Project 4
COMP301 Fall 2020
Due: January 8, 2021 - 23:59 (GMT+3 : Istanbul Time)

In this project, you will work in groups of two or three. To create your group, use the Google Sheet file in the following link: *Link to Google Sheets for Choosing Group Members*.

Note: You need to **self-enroll** to your Project 4 group on BlackBoard (please only enroll to the same group number as your group in the Sheets), please make sure that you are enrolled to Project 4 - Group #YourGroup.

This project contains 2 main parts about 2 different topics, namely: **Parameter Passing** and **Continuation Passing Style**.

Submit a report containing your answers to the written questions in PDF format and Racket files for the coding questions to Blackboard as a zip. Include a brief explanation of your team's workload breakdown in the pdf file. Name your submission files as:

p4_member1IDno_member1username_member2IDno_member2username.zip
Example: *p4_0011111_baristopal20_0022222_etezcan19.zip.*

Important Notice: If your submitted code is not working properly, i.e. throws error or fails in all test cases, your submission will be graded as 0 directly. Please comment out parts that cause to throw error and indicate both which parts work and which parts do not work in your report explicitly.

Please use *Project 4 Discussion Forum* on Blackboard for all your questions. The deadline for this project is Due: January 8, 2021 - 23:59 (GMT+3 : Istanbul Time). **Read your task requirements carefully. Good luck!**

TABLE 1. Grade Breakdown for Project 4

Question	Grade Possible
1. Parameter Passing	
Task 1	10 points
Task 2	15 points
Task 3	25 points
2. Continuation Passing Style	
Task 4	8 points
Task 5	42 points

1. Parameter Passing

Task 1: Why do these pairs below may give different results sometimes for the same expression:

- Call-by-value and call-by-reference
- Call-by-need and call-by-name

What are the advantages and disadvantages of each?

Task 2: To use call-by-need parameter passing variation, some specific changes and additions have to be made to the IREF implementation. 2 of these are given below:

```
; Change
(var-exp (var)
   (let ((ref1 (apply-env env var)))
      (let ((w (deref ref1)))
         (if (expval? w)
            w
            (let ((val1 (value-of-thunk w)))
               (begin
                  (setref! ref1 val1)
                  val1)))))))

; Addition
(define value-of-thunk
   (lambda (th)
      (cases thunk th
         (a-thunk (exp1 saved-env)
            (value-of exp1 saved-env)))))
```

Explain why these code pieces are needed. Analyze how these codes work line by line in detail and state in which file(s) of the IREF implementation code they should be added.

Task 3: Write an expression that gives different results in:

(1) Call-by-reference and call-by-need
(2) Call-by-reference and call-by-name
(3) Call-by-value and call-by-need
(4) Call-by-value and call-by-name

In total, 4 expressions should be written (one for each case). As reference, in the *Parameter Passing* directory of the Project Assignment zip, codes for all of these 4 parameter passing variations are already provided. **Please do not change any files except *tests.scm*!** In all of their *tests.scm* files, a place is reserved for you to add your expression. Please keep in mind that you should add the same expression in both of the parameter passing variations. In other words, if you wrote an expression that gives different outcomes for instance in call-by-value and call-by-need, **please add this expression in both of their *tests.scm* files**.

Notes:

- If your code gives any error, then you will directly receive 0 points from this task.
- For simplicity, assign-exp and begin-exp are also added to the call-by-value codes.
- In call-by-value codes, some expressions and structures such as mutable pairs are not defined. Keep these differences in mind while trying to write your expression.

2. Continuation Passing Style

Task 4: Using Scheme[1], implement a function `fibonacci`, that takes a parameter n and returns the n^{th} Fibonacci number, with **Continuation Passing Style**. The Fibonacci sequence goes like:

$$F = [1, 1, 2, 3, 5, 8, 13, \ldots]$$

where $F[1] = 1, F[2] = 1$ and $F[n] = F[n-1] + F[n-2]$.[2]

Task 5: You are given a LETREC implementation that has CPS with data-structural representations for continuations. Extend this language to include `list` and `map`.[3]

Important: Your implementations **must** use CPS. Furthermore, in your CPS implementations your `value-of` calls should be **tail calls** only. In particular, you must see "End of Computation" message appear **only once** when you run your program. See page 144 of EOPL book for more detail. Here is an example of `diff` expression continuation with a good CPS and bad CPS usage:

```
; good usage, value-of is in a tail call
(diff1-cont (exp2 saved-env saved-cont)
   (value-of/k exp2
      saved-env (diff2-cont val saved-cont)))
(diff2-cont (val1 saved-cont)
   (let ((num1 (expval->num val1))
      (num2 (expval->num val)))
         (apply-cont saved-cont
            (num-val (- num1 num2)))))

; bad usage, value-of is not in a tail call
(diff1-cont (exp2 saved-env saved-cont)
   (apply-cont saved-cont
      (num-val (-
         (expval->num val)
         (expval->num (value-of/k exp2 saved-env (end-cont)))))))
```

We have provided test cases for you in `tests.scm`, and also a few hints can be found within the code as comments. In particular, we have marked where you should write your code in each file as as:

```
;;;;;;;;;;;;;;;;;;;;;;;;; TASK 5 ;;;;;;;;;;;;;;;;;;;;;;;;;;
; some comments
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Do not change anything in the `tests.scm` file! If you would like to run your own code, write it in the console under `top.scm`.

List Implementation. Your list implementation will be similar to how we construct arrays from mutable pairs, or how a Scheme list is constructed as pairs. In fact, this part of the task is very similar to exercise 5.6 of EOPL book, at page 153. You will add two new values to the language:

- `pair` value
- `emptylist` value

[1]You do not need to extend a language or anything, just write a plain Scheme code.

[2]In some mathematical contexts the sequence starts with 0 instead of 1, but this way is a bit easier to implement.

[3]Hint: You will need to make changes in `interp.scm`, `data-structures.scm` and `lang.scm`.

A list expression looks like:

```
list(exp1, exp2, ..., expN)
```

The list is composed of **pairs**. Here is an example:

```
> (run "list(1,2)")
End of computation.
(pair-val (num-val 1) (pair-val (num-val 2) (emptylist-val)))
```

The basic list operations you will implement are:

- `car(expression)` returns the left part of the pair value.
- `cdr(expression)` returns the right part of the pair value.
- `null?(expression)` returns true if the **expression** is an `emptylist` value.
- `emptylist` actually creates an empty list, with the value `emptylist`.

You will also need to implement 2 extractors and a predicate:

- `expval->car` extracts the `car` of the expressed pair value.
- `expval->cdr` extracts the `cdr` of the expressed pair value.
- `expval-null?` returns true if the **expressed value** is an `emptylist` value.

There are several examples in `tests.scm`, but here is one that covers most of these operations.

```
> (run "let x = 3 in let arr = list(x, -(x,1)) in
   let y = if null?(arr) then 0 else car(cdr(arr)) in y")
End of computation.
(num-val 2)
```

Note that in this example, just `cdr(arr)` does not yield 2, but rather we have to do `car(cdr(arr))`. This is because in fact the first `cdr` yields a:

```
(pair-val (num-val 2) (emptylist-val))
```

Map Implementation. The map expression looks like:

```
map(expression, expression)
```

Here, the first expression will be treated like a `proc` expression with one parameter, and the second expression will be treated like a `list` expression. As an example, here is subtracting 5 from each element of the list:

```
> (run "map(proc (v) -(v,5), list(5, 10, 2))")
End of computation.
(pair-val (num-val 0) (pair-val (num-val 5) (pair-val (num-val -3) (emptylist-val))))
```

When you run `top.scm` the tests will run automatically. If everything works fine, you will see “no bugs found” message at the bottom of the console. Even if no bugs are found, if for some test you see more than one “End of Computation.” message, then there is something wrong with how you implemented CPS.