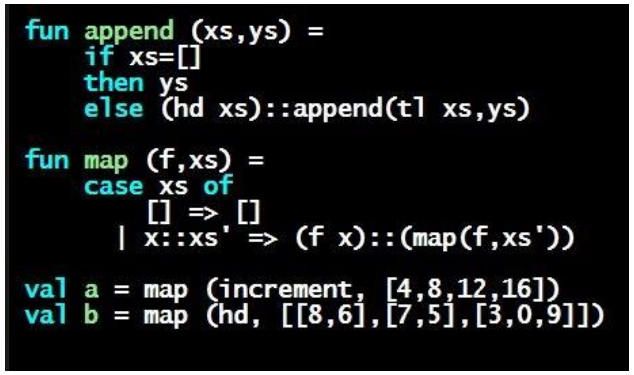


Programming Languages

Dan Grossman

Introduction to Racket

Racket

Next two sections 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
  - Please excuse any mistakes when I speak
- 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  - More of a moving target (notes may become outdated)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

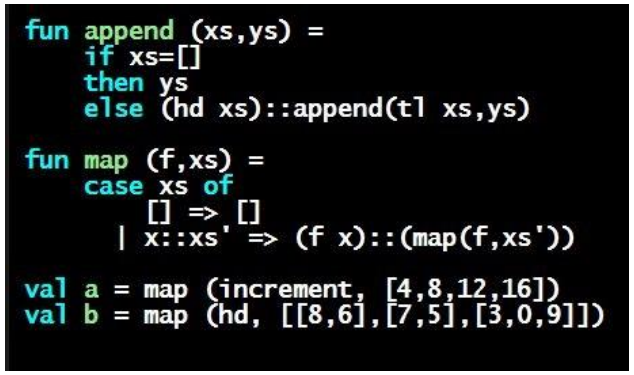
File structure

Start every file with a line containing only

```
#lang racket
```

(Can have comments before this, but not code)

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



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Racket Definitions, Functions, Conditionals

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 you can define your own *variable-arity* functions (not shown here)

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

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

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

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

An old friend: currying

Currying is an idiom that works in any language with closures

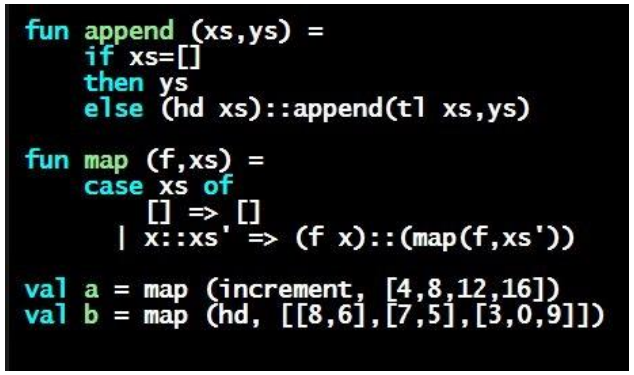
- Less common in Racket because it has real multiple args

```
(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))
```

Sugar for defining curried functions: `(define ((pow x) y) (if …`

(No sugar for calling curried functions)



Programming Languages

Dan Grossman

Racket Lists

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:

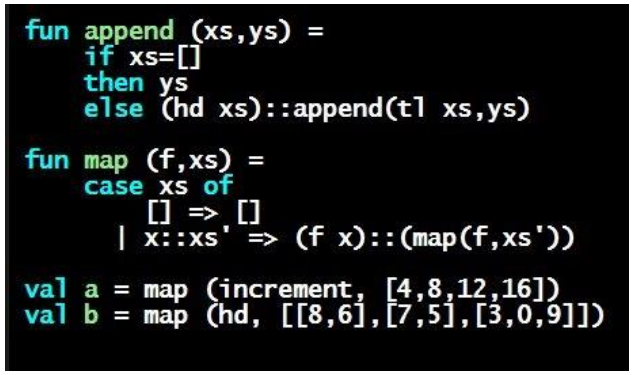
- Unlike Scheme, **`()`** doesn't work for **`null`**, but **`'()`** does
- **`(list e1 … en)`** for building lists
- Names **`car`** and **`cdr`** are a historical accident

Examples

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



Programming Languages

Dan Grossman

Syntax and Parentheses

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)`**
  - 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?

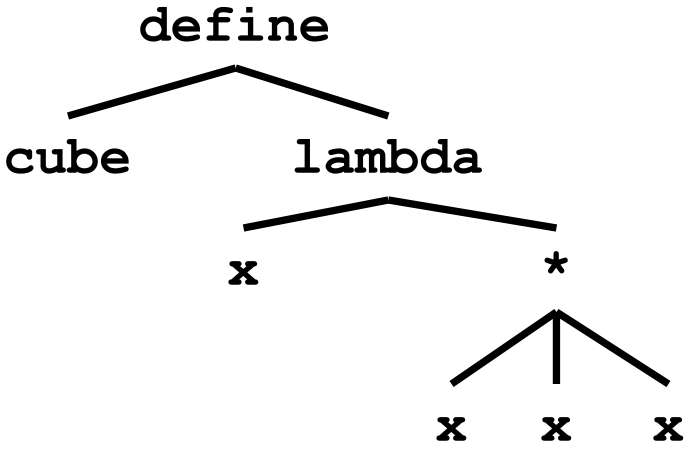
By parenthesizing everything, converting the program text into a tree representing the program (*parsing*) is trivial and unambiguous

- 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 * z`)

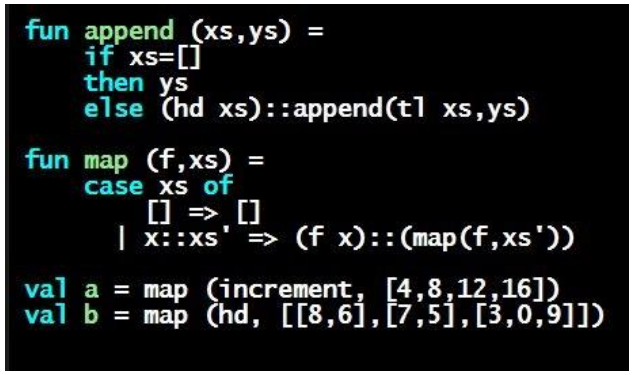
Parenthesis bias

- 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

For fun…

http://xkcd.com/297



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Parentheses Matter! (Debugging Practice)

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)))))
```

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Dynamic Typing

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 * x)` until you test your function
- But can use very flexible data structures and code without convincing a type checker that it makes sense

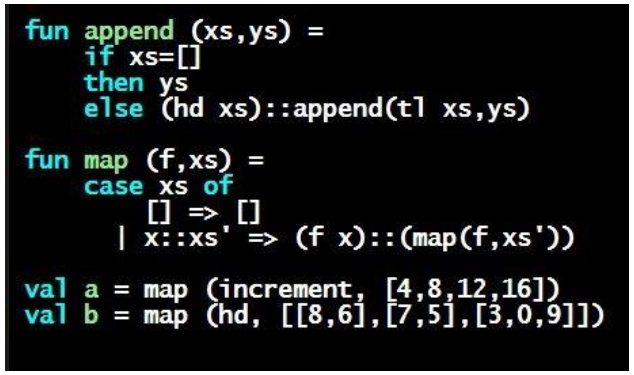
Example:

- A list that can contain numbers or other lists
- Assuming *lists or numbers "all the way down,"* sum all the numbers…

Example

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



Programming Languages

Dan Grossman

Cond

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

```
(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

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

What is true?

For both **`if`** and **`cond`**, test expression can evaluate to anything

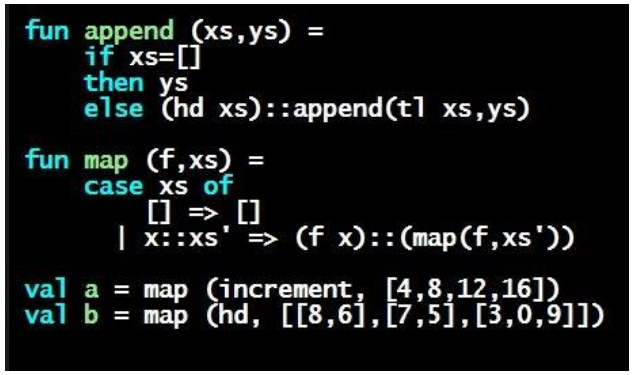
- It is not an error if the result is not **`#t`** or **`#f`**
- (Apologies for the double-negative ☺)

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



Programming Languages

Dan Grossman

Local Bindings

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**

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

- Needed for mutual recursion
- But expressions are still *evaluated in order*: accessing an uninitialized binding produces an error
  - Remember function bodies not evaluated until called

More letrec

- Letrec is ideal for recursion (including mutual recursion)

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

- Do not use later bindings except inside functions
  - This example will produce an error if **x** is not **#f**

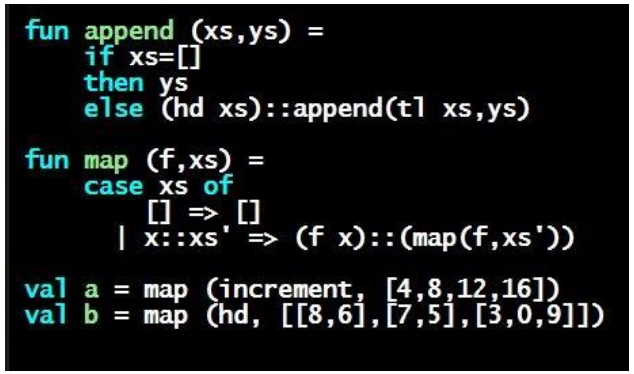
```
(define (bad-letrec x)
  (letrec ([y z]
           [z 13])
    (if x y z)))
```

Local defines

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

```
(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))
```

- 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



Programming Languages

Dan Grossman

Top-Level Bindings

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
  - Will get an error if you access an uninitialized variable
- 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`**
  - ☹
- 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 with interesting difference from ML
  - 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)

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Mutation With **`set!`**

Set!

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

```
(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:

```
(begin e1 e2 … en)
```

Example

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

```
(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:

```
(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:

```
(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

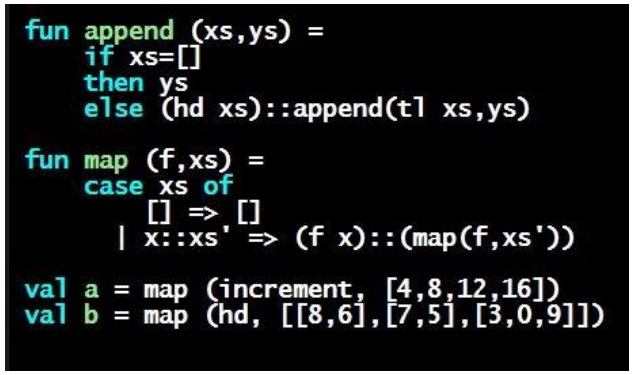
No such madness

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



Programming Languages

Dan Grossman

The Truth About **`cons`**

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`**

```
(define pr (cons 1 (cons #t "hi"))) ; '(1 #t . "hi")
(define lst (cons 1 (cons #t (cons "hi" null))))
(define hi (cdr (cdr pr)))
(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?

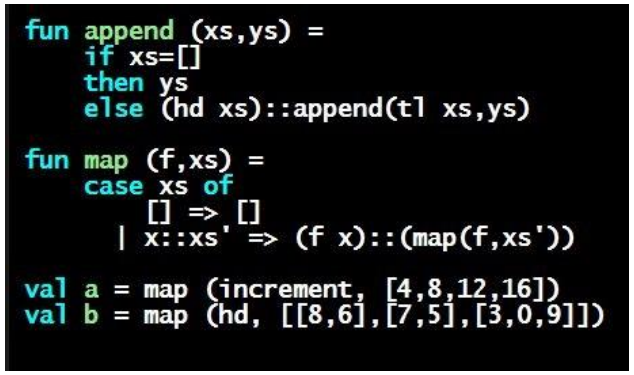
- Pairs are useful
- Without static types, why distinguish **`(e1,e2)`** and **`e1::e2`**

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



Programming Languages

Dan Grossman

`mcons` For Mutable Pairs

cons cells are immutable

What if you wanted to mutate the *contents* of a cons cell?

- In Racket you cannot (major change from Scheme)
- This is good
  - List-aliasing irrelevant
  - Implementation can make **`list?`** fast since listness is determined when cons cell is created

Set! does not change list contents

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

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

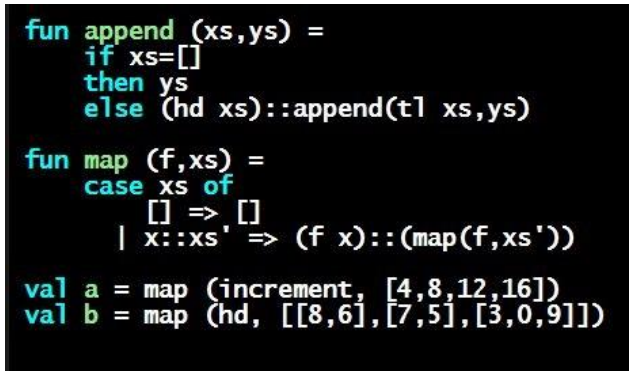
– Like Java's **`x = new Cons(42,null)`**, *not* **`x.car = 42`**

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



Programming Languages

Dan Grossman

Delayed Evaluation and Thunks

Delayed evaluation

For each language construct, the semantics specifies when subexpressions get evaluated. In ML, Racket, Java, C:

- Function arguments are *eager* (call-by-value)
  - Evaluated once before calling the function
- Conditional branches are not eager

It matters: calling **`factorial-bad`** never terminates:

```
(define (my-if-bad x y z)
  (if x y z))

(define (factorial-bad n)
  (my-if-bad (= n 0)
             1
             (* n (factorial-bad (- n 1)))))
```

Thunks delay

We know how to delay evaluation: put expression in a function!

- Thanks to closures, can use all the same variables later

A zero-argument function used to delay evaluation is called a *thunk*

- As a verb: *thunk the expression*

This works (but it is silly to wrap **`if`** like this):

```
(define (my-if x y z)
  (if x (y) (z)))

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

The key point

- Evaluate an expression **`e`** to get a result:

```
e
```

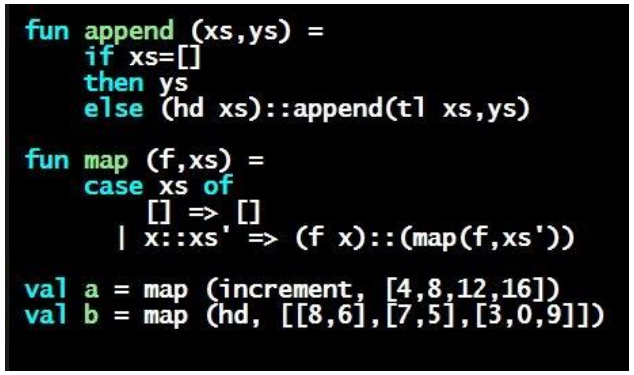
- A function that *when called*, evaluates **`e`** and returns result
  - Zero-argument function for “thunking”

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

- Evaluate **`e`** to some thunk and then call the thunk

```
(e)
```

- Next: Powerful idioms related to delaying evaluation and/or avoided repeated or unnecessary computations



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Avoiding Unnecessary Computations

Avoiding expensive computations

Thunks let you skip expensive computations if they are not needed

Great if you take the true-branch:

```
(define (f th)
  (if (…) 0 (…  (th) …)))
```

But worse if you end up using the thunk more than once:

```
(define (f th)
  (… (if (…) 0 (… (th) …))
     (if (…) 0 (… (th) …))
     …
     (if (…) 0 (… (th) …))))
```

In general, might not know how many times a result is needed

Best of both worlds

Assuming some expensive computation has no side effects, ideally we would:

- Not compute it *until needed*
- *Remember the answer* so future uses complete immediately

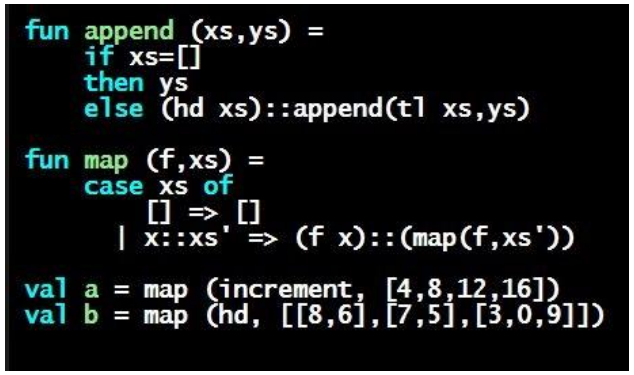
Called *lazy evaluation*

Languages where most constructs, including function arguments, work this way are *lazy languages*

- Haskell

Racket predefines support for *promises*, but we can make our own

- Thunks and mutable pairs are enough



Programming Languages

Dan Grossman

Delay and Force

Best of both worlds

Assuming some expensive computation has no side effects, ideally we would:

- Not compute it *until needed*
- *Remember the answer* so future uses complete immediately

Called *lazy evaluation*

Delay and force

```
(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))))
```

An ADT represented by a mutable pair

- `#f` in *car* means *cdr* is unevaluated thunk
  - Really a one-of type: thunk or result-of-thunk
- Ideally hide representation in a module

Using promises

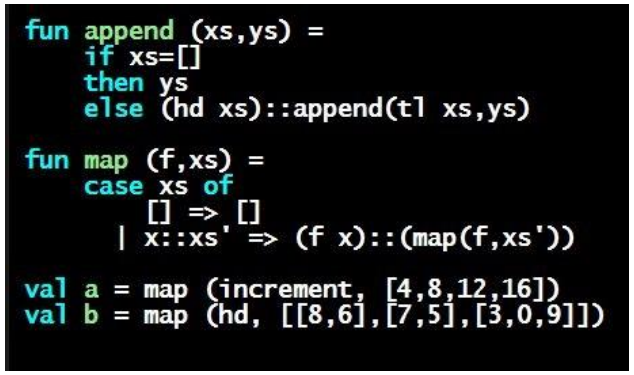
```
(define (f p)
  (… (if (…) 0 (… (my-force p) …))
     (if (…) 0 (… (my-force p) …))
     …
     (if (…) 0 (… (my-force p) …))))
```

```
(f (my-delay (lambda () e)))
```

Lessons From Example

See code file for example that does multiplication using a very slow addition helper function

- With thunking second argument:
  - *Great* if first argument 0
  - Okay if first argument 1
  - *Worse* otherwise
- With precomputing second argument:
  - *Okay* in all cases
- With thunk that uses a promise for second argument:
  - *Great* if first argument 0
  - *Okay* otherwise



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Using Streams

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 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)

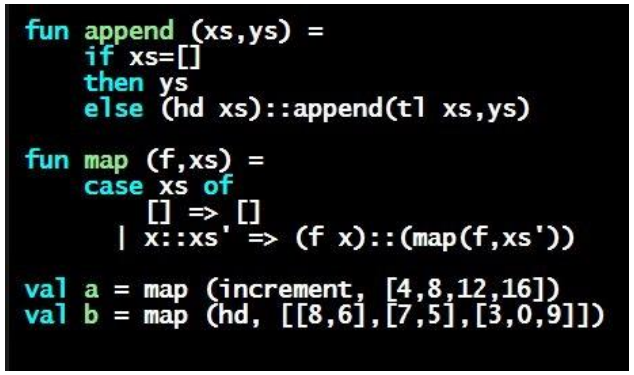
Example using streams

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



Programming Languages

Dan Grossman

Defining Streams

Streams

Coding up a stream in your program is easy

- We will do functional streams using pairs and thunks

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

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

Saw how to use them, now how to make them…

- Admittedly mind-bending, but uses what we know

Making streams

- How can one thunk create the right next thunk? Recursion!
  - Make a thunk that produces a pair where cdr is next thunk
  - A recursive function can return a thunk where recursive call does not happen until thunk is called

```
(define ones (lambda () (cons 1 ones)))

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

(define powers-of-two
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (* x 2)))))])
    (lambda () (f 2))))
```

Getting it wrong

- This uses a variable before it is defined

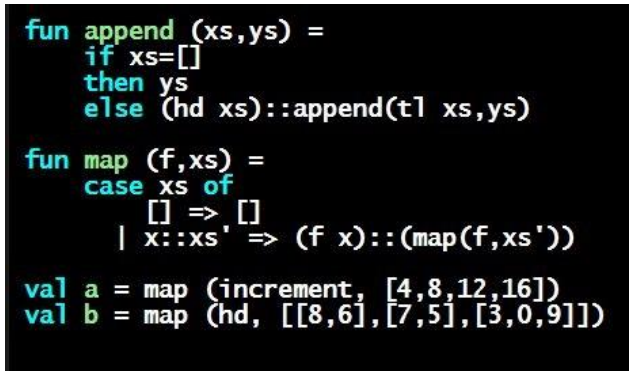
```
(define ones-really-bad (cons 1 ones-really-bad))
```

- This goes into an infinite loop making an infinite-length list

```
(define ones-bad (lambda () cons 1 (ones-bad)))
(define (ones-bad)(cons 1 (ones-bad)))
```

- This is a stream: thunk that returns a pair with cdr a thunk

```
(define ones (lambda () (cons 1 ones)))
(define (ones)(cons 1 ones))
```



Programming Languages

Dan Grossman

Memoization

Memoization

- If a function has no side effects and does not read mutable memory, no point in computing it twice for the same arguments
  - Can keep a *cache* of previous results
  - Net win if (1) maintaining cache is cheaper than recomputing and (2) cached results are reused

- Similar to promises, but if the function takes arguments, then there are multiple "previous results"

- For recursive functions, this *memoization* can lead to *exponentially* faster programs
  - Related to algorithmic technique of dynamic programming

How to do memoization: see example

- Need a (mutable) cache that all calls using the cache share
  - So must be defined *outside* the function(s) using it

- See code for an example with Fibonacci numbers

  - Good demonstration of the idea because it is short, but, as shown in the code, there are also easier less-general ways to make **`fibonacci`** efficient

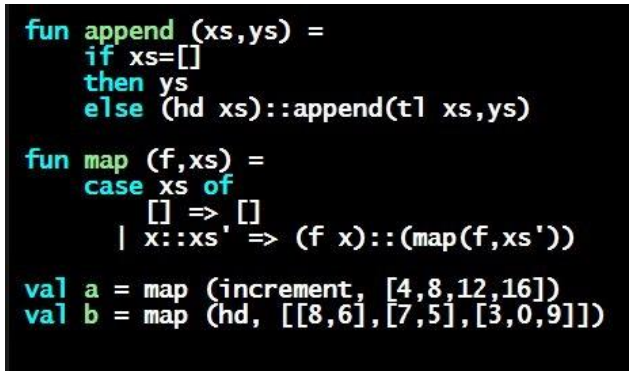
  - (An association list (list of pairs) is a simple but sub-optimal data structure for a cache; okay for our example)

assoc

- Example uses **`assoc`**, which is just a library function you could look up in the Racket reference manual:

  `(assoc v lst)` takes a list of pairs and locates the first element of **`lst`** whose car is equal to **`v`** according to **`is-equal?`**. If such an element exists, the pair (i.e., an element of **`lst`**) is returned. Otherwise, the result is **`#f`**.

- Returns **`#f`** for not found to distinguish from finding a pair with **`#f`** in cdr



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Macros: The Key Points

Rest of Section

- This segment: High-level idea of macros
  - Not needed for this section's homework
  - Needed to build on a little in next section

- Then: Racket's macro system and pitfalls of macros
  - Defining a macro is this section's homework challenge
  - "Macros done right" a key feature of Racket compared to other macro systems (e.g., C/C++)

- Segments *after* this one optional
  - Conserving time and will not build on it

What is a macro

- A *macro definition* describes how to transform some new syntax into different syntax in the source language
- A macro is one way to implement syntactic sugar
  - "Replace any syntax of the form **`e1 andalso e2`** with **`if e1 then e2 else false`**"
- A *macro system* is a language (or part of a larger language) for defining macros
- *Macro expansion* is the process of rewriting the syntax for each *macro use*
  - Before a program is run (or even compiled)

Using Racket Macros

- If you define a macro **`m`** in Racket, then **`m`** becomes a new special form:
  - Use **`(m …)`** gets expanded according to definition

- Example definitions (actual definitions in optional segment):
  - Expand **`(my-if e1 then e2 else e3)`**
    to **`(if e1 e2 e3)`**
  - Expand **`(comment-out e1 e2)`**
    to **`e2`**
  - Expand **`(my-delay e)`**
    to **`(mcons #f (lambda () e))`**

Example uses

It is like we added keywords to our language

- Other keywords only keywords in uses of that macro
- Syntax error if keywords misused
- Rewriting ("expansion") happens before execution

```
(my-if x then y else z)   ; (if x y z)
(my-if x then y then z)   ; syntax error

(comment-out (car null) #f)

(my-delay (begin (print "hi") (foo 15)))
```

Overuse

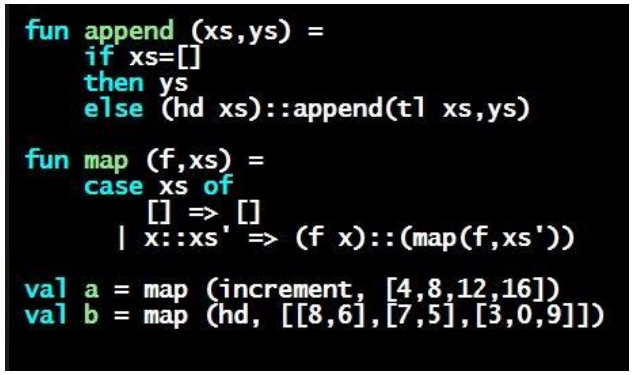
Macros often deserve a bad reputation because they are often overused or used when functions would be better

When in doubt, resist defining a macro?

But they can be used well and the optional material should help

Optional stuff ahead

- How any macro system must deal with tokens, parentheses, and scope
- How to define macros in Racket
- How macro definitions must deal with expression evaluation carefully
  - Order expressions evaluate and how many times
- The key issue of variable bindings in macros and the notion of *hygiene*
  - Racket is superior to most languages here



```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Optional: Tokenization, Parenthesization, and Scope

Tokenization

First question for a macro system: How does it tokenize?

- Macro systems generally work at the level of *tokens* not sequences of characters
  - So must know how programming language tokenizes text

- Example: "macro expand **`head`** to **`car`**"
  - Would not rewrite **`(+ headt foo)`** to **`(+ cart foo)`**
  - Would not rewrite **`head-door`** to **`car-door`**
    - But would in C where **`head-door`** is subtraction

Parenthesization

Second question for a macro system: How does associativity work?

C/C++ basic example:

```
#define ADD(x,y) x+y
```

Probably *not* what you wanted:

`ADD(1,2/3)*4` means `1+2/3*4` not `(1+2/3)*4`

So C macro writers use lots of parentheses, which is fine:

```
#define ADD(x,y) ((x)+(y))
```

Racket won't have this problem:

- Macro use: `(macro-name …)`
- After expansion: `( something else in same parens )`

Local bindings

Third question for a macro system: Can variables shadow macros?

Suppose macros also apply to variable bindings. Then:

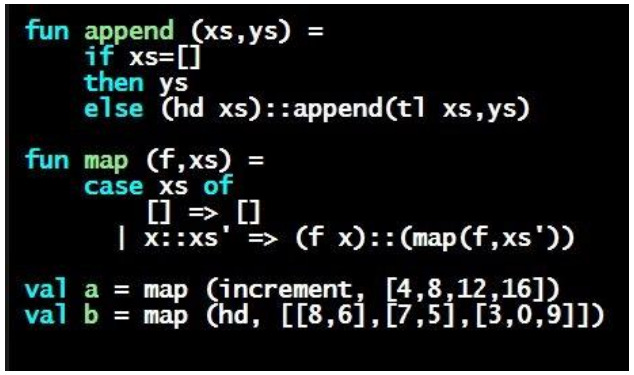
```
(let  ([head 0][car 1]) head) ; 0
(let* ([head 0][car 1]) head) ; 0
```

Would become:

```
(let  ([car 0][car 1]) car) ; error
(let* ([car 0][car 1]) car) ; 1
```

This is why C/C++ convention is all-caps macros and non-all-caps for everything else

Racket does *not* work this way – it gets scope "right"!



Programming Languages

Dan Grossman

Optional: Racket Macros with **`define-syntax`**

Example Racket macro definitions

Two simple macros

```
(define-syntax my-if                    ; macro name
  (syntax-rules (then else)             ; other keywords
    [(my-if e1 then e2 else e3)  ; macro use
     (if e1 e2 e3)]))                   ; form of expansion
```

```
(define-syntax comment-out              ; macro name
  (syntax-rules ()                      ; other keywords
    [(comment-out ignore instead)  ; macro use
     instead]))                         ; form of expansion
```

If the form of the use matches, do the corresponding expansion

- In these examples, list of possible use forms has length 1
- Else syntax error

Revisiting delay and force

Recall our definition of promises from earlier

- Should we use a macro instead to avoid clients' explicit thunk?

```
(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))))
```

```
(f (my-delay (lambda () e)))
```

```
(define (f p)
  (… (my-force p) …))
```

A delay macro

- A macro can put an expression under a thunk
  - Delays evaluation without explicit thunk
  - Cannot implement this with a function
- Now client should *no*t use a thunk (that would double-thunk)
  - Racket's pre-defined **`delay`** is a similar macro

```
(define-syntax my-delay
  (syntax-rules ()
    [(my-delay e)
     (mcons #f (lambda() e))]))
```

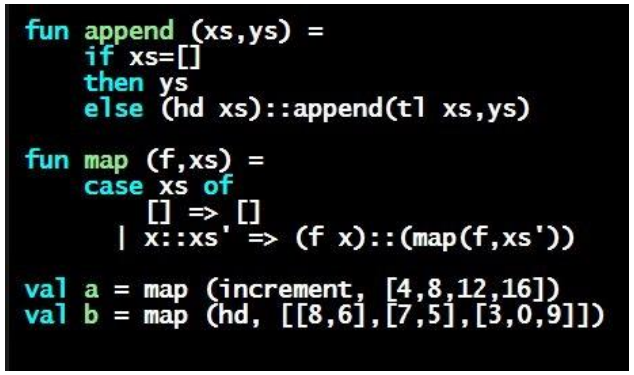
```
(f (my-delay e))
```

What about a force macro?

We could define **`my-force`** with a macro too

- Good macro style would be to evaluate the argument exactly once (use **`x`** below, not multiple evaluations of **`e`**)
- Which shows it is bad style to use a macro at all here!
- Do not use macros when functions do what you want

```
(define-syntax my-force
  (syntax-rules ()
    [(my-force e)
     (let([x e])
        (if (mcar x)
            (mcdr x)
            (begin (set-mcar! x #t)
                   (set-mcdr! p ((mcdr p)))
                   (mcdr p)))))]))
```



Programming Languages

Dan Grossman

Optional: Variables, Macros, and Hygiene

Another bad macro

Any *function* that doubles its argument is fine for clients

```
(define (dbl x) (+ x x))
(define (dbl x) (* 2 x))
```

- These are equivalent to each other

So macros for doubling are bad style but instructive examples:

```
(define-syntax dbl (syntax-rules()[(dbl x)(+ x x)]))
(define-syntax dbl (syntax-rules()[(dbl x)(* 2 x)]))
```

- These are not equivalent to each other. Consider:

```
(dbl (begin (print "hi") 42))
```

More examples

Sometimes a macro *should* re-evaluate an argument it is passed

- If not, as in **`dbl`**, then use a local binding as needed:

```
(define-syntax dbl
  (syntax-rules ()
     [(dbl x)
      (let ([y x]) (+ y y))]))
```

Also good style for macros not to have surprising evaluation order

- Good rule of thumb to preserve left-to-right
- Bad example (fix with a local binding):

```
(define-syntax take
  (syntax-rules (from)
     [(take e1 from e2)
      (- e2 e1)]))
```

Local variables in macros

In C/C++, defining local variables inside macros is unwise

- When needed done with hacks like **`__strange_name34`**

Here is why with a silly example:

- Macro:

```
(define-syntax dbl
  (syntax-rules ()
     [(dbl x) (let ([y 1])
                (* 2 x y))]))
```

- Use:

```
(let ([y 7]) (dbl y))
```

- Naïve expansion:

```
(let ([y 7]) (let ([y 1])
               (* 2 y y)))
```

- But instead Racket "gets it right," which is part of *hygiene*

The other side of hygiene

This also looks like it would do the “wrong” thing

– Macro:

```
(define-syntax dbl
  (syntax-rules ()
     [(dbl x) (* 2 x)]))
```

– Use:

```
(let ([* +]) (dbl 42))
```

– Naïve expansion:

```
(let ([* +]) (* 2 42))
```

– But again Racket’s *hygienic macros* get this right!

How hygienic macros work

A hygienic macro system:

1. Secretly renames local variables in macros with fresh names
2. Looks up variables used in macros where the macro is defined

Neither of these rules are followed by the “naïve expansion” most macro systems use

- Without hygiene, macros are much more brittle (non-modular)

On rare occasions, hygiene is not what you want

- Racket has somewhat complicated support for that

```
fun append (xs,ys) =
    if xs=[]
    then ys
    else (hd xs)::append(tl xs,ys)

fun map (f,xs) =
    case xs of
        [] => []
      | x::xs' => (f x)::(map(f,xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Optional: More Macro Examples

More examples

See the code for macros that:

- A for loop for executing a body a fixed number of times
  - Shows a macro that purposely re-evaluates some expressions and not others

- Allow 0, 1, or 2 local bindings with fewer parens than **`let*`**
  - Shows a macro with multiple cases

- A re-implementation of **`let*`** in terms of **`let`**
  - Shows a macro taking any number of arguments
  - Shows a recursive macro