Lecture 6: Programming with Data

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
“Lots of Insipid Silly Parentheses”
“LIST Processing 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 constructs a pair
### Splitting a Pair

\[
\begin{align*}
> (\textbf{car} \ (\textbf{cons} \ 1 \ 2)) & \quad \text{car} \ \text{cdr} \\
1 & \\
> (\textbf{cdr} \ (\textbf{cons} \ 1 \ 2)) & \quad \text{1} \ \text{2} \\
2 & \\
\end{align*}
\]

- **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** = “Contents of the **Address** part of the **Register**”
  - **cdr** = “Contents of the **Decrement** part of the **Register**” (“could-er”)
- Doesn’t matter unless you have an IBM 704
- Think of them as **first** and **rest**

\[
\begin{align*}
&\text{(define first car)} \\
&\text{(define rest cdr)} \\
&\text{(define first car)} \\
&\text{(define rest cdr)} \\
\end{align*}
\]

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

### Implementing cons, car and cdr

\[
\begin{align*}
(\text{define} \ (\text{cons} \ a \ b) & \quad (\lambda \ (w) \ (\text{if} \ w \ a \ b))) \\
(\text{define} \ (\text{car} \ \text{pair}) \ (\text{pair} \ #t) & \\
(\text{define} \ (\text{cdr} \ \text{pair}) \ (\text{pair} \ #f) & \\
\end{align*}
\]

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!

\[
\begin{align*}
&\text{(define (triple a b c)} \\
&(\text{cons} \ a \ (\text{cons} \ b \ c))) \\
&(\text{define} \ (t\text{-first} \ t) \ (\text{car} \ t)) \\
&(\text{define} \ (t\text{-second} \ t) \ (\text{car} \ (\text{cdr} \ t))) \\
&(\text{define} \ (t\text{-third} \ t) \ (\text{cdr} \ (\text{cdr} \ t))) \\
\end{align*}
\]

### Quadruple

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

\[
\begin{align*}
&\text{(define (quadruple a b c d)} \\
&(\text{cons} \ a \ (\text{triple} \ b \ c \ d)) \\
&(\text{define} \ (q\text{-first} \ q) \ (\text{car} \ q)) \\
&(\text{define} \ (q\text{-second} \ q) \ (t\text{-first} \ (\text{cdr} \ t))) \\
&(\text{define} \ (q\text{-third} \ t) \ (t\text{-second} \ (\text{cdr} \ t))) \\
&(\text{define} \ (q\text{-fourth} \ t) \ (t\text{-third} \ (\text{cdr} \ t))) \\
\end{align*}
\]
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 a group of octopi
- 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 ::= (cons Element List)

List ::= (cons Element 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: (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) \(\Rightarrow\) 0
  (length (list 1 2 3)) \(\Rightarrow\) 3
  (length (list 1 (list 2 3 4))) \(\Rightarrow\) 2

Charge

• It's 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)