

COMPUTER SCIENCE 51

Spring 2009
cs51.seas.harvard.edu

Prof. Greg Morrisett
Prof. Ramin Zabih

computer
science 51



Logistics

- Sectioning tool!!!!
- First problem set!!!!
- This week: *HTDP* ch. 2-4, 9, 6

computer
science 51



Intro to Scheme

- Scheme is a dialect of Lisp
 - Lisp: McCarthy -- roughly 1957
 - Inspired by Church's λ -calculus
 - Scheme: Steele & Sussman -- 1970's
 - PLT/Scheme: Felleisen et al. -- 1990's



computer
science 51



Primary Features of Scheme

- Very simple and uniform
 - Dirt simple models of evaluation
 - Substitution
 - Environment
 - One of only 2 languages with *formal* semantics.
 - For our purposes three advantages:
 - Forces us to use abstraction
 - Gives us analytic models for reasoning
 - Helps to unlearn certain bad habits

computer
science 51



Primary Features of Scheme

- Primarily *functional*
 - All computation done via functions
 - No loops, iterators, ...
 - Connection with recursion theory
 - Functions are 1st class values
 - Can pass/return them to functions
 - Can place them in data structures
 - Minimize mutation
 - i.e., updates
 - "persistent" data structures
- Other functional languages: SML, O'CamL, F#, Haskell, Erlang, E-lisp, ...

computer
science 51



Primary Features of Scheme

- Supports nested, *Lexical Scopes*
 - As opposed to dynamically-scoped Lisp.
 - Roughly:
 - A variable's meaning can be determined from the code, instead of having to run it.
 - Gives us a way to limit variable interactions.
 - Basic composition principle.
 - Surprising consequences.

computer
science 51



Primary Features of Scheme

- Good support for *meta-programming*
 - Code = Data (in particular, trees!)
 - Great for writing programs that manipulate other programs:
 - e.g., compilers, interpreters, mathematics, ...
 - Macros at the level of *abstract syntax*
 - More robust than C's macro support
 - Itself a Turing-complete language
 - Makes it easy to extend the language

computer
science 51

Other Aspects of Scheme

- Call-by-value [aka strict] language
 - As opposed to Haskell's call-by-need [aka lazy]
 - As opposed to CPP's call-by-name
- Dynamically typed language
 - As opposed to Java which is statically-typed
 - As opposed to C which is (very) weakly typed.
- Garbage collected
 - As opposed to C/C++
- Prefix, parenthesized syntax
 - Easy to parse.
 - Takes a while to get used to.

computer
science 51

Places where Scheme is Used

- Teaching!
- Compilers
 - e.g., MIT/Scheme, Chez, PLT
- Systems:
 - scsh
- Scripting:
 - GIMP
 - Synopsys (CAD)
 - Disneyworld
 - Final Fantasy: The Sprits Within Together
 - Viruses...

computer
science 51

PLT Scheme

- Dialect of Scheme for teaching
 - Integrated development environment
 - Editing and REPL
 - Algebraic stepper (Debugger)
 - Analysis, testing & feedback tools
 - Language levels
 - Language extensions & alternatives
 - e.g., modules, objects, mixins, ...
 - Lazy Scheme, Typed Scheme, ...

computer
science 51

Scheme in 2 Slides: Syntax

- Values:
 - Booleans: `true`, `false` (`#t`, `#f`)
 - Numbers: `42`, `-3`, `1/2`, `1.25`, `#i3.1415`
 - Strings: `"Greg"`, `"Cool"`, `"Ramin"`, `"Not"`,
 - Functions: `not`, `string-append`, `+`, `*`, ...
- Compound expressions:
 - `(not true)`
 - `(string-append "CS" "51")`
 - `(<function> <arg1> <arg2> ... <argn>)`

computer
science 51

Notes on Prefix Notation

- The function always goes first:
 - Instead of `3+2` we write `(+ 3 2)`
 - Instead of `true && false` we write `(and true false)`
 - Instead of `3 < 4` we write `(< 3 4)`
- The parentheses are necessary for compound expressions:
 - Instead of `3+2*6` we write `(+ 3 (* 2 6))`
- Extra parentheses are not allowed:
 - We can't write: `(+ (3) 2)`

computer
science 51

Evaluating Functions:

- When you see:
`(define (f x1 x2 ... xn) <exp>)`
- To evaluate compound expression:
`(f arg1 arg2 ... argn)`
 - Evaluate the arguments as usual to values v_1, v_2, \dots, v_n .
 - Substitute value v_i for parameter x_i within `<exp>`.
 - And evaluate the resulting expression.

computer
science 51

Another Example:

```
(define pi 3.1415926)

(define (square x)
  (* x x))

(define (circle-area radius)
  (* pi (square radius)))

(circle-area 3) ↓ #i28.274333882308
```

computer
science 51

If-expressions

```
(if <exp> <exp> <exp>)
```

```
(define (min x y) (if (<= x y) x y))
```

- To evaluate `(if e1 e2 e3)`
 - Evaluate e_1 .
 - If it evaluates to `true`, evaluate e_2 and return its value.
 - If it evaluates to `false`, evaluate e_3 and return its value.

computer
science 51

Special Forms

- `define` and `if` are *special forms*.
 - Don't follow the usual rule of compound expression evaluation.
 - There will be a few more exceptions to the general rule.
- What goes wrong if we try to define `if` as a function?
`(define (if e1 e2 e3) ...)`

computer
science 51

Call-By-Value vs Name

- Call-by-value:
 - Evaluate arguments to values
 - Then substitute them for parameters
- Call-by-name:
 - Substitute arguments for parameters before evaluating them:

```
(define (square x) (* x x))
(square (* 3 2))
(* (* 3 2) (* 3 2))
```

computer
science 51

A note on style...

What's wrong with the following?

```
(define (zero? x)
  (if (= x 0) true false))
```

computer
science 51

More on Style:

It's also bad style to write:

```
(define (convert x)
  (if (string=? x "Red")
      0
      (if (string=? x "Blue")
          -1
          (if (string=? x "Green")
              1
              (error 'convert
                    "bad color"))))))
```

computer
science 51



Cond

```
(define (convert x)
  (cond
    [(string=? x "Red") 0]
    [(string=? x "Blue") -1]
    [(string=? x "Green") 1]
    [else
     (error 'convert
            "Bad color")]))
```

computer
science 51



Syntax for Cond

```
(cond
  [<test1> <exp1>]
  [<test2> <exp2>]
  ...
  [<testn> <expn>]
  [else <exp>])
```

Optionally, leave off the **else** clause.

computer
science 51



Evaluating Cond

To evaluate

```
(cond [<test1> <exp1>]
      [<test2> <exp2>]
      ...
      [<testn> <expn>]
      [else <exp>])
```

- Evaluate $test_1$. If it evaluates to true, return the value of exp_1 .
- Otherwise, evaluate $test_2$. If it evaluates to true, return the value of exp_2 .
- ... If all of the preceding tests fail, evaluate the else expression.

computer
science 51



Same as...

```
(cond [<test1> <exp1>]
      [<test2> <exp2>]
      ...
      [<testn> <expn>]
      [else <exp>])

(if <test1> <exp1>
    (if <test2> <exp2>
        ...
        (if <testn> <expn> <exp>)...))
```

computer
science 51



Recursive Functions

```
; fact : number -> number
; (fact n) returns n!
; Example: (fact 4) ↓ 24
(define (fact n)
  (if (= n 0)
      1
      (* n (fact (- n 1)))))
```

computer
science 51



Addition & Multiplication?

- Imagine we only have `inc`, `dec`, and a test for `zero?`.
- How could we write:
 - Addition?
 - Less-than-or-equal?
 - Multiplication?
 [assuming only non-negative, whole numbers.]

computer
science 51

Structs (aka Records)

- In C, a record or struct is declared as:


```
struct point { int x ; int y ; }
```
- In PLT/Scheme:


```
; define new struct
(define-struct point (x y))
; construct a point
(define p (make-point 3 4))
; access a point's fields
(point-x p) ↓ 3
(point-y p) ↓ 4
```

computer
science 51

Evaluation of Structs

- Given:


```
(define-struct name (x1 x2 ... xn))
```
- Records of values are values.
 - `(make-name v1 v2 ... vn)` is a value.
- `(name? (make-name v1 v2 ... vn))` ↓ true.
- `(name? v)` ↓ false for all other values.
- `(name-xi (make-name v1 v2 ... vn))` ↓ v_i

computer
science 51

For Example:

```
(define-struct book
  (author title year))

(define htdp
  (make-book "M.Felleisen et al."
    "How to Design Programs" 2003))

(book? htdp) ↓ true   (book? 3) ↓ false
(book-year htdp) ↓ 2003
```

computer
science 51

Lists

- empty (aka null)
 - A list value with no elements.
- `(cons x xs)`
 - A list value whose first element is x
 - And the rest of the list is xs
- `(cons 1 (cons 2 (cons 3 empty)))`
- `(cons "amy" (cons "john" empty))`

computer
science 51

List Abbreviation

We write:

```
(list 1 2 3 4 5)
```

to abbreviate:

```
(cons 1
  (cons 2
    (cons 3
      (cons 4 (cons 5 empty))))))
```

computer
science 51

Lists Can Hold Anything

```
(list "john" 9 true) =
  (cons "john"
    cons 9 (cons true empty))

(list (list 1 2 3) (list)) =
  (cons (cons 1 (cons 2
    (cons 3 empty)))
    (cons empty empty))
```

computer
science 51

List Operations

- (empty? x), (cons? x)
- (list? x) =
(or (empty? x) (cons? x))
- (first x)
– a.k.a (car x)
– Returns first element of list
- (rest x)
– a.k.a. (cdr x)
– Returns the rest of the list

computer
science 51

List Processing

```
; length : list -> number
; returns number of elements in
; the list.
; (length (list 1 2 3)) ↓ 3
; (length empty) ↓ 0
(define (length x)
  (if (empty? x)
      0
      (+ 1 (length (rest x)))))
```

computer
science 51

More List Processing

```
; sum : (list number) -> number
; returns the sum of the numbers in
; the list.
; (sum (list 1 2 3)) ↓ 6
; (sum empty) ↓ 0
(define (sum x)
  (if (empty? x)
      0
      (+ (first x) (sum (rest x)))))
```

computer
science 51

Append Two Lists

```
; append : (list list) -> list
; (append (list 1 2 3) (list 4 5)) ↓
; (list 1 2 3 4 5)
; (append (list 1 2 3) empty) ↓
; (list 1 2 3)
(define (append x y)
  (if (empty? x)
      y
      (cons (first x)
            (append (rest x) y))))
```

computer
science 51

More List Processing

```
; reverse : list -> list
; returns the list with the elements
; in reverse order.
; (reverse (list 1 2 3)) ↓ (list 3 2 1)
; (reverse empty) ↓ empty
(define (reverse x)
  (if (empty? x)
      empty
      (append (reverse (rest x))
              (cons (first x) empty))))
```

computer
science 51

Patterns

Almost all list-processing functions have the following pattern:

```
(if (empty? x)
    <base-case>
    <recursive-case>)
```

where the second case uses `(first x)` and recurses on `(rest x)`.

Later on, we'll see how we can factor out this pattern into a very re-usable function.

computer
science 51



Every-Other

Goal: pull out every other element in the list

```
(every-other (list 1 2 3 4 5 6))
```

should give back the list:

```
(list 1 3 5)
```

computer
science 51



Every-Other

```
(define (every-other x)
  (cond [(empty? x) empty]
        [(empty? (rest x)) x]
        [else (cons (first x)
                     (every-other
                      (rest (rest x))))]))
```

computer
science 51



How To Do Even Elements?

```
(define (odds x) (every-other x))
```

```
(define (evens x)
  (if (empty? x)
      empty
      (every-other (rest x))))
```

computer
science 51



Mergesort

```
(define (sort x)
  (cond [(empty? x) empty]
        [(empty? (rest x)) x]
        [else
         (merge
          (sort (odds x))
          (sort (evens x)))]))
```

computer
science 51



The Merge in Mergesort

```
(define (merge x y)
  (cond [(empty? x) y]
        [(empty? y) x]
        [(<= (first x) (first y))
         (cons (first x)
               (merge (rest x) y))]
        [else
         (cons (first y)
               (merge x (rest y)))]))
```

computer
science 51



Local Variables via Let

```
(let ([x 21]
      [y 2])
      (* x y)) ↓ 42
```

More generally:

```
(let ([x1 <exp1>]
      [x2 <exp2>]
      ...
      [xn <expn>]
      <exp>)
```

computer
science 51



Evaluation of Let

```
(let ([x1 <exp1>]
      [x2 <exp2>]
      ...
      [xn <expn>]
      <exp>)
```

- Evaluate all of the $\langle \text{exp}_i \rangle$ to values.
- Then substitute them for their corresponding variables in $\langle \text{exp} \rangle$.

computer
science 51



Subtlety

; Illegal!

```
(let ([x 19]
      [y (+ 2 x)])
      (+ x y))
```

; Right way:

```
(let ([x 19])
      (let ([y (+ 2 x)])
            (+ x y)))
```

computer
science 51



Summary

- General evaluation rule for Scheme:
 - evaluate nested expressions to values
 - apply the function to the values
 - for user-defined functions, substitute the values for the parameters in the body of the function and evaluate the resulting expression.
- A handful of exceptions:
 - if, cond, let

computer
science 51



Scheme Data

- Primitive values: numbers, strings, etc.
- Structs:
 - (define-struct <name> (<x₁> ... <x_n>))
 - constructor: make-<name>
 - predicate: <name>?
 - accessors: <name>-<x>
- Lists:
 - empty, cons are constructors
 - (list 1 2 3) abbreviates (cons 1 (cons 2 (cons 3 empty)))
 - first, rest are accessors
 - empty?, cons?, list? are predicates

computer
science 51



Scheme Control

- All loops done with recursion
 - takes some getting used to.
 - use PLT stepper to visualize.

computer
science 51

