Announcements
Turtle Graphics
Drawing Stars

(forward 100) or (fd 100) draws a line
(right 90) or (rt 90) turns 90 degrees

(define (star n m)
  (let ((a (/ (* 360 m) n)))
    (define (side k)
      (if (< k n) (begin (fd 100) (rt a) (side (+ k 1))))
        (side 0))))

(Demo)
Lists
Scheme Lists

In the late 1950s, computer scientists used confusing names

- **cons**: Two-argument procedure that creates a linked list
  - (cons 2 nil)
- **car**: Procedure that returns the first element of a list
  - > (define x (cons 1 (cons 2 nil))
  - > x
  - (1 2)
  - > (car x)
  - 1
- **cdr**: Procedure that returns the rest of a list
  - > (define x (cons 1 (cons 2 nil)))
  - (1 2)
  - > (cdr x)
  - (2)
- **nil**: The empty list

Important! Scheme lists are written in parentheses with elements separated by spaces

> (cons 1 (cons 2 nil))
(1 2)
> (define x (cons 1 (cons 2 nil))
> x
(1 2)
> (car x)
1
> (cdr x)
(2)
> (cons 1 (cons 2 (cons 3 (cons 4 nil))))
(1 2 3 4)

(Demo)
**List Construction**

**cons** is always called on two arguments: a first value and the rest of the list.

**list** is called on any number of arguments that all become values in a list.

**append** is called on any number of list arguments that all become concatenated in a list.

```
scm> (define s (cons 1 (cons 2 nil)))
(3 1 2)
scm> (list 3 s)
((3) 1 2)
scm> (list 3 s)
(3 (1 2))
scm> (list 3 s)
((3) (1 2))
scm> (list 3 s)
(3 1 (2))
scm> (list 3 s)
((3) (1 (2)))
scm> (list 3 s)
(3 1 (2))
scm> (list 3 s)
((3) (1 (2)))
scm> (list 3 s)
(3 (1 2))
scm> (list 3 s)
((3) (1 2))
scm> (list 3 s)
(3 (1 2))
scm> (list 3 s)
((3) (1 2))
scm> (cons 3 s)
((3) 1 2)
scm> (append 3 s)
--- Error
scm> (list s s)
((3) (1 2))
scm> (list s s)
(3 (1 (2)))
scm> (list s s)
((3) (1 (2)))
scm> (list s s)
(3 (1 (2)))
scm> (list s s)
((3) (1 (2)))
scm> (cons s s)
((3) (1 2))
scm> (cons s s)
((3) (1 2))
scm> (cons s s)
((3) (1 2))
scm> (append s s)
((1 2) 1 2)
scm> (append s s)
((1 2) (1 2))
scm> (append s s)
(1 2 1 2)
```

---
Recursive Construction

To build a list one element at a time, use `cons`
To build a list with a fixed length, use `list`

;;; Return a list of two lists; the first n elements of s and the rest
;;; scm> (split (list 3 4 5 6 7 8) 3)
;;; ((3 4 5) (6 7 8))

(define (split s n)
  ; The first n elements of s
  (define (prefix s n)
    (if (zero? n)  nil  (cons (car s)  (prefix  (cdr s)  (-  n 1))))))

  ; The elements after the first n
  (define (suffix s n)
    (if (zero? n)  s  (suffix  (cdr s)  (-  n 1)))))
    (list  (prefix s n)  (suffix s n)))
Recursive Construction Version 2

To build a list one element at a time, use `cons`
To build a list with a fixed length, use `list`

```scheme
;;; Return a list of two lists; the first n elements of s and the rest
;;; scm> (split (list 3 4 5 6 7 8) 3)
;;; ((3 4 5) (6 7 8))
(define (split s n)
  (if (= n 0)
      (list nil s)
      (let ((split-rest (split (cdr s) (- n 1))))
        (cons (cons (car s) (car split-rest))
              (cdr split-rest))))))
```
Symbolic Programming
Symbolic Programming

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

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

No sign of “a” and “b” in the resulting value

Quotation is used to refer to symbols directly in Lisp.

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

Quotation can also be applied to combinations to form lists.

```lisp
> '(a b c)
(a b c)
> (car '(a b c))
a
> (cdr '(a b c))
(b c)
```

Short for (quote a), (quote b): Special form to indicate that the expression itself is the value.

(Demo)
List Processing
Built-in List Processing Procedures

\[(\text{append s t})\]: list the elements of s and t; append can be called on more than 2 lists

\[(\text{map f s})\]: call a procedure f on each element of a list s and list the results

\[(\text{filter f s})\]: call a procedure f on each element of a list s and list the elements for which a true value is the result

\[(\text{apply f s})\]: call a procedure f with the elements of a list s as its arguments

\[
\begin{align*}
1 & 2 & 3 & 4 \quad \text{; count} \\
\text{(and a 1)} & \text{(and a 2)} & \text{(and a 3)} & \text{(and a 4)} \quad \text{; beats} \\
\text{(and a 1 and a 2 and a 3 and a 4)} \quad \text{; rhythm}
\end{align*}
\]

(\text{define count (list 1 2 3 4)})
(\text{define beats (map \text{lambda} \text{x} \text{list 'and 'a x}) count})
(\text{define rhythm (apply append beats)})