Functional Programming – Laboratory 4

Recursion, Tail recursion

Isabela Drămnesc

March 22, 2012

1 Concepts

- Recursion
- Tail recursion
- The technique that uses a collector variable
- Define new tail recursive functions
- Operations on lists - superficial level
- Operations on lists - any level
- Operations on sets
- The use of (time <expression>)
- The use of (trace <expression>)

2 Questions from Lab 3

- What do you know about variables in Lisp? Give some examples to denote the differences between them.
- What is recursion? Give at least two examples of recursive functions (write the definition).
- How important is the order of the declaration of the clauses when defining a recursive function in Lisp?
- Write functions in Lisp for:
  1. The concatenation of two lists;
  2. The reverse of a list;
  3. The length of a list.

3 Tail recursion. The collector variable.

A recursive function is one that calls itself. Such a call is tail-recursive if no work remains to be done in the calling function afterwards.

A function is *linear recursive* if calls itself one time in order to return the result.

A function is *fat recursive* if calls itself several times in order to return the result.

A function is *tail-recursive* if:

- the value returned is either something computed directly or the value returned by a recursive call;

A recursive call is not a tail call if, after it returns, its value will be passed as argument to another function.

Fat recursion is very inefficient, tail recursion is the most efficient.

In order to transform a recursive function into a tail recursive function we use the technique of the collector variable.

3.1 Factorial

```
;test the functions defined in lab3

;>(time (fact 2485))

;>(time (factorial-cond 2484))

;>(time (factorialn 2000))

;; the technique of the collector variable ;;

(defun factf (n)
       (fact-aux n 1))

(defun fact-aux (n rez)
       (cond ((= n 0) rez)
             (t (fact-aux (- n 1) (* n rez)))
       )
)

;>(time (factf 2700))

;>(trace fact-aux)

;>(fact-aux 5 1)
```

3.2 Fibonacci

```
(defun fib (n)
  (if (< n 2) n
    (+ (fib (- n 2)) (fib (- n 1)))
  )
)

;>(time (fib 3000))

;;;; The tail recursive version

(defun rfib (n)
  (if (< n 2) n
    (fib-aux n 1 0)
  )
)

(defun fib-aux (n fn fn-1)
  (if (= 1 n) fn
    (fib-aux (- n 1) (+ fn fn-1) fn)
  )
)

; >(time (rfib 3000))

;> (trace fib-aux)
```

3.3 Reverse

```
;;; define the function reverse using a collector variable ;;
(defun rev1 (l1 l2)
       "our reverse has 2 arguments"
       (cond ((null l1) l2)
             (t (rev1 (cdr l1) (cons (car l1) l2) ))
       )
)

(defun rev-list (l)
       (rev1 l ())
)

#| > (rev-list '(1 2 3 4 (8 9)))
((8 9) 4 3 2 1)

> (rev1 '(1 2 3) '())
(3 2 1)

> (trace rev1)

>(rev1 '(1 2 3) '())
|#
```

3.4 The length of a list

Similar with the examples above write a tail recursive function which returns the length of a list. Example:

```
> (our-len '(1 2 3 4))
4

> (our-len '(1 2 3 (j k) (m n o p) 4))
6
```

4 The superficial level, any level

4.1 Define a function in Lisp which returns the first atom from a list. Example:

At the superficial level:

```
>(first-atom '(1 2 3))
1

>(first-atom '())
NIL

>(first-atom '((a b) (c d e) l (o p)))
L
```

At any level:

```
>(first-atom-2 '((((2 3 4) 8) 8 9) 9))
2

>(first-atom-2 '((((a b 4) 8) 8 9) 9))
A
```

5 Homework

5.1 To the power of - the tail recursive version

Write a tail recursive function which calculates x^y.

5.2 The sum of the numbers from a list - the tail recursive version

Write a tail recursive function which returns the sum of the elements from a list. Skip the symbols and adds only the numbers from the list. Example:

```
>(add-numbers '(1 2 d 4))
7
>(add-numbers '(1 2 d 4 (5 lalala 5)))
17
```

5.3 Operations with sets

Given two sets A and B, write a recursive function which returns the union of the two sets ($A \cup B$). Similar, write a recursive function for the intersection ($A \cap B$), for the difference ($A\ B$) and for the symmetric difference ($A \triangle B$).

Deadline: next lab.