CSCI 275: Programming Abstractions

Lecture 02: Procedures & Choice

Fall 2024

Stephen Checkoway, Oberlin College
Slides gratefully borrowed from Molly Q Feldman

Announcements

Ice cream social at 4:30 p.m. in the King/Rice courtyard TODAY

Functional Language of the Week: LISP

Really!

John McCarthy invented LISP at MIT around 1960 as a language for AI.

LISP grew quickly in both popularity and power. As the language grew more powerful it required more and more of a system's resources. By 1980, 5 simultaneous LISP users would bring a moderately powerful PDP-11 to its knees.

Guy Steele developed Scheme at MIT 1975-1980 as a minimalist alternative to LISP.

Scheme is an elegant, efficient subset of LISP. It has some nice properties that we will look at that allow it to be implemented efficiently.

We do this on Wednesdays!

Goals for Today

- Basics of Racket
- How do we make choice (i.e., conditionals, etc.)?
- How do we construct and use procedures?

When we talk about code/Racket in this class, I will do my best to use `Font in This Text` to differentiate what is description and what is code

Introducing Racket

Why Racket for CS 275?

All LISP-type languages have lists as the main data structure

- Programs are lists
- Data are lists
- Racket programs can reason about other programs. This makes Racket useful for thinking about programming languages in general.

Racket is a different programming paradigm

- Python, Java, C and other languages are imperative languages. Programs in these languages do their work by changing data stored in variables
- Racket programs can be written as functional programs—they compute by evaluating functions and avoid variable assignments.

Why Racket for CS 275?

Racket is very elegant. It is much less verbose than Java, for instance, which means it is easier to see what is happening in a Racket program.

I think its fun.

It lets you learn functional programming without a lot of extra features.

Racket Basics

We are used to **basic values** in most languages.

- Numbers (Integers & Floats)
- Strings
- Booleans

We are also accustomed to **procedures/functions** which act on elements of these types

These also can look different depending on the language!
‘banana’ is invalid Java, but valid Python

Arithmetic/logical/string operations

3 + 5

$x \cdot (4 + y + z)$

x AND y

x OR y OR z

"hello" + " " + "world" (

Language Design Statement: you know the *semantics* of these terms, even if this *syntax* is not that of a language you've learned before

Everything is prefix in Racket

Language Design Statement:

The order that a language has the operators and operands is *arbitrary*.

In Racket, you put the operator or function call *first* (prefix form)

`(< x 2)` instead of `x < 2`

Equivalent operations in Racket

3 + 5	`(+ 3 5)`
$x \cdot (4 + y + z)$	`(* x (+ 4 y z))`
x AND y	`(and x y)`
x OR y OR z	`(or x y z)`
"hello" + " " + "world"	`(string-append "hello" " " "world")`

Language Design Statement: you know the *semantics* of these terms, even if this *syntax* is not that of a language you've learned before

In most languages, we would compute the arithmetic mean (average) of two numbers (or variables holding numbers) as `(x + y) / 2`. How do we do this in Racket?

A. `(x + y) / 2`

B. `((x + y) / 2)`

C. `(+ x y / 2)`

D. `(+ (/ x y) 2)`

E. `(/ (+ x y) 2)`

What do you think these examples will evaluate to?

```
(+ 5 2)

(zero? x)

(or (and #t #f) (and #t #t))

(+ (- 1 0) (- 2 3))
```

What do you think these examples will evaluate to?

`(+ 5 2)`	7
`(zero? x)`	Depends on `x`
`(or (and #t #f) (and #t #t))`	`#t`
`(+ (- 1 0) (- 2 3))`	0

Procedures in Racket

All the examples we saw on the previous example - e.g. `(zero? X)` and `(+ (- 1 0) (- 2 3))` – are calls to **procedures**.

In general, the structure of a procedure call in Racket is:
`(name-of-procedure arg1 arg2 … argn)`

The parentheses here are the *call* to `name-of-procedure`
The arguments are given in a row, separated with spaces, after the procedure name

Procedures are a special case of Racket Forms

When presented with a sequence `(foo arg1 arg2 …)` Racket looks at the first element of the sequence (here, `foo`)

If `foo` is a special form, Racket follows special instructions (`define`, `and`, etc.)

If `foo` is a procedure (built-in or made by you), it applies that procedure to the arguments and returns the result

This is the most common error in the first couple weeks of class!

Otherwise, error!

`(1 2 3)` is an error because `1` is not special form or procedure

Special Form: `define`

Giving names to expressions – useful!
However, these are not variables.

`(define id s-exp)`

The `define` special form binds an identifier to a value

This modifies the *environment*, the mapping of identifiers to values

```
(define hi “Hello”)
(define professors '("Adam" "Steve" "Cynthia"))
(third professors) => "Cynthia"
```

```
(define x (+ 20 100))
```

Whatever is in `s-exp` evaluates, so `x` is bound to 120

Predicates

Style: predicates in Racket will always have `?` as the last character (they are asking a question!)

Racket has a bunch of procedures that return `#t` if its argument satisfies some property

`(zero? x)` returns `#t` if `x` is equal to 0

`(empty? x)` returns `#t` if `x` is the empty list

`(positive? X)` return `#t` if `x` is a positive number

`(number? x)` returns `#t` if `x` is a number

Tests for equality

Most of the time: Use `equal?`

`(equal? a b)` compares structures recursively

Are you dealing with numbers? Use `=`

`(= a b)` compares only numbers, cannot be used for anything else

`eq?/eqv?` are about referring to the same object in memory; sometimes useful when you care about literal equality

If expression

`(if test-exp then-exp else-exp)`

If `test-exp` evaluates to *anything other than* `#f`, then the whole expression evaluates to the evaluation of `then-exp`

If `test-exp` evaluates to `#f`, then the whole if expression evaluates to the evaluation of `else-exp`

```
(if (= x y)
    (+ x 2)
    y)
(if (empty? lst)
    "The list is empty"
    "The list is not empty")
```

Conditional expressions

```
(cond [test-exp1 exp1] … [test-expn expn])
```

Evaluates the `test-exp` expressions in turn

The first one that evaluates to something other than `#f` has its corresponding `exp` evaluated - this becomes the value of the whole expression

We can (and should!) use `else` as the last test expression

```
(cond [(zero? x) 0]
      [(> x 0) 1]
      [else -1])
```

If your program is more than just a *very simple* if statement, use `cond`. It's good style.

```
(define foo 12)
(cond [(< foo 2) #t]
      [(>= foo 10) #f]
      [(not (zero? foo)) #t]
      [else (error “there is a problem!”)])
```

What does this code evaluate to?

A. #t
B. #f
C. #t or #f, depends on the run
D. Error
E. Something else

Some questions

```
(define foo 12)
(cond [(< foo 2) #t]
      [(>= foo 10) #f]
      [(not (zero? foo)) #t]
      [else (error “there is a problem!”)
```

1. How can I get the `cond` to take an argument, rather than just reference a “global” `foo`?

1. How do I “save” code like that above to be able to reuse it? (i.e. a function!)

- How is/isn’t this related to using `define` to bind identifiers?

Creating procedures: `lambda`

Procedures are creating using the `lambda` special form

```
(lambda parameters body …)
```

`parameters` is an unevaluated list of identifiers which will be bound to the values of the procedure's arguments when procedure is called

`body` is a sequence of s-expressions that form the body of the procedure, they're evaluated in turn

```
(lambda (x y)
  (/ (+ x y) 2))
(lambda (name)
  (displayln "Hello ")
  (displayln name))
```

Naming `lambdas`

Given we have a lambda, we can use it and call it

```
((lambda (x) (+ x 2)) 4)
```

This will evaluate to 6. However, this current structure doesn't allow us to *reuse* the lambda with a different input.

We already have a way to bind a value to an identifier ("name"): that's `define`.

We know `define` attaches a name to an evaluated value

`(define x (+ 20 100))` means `x` is bound to 120

So what does a `lambda` evaluate to? Anything?

BIG IMPORTANT SLIDE

Unlike procedures in most languages, in Racket there is a notion that `lambdas` are values & so can be evaluated

- `lambdas` are like numbers, strings, lists, etc.
- We can pass them around, return them, hold them as their own, evaluated concept
  - This is **really not true** in languages like C, for instance
  - This makes procedures **first-class** in Racket
- Support for higher-order/first-class functions is one of the hallmarks of a language that supports **functional programming**

Closures: what lambdas evaluate to

The expression of `(lambda parameters body…)` evaluates to a *closure* consisting of

- The parameter list (a list of identifiers)
- The body as un-evaluated expressions (often just one expression)
- The environment (the mapping of identifiers to values) **at the time the lambda expression is evaluated**

We'll return to this – becomes important!

`define` + `lambda` = reusable procedures!

We can combine `define` and `lambda`, so that we can get a named procedure!

```
(define add-two
   (lambda (x)
      (+ x 2)))
```

To call it, we then use prefix call notation, as usual:

`(add-two 2)` will give us 4

What have we learned thus far?

- How to call procedures
- Predicates
- `if`
- `cond`
- `define`
- `lambda`
- `define & lambda` together!

```
(define lily
  (lambda (x y)
      (string-append y x)))

(lily "hello" "?")
```

What does this code evaluate to?

A. Error
B. "hello?"
C. "?hello"
D. "hello ?"
E. Something else

```
(define alright
  (lambda (a b)
    (cond  [(equal? a b) “equal”]
           [(positive? a) 17]
           [(and (positive? a) (negative? b)) 5]
           [else “chaos!”]))))
```

What does calling `(alright 10 -30)` evaluate to?

A. ”chaos”
B. Error
C. 5
D. 17
E. ”equal”

Can we use identifiers in lambdas? Sure!

Note: you won't see for loops very often in this class – recursion all the way

Computing factorial in Racket:

```
(define fact
   (lambda (num)
      (if (<= num 1)
            1
            (* num (fact (- num 1))))))
```

A Note on Readings

RPTFW is really a reference guide

- If something didn’t make sense in lecture? Great resource, this textbook or the additional resources I link
  - Honor Code: look it up there, not Google!
- If you want more detail about something? Readings!
- Especially Chapters 1 & 2 teach you about some great Racket operators (hint: member, remove) that we don’t cover in class
- You’ll read about mutability (e.g., set!), for loops and some “useful” Racket that is **not** functional style - refrain from using it if possible, stick to what we learn in class!
- Readings/order of lecture not entirely in sync

Next Up!

See the Schedule for Suggested Readings.

Homework 0 is live

- **If you’ve never used Git/Github locally, please start ASAP**
- Due Friday, September 13 at 23:59

Post on Ed with questions