Lisp: An Overview

Shen-Shyang Ho
CZ3005: Artificial Intelligence and Intelligent Systems

Background: Lisp (LISt Processor)

1. Developed by John McCarthy in 1958.
2. Oldest Language still in use after FORTRAN.
3. Software written in Lisp: e.g. EMACS text editor.
4. Interpreted language: unlike compiled languages, you start an interpreter which can process and respond directly to programs written in LISP. (Well, we can compile LISP codes now)
5. The interpreter runs what is known as a *read-eval-print* loop - reads what you type, evaluates it, and then prints the result.

Basic Data Type: Atom and List

Atoms are represented as sequences of characters of reasonable length.

Lists are recursively constructed from atoms. This means that a given list may contain either atoms or other lists as members.

Examples:

ATOMS	LISTS
a	()
ho	(ho)
30	(a ho 30 cs580)
CS580	((ho 30) a ((cs580)))

One special entity, the empty list, known as ‘‘()’’ or ‘‘nil,’’ which is both an atom and a list.

Some Primitive Functions

Functions that are built into the LISP language are called "**primitive functions**".

e.g., +, -, *, /, exp, log, sqrt, sin, cos, tan, max, min.
Also, second, third, fourth, last, nthcdr, butlast, nbutlast, reverse, caar, cddr, cadr, cdar, etc.

Use the single quote mark, ', in front of an item that you do not wish the interpreter to evaluate. It works as follows:

> 'a
a

List Constructors: Cons, List, and Append; List Selectors: First and Rest

```
>(cons 1 nil)
(1)
>(cons 1 (cons 2 nil))
(1 2)
>(list 2 3 4)
(2 3 4)
> (list 'a '(a s d f))
(a (a s d f))
> (append '(a b) '(c d))
(a b c d)
> (first '(a s d f))
a
> (first '((a s) d f))
(a s)
> (rest '(a s d f))
(s d f)
> (rest '((a s) d f))
(d f)
> (rest '((a s) (d f)))
((d f))
```

Predicates

Predicates are **functions that always return either t or nil**. Atom is a predicate that determines whether its argument is an atom. Listp returns t if its argument is a list, and nil otherwise.

> (atom 'a)
T
> (atom a)
NIL
> (listp 'a)
NIL
> (listp a)
T

More: = (and more), <, integerp, null

Defining Function: defun

A function definition looks like this:
(defun <name> <parameter-list> <body>)

Example:
>(defun square (x) (* x x))
SQUARE
> (square 2)
4
> (square 1.4142158)
2.0000063289696399

More advanced programming allows the use of &rest, &optional, &key in the parameter list to permit variable numbers of arguments

Good programming style:

1. keep the body of a function reasonably short (short enough to read on one screen, for example).
2. building small functions that perform specialized tasks and then using those as building blocks for more complicated tasks.

Conditional Control: If and Cond

if statement has the form:

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

```
(defun absdiff (x y)
    (if (> x y)
        (- x y)
        (- y x)))
```

But things can get a lot worse if you want to have a **long chain of test conditions**. Use **cond** for such situations. The general form of a cond statement is as follows:

```
(cond (<testa> <form1a> <form2a> ... <resulta>)
    (<testb> <form1b> <form2b> ... <resultb>)
    ...
    (<testk> <form1k> <form2k> ... <resultk>))
```

```
(defun absdiff (x y)
    (cond ((> x y) (- x y))
        (t (- y x))))
```

Local variable: let

The general form of a **let** statement is:

```
(let ((<vbl1> <expr1>) ..............  (<vbln> <exprn>))
     <body>)
```

Example:

```
>(let ((x 3)
       (y (- 67 34)))
       (* x y))
99
```

Try to use local variable rather than global variable.

Recursion and Iteration

When should you use iteration, and when use recursion? There are (at least) these three factors to consider:

1. Iterative functions are typically faster than their recursive counterparts. So, if **speed** is an issue, you would normally use iteration.
2. If the **stack limit** is too constraining then you will prefer iteration over recursion.
3. Some procedures are very naturally programmed recursively, and all but unmanageable iteratively. Here, then, the choice is clear.

Many artificial intelligence tasks involve searching through **nested structures**. For example, tree representations of the moves in a game are best represented as a nested list. **Searching the tree involves recursively tracking through the tree**. For this kind of application, recursive function definitions are an essential tool.

Iteration using dotimes and dolist

```
General Form:        (dotimes (<counter> <limit> <result>) <body>)
```

```
 > (defun fun7 (x)
        (let ((z '()))
            (dotimes (y x (reverse z))
            (setf z (cons (* y y) z)))))
 > fun7 10
 (0 1 4 9 16 25 36 49 64 81)
```

```
General Form: (dolist (<next-element> <target-list> <result>)
                        <body>)
```

```
 > (defun fun8 (&optional (lst '(1 2 3 4 5 6 7 8 9 10)))
        (let ((z '()))
        (dolist (y lst z)
        (setf z (cons (* y y) z)))))
 >  (fun8)
    (100 81 64 49 36 25 16 9 4 1)
 >   (fun8 '(2 4 6))
    (36 16 4)
 >   (reverse (fun8 '(2 4 6)))
    (4 16 36)
```

Recursions on Simple List

RULE OF THUMB:

1. When recurring on a list, do three things:
   (a) check for the termination condition;
   (b) use the first element of the list;
   (c) recur with the "rest" of the list.
2. If a function builds a list using "cons", return () at the terminating line.

```
;; Takes a list and returns a count of all the top level elements
in the list.
 (defun length1 (lst)
        (cond ((null lst) 0)
              (t (+ 1 (length1 (rest lst))))))
;; takes a list and an element, and  returns the original list
with the first occurrence of the element removed
 (defun remove (lst elt)
        (cond ((null lst) nil)
              ((equal (first lst) elt) (rest lst))
              (t (cons (first lst)
                       (remove (rest lst) elt)))))
```

Recursion on Nested Lists and Expressions

RULE OF THUMB:

1. check for the termination condition;
2. check if the first element of the list is an atom or a list;
3. recur with the "first" and the "rest" of the list.

```
;; takes a nested list and an atom, and  it returns 't if it finds
;; the atom within the nested list and returns nil otherwise.
        (defun search (lst elt)
             (cond ((null lst) nil)
                   ((atom (first lst))
                    (if (equal (first lst) elt)
                        't
                        (search (rest lst) elt)))
                   (t (or (search (first lst) elt)
                          (search (rest lst) elt)))))
```

More Data structures

We will not discuss here!!

* Association Lists
* Property Lists
* Arrays, Vectors, and Strings
* Defstruct

Printing Output

```
>(print 'this)
THIS                              ;; printed
THIS                              ;; value returned
>(print (+ 1 2))
3                                 ;; printed
3                                 ;; returned
>(+ (print 1) (print 2))
1                                 ;; first print
2                                 ;; second print
3                                 ;; returns sum
>(prin1 "this string")
"this string"          ;; printed
"this string"          ;; returned
>(princ "this string")
this string            ;; no quotes can be more readable
"this string"          ;; string returned
```

Nicer Printing using Format - (1)

```
General:    (format <destination> <control-string>
            <optional-arguments>)
With t as the specified destination, and no control sequences in
the control-string, format outputs the string in a manner similar
to princ, and returns nil.
>(format t "this")
this
NIL

With nil as destination and no control sequences, format simply
returns the string:
>(format nil "this")
"this"

Inserting ~% in the control string causes a newline to be output:
>(format t "~%This shows ~%printing with ~%newlines.~%")

This shows
printing with
newlines.
 NIL
```

Nicer Printing using Format - (2)

~s indicates that an argument is to be evaluated and the result inserted at that point. Each ~s in the control string must match up to an optional argument appearing after the control string.

Here is an example of a function that uses this capability:

```
(defun f-to-c (ftemp)
  (let ((ctemp (* (- ftemp 32) 5/9)))
    (format t
            "~%~s degrees Fahrenheit is ~%~s degrees Celsius~%"
            ftemp                    ;; first ~s
            (float ctemp))           ;; second ~s
    ctemp))                          ;; return ratio value
```

(Float converts a number from integer or ratio to a floating point number, i.e. one with a decimal point.)

```
>(f-to-c 82)
82 degrees Fahrenheit is
27.777777777777777 degrees Celsius
 250/9
```

Eval, Funcall, Apply

```
(apply (function +) '(1 2 3))

(apply #'+ '(1 2 3))

;; funcall does the same thing as apply ;; but does not require
the arguments to be packaged in a list

(funcall #'+ 1 2 3)

;; eval takes an expression, evaluates it and returns its value

(eval '(+ 1 2 3))
```

Mapcar and Lambda Expression

```
;; A tradition: Lambda expression

;; In earlier dialects of Lisp,functions were represented
 ;; internally as LISTS and the way to tell a function
;; from an ordinary list was to check if the first element
 ;; was the symbol "lambda".

> (apply #'(lambda (x y) (+ x y)) '(4 5))
> 9

mapcar: takes a function and one or more lists and returns
 ;; the result of applying the function to elements taken from each
 ;; list, until list runs out.

(defun fun9 () (mapcar #'(lambda (x) (+ x 10)) '(1 2 3)) )
> (fun9)
> (11 12 13)
(defun fun10 () (mapcar #'rest '((1 a) (2 b) (3 c))) )
> (fun10)
> ((a) (b) (c))
```