Announcements

- HW9 due Wednesday

- Ants extra credit due Wednesday
  - See Piazza for submission instructions

- Hog revisions out, due next Monday
Scheme Is a Dialect of 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, sci-fi author

“The most powerful programming language is Lisp. If you don't know Lisp (or its variant, Scheme), you don't appreciate what a powerful language is. Once you learn Lisp you will see what is missing in most other languages.”
- Richard Stallman, founder of the Free Software movement

http://imgs.xkcd.com/comics/lisp_cycles.png
Scheme Fundamentals

Scheme programs consist of expressions, which can be:

- **Primitive expressions:** 2, 3.3, true, +, quotient, ...
- **Combinations:** (quotient 10 2), (not true), ...

Numbers are self-evaluating; symbols are bound to values

Call expressions have an operator and 0 or more operands

> (quotient 10 2)
5
> (quotient (+ 8 7) 5)
3
> (+ (* 3
   (+ (* 2 4)
     (+ 3 5)))
   (+ (- 10 7)
     6))

“quotient” names Scheme’s built-in integer division procedure (i.e., function)

Combinations can span multiple lines (spacing doesn’t matter)
Special Forms

A combination that is not a call expression is a special form:

- **If** expression: \((\text{if } \langle\text{predicate}\rangle \ \langle\text{consequent}\rangle \ \langle\text{alternative}\rangle)\)
- **And** and **or**: \((\text{and } \langle\text{e}_1\rangle \ \ldots \ \langle\text{e}_n\rangle), (\text{or } \langle\text{e}_1\rangle \ \ldots \ \langle\text{e}_n\rangle)\)
- Binding names: \((\text{define } \langle\text{name}\rangle \ \langle\text{expression}\rangle)\)
- New procedures: \((\text{define } (\langle\text{name}\rangle \ \langle\text{formal parameters}\rangle) \ \langle\text{body}\rangle)\)

\[
\begin{align*}
> & \quad \text{(define } \text{pi } 3.14) \\
> & \quad (\ast \text{ pi } 2) \\
> & \quad 6.28
\end{align*}
\]

\[
\begin{align*}
> & \quad \text{(define } \text{abs } x) \\
& \quad (\text{if } (\langle x \ 0 \rangle) \\
& \quad \text{(- } x) \\
& \quad x)) \\
> & \quad (\text{abs } -3) \\
> & \quad 3
\end{align*}
\]
Lambda Expressions

Lambda expressions evaluate to anonymous procedures

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

Two equivalent expressions:

\[(\text{define} \ (\text{plus4} \ x) \ (+ \ x \ 4))\]

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

An operator can be a combination too:

\[(\text{evaluates to the} \ add-x-&-y-&-z^2 \ \text{procedure})\]
Pairs

We can implement pairs functionally:

\[
(\text{define } (\text{pair } x \ y) (\text{lambda } (m) (\text{if } (= m 0) x y)))
\]
\[
(\text{define } (\text{first } p) (p 0))
\]
\[
(\text{define } (\text{second } p) (p 1))
\]

Scheme also has built-in pairs that use weird names:

- **cons**: Two-argument procedure that creates a pair
- **car**: Procedure that returns the first element of a pair
- **cdr**: Procedure that returns the second element of a pair

A pair is represented by a dot between the elements, all in parens

\[
> (\text{cons } 1 \ 2)
\]
\[
(1 . \ 2)
\]
\[
> (\text{car } (\text{cons } 1 \ 2))
\]
\[
1
\]
\[
> (\text{cdr } (\text{cons } 1 \ 2))
\]
\[
2
\]
Recursive Lists

A recursive list can be represented as a pair in which the second element is a recursive list or the empty list.

Scheme lists are recursive lists:

- `nil` is the empty list.
- A non-empty Scheme list is a pair in which the second element is `nil` or a Scheme list.

Scheme lists are written as space-separated combinations.

```scheme
> (define x (cons 1 (cons 2 (cons 3 (cons 4 nil)))))
> x
(1 2 3 4)
> (cdr x)
(2 3 4)
> (cons 1 (cons 2 (cons 3 4)))
(1 2 3 . 4)
```

Not a well-formed list!
Symbols are normally evaluated to produce values; how do we refer to symbols?

> (define a 1)
> (define b 2)
> (list a b)
(1 2)

Quotation prevents something from being evaluated by Lisp

> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)

Quotation can also be applied to combinations to form lists

> (car '(a b c))
a
> (cdr '(a b c))
(b c)
Dots can be used in a quoted list to specify the second element of the final pair

\[ (\text{cdr (cdr } '(1 2 . 3))) \]
3

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

\[ '(1 2 . 3) \]
(1 2 . 3)

\[ '(1 2 . (3 4)) \]
(1 2 3 4)

\[ '(1 2 3 . nil) \]
(1 2 3)

What is the printed result of evaluating this expression?

\[ (\text{cdr } '((1 2) . (3 4 . (5)))) \]
(3 4 5)