EECS 368
Programming Language Paradigms

Dr. Andy Gill

Department of Electrical Engineering & Computer Science
University of Kansas

October 29th, 2015

KU

Rules

- Always check for `null?` when recursing on a list
- Use `cons` to build the result list
- Share parts of the input list when constructing the output list

KU

doubling

```
> (doubleup '(1 2 3))
(list 1 1 2 2 3 3)
```

```
(define doubleup
 (lambda (l)
  (cond
   ((null? l) '())
   (else
      (cons (car l)
            (cons (car l)
                  (doubleup (cdr l)))))))
  )
 )
)
```

drop seconds

```
> (dropseconds '(1 2 3 4))
(list 1 3)
```

KU

```
(define dropseconds
 (lambda (l)
  (cond
    ((null? l) '())
    ((null? (cdr l)) (cons (car l) '()))
    (else (cons (car l)
                (dropseconds (cdr (cdr l))))))
  )
 )
)
```

append

```
> (myappend '(1 2 3 4) '(5 6 7 8))
(list 1 2 3 4 5 6 7 8)
```

KU

```
(define myappend
 (lambda (ls1 ls2)
  (cond
   ((null? ls1) ls2)
   (else (cons (car ls1)
               (myappend (cdr ls1) ls2)))
  )
 )
)
```

KU

length

```
> (length '(a b c))
3
> (length '())
0
```

```
(define length
 (lambda (l)
  (cond
    ((null? l) 0)
    (else (+ 1 (length (cdr l))))
   )
 )
)
```

tup

```
(1 2 3)                ;; yes

(3 4 (5 6) 8)          ;; no

(a b 3 2)              ;; no

()                     ;; yes
```

KU

addtup

```
> (addtup '(1 2 3))
6
> (addtup '())
0
```

KU

```
(define addtup
 (lambda (l)
  (cond
   ((null? l) 0)
   (else (+ (car l)
            (addtup (cdr l))))
  )
 )
)
```

Remove a Member (rember)

```
(define rember
   (lambda (a lat)
      (cond
       ((null? lat) '())
       (else (cond
                ((eq? (car lat) a) (cdr lat))
                (else (cons
                           (car lat)
                           (rember a
                             (cdr lat)))))))))
```

- We are going to fix `rember` to remove **all** the instances of an element

- Learn two new patterns:
  - Recursion over numbers
  - Recursion over trees

KU

Remove a Member (rember)

```
(define rember
 (lambda (a lat)
  (cond
   ((null? lat) '())
   (else (cond
           ((eq? (car lat) a) (cdr lat))
           (else (cons (car lat)
                       (rember a
                         (cdr lat)))))))))
```

KU

The Three Laws of Software Engineering

When writing a function;

- Make it **Correct**; then
- Make it **Clear**; then
- Make it **Fast** (optional).

KU

```
(define rember ;; buggy function
 (lambda (a lat)
  (cond
   ((null? lat) '())
   (else (cond
           ((eq? (car lat) a) (cdr lat))
           (else (cons (car lat)
                       (rember a
                         (cdr lat)))))))))
```

```
> (rember 'a '(a b c d))
(list b c d)
> (rember 'a '(a b c d a))
(list b c d a)
```

KU

```
(define rember
 (lambda (a lat)
  (cond
   ((null? lat) '())
   (else (cond
           ((eq? (car lat) a)
                              ;; also use recursion
                    (rember a (cdr lat)))

           (else (cons (car lat)
                       (rember a
                         (cdr lat)))))))))
```

```
> (rember 'a '(a b c d))
(list b c d)
> (rember 'a '(a b c d a))
(list b c d)
```

KU

Now it is correct, make is clean

```
(define rember
 (lambda (a lat)
  (cond
   ((null? lat) '())
   ((eq? (car lat) a) (rember a (cdr lat)))
   (else (cons (car lat)
               (rember a (cdr lat)))))))
```

KU

High School Arithmetic

Defining * in terms of +.

```
(define mymultiply
 (lambda (n m)
  (cond
   (...)
   (...)
  )
 )
)
```

High School Arithmetic

Defining * in terms of +.

```
(define mymultiply
 (lambda (n m)
  (cond
   ((zero? m) 0)
   (else (+ n (mymultiply n (- m 1)))))
  )
 )
)
```

KU

```
(define rember*
 (lambda (a l)
  (cond
     ...
  )
 )
)
```

```
> (rember* 'a '(a (b c) (a d) d))
(list (list b c) (list d) d)
```

All *-functions ask three questions, working on

- empty
- an atom **cons**ed onto a list, or
- a list **cons**ed onto a list.

```
(define rember*
 (lambda (a l)
  (cond
     ((null? l) '())
     ((atom? (car l))
        (cond
          ((eq? (car l) a) (rember* a (cdr l)))
          (else (cons (car l)
                      (rember* a (cdr l))))))
     (else (cons (rember* a (car l))
                 (rember* a (cdr l)))))
  )
 )
)
```

```
> (rember* 'a '(a (b c) (a d) d))
(list (list b c) (list d) d)
```

We already know the first six commandments!

The First Commandment

When recurring on a list of atoms, lat, ask two questions

- (null? lat)
- else

When recurring on a number, n, ask two questions

- (zero? n)
- else

When recurring on a list of S-expressions, l, ask three questions

- (null? l)
- (atom? (car l))
- else

KU

The Second Commandment

Use `cons` to build lists.

The Third Commandment

When building a list, describe the first typical element, and then `cons` it onto the natural recursion.

KU

The Fourth Commandment

Always change at least one argument while recurring.

...

It must be changed to be closer to termination.

The Fifth Commandment

When build a value using `+`, always use `0` for the terminating line.

When build a value using `*`, always use `1` for the terminating line.

When build a list using `cons`, consider using `'()` for the terminating line.

The Sixth Commandment

Simplify only after the function is correct.

Next class: ‘(quoting)