Announcements

- HW9 due tonight
- Ants extra credit due tonight
  - See Piazza for submission instructions
- Hog revisions out, due Monday
- HW10 out tonight
Pairs

Scheme 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, enclosed in parentheses

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

> (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!
Symbolic Programming

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)

Symbols are now values

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

> (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?

> (cdr '(((1 2) . (3 4 . (5)))))
(3 4 5)
The Let Special Form

Let expressions introduce a new frame, with the given bindings

```
(let ((<name> <exp>) ...) <body>)
```

```
(define (filter fn s)
  (if (null? s)
      s
      (let ((first (car s))
            (rest (filter fn (cdr s))))
        (if (fn first)
            (cons first rest)
            rest)))))
```

> (filter even? '(1 2 3 4 5 6 7))
(2 4 6)
Quick Sort

Quick sort algorithm:
1. Choose a pivot (e.g. first element)
2. Partition into three pieces:
   < pivot, = pivot, > pivot
3. Recurse on first and last piece

\[(\text{define } (\text{filter-comp comp pivot s)})\]
\[\quad (\text{filter } (\text{lambda } (x) (\text{comp } x \text{ pivot})) s))\]

\[(\text{define } (\text{quick-sort s})\]
\[\quad (\text{if } (\leq (\text{length s}) 1)\]
\[\quad \quad s\]
\[\quad \quad (\text{let } ((\text{pivot } (\text{car s})))\]
\[\quad \quad \quad (\text{append } (\text{quick-sort } (\text{filter-comp } < \text{ pivot s}))\]
\[\quad \quad \quad \quad (\text{filter-comp } = \text{ pivot s})\]
\[\quad \quad \quad \quad (\text{quick-sort } (\text{filter-comp } > \text{ pivot s}))))))\]
The Begin Special Form

Begin expressions allow sequencing

\[(\text{begin } \langle \text{exp}_1 \rangle \ \langle \text{exp}_2 \rangle \ \ldots \ \langle \text{exp}_n \rangle)\]

\[(\text{define } (\text{repeat } k \ \text{fn})\]
\[\quad (\text{if } (> \ k \ 0)\]
\[\quad \quad (\text{begin } (\text{fn}) \ (\text{repeat } (- \ k \ 1) \ \text{fn}))\]
\[\quad \quad \text{'done})\]
\[\text{)}\]

\[(\text{define } (\text{tri } \text{fn})\]
\[\quad (\text{repeat } 3 \ (\text{lambda } () \ (\text{fn}) \ (\text{lt } 120))))\]
\[\text{)}\]

\[(\text{define } (\text{sier } d \ k)\]
\[\quad (\text{tri } (\text{lambda } () \ (\text{if } (= \ k \ 1) \ (\text{fd } d) \ (\text{leg } d \ k))))\]
\[\text{)}\]

\[(\text{define } (\text{leg } d \ k)\]
\[\quad (\text{sier } (/ \ d \ 2) \ (- \ k \ 1)) \ (\text{penup}) \ (\text{fd } d) \ (\text{pendown}))\]