Bartu Atabek – 21602229

Section 2

CS 315 “Programming Languages” Homework 3 Report

The description of the Reverse List (foo) function

I defined my recursive function with the help of a helper function so that overall I defined two distinct functions foo and foo-aux. foo function is a simple (non-recursive) function which takes a list as a single argument and calls the auxiliary helper function foo-aux. foo-aux is a tail recursive function which takes the list as its argument and also takes an auxiliary parameter which is the reversed list.

The code in Scheme

```
(define (foo list)
   (foo-aux list '())
)

(define (foo-aux list reversed_list)
   (if (null? list)
       reversed_list
   (foo-aux (cdr list) (cons (car list) reversed_list))))
```

Figure 1: Scheme code for the reverse list function defined by a simple function ‘foo’ and a tail recursive function ‘foo-aux’.

How it works?

First of all the first function foo takes a list which could be a combination of atoms or lists and the elements that are lists can further be a list of other lists, in any depth. Since this function only reverses the order of the elements the atoms of the list are not required to be a specific type, such as numbers, or strings. The foo function calls the helper function with two arguments; the initial list and an empty list for the reverse operation. The foo-aux helper function which is a tail recursive function first checks whether the list is null or not. If it is null it returns the reversed list, otherwise it recursively calls itself again by taking the list without the first element as the first argument then, appends the first element of the list to the reversed list and again takes the appended reversed list as its auxiliary parameter.

The recursion steps for the input (2 3 4) is shown below.

```
(foo ‘(1 2 3))                list = (1 2 3) reversed_list = ()
(foo–aux ‘(1 2 3) ‘())        list = (1 2 3) reversed_list = ()
(foo–aux ‘(2 3) ‘(1))         list = (2 3)   reversed_list = (1)
(foo–aux ‘(3) ‘(2 1))         list = (3)     reversed_list = (2 1)
(foo–aux ‘() ‘(3 2 1))        list = ()      reversed_list = (3 2 1)
```

After all the elements in the input are reversed the function returns the reversed list which is in this case (4 3 2).

Why it is tail recursive?

The definition of a tail recursive function states that; “a function is tail recursive if its recursive call is the last operation in the function”. The auxiliary helper function foo-aux is a tail recursive function since it applies to this statement because it ends with calling itself with the smaller list excluding the first element and an appended reversed list where the first element of the list is added. Since a tail recursive function can be automatically converted by a compiler to use iteration which makes it faster, and the above example supports this argument since we can see the iteration of the auxiliary helper function over the list elements which could also be done by using iteration. Since these facts are coherent with the auxiliary function foo-aux, it is a tail recursive function.

How it is invoked?

In order to invoke the functions and reverse a list, the only thing needs to be done is to call the foo function with a list argument as shown below. The argument list could be in any length and depth and there’s no type specification.

<u>Examples:</u>

```
(foo ‘())
(foo ‘(1 2 3))
(foo '(bartu ((10 (emir)) 9 ) gorkem 8 (((deniz)))))
```

Test results

Input:

```
(foo ‘())
```

Output:

```
()
```

Input:

```
(foo ‘(1 3 5 7 9))
```

Output:

```
(9 7 5 3 1)
```

Input:

```
(foo '(Bartu ((10 (Emir)) 9 ) Utku 8 (((Deniz)))))
```

Output:

```
((((Deniz))) 8 Utku ((10 (Emir)) 9) Bartu)
```

Input:

```
(foo '(5 (12 37 (4 ((((1 100)))) 55)) 86))
```

Output:

```
(86 (12 37 (4 ((((1 100)))) 55)) 5)
```

Input:

```
(foo '((b (c)) (a) (b ((a))))))
```

Output:

```
((b ((a))) (a) (b (c)))
```