CSSE 304 Day 4

Tail-recursive factorial

Anonymous procedures

box-and-pointer diagrams

map and apply

More recursion practice

(preview of next time? lambda and let)

Go for Simple!

- Some students wrote
  - (define first (lambda (x) (car x)))
- Simpler:
  - (define first car)

fact example 1

```
> (define fact
    (lambda (n)
      (cond
        [(zero? n)  1]
        [else (* n (fact (- n 1)))])))
> (fact 4)
24
> (fact  -2)
  C-c C-c
break>q
```

Escape from infinite loop by repeatedly pressing ctrl-c

```
> (trace fact fact2 fact-acc)
(fact fact2 fact-acc)
> (fact 4)
|(fact 4)
| (fact 3)
| |(fact 2)
| | (fact 1)
| | |(fact 0)
| | |1
| | 1
| |2
| 6
|24
24
```

Fact example 2

```
> (define fact2
    (lambda (n)
      (if (or (negative? n)
              (not (integer? n)))
          "error"
          (fact-acc n 1))))

> (define fact-acc
   (lambda (n acc)
     (if (zero? n)
         acc
         (fact-acc (- n 1)
                   (* n acc)))))
```

```
> (trace fact fact2 fact-acc)
(fact fact2 fact-acc)
> (fact2 4)
|(fact2 4)
|(fact-acc 4 1)
|(fact-acc 3 4)
|(fact-acc 2 12)
|(fact-acc 1 24)
|(fact-acc 0 24)
|24
24
```

Make-adder example

```
> (define make-adder
   (lambda (m)
    (lambda (n)
      (+ m n))))
>
```

Make-adder example

```
> (define make-adder
   (lambda (m)
    (lambda (n)
      (+ m n))))
> (define add5 (make-adder 5))
> add5
#<procedure>
> (add5 8)
13
```

Make-adder example

```
> (define make-adder
    (lambda (m)
     (lambda (n)
      (+ m n))))
> (define add5 (make-adder 5))
> add5
#<procedure>
> (add5 8)
13
> ((make-adder 5) 8)
13
```

Make-adder example

```
> (define make-adder
   (lambda (m)
    (lambda (n)
      (+ m n))))
> (define add5 (make-adder 5))
> add5
#<procedure>
> (add5 8)
13
> ((make-adder 5) 8)
13
> (((lambda (m)
     (lambda (n)
      (+ m n)))
     5)
   8)
13
```

Cond

- Similar to **if-elif-...-else** in other languages.

```
(define member?
    (lambda (a ls)
      (cond
        [(null? ls) #f]
        [(eq? (car ls) a) #t]
        [else (member? a (cdr ls))])
     ))
```

This example is in the on-line slides, but we won't do it in class, since we did a slightly simpler version in class on day 2

```
; cond is like if ... else if ... else

(define largest-in-list
  (lambda (L)
    (cond [(null? L)
           (errorf 'largest-in-list
                   "empty list has ~s "
                   " no largest element")]
          [(null? (cdr L)) (car L)]
          [else (max (car L)
                     (largest-in-list
                       (cdr L)))])))

; What's the efficiency issue with this?
```

- Answer: We have to do two null? tests for every recursive call.

```
; more efficient:
(define largest-in-list
  (lambda (ls)
    (if (null? ls)
      (errorf 'largest-in-list
              "list cannot be empty")
      (largest-in-non-empty ls))))


(define largest-in-non-empty
  (lambda (ls)
    (if (null? (cdr ls))
      (car ls)
      (let ([largest-in-cdr
             (largest-in-non-empty (cdr ls))])
        (if (> (car ls) largest-in-cdr)
            (car ls)
            largest-in-cdr)))))
```

Using max is simpler, but this is how we could do it if we did not have or did not remember max.

```
; Now define another version with an accumulator
;    (that is also more robust because it checks for non-numbers)

(define largest-in-list
  (lambda (ls)
    (if (null? ls)
       (errorf 'largest-in-list "list cannot be empty")
       (largest-in-list-acc (cdr ls) (car ls)))))


(define largest-in-list-acc
  (lambda (ls largest-so-far)
    (cond [(null? ls) largest-so-far]
          [(not (number? (car ls)))
           (errorf 'largest-in-list
                   "everything in the list must be a number")]
          [(> (car ls) largest-so-far)
           (largest-in-list-acc (cdr ls) (car ls))]
          [else (largest-in-list-acc (cdr ls)
                                     largest-so-far)])))
```

Count reflexive pairs

- A **relation** is a set of ordered pairs; the set of all first elements is the **domain**. The set of all second elements is the range.
- We represent a relation by a list of 2-lists. A 2-list is a list whose length is 2.
- A **reflexive pair** is a 2-list whose first and last elements are the same.
- Count-reflexive-pairs (work it out live)

Probably won't do this in class, but good practice for you

cons vs. list vs. append

- **box-and-pointer diagrams**
- `(define x '(1 2 3))`
- `(define y '(4 5))`
- `(define z '(6 7))`
- `(cons x y)`

`(list x y)`

`(append x y z)`

apply

What if a procedure expects a number of individual arguments, but we actually have the things that should be its arguments in a list?

More on map and apply soon

We'd like to write

```
(define list-sum (lambda (L) (+ L)))
```

but + doesn't expect a list of arguments.

So we write

```
define list-sum (lambda (L) (apply + L)))
```

Application of `apply` **is like consing** `apply`**'s first argument onto the list that is its second argument, and then evaluating.**

Recursive procedures

- **(make-list n obj)** returns a list of **n** "copies" of **obj**. [If **obj** is a 'by-reference" object, such as a list, it makes **n** copies of the reference].
- **(firsts '((a b) (c d) (e f)))**
  ➔ (a c e)
  - Do it "from scratch".
- **(map-unary f ls)** applies **f** to each element of **ls**, and returns the list of the results.
  - (map-unary (lambda (x) (+ x 2)) '(3 5 9)) ➔ (5 7 11)

map-unary is a special case of built-in procedure **map**.

- How could we use **map** to write **firsts**?

More recursive procedures

- **positives**
  - (positives '(1 -3 6 0 2 -1 7))➔ (1 6 2 7)
  - Write and use **filter-in**
- **sorted?**
  - (sorted? <= '(3 4 2 6)) ➔ #f
  - (sorted? >= '(4 3 2 1)) ➔ #t
- We'll be lucky if we get this far, but, ever the optimist, I included more slides. They are probably a preview of something we'll do next time.

lambda with an improper list of arguments

- Used when procedure expects a variable number of arguments.
  - **(lambda x *body*)**
    - when the resulting procedure is applied, all of the arguments are placed into a list and bound to x.
      Then **body** is evaluated.
  - **(lambda (x y . z) *body*)**
    - when the resulting procedure is applied, the first two arguments are bound to x and y,
    - any remaining arguments are placed into a list and bound to z. Then **body** is evaluated.

Q1

Procedures with an unknown number of arguments

```
> (define count-my-args
    (lambda L
      (length L)))
> (count-my-args 1 1 2 2 3 3 4 5)
8
```

```
> (define two-fixed-args-and-more
   (lambda (x y . z)
     (+ x y (apply + z))))
> (two-fixed-args-and-more 2 3 4 5)
14
> (two-fixed-args-and-more 2)
Exception: incorrect number of arguments to #<p
rocedure>
```

lambda the magnificent review and summary

- **Lambda** is the “function-maker”. **define** is the “variable-assigner”. There is no special connection between the two:

```
> ((lambda (x y) (+ x (* 2 y))) 3 5)
13
```

  - We can store procedures in a data structure without naming them:

```
> (define p2 (list (lambda (x) (* 2 x))
                   (lambda (y) (+ 3 y))))
> p2
(#<procedure> #<procedure>)
> ((car p2) 4)
8
>
```

lambda the magnificent

- We can pass a procedure as an argument to another procedure:

```
> (list car cdr)
(#<procedure car> #<procedure cdr>)
```

lambda the magnificent

- We create a new procedure and return it.

```
(define make-adder          ; a procedure that takes a numeric argument
  (lambda (n)               ; and creates and returns a new procedure.
    (lambda (m)
      (+ m n))))
> (make-adder 3)
#<procedure>
> (define add3 (make-adder 3))
> (add3 4)
7
> ((make-adder 3) 4)
7
```

- Scheme is not the only language with first-class procedures …

A first-class data object

- Can be stored in a data structure
- Can be passed as an argument to a procedure
- Can be returned by a procedure
- In Scheme, **procedures** are first-class

Translation of **let**

```
(define L '(4 3 2))
(let ([first (car L)]
      [second (cadr L)])
  (list (+ first second) (- first second)))
```

The `let` expression is equivalent to

```
((lambda (first second)
   (list (+ first second) (- first second)))
 (car L)
 (cadr L))
```

let and **cond** are both examples of "syntactic sugar", as are **and** and **or**.

more on let

```
(define xxx
   (lambda (L)
     (let ([a  (car L)]
           [b  (cdr L)]
           [c  (car b)])
       (list c a))))
```

What goes wrong if we evaluate `(xxx '(1 2 3))`**?**

Translate the `let` **expression to an application of** `lambda`

let*

What we really wanted was

```
(define xxx
   (lambda (L)
     (let* ([a  (car L)]
            [b  (cdr L)]
            [c  (car b)])
       (list c a))))
```

which translates into

```
(define xxx
   (lambda (L)
     (let ([a (car L)])
       (let ([b  (cdr L)])
          (let ([c  (car b)])
              (list c a))))))
```