CS 162 Programming languages

Lecture 11: A Crash Course in Racket

Yu Feng

Winter 2020

Racket & Rosette

- Next three units will use the **Racket** language and the Solver-aided language **Rosette**
  - Installation/basic usage instructions on course website
- Like ML, functional focus with imperative features: Anonymous functions, closures, pattern-matching, etc
- No static type system: accepts more programs, but most errors do not occur until run-time
- Advanced features like macros, modules, quoting/eval, continuations, contracts, ... (Will do only macros)

File structure

- Start every file with a line containing only
  `#lang racket`

- A comment starts with semicolon “;”

- A file is a module containing a *collection of definitions* (bindings)

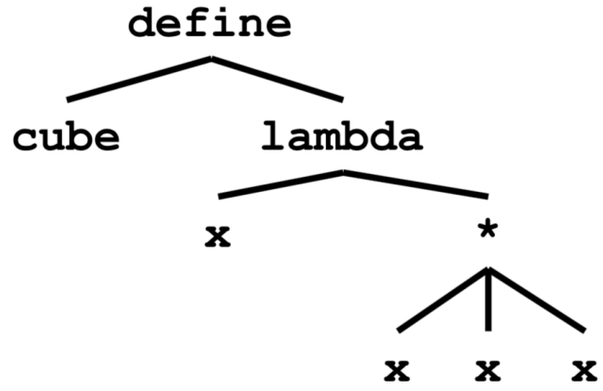
Example

```
#lang racket

(define x 3)
(define y (+ x 2))

(define cube ; function
  (lambda (x)
    (* x (* x x))))

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

Why is this good?

- By parenthesizing everything, converting the program text into a tree representing the program (*parsing*) is trivial and unambiguous
  - Atoms are leaves
  - Sequences are nodes with elements as children
  - (No other rules)

```
(define cube
  (lambda (x)
    (* x x x)))
```



- Also makes indentation easy
- No need to discuss “operator precedence” (e.g., **x + y * z**)

Our old friend: currying/β-reduction

```
#lang racket

(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))
```

Another old friend: list

- Empty list: **null**
- Cons constructor: **cons**
- Access head of list: **car**
- Access tail of list: **cdr**
- Append two lists: **append**
- Check for empty: **null?**
- Notes
    - Empty list is represented as **'() or (list) or null**
    - **(list e1 ... en)** for building lists

Another old friend: list

```
#lang racket

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

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

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

Local bindings

- Racket has 4 ways to define local variables
  - **`let`**
  - **`let*`**
  - **`letrec`**
  - **`define`**

- Variety is good: They have different semantics
- Use the one most convenient for your needs, which helps communicate your intent to people reading your code: If any will work, use **`let`**
- Will help us better learn scope and environments

Let

- A let expression can bind any number of local variables
- The expressions are all evaluated in the environment from **before the let-expression**
  - Except the body can use all the local variables of course
- This is **not** how ML let-expressions work
- Convenient for things like **`(let ([x y] [y x]) ...)`**

```
#lang racket

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

Let*

- Syntactically, a let* expression is a let-expression with 1 more character
- The expressions are evaluated in the environment produced from the **previous bindings**
  - Can repeat bindings (later ones shadow)
- This **is** how ML let-expressions work

```
#lang racket

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

Parentheses matter

- You must break yourself of one habit for Racket:
  - Do not add/remove parens because you feel like it
    - Parens are never optional or meaningless!!!
- (e) means call e with zero arguments
- So ((e)) means call e with zero arguments and call the result with zero arguments