Functional Programming – Laboratory 5

Tail recursion, Control

Isabela Drămnesc

March 28, 2012

1 Concepts

- Tail recursion
- when, unless, let, let*
- block, tagbody, loop
- progn, prog1, prog2
- prog, prog*
- do, do*, dolist, dotimes

2 Questions from Laboratory 4

- How many types of recursion do you know? Which one is the most efficient?
- How do we recognize a tail recursive function? What is the method used for writing tail recursive functions? Give one example of a tail recursive function (write the definition of the function in LISP).
- Simulate (trace functionname), where functionname is the name of your own function.

3 Control

3.1 when, unless

```
> (when (= 8 9) 4 9)
```

```
> (when (< 8 9) (print 'azi) 4 9)
```

```
> (when (> 10 9))
```

```
> (unless (> 10 9) 100 'nothing)
> (unless (> 10 9) 100 (print 'nothing))

> (unless (< 10 9) 100 (print 'nothing))

> (unless (< 10 9) (print 'nothing) 100)
```

when and *unless* are equivalent with:

```
(unless p a b c) == (cond ((not p) a b c))

(unless p a b c) == (when (not p) a b c)

(when p a b c) == (cond (p a b c))

(when p a b c) == (unless (not p) a b c)
```

3.2 block, tagbody, loop

```
;construction of a block -- the structure

(block block_name <form1> <form2> .... <formn>)

; <form1> <form2> .... <formn> are optional
; in the case of missing it will return NIL

> (block block_name (print 'expr1) (print 'expr2) (print 'expr3))

> (block block_name 1 2 3 4)

> (block block_name 2)

;Blocks with abrupt exit
; for this we use
; return-form or return.

;When return-form block_name is evaluated the program exits
; the block block_name with the value resulted from the
; evaluation of the second argument (which is optional
; <form1> <form2> .... <formn>),
; if this is missing, then we get NIL.

> (block block_name3
         (setq x 1)
         (print (1+ x))
         (return-from block_name3 (1+ x))
         (print 4))

> (block block_name33
       (setq x 1)
       (print (1+ x))
       (setq x (1+ x))
       (return-from block_name33 (1+ x))
       (print 4))

; return is used in order to exit blocks with the name NIL
; do, dolist, dotimes and loop include implicitly a
; block with the name nil):

> (dolist (x '(1 2 3)))

> (dolist (x '(1 2 3)) (print x))

> (dolist (x '(2 3 -7 5))
          (print x)
          (if (< x 0) (return 'done)))

; Any function definition is a block with the name the name of the function
; from the body of the function we exit using
; return-from

> (defun f()
       (print 'a)
       (return-from f 10)
       (print 'b))

> (f)

; Blocks within which you can use gotos

> (tagbody again
          (setq x (1+ x))
          (if (< x 5) (go again)
                      (go end)
          )
          end
       (print x)
  )

> (tagbody lala (setq v 9) (print v))

;read a number until that number is greater than 0

> (tagbody retake
          (print 'Introduce>)
          (if (plusp (read))
                     (go retake)
                     'done
          )
)
INTRODUCE>
23
```

```
INTRODUCE>
45
INTRODUCE>
12
INTRODUCE>
-90
NIL

> (block some
        (setq d (1+ 3))
        (print d)
        (if (< d 4) (go some)
                    (return-from some 100))
        )
        (print 'somethingStrage)
)

> (tagbody some
        (setq d (1+ 3))
        (print d)
        (if (< d 4) (go some)
                    (return 100)
        )
        (print 'somethingStrage)
)

> (tagbody some
        (setq d (1+ 3))
        (print d)
        (if (< d 4) (go some)
                    (go lala)
        )
        lala
        (print 'somethingStrage)
)
```

Conclusions:

- (block block-name expression-1 expression-2 ... expression-n)
- (return [result])
- (return-from name [result])
- (return value) == (return-from nil value)
- tagbody accepts symbols - they are not evaluated.
- If we reach the end of a tagbody nil is returned.
- loop repeatedly evaluates his parameters.
- We exit a loop using return or throw.

- Advise: try to not use go!

3.3 progn, prog1, prog2

```
> (setq w 11 xx 22)

> (values w xx)

> (values 1 2 3 4)

> (values '(1 2 3 4))

> (progn 10 (print 20) 30)

> (progn 10 (print 20) 30 (values 10 20 30))

> (progn 100 200 300 (values 10 20 30))

> (progn 1 2 3 4 5 6 7)

> (prog1 1 2 3 4 5 6 7)

> (prog2 1 2 3 4 5 6 7)

> (prog2 'la (values 2 3 4) 9)

> (prog2 'la (values '(2 3 4) 90) 9)

> (progn (+ 2 3) (+ 3 5) (+ 11 22))

> (prog1 (+ 2 3) (+ 3 5) (+ 11 22))

> (prog2 (+ 2 3) (+ 3 5) (+ 11 22))
```

Conclusions:

- *progn* returns the last evaluation form (if the last evaluation returns multiple values, then progn will return them all);
- *prog*1 returns the first form evaluation (only its first value).
- $(prog2abc...z) == (progna(prog1bc...z))$

3.4 let, let*

let is used to group expressions and performs parallel binding of local variables to values.

let* performs sequential binding of local variables to values.

```
    (let[*] (
                (var-1 value-1)
                (var-2 value-2)
                ...
                (var-m value-m)
            )
            expression-1
            expression-2
            ...
            expression-n)

> (let ((x 1) (y 2) (z 3)) (setq w (+ x y z)) (list x y z w))

> (let* ((x 1) (y (+ x 1)) (z (+ y 1))) (list x y z))
```

3.5 prog, prog*

```
     (prog (var-1 var-2 (var-3 init-3) var-4 (var-5 init-5))
           expression-1
        value-1
           expression-2
           expression-3
           expression-4
        value-2
           expression-5
           ...
      )

;      prog is a combination between block, tagbody and let*
;      prog opens implicitly a nil block, therefore
;                   in order to exit we use return [result].
; Example:

> (prog (i (sum 0))
        retake
        (print 'Introduce>)
        (setq i (read))
        (if (> i 0)
               (progn (setq sum (+ sum i)) (go retake))
               (return sum)
        )
)
INTRODUCE>
34
INTRODUCE>
23
INTRODUCE>
11
INTRODUCE>
 4.5
INTRODUCE>
```

```
-3
72.5

; use catch and throw

; Example: if x is greater than 0 returns ok,
; otherwise the result is the value of x

> (defun f1 (x)
        (catch 'ex1 (f2 x)))

> (defun f2 (x)
        (if (minusp x)
                (throw 'ex1 x)
        )
      'ok
)

> (f1 2)

> (f1 -2)

;; the mechanism of catch and throw is:
; - when throw is evaluated, it does not evaluate the
;      forms after throw, but it takes up the forms
;      (located in the interpreter stack)
;       until it reaches catch with the same label as
;       throw (the second argument). In this moment the
; result of throw is the value returned by catch.
```

Concluzii:

- In blocks like *block* we use *return − from* or *return* in order to force the exit;
- In blocks like *tagbody* we have *go*;
- The definition of a function is a block with the name of the function;
- For non-local gotos use *catch* and *throw*
- sequential blocks are: *progn*, *prog*1, *prog*2
- *prog* is a combination between *block*, *tagbody*, *let*∗

3.6 loop,do,do*

Iteration statements in Lisp can be done in multiple ways. The most commonly used id *do*. The general form is:

```
(do ({(<var> [<init> [<step>]])}*)
    (<test-end> {<result>}*)
    <body>
)

;or

(do (  (var1 init1 step1)
       (var2 init2 step2)
        ...
       (varn initn stepn) )
     (test-end result)
   body)
```

The difference between *do* and *do∗* is that *do* is used for binding in parallel and *do∗* is used for binding sequential (similar with *psetq* and *let*, or *setq* or *let∗*

```
;prints the natural numbers between two values:

> (defun printing (start ending)
       (do ((i start (1+ i)))
           ((> i ending))
         (print i)
       )
)

> (printing 2 10)


; if we want to return some values
; then we write the function:

> (defun printing2 (start ending)
       (do ((i start (1+ i)))
           ((> i ending) 'done)
         (print i)
       )
)

> (printing2 2 8)

> (defun factorial (n)
       (do* ((i 1 (1+ i))
        (result 1 (* result i)))
       ((= i n) result)
       )
)

> (factorial 8)

; using loop

> (loop (print 10) (print 20) (print 30) (return))
> (loop (print '>)
        (if (eq (read) 'stop)
                (return 'exit)
        )
  )
>
256
>
7896540
>
9
>
-980
>
lalala
>
stop
EXIT
```

```
>(loop for i from 1 to 6 do (print i))
```

```
> (do ((x 1 (1+ x)) (y 1 (* x y))) ((> x 5) y))
```

DOTIMES is a version of the function DO and is equivalent with FOR. The syntax is:

(DOTIMES (variabe counter result) body)

Equivalent with:

```
FOR variabe=0 until counter-1 DO
        body
RETURN result
```

Examples:

```
> (dotimes (i 10 (1+ i)) (print 'today))
```

```
> (dotimes (i 10 (1+ i)) (prin1 i) (princ "␣"))
```

```
> (dotimes (i 4 (* i 2)))
```

```
> (dotimes (i 4 (* i 2)) (print 'one))
```

```
> (dotimes (i 4 (* i 2)) (1+ 89))
```

```
> (dotimes (i 4 (* i 2)) (print (1+ 89)))
```

```
> (dotimes (i 5 (* i i)) (prin1 i))
```

```
> (dotimes (i 5 (* i i)) (prin1 i) (princ "␣"))
> (dotimes (i 10 (* i i)) (prin1 i) (princ ”␣”))
```

DOLIST is similar to DOTIMES but when using DOTIMES the variable receives values between 0 to counter-1, and when using DOLIST the variable receives all the values of the elements from the list.

The syntax is:

(DOLIST (variable my-list result) body)

Equivalent to:

```
FOR variable=first elem from the list
UNTIL the last element from the list DO
        body
RETUR result
```

Examples:

```
> (dolist (var '(1 2 3)) (print var))

> (dolist (x '(a b c)) (prin1 x) (princ ”␣”))
```

4 Homework - deadline: next lab

1. Write a recursive function which returns a list with all the atoms from a list given as parameter:

```
(squash '(a b c (d e) ((f) g)))
=> (a b c d e f g)

(squash '(a b))
=> (a b)

(squash '(() ((((a)))) ()))
=> (a)
```

2. Write a non-recursive function which calculates the sum of squares of the elements from a list.

```
(square-sum '(1 2 3))
=> 14
```