Syntactic Forms

if, begin, or, and get useful syntactic forms

They have has evaluation, i.e., their subexpressions are not evaluated until required.

Let's look at lazy evaluation and how to exhibit it.

(is (w u o) (display "x") (/ 1 0))

is is evaluated left to right. The "else part" is only evaluated if the conditional expression is false.

Imagine if we were implemented as a procedure. We'd be in trouble.

(begin
  (display "this is line 1 of the message")
  (display "this is line 2 of the message")
  #f)

begins evaluates its subexpressions from left to right and returns the value of the last subexpression.

Syntactic Forms (cont.)

(begin (display "hello") (display "world")) #f

(begin (display "hello") (display "world") (display "hi")) #f

(begin (display "hello") (display "world") (display "hi") (display "morning")) #f

(begin (display "hello") (display "world") (display "hi") (display "morning") (display "good morning")) #f

Clever Exploitation of Syntactic Forms and Lazy Evaluation

(define (valid? message) (msg? message))

(msg? ("hello") (msg? "world") (msg? ("hi") (msg? "morning") (msg? "good morning"))) #f

When Lazy Evaluation isn't our friend

Problem: Sometimes lazy evaluation works against you.

Challenge with valid? is that it's a syntax check function, so it must return #t/or (depending on whether the expression has valid syntax, but you must go through the entire list even after you generate your first. How do you do that?

Hint: There is a construct in Scheme that forces evaluation of a series of expressions before performing some condition on it. (There are many actually.) Let's think of one we've seen in data and use it.

Summary: Functional Programming

- Pure functional languages:
  - Referential transparency
  - No accumulation
  - No iteration, only recursion
  - Implicit storage management
  - Functions are values

- λ-calculus
- LISP, Common LISP, Scheme
- Built-in procedures
- Lists (cons, car, cdr, null?)
- Reading/writing loop
- Inhibiting + activating evaluation (call, nCons)
- Procedure definition and lambda expressions

- Conditions (+, etc.)
- Equality checking (+, =, eq?, eq?)
- Recursion (define, procedure)
- Efficiency concerns
  - helper procedures
  - let, let*, ... accumulation
- Higher-order functions (map, apply, lambda)
- Passing procedures, returning procedures
- Anonymous procedures
- Syntactic forms and lazy evaluation