Happy Friday!

CSCI 275: Programming Abstractions

Lecture 03: Basic Building Blocks

Spring 2025

Stephen Checkoway, Oberlin College
Slides gratefully borrowed from Molly Q Feldman

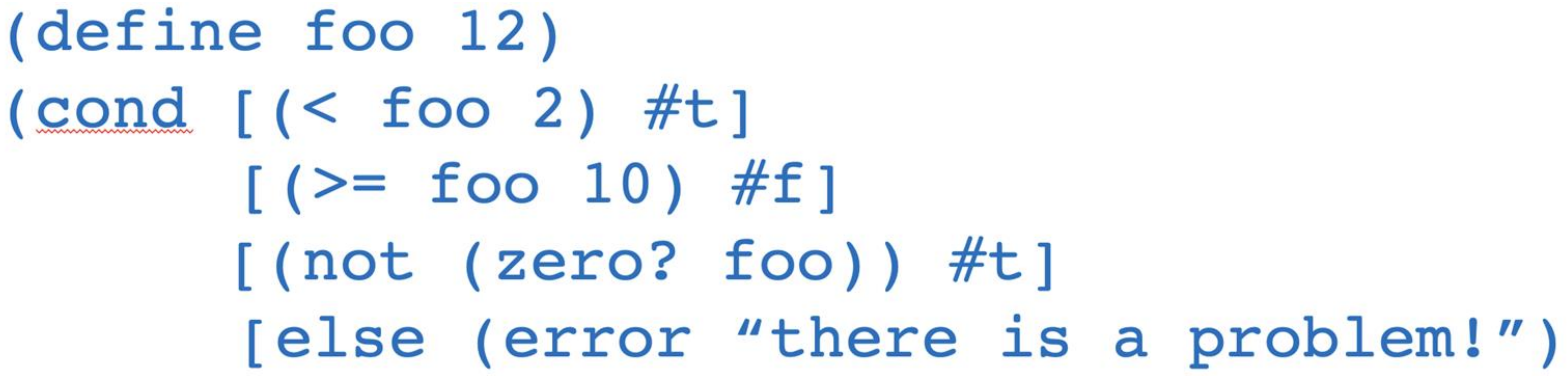
Announcements

- HW0 due date changed to Sunday

Goals for Today

- Procedures
- Introducing our core data type: lists
  - How we construct them
  - How we reference elements of them
  - Recursion with lists

Some questions



```
(define foo 12)
(cond [(< foo 2) #t]
      [(>= foo 10) #f]
      [(not (zero? foo)) #t]
      [else (error “there is a problem!”)
```

1. How can I get the `cond` to take an argument, rather than just reference a “global” `foo`?

2. How do I “save” code like that above to be able to reuse it? (i.e. a function!)

3. How is/isn’t this related to using `define` to bind identifiers?

Creating procedures: `lambda`

Procedures are creating using the `lambda` special form

```
(lambda parameters body …)
```

`parameters` is an unevaluated list of identifiers which will be bound to the values of the procedure's arguments when procedure is called

`body` is a sequence of s-expressions that form the body of the procedure, they're evaluated in turn

```
(lambda (x y)
  (/ (+ x y) 2))
(lambda (name)
  (displayln "Hello ")
  (displayln name))
```

Calling `lambdas`

Given we have a lambda, we can use it and call it

```
((lambda (x) (+ x 2)) 4)
```

This will evaluate to 6. However, this current structure doesn't allow us to *reuse* the lambda with a different input.

We already have a way to bind a value to an identifier ("name"): that's `define`.

We know `define` attaches a name to an evaluated value

`(define x (+ 20 100))` means `x` is bound to 120

So what does a `lambda` evaluate to? Anything?

BIG IMPORTANT SLIDE

Unlike procedures in most languages, in Racket there is a notion that `lambdas` are values & so can be evaluated

- `lambdas` are like numbers, strings, lists, etc.
- We can pass them around, return them, hold them as their own, evaluated concept
  - This is **really not true** in languages like C, for instance
  - This makes procedures **first-class** in Racket
- Support for higher-order/first-class functions is one of the hallmarks of a language that supports **functional programming**

`define` + `lambda` = reusable procedures!

We can combine `define` and `lambda`, so that we can get a named procedure!

```
(define add-two
  (lambda (x)
    (+ x 2)))
```

To call it, we then use prefix call notation, as usual:

`(add-two 2)` will give us 4

```
(define lily
  (lambda (x y)
      (string-append y x)))

(lily “hello” ”?”)
```

What does this code evaluate to?

A. Error

B. “hello?”

C. “?hello”

D. “hello ?”

E. Something else

```
(define alright
  (lambda (a b)
    (cond  [(equal? a b) “equal”]
           [(positive? a) 17]
           [(and (positive? a) (negative? b)) 5]
           [else “chaos!”])))
```

What does calling `(alright 10 -30)` evaluate to?

A. ”chaos”
B. Error
C. 5
D. 17
E. ”equal”

Can we use identifiers in lambdas? Sure!

Note: you won’t see for loops in this class – recursion all the way

Computing factorial in Racket:

```
(define fact
   (lambda (num)
      (if (<= num 1)
           1
           (* num (fact (- num 1))))))
```

What have we learned thus far?

- How to call procedures
- Predicates
- `if`
- `cond`
- `define`
- `lambda`
- `define & lambda` together!
- Recursion with numbers

Lists as the core data structure

- Lists (Arrays) are a pretty core data structure in most languages
- They also are helpful for practicing more recursion!
- For historic, Scheme reasons, lists are fundamental to Racket

Lists

They are what we will use / interact with / explore the most because of this

Lists are the ***most important*** data type in Racket

A list is one of two things:

- The empty list
- A pair `(x . y)` where `x` is an expression and `y` is a list

This is a **recursive** type definition: a type defined in terms of itself!

We will see this idea again when we talk about types!

We'll discuss pairs in more detail shortly

Constructing Lists

There is a built-in procedure called `list` which helps us create lists

`(list 1 3 5 2)` produces the list `'(1 3 5 2)`

`(list #t 5 "foo")` produces the list `'(#t 5 "foo")`

`(list (* 2 3) (and #t #f) 8)` produces `'(6 #f 8)`

1. Note that lists in Racket can be *heterogenous* types
2. Note that with the list procedure, it evaluates the contents passed it!

The empty list

There are three ways to write the empty list, we can pretty much* use them interchangeably.

- `null`
- `empty`
- `'()` — We'll see why this has a leading `'` soon

When working with lists, I recommend using `empty`

Accessing Elements of Lists: Racket

Racket helpfully gives us procedures which can access elements at specific indices in the list

```
(first '(a b c)) => a
(rest '(a b c)) => '(b c)
```

Note rest and second do **not return the same type:** rest returns a list, second returns an element

```
(second '(a b c)) => b
(third '(a b c)) => c
fourth, fifth, sixth, seventh, eighth, ninth, tenth,
last, etc.
```

What does this procedure do?

```
(define foo
   (lambda (lst)
      (cond [(empty? lst) #t]
            [(zero? (first lst)) #f]
            [else (foo (rest lst))])))
```

A. Returns `#t` if `lst` is empty and `#f` otherwise

B. Returns `#t` if `lst` contains a `0` and `#f` otherwise

C. Returns `#f` if `lst` contains a `0` and `#t` otherwise

D. Runs forever because `foo` is called on the rest of `lst`

Recursion with lists

Basic structure

```
(define process-list
  (lambda (lst)
    (cond [(empty? lst) EMPTY-LIST-BASE-CASE]
          [(___ (first lst)) OPTIONAL-BASE-CASE]
          [else (____ (first lst) (process-list (rest lst)))])))
```

Notice

- Use of **`first`** and **`rest`** to access the elements of the list
- The base cases and the recursive case

Creating a list from an existing list: cons

If we have a list like `(list 1 2 3)`, we can add an element to the beginning of the list using `cons`

```
(define languages
  (list "Python" "Java" "Rust"))

(cons "Racket" languages)
```

This returns the list `'("Racket" "Python" "Java" "Rust")`

Return a list containing all nonzero numbers

Problem: Write a function that takes a list of numbers as an argument and returns a list containing the nonzero numbers

Approach: Recursion on the argument list `lst`

1. If `lst` is empty, return the base case [what is the base case?]
2. If the **first** element of `lst` is 0, recurse on the **rest** of `lst`
3. Otherwise the first element is not 0 so return a list consisting of the first element of `lst` and the result of recursing on the rest of `lst`

Nonzero elements of the list

```
(define nonzeros
  (lambda (lst)
    (cond [(empty? lst) empty]
          [(= (first lst) 0) (nonzeros (rest lst))]
          [else (cons (first lst) (nonzeros (rest lst)))])))
```

Notice

- Only one base case this time: when the list is empty
- Two recursive cases: one recursion when the first element of the list is 0 and one for when it’s nonzero
- Using cons to prepend the first element of `lst` to the result of the recursive call

```
> (nonzeros (list -3 2 0 5 0 .1 0))
'(-3 2 5 0.1)
```

Two (Deeper) Questions

1. While we can construct lists with `list`, they print out with a quotation mark. Why?

2. We said that lists were pairs `(x . y)` where `x` is an expression and `y` is a list. What is a pair?

Two (Deeper) Questions

1. While we can construct lists with `list`, they print out with a quotation mark. Why?

2. We said that lists were pairs `(x . y)` where `x` is an expression and `y` is a list. What is a pair?

Quoting in Racket

Placing a `'` before an s-expression "quotes" it

- The quoted expression is treated as ***data***, *not code*
- DrRacket displays lists with the quote

Quoting in Racket

Placing a `'` before an s-expression "quotes" it

- The quoted expression is treated as ***data***, *not code*
- DrRacket displays lists with the quote

`'(1 4 5)` is a 3-element list

We saw `(list (* 2 3) (and #t #f) 8)` produces

```
'(6 #f 8)
```

`'((* 2 3) (and #t #f) 8)` produces

```
'((* 2 3) (and #t #f) 8)
```

Quoting, in general, is how we represent data

Quoting a number, boolean, or string returns that number, boolean, or string

- `'35` gives `35`
- `'#t` gives `#t`
- `'"Hello!"` gives `"Hello!"`

Quoting a variable gives a symbol

- `+` and `string-append` are variables whose values are procedures
- `'+` and `'string-append` are symbols

Quoting a list gives a list of quoted elements

- `'(1 2 x y)` is the same as `(list '1 '2 'x 'y)`
- `'(() (1) (1 2 3))` is the same as `(list '() '(1) '(1 2 3))`

Guidelines for creating lists

If you want to evaluate some expressions and have the resulting values be in the list, use `(list expr1 expr2 ... exprn)`

Example: `(list x (list x y z) z)`

If you want to create a list of literal numbers/strings/booleans/symbols, use `'(...)`

Example: `'(10 15 20 -3)`

Given variables `x` and `y`, how do we create a list containing the values of `x`, `y`, and `x + y`?

i.e., if `x` is 10 and `y` is 15, the list we want is `'(10 15 25)`.

A. `(list x y (+ x y))`

B. `(list 'x 'y (+ 'x 'y))`

C. `(list 'x 'y '(+ x y))`

D. `'(x y (+ x y))`

E. All of the above

Two (Deeper) Questions

1. ~~While we can construct lists with list, they print out with a quotation mark. Why?~~

1. We said that lists were pairs `(x . y)` where `x` is an expression and `y` is a list. What is a pair?

Pair are the (traditional) data structure in Scheme

Pairs hold data. To create a pair you use the `cons` procedure, which takes two arguments: `(cons a b)`

Top Tip: If you evaluate a term and it prints with a `.` in the middle (i.e. `′(2 . 3)`) that is a *pair* not a *list*

`cons` means “create a pair”

- `(cons 'x 'y)` creates the pair `'(x . y)`
- `(cons 2 3)` creates the pair `'(2 . 3)`
- `(cons 5 null)` creates the list `'(5)`

Lists are simply (useful) special cases of pairs –
All operators for pairs also work with lists, but not vice versa

`cons` helps us build up lists, one-by-one

If we have a list `lst` and an element `x`, prepend `x` to `lst`: `(cons x lst)`

```
(cons "c" (list "a" "b")) => '("c" "a" "b")
```

This works because the second argument to `cons` is a list so the result is a list

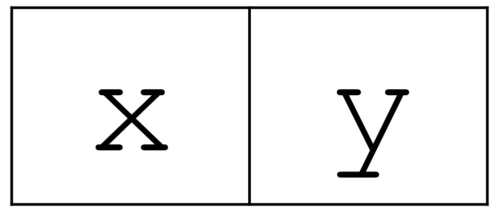
What if we want to append `x` to `lst`? Can we use `(cons lst x)`?

Will `(cons '(1 2 3) 4)` produce `'(1 2 3 4)`?

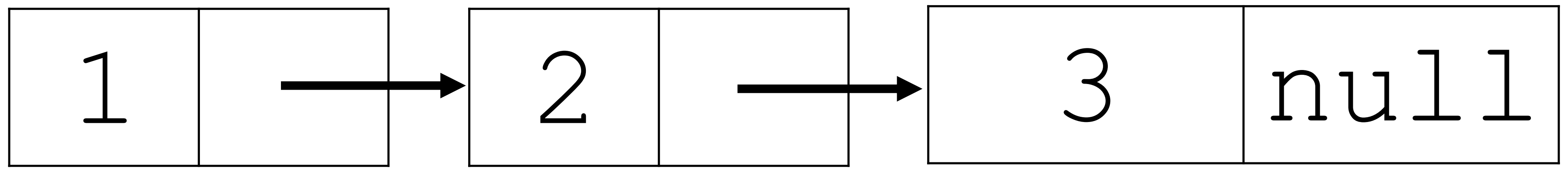
A. Yes
B. No

Cons cells

`(cons x y)` creates a *cons-cell*


x y

`(cons 1 (cons 2 (cons 3 null)))` produces


1 2 3 null

You'll notice that this is a linked list!

This is the same list that's produced by `(list 1 2 3)`

Get the first element from a pair

`car` (Contents of the Address part of a Register*)

Returns the first element of a pair (or the head of a list)

`(car (cons 5 8))` (equivalently `(car '(5 . 8))`) returns `5`

`(car '(1 2 3 4))` returns `1`

`(car (1 2 3 4))` is an error because `(1 2 3 4)` is invalid

* This terminology comes from the IBM 704, an ancient computer

Get the second element of the pair

`cdr` (Contents of the Decrement part of a Register*)

Returns the second element of a pair (or the tail of a list); pronounced "could-er"

`(cdr (cons 5 8))` (equivalently `(cdr '(5 . 8))`) returns 8

`(cdr '(1 2 3 4))` returns the list `'(2 3 4)`

`(cdr '(5))` returns the empty list, DrRacket will display `'()`

Note: `cdr` is equivalent to `rest`, **not second** in Racket terminology

* This terminology comes from the IBM 704, an ancient computer

`car` returns the first element of a pair
`cdr` returns the second element of a pair

If `lst` is a list, how do we get the second element of `lst`?

E.g., if `lst` is `'(2 3 5 7)`, the code should return `3`

A. `(car lst)`

B. `(cdr lst)`

C. `(car (cdr lst))`

D. `(cdr (car lst))`

E. `(cdr (cdr lst))`

Next Up!

See the schedule for readings.

Homework 0 is due on Sunday