6.090
Building Programming Experience

Lecture 2

1/11/2007

Abstracting Computation

- Yesterday we learned how to abstract data
  (define *name value*)
- Now want to abstract computation
- Create our own new procedures
- **Lambda** new special form

lambda

- Special form that returns a procedure
- (lambda (arg1 arg2) body)
- Example:
  - (lambda (x) (* x x))
- Call this a compound procedure
  - Not a built-in primitive

Using compound procedures

- Apply the procedure using a combination
- ((lambda (x) (* x x)) 5)
- Three steps to evaluate expression

1.
2.
3.

Lambda rules

- Lambda syntax:
  (lambda (x y) (/ (+ x y) 2))
- 1st operand: *parameter list*: (x y)
  - List of names (possibly empty): ()
  - Determines the number of operands required
- 2nd operand: the *body*: (/ (+ x y) 2)
  - Any expression
  - Not evaluated when the lambda is evaluated

Scheme Basics

- Rules For Evaluation:
  - If **self-evaluating**, return value
  - If a **name**, return value associated with the name in the environment
  - If a **special form**, do something special
  - If a **combination**, then
    - *Evaluate* all of the subexpressions in any order
    - *Apply* the operator to the values of the operands and return the result
- Rules For Application:
  - If it's a primitive procedure, just do it
  - It it's a compound procedure then:
    evaluate the body of the procedure with each formal parameter replaced by the corresponding actual argument value

Abstracting Procedures

- In Scheme a procedure is a type just like a number or string
- We can assign a name to a procedure using **define**
- (define increment (lambda (x) (+ x 1)))
- Expression does two things

Lecture Problems

- Write a procedure cube that returns the cube of its input
- (define cube
- Write a procedure neg that takes a number and negates it

Lecture Problem

- Given a margin width m, which is the top, bottom, left, and right margin of the page, write a procedure which computes the useable (non-margin) area of an 8.5in by 11in sheet of paper.
- (usable-page 0) -> 93.5
- (usable-page 1) -> 58.5
- (define usable-page

- Now modify usable-page to take four separate arguments for different margins: top, bottom, left, right

if

- Special form
- (if *test consequent alternative*)
- Type this into DrScheme: (if (> 6 5) 0 1)

Lecture Problem

- Write a procedure the-answer?, which returns true (#t) if the input is the number 42, and false otherwise.
- (define the-answer?

Recursion

- Not all computation has a fixed number of steps
- Example: Sum numbers from 1 to n
  - (define sumto (lambda (n) …
- Procedures can call themselves

Sum to n

- What cases do we know the answer in a fixed set of computations?
- Write the expression to compute sumto in these cases
- Call this the **base case**

Sum to n

- Let's compute the sum from 1 to 5
- Assume that sumto works for numbers less than 5
- Combine that answer with some simple computation to get an answer
- This part is the **recursive case**

Put it all together

- We need three things:
  - A way to solve the base case
  - A way to combine a simpler application of sumto with some simple operations
  - A way to choose which case to use

Use **if**

- (if *test consequent alternative*)
- Basic idea:
  - (define sumto
(lambda (n)
(if *base_case*
base_expression
(op (sumto (*op* n)))))

Solution

- (define sumto
```
  (lambda (n)
    (if (= n 1)
        1
        (+ n (sumto (- n 1))))))
```

Writing factorial

- Same pattern applies to calculating factorials
- Pick a plan:
  - What's the base case?
  - What's the recursive case?

Factorial

```
(define fact
  (lambda (n)
    (if (= n 1)
        1
        (* n (fact (- n 1))))))
```

Exponentiation

- Calculate x^n where n>=0 and n is an integer
- Base case
- Recursive case?

Recursion is like proof by induction

- Prove: sum from 1 to n = n * (n+1) / 2