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
  \[(\text{define} \ (<\text{name}> \ <\text{formal parameters}>\)) \ (<\text{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:**
  - \[(\text{define} \ (\text{square} \ x) \ (* \ x \ x))\]
  - \[(\text{define} \ (\text{sum-of-squares} \ x \ y) \ (+ \ (\text{square} \ x) \ (\text{square} \ y))))\]
  - \[(\text{define} \ (\text{circle-area} \ \text{rad}) \ (* \ 3.14 \ (\text{square} \ \text{rad}))))\]

- **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!*
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:

- 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 (* 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 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 g)) ;; Scheme’s assignment operator, meaning g = g*g
    (+ a g))

  ]=> (f 7)
  107
  ]=> (f 7)
  10007 ;; BADD
Scheme: predicate functions

- Function that return #t (true) or #f (false)
  - Note: some scheme interpreters use empty list () to indicate #f

- Predefined 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
  - (eq? … ) ; mathematical equivalence relation (it is symmetric, reflexive, and
  - (equal? … ) ; transitive)
  - (equal? … ) ; the finest or most discriminating
  - (equal? … ) ; the coarsest
Scheme: predicate functions - cont’d

• **eq? eqv? equal?**
  – **eq?** tests for the same object (similar to a pointer comparison)
  – **eqv?** extends **eq?** to look into value of numbers and characters
  – **equal?** Goes further than **eqv?** And looks recursively into the contents of lists, vectors, etc..

• **Rules:**
  – For all three tests, objects of different types are never equal.
  – Exact and inexact numbers are considered different types

• **Example:**

```
(define (f x y) (list x y))
so (f ‘a ‘a) yields (a a).
How does Scheme implement this?
It binds both x and y to the same atom a.
eq? checks that x and y both point to the same place

Say we called (f ‘(a) ‘(a)). then x and y don’t point to the same list at all!
```

```
the atom: a

the arguments: (a)
```
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)) 1 2 3)` ; ?

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:
  \[
  \text{cond} \quad \langle p_1 \rangle \quad \langle e_1 \rangle \\
  \langle p_2 \rangle \quad \langle e_2 \rangle \\
  \langle p_m \rangle \quad \langle e_m \rangle \\
  \text{else} \quad \langle e_n \rangle 
  \]
  – \text{cond} \text{ is a built in primitive}
  – \text{p}_i \text{ 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 (\text{e}_i) 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
    \[
    \text{if} \quad \langle \text{predicate} \rangle \quad \langle \text{consequent} \rangle \quad \langle \text{alternative} \rangle \\
    \text{if} \quad \langle \text{predicate} \rangle \quad \langle \text{consequent} \rangle 
    \]
  – 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:

```scheme
(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>`

Example: simple quadratic solver

\[ ax^2 + bx + c = 0, \quad x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \]

```scheme
(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))))
)
```

\[
\int xe^x = xe^x - \int e^x dx = xe^x - e^x + C.
\]
Scheme: let, scope & symbol table

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

• **Symbol table**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>year</td>
<td>2000</td>
</tr>
<tr>
<td>class</td>
<td>senior</td>
</tr>
<tr>
<td>gpa</td>
<td>3.75</td>
</tr>
<tr>
<td>heroes</td>
<td></td>
</tr>
</tbody>
</table>

(define year 2000)
(define class 'senior)
(define gpa 3.75)
(define heroes (list 'turing 'church 'mearthy))

• **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***

```scheme
(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
)
```

```scheme
(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.
• **letrec** is similar to **let**:
  – Assign values to variables that are visible within the current scope

• **Syntax:**

```scheme
(letrec (
  (var₁ val₁)
  (var₂ val₂) ...
)
  exp₁ exp₂ ...)
```

• **Example:**

```scheme
]=> (letrec ( (sum (lambda (lst)
    (if (null? lst) 0
    (+ (car lst) (sum (cdr lst) )) ) ) ) )
  (sum '(1 2 3 4 5)))
15
```
Scheme: scope - cont’d

- **let v.s. letrec:**
  - Mutually recursive functions can’t be defined using let
  - E.g.:
    
    ```scheme
    => (let ((odd? (lambda (x)
                    (if (zero? x) #f
                    (even? (- x 1)))))
      (even? (lambda (x)
                    (if (zero? x) #t
                    (odd? (- x 1)))))
      (even? 10) )
    Error!
    
    => (letrec ((odd? (lambda (x)
                        (if (zero? x) #f
                        (even? (- x 1)))))
      (even? (lambda (x)
                    (if (zero? x) #t
                    (odd? (- x 1)))))
      (even? 10) )
    #t
    ```
**Scheme: scope - cont’d**

- **define v.s. let**
  - Sequence of define's are nested let's
  - E.g.:  
    - (define x 1)  
    - (define y (+ x 10))  
    - (+ x y)
  - Can we replace this with a pair of define?
    - (lambda (x y)  
      - (let ((y (+ x y))  
        (x (- x y)))  
      (/ x y))))

- **define v.s. letrec:**
  - E.g.:  
    - (define (f x) (+ x (g x)))  
    - (define (g x) x)  
    - (f 10)
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)
Consider a function named \textit{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

\[
X^N = \underbrace{X \times X \times X \times \ldots \times X}_{N \text{ times}}
\]

We could also write this formula as:

\[
X^N = \underbrace{X \times X \times X \times X \times \ldots \times X}_{(N-1) \text{ times}}
\]

Or even as:

\[
X^N = \underbrace{X \times X \times X \times X \times \ldots \times X}_{(N-2) \text{ times}}
\]
In fact we can write the formula:

\[ X^N = X \times X^{N-1} \]

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 \( \rightarrow X^1 \) is \( X \)

Pseudocode:

```
IF n is 1
    Return x
ELSE
    Return x * Power(x, n - 1)
```
Scheme: recursive example - cont’d

- Trace:

Power(2,3) → Returns 8

Call 1:

x n 2 3 → Returns 4

Call 2:

x n 2 2 → Returns 2

Call 3:

x n 2 1

IF n is 1
  Return x
ELSE
  Return x * Power(x, n - 1)
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:

```plaintext
Factorial (In: n)

IF n is 0
    Return 1
ELSE
    Return n * Factorial(n - 1)
```

Scheme: recursive example2
In Scheme:

\[
\text{(define } (\text{factorial } n) \\
\quad (\text{if } (= n 0) ; \text{ base condition} \\
\quad 1 \\
\quad (* n (\text{factorial } (- n 1)))); \text{ recursive case})
\]

Trace:

```
Factorial(4)  Returns 24
  Call 1  n 4  Returns 6
    Call 2  n 3  Returns 2
      Call 3  n 2  Returns 1
        Call 4  n 1  Returns 1
          Call 5  n 0
```
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.
    \[
    \Rightarrow (\text{define} \ (\text{append} \ \text{lst1} \ \text{lst2})
    
    \begin{cases}
      (\text{if} \ (\text{null?} \ \text{lst1}) \ \text{lst2} \ ; \text{base case}) \\
      \ (\text{cons} \ (\text{car} \ \text{lst1}) \ (\text{append} \ (\text{cdr} \ \text{lst1}) \ \text{lst2}))) \ ; \text{recursive case}
    \end{cases}
    \]

    \[
    \Rightarrow (\text{append} \ '(a \ b \ c) \ '((d \ e \ f)))
    \]

    \[
    (a \ b \ c \ d \ e \ f)
    \]
Scheme: recursion types

- **Deep(tree) Recursion:**
  - If the recursion is applied over *all* the items of a list.
  - E.g.

```
]=> (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
```
Scheme: tail recursion

- **Tail Recursion:**
  - When the last thing a function does is to call itself.
  - 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
    > (trace fact)
    (fact)
    > (fact 5)
    | (fact 5)
    | (fact 4)
    | | (fact 3)
    | | | (fact 2)
    | | | | (fact 1)
    | | | | | (fact 0)
    | | | | | 1
    | | | | 1
    | | 2
    | 6
    | 24
    | 120
    120
    ```
Scheme: function as an argument

- **Functions are passed as any other argument**
  - E.g.:
    ```scheme
    (let ((double-any (lambda (func x) (func x x))))
      (list (double-any + 13) (double-any list '(a))))
    (26 ((a) (a)))
    ```

    ```scheme
    (define foo (lambda (lst)
      (cond ((null? lst) 0 )
            (else (+ (car lst) (foo (cdr lst))))))
    ```

    ```scheme
    (define generic-func (lambda (func lst)
      (cond ((null? lst) '()
            (else (cons (func (car lst))(generic-func func (cdr lst))))))
    ```

    ```scheme
    (generic-func foo '((3 1) (6 29 4) (5) () ) )
    (4 39 5 0)
    ```
• Using *apply* function:
  – Applies a function to each element in a list
  – Syntax:
    
    \[
    \text{(apply function-name (arg1 arg2 ...))}
    \]
  – The arguments must be in a list, even if it is one argument
  – E.g.:
    
    \[
    \] => (apply + (list 3 4 5 6))
    18
    
    \[
    \] => (apply + 3 4 '(5 6))
    18
    
    \[
    \] => (define compose (lambda (func1 func2)
      
    (lambda args
      
    (lambda (args
      
    (apply func1 (list (apply func2 args)))))))
    
    \[
    \] => ((compose sqrt *) 12 75)
    30
Scheme: function – variable arguments

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

- Pair notation
  - Pairs are created by the procedure cons
  - Eg.
    \[
    \text{\texttt{=> (define foobar (cons 'foo 'bar))}}
    \]
    \[
    \text{\texttt{=> foobar}}
    \]
    \[
    \text{\texttt{(foo . bar)}}
    \]
  - \((a \ b \ c \ d \ e)\) and \((a . (b . (c . (d . (e . ()))))\)) are equivalent notations of a list of symbols

- Example of variable num parameters using pairs
  \[
  \text{\texttt{=> (define func(lambda (param1 param2 . varparam))}}
  \]
  \[
  \text{\texttt{(display varparam)))}}
  \]
  \[
  \text{\texttt{=> (func '1) '(2) '(a b c) '(x y) )}}
  \]
  \[
  \text{\texttt{=> (func '1) '(2) '(a b c) '(x y) '(z) )}}
  \]
  \[
  \text{\texttt{=> (func '1) '(2) )}}
  \]
  \[
  \text{\texttt{;; ??}}
  \]
A higher order function used to apply another function to every element of a list:

\[
\text{map } \langle \text{func} \rangle \langle \text{arg-list} \rangle
\]

- Arguments: a function and arg-list
- func must be a function taking as many arguments as there are in arg-list and returning a single value
- The dynamic order in which func is applied to the elements of the arg-list is unspecified.

**Examples:**

\[
\Rightarrow (\text{map } + \langle 1 \ 2 \ 3 \rangle \langle 4 \ 5 \ 6 \rangle)
\]

\((5 \ 7 \ 9)\)

\[
\Rightarrow (\text{map } (\text{lambda} \ (x) (+ \ 1 \ x)) \langle 1 \ 2 \ 3 \rangle)
\]

\((2 \ 3 \ 4)\)

\[
\Rightarrow (\text{map } \text{abs} \langle -1 \ 2 \ -3 \ -4 \rangle)
\]

\((1 \ 2 \ 3 \ 4)\)
Scheme: higher order func - reduce

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

• Syntax:

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

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

• Implementation:

  \[(\text{define } (\text{reduce } \text{func } \text{list } \text{id}))\]

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

• Example:

  \[
  ]=> (\text{reduce } + '(1 2 3 4 5) 0) ;15
  ]=> (\text{reduce } * '(1 2 4 6 8 10) 1) ;3840
  ]=> (\text{reduce} \text{ append } '((1 2 3) (4 5 6) (7 8)) () ) ;(1 2 3 4 5 6 7 8)
  ]=> (\text{reduce} \text{ expt } '(2 2 2 2) 1) ;65536
Another high order function *(map, reduce)*

**Syntax:**

```scheme
(for-each func list)
```

- Applies function to each element of a list, (or to corresponding elements of a set of lists) and returns the value from last application.

**Implementation:**

```scheme
(define (for-each proc list)
  (cond ((null? (cdr list)) (proc (car list))) ; one-element list
        (else (proc (car list)) (for-each proc (cdr list))))))
```

**Example:**

```scheme
]=> (let ((v (make-vector 5)))
          (for-each (lambda (i)  (vector-set! v i (* i i))) '(0 1 2 3 4)) v)
#(0 1 4 9 16)
```
Scheme: expressions and short circuit evaluation

• `(and ....)`
  – E.g.
    ```scheme
    (if (and (try-first-thing)
              (try-second-thing)
              (try-third-thing))...)
    ```
  – 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.

• `(or ....)`
  – E.g.
    ```scheme
    (if (or (try-first-thing)
             (try-second-thing)
             (try-third-thing)) ...)
    ```
  – Likewise, it stops when it gets a true value
Scheme: debugging

- **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.
  - E.g.:
    ```scheme
    1 ]=> (begin
      (write-line 'foo)
      (bkpt 'test-2 'test-3)
      (write-line 'bar)
      'done)
    
    foo
    test-2 test-3
    ;To continue, call RESTART with an option number:
    ; (RESTART 2) => Return from BKPT.
    ; (RESTART 1) => Return to read-eval-print level 1.
    
    2 bkpt> (+ 3 3)
    ;Value: 6
    
    2 bkpt> (continue)
    bar
    ;Value: done
    ```
• **pp object [output-port [as-code?]]**
  – Prints the source code of a given procedure.
  – When debugging, you will have a procedure object but will not know exactly what procedure it is.

• **pa procedure**
  – Prints the arguments of a procedure

• **apropos string**
  – Search an environment for bound names containing string and print out the matching bound names.
Scheme: programming style

• Use special suffix:
  – "?" 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)
• Deeply nested cars 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.