Symbolic Programming in LISP

Prof. Mauro Gaspari

Dipartimento di Informatica Scienze e Ingegneria (DISI)

mauro.gaspari@unibo.it

LISP

- Invented by John McCarthy in 1958

```
(defun factorial (n)
   (if (<= n 1)
       1
       (* n (factorial (- n 1)))))
```

- Common LISP 1984

Why LISP

- AI (programming) languages features.
- Still the best programming language for symbolic programming.
- Lisp like syntax and notations are often used in AI systems.
- Introducing basic symbolic programming techniques.

Functional languages

- In functional languages programs are expressed as a set of function definitions:

```
f1(X,Y) = +(X,Y)
f2(X) = f1(X,X)
f3(Y,X) = f2(f1(Y,X))
```

- Program execution is expressed as a sequence of function evaluations:

```
f2(3) -> f1(3,3) -> +(3,3) -> 6
```

Evaluation rules

- **Normal evaluation** of a lambda expression is the repeated application of the leftmost reducible function application. In other words, normal evaluation is the strategy that substitutes the function definition without evaluating the arguments: (call-by-name).
- **Applicative evaluation** means that a function's arguments are evaluated before the function is applied. In other words, with applicative-order evaluation, internal reductions are applied first, and only after all internal reductions are complete, the function is reduced: (call-by-value).

Examples

Function: λ `X,Y. +(X,Y)`

Arguments: `X=*(5,2) e Y=+(1,3)`

Normal evaluation: (λ `X,Y. +(X,Y)) *(5,2) +(1,3)`

`+(*(5,2) ,+(1,3))`

`+(10,4)`

`14`

Applicative evaluation: (λ `X,Y. +(X,Y)) *(5,2) +(1,3)`

(λ `X,Y. +(X,Y)) 10 4`

`+(10,4)`

`14`

LISP Evaluation Rule

- The applicative evaluation rule cannot be use for evaluating condictional functions like: if(Condition,Then,Else) because both the Then and Else parts are always evaluated.
- These constructs may cause inefficiencies and termination problem when coupled with recursion.
- Lisp adopts the applicative evaluation rule and introduces an ad-hoc implementation for conditionals and other special forms.

Lisp Interpreter

```
unix# sbcl
* 486
486
* (+ 2 3)
5
* (* 5 99)
495
* (a b c)
.....undefined function: a.....
* (quote (a b c))
(a b c)
* '(a b c)
(a b c)
* ()
NIL
* NIL
NIL
* (quit)
unix#
```

Lisp is a tool used to illustrate
several concepts,
t's not a tool for programming.

- * is the sbcl prompt
- The lisp interpreter always tries to evaluate the input.
- The quote function (‘ as short form) suspend evaluation for its argument.

Lisp building blocks

Lisp Symbols

- Symbols in lisp are used for several purposes, as variables, function names, paramethers and for representing symbolic information.

```
* foo
....The variable FOO is unbound....
* ‘foo
FOO
* (symbol-name 'foo)
"FOO"
* (set 'foo 5)
5
* foo
5
* (setq foo 6)
6
* (symbol-value 'foo)
6
```

Lisp has not static types: the type of an entity is determined at runtime.
To derive the type of a given entity, Lisp exploits several predicates.

A Lisp symbol have several properties associated to them like the name and the value. Lisp is not case sensitive.

Defining Functions

white spaces separate the parameters

```
(defun FUN-NAME (PAR1 PAR2 ... PARm)
                (EXPR1)
                (EXPR2)
             ....
                (EXPRn) )
```

The BODY of a function is a sequence of expressions..

For functions without parameters m=0 we use the empty list:

```
(defun  FUN-NAME  ()
          (EXPR)
          )
```

A function returns the value of the last expression: EXPRn.

```
* (defun square (X) (* X X))
square
* (square 21)
441
* (square (+ 2 5))
49
```

There is no "return" expression: Lisp always assumes that the last value evaluated in a function has to be returned.

Examples

```
* (defun sum-of-squares (X Y)
            (+ (square X) (square Y)))

sum-of-squares

* (sum-of-squares 3 4)


25
* (defun f (a)
            (sum-of-squares (+ a 1) (* a 2)))

F

* (f 5)

136
```

Conditionals

condition part   expression part

```
(cond  (<p1> <e1>)
       (<p2> <e2>)
       (<p3> <e3>)
        ...
       (<pn> <en>))
```

Very important topic: conditional allows us to implement recursion.

- The cond special form contains a sequence of pairs (Condiction Expression).
- Pairs are evaluated in sequence.
- When the first condition succeeds (pi) the corresponding expression is evaluated (ei) and cond returns its value.
- If none of the conditions are true cond returns NIL (NIL represents false in Lisp).

Everything that is different from NIL is True.

Example

The function **(= A B)** works only with numbers.

$$\text{Abs}(X) \begin{cases} \mathbf{X} & \text{if } X>0 \\ \mathbf{0} & \\ \mathbf{-X} & \text{if } X<0 \end{cases}$$

- Implementation in Lisp:

```
(defun abs (X)
        (cond ((> X 0) X)
              ((= X 0) 0)
               ((< X 0) (- X))))
```

Recursion

The function **(eq A B)** returns true (T in lisp) in A and B are the same, NIL otherwise.

<u>Factorial:</u>

$$\texttt{fact(N)} \begin{cases} \mathbf{1} \texttt{ if N=0} \\ \\ \texttt{N*fact(N-1)} \end{cases}$$

The Lisp implementation is immediate:

```
* (defun fact (N)
       (cond ((eq N 0) 1)
             (T (* N (fact (- N 1))))))

fact
*  (fact 3)
6
```

T = True

Example

Fibonacci:

$$
\mathrm{Fib}(N) \begin{cases} 0 \text{ se } N=0 \\ 1 \text{ se } N=1 \\ \mathrm{Fib}(N-1) + \mathrm{Fib}(N-2) \text{ altrimenti} \end{cases}
$$

```
(defun fib (N)
       (cond ((eq N 0) 0)
             ((eq N 1) 1)
             (T (+ (fib (- N 1)) (fib (- N 2))))))
```

LIST Processing

- Creating lists:

constructor function

```
*   (cons 'a 'b)
(a . b)
```

first rest (like Prolog "head" and "tail" -> [H|T])

```
*   (cons a NIL)
(a)
*   (cons a (cons b NIL))
(a b)
```

The notation (a . b) indicates a CONS CELL: A cell that represents an element of a list including a value and the pointer to the rest of the list.

(a) = (a . NIL)

(a b) = (a . (b . NIL))

pay attention!

- Accessing lists:

```
* (car (cons 'a 'b))    #first
a
* (cdr (cons 'a 'b))    #rest
b
```

Examples

```
* (setq mylist (cons 'a (cons 'b (cons 'c ())))))
(a b c)
* (car (cdr (car (cdr '((1 2) (3 4)))))))
4
```

equivalent

```
* (cadadr '((1 2) (3 4)))
4
* mylist
(a b c)
```

Member function

```
(defun member (A L)
       (cond  ((null L) NIL)
              ((eq A (car L)) L)
              (T (member A (cdr L)))
       ))
```

The function **(null A)** returns true (T in lisp) in A is NIL, NIL otherwise.

```
* (member 'd '(a b c d))
```

(d)

```
* (member 'c '(a b c d))
```

(c d)

```
* (member '(1 2) '((3 4) a b '(1 2) c))
```

NIL

Comparing Lists

it's a predicate which indicates an "atom"

```
(defun equal (X Y)
        (cond ((and (atom X) (atom Y))
                    (eq X Y))
              ((equal (car X) (car Y))
               (equal (cdr X) (cdr Y)))
              (T NIL)
        )
```

```
* (equal '(a (a b) c) '(a (a b) c))
T
```

The function **(and A B)** returns true (T in lisp) if both A and B are different from NIL, NIL otherwise.

The function **(atom A)** returns true (T in lisp) if A is an atom (number, symbol, NIL), NIL otherwise.

Append and mapcar

Appends the two list L1 and L2:

```
(defun append (L1 L2)
   (cond ((null L1) L2 )
         (T (cons (car  L1) (append (cdr L1) L2))
   ))
```

Mapcar maps the function F on all the arguments of list L building the list of results.

```
(defun mapcar (F L)
      (cond ((eq L NIL) NIL)
            (T (cons (funcall F (car L)) (mapcar F (cdr L))))))
```

Lisp uses applicative reduction, while prolog uses resolution (and so unification).

Examples

The first argument of mapcar can be a quoted symbol associated to a function or a lambda expression:

```
* (append '(a b c)  '(d e))
(a b c d e)
* (defun square (X) (* X X))
square
* (symbol-function 'square)
#<FUNCTION SQUARE>
* (mapcar 'square '(1 2 3 4 5))
(1 4 9 16 25)
* (mapcar (lambda (X) (* X 10)) '(1 2 3 4 5))
(10 20 30 40 50)
```

mapcar is a second order function which builds a list applying its first parameter (a function) on its second parameter (a list).

lambda is a keyword in Lisp: it allows us to create a function without name (the lambda function in other programming languages).

Macros

- Lisp macros allow to define new operators (special forms) that are implemented by transformation.
- They can be uses when the applicative reduction rules is not adequate.
- Although a macro definition is similar to a function definition it works differently.
- A function produces results, while a macro produce expressions which when evaluated produce results.

Defining macros

- Example: if we ~~wont~~ WANT to define a new conditional form:

  (NEWIF PREDICATE THEN ELSE)

  we cannot implement this form as a function as follows, because both e1 and e2 will be evaluated in the function call independently from the value of p:

```
(defun newif (p e1 e2)
   (cond (p e1) (T e2)))
```

this definition of newif cannot be used, because both the parameters (e1 and e2) are evaluated when newif is called, potentially causing an infinite loop.

Newif Macro

- For implementing a macro the transformation should be clear:

  (newif p e1 e2) ===> (cond (p e1)(T e2))

- This transormation can be obtained with the followin code:

```
(defmacro newif (p e1 e2)
    (cons (quote cond) (cons (cons p (cons e1 NIL))
                             (cons (cons T (cons e2 NIL)) NIL)))
```

equivalent

- Another compact solution using backquote operator and commas:

```
(defmacro newif (p e1 e2)
        `(cond (,p ,e1) (T ,e2)))
```

backquote operator suspends the evaluation (like quote), but the expression after the commas are evaluated and inserted in the list.

The ` backquote operator is similar to quote but it can include entry points: the expressions after commas are evaluated and then inserted in the list.

Expanding Macros

- When a macro call appears in a program,
    1. the input expressions are transformed according to the macro definition (macroexpansion).
    2. the resulting expression is evaluated.

- Macroexpand-1: allows programmers to check the generated code:

* (macroexpand-1 '(newif 1 2 3))

(COND (1 2) (T 3))

T

We are not goint to programm in Lisp in this module.

Tools

- SBCL (Steel Bank Common Lisp):

  http://www.sbcl.org/

- CLISP an ANSI Common Lisp:

  https://sourceforge.net/projects/clisp/

- Commercial tools: Lispworks, allegroCL.

- Clojure a Java based implementation of a Lisp dialect (not a common Lisp):

  https://clojure.org/

They won't be in the exam!

Exercises

1. Build the list `(1 (3 8) (4 . 2) ((7 2)))` using cons.
2. Access to number 7 in the above list using car and cdr (or first and rest).
3. Implement the function (exp M N) that computes the exponential M^N, using recursion, cond and *.
4. Define the reverse function which takes a list as input and returns the reverse.
5. Define the (Xor A B) function using a macro (Xor is the esclusive or).

References

- Common Lisp The Language (Guy L. Steele): a Common Lisp Manual avaliable on line.

  https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html

- On Lisp: Advanced Techniques for Common Lisp (Paul Graham), Prentice Hall, 1994.