NCCU
Programming Languages
程式語言原理

Spring 2006

Lecture 8: Scheme, II

Agenda

- Scheme Programming
  - Binding
  - Higher-Order Functions
  - Lazy Evaluation and Streams
  - Substitution Model vs. Environment Model

Bindings

- A binding is an association between a name and a Scheme value
- Examples:

  name: x value: 10

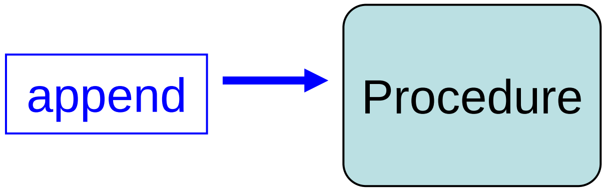
  name: y value: #f

  name: square value: (lambda (x) (* x x))

  Function value (first-class value)

Binding Constructs in Scheme

- Binding: Associate a symbol to something (meaning) in a context



1. `define`
   - binds value to a name.
2. λ-function application
   - binds formal parameters to actual argument values.
3. `let`-constructs

   introduces local bindings (local variables)
   - `let`
   - `let*`
   - `letrec`

`let`-construct

```
( let
        ( (var1 exp1) … (varn expn))
        exp
)
```

- `exp1` to `expn` are evaluated in the surrounding context.
- `var1,…,varn` are visible *only* in `exp.`
  (local variables)

```
> (let ( (x 2) (y 7) ) y)
 => 7
```

```
> (let ( (x y) (y 7) ) y)
```
=>*error* "*y*" undefined

```
> (define y 5)
> (let ( (x y) (y 7) ) y)
=>7
> (let ( (x y) (y 7) ) x)
=>5
> (let ( (y 7) (x y) ) x)
```
=>5 (not 7)

"let" is a syntactic sugar

(let ((x 2) (y 5)) (+ x y))	等於 →	((lambda (x y) (+ x y)) 2 5)

```
(let ((x  2)  (y  5))
     (+ x  y))
```

等於 →

```
( (lambda (x y) (+ x y))
2  5)
```

Suppose we wish to implement the function

$$f(x,y) = x(1+x*y)^2 + y(1-y) + (1+x*y)(1-y)$$

We can also express this as

$$a = 1+x*y$$

$$b = 1-y$$

$$f(x, y) = xa^2 + yb + ab$$

The syntactic sugar “let”

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (f-helper (+ 1 (* x y))
            (- 1 y)))
```

```
(define (f x y)
  ((lambda (a b)
     (+ (* x (square a))
        (* y b)
        (* a b)))
   (+ 1 (* x y))
   (- 1 y)))
```

```
(define (f x y)
  (let ((a (+ 1 (* x y)))
        (b (- 1 y)))
    (+ (* x (square a))
       (* y b)
       (* a b))))
```

The syntactic sugar “Let”

```
(let ((<var1> <exp1>)
      (<var2> <exp2>)
        ..
      (<varn> <expn>))
    <body>)


((lambda (<var1> ….. <varn>)
    <body>)
  <exp1>
  <exp2>
  …
  <expn>)
```

Nested let's

```
> (define y 5)
> (let ( (y 7) (x y) ) x)
=> 5
> (let ( (y 7) )
     (let ( (x y) ) x) ) ;;nested let's
=> 7

> (let* ( (y 7) (x y) ) x)
=> 7
```

- `let*` abbreviates nested-`lets`.
- Recursive and mutually recursive functions cannot be defined using `let` and `let*`.

`letrec`-construct

```
( letrec
        ( (var1 exp1) … (varn expn))
        exp
)
```

- $var_1, …, var_n$ are *visible* in exp_1 to exp_n in addition to `exp`.

```
>  (letrec  ( (x (lambda() y)
              (y (lambda() x)  )
        x
   )
```

`letrec`-construct

```
> (letrec ((f  (lambda(n)  (if  (zero? n) 1  (f (- 1 n)) ))) )
                                   (f  5)
  )
> 1

> (letrec (  ( f    (lambda () g)   )
                            ( g    2 )                 )
           ( f )
  )
> 2
```

Higher-Order Functions (Procedures)

- Functions are *first-class values*
- Functions (Procedures) can get functions (procedures) as *arguments*
- Functions (Procedures) can *return functions* (procedures) as values

(Review) Functions in Scheme

- Create a function by evaluating a lambda expression:

  (lambda (id1 id2 ...) exp1 exp2 ...)

  - id1 id2 ... - formal parameters
  - exp1 exp1 ... - body of the function
  - return value of function - last expression in body
  - return value of lambda expression - the (un-named) function

  (lambda (x) (* x x)) => #<procedure>

  - Returns an un-named function that takes a parameter and returns its square

(Review) Functions in Scheme

- Call a function by applying the evaluated lambda expression on its actual parameters:

```
((lambda (x) (* x x)) 3)      => 9
```

the function   the actual parameter

- How can you reuse the function?
  - You can't!
- Why is it then useful?
  - Return a function from another function
- What if you REALLY want to reuse it?

Functions are First-Class Values

C

```
if (a == 0)
    return f(x,y) ;
else
    return g(x,y) ;
```

repeated arguments (x,y)

- Can we write it better in Scheme?

Scheme

```
((if (= a 0)
    f
    g)
 x y)
```

evaluates to either #<procedure f> or #<procedure g>

it is then applied on arguments x and y

Passing Functions to Other Functions

We considered the following three sums

- $1 + 2 + \ldots + 100 = (100 * 101)/2$
- $1 + 4 + 9 + \ldots + 100^2 = (100 * 101 * 102)/6$
- $1 + 1/3^2 + 1/5^2 + \ldots + 1/101^2 = \pi^2/8$

$$\sum_{k=1}^{100} k$$

$$\sum_{k=1}^{100} k^2$$

$$\sum_{k=1,odd}^{101} k^{-2}$$

In mathematics they are all captured by the notion of a **sum**:

$$\sum_{x \in l} f(x)$$

Can we express this abstraction directly?

Let's have a look at the three programs

$$\sum_{k=1}^{100} k$$ = (sum-integers 1 100)

```
(define (sum-integers k  n)
   (if  (>  k  n)
        0
        (+  k
             (sum-integers (+ 1 k)  n))))
```

$$\sum_{k=1}^{100} k^2$$ = (sum-squares 1 100)

```
(define (sum-squares  k  n)
   (if  (>  k  n)
        0
        (+  (square K)
             (sum-squares  (+ 1 k)  n))))
```

$$\sum_{k=1,odd}^{101} k^{-2}$$ = (pi-sum 1 101)

```
(define (pi-sum  k  n)
   (if  (>  k  n)
        0
        (+  (/  1  (square  k))
            (pi-sum   (+  k 2)  n))))
```

Abstracting from the three programs

$$\sum_{x \in l} f(x)$$

```
(define (sum  f  k  next n)
  (if  (>  k  n)
       0
       (+  (f  k)
           (sum  f  (next k) next n))
 ))
```

```
(define (sum-integers k  n)
   (if  (>  k  n)
        0
        (+  k
             (sum-integers (+ 1 k)  n))))
```

```
(define (sum-squares  k  n)
   (if  (>  k  n)
        0
        (+  (square a)
             (sum-squares  (+ 1 k)  n))))
```

```
(define (pi-sum  k  n)
   (if  (>  k  n)
        0
        (+  (/  1  (square  k))
            (pi-sum   (+  k 2)  n))))
```

Higher Order Procedures $\sum_{x \in l} f(x)$

A higher order procedure:

takes a procedure as an argument or

returns one as a value

Examples:

1. (define (sum-integers1 k n)
   (sum <u>(lambda (x) x)</u> k <u>(lambda (x) (+ x 1))</u> n))

2. (define (sum-squares1 k n)
   (sum <u>square</u> k <u>(lambda (x) (+ x 1))</u> n))

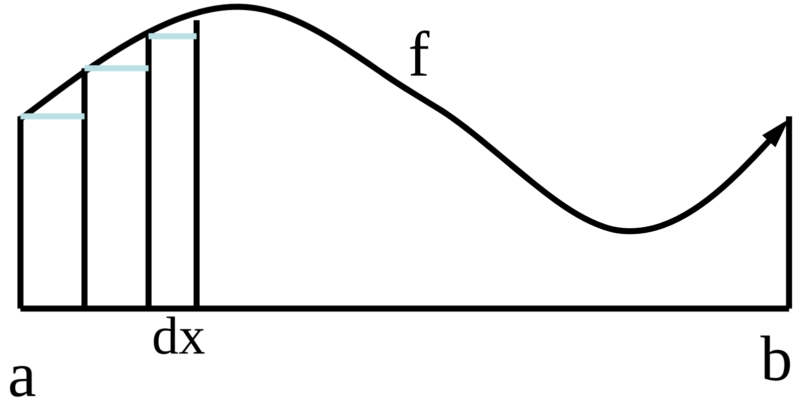
3. (define (pi-sum1 k n)
   (sum <u>(lambda (x) (/ 1 (square x)))</u> k <u>(lambda (x) (+ x 2))</u> n))

sum: $\underbrace{\text{(number} \rightarrow \text{number}}_{\text{procedure}}\text{, number, } \underbrace{\text{number} \rightarrow \text{number}}_{\text{procedure}}\text{, number)} \rightarrow \text{number}$

Integration as a procedure

Integration under a curve f is given roughly by

dx (f(a) + f(a + dx) + f(a + 2dx) + … + f(b))




```
(define (integral f a b) // f is a function
    (*  (sum f a (lambda (x) (+ x dx)) b)  dx))
(define dx 1.0e-3)
(define atan (lambda (a)
                  (integral (lambda (x) (/ 1 (+ 1 (square x))))  0  a)))
```

`Apply`-function

```
(apply  cons  ’( x  (y z)))
= (cons  ’x   ’(y z))
= (x y z)

(apply   length  ‘((1 a 2 b)))
= (length  ‘(1 a 2 b))  =  4

(apply   f    ’(a1 a2 ... an))
= (f   ’a1  ’a2 ...  ’an)
```

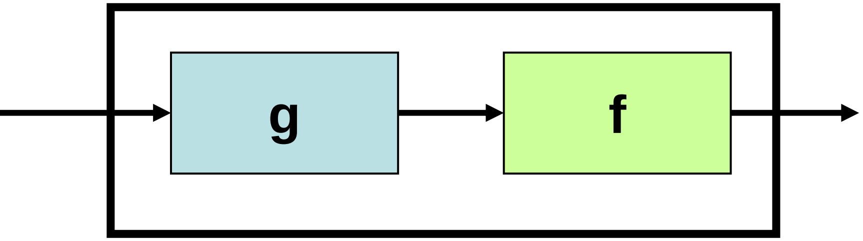
(apply <*func*> <*list-of-args*>)

Function Composition as a HOF

- Given two functions *f* and *g*, the composition

  $f \circ g$

  is yet another <u>function</u>, a higher-order one.




```
(define compose
    (lambda (f  g)
        (lambda (x)  (f  (g  x)))
    )
)
```

```
➢ (compose  car  cdr)
[compound procedure]

➢ ((compose car cdr)  '(1  2  3))
2
```

HOF for List Manipulation

Consider the following three functions:

; double each list element

```
(define  (double  l)  (if  (null?  l)  ‘()
                              (cons  (* 2 (car  l))  (double (cdr  l)))        ))
```

; invert each list element

```
(define  (invert  l)  (if  (null?  l)  ‘()
                              (cons  (/ 1  (car  l))  (invert  (cdr  l)))        ))
```

; negate each list element

```
(define  (negate  l)  (if  (null?  l)  ‘()
                              (cons  (not  (car  l))  (negate  (cdr  l)))     ))
```

map: Higher-Order Functions over List

Higher-order function approach:

```
(define (map f l)
        (if (null? l)
        '()
        (cons (f (car l)) (map f (cdr l) ) ))
)
```

(map (lambda (n) (* 2 n)) '(1 2 4))

value: (2 4 8)

- (map (lambda (n) (/ 1 n)) '(1 2 4))

value: (1 0.5 0.25)

- (map not '(#t #f #t))

value: (#f #t #f)

Also:

> (define (negate l) (map not l))

In-Class Exercise

- (map car ‘((1 2 3) (a b c) (w x y))
  (1 a w)
- (map (lambda (x) (cons ‘H X)) ‘((1 2 3) (a b c)))
  ((H 1 2 3) (H a b c))

- Generate all subsets of given set (represented by a list)?
  (subset ‘()) → (())
  (subset ‘(1)) → (() (1))
  (subset ‘(1 2)) → (() (1) (2) (1 2))

```
(define (subset s)
   (if  (null? s)
        (list s)
        (let  ((cdrsubset   (subset  (cdr s))) )
            (append  cdrsubset   (map  (lambda (x) (cons (car s) x))   cdrsubset) )) )
)
```

More on Functions as parameters

Consider these functions:

```
; sum list elements
(define  (sum  l)   (if  (null?  l)  0   (+ (car  l)  (sum (cdr  l)))))

; take product of list elements
(define  (prod l)   (if  (null?  l)  1   (*  (car  l)  (prod  (cdr  l)))  ))

; logical or list elements
(define  (l_or  l)   (if  (null?  l)  #f   (or  (car  l)  (l_or  (cdr  l)))  ))

; logical and list elements
(define  (l_and  l)   (if  (null?  l)  #t   (and  (car  l)  (l_and  (cdr  l)))  ))
```

More on Functions as parameters, Cont'd

Where are these functions different?

```
; sum list elements
(define  (sum  l)   (if  (null?  l)  0   (+ (car  l)  (sum (cdr  l)))  ))

; take product of list elements
(define  (prod l)   (if  (null?  l)  1   (*  (car  l)  (prod  (cdr  l)))  ))

; logical or list elements
(define  (l_or  l)   (if  (null?  l)  #f   (or  (car  l)  (l_or  (cdr  l)))  ))

; logical and list elements
(define  (l_and  l)   (if  (null?  l)  #t   (and  (car  l)  (l_and  (cdr  l)))  ))
```

`reduce`: Higher-Order List Manipulation Function

Higher-order function approach:

```
(define (reduce l op id)
      (if (null? l)
              id
              (op (car l) (reduce (cdr l) op id))
      )
)
```

- (reduce ‘(1 2 4) + 0)

value: 7

- (reduce ‘(1 2 4) * 1)

value: 8

- (reduce ‘(#t #f #f) boolean/or #f)

value: #t

- (reduce ‘(#t #f #f) boolean/and #t)

value: #f

Also:
> (define (sum l) (reduce l + 0))

Using `map` and `reduce`

To implement summation: $$\sum_{x \in l} f(x)$$

(define (sigma f l) (reduce (map f l) + 0))

E.g.,

$\sum(x)$: > (sigma (lambda (x) x) '(1 2 3))

value: 6

$\sum(x^2)$: > (sigma (lambda (x) (* x x)) '(1 2 3))

value: 14

Using map and reduce

Use sum to count the symbols in a list:

```
➢ (define (atomcount  s)
              (cond  ( (null?  s)   0)
                     ( (atom? s)  1)
                     (else  (sigma   atomcount  s))
              )
       )
```

➢ (atomcount ‘(1 2))

value: 2

➢ (atomcount ‘(1 (2 (3)) (4)))

value: 4

```
(define (atom? a)
   (not (pair? a)))
```

Function Types

- A new type constructor

  (T1,T2,...,Tn) -> T0

  Takes n arguments of type *T1, T2, ..., Tn* and returns a value of type *T0*

  Unary function: T1 -> T0   Nullary function: () -> T0

- Example:

```
sort ( A: int[], order: (int,int)->boolean ) {
    for (int i = 0; i<A.size; i++)
        for (int j=i+1; j<A.size; j++)
            if (order(A[i],A[j]))
                switch A[i] and A[j];
 }
 boolean leq ( x: int, y: int ) { return x <= y; }
 boolean geq ( x: int, y: int ) { return x >= y; }
 sort(A, leq)
 sort(A, geq)
```

How can you do this in Java?

```
interface Comparison {
  boolean compare ( int x, int y );
}
void sort ( int[] A, Comparison cmp ) {
    for (int i = 0; i<A.length; i++)
          for (int j=i+1; j<A.length; j++)
              if (cmp.compare(A[i],A[j]))
                 ...
}
class Leq implements Comparison {
  boolean compare ( int x, int y ) { return x <=y; }
}
sort(A,new Leq);
```

How to Develop a Higher-Order Sorting

- Parameterize the comparison function!

- In C?
- In Scheme
- Assignment

Functions Returned from Function Calls

Common in Math:

$$f(x) = \frac{d}{dx}(F(x))$$

$$F(x) = \int f(x)dx$$

An example:

- Consider defining all these functions:

```
(define add1  (lambda (x) (+ x 1))
(define add2  (lambda (x) (+ x 2))
(define add3  (lambda (x) (+ x 3))
(define add4  (lambda (x) (+ x 4))
(define add5  (lambda (x) (+ x 5))
```

- …repetitive, tedious.

The D.R.Y. principle

- D.R.Y. → "Don't Repeat Yourself“

- Whenever we find ourselves doing something rote/repetitive… ask:
  - Is there a way to abstract this?
  - Here, "abstract" means:
    - capture common features of old procedures in a *more general* new procedure

Abstracted adder function

- Generalize:

```
(define add1  (lambda (x) (+ x 1))
(define add2  (lambda (x) (+ x 2))
(define add3  (lambda (x) (+ x 3))
...
```

- to:

```
(define (make-addn n)
        (lambda (x) (+ x n))
)
```

Return a function

Abstracted adder function

- Generalize to a function that can *create* adders:

```
(define (make-addn n)
    (lambda (x) (+ x n)))
```

- Equivalently:

```
(define make-addn
  (lambda (n)
      (lambda (x) (+ x n))))
```

- note the *nested* lambda expressions!

How do I use it?

```
(define (make-addn n)
    (lambda (x) (+ x n)))
```

- (define add2 (make-addn 2))
- (define add3 (make-addn 3))
- (add3 4)
- 7

Evaluating…

- (define add3 (make-addn 3))
  - Evaluate (make-addn 3)
    - evaluate 3 → 3.
    - evaluate make-addn
      → (lambda (n) (lambda (x) (+ x n)))
    - apply make-addn to 3…
      - *substitute* 3 for n in (lambda (x) (+ x n))
      → (lambda (x) (+ x 3))
  - Make association:
    - add3 bound to (lambda (x) (+ x 3))

Evaluating (add3 4)

- (add3 4)
- Evaluate 4
- Evaluate add3
  → (lambda (x) (+ x 3))
- Apply (lambda (x) (+ x 3)) to 4
  - substitute 4 for x in (+ x 3)

  → (+ 4 3)

  → 7

Determining the Meaning of a Scheme Expression

- Substitution Model (Lambda calculus)
- Environment Model

Slides taken from CS 1 of Caltech Fall 2004.

Precision

- human (natural) language is imprecise
  - full of ambiguity
- computer languages must be precise
  - only one meaning
  - no ambiguity

Scheme combinations

- recall:

  (operator operand1 operand2 …)

  – delimited by parentheses
  – first element is the **Operator**
  – rest are **Operands**

- What does a Scheme expression mean?
- in other words:
- How do we know what value will be calculated by an expression?

Substitution Model

(operator operand1 operand2 …)

To evaluate a scheme expression:

1. **Evaluate** the operands
2. **Evaluate** its operator (a function or procedure)
3. **Apply** the operator to the evaluated operands
   – Using substitution if theer are variables involved

example expression eval

No variables

- example: (+ 3 (* 4 5))
  - evaluate 3
  - evaluate (* 4 5)
    - evaluate 4
    - evaluate 5
    - evaluate *
    - apply * to 4, 5 → 20
  - evaluate +
  - apply + to 3, 20 → 23

evaluation with variables

- An assignment provides an association between a variable and its value
  - (define x 3)
- To evaluate a variable:
  - look up the value associated with the variable
  - and replace the variable with its value

variable evaluation example

- (define x 3)
- then evaluate **x**
  - look up value of x
  - x has value 3 (due to define)
  - result: 3

simple expression evaluation

- assignment and evaluation
  - (define X 3)
  - (define Y 4)
  - evaluate (+ X Y)
    - evaluate X $\rightarrow$ 3
    - evaluate Y $\rightarrow$ 4
    - evaluate + $\rightarrow$ [primitive procedure +]
    - apply + to 3, 4 $\rightarrow$ 7

special forms

- ***N.B.*** There are a few special forms which do ***not*** evaluate in the way we've described.
- **define** is one of them
  - (define x 3)
  - We do **not** evaluate x before applying define to x and 3
  - instead, we
    - evaluate the second operand (3 → 3)
    - make an association between it and the first operand (x)

lambda

- lambda is also a special form:
  - result of a lambda expr is always a function
    - also known as a procedure
  - we do ***not*** evaluate its *contents*
    - ***none*** of the operands get evaluated
    - just “save them for later”

Scheme function definition

- translate: a(r)= pi * r^2

which performs the operation

- (define a (lambda (r) (* pi (expt r 2))))

of one variable, r

define a to be a function

evaluating lambda

- (define *a* (lambda (r) (* pi (expt r 2))))
  - eval: (lambda (r) (* pi (expt r 2)))
    - create *procedure* with one *argument* r and *body* (* pi (expt r 2))
    - details aren't important
    - can write as
    - (lambda (r) (* pi (expt r 2))) → (lambda (r) (* pi (expt r 2)))
  - make association *(binding)* between *a* and the *new procedure (come back to this later)*

evaluating a function call

to evaluate a function call:

[use the standard rule]

1. evaluate the operands (arguments)
2. apply the operator (function) to the (evaluated) operands (arguments)

to apply a function to its arguments:

1. substitute the *function argument* variables with the *values* given in the call everywhere they occur in the function body
2. evaluate the resulting expression

example 1

- (define f

  (lambda (x)

  (+ x 1)))
- evaluate (f 2)
  - evaluate 2 → 2
  - evaluate f → (lambda (x) (+ x 1))
  - apply (lambda (x) (+ x 1)) to 2
    - substitute 2 for x in the expression (+ x 1) → (+ 2 1)
    - evaluate (+ 2 1) → (skip obvious steps) → 3

example 2

- (define f (lambda (x y)

  (+ (* 3 x) (* -4 y) 2)))
- evaluate (f 3 2)
  - evaluate 3 → 3
  - evaluate 2 → 2
  - evaluate f → (lambda (x y) (+ (* 3 x) (* -4 y) 2))
  - apply (lambda (x y) …) to 3, 2
    - substitute 3 for x, 2 for y in body

      → (+ (* 3 3) (* -4 2) 2)
      - evaluate … 3

syntactic sugar

- equivalent expressions:

  (define f (lambda (x) (+ x 1))

  (define (f x) (+ x 1))
- simply an alternate syntax
  - allows us not to write lambda everywhere
  - feels more natural
  - means the **same** thing

evaluating *define*

To evaluate: (define (f x) (+ x 1))

1. "desugar" it into lambda form:
   - (define f (lambda (x) (+ x 1)))
2. now evaluate like any define:
   - create the function (lambda (x) (+ x 1))
   - create an association *(binding)* between the name f and the function

example

- (define **sq** (lambda (x) (* x x)))
- (define **d** (lambda (x y) (+ (sq x) (sq y))))
- evaluate: (**d** 3 4)
  - evaluate 3 → 3
  - evaluate 4 → 4
  - evaluate d → (lambda (x y) (+ (sq x) (sq y)))
  - apply (lambda (x y) …) to 3, 4

example cont'd

- apply (lambda (x y) (+ (sq x) (sq y))) to 3, 4
  - substitute 3 for x, 4 for y in (+ (sq x) (sq y))
  - evaluate (+ (sq 3) (sq 4))
    - evaluate (sq 3)
      - evaluate 3 → 3
      - evaluate sq → (lambda (x) (* x x))
      - apply (lambda (x) (* x x)) to 3
        - » substitute 3 for x in (* x x)
        - » evaluate (* 3 3)
        - » evaluate 3 → 3
        - » evaluate 3 → 3
        - » apply * to 3, 3 → 9

example cont’d 2

- apply (lambda (x y) (+ (sq x) (sq y))) to 3, 4
  - substitute 3 for x, 4 for y in (+ (sq x) (sq y))
  - evaluate (+ (sq 3) (sq 4))
    - evaluate (sq 3) → [many steps, previous slide] → 9
    - evaluate (sq 4)
      - evaluate 4 → 4
      - evaluate sq → (lambda (x) (* x x))
      - apply (lambda (x) (* x x)) to 4
        - » substitute 4 for x in (* x x)
        - » evaluate (* 4 4)
        - » evaluate 4 → 4
        - » evaluate 4 → 4
        - » apply * to 4, 4 → 16

example cont'd 3

- apply (lambda (x y) (+ (sq x) (sq y))) to 3, 4
  - substitute 3 for x, 4 for y in (+ (sq x) (sq y))
  - evaluate (+ (sq 3) (sq 4))
    - evaluate (sq 3) → [many steps, 2 slides back] → 9
    - evaluate (sq 4) → [many steps, previous slide] → 16
    - evaluate + → [primitive procedure +]
    - apply + to 9 and 16 → 25
- which is final result
- (d 3 4) → 25

Substitution Model

- gives precise model for evaluation
- can carry out mechanically
  - by you
  - by the computer
- will be basis of our understanding for now
- …will expand and revise later

make-addn's "signature"

```
(define (make-addn n)
    (lambda (x) (+ x n)))
```

- takes in a numeric argument n
- *returns a function*…
  - …which has, within it, a value "*pre-substituted*" for n.
- Notice: Standard substitution model holds!
  - *with one small clarification*...

Evaluating a function call

To evaluate a function call…

1. Evaluate the operands (arguments)
2. Evaluate the operator (function)
3. Apply the function to its arguments

To apply a function call…

Clarify

1. Replace the function argument variables with the values given in the call everywhere they occur
2. Evaluate the resulting expression

Clarify Substitution Model

1. Replace the function argument variable (*e.g.* n) with the value given in the call everywhere it occurs

…a.k.a. "deep substitution"

– happily plow through nested expressions, etc.

Except:

– Do **not** substitute for the variable inside any nested lambda expression that also uses the **same** variable as one of **its** arguments

The lambda shield

(expression involving x)

λ(x)

attempt to substitute value into x

- cannot substitute for x in body of this lambda expression
  - since x is an argument
- other substitutions will succeed

λ 

Example

- → apply (lambda (n) (lambda (n) (+ n n))) to 3
- → substitute 3 for n in (lambda (n) (+ n n))
- gives what?
  - (lambda (3) (+ 3 3)) ;; ??? nonsense!
  - (lambda (n) (+ 3 3)) ;; nope!
  - (lambda (n) (+ n n)) ;; correct!
- The *lambda shield* protects n argument from being substituted into

Examples

- apply (lambda (x) (+ x x)) to 3
  - → substitute 3 for x in (+ x x) (no shielding)
  - → (+ 3 3) → 6
- apply (lambda (x) (lambda (y) (+ x y))) to 3
  - → substitute 3 for x in (lambda (y) (+ x y))
  - → (lambda (y) (+ 3 y)) (shielding not needed)
- apply (lambda (x) (lambda (x) (+ x x))) to 3
  - → substitute 3 for x in (lambda (x) (+ x x))
  - → (lambda (x) (+ x x)) (x is shielded)

Automatic Renaming: (lambda (x) (lambda (x) (+ x x)))
→(lambda (x) (lambda (y) (+ y y)))
Called α-conversion in Lambda Calculus

Substitution Model

- gives precise model for evaluation
- can carry out mechanically
  - by you
  - by the computer
- β-reduction of Lambda-Calculus
- Shortcomings:
  - Implementation inefficiency
  - Cannot handle assignment and state (next time)

Function Values, revisited

(lambda (x) (* x x))

=> #<procedure>

What exactly is “#<procedure>”?

Free Variables in a Lambda Abstraction

```
(define make-addn
    (lambda (n)
        (lambda (x) (+ x n))))
```

What is the scope of n?

Variable n is *free* in the inner lambda function.

Evaluate **(make-addn 3)**
evaluate 3 → 3.
evaluate make-addn
→ (lambda (n) (lambda (x) (+ x n)))
apply make-addn to 3…
substitute 3 for n in (lambda (x) (+ x n))
→ (lambda (x) (+ x 3))

But substitution is expensive!

Free Variables in a Lambda, 2

```
>(define y 10)
10
>(define (f x) (+ x y))  ;; which y?
#procedure f
>(f 5)
15
```

```
>(set! y 15)
15
>(f 5)
??
```

- Dynamic scope: 20
- Static scope: 15

Original Lisp is Wrong

- Treat functions as *quoted values*. Leads to dynamic scope.

```
(define (map fun lis)
   (cond ((null lis) '())
           (else (cons (fun (car lis))  (map fun (cdr lis))))))
```

```
(define (prefix-first lis)
   (map  '(lambda (item) (list (car lis) item)) lis))
```

(prefix-first ‘(A B C)) →? ((A A) (A B) (A C))

(prefix-first ‘(A B C)) →? ((A A) (B B) (C C))

Upward Funarg Problem

[Environment: Id → Value]

```
(define (map fun lis)
  (cond ((null lis) '())
        (else (cons (fun (car lis))  (map fun (cdr lis))))))

 (define (prefix-first lis)
   (map  '(lambda (item) (list (car lis) item)) lis))
```

(prefix-first ‘(A B C))

→ (map '(lambda (item) (list (car lis) item)) ‘(A B C))

[lis→’(A B C)]

→(cons ('(lambda (item) (list (car lis) item)) (car *lis*)))

(map '(lambda (item) (list (car lis) item)) (cdr *lis*)))

[lis→’(B C)]

→... (cons ('(lambda (item) (list (car lis) item)) (car *lis*))) (map …)

Upward Funarg Problem

```
(define (map fun lis)
   (cond ((null lis) '())
            (else (cons (fun (car lis))  (map fun (cdr lis))))))
```

```
(define (prefix-first lis)
   (map  '(lambda (item) (list (car lis) item)) lis))
```

Captured

Free variable

- Follows *dynamic scope*

(prefix-first ‘(A B C)) →? ((A A) (B B) (C C))

Downward Funarg Problem of Lisp

```
(define make-addn
     (lambda (n)
              (lambda (x) (+ x n))))
```

Example:

```
(define (trap n)  (lambda (f) (f n)))
```

((trap 5) (make-addn 10))

→((trap 5) (lambda (x) (+ x *n*)))

[n→5]

→((lambda (f) (f n)) (lambda (x) (+ x n)))

[n→5; f→(lambda (x) (+ x n))]

→(f n)

→((lambda (x) (+ x n)) 5)

→ 10

;A Function evaluates itself.

Leads to dynamic scope

Scheme: Functions are Closures

- A function is evaluated to a closure.
- Closure = <fun-def, environment>

  fun-def =<parameters, body>

```
(define (prefix-first lis)
   (map (lambda (item) (list (car lis) item)) lis))
```

(prefix-first ‘(A B C))

[lis→’(A B C)]

→(map (lambda (item) (list (car lis) item)) lis)

→(map <*(lam (item) (list (car lis) item)), [lis→’(A B C)]*> lis)

→…

→((A A) (A B) A C)) ;; static scope

Functions are Closures

```
(define make-addn
    (lambda (n)
        (lambda (x) (+ x n))))
```

(make-addn 10)
→ #procedure ;<(lambda (x) (+ x n)), [n→10]> a closure

((trap 5) (make-addn 10))
→((trap 5) <(lambda (x) (+ x *n*)), [n→10]>); static scope
[n→5]
→((lambda (f) (f n)) <(lambda (x) (+ x *n*)), [n→10]>)
[n→5; f→<lambda (x) (+ x n)), [n→10]>]
→ (f n)
→15

Currying

Consider

- (define (f1 x y) (* x y))
- (f1 1 2)

value: 2

- (f1 1)

error: wrong number of arguments

Why not make *(f1 1)* meaningful? It is a *function* after all…

- *(define f2 (lambda (x) (lambda (y) (* x y))))*
- (f2 1)

value: compound procedure (in environment where x = 1)

- ((f2 1) 2)

value: 2

Currying Common Binary Functions

Currying is the process of reducing n-ary functions to n applications of functions of 1 argument

```
(define (curry bop)
    (lambda (x)  (lambda (y) (bop x y))))
```

```
> (((curry >) 5)  3)
 #t
```

- (map ((curry =) 10) ‘(5 10 20))
  (#f #t #f)

List funs. `filter`: select all the elements satisfying a given condition.

```
-> (define (filter p? l)
     (if (null? l)
       '()
       (if (p? (car l))
           (cons (car l) (filter p? (cdr l)))
           (filter p? (cdr l)))))
```

```
-> (filter (lambda (n) (> n 0)) '(1 2 -3 -4 5 6))
(1 2 5 6)
-> (filter (lambda (n) (<= n 0)) '(1 2 -3 -4 5 6))
(-3 -4)
-> filter ( (curry <) 0) '(1 2 -3 -4 5 6))
(1 2 5 6)
-> (filter ( (curry >=) 0) '(1 2 -3 -4 5 6))
(-3 -4)
```

Delayed Evaluation and Streams

Generating an infinite list of integers

How to Generate an Infinite List?

- Scheme uses call-by-value (eager evaluation), so the following code doesn't work.
  - *(define (intsfrom n) (cons n (intsfrom (+ n 1))))*
  - *(intsfrom 3) → (3 (intsfrom (+ 3 1)))*
    → (3 (4 (intsfrom 5))
    → …
    →

infinite loop

Thunk

- Use parameterless lambda function to delay evaluation.
- Stop evaluating arguments to "cons" as follows.

(define l (cons (lambda () 3) (lambda () (+ 3 1)))

→ ((lambda () 3) . (lambda () (+3 1))

((car l))
→ 3
((cdr l))
→ 4

Lazy cons

A macro-like definition

```
(define lazy-cons
    (lambda (Exp1 Exp2)
            (cons Exp1' Exp2'))
    )
)
```

Exp1' = (lambda () Exp1)

Lazy Lambda and Demand

- Define "demand" as follows
  - *(define (demand encapsulated)*
    (if (procedure? encapsulated) (encapsulated)
    encapsulated)
)

- Define "demandcar" and "demandcdr" as follows
  - *(define (demandcar l) (demand (car l)))*
  - *(define (demandcdr l) (demand (cdr l)))*

```
(define demandcar (compose demand car))
(define demandcdr (compose demand cdr))
```

(demandcar (cons (lambda () 3) (lambda () (+ 3 1))))
→3
(demandcdr (cons (lambda () 3) (lambda () (+ 3 1))))
→ 4

Generating an Infinite List

- So we can easily write a function to generate an infinite list of integers as follows.
  - *(define (intsfrom n)*

    (cons (lambda () n)

    (lambda () (intsfrom (+ n 1)))

    ))
  - *(demandcar (intsfrom 3))* → *3*
  - *(demandcar (demandcdr (intsfrom 3)))* → *4*

- Get the first m integers from an infinite list:

```
(define (firstn n lst)
    (if  (= n 0)  '()
         (cons (demandcar lst) (firstn (- n 1) (demandcdr lst)))))

(firstn 4 (intsfrom 3)) → (3 4 5 6)
```