## Class Information

- Homework problem set 6 will be posted by tonight.

## Review: 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

2. 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
3. Control flow is governed by function calls and conditional expressions

   $\Rightarrow$ no iteration

   $\Rightarrow$ recursion is widely used

## Pure Functional Languages

4. All storage management is implicit
   - needs garbage collection
5. 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:

```
> (define length
    (lambda (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

## Lambda calculus

- formalism for studying ways in which functions can be formed, combined, and used for computation
- computation is defined as rewriting rules (operational semantics) $\Rightarrow$ $\beta$ *reduction*
- the syntactic notion of computation was developed first; a mathematical semantics followed much later

Examples:

| | | |
|---|---|---|
| f(x) = x+2 | $\lambda$x.x+2 | different notation |
| ($\lambda$x.x+2) 1 | 1+2 = 3 | function application and substitution |
| ($\lambda$x.x) ($\lambda$y.y) | | arguments and returned "values" can be functions |
| $\lambda$x.xx | | untyped lambda calculus f(x) = x(x) |

## SCHEME

- Developed in 1975 by G. Sussman and G. Steele
- A version of LISP
- Simple syntax, small language
- Closer to initial semantics of LISP as compared to COMMON LISP
- Provides basic list processing tools
- Allows functions to be first class objects

## SCHEME

- Expressions are written in prefix, parenthesized form
  - (function $\texttt{arg}_1$ $\texttt{arg}_2$ ...$\texttt{arg}_n$)
  - `(+ 4 5)`
  - `(+ (* 3 4 5) (- 5 3))`
- Operational semantics: In order to evaluate an expression:
  1. evaluate `function` to a function value
  2. evaluate each $\texttt{arg}_i$ in order to obtain its value
  3. apply the function value to these values

## S-expressions

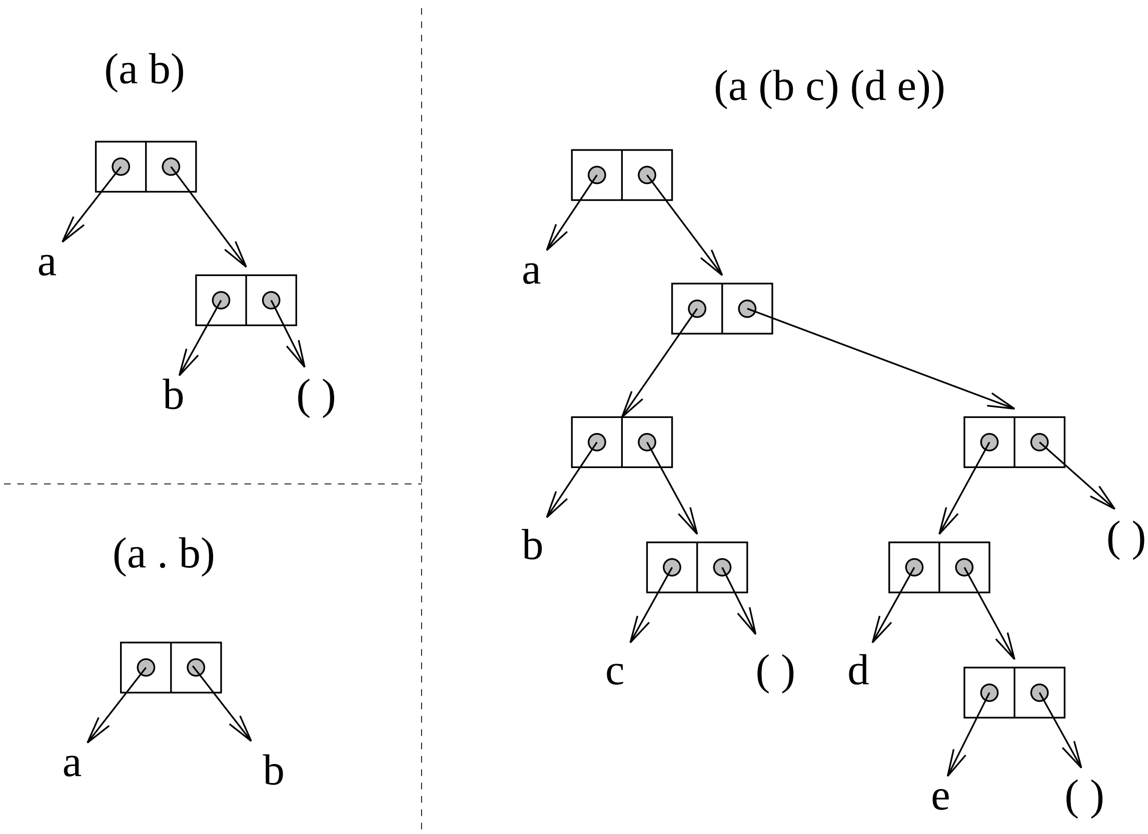
S-expression ::= Atom | '(' { S-expression } ')'
Atom ::= Name | Number | #t | #f

```
#t

()

(a b c)

(a (b c) d)

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

(1 (b) 2)
```

Lists have nested structure.

## 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:
  - `(car (cdr (cdr '((a) b (c d)))))` =
  - `(caddr '((a) b (c d)))`
- `cons` can construct any list:
  - `(cons 'a '())` =
  - `(cons 'd '(e))` =
  - `(cons '(a b) '(c d))` =
  - `(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))` $\Rightarrow$ type error
- `(list? (+ 3 4))` evaluates to `#f`
- `(list? '(+ 3 4))` evaluates to `#t`

## READ-EVAL-PRINT Loop

The Scheme interpreters on the ilab machines are called mzscheme, racket, and drracket. “drracket” is an interactive environment, the others are command-line based. For example: Type `mzscheme`, and you are in the READ-EVAL-PRINT loop. Use Control D to exit the interpreter.

**READ:** Read input from user:
a function application

**EVAL:** Evaluate input:
(f $\texttt{arg}_1$ $\texttt{arg}_2$ ...$\texttt{arg}_n$)

1. evaluate `f` to obtain a function
2. evaluate each $\texttt{arg}_i$ to obtain a value
3. apply function to argument values

**PRINT:** Print resulting value:
the result of the function application

You can write your Scheme program in file <name>.ss and then read it into the Scheme interpreter by saying at the interpreter prompt: `(load "<name>.ss")`

# 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
```

## Scheme Programming and Emacs

You can invoke the interpreter **mzscheme** Scheme interpreter on the ilab cluster from within `emacs` by executing the commands: ESC-x run-scheme.

Typically, you want to split your emacs window into two parts (CTRL-x 2), and then edit your Scheme file in one window, and execute it in the other. To read a Scheme program into the interpreter, say (load "<name>.ss"). You can switch between windows by saying CTRL-x o.

You can save the "scheme interpreter" window into a file to inspect it later, i.e., to keep a record on what you have done. This may be useful during debugging.

## Defining Global Variables

The define constructs extends the current interpreter environment by the new defined (name, value) association.

```
> (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`

## 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>)
      (else <else-result>)) ; 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>`

## Conditional Execution: cond

```
(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 '()))))
```