Principles of Programming Languages III

Wael Aboelsaadat
wael@cs.toronto.edu
http://www.cs.toronto.edu/~wael/324.html

Today

• Scheme
  – Functions
  – Let & Scope
  – Recursion
  – Debugging & Style

Scheme: functions

• The general form of a Scheme function is
  
  (define (<name> <formal parameters>) (<body>))

• What’s the difference between a function and a procedure?
  – The term procedure is used in Scheme to describe what are actually non
    side-effecting functions
  – Imperative languages like C/C++/Pascal use the term function generically
    to describe side-effecting procedures

• 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 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.

• 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:
  – Once we know the local behaviors of functions, 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 identify identical functional calls as equivalent objects and simplify
    the code accordingly.
    – E.g. (f 5 10) (f 5 10) simplifies to (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 bad...

• 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 a))) ; ; Scheme’s assignment operator, meaning g = g + a
  
  > (f 7)      ; > (f 7)
  107
  10007 ; BADDDD

What you see is what you get!
Scheme: predicate functions

- Function that return #t (true) or #f (false)
  - Note: some scheme interpreters use empty list () to indicate #f
- Predifined functions:
  - (= ...) ; comparison for numbers
  - (>) ... ; also (< ...)
  - (and ...) ; also (or ...) (not ...)
  - (negative?)
  - (number?)
  - (symbol?)
  - (zero?)
  - (string?)
  - (boolean ?)
  - (list?)
  - (null?)
  - (char?)
  - (eqv ?)
    - An equivalence predicate is the computational analogue of a mathematical equivalence relation (it is symmetric, reflexive, and transitive)
  - (eq? ...)
    - the finest or most discriminating
  - (equal? ...)
    - the coarsest

Scheme: anonymous functions

- Anonymous functions are defined using lambda expressions:
  - (lambda (<formal parameters>) (<body>))
- Examples:
  - ((lambda (symbol) (eq? 'joe symbol)) 'fed)
  - ((lambda (x) (* x x)) 3)
- A lambda expression can be used to construct a procedure to be used as an operator
  - E.g.:
  - (lambda (x y z) (+ x y z))
- We could use a lambda expression to construct an anonymous function and then later on, we can bind a name to that function if we want.

Scheme: selection statements

- A Conditional expression are of the form:
  - (cond (<p1> <e1>)
    (<p2> <e2>)
    (xm <em>)
    (else <en>))
  - cond is a built-in primitive
  - pi 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 (ei) is evaluated and returned as the result of the cond expression.
  - cond is like nested if then else 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: let

- Allows the definition of a bunch of local bindings
- The general form of a let expression is:
  - (let ( <var1> <exp1> ) ( <var2> <exp2> ) ... ( <var> <exp> ) <body>)
  - The expressions <expi> are all evaluated, the <body> is evaluated with each <var> bound to the value obtained from evaluating each <expi>
- Example:
  - simple quadratic solver:
  - ax2 + bx + c = 0, x = (-b ± sqrt(b2 - 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 ) )

Scheme: let, scope & symbol table

- Scope:
  - the textual region of the program in which a specific set of variable are active is called the scope.
  - Symbol table
  - Scheme is a block-structured language with nested scopes:
    - You can declare local variables whose scope is a block of code, and blocks can have blocks inside them with their own local variables.
Scheme: let, let* & scope

- **Scope**: the textual region of the program in which a specific set of variable are active is called the scope.

- **Scheme** is a block-structured language with nested scopes:
  - You can declare local variables whose scope is a block of code, and blocks can have blocks inside them with their own local variables.
  - `let` and `let*` bind local variables to a certain value.

  - **let vs. let**
    - 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.

  - **letrec** is similar to `let`:
    - Assign values to variables that are visible within the current scope

  - **Syntax**: `define ( (var var1 val1) (var var2 val2) ... ) exp1 exp2 ...`  

  - **Example**:
    ```scheme
    => (define (sum (lambda (lst)
      (if (null? lst) 0
        (+ (car lst) (sum (cdr lst)) ))) )
    (sum '(1 2 3 4 5)))
    15
    ```

Scheme: scope - cont’d

- **letrec** is similar to `let`:
  - Assign values to variables that are visible within the current scope

  - **Syntax**: `define ( (var var1 val1) (var var2 val2) ... ) exp1 exp2 ...`

  - **Example**:
    ```scheme
    => (define (odd? (lambda (x)
      (if (zero? x) #f
        (even? (- x 1)) )))
    (even? (lambda (x)
      (if (zero? x) #t
        (odd? (- x 1)) )))
    (even? 10) )
    Error!
    => (define (odd? (lambda (x)
      (if (zero? x) #f
        (even? (- x 1)) )))
    (even? (lambda (x)
      (if (zero? x) #t
        (odd? (- x 1)) )))
    (even? 10) )
    #t
    ```

- **define v.s. let**
  - Mutually recursive functions can’t be defined using `let`
  - **Example**:
    ```scheme
    => (define (even? x)
      (if (zero? x) #t
        (odd? (- x 1)))
    (odd? x))
    => (even? 10) )
    #t
    ```

Scheme: recursion

- **Recursion** is the preferred way to do computation in functional languages.

  - **Did the right hand draw the left hand first?**
  - **Or did the left hand draw the right hand that draws the left hand?**
  - This famous art work by M.C. Escher is an example of recursion

Scheme: how to approach recursion?

1. **Strategy**
   - How to reduce the problem?
2. **Header**
   - What info needed as input and output?
   - Write the function header
   - Use a noun phrase for the function name
3. **Spec**
   - Write a method specification in terms of the parameters and return value.
   - Include preconditions
4. **Base cases**
   1. When is the answer so simple that we know it without recursing?
   2. What is the answer in these base cases(s)?
   3. Write code for the base case(s)
5. **Recursive Cases**
   1. Describe the answer in the other case(s) in terms of the answer on smaller inputs
   2. Simplify if possible
   3. Write code for the recursive case(s)
Scheme: recursive example

- Consider a function named `Power` which calculates the result of raising an integer to a positive power. If `X` is an integer and `N` is a positive integer, the formula for `X^N` is:
  \[ X^N = X \times X \times X \times \ldots \times X \]
  \[ \text{N-times} \]

- We could also write this formula as:
  \[ X^N = X^{N-1} \times X \]

- Or even as:
  \[ X^N = X^{N-2} \times (N-1) \times X \]

Scheme: recursive example - cont’d

- In fact we can write the formula:
  \[ X^N = X^{N-1} \times X \]

- This definition of `X^N` is a classic recursive definition, a definition given in terms of a smaller version of itself.

- When does the process stop?
  - When we have reached a case where we know the answer without resorting to a recursive definition.
  - Base case: `N` is 1 \( \rightarrow X^1 = X \)

- Pseudocode:

```
If n is 1
   Return x
Else
   Return x * Power(x, n - 1)
```

Scheme: recursive example2

- Consider the factorial of a number `N` (written `N!`) is `N` multiplied by `N-1`, `N-2`, `N-3` and so on. Hence, another way of expressing factorial is:
  \[ N! = N \times (N-1)! \]

- When does the process stop?
  - `0! = 1`

- Pseudocode:

```
Factorial(In: n)
If n is 0
   Return 1
Else
   Return n * Factorial(n - 1)
```

Scheme: recursion types

- **Terminology**
  - Items in a list at nesting level 1 are top-level items
  - E.g. the top-level items in the list `(a b c) (d e f)` are `a`, `(b c)` and `(d e f)`
  - What are the top level items in `((a b (c d)) e (f g) h)`?

- **Flat Recursion**
  - If the recursion is applied over the top items of a list.
  - E.g.

```
(=> define (append lst1 lst2)
   (if (null? lst1) ; base case
      lst2
      (cons (car lst1) (append (cdr lst1) lst2))) ; recursive case

(=> append' (a b c) (d e f) (a b c d e f))
```

```
;; Scheme recursive example2
;; cont'd
```

```
;; Scheme: recursive example

;; Consider a function named Power which calculates the result of raising an integer to a positive power. If X is an integer and N is a positive integer, the formula for X^N is:
;; \[ X^N = X \times X \times X \times \ldots \times X \]
;; \[ \text{N-times} \]

;; We could also write this formula as:
;; \[ X^N = X^{N-1} \times X \]

;; Or even as:
;; \[ X^N = X^{N-2} \times (N-1) \times X \]

;; Scheme: recursive example - cont’d

;; In fact we can write the formula:
;; \[ X^N = X^{N-1} \times X \]

;; This definition of `X^N` is a classic recursive definition, a definition given in terms of a smaller version of itself.

;; When does the process stop?
;; - When we have reached a case where we know the answer without resorting to a recursive definition.
;; - Base case: `N` is 1 \( \rightarrow X^1 = X \)

;; Pseudocode:

```
If n is 1
   Return x
Else
   Return x * Power(x, n - 1)
```

;; Scheme: recursive example2

;; Consider the factorial of a number `N` (written `N!`) is `N` multiplied by `N-1`, `N-2`, `N-3` and so on. Hence, another way of expressing factorial is:
;; \[ N! = N \times (N-1)! \]

;; When does the process stop?
;; - `0! = 1`

;; Pseudocode:

```
Factorial(In: n)
If n is 0
   Return 1
Else
   Return n * Factorial(n - 1)
```

;; Scheme: recursion types

;; Terminology
;; - Items in a list at nesting level 1 are top-level items
;; - E.g. the top-level items in the list `(a b c) (d e f)` are `a`, `(b c)` and `(d e f)`

;; Flat Recursion:
;; - If the recursion is applied over the top items of a list.
;; - E.g.
```
(=> define (append lst1 lst2)
   (if (null? lst1) ; base case
      lst2
      (cons (car lst1) (append (cdr lst1) lst2))) ; recursive case

(=> append' (a b c) (d e f) (a b c d e f))
```
Scheme: recursion types

- Deep (tree) Recursion:
  - If the recursion is applied over all the items of a list.
  - E.g.:
    ```scheme
    (define (count-leaves lst)
      (cond ((null? lst) 0)
            ((atom? lst) 1)
            (else (+ (count-leaves (car lst))
                   (count-leaves (cdr lst))))))
    (count-leaves '(a (b c (d e)))))
    5
    ```

- Tail Recursion:
  - When the last thing a function does is to call itself.
  - When semantically equivalent to iteration, tail-recursive programs can be compiled as efficiently as iterative programs.
  - E.g.:
    ```scheme
    (define fact (lambda (n)
                   (if (= n 0)
                       1
                       (* n (fact (-n 1))))))
    (fact 5)
    120
    ```

Scheme: function as an argument

- Functions are passed as any other argument
  - E.g.:
    ```scheme
    (define foobar (cons 'foo 'bar))
    foobar
    (foo bar)
    (a b c d e) and (a . (b . (c . (d . (e . ()))))) are equivalent notations of a list of symbols
    ```

- Example of variable num parameters using pairs
  ```scheme
  (define func (lambda (param1 param2 . varparam)
                (display varparam)))
  (func '(1) '(2) '(a b c) '(x y) '(z))
  (func '(1) '(2))
  ```

Scheme: function as an argument - cont’d

- Using apply function:
  - Applies a function to each element in a list
  - Syntax: `(apply function-name (arg1 arg2 ...))`
  - The arguments must be in a list, even if it is one argument
    ```scheme
    (define compose (lambda (func1 func2)
                           (lambda args
                                      (apply func1 (list (apply func2 args))))))
    (compose sqrt *) 12 75
    30
    ```

- Some functions can be called at different times with different numbers of arguments.

- Pair notation:
  - Pairs are created by the procedure cons
  ```scheme
  (cons 'foo 'bar)
  ```
  - (a b c d e) and (a . (b . (c . (d . (e . ()))))) are equivalent notations of a list of symbols

- Example of variable num parameters using pairs
  ```scheme
  (define func (lambda (param1 param2 . varparam)
                (display varparam)))
  (func '(1) '(2) '(a b c) '(x y) '(z))
  (func '(1) '(2))
  ```

Scheme: function – variable arguments

- A higher order function used to apply another function to every element of a list:
  ```scheme
  (map <func> <arg-list>)
  ```
  - Arguments: a function and an arg-list
  - The dynamic order in which func is applied to the elements of the arg-list is unspecified.

- Examples:
  ```scheme
  (map + '(1 2 3) '(4 5 6))
  (7 7 7)
  ```
  ```scheme
  (map (lambda (x y) (+ x y)) '(1 2 3) '(4 5 6))
  (5 7 9)
  ```
  ```scheme
  (map (lambda (x) (+ 1 x)) '(1 2 3))
  (2 3 4)
  ```
  ```scheme
  (map abs '(-1 2 -3 4))
  (1 2 3 4)
  ```
Scheme: higher order func - reduce

- A high order function (we have already seen map)

- Syntax:
  
  \( \text{reduce \ func \ initial-list} \)

  - Reduce a list of values to a single value by repeatedly applying a binary function to the list of values

- Implementation:
  
  \[
  \text{define (reduce \ func \ initial-list)}
  \]

  \[
  \text{id}
  \]

  \[
  \text{if \ (null? \ list) \ id \ (func \ (car \ list) \ (reduce \ func \ (cdr \ list) \ id))}
  \]

- Example:
  
  \[
  \text{>\> (reduce \+ \ '(1 \ 2 \ 3 \ 4 \ 5) \ 0) ;15}
  \]

  \[
  \text{>\> (reduce \* \ '(1 \ 2 \ 4 \ 6 \ 8 \ 10) \ 1) ;3840}
  \]

  \[
  \text{>\> (reduce \ append \ '((1 \ 2 \ 3) \ (4 \ 5 \ 6) \ (7 \ 8)) \ ( )) ;(1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8)}
  \]

  \[
  \text{>\> (reduce \ expt \ '(2 \ 2 \ 2 \ 2) \ 1) ;65536}
  \]

Scheme: debugging

- \( \text{bkpt \ datum \ argument} \)
  - Sets a breakpoint. When the breakpoint is encountered, datum and the arguments are typed (just as for error) and a read-eval-print loop is entered.

  \[
  \text{>\> (bkpt \ test-2 \ test-3) \=> \ test-2 \ test-3}
  \]

- \( \text{to continue, call \text{RESTART} with an option number:} \)

  \[
  \text{\text{RESTART} 2 \Rightarrow \ Return \ from \ BKPT.}
  \]

  \[
  \text{\text{RESTART} 1 \Rightarrow \ Return \ to \ read-eval-print \ level \ 1.}
  \]

- Examples:

  \[
  \text{\text{RESTART} 2 \Rightarrow \ Return \ from \ BKPT.}
  \]

  \[
  \text{\text{RESTART} 1 \Rightarrow \ Return \ to \ read-eval-print \ level \ 1.}
  \]

Scheme: debugging - cont’d

- \( \text{go \ object \ [output-port \ [as-code?] \]} \)
  - Prints the source code of a given procedure.

- \( \text{go \ procedure} \)
  - Prints the arguments of a given procedure

- \( \text{apropos \ string} \)
  - Search an environment for bound names containing string and print out the matching bound names.

Scheme: expressions and short circuit evaluation

- \( (\text{and} \ ... \ ) \)

  - E.g.

  \[
  \text{if \ (and \ (try \ first-things) \ (try \ second-things))} \)

  \[
  \text{if \ (and \ (try \ first-things) \ (try \ second-things)) \}
  \]

  - If the three calls all return true values #t

  - If any of them returns #f, however, none of the rest are evaluated, and #f is returned as the value of the overall expression.

- \( (\text{or} \ ... \ ) \)

  - E.g.

  \[
  \text{if \ (or \ (try \ first-things) \ (try \ second-things))} \)

  - Likewise, it stops when it gets a true value

Scheme: debugging - cont’d

- \( \text{pp \ object \ [output-port \ [as-code?] \]} \)
  - Prints the source code of a given procedure.

- \( \text{pa \ procedure} \)
  - Prints the arguments of a given procedure

- \( \text{apropos \ string} \)
  - Search an environment for bound names containing string and print out the matching bound names.

Scheme: programming style

- Use special suffixes:

  - "?" for predicates (i.e. functions returning #t or #f, e.g. member?)

  - "!" for any procedure with "side effects" (i.e. changes of bindings for non-local variables, e.g. set!)

- Procedure definitions should be brief

  - Oriented towards a single, well-defined task

  - Should be split into a number of subtasks if > 1 page

- Comments:

  - ; for comments on the same line with code

  - ;; for comments that run from beginning of line

  - ; for comments that describe the contents of the file (usually first in file)

- Indentation:

  - Indent procedure definitions like this, with the body starting a new line, and indented a few characters (define (foo) 15)
Scheme: programming style - cont’d

• Deeply nested cons and cdrs are often difficult to understand, and should therefore be avoided.

• Since Scheme is a dynamically typed language, the names of parameters should reflect their value.

• Most general guidelines on programming style also apply to Scheme programs.
  – Kernighan & Plauger (1974) and Ledgard (1974) give good summaries of these.