Announcements

- Homework 7 due Wednesday 11/5 @ 11:59pm
- Project 1 composition revisions due Wednesday 11/5 @ 11:59pm
- Make changes to your project based on the composition feedback you received
- Earn back any points you lost on project 1 composition
- Composition of other projects is delayed, as we transition to new grading software
- Quiz 2 released Wednesday 11/5 & due Thursday 11/6 @ 11:59pm
- Open note, open interpreter, closed classmates, closed Internet
- CS 61A Flash mob Wednesday 3:03pm-3:09pm in Memorial Glade

Scheme is a Dialect of Lisp

What are people saying about Lisp?

- "The greatest single programming language ever designed."
  - Alan Kay, co-inventor of Smalltalk and OOP
- "The only computer language that is beautiful."
  - Neal Stephenson, DeNero's favorite sci-fi author
- "God's programming language."
  - Brian Harvey, Berkeley CS instructor extraordinaire

Scheme Fundamentals

Scheme programs consist of expressions, which can be:
- Primitives: 2, 3.3, true, +, quotient, ...
- Combinations: (quotient 10 2), (not true), ...

Numbers are self-evaluating; symbols are bound to values

Call expressions include an operator and 0 or more operands in parentheses

> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (define pi 3.14)
> (* pi 2)
6.28
> (define (abs x)
   (if (< x 0)
       (- x)
       x))
> (abs -3)
3

Special Forms

A combination that is not a call expression is a special form:
- if expression: (if <predicate> <consequent> <alternative>)
- and or: (and <e1> ... <en>), (or <e1> ... <en>)
- Binding symbols: (define <symbol> <expression>)
- New procedures: (define (<symbol> <formal parameters>) <body>)

Scheme Interpreters
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

\[ \lambda \text{ (formal-parameters) } \text{body} \]

Two equivalent expressions:

- \( \text{define (plus4 x) (+ x 4)} \)
- \( \text{define plus4 (lambda (x) (+ x 4))} \)

An operator can be a call expression too:

\[ ((\text{lambda (x y z) (+ x y (square z))) 1 2 3}) \]

Evaluates to the \( x+y+z^{2} \) procedure

Pairs and Lists

In the late 1950s, computer scientists used confusing names:

- \( \text{cons} \): Two-argument procedure that creates a pair
- \( \text{car} \): Procedure that returns the first element of a pair
- \( \text{cdr} \): Procedure that returns the second element of a pair
- \( \text{nil} \): The empty list

They also used a non-obvious notation for linked lists:

- A (linked) list in Scheme is a pair in which the second element is \text{nil} or a Scheme list.

Important! Scheme lists are written in parentheses separated by spaces

- A dotted list has any value for the second element of the last pair; maybe not a list!

\[
\text{> (define x (cons 1 2))}
\text{> x}
\text{(1 . 2)}
\text{> (car x)}
\text{1}
\text{> (cdr x)}
\text{2}
\text{> (cons 1 (cons 2 (cons 3 (cons 4 nil)))))}
\text{(Demo)}
\]

Symbolic Programming

Symbols normally refer to values; how do we refer to symbols?

- \( \text{> (define a 1)} \)
- \( \text{> (define b 2)} \)
- \( \text{> (list a b)} \)

Quotation is used to refer to symbols directly in Lisp.

- \( \text{> (car '(a b c))} \)
- \( \text{> (cdr 'a b c)} \)

Symbols are now values

Dots can be used in a quoted list to specify the second element of the final pair.

- \( \text{> (cdr (cdr '(1 . 3)))} \)
  \(3\)

However, dots appear in the output only of ill-formed lists.

- \( \text{> '(1 . 3)} \)
  \(1, 3\)
- \( \text{> '(1 . 3 . 4)} \)
  
- \( \text{> '(1 . 3 . nil)} \)
  \(1, 3\)

What is the printed result of evaluating this expression?

- \( \text{> (cdr '((1 . 2) . (3 . 4 . (5)))))} \)
  \(3, 4, 5\)

(Semicolon)