Announcements

- Reminder: **First project** deadline: March 4. Do you need an extension?

- Homework sample solutions 1 and 3 are now available on sakai (resources). Homework 2 sample solution will be posted soon.

- **Midterm: March 9**, in class, 80 minutes, closed book / notes;

- Spring break: March 12 - March 20.
Functional Programming

Pure Functional Languages

Scott Chapter 10.

Fundamental concept: application of (mathematical) functions to values

1. Referential transparency: The value of a function application is independent of the context in which it occurs
   
   • value of \( f(a, b, c) \) depends only on the values of \( f \), \( a \), \( b \) and \( c \)
   
   • It does not depend on the global state of computation

   ⇒ all vars in function must be local, or parameters
Pure Functional Languages

1. The concept of assignment is **not** part of functional programming
   - no explicit assignment statements
   - variables bound to values only through the association of actual parameters to formal parameters in function calls
   - function calls have no side effects
   - thus no need to consider global state

2. Control flow is governed by function calls and conditional expressions
   - no iteration
   - recursion is widely used
Pure Functional Languages

1. All storage management is implicit
   • needs garbage collection

2. Functions are *First Class Values*
   • Can be returned as the value of an expression
   • Can be passed as an argument
   • Can be put in a data structure as a value
   • (Unnamed) functions exist as values
Pure Functional Languages

A program includes:

1. A set of function definitions
2. An expression to be evaluated

E.g. in Scheme:

```scheme
> (define (length x)
   (if (null? x)
       0
       (+ 1 (length (rest x))))

> (length '(A LIST OF 5 THINGS))
5
```
LISP

- Functional language developed by John McCarthy in the mid 50’s
- Semantics based on *Lambda Calculus*
- All functions operate on lists or symbols: (called “S-expressions”)
- Only five basic functions: list functions **cons**, **car**, **cdr**, **equal**, **atom** and one conditional construct: **cond**
- Useful for list-processing applications
- Programs and data have the same syntactic form: S-expressions
- Used in Artificial Intelligence
- **SCHEME**: Developed in 1975 by G. Sussman and G. Steele as a version of LISP

⇒ we are using SCHEME here

You can call SCHEME interpreters on the ilab cluster by saying: **mzscheme** or **racket** (command line interpreter); **drracket** (window-based interpreter).
S-expressions

S-expression ::= Atom | ‘(’ { S-expression } ‘)’
Atom ::= Name | Number | #t | #f

#t

()

(a b c)

(a (b c) d)

(((a b c) (d e (f))))

(1 (b) 2)

Lists have nested structure.
S-expressions are lists in Scheme

The building blocks for lists are pairs or cons-cells. Lists use the empty list ( ) as an “end-of-list” marker.

Note: (a.b) is not a list!
Special (Primitive) Functions

- **eq?**: identity on names (atoms)
- **null?**: is list empty?
- **car**: selects first element of list (contents of address part of register)
- **cdr**: selects rest of list (contents of decrement part of register)
- **(cons element list)**: constructs lists by adding element to front of list
- **quote** or ’**: produces constants
Special (Primitive) Functions

- '() is the empty list
- (car '(a b c)) =
- (car '((a) b (c d))) =
- (cdr '(a b c)) =
- (cdr '((a) b (c d))) =
Special (Primitive) Functions

- **car** and **cdr** can break up any list:

  - \((\text{car} \ (\text{cdr} \ (\text{cdr} \ '((a) \ b \ (c \ d)))))\) =

  - \((\text{caddr} \ '((a) \ b \ (c \ d)))\)

- **cons** can construct any list:

  - \((\text{cons} \ 'a \ '())\) =

  - \((\text{cons} \ 'd \ 'e())\) =

  - \((\text{cons} \ '((a) \ b) \ '(c \ d))\) =

  - \((\text{cons} \ '((a) \ b \ c) \ '((a) \ b))\) =
Other Functions

• + - * / numeric operators, e.g.,
  (+ 5 3) = 8, (- 5 3) = 2
  (* 5 3) = 15, (/ 5 3) = 1.6666666

• = < > comparison operators for numbers

• Explicit type determination and test functions:

  ⇒ All return Boolean values: #f and #t
  - (number? 5) evaluates to #t
  - (zero? 0) evaluates to #t
  - (symbol? 'sam) evaluates to #t
  - (list? '(a b)) evaluates to #t
  - (null? '()) evaluates to #t

Note: SCHEME is a strongly typed language.
Other Functions

- `(number? 'sam)` evaluates to `#f`
- `(null? '(a))` evaluates to `#f`
- `(zero? (- 3 3))` evaluates to `#t`
- `(zero? '(- 3 3)) ⇒ type error`
- `(list? (+ 3 4))` evaluates to `#f`
- `(list? '(+ 3 4))` evaluates to `#t`
READ-EVAL-PRINT Loop

READ: Read input from user:
    a function application

EVAL: Evaluate input:
    \( (f \ arg_1 \ arg_2 \ldots \ arg_n) \)
    1. evaluate \( f \) to obtain a function
    2. evaluate each \( \arg_i \) to obtain a value
    3. apply function to argument values

PRINT: Print resulting value:
    the result of the function application
READ-EVAL-PRINT Loop Example

> (cons 'a (cons 'b '(c d)))
(a b c d)

1. Read the function application
   (cons 'a (cons 'b '(c d)))

2. Evaluate cons to obtain a function

3. Evaluate 'a to obtain a itself

4. Evaluate (cons 'b '(c d)):
   (a) Evaluate cons to obtain a function
   (b) Evaluate 'b to obtain b itself
   (c) Evaluate '(c d) to obtain (c d) itself
   (d) Apply the cons function to b and (c d) to obtain (b c d)

5. Apply the cons function to a and (b c d) to obtain (a b c d)

6. Print the result of the application:
   (a b c d)
Quotes Inhibit Evaluation

;;;Same as before:
> (cons 'a (cons 'b '(c d)))
(a b c d)

;;;Now quote the second argument:
> (cons 'a '(cons 'b '(c d)))
(a cons (quote b) (quote (c d)))

;;;Instead, un-quote the first argument:
> (cons a (cons 'b '(c d)))
ERROR: unbound variable: a
Quotes Inhibit Evaluation

;;Some things evaluate to themselves:
> (list 1 2 #t #f)
(1 2 #t #f)

;;They can also be quoted:
> (list '1 '2 '#t '#f)
(1 2 #t #f)
READ-EVAL-PRINT Loop

Can also be used to define functions.

**READ:** Read input from user:
- a symbol definition

**EVAL:** Evaluate input:
- store function definition

**PRINT:** Print resulting value:
- the symbol defined

Example:

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

#<unspecified>
Defining Global Variables

> (define foo '(a b c))
#<unspecified>

> (define bar '(d e f))
#<unspecified>

> (append foo bar)
(a b c d e f)

> (cons foo bar)
(((a b c) d e f)

> (cons 'foo bar)
(foo d e f)
Defining Scheme Functions

(define <fcn-name> (lambda (<fcn-params>)
  <expression>))

Example: Given function pair? (true for non-empty lists, false o/w) and function not (boolean negation):

(define atom? (lambda (object) (not (pair? object))))

Evaluating (atom? '(a)):
1. Obtain function value for atom?
2. Evaluate '(a) obtaining (a)
3. Evaluate (not (pair? object))
   a) Obtain function value for not
   b) Evaluate (pair? object)
      i. Obtain function value for pair?
      ii. Evaluate object obtaining (a)
       Evaluates to #t
    Evaluates to #f
Evaluates to #f
Function Definition

Two syntaxes for definition:

1. (define (<fcn-name> <fcn-params>)
   <expression>)

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

   (define (mean x y)
     (/ (+ x y) 2))

2. (define <fcn-name> (lambda (fcn-params)
   <expression>))

   (define square
     (lambda (n) (* n n)))

   (define mean
     (lambda (x y) (/ (+ x y) 2)))
Conditional Execution: if

(if <condition> <result1> <result2>)

1. Evaluate <condition>

2. If the result is a “true value” (i.e., anything but #f), then evaluate and return <result1>

3. Otherwise, evaluate and return <result2>

(define abs-val
  (lambda (x)
    (if (>= x 0) x (- x))))

(define rest-if-first
  (lambda (e l)
    (if (eq? e (car l)) (cdr l) '())))
Conditional Execution: cond

(cond (<condition1> <result1>)
   (<condition2> <result2>)
   ...
   (<conditionN> <resultN>)) ; optional else clause

1. Evaluate conditions in order until obtaining one that returns a true value
2. Evaluate and return the corresponding result
3. If none of the conditions returns a true value, evaluate and return <else-result>
(define abs-val
  (lambda (x)
    (cond ((>= x 0) x)
          (else (- x)))))

(define rest-if-first
  (lambda (e l)
    (cond ((null? l) '())
          ((eq? e (car l)) (cdr l))
          (else '()))))
Recursive Scheme Functions: Length

(define (length x)
  (if (null? x)
      0
      (+ 1 (length (cdr x)))))

trace (length '(1 2)):

(length '(1 2))
  x = (1 2)
  (length '(2))
  x = (2)
  (length '())
  x = ()
  value: 0
  value: (+ 1 0) = 1
  value: (+ 1 1) = 2
Recursive Scheme Functions: Abs-List

- `(abs-list '(1 -2 -3 4 0)) ⇒ (1 2 3 4 0)
- `(abs-list '()) ⇒ ()

```
(define abs-list
  (lambda (l)
    (if (null? l)
        '()
        (cons (abs-val (car l)) (abs-list (cdr l))))))
```

Recursive Scheme Functions: Append

(append '1 2 '3 4 5) ⇒ (1 2 3 4 5)
(append '1 2 '3 (4) 5) ⇒ (1 2 3 (4) 5)
(append ' '1 4 5) ⇒ (1 4 5)
(append '1 4 5 ') () ⇒ (1 4 5)
(append ' ' ' ) ⇒ ()

(define append
  (lambda (x y)
    (cond ((null? x) y)
          ((null? y) x)
          (else (cons (car x) (append (cdr x) y))))))