PAUL G. ALLEN SCHOOL OF COMPUTER SCIENCE & ENGINEERING

CSE 341
Section 6

Racket Basics, Lists, and Delayed Evaluation

Learning Objectives

- Become familiar with the Racket IDE and REPL
- Review the basics, comparing with ML: variables, functions, conditions, functions
- Build and process lists in Racket using functions we've already seen in ML
- Know how (and when) to use delayed evaluation with thunks

Racket

Next two units will use the Racket language (not ML) and the DrRacket programming environment (not Emacs)

- Installation / basic usage instructions on course website

- Like ML, functional focus with imperative features
  - Anonymous functions, closures, no return statement, etc.
  - No pattern-matching
- No static type system
  - Accepts more programs, but most errors do not occur until run-time
- Really minimalist syntax
- Advanced features like macros, modules, quoting/eval, continuations, contracts, …
  - We'll do only a couple of these

The Racket Guide/Reference

- Racket has amazingly good documentation; use it!
- The Racket Guide introduces and explains features of the language in detail
- The Racket Reference defines the core language and common libraries; good way to look up a particular function. (Right-clicking on a function name in DrRacket will give you a link to the relevant doc page.)

DrRacket Tips

- Hitting tab will add the appropriate amount of whitespace to the beginning of the line your cursor is on. You can also reindent all with cmd-i (find the command under the Racket tab).
- Mousing over a variable shows an arrow to where it's defined
- Putting #; in front of a block enclosed in parentheses will comment the whole block out. You can also comment multiple lines with a command under the Racket tab
- At the top of the window, clicking where it says "(define ...)" will give a list of the variables all your definitions are bound to.
- In the interaction window, alt-p will repeat entries from your history, like the up arrow at the command line. (Alt is bound to Esc for OSX)
- Instead of `lambda`, you can use cmd-\ to use a λ character

SML *vs.* *Racket*

```
val x = 3
val y = x + 2

fun cube x = x * x * x;



fun pow (x, y) =
  if y = 0
  then 1
  else x * pow (x, y - 1)
```

```
#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))))))
```

Examples

```
(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)))))
```

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!!!
- In most places **`(e)`** means call **`e`** with zero arguments
- So **`((e))`** means call **`e`** with zero arguments and call the result with zero arguments

Without static typing, often get hard-to-diagnose run-time errors

Review: What are the errors?

Correct:

```
(define (fact n)(if (= n 0) 1 (* n (fact (- n 1)))))
```

Treats **`1`** as a zero-argument function (run-time error):

```
(define (fact n)(if (= n 0) (1)(* n (fact (- n 1)))))
```

Gives **`if`** 5 arguments (syntax error)

```
(define (fact n)(if = n 0 1 (* n (fact (- n 1)))))
```

3 arguments to define (including **`(n)`**) (syntax error)

```
(define fact (n)(if (= n 0) 1 (* n (fact (- n 1)))))
```

Treats **`n`** as a function, passing it **`*`** (run-time error)

```
(define (fact n)(if (= n 0) 1 (n * (fact (- n 1)))))
```

Scope

Consider the following Racket code:

`(define x 3)`
`(define f1`
`  (lambda (x)`
`    (let ([y (+ x 1)])`
`      (+ y x))))`
	`(define x 3)`
	`(define f2`
	`  (let ([y (+ x 1)])`
	`    (lambda (x)`
	`      (+ y x))))`

What is **`(f1 2)`** bound to?

What is **`(f2 2)`** bound to?

Lists in Racket

Empty list: **`null`**

Cons constructor: **`cons`**

Access head of list: **`car`**

Access tail of list: **`cdr`**

Check for empty: **`null?`**

Notes:

- Can also use **`(list e1 … en)`** for building lists

Examples:

```
(define list1 (cons 3 (cons 4 (cons 1 null))))
(define list2 (list 3 4 1))
```

SML *vs.* *Racket*

```
val empty = []
val list1 = [1,2,3]
val list2 = 1 :: 2 :: 3 :: []
val b1 = null empty
val h1 = hd list1
val t1 = tl list1
```

```
#lang racket

(define empty null)
(define list1 (list 1 2 3))
(define list2
    (cons 1 (cons 2 (cons 3 null))))
(define b1 (null? empty))
(define h1 (car list1))
(define t1 (cdr list1))
```

Practice with Lists

See worksheet Q4/5

Delayed Evaluation with Thunks

Thunks:

Zero-argument functions which wrap around an expression to be evaluated when needed:

```
(lambda() e)
```

Delay and Force: Review

Q: What do the following functions do?

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

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

Q: Where are any thunks used here?

Streams: Example

```
(define nats
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (+ x 1)))))])
    (lambda () (f 1))))
```

Q:

How would you get the second number in this stream and save it as a variable x?

Streams

- A stream is an *infinite sequence* of values
  - So cannot make a stream by making all the values
  - Key idea: Use a thunk to delay creating most of the sequence
  - Just a programming idiom
- A powerful concept for division of labor:
  - Stream producer knows how to create any number of values
  - Stream consumer decides how many values to ask for
- Some examples of streams you might (not) be familiar with:
  - User actions (mouse clicks, etc.)
  - UNIX pipes: **`cmd1 | cmd2`** has **`cmd2`** "pull" data from **`cmd1`**
  - Output values from a sequential feedback circuit

Using Streams

We will represent streams using pairs and thunks

Let a stream be a thunk that *when called* returns a pair:

`'(next-answer . next-thunk)`

So given a stream **`s`**, the client can get any number of elements

- First: **`car (s))`**
- Second: **`(car ((cdr (s))))`**
- Third: **`(car ((cdr ((cdr (s))))))`**

(Usually bind **`(cdr (s))`** to a variable or pass to a recursive function)

Streams

- Functions which represent an infinite sequence of values
- When a stream **`s`** is evaluated, results in a pair with a value in **`(car s)`** and another stream in **`(cdr s)`**

Practice with Thunks and Streams

Select worksheet questions

Example using streams

This function returns how many stream elements it takes to find one for which tester does not return **`#f`**

- Happens to be written with a tail-recursive helper function

```
(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)))
```

- **`(stream)`** generates the pair
- So recursively pass **`(cdr pr)`**, the thunk for the rest of the infinite sequence