Racket

Next 2+ weeks 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.
    - But doesn’t rely on pattern-matching
  - Unlike ML, 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, …
    - Will do only a couple of these

Racket vs. Scheme

- Scheme and Racket are very similar languages
  - Racket "changed its name" in 2010
  - Notes and instructor may occasionally slip up
- Racket made some non-backward-compatible changes…
  - How the empty list is written
  - Cons cells not mutable
  - How modules work
  - Etc.
  - … and many additions
- Result: A modern language used to build some real systems
  - More of a moving target (notes may become outdated)
  - Online documentation, particular “The Racket Guide”

Getting started

DrRacket “definitions window” and “interactions window” very similar to how we used emacs and a REPL
- DrRacket has always focused on good-for-teaching
- See usage notes for how to use REPL, testing files, etc.
  - You need to get good at learning new tools on your own, but today’s demos (more code than in slides) will help

Start every file with a line containing only
```
#lang racket
```
(Can have comments before this, but not code)

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))))))
```

Some niceties

Many built-in functions (a.k.a. procedures) take any number of args
- Yes * is just a function
- Yes we’ll show you later how to define variable-arity functions

```
(define cube x)
  (* x x x)
(define (pow x y)
  (if (= y 0)
    1
    (* x (pow x (- y 1)))))
```

Better style for non-anonymous function definitions (just sugar):
```
(define (cube x)
  (* x x x))
```
Old-friend #1: currying
Currying is an idiom that works in any language with closures
  – Less common in Racket because it has real multiple args

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

Sugar for defining curried functions:
```
(define (pow x) (if ...)
```
(No sugar for calling curried functions)

Old-friend #2: List processing
Empty list: null (unlike Scheme, () doesn’t work, but ’() does)
Cons constructor: cons (also (list e1 … en) is convenient)
Access head of list: car (car and cdr a historical accident)
Access tail of list: cdr
Check for empty: null?

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

Racket syntax
Ignoring a few bells and whistles,
Racket has an amazingly simple syntax

A **term** (anything in the language) is either:
  – An **atom**, e.g., #t, #f, 34, "hi", null, 4.0, x, …
  – A **special form**, e.g., define, lambda, if
    • Macros will let us define our own
    – A **sequence** of terms in parens: (t1 t2 … tn)

Note: Can use [ anywhere you use (, but must match with ]
  – Will see shortly places where [...] is common style
  – DrRacket lets you type ) to match

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)
Also makes indentation easy

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

Parenthesis bias
• If you look at the HTML for a web page, it takes the same approach:
  – (foo written <foo>
  – ) written </foo>

• But for some reason, LISP/Scheme/Racket is the target of subjective parenthesis-bashing
  – Bizarrely, often by people who have no problem with HTML
  – You are entitled to your opinion about syntax, but a good historian wouldn’t refuse to study a country where he/she didn’t like people’s accents

```
LISP is over half a century old and it still has this perfect timeless awkwardness.
I WOULD IF THE CYCLES WOULD CONTINUE FOREVER.
TRANQUIL WARRIOR, NEW GENERATION ... DISCOVERING THE LISP ARMS.
LISP invented around 1959 by
  • Invented garbage collection
```
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

Example

Correct:

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

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

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

Gives if 5 arguments (syntax error)

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

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

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

Dynamic typing

Will spend a later lecture contrasting static typing (e.g., ML) with dynamic typing (e.g., Racket)

For now:
- Frustrating not to catch “little errors” like (n * x) until you test your function
- But can use very flexible data structures and code without convincing a type checker that it makes sense

Example:
- A list that can contain numbers or other lists
- Assuming lists or numbers “all the way down,” sum all the numbers…

Example

```scheme
(define (sum xs)
  (cond [(null? xs) 0]
        [(number? (car xs))
          (+ (car xs) (sum (cdr xs)))
        [#[t]
          (+ (sum (car xs)) (sum (cdr xs)))]))
```

- No need for a fancy datatype binding, constructors, etc.
- Works no matter how deep the lists go
- But assumes each element is a list or a number
  - Will get a run-time error if anything else is encountered

Better style

Avoid nested if-expressions when you can use cond-expressions instead
- Can think of one as sugar for the other

General syntax: (cond [(e1a e1b) [e2a e2b] ... [eNa eNb]])
- Good style: eNa should be #t

Example:

```scheme
(define (sum xs)
  (cond [(null? xs) 0]
        [(number? (car xs))
          (+ (car xs) (sum (cdr xs)))
        [#[t]
          (+ (sum (car xs)) (sum (cdr xs)))]))
```

A variation

We could change our spec to say instead of errors on non-numbers, we should just ignore them (same as adding 0)

So this version can work for any argument in all of Racket – will never raise an error
- Compare carefully, we did not just add a branch


**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
  - Like in ML, the 3 kinds of let-expressions can appear anywhere

**Let**

A let expression can bind any number of local variables
- Notice where all the parentheses are

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]) ...)`

```
(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

```
(define (silly-double x)
  (let* ([x (+ x 3)]
         [y (+ x 2)])
    (+ x y -8)))
```

**Letrec**

Syntactically, a letrec expression is also the same

The expressions are evaluated in the environment that includes **all the bindings**
- Needed for mutual recursion
- But expressions are still evaluated in order: accessing an uninitialized binding would produce `<undefined>`
  - Would be bad style and surely a bug
  - Remember function bodies not evaluated until called

```
(define (silly-triple x)
  (letrec ([y (+ x 2)]
            [f (lambda(z) (+ z y w x))]
            [w (+ x 7)])
    (f -9)))
```

**More letrec**

- Letrec is ideal for recursion (including mutual recursion)

```
(define (silly-mod2 x)
  (letrec ([even? (lambda(x)
                   (if (zero? x) #t (odd? (- x 1))))]
           [odd? (lambda(x)
                   (if (zero? x) #f (even? (- x 1))))]
           [if (lambda(x y z)
                (if (zero? x) y z)])
           (if (even? x) 0 1))
    (if (even? x) 0 1))
```

- Do not use later bindings except inside functions
  - This example will return `<undefined>` if `x` is true
  - (By the way, everything is true except `#f`)

```
(define (bad-letrec x)
  (letrec ([y z]
            [z 13])
    (if (even? x) y z)))
```

**Local defines**

- In certain positions, like the beginning of function bodies, you can put defines
  - For defining local variables, same semantics as `letrec`

```
(define (silly-mod2 x)
  (define (even? x)
    (if (zero? x) #t (odd? (- x 1))))
  (define (odd? x)
    (if (zero? x) #f (even? (- x 1))))
  (if (even? x) 0 1))
```
Top-level

The bindings in a file / module work like local defines, i.e., \texttt{letrec}
\begin{itemize}
  \item Like ML, you can refer to earlier bindings
  \item Unlike ML, you can refer to later bindings
  \item But refer to later bindings only in function bodies
    \begin{itemize}
      \item Detail: Will get an error instead of \texttt{#<undefined>}
      \item Unlike ML, cannot define the same variable twice in module
      \item Would make no sense; can’t have both in environment
    \end{itemize}
\end{itemize}

If each file is its own module, what is externally visible and how do you refer to bindings in other files?
\begin{itemize}
  \item Later lecture
  \item See usage notes for a way to test homework from a second file
\end{itemize}