Programming Languages

Tevfik Koşar

Lecture - XIV

March 7th, 2006

Quiz - 2

1) Consider the following Scheme definition:

```
>> (define (test x y) (if (= x 3) 1 y))
```

Write the output of the following call to this function

```
>> (test 3 (/ 5 0))
```

a) assuming normal-order evaluation is used

b) assuming applicative-order evaluation is used

Quiz - 2

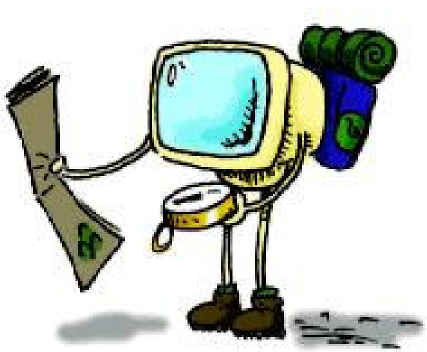
2) Write the output of the following programs:

a) >>
```
(let  ((a 3) (b 4))
      (let ((a 5) (b a))
            (let ((a b) (c a))
                  (+ a b c)) ) )
```

b) >>
```
(let  ((a 3) (b 4))
      (let* ((a 5) (b a))
            (let* ((a b) (c a))
                  (+ a b c)) ) )
```

Roadmap

- Scheme
  - List Operations
  - Assignment
  - Loops
  - Evaluation Order in Scheme
- Lambda Calculus
- Comments on Midterm Exam



List Operations

Scheme provides a variety of procedures for operating on lists:

- length takes one argument (a list) and returns an integer giving the length of the list.

```
>> (length '(0 #t #f))
3
```

- list takes one or more arguments and constructs a list of those items.

```
>> (list 1)                ; == (cons 1 `())
(1)
>> (list 1 2 3)
(1 2 3)
```

List Operations

- append takes two or more lists and constructs a new list with all of their elements.

```
>> (append '(1 2) '(3 4))
(1 2 3 4)
```

➔Notice that this is different from what list does:

```
>>(list '(1 2) '(3 4))
((1 2) (3 4))

>> (append '((1 2) (3 4)) '((5 6) (7 8)))
((1 2) (3 4) (5 6) (7 8))
```

List Operations

- reverse takes one list, and returns a new list with the same elements in the opposite order.

```
>> reverse '(1 2 3 4))
(4 3 2 1)
```

Assignment

- set!
- set-car!
- set-cdr!

```
>> (let ((x 2)
        (l `(a b)))
        (set x! 3)
        (set-car! l `(c d))
        (set-cdr! l `(e))
>> x
3
>>l
((c d) e)
```

Loops

```
>> (do ((i 0 (+ i 1))          -> initialize & increment
        ( a 0 b)               ->    “              “
        (b 1 (+ a b)))         ->    “              “
        ((= i n)) b)           -> termination test
      (display b)              -> body of the loop
      (display “ ”))))         ->    “              “
```

Loops

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

6
20
42
()
```

Evaluation Order in Scheme

- Scheme uses applicative-order evaluation
  - Evaluate arguments before passing them to subroutine

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

Applicative order evaluation (as in Scheme):

```
>> (double (* 3 4))
➔(double 12)
➔(+ 12 12)
➔24
```

Evaluation Order in Scheme

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

Normal order evaluation:

```
>> (double (* 3 4))
➔(+ (* 3 4) (* 3 4))
➔(+ 12 (* 3 4))
➔(+ 12 12)
➔24
```

➔Normal order causes us to evaluate (* 3 4) twice

Evaluation Order in Scheme

- In specific cases, the outcome can be different.

```
Eg. (define switch (lamda x a b c)
            (cond       ((< x 0) a)
                        ((= x 0) b)
                        ((> x 0) c))))
```

Using applicative order:

```
>> (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  ((< x 0) 3)
          ((= x 0) 5)
          ((> x 0) 7))
➔  (cond  (#t 3)
          ((= x 0) 5)
          ((> x 0) 7))
```

Evaluation Order in Scheme

Using normal order:

```
>> (switch -1 (+ 1 2) (+ 2 3) (+ 3 4))

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

➔ Normal order avoids evaluating (+ 2 3) and (+ 3 4)

Comments in Scheme

You can and should put comments in your Scheme programs. Start a comment with a semicolon. Scheme will ignore any characters after that on a line. (This is like the // comments in C++.)

```
>> (define foo 22) ; define foo with an initial value of 22
```

Lambda Calculus

- **lambda calculus** is a formal system designed to investigate function definition, function application, and recursion.
- was introduced in 1930's
- the smallest universal programming language
- lambda calculus consists of a single transformation rule (variable substitution) and a single function definition scheme
- lambda calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism

Lambda Calculus

- In lambda calculus, every expression stands for a function with a single argument
- the argument of the function is in turn a function with a single argument
- and the value of the function is another function with a single argument

Eg.

$f(x) = x + 2$ would be expressed as $\lambda\, x.\ x + 2$

and the number $f(3)$ would be written as $(\lambda\, x.\ x + 2)\ 3$

Lambda Calculus

- A function of two variables is expressed in lambda calculus as a function of one argument which returns a function of one argument

Eg. the function $f(x, y) = x - y$ would be written as:

$\lambda\, x.\, \lambda\, y.\, x - y.$

A common convention is to abbreviate curried functions as, for instance, $\lambda\, x\, y.\, x - y.$

Example:

$$(\lambda\, x\, y.\, x - y)\ 7\ 2$$
$$= (\lambda\, y.\, 7 - y)\ 2$$
$$= 7 - 2$$
$$= 5$$

Lambda Calculus

- Consider a function which takes another function as argument and applies it to the argument 3: $\lambda f.\ f\ 3$.

- This latter function could be applied to our earlier "add-two" function as follows: $(\lambda f.\ f\ 3)\ (\lambda x.\ x+2)$.

➔ $(\lambda f.\ f\ 3)\ (\lambda x.\ x + 2)$

$= (\lambda x.\ x + 2)\ 3$

$= 3 + 2$

$= 5$

Lambda Calculus

Formal definition:

The set of all lambda expressions can then be described by the following context-free grammar in BNF:

expr → identifier
expr → λ identifier . expr
expr → expr expr

Lambda Calculus

Mapping Lambda Calculus to Scheme:

Math: $f(x) = x + 2$

Lambda Calculus: $\lambda\, x.\ x + 2$

Scheme: (define f (lambda (x) (+ x 2)))

Math: $f(x, y) = x - y$

Lambda Calculus: $\lambda\, x.\ \lambda\, y.\ x - y$

Scheme: (define f (lambda (x y) (- x y)))

Midterm will cover:

- Ch 1.1-1.6
- Ch 2.1-2.3
- Ch 3.1-3.3
- Ch 4.1-4.6
- Ch 6.1-6.7
- Ch 10.1-10.5
- Scheme

- Check the class notes! Anything not covered in the class will not be asked!
- Expect similar questions to the homework assignments and quizzes.

Midterm Hints

- Be sure that you know:
  - difference between compilation and interpretation; lexical, syntactic and semantic analysis; linking and binding
  - how to write regular expressions
  - how to generate NFA and DFA from regular expressions
  - how to generate parse trees
  - difference between top-down and bottom-up parsing; LL vs LR grammars
  - difference between stack-based vs heap-based allocation; static vs dynamic scoping
  - how to generate attribute grammars; decorate parse trees; and generate syntax trees
  - expression evaluation orders; applicative vs normal-order evaluation
  - basic scheme functions; lists, searching, and scoping in Scheme