Functional Programming – Laboratory 4

Recursion, Tail recursion

Isabela Drămnesc

March 17, 2014

1 Concepts

- Recursion
- Tail recursion
- Accummulators
- Define new tail recursive functions
- Operations on lists at superficial level
- Operations on lists at any level
- Operations on sets
- Using (time <expression>)
- Using (trace <expression>)

2 Questions from Lab 3

- How do we define local and global variables in Racket? Give some examples.
- What is recursion? Give at least two exaples of recursive functions (write their definitions in Racket).
- Is it important the order of the declaration of the caluses when writing a recursive definition in Racket?
- Write the corresponding definition in Racket for each of the following (without using predefined functions as: reverse, length, append):
  1. Appending two lists;
  2. The reverse of a list;
  3. The length of a list.

3 Tail recursion. The collector variable technique.

A recursive function is a function which 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 the function 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 return, 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 (accummulator).

3.1 Factorial

```
;testing the function defined in lab3
; factorial from lab 3 (without accummulator)
(require racket/trace)

(define (fact n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))
      )
  )
(trace fact)

> (fact 5)

> (untrace fact)

>(time (fact 1000))

; the version which uses accummulator (the collector variable)

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

(define (fact-aux n rez)
  (cond ((= n 0) rez)
```

```
        (#t (fact-aux (- n 1) (* n rez)))
   )
  )
(trace fact-aux)

 >(fact-aux 5 1)

> (untrace fact-aux)

> (factf 10)

> (factf 100)

> (factf 10000)

> (time (factf 1000))
```

3.2 Fibonacci

```
; another example from lab 3
; fibonacci without accummulators

(define (fibonacci n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (#t (+ (fibonacci (- n 1))
               (fibonacci (- n 2))
            )
        )
  )
)
; >(time (fibonacci 300))

(trace fibonacci)

> (fibonacci 11)

> (untrace fibonacci)

;;;; the tail recursive version

(define (rfib n)
  (if (< n 1)
      1
      (fib-aux n 1 1)
   )
 )

(define (fib-aux n fn-1 fn-2)
  (if (= n 1)
```

```
        fn-1
        (fib-aux (- n 1) (+ fn-1 fn-2) fn-1)
    )
  )
(trace fib-aux)

> (fib-aux 5 1 1)
 > (rfib 2)

> (rfib 3)

> (rfib 4)

> (rfib 5)

> (rfib 11)

> (untrace rfib)

> (rfib 12)

> (untrace fib-aux)

> (rfib 11)

> (time (rfib 100))

> (time (rfib 1000))
```

3.3 Define a tail recursive function which calculates x * y as follows:

```
> (xtimesy 2 -1)
-2
> (xtimesy 2 -2)
-4
> (xtimesy 2 -3)
-6
> (xtimesy 2 -7)
-14
> (xtimesy 2 -10000)
-20000
> (xtimesy 0 -10000)
0
> (xtimesy -6 -10000)
60000
```

3.4 The reverse of a list

```
;;; defining reverse with accumulator
```

```
(define (rev l)
  (rev-aux l '())
  )

(define (rev-aux l aux)
  (cond ((null? l) aux)
        (#t (rev-aux (cdr l) (cons (car l) aux)))
   )
 )

> (rev '(a b d))

> (rev '(a b d (1 2)))

(trace rev-aux)

> (rev-aux '(1 2 a b d) '())

> (untrace rev-aux)

>(rev '(1 (a b) 2 3))
```

3.5 The length of a list

Following the above examples write a tail recursive function which returns the length of a list. Examples:

```
> (rlen '(1 2 3 4))
4

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

3.6 Write a tail recursive function which calculates the sum of the numbers from a list (the function ignores the symbols).

```
>(rsum '(1 2 d 4))
7
>(rsum '(1 2 d 4 (5 lalala 5)))
17
```

4 Superficial level, any nivel

4.1 Define a function in Racket which returns the first atom from a list. The behaviour of the function is as follows:

At the superficial level:

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

>(first-elem '())
#f

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

At any level (without using FLATTEN):

```
>(first-elem2 '((((2 3 4) 8) 8 9) 9))
2

>(first-elem2 '((((a b 4) 8) 8 9) 9))
'a
```

5 Homework

5.1 To the power of - tail recursive.

Write a tail recursive function which calculates x^y. Analize all the possible cases (including the case when y is negative.

5.2 Operations on 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 ($A \triangle B$). Hint: verify if the input list is a set.

Deadline: Next laboratory.