Logic and Functional Programming

Basic Elements of LISP

Dr. Cristian-Paul Bara

Computer Science Department
Faculty of Mathematics and Computer Science,
Babeş Bolyai University,
Cluj-Napoca, Romania

Table of Contents

Lambda functions.

- A function to be used one time that is too simple to be defined
- A function that must be dynamically synthesized, not defineable with `defun`
- definition: `(lambda <parameter-list> <body>)`
- Its arguments are evaluated when called. If the arguments should not be evaluated then `qlambda` should be used
- Application: `((lambda <parameter-list> <body>) <arguments>)`
- Examples:

```
((lambda (l) (cons (car l) (cdr l))) '(1 2 3)) = (1 2 3)
((lambda (l1 l2) (append l1 l2)) '(1 2) '(3 4)) = (1 2 3 4)
```

Lambda functions.

Define a function that takes a multi-level list and checks if there are no numeric atoms at top level.

```
(defun f(l)
    (cond
        ((null l) t)
        (((lambda (v)
            (cond
                ((numberp v) t)
                (t nil)
            )
        )
        (car l)
        ) nil)
        (t (f (cdr l)))
    )
)
```

Labels

A special form for defining local functions.

```
(labels ((fct (l) (cdr l))) (fct '(1 2))) = (2)
(labels
    ((temp (n)
        (cond
            ((= n 0) 0)
            (t (+ 2 (temp (- n 1))))
        ))
    )
    (temp 3)
) = (6)
```

Labels

Define a fucntion that takes a list of lists of atoms and checks if all sublists contains only numeric atoms

```
(test '((1 2) (3 4))) = T
(test '((1 2) (a 4))) = NIL
(test '((1 (2)) (a 4))) = NIL
```

Labels

Define a fucntion that takes a list of lists of atoms and checks if all sublists contains only numeric atoms

```
(DEFUN TEST (L)
    (COND
        ((NULL L) T)
        ((LABELS ((TEST1 (L)
                (COND
                    ((NULL L) T)
                    ((NUMBERP (CAR L)) (TEST1 (CDR L)))
                    (T NIL))))
            (TEST1 (CAR L)))
            (TEST (CDR L)))
        (T NIL)))
```

Using lambda functions to avoid repeated calls

```
(defun g (l)
    (cond
        ((null l) nil)
        (t (cons (car (f l)) (cadr (f l))))
    )
)
```

We can use an anonymous function to avoid callinf `(f l)` twice.

Using lambda functions to avoid repeated calls

Option 1.

```
(defun g (l)
    (cond
        ((null l) nil)
        (t ((lambda (v) (cons (car v) (cadr v))) (f l)))
    )
)
```

Using lambda functions to avoid repeated calls

Option 2.

```
(defun g (l) (
    (lambda (v)
        (cond
            ((null l) nil)
            (t (cons (car v) (cadr v)))
        )
    )
    (f l)
))
```

Using lambda functions to avoid repeated calls

Let's consider the function that generates the lists of all subsets of a list.

```
(defun subsets(l)
    (cond
        ((null l) (list nil))
        (t (append
            (subsets (cdr l))
            (pushFirst (car l) (subsets (cdr l)))
        ))
    )
)
```

Using lambda functions to avoid repeated calls

Let's consider the function that generates the lists of all subsets of a list.

```
(defun subsets(l)
    (cond
        ((null l) (list nil))
        (t (
            (lambda (s) (append s (pushFirst (car l) s)))
            (subsets (cdr l))
        ))
    )
)
```

Closures

- The combination between a function and the scope of its definition
- Closure
  - Function
    - Parameters
    - Body
  - Environment
    - Local Variables
    - Global Variables

Closures

Let's see the following evaluations

```
> (defun f () 10)
F
> (setq f '11)
11
> (f)
10
> f
11
> (function f)
#<CLOSURE F NIL (DECLARE (SYSTEM::IN-DEFUN F)) (BLOCK F 10)>
```

Closures

Let's see the following evaluations

```
> (setq g 7)
7
> (function g)
undefined function G
> (quote g) ;; equivalent to 'g
G
> (function car)
#<SYSTEM-FUNCTION CAR>
> (function (lambda (l) (cdr l)))
#<CLOSURE :LAMBDA (L) (CDR L)>
```

Closures

Notice that **`and`** and **`or`** are operators and not functions

```
> (function not)
#<SYSTEM-FUNCTION NOT>
> (function and)
undefined function AND
> (function or)
undefined function OR
```

Closures

From a syntax perspective CommonLisp requires the indication is a term is a function or a symbor

Function **f**	#'f	(function f)
Symbol **s**	's	(quote s)

There are also differences between dialects

Standard	#'f
	#'(lambda ...)
CLisp	'f
	(lambda ...)
GCLisp, Emacs Lisp, others	'f
	'(lambda ...)

The `eval` form

- Equivalent to calling the interpreter, returns the evaluation of its argument

```
(SETQ X '((CAR Y) (CDR Y) (CADR Y)))
(SETQ Y '(A B C))
(CAR X) --> (CAR Y)
(EVAL (CAR X)) --> A
```

The `eval` form

- Similar to working with pointers

```
(SETQ L '(1 2 3))
(SETQ P '(CAR L))
P --> (CAR L)
(EVAL P) --> 1
(SETQ B 'X)
(SETQ A 'B)
(EVAL A) --> X
(SETQ L '(+ 1 2 3))
L --> (+ 1 2 3)
(EVAL L) --> 6
```

The `eval` form

- Lisp does not evaluate the first element of a list, it applies it

```
(SETQ Q 'CAR)
(SETQ P 'Q)
(EVAL P) --> CAR
((EVAL P) '(A B C)) --> "Bad function when ..."
```

Functional forms

- There are situations when the form of a function is unknown, but has to be applied and we would nee simething like:
  `(<functional-expression> <parameter-list>)`
- `(apply <function> <parameter-list>)`

```
(APPLY #'CONS '(A B)) --> (A . B)
(APPLY (FUNCTION CONS) '(A B)) --> (A . B)
(APPLY #'MAX' (1 2 3)) --> 3
(APPLY #'+' (1 2 3)) --> 6
(DEFUN F(L) (CDR L))
(APPLY #'F '((1 2 3))) --> (2 3)
```

Functional forms

- There are situations when the form of a function is unknown, but has to be applied and we would nee simething like:
  `(<functional-expression> <parameter-list>)`
- `(apply <function> <parameter-list>)`

  ```
  (APPLY #'(LAMBDA (L) (CAR L)) '((A B C))) --> A
  (SETQ P 'CAR)
  (APPLY P '((A B C))) --> A
  (APPLY #'P '((A B C))) --> undefined function P
  (SETQ Q 'CAR)
  (SETQ P 'Q)
  (APPLY (EVAL P) '((1 2 3))) --> 1
  ```

Functional forms

- `funcall` is a variant of apply that allows the application of a functional form
- (funcall <function> <parameters>)

```
(FUNCALL #'CONS 'A 'B) --> (A . B)
(FUNCALL (FUNCTION CONS) 'A 'B) --> (A . B)
(FUNCALL #'MAX '1 '2 '3) --> 3
(FUNCALL #'+ '1 '2 '3) --> 6
(DEFUN F(L) (CDR L))
(FUNCALL #'F '1 '2 '3) --> (2 3)
(FUNCALL #'(LAMBDA (L) (CAR L)) '(A B C)) --> A
```

Functional forms

- `funcall` is a variant of apply that allows the application of a functional form
- `(funcall <function> <parameters>)`

```
(SETQ P 'CAR)
(FUNCALL P '(A B C)) --> A
(FUNCALL # 'P '(A B C)) --> undefined function P
(SETQ Q 'CAR)
(SETQ P 'Q)
(FUNCALL (EVAL P) '(1 2 3)) --> 1
```

Functional forms

- Take care when using **`and`** and **`or`** since they are not functions!

```
(APPLY #'AND '((T NIL))) --> undefined function AND
(APPLY #'OR '((T NIL))) --> undefined function OR
(FUNCALL #'AND '(T NIL)) --> undefined function AND
(FUNCALL #'OR '(T NIL)) --> undefined function OR
```

Functional forms

- The solution is defining a function that evaluates to apllying `and` and `or` to a the elements of a list with t/nil values

```
(defun SI(l)
    (cond
        ((null l) t)
        ; (t (and (car l) (SI (cdr l))))
        ((not (car l)) nil)
        (t (SI (cdr l)))
    )
)
(FUNCALL #'SI '(T NIL T)) --> NIL
(SI '(T T T)) --> T
```

Functional forms

```
(DEFUN F() #'(LAMBDA (x) (CAR x)))
(FUNCALL (F) '(1 2 3)) --> ???
(APPLY (F) '((1 2 3))) --> ???
(FUNCALL (FUNCTION (LAMBDA (x) x)) 1) --> ???
```

Functional forms

```
(DEFUN F() #'(LAMBDA (x) (CAR x)))
(FUNCALL (F) '(1 2 3)) --> 1
(APPLY (F) '((1 2 3))) --> 1
(FUNCALL (FUNCTION (LAMBDA (x) x)) 1) --> 1
```

Functional forms

```
(defun increment (x)
    (lambda (y) (+ x y))
)
(setq inc5 (increment 5))
; returns a function (closure) that adds 5 to its argument
(print (funcall inc5 3)) --> 8
```

MAP Functions.

- The role of a map function is to apply a function repeadedly onto the elements of a list
- `(mapcar f l1 l2 ...)`
- The rezults are grouped with `list`

```
(MAPCAR #'CAR' ((A B C) (X Y Z))) --> (A X)
(MAPCAR #'EQUAL' (A (B C) D) '(Q (B C) D X)) --> (NIL T T)
(MAPCAR #'LIST' (A B C)) --> ( (A) (B) (C) )
(MAPCAR #'LIST' (A B C) '(1 2)) --> ( (A 1) (B 2) )
(MAPCAR #'+' (1 2 3) '(4 5 6)) --> (5 7 9)
```

MAP Functions.

- The role of a map function is to apply a function repeadedly onto the elements of a list
- `(mapcan f l1 l2 ...)`
- The rezults are grouped with `nconc`

```
(SETQ L1 '(A B C) L2 '(D E)) --> (D E)
(APPEND L1 L2) --> (A B C D E)
L1 --> (A B C)
L2 --> (D E)
(SETQ L3 (NCONC L1 L2)) --> (A B C D E)
L3 --> (A B C D E)
```

MAP Functions.

- The role of a map function is to apply a function repeadedly onto the elements of a list
- `(mapcan f l1 l2 ...)`
- The rezults are grouped with `nconc`

```
(SETQ L1 '(A) L2 '(B) L3 '(C)) --> (C)
L1 --> (A)
L2 --> (B)
L3 --> (C)
(NCONC L1 L2 L3) --> (A B C)
L1 --> (A B C)
L2 --> (B C)
L3 --> (C)
```

MAP Functions.

- The role of a map function is to apply a function repeadedly onto the elements of a list
- `(mapcan f l1 l2 ...)`
- The rezults are grouped with `nconc`

```
(MAPCAN #'CAR '((A B C) (X Y Z))) --> NIL,
; because nconc requires lists, so (NCONC 'A 'X) is NIL
(MAPCAN #'LIST '(A B C) '(1 2)) --> (A 1 B 2)
(MAPCAN #'LIST '(A B C)) --> ( A B C )
(MAPCAN #'EQUAL '(A (B C) D) '(Q (B C) D X)) --> NIL
(MAPCAN #'+ '(1 2 3) '(4 5 6)) --> NIL
```

MAP Functions.

- The role of a map function is to apply a function repeadedly onto the elements of a list
- `(maplist f l1 l2 ...)`
- The rezults are grouped with `list`

```
(MAPLIST #'APPEND '(A B C) '(1 2 3))
    --> ((A B C 1 2 3) (B C 2 3) (C 3))
(MAPLIST #'(LAMBDA (X) X) '(A B C)) --> ((A B C) (B C) (C))
(SETF TEMP '(1 2 7 4 6 5))
(MAPLIST #'(LAMBDA (XL YL) (< (CAR XL)(CAR YL)))
    TEMP (CDR TEMP)
) --> (T T NIL T NIL)
```

MAP Functions.

- The role of a map function is to apply a function repeadedly onto the elements of a list
- `(maplist f l1 l2 ...)`
- The rezults are grouped with `list`

```
(MAPLIST #'CAR '((A B C) (X Y Z))) --> ((A B C) (X Y Z))
(MAPLIST #'LIST '(A B C) '(1 2)) --> ( ((A B C) (1 2)) ((B C) (2)))
(MAPLIST #'LIST '(A B C)) --> ( ((A B C)) ((B C)) ((C)) )
(MAPLIST #'EQUAL '(A (B C) D) '(Q (B C) D X)) --> (NIL NIL NIL)
(MAPLIST #'+ '(1 2 3) '(4 5 6))
    --> \argument to + should be a number: (1 2 3)".
```

MAP Functions.

- The role of a map function is to apply a function repeadedly onto the elements of a list
- `(mapcon f l1 l2 ...)`
- The rezults are grouped with `list`

```
(MAPCON #'CAR '((A B C) (X Y Z))) --> (A B C X Y Z)
(MAPCON #'LIST '(A B C) '(1 2)) --> ((A B C) (1 2) (B C) (2))
(MAPCON #'LIST '(A B C)) --> ((A B C) (B C) (C))
(MAPCON #'EQUAL '(A (B C) D) '(Q (B C) D X)) --> NIL
```

MAP Functions.

- The role of a map function is to apply a function repeadedly onto the elements of a list
- `(mapcon f l1 l2 ...)`
- The rezults are grouped with `list`

```
(MAPCON #'+ '(1 2 3) '(4 5 6))
    --> \argument to + should be a number: (1 2 3)"
(DEFUN G(L)
    (MAPCON #'LIST L)
)
(G '(1 2 3)) --> ((1 2 3) (2 3) (3))
(MAPCON #'(LAMBDA (L) (MAPCON #'LIST L)) '(1 2 3))
    --> ((1 2 3) (2 3) (3) (2 3) (3) (3))
```