- **the description of your function**

  My function checks whether the list is a palindrome or not, meaning that it checks if the list is read the same backwards as forwards. For example (1 2 3 2 1) is a palindrome whereas (1 2 3 4 5) isn't.

  I also treated empty and single-element lists as palindromes.

  My function works regardless of the type of the elements of the list. The parameter list can contain booleans, numbers, strings and other lists all at the same time and the function will work correctly.

  If two variables hold the same value and they are the first and second elements of a list, my function will say that list is not a palindrome. Let's say x and y are both defined to be 5. If we put the list (x y) as parameter into foo the output will be false. I think this is not a problem because palindromes are based on how we read things and not on how we evaluate things. We don't read (x y) as "5 5" therefore it is not a palindrome.

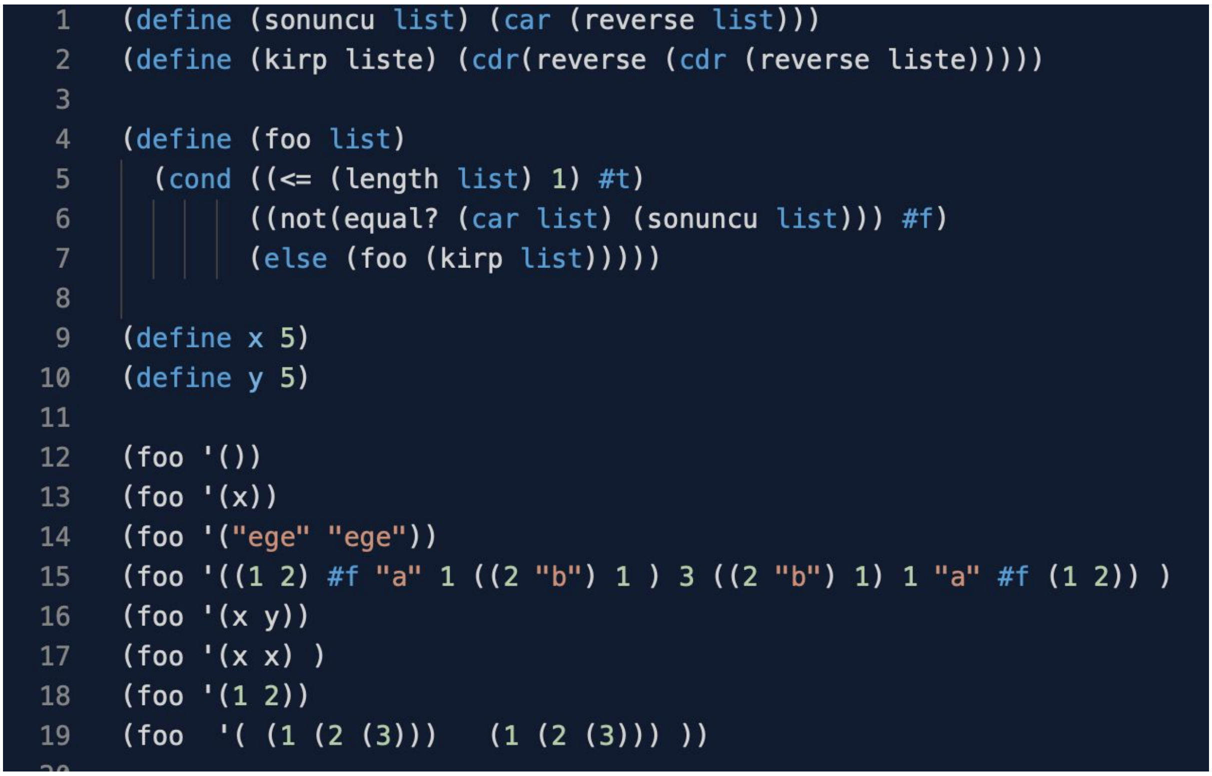
  I should also note that when the given list contains many lists, my function does not compare the contents of the lists to make sure that they are symmetric, it just compares if the lists are equal or not. For example the list ((1 2) (2 1)) is not considered a palindrome in my function whereas ((1 2) (1 2)) is. This is a result of my implementation, I treat every list like a single element and check if two lists are equal or not. I don't check if one list is the reverse of the other. This was my choice and I could have easily added a new line to my function so that when comparing two lists it does not compare if they are equal but compares if they are the reverse of each other.

  ```
  (

  ((and (list? (car list) ) (list? (sonuncu list))  (not
  (equal? (car list) (reverse (sonuncu list))))) #f)

  )
  ```

- **the code in Scheme**



```scheme
(define (sonuncu list) (car (reverse list)))
(define (kirp liste) (cdr(reverse (cdr (reverse liste)))))

(define (foo list)
  (cond ((<= (length list) 1) #t)
        ((not(equal? (car list) (sonuncu list))) #f)
        (else (foo (kirp list)))))

(define x 5)
(define y 5)

(foo '())
(foo '(x))
(foo '("ege" "ege"))
(foo '((1 2) #f "a" 1 ((2 "b") 1 ) 3 ((2 "b") 1) 1 "a" #f (1 2)) )
(foo '(x y))
(foo '(x x) )
(foo '(1 2))
(foo  '( (1 (2 (3)))   (1 (2 (3))) ))
```

text=

```scheme
(define (sonuncu list) (car (reverse list)))

(define (kirp liste) (cdr(reverse (cdr (reverse liste)))))



(define (foo list)

 (cond ((<= (length list) 1) #t)

        ((not(equal? (car list) (sonuncu list))) #f)

       (else (foo (kirp list)))))



(define x 5)

(define y 5)
```

```
(foo '())

(foo '(x))

(foo '("ege" "ege"))

(foo '((1 2) #f "a" 1 ((2 "b") 1 ) 3 ((2 "b") 1) 1 "a" #f (1
2)) )

(foo '(x y))

(foo '(x x) )

(foo '(1 2))

(foo  '( (1 (2 (3)))   (1 (2 (3))) ))
```

- **explanation of how it works**

  I used 2 simple functions because they increase readability and writability. My code would still work if I didn't use any other functions, for example I could write `(car (reverse list))` instead of `(sonuncu list).`

  "sonuncu" yields the last element of a list, it is the symmetric of "car".

  "kirp" is similar to "cdr", it yields the list without the first and last element.

  The function foo first checks the base case. If the list given to foo as a parameter has a length of 0 or 1 the function returns true.

  In any other case it compares the first and last elements of the list, if they are not equal the function returns false.

  If they are equal foo is called again on a list that doesn't have the first and last elements of the given list(parameter).This step is repeated until the base case is reached or two elements with different values are evaluated. If the base case is reached the function returns true. If the case where a list has more than 1 element and the first and last element of this list aren't equal is reached the function returns false.

- **explanation of why it is tail recursive**

  The last thing executed by foo is foo itself. We don't do anything with the result that the recursive call returns, the last thing we do in the foo function is call another foo function (If we did something like `(else (* 5 (foo (liste))))` the last thing we do inside the function would not be to call foo but to multiply)

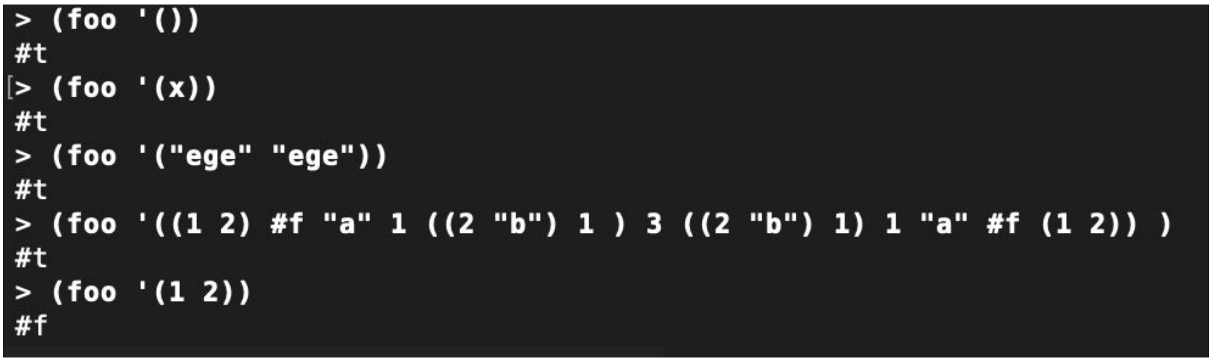
Once we call a new foo function inside a foo function we don’t need the old foo anymore. We go into each recursion with the knowledge that the prior cases were a success and we do not need to hold them in the memory, this means that the activation records of foo calls can be popped from the stack once they call another foo.

As the last statement we evaluate inside the foo function is the recursive call, foo is tail recursive.

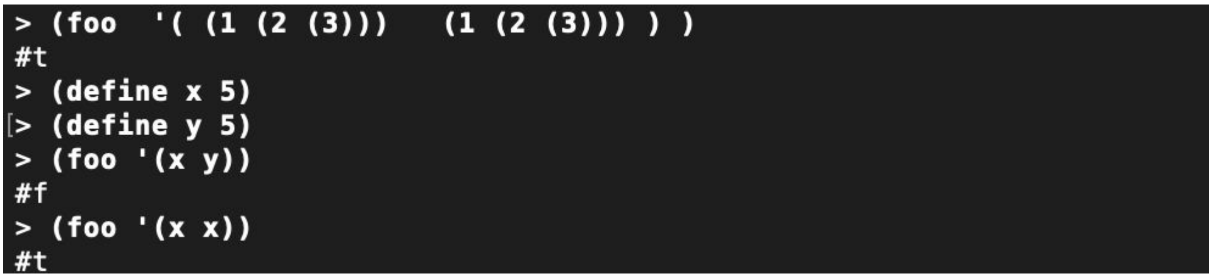
- **how it is invoked**

My foo does not use an auxiliary function so it can be directly called by giving any list as a parameter. For example `(foo ‘(1 2))`

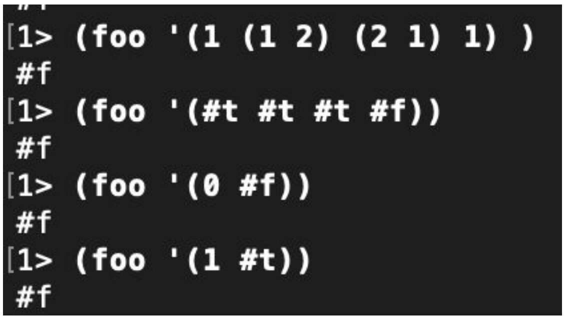
- **test results**



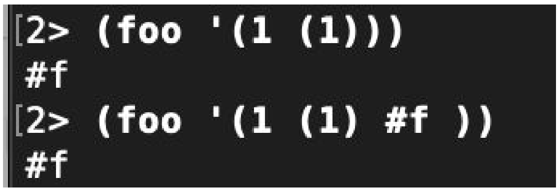
```
> (foo '())
#t
[> (foo '(x))
#t
> (foo '("ege" "ege"))
#t
> (foo '((1 2) #f "a" 1 ((2 "b") 1 ) 3 ((2 "b") 1) 1 "a" #f (1 2)) )
#t
> (foo '(1 2))
#f
```



```
> (foo  '( (1 (2 (3)))   (1 (2 (3))) ) )
#t
> (define x 5)
[> (define y 5)
> (foo '(x y))
#f
> (foo '(x x))
#t
```



```
[1> (foo '(1 (1 2) (2 1) 1) )
#f
[1> (foo '(#t #t #t #f))
#f
[1> (foo '(0 #f))
#f
[1> (foo '(1 #t))
#f
```



```
[2> (foo '(1 (1)))
#f
[2> (foo '(1 (1) #f ))
#f
```