Scheme Lists

Announcements

More Special Forms

Cond

The **cond** special form that behaves like if-elif-else statements in Python

```python
if x > 10:
    print('big')
elif x > 5:
    print('medium')
else:
    print('small')
```

```scheme
(cond ((> x 10) (print 'big))
      ((> x 5)  (print 'medium))
      (else     (print 'small)))
```

```scheme
(print
  (cond ((> x 10) 'big)
        ((> x 5)  'medium)
        (else     'small)))
```

You can just use **if** repeatedly instead

```scheme
(if (> x 10)  (print 'big)
  (if (> x 5) (print 'medium)
              (print 'small)))
```

This is the *alternative* (3rd subexpression) of the outer **if**

```scheme
(print
  (if (> x 10)  'big
    (if (> x 5) 'medium
                'small)))
```

Begin

The **cond** special form that behaves like if-elif-else statements in Python

```
if x > 10:
    print('big')
elif x > 5:
    print('medium')
else:
    print('small')
```

```
(cond ((> x 10) (print 'big))
      ((> x 5)  (print 'medium))
      (else     (print 'small)))
```

```
(print
  (cond ((> x 10) 'big)
        ((> x 5)  'medium)
        (else     'small)))
```

The **begin** special form combines multiple expressions into one expression

```
if x > 10:
    print('big')
    print('guy')
else:
    print('small')
    print('fry')
```

```
(cond ((> x 10) (begin (print 'big)   (print 'guy)))
      (else     (begin (print 'small) (print 'fry))))
```

```
(if (> x 10) (begin
                (print 'big)
                (print 'guy))
             (begin
                (print 'small)
                (print 'fry)))
```

Example: Euclid's Algorithm for Greatest Common Divisor (GCD)

From lab: The Greatest Common Divisor (GCD) is the largest integer that evenly divides two positive integers.

Write the procedure gcd, which computes the GCD of numbers a and b using Euclid's algorithm, which recursively uses the fact that the GCD of two values is either of the following:

- the smaller value if it evenly divides the larger value, or
- the greatest common divisor of the smaller value and the remainder of the larger value divided by the smaller value

```
252 % 105 = 42
105 %  42 = 21
 42 %  21 = 0
```

21 is GCD(252, 105)

An idea: make sure **a** ≥ **b** (so that **a** is the "larger value")

(modulo a b) was 0 in the previous call

```
(define (gcd a b)
  (cond ((zero? b) a)
        ((< a b) (gcd b a)) ; Swap them!
        (else    (gcd b (modulo a b)))))
```

```
(define (gcd a b)
  (if (< a b) ; Swap them!
    (begin
      (define tmp a)
      (define a b)
      (define b tmp)))
  (if (zero? b) a (gcd b (modulo a b))))
```

This is the *consequent* (2nd subexpression) of the **if**

Example: Euclid's Algorithm for Greatest Common Divisor (GCD)

From lab: The Greatest Common Divisor (GCD) is the largest integer that evenly divides two positive integers.

Write the procedure gcd, which computes the GCD of numbers a and b by recursively using the fact that the GCD of two values is either of the following:

- the smaller value if it evenly divides the larger value, or
- the greatest common divisor of the smaller value and the remainder of the larger value divided by the smaller value

```
252 % 105 = 42
105 %  42 = 21
 42 %  21 = 0
```

21 is gcd(252, 105)

An idea: make sure **a** ≥ **b** (so that **a** is the "larger value")

(modulo a b) was 0 in the previous call

```
(define (gcd a b)
  (cond ((zero? b) a)
        ((< a b) (gcd b a)) ; Swap them!
        (else    (gcd b (modulo a b)))))
```

Discuss: Why is **c** here?
Why didn't we need **c** in the other version?

```
(define (gcd a b)
  (if (< a b) ; Swap them!
    (begin
      (define c a)
      (define a b)
      (define b c)))
  (if (zero? b) a (gcd b (modulo a b))))
```

This is the *consequent* (2nd subexpression) of the **if**

Turtle Graphics

Drawing Stars

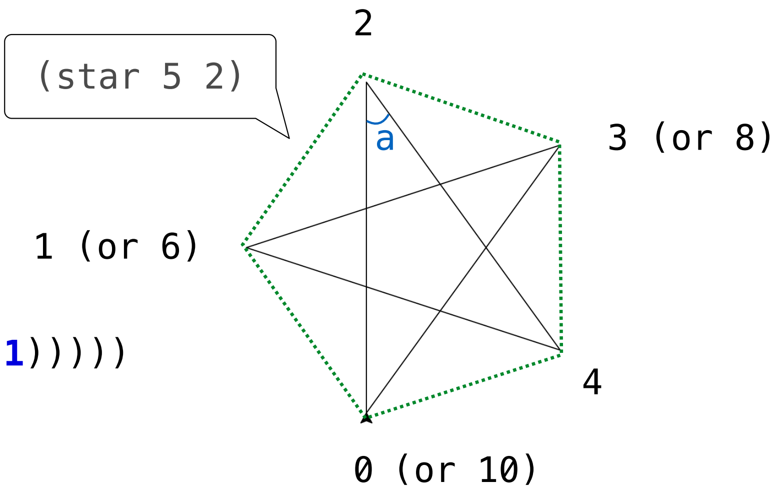
(forward 100) or (fd 100) draws a line

(right 90) or (rt 90) turns 90 degrees

Number of sides

Where to go next

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



(Demo)

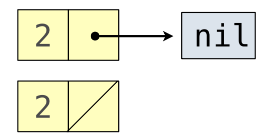
Lists

Scheme Lists

In the late 1950s, computer scientists used confusing names

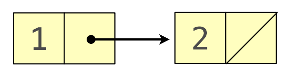
- **cons:** Two-argument procedure that creates a linked list
- **car:** Procedure that returns the first element of a list
- **cdr:** Procedure that returns the rest of a list
- **nil:** The empty list

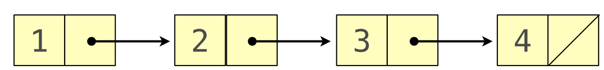
(cons 2 nil)



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)))

scm> (list 3 s)

scm> (cons 3 s)

scm> (append 3 s) Error

scm> (list s s)

scm> (cons s s)

scm> (append s s)
```

pollev.com/cs61a

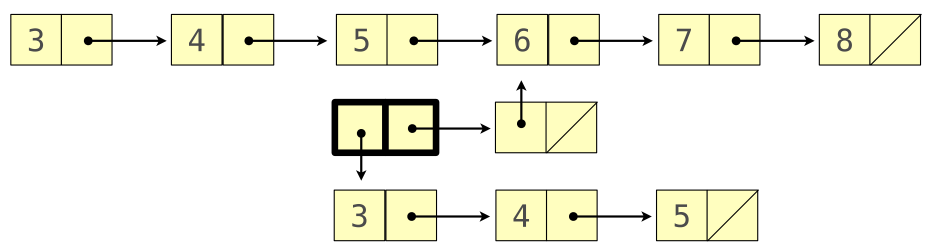
```
(3 1 2)
((3) 1 2)
(3 (1 2))
((3) (1 2))
(3 1 (2))
((3) 1 (2))
(3 (1 (2)))
((3) (1 (2)))

((1 2) (1 2))
((1 2) 1 2)
(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**




```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)
  ; 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)))
```

pollev.com/cs61a

Symbolic Programming

Symbolic Programming

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

```
> (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.

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

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

Quotation can also be applied to combinations to form lists.

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

(Demo)

List Processing

Built-in List Processing Procedures

(append s t): list the elements of s and t; append can be called on more than 2 lists

(map f s): call a procedure f on each element of a list s and list the results

(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

(apply f s): call a procedure f with the elements of a list s as its arguments

```
(1 2 3 4)                                ; count
((and a 1) (and a 2) (and a 3) (and a 4)) ; beats
(and a 1 and a 2 and a 3 and a 4)        ; rhythm
```

```
(define count (list 1 2 3 4))

(define beats (map (lambda (x) (list 'and 'a x)) count)

(define rhythm (apply append beats))
```

pollev.com/cs61a