Introduction to Functional Programming (Common Lisp)

Tutorial for

Programming Languages Laboratory (CS 431)

July to November -2016

Instructor: Dr. Samit Bhattacharya

Indian Institute of Technology Guwahati

Prelude

- Programming paradigms – broadly two types
  - Imperative & declarative
- Imperative – we are familiar with this paradigm (procedural/OOP)
  - Uses statements to change system state
  - Main concern – ‘how’ to do things
- Declarative – programming is done with ‘expressions’ or ‘declarations’ rather than statements
  - Main concern – ‘what’ to do, not ‘how’

Prelude

- Declarative paradigm
  - Logic programming – we have already seen it; centers around ‘relations’
  - Functional programming – going to discuss

Functional Programming

- Key idea – computation as "evaluation of mathematical functions"
  - Idea originated from Lambda Calculus formalism

Ex: let's compute (2*3 *5)

- You "simply" type in: (* 2 3 5)

  $(* 2 3 5)   // input

- You get 30   as output

Note the emphasis on 'ease' of problem solving ('what') rather than 'efficient' code implementation ('how') – programmers convenience!

History (brief)

- Lambda calculus (developed in 1930s) forms the core of FPLs
- Early LISP (1950s) – contained elements of FPL
  - later improvements - Scheme (1970s) and Common Lisp (ANSI standard)
- Many more FPLs developed – mostly of academic interest
- Haskell – a prominent language with open standards defined for FP (1990s)
- Its power in parallel programming makes it hot again
  - Google MapReduce built on FP

Lisp

- Lisp is the second oldest high-level programming language after Fortran

- Lisp stands for "LISt Processor"
  - Simple data structure (atoms and lists)
  - Heavy use of recursion
  - Interpretive language

- Common Lisp (de facto industrial standard)

LISP Working Environment

```
loop
    read in an expression from the console;
    evaluate the expression;
    print the result of evaluation to the console;
end loop.
```

Examples

- Note: assume the prompt to be “**$**”.

- Simple test
- $1 //input
- 1 // lisp output

- Compute (2+4)
- $ (+ 2 4) //input
- 6 // lisp output

Common Lisp – Getting Started

- We will be using Common LISP
  - The development of the original LISP language has been frozen for a long time
  - There are many implementations for Common LISP interpreter
    - CLISP, CMUCL, GNU Common Lisp, Emacs Lisp, …etc
  - The syntax of Common LISP is very much the same with the original LISP

- Downloading :
  - GNU Common LISP
    ftp://ftp.gnu.org/pub/gnu/gcl/binaries/stable/

Getting Started

- Command-line invocation

  Prompt$ clisp

  (+ 1 2)

        3

  “Hello”

        “Hello”

- Loading file

  (load “path/to/file”)

- Exiting Lisp:

  prompt$ (exit)

- Conventional lisp files have *.lisp* extension

Save/Load LISP programs

- Edit a lisp program:
  - Use a text editor to edit a lisp program and save it as, for example, helloLisp.lisp
- Load a lisp program:
  - (load "helloLisp.lisp")
- Compile a lisp program:
  - (compile-file "helloLisp.lisp")
- Load a compiled lisp program
  - (load "helloLisp")

Syntax

- Basic idea: everything written in the form of *symbolic expressions* (also called **s-expressions**)

- It is defined as
  - An *atom* or
  - An expression of the form (x y) where x & y are s-expressions themselves

Syntax

- Prefix notation

- Simple and clean syntax
  - Expressions are delimited by parentheses

- The same syntax is used for programs and for data
  - (1 2 3 4 5 10 “10”)
  - (a b c d e f)
  - (+ 1 2)
  - (subseq “abcdefg” 0 2)

Expressions and Evaluations

- (+ 3 4 5)
  - '+' is a function
  - (+ 3 4 5) is a function call with 3 arguments
- Arguments are evaluated
  - If any of the args are themselves expressions, they are evaluated in the same way
  - (+ 1 (+ 3 4))
- Turning off evaluation with Quote
  - '(+ 1 3) ----> (+ 1 3)

Evaluations

- LISP evaluates function calls in *applicative order* - all the argument forms are evaluated before the function is invoked
  - e.g. (+ (sin 0) (+ 1 5)) → (sin 0) and (+ 1 5) are respectively evaluated to the values *0* and 6 before they are passed as arguments to "+" function

- Numeric values are called *self-evaluating forms*: they evaluate to themselves

Data types

- Numbers – integers, ratio (exact fraction), floating point, complex numbers
- Characters - ASCII as well as Unicode
- Strings
- Symbols – a unique, named data object with several parts
  - *value cell* and *function cell* are the most important
  - When a symbol is evaluated, its value is returned
  - Some symbols evaluate to themselves (e.g., Boolean values represented by the self-evaluating symbols T and NIL)

Structures

- There were two types of data objects in the original LISP
  - Atoms
  - Lists

- List form
  - parenthesized collections of sub-lists and/or atoms
  - e.g., `(A B C)`, `(A B (C D) E)`

- LISP lists are stored internally as single-linked lists
  - Each cell is called “cons” and composed of two pointers: car (data) & cdr (next)

Variables

- Global variable values can be set and used during a session
- Declarations not needed

```
$(setq x 5)
   -> 5
$x
   -> 5
$(+ 3 x)
   -> 8
$(setq y “atgc”)
   -> “atgc”
```

Defining Symbols: SETQ

- `SETQ`
  - To set value to a symbol
  - E.g.

```
$(SETQ a 5)
  ->5
$a
  ->5
$(+ a 5)
  ->10
```

LISP Functions

- A function is expressed in the same way data is expressed
- Function form:
  - `(func_name arg_1 … arg_n)`
  - E.g. `(A B C)`, `(+ 5 7)`
- If the list `(A B C)` is interpreted as data
  - It is a simple list of three atoms, `A`, `B`, and `C`
- If it is interpreted as a function application
  - it means that the function named `A` is applied to the two arguments `B` and `C`

Functions for Constructing Functions: DEFUN

- DEFUN

  A Function for Constructing Functions

  Forms:

  ```
  (DEFUN func_name (arg_name) (func_body) )
  ```

  Ex.

  ```
  $ (DEFUN cube (x) (* x x x))
  $ (cube 2)
        ->8
  ```

Defining Symbol Within Function

- LET
  - To set value to a symbol temporarily within a function
  - E.g.

```
$ (DEFUN f (x)
        (LET (i 2)
             (j 3))
        (* x i j))
```

Built-in Functions

- **Printing:** `print, format`
  - `(print "string")` → print output
  - `(format …)` → formatted output
- **Predicates:** `listp, numberp, stringp, atom, null, equal, eql, and, or, not`
- **Special forms:** `setq/setf, quote, defun, defparameter, defconstant, if, cond, case, progn, loop`

Arithmetic Functions

➢ `+, -, *, /, ABS, SQRT, MOD, REMAINDER, MIN, MAX`

expression	value
`42`	`42`
`(+ 5 7)`	`12`
`(+ 5 7 8)`	`20`
`(- 15 7 2)`	`6`
`(- 24 (* 4 3))`	`12`

EVAL, QUOTE

- EVAL
  - A function that can evaluate any other functions
  - An implementation of `EVAL` could serve as a LISP interpreter

- QUOTE
  - It returns the parameter without evaluation
  - `QUOTE` can be abbreviated with the apostrophe prefix operator

    `'(A B)` is equivalent to `(QUOTE (A B))`

CAR, CDR

- CAR
  - Takes a list parameter; returns the first element of that list
  - e.g.,

    `(CAR '(A B C))` yields `A`

    `(CAR '((A B) C D))` yields `(A B)`
- CDR
  - Takes a list parameter; returns the list after removing its first element
  - e.g.

    `(CDR '(A B C))` yields `(B C)`

    `(CDR '((A B) C D))` yields `(C D)`

CONS, LIST

- CONS
  - takes two parameters, the first of which can be either an atom or a list and the second of which is a list;
  - returns a new list that includes the first parameter as its first element and the second parameter as the remainder of its result
  - e.g. `(CONS 'A '(B C))` returns `(A B C)`
- LIST
  - takes any number of parameters; returns a list with the parameters as elements
  - e.g. `(LIST A B C)` returns `(A B C)`

Predicate Function: `EQ`

- `EQ`
  - takes two symbolic parameters; it returns `T` if both parameters are atoms and the two are the same; otherwise `NIL`
  - e.g.,

    `(EQ 'A 'A)` yields `T`

    `(EQ 'A 'B)` yields `NIL`

    `(EQ 2 2)` yields `T`

Numeric Predicate Functions

- Comparison of numbers
  - `=, />, >, <, >=, <=`
  - e.g.,

    `(= 2 2)` yields `T`

    `(= 'A 'A)` yields error

- Type predicates for numbers
  - `ZEROP, NUMBERP, EVENP, ODDP, INTEGERP, FLOATP`

List Predicate Functions

- `NULL`
  - takes one parameter; it returns `T` if the parameter is an empty list; otherwise `NIL`
- `ATOM`
  - takes one parameter; it returns `T` if the parameter is an atom; otherwise `NIL`
- `LISTP`
  - takes one parameter; it returns `T` if the parameter is a list; otherwise `NIL`

Control Flow: `COND`

- `COND`
  - Forms:

    (COND
    (*predicate_1 body_1*)
    (*predicate_2 body_2*)
    . . .
)

  - Returns the value of the last expression in the first pair whose predicate evaluates to true

COND Example

```
$ (DEFUN compare (x y)
    (COND
      ((> x y) "x is larger")
      ((< x y) "y is larger")))
$ (compare 1 2)
    ->"x is larger"
```

Control Flow: `IF`

- `IF`
  - Form:

    (IF *predicate then_body else_body*)
  - Ex

```
$ (DEFUN avg (total count)
     (IF (= count 0)
        0
        (/ total count)))
$ (avg 100 5)
 ->20
```

Higher-order Functions

- A higher-order function, or *functional form*, is one that
  - either takes functions as parameters,
  - or yields a function as its result,
  - or both

- Some kinds of function forms
  - Function composition
  - Apply-to-all (map)
  - Construction
  - Reduce

EXAMPLES: APPEND

- It takes two lists; returns the first parameter list with the elements of the second parameter list appended at the end

```
$ (DEFUN my-append (li1 li2)
    (IF (NULL li1)
      li2
      (CONS (CAR li1) (my-append (CDR li1) li2))))
$ (my-append '(1 2) '(3 4))
   ->(1 2 3 4)
```

REVERSE

- It takes a list; returns a list that reverses the order of the elements in the given list

```
$ (DEFUN my-reverse (li)
   (IF (NULL li)
    '()
    (APPEND (my-reverse (CDR li)) (LIST (CAR li)))))
$ (my-reverse '(1 2 3))
   ->(3 2 1)
```

CONTAINS

- It takes a simple list and an atom as parameters; returns `T` if the atom is in the list; otherwise returns `NIL`

```
$ (DEFUN contains (li a)
    (COND
      ((NULL li) nil)
      ((EQ a (CAR li)) t)
      (t (contains (CDR li) a))))
$ (contains '(a b c) b)
   ->T
```

EQLI

- It takes two simple lists; returns whether the two lists are equal

```
$ (DEFUN eqli (li1 li2)
    (COND
       ((NULL li1) (NULL li2))
       ((NULL li2) nil)
       ((EQ (CAR li1) (CAR li2))
         (eqli (CDR li1) (CDR li2)))
       (t nil)))
```

*n*th Fibonacci number

```
(defun fib (n)
  "Computes the nth Fibonacci number."
  (cond ((or (not (integerp n)) (< n 0)) ; error case
         (error "~s must be an integer >= 0.~&" n))
        ((eql n 0) 0)           ; base case
        ((eql n 1) 1)           ; base case
        (t (+ (fib (- n 1))     ; recursively compute fib(n)
              (fib (- n 2))))))
```

Useful help facilities

- `(apropos 'str)` → list of symbols whose name contains 'str
- `(describe 'symbol)` → description of symbol
- `(describe #'fn)` → description of function
- `(trace fn)` → print a trace of fn as it runs
- `:a` → abort one level out of debugger

Myths and Facts about LISP

- Myth: Lisp runs interpreted only
  - Fact: All major Lisp implementations have compilers
- Myth: Lisp uses huge amounts of memory
  - Fact: Baseline Lisp installation requires 8-10MB
- Myth: Lisp is complicated
  - Fact: Lisp is much simpler and more elegant
  - however, it is often called *Lost In Stupid Parentheses*, or *Lots of Irritating Superfluous Parentheses* – so, be careful with the parenthesis

References

[1] Guy Cousineau, Michel Mauny and K. Callaway, *The Functional Approach to Programming,* Cambridge University Press; 1 edition (October 29, 1998)

[2] Patrick Henry Winston and Bertbold Klaus Paul Horn, *Lisp*, Pearson Education, 2000.

[3] David S. Touretzky, *Common Lisp: A Gentle Introduction to Symbolic Computation,* Dover Publications, 2013.

[4] http://www.tutorialspoint.com/lisp/

END