EECS 368 Programming Language Paradigms

Dr. Andy Gill

Department of Electrical Engineering & Computer Science University of Kansas

October 22, 2015



cons command

```
(define ls1 '(a b c))
(define ls2 '((a b c) x y z))
(define ls3 'hotdog)
(define ls4 '())
(define ls5 '(((hotdogs)) (and) (pickle) relish))
> (cons ls3 ls1)
(list 'hotdog 'a 'b 'c)
> (cons ls1 ls1)
(list (list 'a 'b 'c) 'a 'b 'c)
> (cons ls4 ls1)
(list '() 'a 'b 'c)
> (cons ls4 ls3)
???
                                   ◆□→ ◆□→ ◆□→ ◆□→ □
```

2/60

The Law of Cons

The primitive cons take two arguments, and the second argument must be a list, and the result is a list.

null? command

```
(define ls1 '(a b c))
(define ls2 '((a b c) x y z))
(define ls3 'hotdog)
(define ls4 '())
> (null? ls1)
false
> (null? ls2)
false
> (null? ls3)
```

true (or #t) only for ().

false

true

> (null? 1s4)



eq? command

```
(define ls1 '(a b c))
(define ls2 '((a b c) x y z))
(define ls3 'hotdog)
(define ls4 '())
```

```
> (eq? ls1 ls1)
true
> (eq? '(a b c) '(a b c))
false
```

The lists have to be the same list, not just look the same, for eq? to return true

eq? command

```
(define ls1 '(a b c))
(define ls2 '((a b c) x y z))
(define ls3 'hotdog)
(define ls4 '())
```

```
> (eq? ls1 ls1)
#t
> (eq? '(a b c) '(a b c))
#f
```

The lists have to be the same list, not just look the same, for eq? to return true

equal? command

```
(define ls1 '(a b c))
(define ls2 ls1)
> (equal? ls1 ls1)
#t
> (equal? '(a b c) '(a b c))
#t.
> (equal? ls1 ls2)
#t
equal? does a deep comparison, eq? does pointer
equality.
```

pair? command

```
(define ls1 '(a b c))

> (pair? ls1)
#t
> (pair? '(x y z))
#t
> (pair? 'ls3)
#f
```

Review

- Expressions built from lists and atoms
 - Lists are expressions inside (...)
 - Atoms are names or labels, like turkey or 22.
- Scheme also supports Strings
- Expressions can be quoted, using '.
- Built in functions: car, cdr, cons, null?, eq?, equal?, and pair?.

Writing the atom? command

```
(define atom?
  (lambda (x)
    (and (not (pair? x)) (not (null? x)))))
(define bla 'thing)
(define ls1 '(Jack Sprat chicken))
(define 1s2 '(Jack (Sprat) chicken))
(define ls3 '())
```

```
We can now use atom?.
> (atom? bla)
#t
> (atom? ls1)
#f
> (atom? 1s2)
#f
> (atom? 1s3)
#f
```

The lat? (list-of-atoms) command

> (lat? ls1)

```
(define lat? ...)
(define bla 'thing)
(define ls1 '(Jack Sprat chicken))
(define ls2 '(Jack (Sprat) chicken))
(define ls3 '())
```

```
#t
> (lat? ls2)
#f
> (lat? ls3)
#t
> (lat? bla)
car: expects argument of type <pair>; given thing
```

The lat? (list-of-atoms) command

define

```
(define lat? ...)
```



lambda

```
(define lat?
  (lambda (l) ...))
```



cond

```
(define lat?
  (lambda (l)
    (cond
     ((\ldots))
     ((\ldots)\ldots)
     (else ...)
    )))
```



```
(define lat?
  (lambda (l)
    (cond
     ((null? 1) ...)
     ((atom? (car 1)) ...)
     (else ...)
    )))
```



The lat? (list-of-atoms) command

The lat? command in Java (while loop)

```
public Expression isLat(Expression 1) {
   while (l != null) {
     if (!isAtom(car(1))) {
        return false;
     1 = cdr(1);
   return true;
```

The lat? command in Java (recursion)

```
public Expression isLat(Expression 1) {
   if (1 == null) {
      return true;
   if (isAtom(car(1))) {
      return isLat(cdr(1));
   return false;
```

```
package Scheme;
abstract public class Exp {
    abstract public boolean isAtom();
    abstract public boolean isPair();
    public static boolean isAtom(Exp e) {
      return e.isAtom();
    }
    public static boolean isPair(Exp e) {
      return e.isPair();
    public static Exp cons(Exp h,Exp t) {
      return new Pair(h,t);
    . . .
```

```
package Scheme;
public class Atom extends Exp {
    private String name;
    public Atom(String name) {
        this.name = name;
    public boolean isAtom() { return true; }
    public boolean isPair() { return false; }
    public String toString() {
        return name;
```

```
package Scheme;
public class Pair extends Exp {
    private Exp h;
    private Exp t;
    public Pair(Exp h,Exp t) {
        this.h = h:
        this.t = t:
    }
    public boolean isAtom() { return false; }
    public boolean isPair() { return true; }
    public Exp car() { return h; }
    public Exp cdr() { return t; }
    . . .
```

```
abstract public class Exp {
    public static Exp car(Exp e) {
        if (e instanceof Pair) {
            return ((Pair)e).car();
        throw new RuntimeException("car(..) on a non Pair");
    public static Exp cdr(Exp e) {
        if (e instanceof Pair) {
            return ((Pair)e).cdr();
        }
        throw new RuntimeException("cdr(..) on a non Pair");
```

```
public class Pair extends Exp {
    public String toString() {
        String msg = "(";
        Pair r = this;
        while (r != null) {
            msg = msg + " " + r.car().toString();
            r = (Pair) r.cdr();
        return msg + " " + ")";
```

```
public class Main {
    public static void main(String[] args) {
        Exp e1 = Exp.cons(new Atom("1"),
                          Exp.cons(new Atom("2").
                                   null)):
        System.out.println("e1 = " + e1.toString());
        Exp e2 = Exp.cons(e1,e1);
        System.out.println("e2 = " + e2.toString());
        Exp e3 = Exp.cons(e1,e2);
        System.out.println("e3 = " + e3.toString());
        Exp e4 = Exp.cdr(e3);
        System.out.println("e4 = " + e4.toString());
        System.out.println("isLat(e4) = " + isLat(e4));
        System.out.println("isLat(e1) = " + isLat(e1));
    }
```

```
public static boolean isLat(Exp 1) {
    if (1 == null) {
        return true;
    }
    if (Exp.isAtom(Exp.car(1))) {
        return isLat(Exp.cdr(1));
    }
    return false;
}
```

Back to Scheme

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 will be returning to rember later.



#lang racket

in file, to enable racket.



Review

- Expressions built from lists and atoms
 - Lists are expressions inside (...)
 - Atoms are names or labels, like turkey or 22.
- Scheme also supports Strings
- Expressions can be quoted, using '.
- Built in functions: car, cdr, cons, null?, eq?, equal?, and pair?.

The Second Commandment Use cons to build lists.





What does cons do?

```
'(a b c)
(cons 'a (cons 'b (cons 'c '())))
```



A Silly Example

```
(define silly
  (lambda (e)
        (cons (car e) (cdr e))))
```

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 will be returning to rember later.



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

Applying the Third Commandment

• What are the functions we have seen so far?

• What are the functions we have seen so far?

```
car cdr cons null? eq? equal? atom? cons?
```

- What are the functions we have seen so far?car cdr cons null? eq? equal? atom? cons?
- What keywords we have seen?

- What are the functions we have seen so far?car cdr cons null? eq? equal? atom? cons?
- What keywords we have seen?define lambda cond

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



doubling

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



```
(define doubleup
 (lambda (l)
  (cond
   ((null? 1) '())
   (else
      (cons (car 1)
             (cons (car 1)
                   (doubleup (cdr 1)))))
```



drop seconds

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





```
(define dropseconds
 (lambda (l)
  (cond
    ((null? 1) '())
    ((null? (cdr 1)) (cons (car 1) '()))
    (else (cons (car 1)
                (dropseconds (cdr (cdr 1)))))
```



append

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



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



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



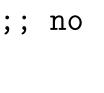
```
(define length
 (lambda (l)
  (cond
    ((null? 1) 0)
    (else (+ 1 (length (cdr 1))))
```



(a b 3 2)

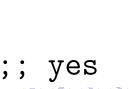
```
(1 \ 2 \ 3)
```

(3 4 (5 6) 8)



;; yes

;; no





addtup

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



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

