Lisp

Aditya Thakur

ECS 140A Programming Languages – Winter 2019

Lisp

- Functional programming language
- Developed by John McCarthy at MIT in late 1950's
- Originally intended for AI applications
- Stands for "LISt Processing"
- Many variants
  – MacLisp
  – Scheme (quite different)
  – Franz Lisp
  – Common LISP
- We'll use Common LISP and study some main features

Outline

- Basic list operations (`car`, `cdr`, `cons`, `list`)
- User-defined functions (`defun`)
- Predicates (`atom`, `null`, `equal`, etc.)
- Basic Lisp constructs
  - `let` and `let*`
  - `cond`
  - `if`
  - `quote`, `eval`, `funcall`, `apply`
- Lambda functions
- High-order functions (`mapcar`, `mapc`, `maplist`)
- Closures

s-expressions

- Everything (program & data) is a symbolic expression
- Two kinds of s-exprs
  - Atoms
    - Numbers: integers, reals, and complex

      `1, 3.1, 2044, -20, +6, 2.8e-10, #c(2.18 3.14)`
    - Symbols: names that can be bound to objects

      `(setq x 5)  ;; assign value 5 to symbol x`
  - Lists: constructed from atoms and lists
    - Empty list: `nil`, or `()`
    - Constructed lists: `(* 2 3)`, `(1 2 3)`, `(1 (a 2) 3)`

s-expressions

- All s-expressions evaluate to a value
  - Numbers evaluate to themselves
  - Symbols evaluate to the last bound value
  - Lists: first function, rest arguments to that function
  - Evaluating special forms such as `setq, quote,` etc.

Basic List Operations (I)

- Four of them: car, cdr, cons, list (many others)

car: return first element (head) of a list

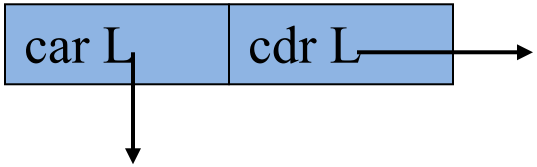
cdr: return the remaining elements (tail) of a list

```
> (car '(1 2 3))
1
> (cdr '(1 2 3))
(2 3)
```

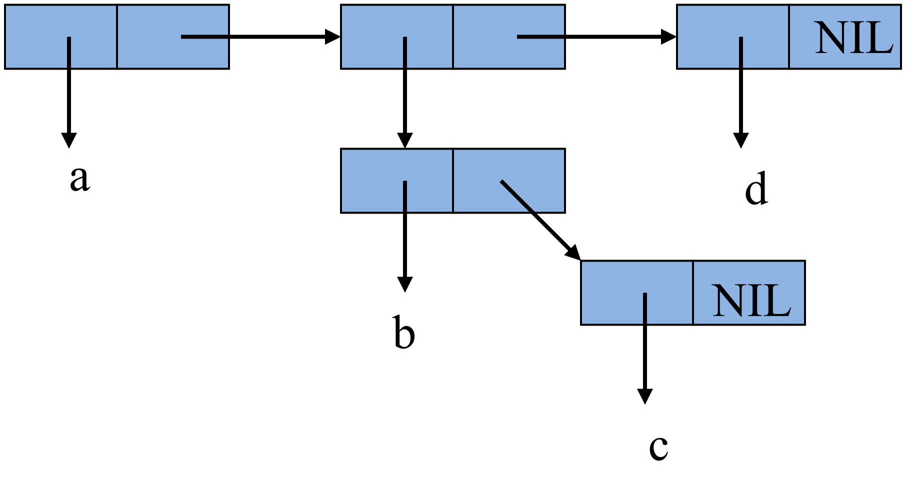
On the first machine where LISP was implemented, there were two instructions CAR and CDR which stood for "contents of address register" and "contents of decrement register"

Internal Representation of Lists

- Lists are represented as binary trees
  - Node containing two pointers
    - left to car
    - right to cdr



- Example: (a (b c) d) = (a . ((b . (c . nil)) . (d . nil)))



Basic List Operations (II)

- cons: A two field record

```
> (cons 1 2)    ;; allocate a cons, set the car to 1, and cdr to 2
(1 . 2)
> (cons 'a 'b)     ;; where cdr is an atom
(A . B)
```

- cons: builds a list

```
> (cons 'a nil)
(A)
> (cons 'a (cons 'b nil))
(A B)
```

Basic List Operations (III)

- list: constructs a list of any length

```
> (list 'a 'b 'c 'd '10)
(a b c d 10)
> (list 'a (list 'b 'c) 'd)
(a (b c) d)
```

User-Defined Functions (I)

- There are several ways and kinds of functions
- We will look at one: `defun`
- General form:

```
(defun funname (v1 v2 ... vn)  ;; formals
     (body1)   ;; just s-exprs
     ...
     (bodym)
)
```

- Binds function body to funname:

```
(defun double (x)
    (* 2 x)
)
```

User-Defined Functions (II)

- Function invocation via `(funname arg1 arg2 ... argn)`
  - evaluate arg1, arg2, ...
  - binds actuals to formals--must be the same number
  - evaluates body1, body2, ... in order
  - returns value of last body evaluated
  - unbinds
- Example:

```
> (defun sum (x y) (+ x y))
> (sum 2 3)

(defun fac (n)
   (if (> n 0)
       (* n (fac (- n 1)))
       1))
```

More Function Examples

- An identifier can be bound to a function and a value simultaneously:

```
> (defun b (x) (+ x 5))
 B
 > (setq b 3)
 3
 > (b b)
 8
```

Predicates (I)

- Used to test whether a condition holds or not
  - NIL: false
  - T: true
- Examples:

```
> (atom 3)
T
> (listp 3)
NIL
> (numberp 3)
T
> (null '(1 2))
NIL
> (zerop 0)
T
```

```
> (atom '(a b))
NIL
> (listp '(a b))
T
> (numberp 'b)
NIL
> (null nil)
T
> (zerop '(a b))
NIL
```

Predicates (II): Equality

- A few equality comparisons: eq, eql, equal
  - eq: shallow comparison of pointer values only
  - eql: true if same object, or numbers of same type and same value
  - equal: deep comparison to see if two objects printed the same
- A few notes
  - eql: the same as eq for symbols and = for numbers
  - equal: the same to eql for symbols and numbers
  - eq and eql work on symbols and numbers, not lists
  - equal also works on lists

Predicates (III): Examples

```
(setq a '(a))
(setq b a)
(eq a b) --> true
(eql a b) --> true  ;; pointer comparison, both a and b point
to the same list


(eq '(a) '(a)) --> false
(eql '(a) '(a)) --> false
(equal '(a) '(a)) --> true

(eql '(a) (cons 'a nil)) --> false
(equal '(a) (cons 'a nil)) --> true
```

`cond` Construct

- General form:

```
(cond (b1 e11 e12 ... e1m )
        …
      (bn en1 en2 ... enm))
```

- Evaluation:
  - First evaluate **`b1`**
  - If **`b1`** is **`true`**, evaluate **`e11, e12,…`** return the value of **`e1m`**
  - If **`b1`** is **`NIL`**, move on to **`b2`**
  - If all **`b1`** through **`bn`** are **`NIL`**, return **`NIL`**

cond Construct

- Example:

```
> (defun cond-example1 (x)
      (cond ((> x 5) (- x 1))
            ((eql x 5) x)
            ((< x 5) (+ x 1))))

> (cond-example1 3)
4
```

`if` Construct

- General form:

```
(if b1 e1 e2)
```

- Example:

```
> (defun if-example (x)
       (if (atom x) (list x) x))

> (if-example 4)
(4)

> (if-example ‘(a b))
(A B)
```

`let` Construct

- A control block is a section of code with local variables
  - Two ways to do this: let and let*

```
(let ((var1 val1)
            ...
      (varn valn))
    body)
```

  - Variables `var1,..., varn` are visible within the body
  - With let, evaluate each of the values independently and bind them to the variables, then evaluate the `body`

```
> (let ((a 3))
       (let ((a 4) (b a))
          (+ a b)))
7
```

`let*` Construct

- let* binds variables sequentially, so that once an identifier has a value bound to it, later evaluations of values may use that value

```
> (let ((a 3))
        (let* ((a 4) (b a))
           (+ a b)))
8
```

```
> (let* ((w (+ 3 4))
         (x (* w 3)))
      x)
21
```

Practice Exercise

Write a function that calculates the distance between two points:

$$d = \mathrm{sqrt}((x_2 - x_1)^2 + (y_2 - y_1)^2)$$

Solution:

```
(defun distance (p1 p2)
         (let ((xdiff (- (car p1) (car p2)))
               (ydiff (- (cadr p1) (cadr p2))))
              (sqrt (+ (* xdiff xdiff) (* ydiff ydiff)))))
```

Delay or Force Evaluation (I)

- Normally the LISP interpreter only evaluates and applies
- Lisp allows the user to control blocking and applying evaluations

– quote or ' delays default evaluation

```
> (cons 'a (cons 'b nil))
(a b)
> (cons 3 '(+ 5 6))
(3 + 5 6)
> (cons a '(b c))
(10 b c)   ;; assuming a evaluates to 10
> '('a)
('a)
```

– conversely, eval forces evaluation

```
> (setq a (list '* 4 5))
(* 4 5)
> (cons a (list 'evaluates 'to a))
((* 4 5) EVALUATES TO (* 4 5))
> (cons a (list 'evaluates 'to (eval a)))
((* 4 5) EVALUATES TO 20)
```

Delay or Force Evaluation (II)

- Suppose

```
> (setq a (cons 3 ''(+ 5 6)))
(3 QUOTE (+ 5 6))
```

- How do I get 11 from a?

```
> (eval (eval (cdr a)))
```

- Also force application
  - `apply`: 2 args, function and list of args
  - `funcall`: function and args (not in a list)

```
> (apply 'cons '(a (b c)))
> (apply '+ '(1 2 3 4 5))
> (apply '* 2 3 '(4 5 6))
> (funcall 'cons 'a '(b c))
```

Lambda and Anonymous Functions

- Lambda functions come from lambda-calculus
- Usually used for creating temporary functions
  - General form: `(lambda (args) (body))`
- Can be handed directly to apply, funcall, or mapcar

```
> (funcall (lambda (n) (+ n 1)) 3)
4
> (apply (lambda (a b c) (* a (+ b c))) '(4 3 5))
32
```

- Lambda expressions can be used to create local variables that store intermediate results

```
> ((lambda (temp) (f2 temp) (f3 temp)) (f1 x y z))
```

instead of

```
> (lambda (x y z) (f2 (f1 x y z)) (f3 (f1 x y z)))
```

High-Order Functions (I)

- Functions can be used as arguments
- Mapping applies a function to all elements of a list
  - `(mapcar func arglist1 ... arglistn)`
  - `func` must take n arguments
  - `func` is applied to each of the ith elements of the n lists
  - the list of results is returned

```
> (mapcar #'+ '(1 2 3) '(10 20 30))
(11 22 33)
> (mapcar #'equal '(1 2 3) '(1 3 a))
(T NIL NIL)
> (mapcar #'(lambda (x) (* x x)) '(1 2 3))
(1 4 9)
```

High-Order Functions (II)

- `mapc` evaluates the same, but returns the second argument
- `maplist` applies the function to whole lists and then successive cdrs

```
> (maplist #'cons '(a b) '(x y))
(((A B) X Y) ((B) Y))

> (maplist #'(lambda (x y) (mapcar '+ x y)) '(2 3 4) '(5 6 7))
((7 9 11) (9 11) (11))

> (defun f1 (op) ((lambda () (mapcar op '(2 3 4) '(5 6 7)))))
F1

> (F1 #'+)
(7 9 11)

> (F1 #'(lambda (x y) (+ x y 5)))
(12 14 16)
```

Practice Exercises

- Write a function **`length`** that takes a list and returns its length
- Write a function **`our-equal`** that takes two lists, and determines whether they are equal by checking each of their corresponding elements
- Write a function **`prefix`** so that:

```
> (prefix '(x x x) '(a b c d e))
(a b c)
```

Exercise Solutions (I)

- length

```
(defun length (l)
   (if (null l) 0
       (+ 1 (length (cdr l)))))
```

- equal (our equal)

```
(defun our-equal (x y)
   (or (eql x y)
       (and (consp x) (consp y)  ;; check whether
 x,y are non-nil
            (our-equal (car x) (car y))
            (our-equal (cdr x) (cdr y)))))
```

Exercise Solutions (II)

- prefix (prefix of a list)
  - The first argument describes the length of prefix

```
(defun prefix (block lst)
   (if (null block) '()      ;; empty block
       (if (null lst) '()      ;; empty list
           (cons (car lst)
                 (prefix (cdr block) (cdr lst))))))

> (prefix '(x x x) '(a b c d e))
(a b c)
```

Closure

- High-order functions
  - return a function as return value

```
(setq fn (let ((a 3))
              (lambda (x) (+ x a))))

> (funcall fn 2)
5
```

  - `fn` refers to the local binding that `a` is `3`
  - `a = 3` must exist as long as `fn` exists
- When a function refers to a variable defined outside of the function, it is called a *free variable*
- A function with its free lexical variable bindings is called a *closure*

Example: Timestamps

```
(let ((count 0))
     (defun reset ()
         (setq count 0))
     (defun stamp ()
         (setq count (+ count 1))))

> (list (stamp) (stamp) (reset) (stamp))
(1 2 0 1)
```

Practice Exercise

- Represent a binary tree as a list so that:
  - Left branch car
  - Right branch cdr

- Write a function `copy-tree` that makes a copy of the tree
- Write a function `count-leaf` that count the number of leaves in the tree

Exercise Solution

```
(defun copy-tree (tree)
    (if (atom tree) tree
        (cons (copy-tree (car tree))
              (if (cdr tree)
                  (copy-tree (cdr tree))))))

(defun count-leaf (tree)
    (if (atom tree) 1
        (+ (count-leaf (car tree))
           (or (if (cdr tree)
                   (count-leaf (cdr tree)))
               1))))

(count-leaf '((1 2 (3 4)) (5) 6))
10
```

Innovations in the Design of Lisp

- Expression-oriented
  - Function expressions
  - Conditional expressions
  - Recursive functions
- Abstract view of memory
  - Cells instead of array of numbered locations
  - Garbage collection
- Programs as data
- Higher-order functions

Summary: Contributions of Lisp

- Successful language
  - Symbolic computation, experimental programming
- Specific language ideas
  - Expression-oriented: functions and recursion
  - Lists as basic data structures
  - Programs as data, with universal function eval
  - Idea of garbage collection