Scheme

4190.310
Programming Languages
Spring 2014

Lecture 04

Reading assignments: Chapter 10

Scheme

- Scheme is a particularly elegant Lisp
- Interpreter runs a read-eval-print loop
- Things typed into the interpreter are evaluated (recursively) once
- Anything in parentheses is a function call (unless quoted)
- Parentheses are NOT just grouping, as they are in Algol-family languages
- Adding a level of parentheses changes meaning
  - (+ 3 4) ⇒ 7
  - ((+ 3 4))) ⇒ error
    - The ' ⇒' arrow means 'evaluates to'

S-expression

- Symbolic expression
- Used in Lisp
  - A notation for nested list

- Defined inductively
  - An atom
  - An expression of the form (x . y) where x and y are s-expressions

- (x y z)
- (x . (y . (z . ())))

- The first element of an S-expression is commonly a function and the remaining elements are arguments to the function
  - Prefix notation or Cambridge polish notation

An Overview of Scheme

- (+ 3 4) ⇒ 7
- 7 ⇒ 7
  - The number 7 is already fully evaluated

- (load “my_Scheme_program”) 스킴 함수를 불러옴
  - Input from a file

- ((+ 3 4))
  - a run-time error

Quote

- (quote (+ 3 4)) ⇒ (+ 3 4)
- '(+ 3 4) ⇒ (+ 3 4)

- Every expression has a type in Scheme
  - Generally not determined until run time
  - Predefined functions check dynamically to make sure that their arguments are appropriate types

  - (if (> a 0) (+ 2 3) (+ 2 "foo"))
    - 5 if a is positive
    - Otherwise a run-time type clash error

Predefined Type Predicates

- (boolean? x)
- (char? x)
- (string? x)
- (symbol? x)
- (number? x)
- (pair? x)
- (list? x)

Symbol in Scheme

- Comparable to what other languages call an identifier
  - Identifiers are permitted to contain a wide variety of punctuation marks
  - (symbol? 'x$_%:&=*!) ⇒ #t
- Boolean value
  - #t
  - #f

Functions

- (lambda (x) (* x x)) ⇒ function
  - A lambda expression does not give its function a name
- When a function is called, the language implementation restores the referencing environment that was in effect when the lambda expression was evaluated
  - Then, it augments this environment with bindings for the formal parameters and evaluates the expression of the function body in order
- ((lambda (x) (* x x)) 3) ⇒ 9

Functions (contd.)

- Standard functions
  - arithmetic
  - boolean operators
  - equivalence
  - list operators
  - symbol?
  - number?
  - complex?
  - real?
  - rational?
  - integer?
  - ...

Bindings

- Names can be bound to values by introducing a nested scope
- 'let' takes two arguments
  - The scope of the bindings produced by let is its second argument only
  - "All at once" semantics

```
(let ((a 3)
      (b 4)
      (square (lambda (x) (* x x)))
      (plus +))
  (sqrt (plus (square a) (square b)))) ⇒ 5.0
```

```
(let ((a 3)
  (let ((a 4)
        (b a))
    (+ a b))) ⇒ 7
```

Recursion

- let and letrec allow the user to create nested scopes

```
(letrec((fact
          (lambda (n)
            (if (= n 1) 1
              (* n (fact (- n 1)))))))
  (fact 5)) ⇒ 120
```

Conditional Expressions

- (if (< 2 3) 4 5) ⇒ 4
- (if #f 2 3) ⇒ 3
- (cond ((< 3 2) 1) ((< 4 3) 2) (else 3)) ⇒ 3

- 'if' checks to see whether the first argument evaluates to #t
  - If so, it returns the value of the second argument without evaluating the third
  - Otherwise, it returns the value of the third without evaluating the second
- Conditional expressions treat anything other than #f as true
- Scheme expressions are evaluated in applicative order in general

Creating a Global Binding of a Name

```
(define hypot
  (lambda (a b)
    (sqrt (+ (* a a) (* b b)))))
(hypot 3 4) ⇒ 5
```

Lists and Numbers

- car - returns the head of a list
  - (car '(2 3 4)) ⇒ 2
  - '(2 3 4) - a proper list (the final element is the empty list)
  - (car '(2)) ⇒ ()
- cdr - returns the rest of the list everything after the head
  - (cdr '(2 3 4)) ⇒ (3 4)
- cons - joins a head to the rest of the list
  - (cons 2 '(3 4)) ⇒ (2 3 4)
  - (cons 2 3) ⇒ (2 . 3) (improper list)

Equality Testing and Searching

- eq? - tests whether its arguments refer to the same object
  - (memq 'z '(x y z w)) ⇒ (z w)
  - (let ((p (lambda (x) x))) (eq? p (lambda (x) x))) ⇒ #f
  - (let ((p (lambda (x) x))) (eq? p p)) ⇒ #t
  - (eq? (lambda (x) x) (lambda (x) x)) ⇒ #f

- eqv? - tests whether its arguments are provably semantically equivalent
  - (memv '(z) '(x y (z) w)) ⇒ #f
  - (let ((p (lambda (x) x))) (eqv? p (lambda (x) x))) ⇒ #f
  - (eqv? (lambda (x) x) (lambda (x) x)) ⇒ #f

- equal? - tests whether its arguments have the same recursive structure
  - (member '(z) '(x y (z) w)) ⇒ ((z) w)
  - (let ((p (lambda (x) x))) (equal? p (lambda (x) x))) ⇒ #f
  - (equal? (lambda (x) x) (lambda (x) x)) ⇒ #f

Assignments and Sequencing

```
(let ((x 2))
     (set! x 3)
  x) ⇒ 3
```

```
(let ((l '(a b)))
     (set-car! l '(c d))
     (set-cdr! l '(e))
  l) ⇒ ((c d) e)
```

```
(begin
  (display "hi")
  (display "SNU"))
```

Iteration

```
(define iter-fib (lambda (n)
   (do (( i 0 (+ i 1))
        (a 0 b)
        (b 1 (+ a b)))
      ((= i n) b)
     (display b)
     (display " "))))
```

```
(for-each (lambda (a b) (display (* a b)) (newline))
  '(2 4 6) '(3 5 7))
```

Programs as Lists

- Homoiconic
  - Programs look like ordinary data structures, and can be created, modified, and executed on the fly

- eval function
  - Evaluates a list that has been created as a data structure
  - list returns a list consisting of its (evaluated) arguments

```
(define compose
  (lambda (f g)
   (lambda (x) (f (g x)))))
((compose car cdr) '(1 2 3)) ⇒ 2

(define compose2
  (lambda (f g)
    (eval (list 'lambda '(x) (list f (list g 'x)))
          (scheme-report-environment 5))))
((compose2 car cdr) '(1 2 3)) ⇒ 2
```

Evaluation Order

- Applicative order
  - What you're used to in imperative languages
  - Usually faster

- Normal order
  - Like call-by-name: don't evaluate arg until you need it
  - Sometimes faster
  - Terminates if anything will (Church-Rosser theorem)

```
(define double (lambda (x) (+ x x)))
```

Evaluation Order (contd.)

```
(double (* 3 4))
⇒ (double 12)
⇒ (+ 12 12)
⇒ 24
```

```
(double (* 3 4))
⇒ (+ (* 3 4) (* 3 4))
⇒ (+ 12 (* 3 4))
⇒ (+ 12 12)
⇒ 24
```

Evaluation Order (contd.)

```
(define switch (lambda (x a b c)
  (cond ((< x 0) a)
        ((= x 0) b)
        ((> x 0) c))))

(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (switch -1 3 (+ 2 3) (+ 3 4))
⇒ (switch -1 3 5 (+ 3 4))
⇒ (switch -1 3 5 7)
⇒ (cond ((< -1 0) 3)
        ((= -1 0) 5)
        ((> -1 0) 7))
⇒ (cond (#t 3)
        ((= -1 0) 5)
        ((> -1 0) 7))
⇒ 3
```

Evaluation Order (contd.)

```
(define switch (lambda (x a b c)
  (cond ((< x 0) a)
        ((= x 0) b)
        ((> x 0) c))))
```

```
(switch -1 (+ 1 2) (+ 2 3) (+ 3 4))
⇒ (cond ((< -1 0) (+ 1 2))
        ((= -1 0) (+ 2 3))
        ((> -1 0) (+ 3 4)))
⇒ (cond (#t (+ 1 2))
        ((= -1 0) (+ 2 3))
        ((> -1 0) (+ 3 4)))
⇒ (+ 1 2)
⇒ 3
```

Strictness

- Under applicative order evaluation, an unneeded subexpression may not terminate
- A strict language requires all arguments to be well-defined, so applicative order can be used
  - ML, Scheme
- A non-strict language does not require all arguments to be well-defined
  - It requires normal-order evaluation
  - Miranda, Heskell

Lazy Evaluation

- Lazy evaluation gives the best of both worlds
  - But not good in the presence of side effects
- Delay and force in Scheme

```
(define naturals
  (letrec ((next
    (lambda (n) (cons n (delay (next (+ n 1)))))))
    (next 1)))
(define head car)
(define tail (lambda (stream) (force (cdr stream))))

(head naturals) ⇒ 1
(head (tail naturals)) ⇒ 2
(head (tail (tail naturals))) ⇒ 3
```

Lazy Evaluation (contd.)

- Any attempt to evaluate the argument sets the value in the memo as a side effect
- (double (* 3 4)) will be compiled as (double (f))

```
(define double (lambda (x) (+ x x)))

(define f
  (lambda ()
    (let ((done #f)
          (memo '())
          (code (lambda () (* 3 4))))
      (if done memo
        (begin
          (set! memo (code))
          (set! done #t)
          memo)))))
...
(double (f)) ⇒ (+ (f) (f))
    ⇒ (+ 12 (f)) ⇒ (+ 12 12) ⇒ 24
```

Streams

- Source of side effects
  - read will generally return a different value every time it is called
  - multiple calls to display must occur in the proper order if the program is to be considered correct

- One way to avoid these side effects is to model input and output as streams
  - Unbounded-length lists whose elements are generated lazily

```
(define output (my_prog input))

(define driver (lambda (s)
  (if (null? s) '()
    (display (car s))
    (driver (cdr s)))))
(driver output)
```

Streams (contd.)

- If Scheme employed lazy evaluation of input and output streams, prompts, inputs, and outputs would be interleaved naturally in time

```
(define squares (lambda (s)
  (cons "please enter a number\n"
        (let ((n (car s)))
          (if (eof-object? n) '()
            (cons (* n n)
                  (cons #\newline (squres (cdr s)))))))))

(define output (squares input)))

(car output)
(cadr output)
(caddr output)
...
```

Higher-Order Functions

- Take a function as argument, or return a function as a result
  - Great for building things

```
(map * '(2 4 6) '(3 5 7)) ⇒ (6 20 42))

(define fold (lambda (f i l)
  (if (null? l) i
    (f (car l) (fold f i (cdr l))))))

(fold + 0 '(1 2 3 4 5))
(fold * 1 '(1 2 3 4 5))
```

Currying

- After Haskell Curry, the same guy Haskell is named after
- Replacing a multi-argument function with a function that take a single argument and returns a function that expects the remaining arguments
- ML, Miranda, and Haskell have especially nice syntax for curried functions

```
(define curried-plus
        (lambda (a) (lambda (b) (+ a b))))

((curried-plus 3) 4)

(define plus-3 (curried-plus 3))

(plus-3 4)
```

Currying (contd.)

```
(define curry
        (lambda (f)
          (lambda (a)
            (lambda (b) (+ a b)))))

(((curry +) 3) 4)

(define curried-plus (curry +))
```