CS 314 Lecture 8

Functional programming

February 19, 2019

Resources

- Beating the averages: http://www.paulgraham.com/avg.html
- Scheme language: https://scheme.com/tspl4/
- Racket language: https://racket-lang.org/
- Realm of Racket: https://realmofracket.com/
- Structure and Interpretation of Computer Programs: https://github.com/sarabander/sicp-pdf/raw/master/sicp.pdf
- The Little Schemer: http://www.ccs.neu.edu/home/matthias/BTLS/

Functional programming

Functional programming

Fundamental concept: application of (mathematical) functions to values

- 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
    - $\Rightarrow$ all vars in function must be local (or parameters)

Pure Functional Languages

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

Pure Functional Languages

Control flow is governed by function calls and conditional expressions

- no iteration
- recursion is widely used

Pure Functional Languages

All storage management is implicit

- needs garbage collection

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:

- A set of function definitions
- 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

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 arg 1 arg 2 ...arg n)
- (+ 4 5)
- (+ (* 3 4 5) (- 5 3))

Operational semantics: In order to evaluate an expression:

- evaluate function to a function value
- evaluate each arg i in order to obtain its value
- apply the function value to these values

S-expressions

$$\langle \textit{S-expression} \rangle ::= \langle \textit{Atom} \rangle \mid \text{`('} \; \{\langle \textit{S-expression} \rangle\} \; \text{`)'}$$

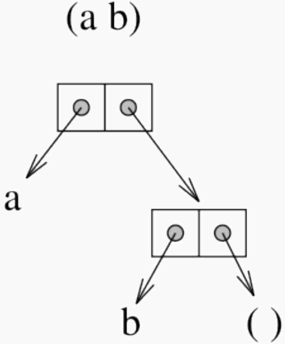
$$\langle \textit{Atom} \rangle ::= \langle \textit{Name} \rangle \mid \langle \textit{Number} \rangle \mid \texttt{\#t} \mid \texttt{\#f}$$

```
#t
()
(a b c)
(a (b c) d)
((a b c) (d e (f)))
(1 (b) 2)
```

Lists have nested structure.

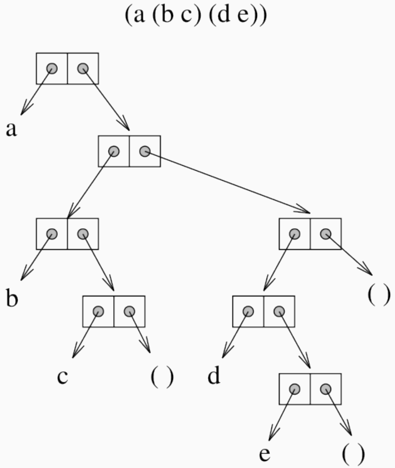
Lists in Scheme

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



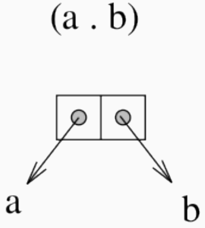
Lists in Scheme

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



Lists in Scheme

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:

- (+ 5 3) = 8, (- 5 3) = 2
- (* 5 3) = 15, (/ 5 3) = 1.6666666

`= < >` comparison operators for numbers

Other Functions

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`

Other Functions

Note: Scheme is a strongly typed language.

- (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 (REPL)

The Scheme interpreters we'll be using on ilab are called racket and drracket. "drracket" is an interactive environment, "racket" is command-line.

Type racket, and you are in the REPL. Use ctrl-d to exit.

- READ: Read input from user: a function application
- EVAL: Evaluate input: (f arg_1 arg_2 ...arg_n)
  - evaluate f to obtain a function
  - evaluate each arg_i to obtain a value
  - apply function to argument values
- PRINT: Print resulting value: the result of the function application

Read-eval-print loop (REPL)

You can write your program in file myFile.rkt and then read it into the interpreter by saying at the interpreter prompt: (load "myFile.rkt")

REPL example

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

- Read the function application (cons 'a (cons 'b '(c d)))
- Evaluate cons to obtain a function
- Evaluate 'a to obtain a itself
- Evaluate (cons 'b '(c d)):
  - Evaluate cons to obtain a function
  - Evaluate 'b to obtain b itself
  - Evaluate '(c d) to obtain (c d) itself
  - Apply the cons function to b and (c d) to obtain (b c d)
- Apply the cons function to a and (b c d) to obtain (a b c d)
- 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 racket interpreter on the ilab cluster from within emacs by specifying:

```
(setq scheme-program-name "racket")
```

and executing the commands: Meta-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.

You can switch between windows by saying Ctrl-x o.

You can save the 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 functions

```
(define <name> (lambda (<params>) <expression>))
```

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

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

Defining Scheme Functions

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

Evaluating (atom? '(a)):

- Obtain function value for atom?
- Evaluate '(a) obtaining (a)
- Evaluate (not (pair? object))
    - Obtain function value for not
    - Evaluate (pair? object)
        - Obtain function value for pair?
        - Evaluate object obtaining (a)
        - Evaluates to `#t`
    - Evaluates to `#f`
- Evaluates to `#f`

Conditional Execution: if

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

- Evaluate condition
- If the result is a "true value" (i.e., anything but `#f`), then evaluate and return result1
- 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) '())))
```