Lecture 4: Programming with Data

Eval Rule 3a: Application
(square 4)

Eval Rule 2: Names
Eval Rule 1: Primitive
(lambda (x) (* x x)) 4

Eval Rule 3b: Application
Apply Rule 2: Compound Application
(* 4 4)

Eval Rule 1: Primitive
Eval Rule 1: Primitive
#<primitive:*>

Eval Rule 3b: Application
Apply Rule 1: Primitive Application 16

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

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?

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

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

Pairs are fine, but how do we make threesomes?

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)))

(define first car)  
(define rest cdr)  
(The DrScheme “Pretty Big” language already defines these, but they are not part of standard Scheme)
Quadruple

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

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

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 group of octupi
- A ? is a pair where the second part is a ...

Lists

\[
\text{List ::= (cons Element List)}
\]

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!

Null

\[
\begin{align*}
\text{List ::= (cons Element List)} \\
\text{List ::= null}
\end{align*}
\]

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

List Examples

\[
\begin{align*}
> \text{null} \\
> (\text{cons 1 null}) \\
> (\text{list? null}) \\
> (\text{list? (cons 1 2))} \\
> (\text{list? (cons 1 null))}
\end{align*}
\]
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)

Charge

Please don’t leave until I take your picture!

• You know everything you need for PS2
• Friday, next week - lots of examples of:
  – Programming with data
  – Programming with procedures
  – Recursive definitions
• But, if you understand the Scheme evaluation rules, you know it all already!