Principles of Programming Languages
Lecture 10

Wael Aboulsaadat

wael@cs.toronto.edu

http://portal.utoronto.ca/

Acknowledgment: parts of these slides are based on material by Diane Horton & Eric Joanis @ UoT

References: Scheme by Dybvig
PL Concepts and Constructs by Sethi
Concepts of PL by Sebesta
ML for the Working Prog. By Paulson
Prog. in Prolog by Clocksin and Mellish
PL Pragmatics by Scott

Scheme: Functions

- **The general form of a Scheme function is**

  (define (<name> <formal parameters>) (<body>))

- **Examples:**
  - (define (square x)
    (* x x))
  - (define (sum-of-squares x y)
    (+ (square x) (square y)))
  - (define (circle-area rad)
    (* 3.14 (square rad)))

Scheme: Functions

- **Scheme functions are first-class objects:**
  - Can be created dynamically
  - Can be stored in data structures,
  - Can be returned as results of expressions or procedure.

This means that a scheme program can evolve its behavior as it runs!

Scheme: Pure Functions

- **A pure function is one that simply uses its input arguments to compute a return value, without performing any side-effects….?**
  - Side-effects are changes to the system's computational state that could affect future calls to itself or other functions.

- **In a language that only uses pure functions, any function call with instantiated arguments, e.g. (f 5 10), ALWAYS returns the same value and hence *means the same thing* in the context of a particular program.**

- **The absence of side effects makes it much easier to *formally analyze* the behavior of a system, since:**
  - We can reason about the system in terms of independent function calls, without having to worry about the (side) effects of these calls on future calls.
  - We can simplify the code accordingly.
    - E.g. **(+ (f 5 10) (f 5 10))** simplifies to **(* 2 (f 5 10))** for any numeric function f

Scheme: Pure Functions – cont'd

- **Referential Transparency: syntactically identical expressions mean the same thing, (i.e. return the same result when evaluated) regardless of WHERE they appear in a program.**

- **Manifest Interface Principle (of Programming Languages): All interfaces should be apparent (manifest) in the syntax.**

- **When all functions are pure, referential transparency and the manifest interface principle are upheld, and thus:**
  - Programs are much easier to formally analyze
  - Programs are much easier to DEBUG!!!
    - You can understand programs by just looking at the static source code. You need not think about the underlying computational states and how they are affected by program dynamics.

What you see is what you get!

Scheme: To assign is evil…

- **When an assignment statement is applied to variables (i.e. memory locations) that**

  a) will be maintained AFTER the function call is completed,

  b) will be used for their values during later function calls (to the same or other functions),

  It *violates referential transparency* and destroys one's ability to statically analyze source code (both formally or intuitively).

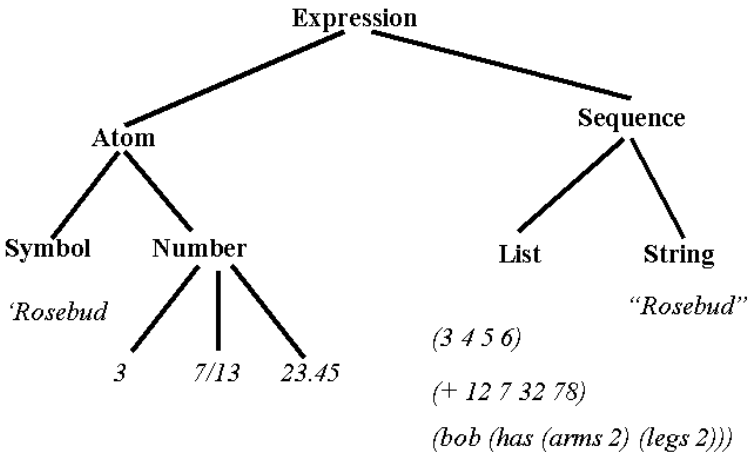
- **Example:**

```
(define g 10)          ;; define a global variable, g.
(define (f a)          ;; define function f, with one argument, a.
    (set! g (* g g))   ;; Scheme's assignment operator, meaning g = g*g
    (+ a g))
]=> (f 7)
107
]=> (f  7)
10007                  ;; BADDD ☹
```

Scheme: Predicate Functions

- **Function that return #t (true) or #f (false)**
  - Some scheme interpreters use empty list () to indicate #f

- **Predefined functions:**
  - (= …) ; comparison for numb
  - (> …) ; also (< ..)
  - (and …) ; also (or …) (not….)
  - (negative? …)
  - (number? …) ;
  - (symbol? …) ;
  - (zero? …) ;
  - (string? …) ;
  - (boolean? …) ;
  - (list? …) ;
  - (null? …) ;
  - (char? …) ;



Scheme: Predicate Functions cont'd

- **eqv?** *obj1 obj2* ***(eq?....)***
  - returns #t if *obj1* and *obj2* should normally be regarded as same
  - returns #t if
    obj1 and *obj2* are both numbers, are numerically equal
    (eqv? 2 2)

    obj1 and *obj2* are both characters and are the same character according to the char? procedure
    (eqv? 'a 'a)

    both *obj1* and *obj2* are the empty list.
    (eqv? '() '())

    obj1 and *obj2* are pairs, vectors, or strings that denote the same locations in memory
    (eqv? (cons 1 2) (cons 1 2)) ==> #f

Scheme: Predicate Functions cont'd

- **equal?** *obj1 obj2*
  - Equal? recursively compares the contents of pairs, vectors, and strings, applying eqv? on other objects such as numbers and symbols.
  - A rule of thumb is that objects are generally equal? if they print the same.
  - Equal? may fail to terminate if its arguments are circular data structures.

```
(equal? 'a 'a)                  ==> #t
(equal? '(a) '(a))              ==> #t
(equal? '(a (b) c) '(a (b) c))  ==> #t
(equal? "abc" "abc")            ==> #t
(equal? 2 2)                    ==> #t
```

Scheme: Let

- **Allows the definition of local variable bindings.**

- **The general form of a let expression is:**

(let ((<var_1> <exp_1>)
(<var_2> <exp_2>)
……
(<var_n> <exp_n>))
<body>)

The expressions <exp_i> are all evaluated, the <body> is evaluated with each <var_i>in <body> bound to the value obtained from evaluating each <exp_i>

Scheme: Let

- **Allows the definition of local variable bindings.**

- **Example:** simple quadratic solver

  $ax^2 + bx + c = 0$, $x = (-b \; a \; sqrt(b^2 - 4ac)) / 2a$

```
(define (quadratic-solutions a b c )
        (display (/ (+ (- 0 b) sqrt(-(square b) (* 4 a c ) ) ) ) (* 2 a)))
        (display (/ (- (- 0 b) sqrt(-(square b) (* 4 a c ) ) ) ) (* 2 a))))

(define (quadratic-solutions a b c )
        (let ((root-part ( sqrt(- (square b) (* 4 a c )  )  )  ) )
        (display (/ (+ (- 0 b) root-part)  (* 2 a)))
        (display (/ (-  (- 0 b) root-part)  (* 2 a ) ) ) ) )
```

Scheme: Let, Let* & Scope

- **Let vs. Let***

```
(let ((a-structure (some-procedure))
      (a-substructure (get-some-subpart a-structure))
      (a-subsubstructure (get-another-subpart a-substructure)))
 +-------------------------------------------------------------------+
 | (foo a-substructure)          ; scope of all three variables | )))
 +-------------------------------------------------------------------+


(let* ((a-structure (some-procedure))
 +-------------------------------------------------------------------+
 |     (a-substructure (get-some-subpart a-structure))               |
 +-------------------------------------------------------------------+  |
 |     (a-subsubstructure (get-another-subpart a-substructure)))  |  |
 +-------------------------------------------------------------+  |  |
 | (foo a-subsubstructure)                                     |  |  | )))
 +-------------------------------------------------------------+--+--+
```

  - Each initial value clause is in the scope of the previous variable in the let*.
  - From the nesting of the boxes, we can see that bindings become visible one at a time, so that the value of a binding can be used in computing the initial value of the next binding.

Scheme: Selection Statements

- **A Conditional expression are of the form:**

(cond (<p_1> <e_1>)
(<p_2> <e_2>)
(<p_m> <e_m>)
(else <e_n>))

  - *cond* is a built in primitive
  - p_i is a predicate (truth function) that evaluates to either #t or #f
  - Each predicate expression is evaluated in the order it appears. As soon as one is found to be true, the corresponding expression (e_i) is evaluated and returned as the result of the *cond* expression.
  - *cond* is like nested if then else statements

Scheme: Selection Statements

- **if Expression:**
  - Can be used when there are a maximum of 2 cases

    (if <predicate> <consequent> <alternative>)
    (if <predicate> <consequent>)

  - Example:
    - (if (< x 0)
      (- x)
      x)

    - (define (zerocheck? x)
      (if (= x 0)
      #t
      #f))

Scheme: higher order func - map

- **A higher order function used to apply *another function* to every element of a list:**

  (map <func> <arg-list>)

  – Arguments: a function and arguments lists

  – <func> must be a function taking as many arguments as there are in <arg-list> and returning a single value

- **Examples:**

```
]=> (map + '( 1 2 3) '( 4 5 6 ) )
(5 7 9)

]=>(map abs '(-1 2 -3 -4))
(1 2 3 4)
```

Scheme: I/O

- **Input/output**
  - (read …) ;reads and returns an expression
  - (read-char …) ;reads and returns a character
  - (peek-char …) ; returns next available character w/o updating
  - (char-ready? …) ; returns #t if char has been entered
  - (write-char …) ; outputs a single character
  - (write OBJECT …) ; outputs the object (strings are in quotes,.)
  - (display OBJECT…) ; outputs the object in a more readable form
  - (newline) ; outputs end-of-line

- **Example:**

```
> (display “hello world”)
> (define r (read))
> (display r)
```

Scheme Practice 1

Write a function member? which takes a list and an element and returns true if the a element is a member of the list.

```
(define (member? x list)
    (if (null? list)
          #f
       (if (equal? x (car list))
          #t
          (member? x (cdr list)))))
```

Scheme Practice 2

Write a function append which takes 2 lists and append one to the other

```
(define (append list1 list2)
        (if (null? list1)
            list2
            (cons (car list1) (append (cdr list1) list2))))
```

Scheme Practice 3

Write a function dup? which returns true if there are duplicate in a passed list. Assume function (member? x L) is available and returns #t if and only if x is a member of list L.

```
(define (dup? lst)
        (cond    ((null? lst)
                        #f )
                 ((member?
                        (car lst) (cdr lst)) #t)
                 (else
                        (dup? (cdr lst) ) )
```