

COMPUTER SCIENCE 51

Spring 2009
cs51.seas.harvard.edu

Prof. Greg Morrisett
Prof. Ramin Zabih

computer
science 51



MANDATORY FUN DAY



computer
science 51



Scheme in 2 Slides: Syntax

- Values (a.k.a. atoms):
 - Booleans: `true`, `false` (`#t`, `#f`)
 - Numbers: `42`, `-3`, `1/2`, `1.25`, `#i3.1415`
 - Strings: `"Moo"`, `"Cow"`
 - Functions: `+`, `*`, `<=`, `first`, `rest`, ...
- Compound expressions :
 - `(not true)`
 - `(string-append "CS" "51")`
 - `(<function> <arg1> <arg2> ... <argn>)`

computer
science 51



Scheme in 2 Slides: Semantics

- Values evaluate to themselves.
 - `true` ↓ `true`
 - `3` ↓ `3`
 - `"Greg"` ↓ `"Greg"`
- To evaluate a compound expression (except for special forms):
 1. Evaluate all of the nested expressions to values.
 2. Apply the primitive to the values to get out a value.

computer
science 51



Definition Evaluation:

`(define <var> <exp>)`

- Evaluate the expression to a value `v`.
- Substitute `v` everywhere you encounter the variable.

`(define x (+ 3 (* 2 6)))`

`(define y (+ x 3))`

computer
science 51



Evaluating Functions:

- When you see:
 - `(define (f x1 ... xn) <exp>)`
- To evaluate compound expression:
 - `(f <arg1> ... <argn>)`
 - Evaluate the arguments as usual to values `v1`, ..., `vn`.
 - Substitute value `vi` for parameter `xi` within `<exp>`.
 - And evaluate the resulting expression.

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 `e1`.
 - If it evaluates to `true`, evaluate `e2` and return its value.
 - If it evaluates to `false`, evaluate `e3` and return its value.

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 `test1`. If it evaluates to `true`, return the value of `exp1`.
- Otherwise, evaluate `test2`. If it evaluates to `true`, return the value of `exp2`.
- ... If all of the preceding tests fail, evaluate the else expression.

computer
science 51



Structs (aka Records)

```
; 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))` ↓ `vi`

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` (a.k.a. `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 the 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

Patterns

Almost all list-processing functions have the following pattern:

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

where the recursive 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



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



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) =
  (cons 1 (cons 3 (cons 5 empty)))
```

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



Merging Ordered Lists

```
(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

Mergesort

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

computer
science 51

Mergesort Using "Let"

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

computer
science 51

Evaluation of Let

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

- Evaluate all of the <exp_i> to values.
- Then substitute them for their corresponding variables in <exp>.

computer
science 51

Example Let Evaluation

```
(let ([x (+ 1 20)]
      [y (+ 1 1)])
  (* x y)) ⇒

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

computer
science 51

Subtlety

```
; Illegal!
(let ([x 19]
      [y (+ 2 x)]) ← x not yet defined!
  (+ x y))

; Right way:
(let ([x 19]
      (let ([y (+ 2 x)])
        (+ x y))))
```

computer
science 51

Association Lists

Associate keys (e.g., phone numbers) with a value (e.g., name).

```
(define-struct entry (key value))

(list (make-entry 1337 "Greg")
      (make-entry 1876 "Ramin")
      (make-entry 7734 "Gideon")
      (make-entry 9526 "Victor"))
```

computer
science 51



Lookup

```
(define (lookup db key)
  (cond
    [(empty? db)
     (error 'lookup "not found")]
    [(equal? key (entry-key (first db)))
     (entry-value (first db))]
    [else (lookup (rest db) key)]))
```

- How could we make this faster?

computer
science 51



Trees?

What if we wanted to use a binary tree instead of an association list?

```
(define-struct node (key value left right))

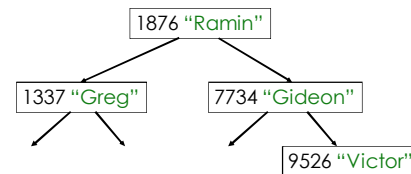
(make-node 1876 "Ramin"
          (make-node 1337 "Greg" empty empty)
          (make-node 7734 "Gideon"
                    empty
                    (make-node 9526 "Victor" empty empty)))
```

computer
science 51



Visualizing the Tree

```
(make-node 1876 "Ramin"
          (make-node 1337 "Greg" empty empty)
          (make-node 7734 "Gideon"
                    empty
                    (make-node 9526 "Victor" empty empty)))
```



computer
science 51



Structs as Lists

```
(define-struct node (key value left right))
```

can be simulated with:

```
(define (make-node key value left right)
  (cons key
        (cons value
              (cons left (cons right empty))))))

(define (node-key n) (first n))
(define (node-value n) (first (rest n)))
(define (node-left n) (first (rest (rest n))))
(define (node-right n)
  (first (rest (rest (rest n)))))
```

computer
science 51



Directed Graphs

What are some possible representations?

- list of edges, where each edge is a pair of nodes.
 - (list (make-edge "Greg" "Ramin")
 (make-edge "Greg" "Gideon")
 (make-edge "Victor" "Gideon"))
- association list, where we associate a node to a list of neighboring nodes.
 - (list (make-neighbors "Greg"
 (list "Ramin" "Gideon"))
 (make-neighbors "Victor"
 (list "Gideon"))
 (make-neighbors "Ramin" empty))

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:
 - define, 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_i>
- 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

