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.
  - But we will not use 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
- 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  - More of a moving target: notes may become outdated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, particularly “The Racket Guide”

Getting started

DrRacket “definitions window” and “interactions window” very similar to how we used Emacs and a REPL, but more user-friendly
- DrRacket has always focused on good-for-teaching
- See usage notes for how to use REPL, testing files, etc.
- Easy to learn to use on your own, but lecture demos will help

Free, well-written documentation:
- http://racket-lang.org/
- The Racket Guide especially,
  http://docs.racket-lang.org/guide/index.html

File structure

Start every file with a line containing only

```racket
#lang racket
```
(Can have comments before this, but not code)

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

Example

```racket
#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:

- `*` is just a function
- You can define your own variable-arity functions (not shown here)

Better style for non-anonymous function definitions (just sugar):

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

```scheme
(define (cube x) (* x x x))
```

```scheme
(define (pow x 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))
```

Sugar for defining curried functions:

```scheme
(define ((pow x) y) (if …
```

(No sugar for calling curried functions)

---

**An old friend: currying**

Currying is an idiom that works in any language with closures:

- Less common in Racket because it has real multiple args

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

---

**Another old-friend: List processing**

Empty list: `null`
Cons constructor: `cons`
Access head of list: `car`
Access tail of list: `cdr`
Check for empty: `null?`

Notes:
- Unlike Scheme, (``) doesn’t work for `null`, but `'( )` does
- `'(list e1 ... en)` for building lists
- Names `car` and `cdr` are a historical accident

---

**Examples**

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

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

```scheme
(define (my-map f xs)
  (if (null? xs)
    null
    (cons (f (car xs)) (my-map f (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 paren: `(t1 t2 ... tn)`
  - If t1 a special form, semantics of sequence is special
  - Else a function call

- Example: `(+ 3 (car xs))`
- Example: `(lambda (x) (if x "hi" #t))`

---

**Brackets**

Minor note:

Can use `[ ]` anywhere you use `, but must match with ]`
- Will see shortly places where `[...]` is common style
- DrRacket lets you type `)` and replaces it with `]` 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

Example:

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

No need to discuss "operator precedence" (e.g., \(x + y \times z\))

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

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

Examples (more in code)

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)))))
```

Dynamic typing

Major topic coming later: contrasting static typing (e.g., ML) with dynamic typing (e.g., Racket)

For now:
- Frustrating not to catch "little errors" like \(n \times x\) 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)
  (if (null? xs)
      0
      (+ (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

As before, we could change our spec to say instead of errors on non-numbers, we should just ignore them
So this version can work for any list (or just a number)
- Compare carefully, we did not just add a branch

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

What is true?

For both if and cond, test expression can evaluate to anything
- It is not an error if the result is not #t or #f
- (Apologies for the double-negative 😊)

Semantics of if and cond:
- “Treat anything other than #f as true”
- (In some languages, other things are false, not in Racket)

This feature makes no sense in a statically typed language

Some consider using this feature poor style, but it can be convenient

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 \( \text{let } ([x \ y] [y \ x]) \ldots \)

\[
\text{(define (silly-double x)}
\text{(let ([x (+ x 3)] [y (+ x 2)]) [+ x y -5]))}
\]

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

\[
\text{(define (silly-double x)}
\text{(let* ([x (+ x 3)] [y (+ x 2)]) [+ x y -8]))}
\]

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 raises an error
  • Remember function bodies not evaluated until called

\[
\text{(define (silly-triple x)}
\text{(letrec ([y (+ x 2)] [f (lambda(z) (+ z y w x)]) [w (+ x 7)]) [+ x y -5]))}
\]

More letrec

• Letrec is ideal for recursion (including mutual recursion)

\[
\text{(define (silly-mod2 x)}
\text{(letrec ([even? (lambda(x) (if (zero? x) #t (odd? (- x 1))))]
[odd? (lambda(x) (if (zero? x) #f (even? (- x 1))))]
(if (even? x) 0 1))}
\]

• Do not use later bindings except inside functions
  • This example will raise an error when called

\[
\text{(define (bad-letrec x)}
\text{(letrec ([y z]
[z 13])}
\text{(if x y z))}
\]

Local defines

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

\[
\text{(define (silly-mod2 x)}
\text{(define (even? x) (if (zero? x) #t (odd? (- x 1))))
(define (odd? x) (if (zero? x) #f (even? (- x 1))))
(if (even? x) 0 1))}
\]

• Local defines is preferred Racket style, but course materials will avoid them to emphasize let, let*, letrec distinction
  • You can choose to use them on homework or not

Top-level

The bindings in a file work like local defines, i.e., letrec
– Like ML, you can refer to earlier bindings
– Unlike ML, you can also refer to later bindings
– But refer to later bindings only in function bodies
  • Because bindings are evaluated in order
  • Get an error if try to use a not-yet-defined binding
– Unlike ML, cannot define the same variable twice in module
  • Would make no sense: cannot have both in environment
**REPL**

Unfortunate detail:
- REPL works slightly differently
  - Not quite `let*` or `letrec`
  - `let`
- Best to avoid recursive function definitions or forward references in REPL
  - Actually okay unless shadowing something (you may not know about) – then weirdness ensues
  - And calling recursive functions is fine of course

---

**Optional: Actually…**

- Racket has a module system
  - Each file is implicitly a module
    - Not really “top-level”
  - A module can shadow bindings from other modules it uses
    - Including Racket standard library
  - So we could redefine + or any other function
    - But poor style
    - Only shadows in our module (else messes up rest of standard library)

- (Optional note: Scheme is different)

---

**Set!**

- Unlike ML, Racket really has assignment statements
  - But used only-when-really-appropriate!
  
    ```racket```

    ```(set! x e)```

- For the `x` in the current environment, subsequent lookups of `x`
  get the result of evaluating expression `e`
  - Any code using this `x` will be affected
  - Like `x = e` in Java, C, Python, etc.

- Once you have side-effects, sequences are useful:

  ```racket```

  ```(begin e1 e2 ... en)```

---

**Example**

Example uses `set!` at top-level; mutating local variables is similar

```racket```

```(define b 3)
(define f (lambda (x) (* 1 (+ x b))))
(define c (+ b 4)) ; 7
(set! b 5)
(define z (f 4)) ; 9
(define w c) ; 7```

Not much new here:
- Environment for closure determined when function is defined, but body is evaluated when function is called
- Once an expression produces a value, it is irrelevant how the value was produced

---

**Top-level**

- Mutating top-level definitions is particularly problematic
  - What if any code could do `set!` on anything?
  - How could we defend against this?

- A general principle: If something you need not to change might change, make a local copy of it. Example:

```racket```

```(define b 3)
(define f
  (let ([b b]
        [+ +]
        [* *])
    (lambda (x) (* 1 (+ x b)))))```

Could use a different name for local copy but do not need to

---

**But wait…**

- Simple elegant language design:
  - Primitives like + and * are just predefined variables bound to functions
  - But maybe that means they are mutable
  - Example continued:

```racket```

```(define f
  (let ([b b]
        [+ +]
        [* *])
    (lambda (x) (* 1 (+ x b))))))```

- Even that won’t work if `f` uses other functions that use things that might get mutated – all functions would need to copy everything mutable they used
No such madness

In Racket, you do not have to program like this
- Each file is a module
- If a module does not use `set!` on a top-level variable, then Racket makes it constant and forbids `set!` outside the module
- Primitives like `+`, `*`, and `cons` are in a module that does not mutate them

Showed you this for the concept of copying to defend against mutation
- Easier defense: Do not allow mutation
- Mutable top-level bindings a highly dubious idea

The truth about `cons`

`cons` just makes a pair
- Often called a `cons cell`
- By convention and standard library, lists are nested pairs that eventually end with `null`

```scheme
(define pr (cons 1 (cons #t "hi"))) ; '(1 #t . "hi")
(define lst (cons 1 (cons #t (cons "hi" null)))))
(define hi-again (car (cdr (cdr lst))))
(define hi-another (caddr lst))
(define no (list? pr))
(define yes (pair? pr))
(define of-course (and (list? lst) (pair? lst)))
```

Passing an improper list to functions like `length` is a run-time error

The truth about `cons`

So why allow improper lists?
- Pairs are useful
- Without static types, why distinguish `(el,e2)` and `el::e2`

Style:
- Use proper lists for collections of unknown size
- But feel free to use `cons` to build a pair
  - Though structs (like records) may be better

Built-in primitives:
- `list?` returns true for proper lists, including the empty list
- `pair?` returns true for things made by `cons`
  - All improper and proper lists except the empty list

Set! does not change list contents

This does not mutate the contents of a `cons` cell:

```scheme
(define x (cons 14 null))
(define y x)
(set! x (cons 42 null))
(define fourteen (car y))
```

- Like Java's `x = new Cons(42,null); not x.car = 42`

Mcons cells are mutable

Since mutable pairs are sometimes useful (will use them soon), Racket provides them too:
- `mcons`
- `mcar`
- `mcdr`
- `mpair?`
- `set-mcar!`
- `set-mcdr!`

Run-time error to use `mcar` on a `cons` cell or `car` on an `mcons` cell