Lecture 6: Programming with Data

CS150: Computer Science
University of Virginia
Computer Science

David Evans
http://www.cs.virginia.edu/evans

Ways to Design Programs

1. Think about what you want to **do**, and turn that into code.
2. Think about what you need to **represent**, and design your code around that.

Which is better?

History of Scheme

- Scheme [Guy Steele & Gerry Sussman, 1975]
  - Guy Steele co-designed Scheme and created the first Scheme interpreter for his 4th year project
  - More recently, Steele specified Java [1995]
  - "Conniver" [1973] and "Planner" [1967]
- Based on LISP [John McCarthy, 1958]
  - Based on Lambda Calculus
  - Alonzo Church, 1930s
  - Last few lectures in course

LISP

"**L**ots of **I**nsipid **S**illy **P**arentheses"

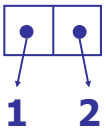
"**LIS**t **P**rocessing language"

Lists are pretty important – hard to write a useful Scheme program without them.

Making Lists

Making a Pair

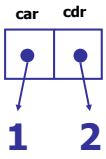
> (**cons** 1 2)
(1 . 2)



cons **cons**tructs a pair

Splitting a Pair

```
> (car (cons 1 2))
1
> (cdr (cons 1 2))
2
```



car extracts first part of a pair
cdr extracts second part of a pair

Why "car" and "cdr"?

- Original (1950s) LISP on IBM 704
  - Stored cons pairs in memory registers
  - **car** = "**C**ontents of the **A**ddress part of the **R**egister"
  - **cdr** = "**C**ontents of the **D**ecrement part of the **R**egister" ("could-er")
- Doesn't matter unless you have an IBM 704
- Think of them as **first** and **rest**

```
(define first car)
(define rest cdr)
```

(The DrScheme "Pretty Big" language already defines these, but they are not part of standard Scheme)

Implementing cons, car and cdr

```
(define (cons a b)
  (lambda (w) (if w a b)))

(define (car pair) (pair #t))
(define (cdr pair) (pair #f))
```

Scheme provides primitive implementations for cons, car, and cdr. But, we could define them ourselves.

Pairs are fine, but how do we make threesomes?

Triple

A triple is just a pair where one of the parts is a pair!

```
(define (triple a b c)
   (cons a (cons b c)))
(define (t-first t) (car t))
(define (t-second t) (car (cdr t)))
(define (t-third t) (cdr (cdr t)))
```


Quadruple

A quadruple is a pair where the second part is a triple

```
(define (quadruple a b c d)
   (cons a (triple b c d)))
(define (q-first q) (car q))
(define (q-second q) (t-first (cdr t)))
(define (q-third t) (t-second (cdr t)))
(define (q-fourth t) (t-third (cdr t)))
```

Multuples

- A quintuple is a pair where the second part is a quadruple
- A sextuple is a pair where the second part is a quintuple
- A septuple is a pair where the second part is a sextuple
- An octuple is group of octupi
- A ? is a pair where the second part is a …?

Lists

List ::= **(cons** *Element List***)**

A *list* is a pair where the second part is a *list*.

One big problem: how do we stop?
This only allows infinitely long lists!

Lists

List ::= **(cons** *Element List***)**

List ::=

It's hard to write this!

A *list* is either:
a pair where the second part is a *list*
or, empty

Null

List ::= **(cons** *Element List***)**

List ::= **null**

A *list* is either:
a pair where the second part is a *list*
or, empty (**null**)

List Examples

```
> null
()
> (cons 1 null)
(1)
> (list? null)
#t
> (list? (cons 1 2))
#f
> (list? (cons 1 null))
#t
```


More List Examples

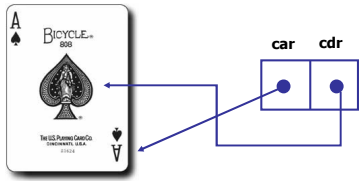
```
> (list? (cons 1 (cons 2 null)))
#t
> (car (cons 1 (cons 2 null)))
1
> (cdr (cons 1 (cons 2 null)))
(2)
```

Recap

- A *list* is either:
  a pair where the second part is a *list*
  or **null** (note: book uses **nil**)
- Pair primitives:
  (cons a b) Construct a pair <a, b>
  (car pair) First part of a pair
  (cdr pair) Second part of a pair

Problem Set 2: Programming with Data

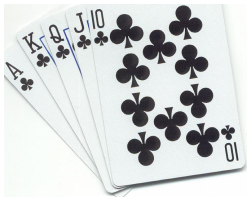
- Representing a card



Pair of rank (Ace) and suit (Spades)

Problem Set 2: Programming with Data

- Representing a card: (cons <rank> <suit>)
- Representing a hand



(list (make-card Ace clubs)
(make-card King clubs)
(make-card Queen clubs)
(make-card Jack clubs)
(make-card 10 clubs)

length

- Define a procedure that takes as input a list, and produces as output the length of that list.

(length null) ➔ 0
(length (list 1 2 3)) ➔ 3
(length (list 1 (list 2 3 4))) ➔ 2

Charge

- Its okay if you are confused now.
- Lots of opportunities to get unconfused:
  - Problem Set 2 (and PS3 and PS4)
    - Lab hours Sunday, Tuesday, Wednesday, and Thursday
  - Read the Course Book
  - Class Wednesday and Friday – lots of examples programming with procedures and recursive definitions
  - Office Hours (Wednesdays and Thursdays)