Modern Programming Languages

Lecture 18-21

LISP Programming Language
An Introduction

Functional Programming Paradigm and LISP

Functional programming is a style of programming that emphasizes the evaluation of expressions, rather than execution of commands. The expressions in these languages are formed by using functions to combine basic values. A functional language is a language that supports and encourages programming in a functional style.

LISP is a representative language of this style of programming.

Lisp stands for “LISt Process”. It was invented by John McCarthy in 1958 at MIT. It has simple data structure (atoms and lists) and makes heavy use of recursion. It is an interpretive language and has many variations. The two most popular versions are Scheme and Common Lisp (de facto industrial standard). It is the most widely used AI programming language.

Valid Objects

A LISP program has two types of elements: atoms and lists.

Atoms:

Atoms include numbers, symbols, and strings. It supports both real numbers and integers.

symbols: a symbol in LISP is a consecutive sequence of characters (no space). For example **a, x,** and **price-of-beef** are symbols. There are two special symbols: **T** and **NIL** for logical true and false.

strings: a string is a sequence of characters bounded by double quotes. For example **"this is red"** is a string.

S-expression

An **S-expression** (S stands for symbolic) is a convention for representing data or an expression in a LISP program in a text form. It is characterized by the extensive use of prefix notation with explicit use of brackets (affectionately known as Cambridge Polish notation).

S-expressions are used for both code and data in Lisp. S-expressions were originally intended only as machine representations of human-readable representation of symbols, but Lisp programmers soon started using S-expressions as the default notation.

S-expressions can either be single objects or atoms such as numbers, or lists.

Lists
List is the most important concept in LISP. A list is a group of atoms and/or lists, bounded by **(** and **)**. For example **(a b c)** and **(a (b c))** are two lists. In these examples the list **(a b c)** is a list of atoms **a, b,** and **c,** whereas **(a (b c))** is a list of two elements, the first one an atom **a**, and the second one a list **(b c)**.

Top elements of a list

The first-level elements in LISP are called top-level elements. For example top elements of list (a b c) are a, b, and c. Similarly, top elements of list (a (b c)) are a and (b c).

An empty list is represented by **nil.** It is the same as **()**.

Function calls

In LISP, a function and a function call is also a list. It uses prefix notation as shown below:

```
(function-name arg1 ... argn)
```

A function call returns function value for the given list of arguments. Functions are either provided by LISP function library or defined by the user.

Following are some examples of function calls. Note that the first symbol is the name of the function and the rest are its arguments.

```
>(+ 1 3 5)
9
```

Note that the arithmetic functions may take more than two arguments.

```
>(/ 3 5)
3/5
```

When we use two integers in division, the answer is converted into fraction.

```
>(/ 3.0 5)
0.59999999999999998
```

Note the difference between integer and real division.

```
>(sqrt 4)
2
```

Evaluation of S-expression

S-expressions are evaluated as follows:

1) Evaluate an atom.
   - numerical and string atoms evaluate to themselves;
   - symbols evaluate to their values if they are assigned values, return Error, otherwise;
   - the values of T and NIL are themselves.

2) Evaluate a list
   - evaluate every top element of the list as follows, unless explicitly forbidden:
   - the first element is always a function name; evaluating it means to call the function body;
   - each of the rest elements will then be evaluated, and their values returned as the arguments for the function.

Examples:

1.
```
>(+ (/ 3 5) 4)
23/5
```

The first element is + so we make a call to + function with (/ 3 5) and 4 as arguments. Now (/ 3 5) is evaluated and the answer is 3/5. After that 4 is evaluated to itself and thus returns 4. So, 3/5 and 4 are added to return 23/5.

2.
```
>(+ (sqrt 4) 4.0)
6.0
```

This example is similar to the first example.

3.
```
>(sqrt x)
Error: The variable
        X is unbound.
```

This code generates an error as the symbol x is not associated with any value and thus cannot be evaluated.

setq, set, and **setf** are used to assign a value to a symbol.

For example, the following code assigns a value 3 to symbol x.

```
>(setq x 3.0)
 3.0
```

setq is a special form of function (with two arguments). The first argument is a symbol which will not be evaluated and the second argument is a S-expression, which will be evaluated. The value of the second argument is assigned to be the value of the first argument.

To forbid evaluation of a symbol a **quote** or **’** is used.

For example

```
>(quote x)
 x
```

Similarly if we have

```
>(setq x 3.0)
 3.0
```

and then we do

```
>(setq y x)
3.0
```

then y will also have a value 3.0

But if we use a quote as shown below

```
<(setq y ‘x)
x
```

then x will not be evaluated and y will get the value x instead of 3.0.

Functions

There are many built-in function is LISP. This includes math functions as well as functions for manipulating lists. The math functions include:

- +, -, *, /, exp, expt, log, sqrt, sin, cos, tan, max, min

with the usual meanings.

List Constructors

Lists can be constructed (created and extended) with the help of three basic functions. These are **cons, list** and **append.**

cons takes two arguments. The first argument is a symbol and the second one is a list. It inserts the first argument in front of the list provided by the second argument. It is shown in the following example:

```
>(cons 'x L)       ; insert symbol x at the front of list L, which is
(X A B C)        ; (A B C)
```

list takes arbitrary number of arguments and makes a list of these arguments as shown below:

```
>(list 'a 'b 'c)     ; making a list with the arguments as its elements
 (A B C)           ; if a, b, c have values A, B, C, then (list a b c) returns list (A B C)
```

append takes two lists as arguments and appends one list in front of the other as shown below:

```
>(append '(a b) '(c d))
(A B C D)        ; appends one list in front of another
```

List Selectors

In order to select elements from a list, selectors functions are used. There are two basic selector functions known as **first** (or **car**) and **rest** (or **cdr**). The rest can be build with the help of these functions.

first (or **car**) takes a list as an argument and returns the first element of that list as shown in the following examples:

```
>(first '(a s d f))
 a

>(first '((a s) d f))
 (a s)
```

```
>(setq L '(A B C))
(A B C)

>(car L)
A
```

rest (or **cdr**) takes a list as its argument and returns a new list after removing the first element from the list. This is demonstrated in the following examples:

```
>(rest '(a s d f))
(s d f).

>(rest '((a s) d f))
 (d f)

>(rest '((a s) (d f))
 ((d f))

>(setq L '(A B C))
(A B C)

>(cdr L)
(B C)
```

Some of the other useful list manipulation functions are:

```
>(reverse L)     ; reverses a list
   (C B A)
```

and

```
>(length L)      ; returns the length of list L
 3
```

Predicates

A predicate is a special function which returns NIL if the predicate is false, T or anything other than NIL, otherwise. Predicates are used to build Boolean expressions in the logical statements.

The following comparative operators are used as functions for numerical values and return a T or NIL. **=**, **>**, **<**, **>=**, **<=**;

For example:

- ➢ (= (+ 2 4) (* 2 3))
- ➢ T

- ➢ (> (- 5 2) (+ 3 1))
- ➢ NIL

For non numeric values you can only check for equality using **equal** or **eq.**

Some other useful predicates are listed below:
atom: test if x is an atom
listp: test if x is a list

Also **number**, **symbolp**, **null** can be used to test whether the operand is a number, symbol, or a null value.

Set Operations

A list can be viewed as a set whose members are the top elements of the list. The list membership function is a basic function used to check whether an element is a member of a list or not. It is demonstrated with the help of the following code:

```
>(setq L '(A B C))

>(member 'b L) ; test if symbol b is a member (a top element) of L
(B C)          ; if yes, returns the sublist of L starting at the
                ; first occurrence of symbol b

>(member ‘b (cons 'b L))
(B A B C)
```

Note here that the mathematical definition of a set is different from the LISP definition. In Mathematics, a symbol cannot be repeated in a set whereas in LIST there is no such restriction.

If an element is not present in the list, it returns NIL as shown below.

```
>(member x L)
NIL            ; if no, returns NIL
```

Some of the other set operations are defined as below:

```
>(union L1 L2)            ; returns the union of the two lists
>(intersection L1 L2)     ; returns the intersection of the two lists
>(set-difference L1 L2)   ; returns the difference of the two lists
```

Defining LISP functions

In LISP, defun is used to write a user-defined function. Note that different dialects of LISP may use different keywords for defining a function. The syntax of defun is as below:

(**defun** func-name (arg-1 ... Arg-n) func-body)

That is, a function has a name, list of arguments, and a body. A function returns the value of last expression in its body. This concept is demonstrated with the help of the following example:

```
>(defun y-plus (x) (+ x y))            ;definition of y-plus
```

This function adds x to y where x is the parameter passed to the function and y is a global variable since it is not defined inside the function. It work in the following manner:

```
>(setq y 2)
>(y-plus 23)
25
```

With this we introduce the concept of local and global variables. Local variables are defined in function body. Everything else is global.

Conditional control: if, when and **cond**

LISP has multiple conditional control statements. The set includes **if**, **when**, and **cond**. In the following pages we study these statements one by one.

if statement

The **if** statement has the following syntax:

```
 (if <test> <then> <else>)
```

That is, an if statement has three parts: the **test**, the **then** part, and the **else** part. It works almost exactly like the if statement in C++. If the test is TRUE then the **then** part will be executed otherwise the **else** part will be executed. If there is no else part then if the test is not true then the if statement will simply return NIL. Here is an example that shows the **if** statement:

```
> (setq SCORE 78)
> 78
> (if (> score 85) 'HIGH
                    (if (and (< score 84) (> score 65)) 'MEDIUM 'LOW))
```

```
> MEDIUM
```

In the above **if** statement, the **then** part contains ‘HIGH and the **else** part is another **if** statement. So, with the help of nested **if** statements we can develop code with multiple branches.

cond statement

The **cond** statement in LISP is like the switch statement in C++. There is however a slight different as in this case each clause in the **cond** requires a complete Boolean test. That is, just like multiple else parts in the **if** statement where each needs a separate condition. Syntax of **cond** is as shown below:

```
>(cond (<test-1> <action-1>)
       (<test-2> <action-2>)
            ...
       (<test-k> <action-k>))
```

Each (<test-i> <action-i>) is called a clause. If test-i (start with i=1) returns T (or anything other than NIL), this function returns the value of action-i, else, it goes to the next clause. Usually, the last test is T, which always holds, meaning otherwise. Just like the **if** statement, **cond** can be nested (action-i may contain (cond ...))

This statement is explained with the help of the following example:

```
> (setf operation ‘area L 100 W 50)
> 50

> (cond ((eq operation ‘perimeter) (* 2 (+ L W)))
         (eq operation ‘area) (* L W))
         (t ‘i-do-not-understand-this)
        )
  )
> 5000
```

This program has three clauses. The first one checks whether the operation is the calculation of the perimeter or not. In this case it uses the length and width to calculate the perimeter. The control will come to the second clause if the first test is evaluated to be false or NIL. The second clause checks if the operation is about area and if it is true then it will calculate area. The third clause is the default case which will always be true. So if the first two tests fail, it will print **‘i-do-not-understand-this**

Recursion

Recursion is the main tool used for iteration. In fact, if you don't know recursion, you won't be able to go too far with LISP. There is a limit to the depth of the recursion, and it depends on the version of LISP. Following is an example of a recursive program in LISP. We shall be looking at a number of recursive functions in the examples.

```
(defun power (x y)
         (if (= y 0) 1 (* x (power x (1- y)))))
```

This function computes the power of x to y. That is, it computes x^y by recursively multiplying x with itself y number of times.

So

```
>(power 3 4)
81
```

Let us look at another example. In this example, we compute the length of a list, that is, we compute the number of elements in a list. The function is given as follows:

```
(defun length (x)
         (if (null x) 0
            (+ length (rest x) 1)
   )
)

> (Length '(a b c d))
> 4
```

Here are some more examples: The first function determines whether a symbol is present in a list or not and the second function computes the intersection of two lists. These functions are given below:

```
(defun member (x L)              ; determine whether x is in L or not
     (cond ((null L) nil)        ; base case 1: L is empty
          ((equal x (car L)) L)   ; base case 2: x=first(L)
          (t (member x (cdr L)))  ; recursion: test if x is in rest(L)
     ))

  (defun intersection (L1 L2)  ; compute intersection of two lists
     (cond ((null L1) nil)
            ((null L2) nil)
            ((member (car L1) L2)
            (cons (car L1) (intersection (cdr L1) L2)))
        (t (intersection (cdr L1) L2))
    ))
```

It may be noted that the intersection function is different from the mathematical definition of set intersection. In this case the function looks at all the elements of the first list, one at a time, and checks whether that element is present in the second list. You may recall from our earlier discussion that our definition of set if different from the mathematical definition of a set where duplications are not allowed. This concept is elaborated with the help of the following examples where you pass same lists in different order and get different results.

```
> (intersection '(a b c) '(b a b c))
> (a b c)

> (intersection '(b a b c) '(a b c))
> (b a b c)
```

Following is yet another example to compute the set difference of two sets.

```
(defun set-difference (L1 L2)
      (cond ((null L1) nil)
            ((null L2) L1)
            ((not (member (car L1) L2))
              (cons (car L1) (set-difference (cdr L1) L2)))
            (t (set-difference (cdr L1) L2))
      ))
```

Iteration: **dotimes** and **dolist**

Apart from recursion, in LISP we can write code involving loops using iterative non-recursive mechanism. There are two basic statements for that purpose: **dotimes** and **dolist.** These are discussed in the following paragraphs.

dotimes

dotimes is like a counter-control for loop. Its syntax is given as below:

(**dotimes** (count n result) body)

It executes the **body** of the loop n times where count starts with 0, ends with n-1.

The **result** is optional and is to be used to hold the computing result. If ***result*** is given, the function will return the value of ***result***. Otherwise it returns NIL. The value of the **count** can be used in the loop body.

dolist

The second looping structure is **dolist.** It is used to iterate over the list elements, one at a time. Its syntax is given below:

(**dolist** (x L result) body)

It executes the **body** for each top level element x in L. x is not equal to an element of L in each iteration, but rather x takes an element of L as its value. The value of x can be used in the loop body. As we have seen in the case of **dotimes,** the **result** is optional and is to be used to hold the computing result. If ***result*** is given, the function will return the value of ***result***. Otherwise it returns NIL.

To understand these concepts, let us look at the following examples:

```
> (setf cold 15 hot 35)
> 35
```

The following function use **dolist** to computes the number of pleasant days, that is, between cold and hot.

```
> (defun count-pleasant (list-of-temperatures)
                              (let ((count-is 0))                              ; initialize
                                (dolist (element my-list count-is)
                                     (when (and (< element hot)
                                                     (> element cold))
                                       (setf count-is (+ count-is 1))))))
```

```
> (count-pleasant ‘(30 45 12 25 10 37 32))
> 3
```

Here is a very simple example which uses **dotimes:**

```
> (defun product-as-sum (n m)
                      (let ((result 0))
                          (dotimes (count n result)
                             (setf result (+ result m)))))
```

```
> (product-as-sum 3 4)
> 12
```

Property Lists

Property lists are used to assign/access properties (attribute-value pairs) of a symbol. The following functions are used to manipulate a property list.

To assign a property: (**setf** (**get** object attribute) value)
To obtain a property: (**get** object attribute)
To see all the properties; (**symbol-plist** object)

Here is an example that elaborates this concept:

```
>(setf (get 'a 'height) 8)  ; cannot use "setq" here
                             ; setting the height property of symbol a.
8

>(get 'a 'height)  ; setting the height property of symbol a.
8

>(setf (get  'a 'weight) 20)
                             ; setting the weight property of symbol a.
20
```

Now we list all properties of **a**:

```
>(symbol-plist 'a)
(WEIGHT 20 HEIGHT 8)
```

We can remove a property by using the **remprop** function as shown below:

```
> (remprop 'a 'WEIGHT)
T

 >(symbol-plist 'a)
(HEIGHT 8)

> (remprop 'a 'HEIGHT)
T

 >(symbol-plist 'a)
NIL
```

We can use the property list to build more meaningful functions. Here is one example:

Assume that if the name of a person's father is known, the father's name is given as the value of the father property. We define a function GRANDFATHER that returns the name of a person’s paternal grandfather, if known, or NIL otherwise. This function is given below:

```
(defun grandfather (x)
        (if (get x ‘father)
                (get (get x ‘father) ‘father)
                nil))
```

We now make a list of the paternal line:

```
(defun lineage (x)
        (if  x
                (cons (x (lineage (get x ‘father))))))
```

The following would trace the family line from both father and mother’s side:

```
(defun ancestors (x)
        (if x
                (cons x (append (ancestors (get x ‘father))
                                (ancestors (get x ‘mother))))))
```

Arrays

Although the primary data structure in LISP is a list, it also has arrays. These are data type to store expressions in places identified by integer indexes. We create arrays by using the **linear-**array function as shown below:

```
                (setf linear-array (make-array ‘(4)))
```

This statement creates a single dimension array of size 4 by the name of **linear-array** with indices from 0 to 3.

Here is an example of a two-dimensional array:

```
                (setf chess-board (make-array ‘(8 8)))
```

Once created, we can store data at the desired index as follows:

```
                (setf (aref chess-board 0 1) ‘WHITE-KNIGHT)
                (setf (aref chess-board 7 4) ‘BLACK-KING)
```

Here, **aref** is the array reference. The above statements say that store symbol **WHITE-KNIGHT** at **chess-board** position **0 1** and store **BLACK-KING** at position **7 4**.

aref can be used to access any element in the array as shown below:

```
(aref chess-board 0 2)
WHITE-BISHOP

(aref chess-board 7 2)
BLACK-BISHOP
```

What made Lisp Different?

LISP was one of the earliest programming language. It was designed at MIT for artificial intelligence and it has since been the defacto standard language for the AI community, especially in the US.

LISP program composed of expression. They are in fact trees of expression, each of which returns a value. It has Symbol types: symbols are effectively pointer to strings stored in a hash table. One very important aspect of LISP is that the whole language is there. That is, there is no real distinction between read-time, compile-time and runtime. You can compile or run code while reading, read or run code while compiling, and read or compile code at runtime. That is, programs are expressed directly in the parse trees that get build behind the scenes when other languages are parsed, and these trees are make of lists, which are Lisp date structures. This provides a very powerful feature allowing the programmer *to write programs that write programs*.

From a programming language design point of view, LISP was the first to introduce the following concepts:

- Conditionals: such as if-then-else construct.
- Function types: where functions are just like integers and strings
- Recursion: first language to support it.
- Dynamic typing: all variable are pointers.
- Garbage-Collection.
- Programs composed of expressions.
- A symbol type.
- The whole program is a mathematical function