comparison Expressions

So far, we have boolean values, but no non-value expressions that result in boolean values. The *less than* comparison expression results in a boolean value. Its **syntax** is:

```
(< e1 e2)
```

Here, `e1` and `e2` are expressions. The **evaluation rule** for *less than* is:

1. Evaluate `e1` to a value `v1` then evaluate `e2` to a value `e2` under the current dynamic environment.
2. If `v1` and `v2` are both number values, then:
   - the result is the boolean value `#t` if the number `v1` is less than the number `v2`;
   - otherwise the result is the boolean value `#f`

   Otherwise (if one or both values are not numbers), a type error occurs.

Other number comparison expressions with analogous syntax and evaluation rules are: `>`, `<=`, `>=`, and `=`.

Conditional Expressions

Recall that, as a functional language, Racket is focused on the evaluation of expressions. There are no *statements* or *commands*, so the *if statement* that is familiar from languages like Java or C does not make sense. Instead Racket has an ***if expression***. Its **syntax** is:

```
(if e1 e2 e3)
```

Here, `e1`, `e2`, and `e3` are all expressions.

The **evaluation rule** is:

1. Evaluate `e1` to value `v1` under the current environment.
2. If `v1` is any value *except* `#f`, the result of evaluating the entire *if* expression is the result of evaluating `e2` under the current environment.
   Otherwise, the result of evaluating the entire *if* expression is the result of evaluating `e3` under the current environment.

As an *expression* the `if` expression can appear anywhere any expression can appear. For example, we might write:

```
(if (< 9 (- 251 240))
    (+ 4 (* 3 2))
    (+ 4 (* 3 3)))
```

Or equivalently, with fewer symbols:

```
(+ 4 (* 3 (if (< 9 (- 251 240)) 2 3))))
```

Note that, unlike the Racket addition expression, the Racket *if* expression does *not* evaluate all of its subexpressions! So far, it might seem that this is not an important distinction, since all we are doing is arithmetic, but consider the following two expressions:

```
(if #t 251 (/ 251 0))

(if #f (+ #t 251) 251)
```

Both expressions contain subexpressions that, if *evaluated*, cause errors: division by zero in the first case and attempting to add a non-number value in the second. In both cases, our intuition for what the value of this expression should be (hopefully) matches the evaluation rule. Even though one subexpression of the `if` contains an expression whose value is undefined, would cause an error when evaluated, it does not matter, because the condition expression selects the other branch of the `if`. As we generalize some ideas later, we will see that this makes `if` "special" as compared to `+`, for example.

Applying Evaluation Rules: A Glorified Calculator?

So far, the Racket expressions we have seen are enough to use as a simple calculator. Evaluating expressions by hand (trivial as they may seem so far) is good practice to become familiar with the evaluation rules or to help debug programs. Some simple notation makes it easy to write down facts *about* expressions and their evaluation. (Read that sentence again – this notation is ***not*** Racket syntax. It is notation for us to use while talking ***about*** Racket programs.)

`e` ↓ `v` is an ***evaluation assertion***. It asserts that expression `e` evaluates to value `v`, over potentially many applications of the evaluation rules.

For example, it should be obvious that `(+ 3 (+ 5 4))` ↓ `12`. We justify this with an ***evaluation derivation*** in which we apply all of the evaluation rules necessary to prove the assertion.

- `(+ 3 (+ 5 4))` ↓ `12`, by the addition evaluation rule, because:
  - `3` ↓ `3`, by the value evaluation rule
  - and `(+ 5 4)` ↓ `9`, by the addition evaluation rule, because:
    - `5` ↓ `5`, by the value evaluation rule

        - and `4` ↓ `4`, by the value evaluation rule
        - and `5` and `4` are both number values
        - and adding `5` and `4` results in `9`
    - and `3` and `9` are both number values
    - and adding `3` and `9` results in `12`

The recursive structure of expressions and the recursive nature of their evaluation becomes apparent in this derivation.

There are two ways to view this derivation:

1. It corresponds to a sequence of recursive calls an evaluator program (a.k.a. *interpreter*) might make to evaluate this expression on a computer.
2. It corresponds to a proof that this expression has this value.

Bindings, Variables, and Environments

Our evaluation rules for arithmetic expressions have referenced "dynamic environments" even though these environments have not been used during evaluation so far. Environments will allows to store *bindings* of *variables* to *values*, which we must do to write any interesting programs.

Bindings

A *definition* is a kind of *binding*. Definitions and bindings in general appear in a few different shapes. For now, we will consider definitions with the following **syntax**:

```
(define x e)
```

Here, `define` is a keyword, `x` can be any *identifier* (a.k.a. *variable*), and `e` can be any *expression*.

The **evaluation rule** for a binding is:

1. Use the "current dynamic environment" (the values of previous bindings) to evaluate `e` to a value `v`; then
2. Produce a "new dynamic environment" that is the same as the current environment, but with `x` bound to the value `v`.

Remember to resist the temptation to associate words like *variable* with meanings from other programming languages you know. Here, we will treat a *variable* as in math – a name for an unknown value. Once we attach the name to a particular value, we never change it.[1] (If you must use an analogy, choose the notion of a *constant* from other languages.)

Definitions are *not* expressions. A Racket program is a sequence of definitions and expressions.

Variables

Variables are expressions. Their **syntax** is `x`, where `x` is a valid identifier. (See the Racket Guide for the long list of characters allowed as part of identifiers. The **evaluation rule** for a variable is:

- Look up the variable identifier in the current dynamic environment and use the associated value. If the variable identifier has no entry in the environment, an error occurs.

Finally we use the environment. We will use it in more interesting ways as we add functions, to come…

Examples

Lecture examples are in define.rkt.

Functions

With all the commotion about Racket being a *functional programming language*, we should probably look at functions. The big reveal here is that ***functions are values***. It will take us a while to appreciate all the implications of this fact, but for now, we can start small.

Function Definitions

A *function definition* is an *expression* with the following **syntax**:

```
(lambda (x1 ... xn) e)
```

Here, `lambda` is a keyword indicating a function definition, `x1` through `xn` are zero or more identifiers (a.k.a. variable names, also called *parameters* here in a function definition), and `e` is any expression.

The **evaluation rule** for a function definition is simple, since *a function is a value*. A function definition simply creates a new function value – there is no further evaluation. Specifically, we do ***not*** evaluate the expression `e` now.

Function Application

Function definitions only become useful if we can apply the functions defined. *Function application* or *function call* has the following **syntax**:

```
(e0 e1 ... en)
```

Here, `e0` is a required expression and `e1` through `en` are zero or more expressions. Note that, syntactically, function application looks similar to many other expressions (and one non-expression) seen so far. `e0` is referred to as the *function expression* and `e1` through `en` are the *argument expressions*

The **evaluation rule** for function application is:

1. Evaluate `e0` through `en` to values `v0` through `vn`, in order, using the current dynamic environment.
2. If `v0` is a function `(lambda (x1 ... xn) e)`, that is, if `v0` is a function that expects exactly the same number of *arguments* as are provided in the function call, then perform the next step, otherwise there is an error.
3. Evaluate `e` in an environment extended with `x1` → `v1`, `x2` → `v2`, etc., up through `xn` → `vn`. The resulting value, `v`, is the value of the function application expression.

Which environment should we extend with the arguments in step 3? We always extend the environment that *was current when the function was defined*, **not** the current environment at the function call. For now, we will carefully avoid the implications of this distinction, but we will study it in great detail later.

Thus, we can write a simple function application for a function that squares its argument:

```
((lambda (x) (* x x)) (- 12 8))
```

This should evaluate to `16`, starting in the empty environment. (Warning: this derivation works for this example, but our characterization of function values is not the whole story. Stay tuned for more.)

`((lambda (x) (* x x)) (- 12 8))` ↓ `16`, by the function application evaluation rule, because:

- `(lambda (x) (* x x))` ↓ `(lambda (x) (* x x))`, by the value evaluation rule;
- and `(- 12 8 )` ↓ `4`, by the subtraction evaluation rule, because:
  - `12` ↓ `12`, by the value evaluation rule;
  - and `8` ↓ `8`, by the value evaluation rule;

    - and `12` and `8` are both number values;
    - and *12 - 8* is 4.
- and `(lambda (x) (* x x))` is a function that takes one argument and one argument is given;
- and `(* x x)` evaluates to `16` in the environment where `x` → `4`, by the multiplication evaluation rule because:
    - `x` evaluates to `4` in the environment where `x` → `4`, by the variable evaluation rule;
    - and `x` evaluates to `4` in the environment where `x` → `4`, by the variable evaluation rule;
    - and `4` and `4` are both number values;
    - and *4 * 4* is 16.

Evaluating a function definition in the DrRacket REPL shows the resulting function *value*, although the way it displays the value is only mildly informative:

```
> (lambda (x) (* x x))
#<procedure>
```

DrRacket has printed an *opaque* representation of the function value created by the definition, showing that it is *some* procedure (referring to function). A function is a value, so why does it not show us `(lambda (x) (* x x))` as the resulting value? The reason has to do with the expression for a called function being evaluated in the environment *in which the function was defined*. In fact, representing the function value in our derivation above as `(lambda (x) (* x x))` is not the whole story. We will discuss this in detail later.

Function Bindings and Recursion

Notice that our evaluation rule for function *definition* does *not* add any bindings to the current dynamic environment, nor does it require any further evaluation. In other words, we have defined a function, but we have not given it a name. It is *anonymous*. Creating a function with a name is not necessary to use it, but is nonetheless easy to do. In fact, you may notice that we currently have no way to define a recursive function. The *fix* is simple.

Just as with other values, functions may be used in `define` bindings. Recall the syntax and evaluation rules for variable bindings. Evaluation of `(define x e)` first evaluates expression `e` to value `v`, then creates a new dynamic environment by adding the mapping `x` → `v` to the current dynamic environment. Since a function definition is an expression, it may be used in a binding. For example, here is the square function, now named:

```
(define square
    (lambda (x)
        (* x x)))
```

Now we can easily reuse this function to call `(square 4)` or `(square 251)`. Now, consider the following function, which defines an intended recursive function that computes the result of raising a given base to a given non-negative exponent, and binds the name `pow` to it.

```
(define pow
    (lambda (base exp)
        (if (< exp 1)
            1
            (* base (pow base (- exp 1))))))
```

The `define` binding actually does a little more than we described earlier. It adds the binding for `x` to the environment for following bindings and expressions, but it *also adds the binding for* `x` *to the environment used by the function itself*. This is crucial to enable recursion. Recall that a function call evaluates the function body in the environment where the function was defined, extended with bindings of its parameters to the actual function argument values. If the function's own name is not in this environment, the function cannot call itself. We will return to define this a bit more precisely when we discuss a larger related issue later. For now, `define` introduces bindings that support recursion.

Syntactic Sugar

In fact, introducing a `define` binding for a function definition is common enough that it has its own special syntax. The following code is semantically equivalent to the previous code:

```
(define (pow base exp)
    (if (< exp 1)
        1
        (* base (pow base (- exp 1)))))
```

Special (typically, shorter) syntax for something that is already expressible in the language is called ***syntactic sugar***.

Functions and Special Forms

Having defined functions, let us try to generalize some of our earlier definitions. We initially defined new syntax and evaluation rules for each arithmetic operation. However, examining the syntax, expressions like `(+ 1 2)` are syntactically similar to function calls. Their evaluation rules even look suspiciously similar to the evaluation rules for function application. In fact, we may think of `+`, `-`, `*`, `/`, `quotient`, `<`, and friends as bindings to functions that are predefined in the language. Treating them as bindings to built-in functions, we no longer need specific evaluation rules for these constructs: they are simply functions applied to arguments.

By adding functions, we might argue that our language definition has actually gotten smaller! How far can we go? Can we consider `define` to be the name of a special function? Unfortunately, no, since `(define x e)` is not an expression, it cannot be a function application. `define` and a handful of other keywords are so-called *special forms* in Racket.

What about `(if e1 e2 e3)`? Can we consider `if` to be a binding for a specially-defined function that returns its second argument if its first argument is not `#f` and returns its third argument otherwise?

No. `(if e1 e2 e3)` is an expression, but consider its evaluation rules as compared to those of function application. While function application evaluates *all* of its subexpressions before applying the function, `if` evaluates only two out of three of its subexpressions. In fact, the *short-circuit boolean operations* `and` and `or` (like `&&` and `||` in many other languages) are also not functions, for the same reason: they do not always evaluate all of their subexpressions. However, we can define short-circuit `and` and `or` using syntactic sugar:

```
(and e1 e2)
```

is equivalent to

```
(if e1 e2 #f)
```

and

```
(or e1 e2)
```

is (almost) equivalent to

```
(if e1 #t e2)
```

Consider a few examples and try to describe exactly the conditions that lead to `e1` or `e2` being evaluated in each of these cases.

Of the constructs we have defined so far, `if`, `and`, `or`, `define`, and `lambda` are ***not*** simply bindings for built-in functions. On the other hand, `not` can be defined as a function, since it has one argument that is always evaluated:

```
(define (not x)
    (if x #f #t))
```

Cons Cells and Lists

Racket is descended from Scheme, which is descended from LISP, the first functional programming language, invented in the 1950s at MIT. LISP stood for LISt Processing. Support for lists is central to the LISP language and its descendants.

Thus far, the *values* we have seen are ***atoms***, values that are indivisible and are not composed of smaller values. The other major category of values is built using the *cons cell*.[2]

Cons Cells

A ***cons cell*** is a value that is a pair of other values. The first value in the pair is called the ***car*** and the second value is called the ***cdr*** (pronounced *could-er*). "Cons" stands for "*cons*truct," which makes sense in context of functions used to manipulate cells. The other two names are historical accidents that refer to hardware features of the first computer on which the LISP language was implemented, but were too ingrained in LISP culture to change by the time they were reconsidered. (A mnemonic to remember their order is that *car* comes first in alphabetical order, so it accesses the first of the pair.)

A new cons cell may be created with the built-in `cons` functions, which takes any two arguments (in order) to build a pair. There is no requirement what type of values are pair in the cell. Given a pair as its single argument, the built-in `car` function returns the first value in the pair. Given a pair as its single argument, the `cdr` function returns the second value in the pair.

Like any value, a cons cell, once constructed, evaluates to itself – there is no further computation required. Furthermore, a cons cell may be the result of a function application, which makes it simple to return a pair of values from a function, something that is more cumbersome in a language like Java.

Evaluating `(car (cdr (cons 1 (cons 2 3))))` yields 2. Step through the evaluation to convince yourself. Recall that `car`, `cdr`, and `cons` are bindings of built-in functions, so this expression consists of several nested function applications.

The following function takes a pair of values and results in a new pair in which their order is reversed. It does ***not*** change the `pair` originally passed to the function.

```
(define (swap-pair pair)
    (cons ((cdr pair) (car pair))))
```

Here is a silly function to "sort" a pair of values given in a cons cell, reusing the swap-pair function:

```
(define (sort-pair pair)
    (if (< (car pair) (cdr pair))
        pair
        (swap pair)))
```

Dot Notation

DrRacket displays cons cells as follows:

```
> (cons 1 2)
'(1 . 2)
```

We will discuss the significance of this notation later. Do ***not*** try to create cons cells with the dot notation in your Racket programs. Use `cons`.

Lists

Although a cons cell may hold any pair of values, it is most commonly used to represent lists in a *singly-linked list* form. A ***list*** is a recursive data structure that is either:

- `null`, an atom used to represent the empty list, with zero elements; or
- a cons cell whose *car* is the first element in the list and whose *cdr* is a list that is the rest of the elements in the list.

For example, the following expression builds the list "1, 2, 3":

```
(cons 1 (cons 2 (cons 3 null)))
```

Lists are so central to Racket that there is another shorter expression that results in the same list structure:

```
(list 1 2 3)
```

This is actually a function that takes a variable number of arguments and returns a chain of cons cells representing the desired list. (Surprise, the arithmetic functions accept variable numbers of arguments too!)

Quoted List Notation

DrRacket displays linked cons cells that form a valid list as follows:

```
> (cons 1 (cons 2 (cons 3)))
'(1 2 3)
> (list 4 5 6)
'(4 5 6)
```

We will discuss the significance of this notation later. Do ***not*** try to create lists by writing the quoted notation in your Racket programs. Use `cons` or `list`.

List Functions

Given the representation of lists as chains of cons cells, applying the `car` function to a list returns the ***head*** of that list, *i.e.*, its first element. The `cdr` function returns the ***tail*** (or *rest*) of the given list, *i.e.*, a list containing all but the first element of the given list. Note that the term *tail* is used differently when describing lists in a functional language than when describing the tail of a linked list data structure in imperative languages, where it typically refers to the *last element* instead of the list of all elements except the first.

Many functions over Racket lists are naturally recursive. To distinguish between the *base case* (an empty list represented by `null`) and the *recursive case* (a non-empty list represented by a cons cell), two built-in functions are useful:

- `null?` returns `#t` if its argument is `null` and `#f` if its argument is not `null`.
- `pair?` returns `#t` if its argument is a cons cell and `#f` if its argument is not a cons cell.

Our first recursive list function is `sum`, which takes a list and returns the sum of its elements, assuming they are all numbers:

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

(Note, `xs` is pronounced *exes*, as the plural of `x`. This is a naming idiom that will make more sense as we go along.) What does this function do?

```
(define (countdown n)
    (if (= n 0)
        null
        (cons n (countdown (- n 1)))))
```

Append two lists to create one long list:

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

Any list function that might ever care about more than the first element of the list is recursive. Writing recursive list functions can be beautifully simple when compared to write loops and worrying about index variables and assignment statements. Recursive list functions must simply define:

- the answer for the ***base case***, if the list in question is the empty list; and
- the answer for the ***recursive case***, if the list in question is a non-empty list, in terms of the answer for the rest of the list.

Consider the `list-append` function, which employs recursion over its first argument, `xs`. If `xs` is empty, then appending a list `ys` to the end of the empty list is the same as `ys` itself. Otherwise, we want to append `ys` after the list of all `xs` elements except the first (with a recursive call to `list-append`), which is almost all of the appended list. Finally, we need to `cons` the first element of `xs` onto the front of the resulting appended list. (Yes, `cons` is also a verb, but `car` and `cdr` are nouns.) This works by requesting append operations with shorter and shorter versions of `xs`, until reaching the empty list, at which point it begins to return, `cons`ing the elements of `xs` onto the beginning of `ys` one at a time, from last to first.

Let Expressions

Earlier, we described `define` bindings to bind variables to values. Unfortunately, as they are not expressions, `define` bindings are poorly suited for use within function definitions (or any expressions for that matter), where your programming experience so far has hopefully shown *local variables* to be invaluable.

Let expressions introduce bindings into a local scope and are themselves expressions. They are not just a nicety; they also support efficient code that avoids recomputing results.

The ***syntax*** of a let expression is:

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

First, note the use of square brackets. Racket allows `[` `]` anywhere it allows `(` `)`, as long as they are matched (*i.e.*, `(+ 2 3]` is not legal). When nesting many S-expressions (the name for the parenthesis-bounded syntactic units), judicious use of the different bracket types and line breaks can be useful to help with readability. In fact, you may notice that this document often splits `if` expressions into three lines for clarity. A similar convention is often applied to `let` expressions:

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

The amount or type of whitespace does not matter. Here, `x1` through `xn` are one or more variable names and `e1` through `en` are expressions.

The **evaluation rule** for let expressions is:

1. Evaluate `e1` through `en` to `v1` through `vn`, in order, in the current dynamic environment.
2. Create a new dynamic environment by extending the current dynamic environment with bindings `x1` → `xn`, in order.
3. Evaluate `e` to a value `v` in the new dynamic environment. `v` is the result of the entire `let` expression.

Scope

Every binding has a ***scope***, the expressions in which the variable it binds can be used. The scope of a `define` binding is the entire remainder of the enclosing scope (typically the whole file) following the binding. The scope of a `let` binding is the *body* (the main subexpression, `e` in the syntax above) of the `let` expression. The following silly example shows two uses of the `x` variable. The first (as the *body* of the `let` expression) is within the scope of the binding shown here. The second is outside that binding's scope.

```
(+  (let ([x 4]) x)  x)  ; error: second use of x is outside scope of let
```

The binding introduced within the `let` expression is visible only within its body. In other words, the binding is discarded as evaluation leaves the associated `let` expression.

We discuss the scope of bindings introduced in `let` expressions below. Later, we will explore scope in more depth along with functions.

Scope Within Bindings

Notice that all of the expressions `e1` through `en` are evaluated in the environment that is in place when entering the `let` expression. This means that none of the bindings introduced by the `let` are visible when evaluating later bindings in the same `let`. For example, in the following code, the expression `(+ x x)` does not refer to the binding of `x` introduced in this `let` expression, but `(* x y)` does. Comments to the right show the environment after evaluating the line. `.` indicates the initial environment.

```
                    ; env: .
(let ([x 4]         ; env: .
      [y (+ x x)])  ; error, x not bound
  (* x y))
```

Assuming there is not already a binding for `x` in the environment in which the entire `let` expression is evaluated, evaluating this expression would result in an error, since `x` is not bound in the environment in which `(+ x x)` is evaluated.

The likely desired behavior here can be implemented by nesting `let` expressions. Comments to the right show the environment after evaluating the line:

```
                        ; env: .
(let ([x 4])            ; env: x --> 4, .
    (let ([y (+ x x)])  ; env: y --> 8, x --> 4, .
        (* x y)))       ; env: .
```

Consider the evaluation rules carefully to convince yourself that this expression evaluates to `32`. The bindings introduced within the `let` expressions are only visible within their bodies. The bindings are discarded as evaluation leaves their associated `let` expressions.

Another form, `let*`, uses the same **syntax** (except with `let` replaced by `let*`). Its evaluation evaluates each binding in turn, starting with `e1` to `v1` in the original environment, then evaluating `e2` in the current environment extended with `x1` → `v1`, and so on, where each expression is evaluated in the new environment created by the previous binding. The following `let*` expression evaluates to `32`. The comment to the right of each line shows the environment *after* evaluating that line.

```
                    ; env: .
```

```
(let* ([x 4]          ; env: x --> 4, .
       [y (+ x x)])   ; env: y --> 8, x --> 4, .
   (* x y))           ; env .
```

The bindings introduced within the `let` expression are only visible within later bindings in the `let*` expression and in the main expression. They are discarded as evaluation leaves the scope of the `let*`.

Shadowing

When looking up variables in environments, the matching binding most recently added to the current environment is used. This means that if the same variable is bound twice, the "closer enclosing binding wins." This is quite simple to visualize in code in two different ways. Consider the following code, where expressions in each line are evaluated in the environment listed on the line above and the resulting environment is listed to the right of the line:

```
                        ; env: .
(let ([x 2])            ; env: x --> 2, .
    (let ([x (* x x)])  ; env: x --> 4, x --> 2, .
        (+ x 3)))       ; env: .
```

This expression evaluates to `7`. Each use of the variable `x` refers to the closest enclosing binding of `x`. DrRacket includes a nice visual representation of this if you hover over variables. Additionally, bindings are added to the environment (as notated here) on the left, which represents the closest enclosing scope. Lookup also happens from the left, and the first matching binding is taken. Thus when evaluating `x` in `(+ x 3)` above, looking up `x` in the current environment returns `4`, as added by the closest enclosing `let` binding. Here, we say the binding `x` → `4` ***shadows*** the binding `x` → `2`.

Understanding shadowing helps cement your understanding of the environment, but using shadowing in code can be confusing and should generally be avoided for style reasons. In fact, Racket prohibits shadowing of `define` bindings by other `define` bindings in the top level of a Racket file (for reasons we will not consider), but allows any shadowing in the REPL and allows `let` bindings to shadow anywhere.

Local Function Bindings

Reasonable helper functions are good style. In Java, we might define helper methods and make them `private`, since they are an implementation detail, but this still leaves the helper methods visible to other methods in the same class, which is not always desired. Racket lets us go even further. Since functions definitions are expressions, we can bind a name to a function in a local `let` binding, as seen in this silly example which is not particularly good style:

```
(define (quad x)
    (let ([square (lambda (x) (* x x))])
        (square (square x))))
```

However, unlike like `define`, `let` does not support recursion. To define a local recursive function, we need a new form, `letrec`, which makes its bindings visible for recursive function definitions, in the spirit of `define`, but in the form of an expression.

For example, here is our first attempt at a function that returns a list with *x* elements, the integers 1 through *x*:

```
(define (count-up-from-1 x)
    (letrec ([count (lambda (from to)
                      (if (= from to)
                          (cons to null)
                          (cons from (count (+ from 1) to))))])
       (count 1 x)))
```

This works and it hides its helper function from all others in good style, but the helper function itself is not quite in good functional style yet. Recall that a function call evaluates the function body in the environment where the function was defined, extended with bindings of its parameters to the actual function argument values. Our call to `(count 1 x)` then binds the variable `to` to the value of `x`. In the definition of `count`, even in recursive calls, we always pass the same value for `to`. Yet `x` is bound to this same value in the environment in which `count`'s body is evaluated. A better version omits `to` entirely:

```
(define (count-up-from-1-better x)
    (letrec ([count (lambda (from)
                      (if (= from x)
                          (cons x null)
                          (cons from (count (+ from 1))))))])
       (count 1)))
```

There is another way to introduce a local recursive function binding using `define`, but we will omit it to avoid dealing with the fact that a `define` binding is not an expression.

Let for Efficiency

One important use of `let` expressions is to avoid reevaluating an expensive expression because its value is needed more than once.

Consider this naive implementation of a function that returns the maximum value in a list of numbers:

```
(define (bad-max xs)
    (if (null? xs)
        null ; max is not defined on empty list
        (if (null? (cdr xs))
            (car xs)
            (if (> (car xs) (bad-max (cdr xs)))
                (car xs)
                (bad-max (cdr xs))))))
```

First, note that *maximum* is not defined on the empty set (or empty list). To represent this, `bad-max` returns `null` for this case instead of a number. This is not the best design choice, but we will leave it alone and consider a worse design choice.

How many (recursive) calls to `bad-max` occur in the application `(bad-max (list 1))`? `(bad-max (list 1 2 3 4))`? For the list holding `1` through `30`, in order? Things rapidly get out of hand, passing 1 billion recursive calls for the 30-element ordered list. This is *exponential blowup*. The `bad-max` function has $O(2^n)$ running time for an n-element list. Each recursive level makes 2 calls, each of which makes 2 calls, each of which… (To keep things tricky, though, a list of 30 numbers in descending order takes only 30 recursive calls.)

The max-finding algorithm need not be so expensive, and `let` expressions help avoid this level of inefficiency:

```
(define (good-max xs)
    (if (null? xs)
        null
        (if (null? (cdr xs))
            (car xs)
            (let ([rest-max (good-max (cdr xs))])
                (if (> (car xs) rest-max)
                    (car xs)
                    rest-max)))))
```

This implementation has running time that is $\Theta(n)$. Each recursive call makes 0 or 1 recursive calls.

Let is Sugar

`let` expressions are not special – they can be implemented using functions. A `let` expression of the form:

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

is semantically equivalent to the following anonymous function application:

```
((lambda (x1 ... xn) e)
    e1 ... en)
```

Consider the evaluation rules of both constructs to see how this is true.

Properly Desugaring `(or e1 e2)`

Above, we showed how the expression `(or e1 e2)` could be desugared as `(if e1 #t e2)`. This is not *quite* accurate. Unlike short-circuit `&&` and `||` in languages you like know already, Racket's `and` and `or` do not guarantee to return exactly `#t` or `#f`. Specifically, while `#f` is the only "false" value, any non-`#f` is "true". Thus an expression like `(or e1 e2)` actuall returns `#f` if both `e1` and `e2` evaluate to `#f`. It returns the result of evaluating `e1` if this result is non-`#f`, otherwise it returns the result of evaluating `e2`. While the `(and e1 e2)` desugaring given earlier still holds `(if e1 e2 #f)`, the `(or e1 e2)` desugaring is more accurately:

```
(let ([v1 e1])
(if v1 v1 e2))
```

where `v1` is some variable name not used without first being bound within `e1` or `e2`.

First-Class Functions and Closures

A language with ***first-class functions*** provides all the privileges of a value to a function: a first-class function *is* a value. Examples of these privileges are: passing a function as an argument to another function, returning a function as the result of a function, storing a function in a data structure, and several more.

A **higher-order function*** is a function that takes other functions as arguments or returns other functions as results.

A ***function closure*** is a function that refers to bindings created outside the function that were present in the environment where the function was defined.

These three ideas are powerful (and often confused with each other). In Racket, we have already seen that functions are values, but we will now consider many more interesting implications of this fact.

The ill-defined concept of a ***functional language*** or programming style is often attached to these features, but also more generally to many others, like (or at least a minimum) of mutation, a focus on evaluation of expressions instead of execution of commands, a syntax that looks closer to math than to C. Yet it is possible to program in a functional style, even in an imperative, curly-brace, statement language like C or Java. In fact, after experience programming in the "functional languages" we consider in this course, you may find that your Java or C programming begins to use more immutable data, fewer loops, or trying to emulate some of the benefits of first-class functions where they are not present by cobbling together other language features.

Functions as Arguments

Higher-order functions that take other functions as arguments are classic tools in the functional programming toolbox. They support the goal of abstracting repeated patterns in a way that is arguably more flexible than allowed by many languages without higher-order functions.

Map

A classic function that takes functions as arguments is `map`. The `map` function takes a function, `f`, and a list, `xs`, and returns a new list where each element is the result of applying `f` to the corresponding element of `xs`:

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

For example, given the increment function:

```
(define (inc x) (+ x 1))
```

the following function application:

```
(map inc (list 1 2 3 4 5))
```

results in the list containing incremented versions of the original values: `'(2 3 4 5 6)`.

The `map` function is so heavily use that it is available in the default environment in Racket.

Filter

Another classic higher-order function is `filter`, which takes a function `f` and a list `xs` and returns a list whose elements are all elements `x` of `xs` for which `(f x)` evaluates to a non-`#f` value.

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

Given the function `even?` which returns `#t` for even numbers and `#f` for others:

```
(define (even? x) (= 0 (modulo x 2)))
```

the following function application:

```
(filter even? (list 1 2 3 4 5))
```

evaluates to the list containing just `2` and `4`: `'(2 4)`.

The `filter` function is so heavily use that it is available in the default environment in Racket.

Anonymous Functions as Arguments

Anonymous functions (`lambda`) really shine with higher-order functions like `map` and `filter`. For example, let us use a one-off function written just for this particular task: find the multiples of 23 in a list of numbers:

```
(filter (lambda (x) (= 0 (modulo x 23)))
    (list 46 87 49 23 14 92 99))
```

Lexical Scope

As promised earlier when defining evaluation rules for functions, we have taken care to keep things simple when using functions as values. The functions we have passed as arguments thus far have all been ***closed***, meaning their bodies refer only to their own parameters or locally defined variables and not to any other variables bound outside the function definition. However, our evaluation rule for function calls says that a function's body is evaluated in an environment including bindings of its parameters to its arguments *and* any bindings present in the environment where the function was defined. Exploiting this last part holds great potential, but only if we get the semantics right!

This idea is so important, we will "learn this" at least twice.

*When a function is called, its body is evaluated in the environment where the function was **defined**, not the environment where the function is called.*

This example demonstrates the difference:

```
(define x 1)
(define f (lambda (y) (+ x y)))
(define z
    (let ([x 2]
          [y 3])
      (f (+ x y)))))
```

The name `f` is bound to a function with parameter `y`. Its body also looks up `x` in the environment where `f` was defined. *Regardless of where it is called*, this function *always* increments its argument, because it *always* refers to the binding of `x` that was in scope when it was defined: `1`. Later, we have a different environment where `f` maps to this function, `x` maps to `2`, `y` maps to `3`, and we evaluate the function call `(f x)`. Let us apply the function call evaluation rule to see the result:

1. Evaluate the function expression `f`. `f` is a variable, so we lookup its value in the environment, which yields the function we described.

2. Evaluate the argument expression `(+ x y)`. This looks up `x` and `y` in the current environment. They are mapped to `2` and `3`, respectively, so the result is `5`.

3. Call the function with the argument `5`, which evaluates the body, `(+ x y)` in the environment that was in effect when the function was defined (`x` → `1`) extended by mapping the parameter variable `y` to the argument value `5`. Evaluating the addition requires looking up `x` and `y` in this environment, yielding `1` and `5`, so the sum—and thus the result of the function call—is `6`.

The argument was evaluated in the *current* environment, but the function body was evaluated in the "old" environment. This rule is called ***lexical scope***.

We will consider the alternative—and generally more problematic—*dynamic scope* later. (It would yield `7` above.) The original LISP language of the 1950s employed *dynamic scope*, which was later "fixed" in Scheme (Racket's parent) and other languages.

Lexical Scope via Environments and Closures

We have been a little loose with our claim that *functions are values*. A ***function value***, known as a ***function closure***, or just *closure* for short, has to parts:

- The *function definition* (its parameter names and its body expression); and

- The *environment* where the function definition was evaluated.

To be clear, this pair is not a cons cell that can be manipulated within the language. **Closures are created only by the evaluation of function definitions and their parts are used only by function applications.**

The term *closure* arises from the fact that the act of attaching an environment to a function definition *closes* an otherwise *open* function—a function whose body may contain *free variables*, variables that are not local variable bindings or parameters of the function. **Creating a closure *closes* up the function with everything it needs for later application**. A closure is *closed* in that **no binding in the environment of a function call can affect the evaluation of the function on a given argument.** The closure is a black box whose only input is the function argument.

Here are two tricky examples to illustrate the details of lexical scope and closures.

Example 1

```
(define x 1)
(define f
    (lambda (y)
        (let ([x (+ y 1)])
            (lambda (z) (+ x y z)))))
(define z
    (let ([x 3]
          [g (f 4)]
          [y 5])
      (g 6)))
```

This example binds `f` to a closure that maps `x` to `1`. When we later evaluate `(f 4)`, we evaluate the function's body `(let ([x (+ y 1)]) (lambda (z) (+ x y z)))` in an environment where `x` maps to `1`, extended to map `y` to `4`. But then due to the `let` binding of `x`, we shadow `x` and evaluate `(lambda (z) (+ x y z))` in an environment where `x` maps to `5` and `y` maps to `4`. The `(lambda (z) (+ x y z))` expression itself evaluates to a closure with the environment we just described. So `(f 4)` returns a closure that, when called, will always add `5` and `4` to its argument, no matter the environment where it is called. Finally, in the last line, `(g 6)` evaluates to `15`, even though the current environment maps `x` to `3` and `y` to `5`. So `z` is bound to `15`.

Example 2

```
(define f
    (lambda (g)
        (let ([x 3])
            (g 2))))
(define x 4)
(define h
    (lambda (y)
        (+ x y)))
(define z (f h))
```

In this example, `f` is bound to a closure that takes another function as an argument and returns the result of applying that function to the value `2`. The closures bound to `h` *always* adds `4` to its argument because the argument is `y`, the body is `(+ x y)`, and the function is defined in an environment where `x` maps to `4`. In the last line, `z` will be bound to `6`. The `let` binding of `x` to `3` is irrelevant: the call `(g 2)` is evaluated by looking up `g` to get the closure that was passed in and then using that closure with its *environment* (in which `x` maps to `4`) with `2` for an argument.

Why Lexical Scope

While it takes practice to unlock the power of lexical scope and higher-order functions, decades of experience make clear that this semantics is what we want. Much of the rest of this section will describe various widespread, powerful idioms that rely on lexical scope.

But first we can also motivate lexical scope by showing how dynamic scope (where you just have one current environment and use it to evaluate function bodies) leads to some fundamental problems.

First, suppose in Example 1 above the body of `f` was changed to `(let ([q (+ y 1)]) (lambda (z) (+ q y z)))`. Under lexical scope this is fine: we can always change the name of a local variable and its uses without it affecting anything. Under dynamic scope, now the call to `(g 6)` will make no sense: we will try to look up `q`, but there is no `q` in the environment at the function call site.

Similar issues arise with Example 2: The body of `f` in this example is awful: we have a local binding we never use. Under lexical scope we can remove it, changing the body to `(g 2)`, and know that this has no effect on the rest of the program. Under dynamic scope it would have an effect.
Also, under lexical scope we know that any use of the closure bound to `h` will add `4` to its argument regardless of how other functions like `g` are implemented and what variable names they use. This is a key *separation of concerns* that only lexical scope provides.

For "regular" variables in programs, lexical scope is the way to go. There are some compelling uses for dynamic scoping for certain idioms, but few languages have special support for these (Racket does, but we will not explore it in depth) and very few if any modern languages have dynamic scoping as the default. Interestingly, LISP (origin ancestor of Racket) "got this wrong" and used dynamic scope.

More to come…

Programs as data, quoting, etc.

Additional Reference

Racket Guide: http://docs.racket-lang.org/guide/index.html

The Racket Guide is an excellent, readable reference for the full Racket language. We study only the core of the language for our purposes in 251. There are many other interesting language features, some of which we may return to consider later, including: contracts, a dynamic checking system that can check properties beyond dynamic type-checking; hygienic macros, a system for principled definitions of syntactic sugar by end-programmers; and first-class continuations, a construct for capturing, saving, and manipulating the "rest of the computation to be done" that differs from traditional control flow.

Acknowledgments

These notes draw heavily (through reuse and adaptation) on notes on the ML and Racket languages by Dan Grossman at the University of Washington, by permission. Any mistakes, typos, or lack of clarity are my own.

Footnotes:

1. I have chosen words carefully here. By "we never change it", I mean: Racket will allow us to mutate bindings via something like assignment, but we will avoid this feature entirely in our use of the language. In general, mutation of bindings is available, but used rarely in Racket programs. As we leave Racket behind for our next case study language, ML, bindings really will be immutable. ↩

2. Racket includes other compound values besides cons cells, but we focus on cons cells alone, which were invented for LISP. ↩