6.001 SICP

6001-Introduction
logistics
r: Srin Devadas (devadas@mit.edu)
ta: David Ziegler (dziegler@mit.edu)
ta: David Pritchard (davp@mit.edu)
section web page: http://cs.lcs.mit.edu/~devadas/6.001
Tutorial room 36-117 code: 5312
34-501 Lab outer door: 47519
Lab inner door: 72962

basic-scheme
read eval print
expressions
project 0

Scheme Basics

• Language provides us with a basic set of data elements
  • Numbers
  • Characters
  • Booleans
• And a basic set of operations on these primitive elements
• Use basic elements to construct more complex processes

Scheme Basics

• Legal expressions have rules for constructing from simpler pieces

Example: (+ 2 3) (* (+ 1 2) 3)

• (Almost) every expression has a value, which is "returned" when an expression is evaluated.

Example: (+ 2 3) 5

Syntax

Things that make up scheme programs:

• self-evaluating 23, "hello", #t
• names +, pi
• combinations (+ 2 3) (* pi 4)
• special forms (define pi 3.14) (lambda ...) (if ...)
Note that special forms are not combinations, even though they are syntactically similar.
A combination requires that the first subexpression be a procedure.

Evaluation Rules

• A numeral, string or boolean evaluates to itself
  (number, string, #t, #f).

• A name evaluates to the value associated with that name in the environment

• A combination is evaluated as follows:
  1. Evaluate the subexpressions in any order
  2. Apply the value of the operator subexpression to the values of the remaining subexpressions.

Examples

7
8.5
(* 2 4)
(+ 5 3) (/ 9 3))
6
(7 4)
error
Examples

(> 7 3)
#t
y
unbound
(+ y 1)
unbound
(define y 17)
y -> 17
y
17
(+ y 1)
18

What about...

(+ 4)
-4
(- 3)
-3
(/ 5)
-2
(/ 60 5 2 3)
2
(*)
0
(*)
1
(-)
*error*

Define

(define x 4)
x
= 4
(+ x 5)
9
(define y (+ x 2))
y
6
(define x 2)
y
??

Language elements -- abstractions

• Need to capture ways of doing things – use procedures

  To process something multiply it by itself

  (lambda (x) (* x x))

  • Special form – creates a procedure and returns it as value

Interaction of define and lambda

(l lambda (x) (* x x))

=== #[compound-procedure 9]

(define square (lambda (x) (* x x)))

=== undef

(square 4)

=== 16

(((lambda (x) (* x x)) 4)

=== 16

Fill in each box...

(define twice
(l lambda (x) (* 2 x))

(twice 2) =>> 4
(twice 3) =>> 6

(define constant2 (lambda (x) 2))

(constant2) =>> 2

(define second
(l lambda (x y z) y))

(second 2 15 3) =>> 15
(second 34 -5 16) =>> -5