Project 4
COMP301 Spring 2023

Deadline: May 21, 2023 - 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 link below. You are free to make changes in your groups. We will pair those who listed themselves under the "individuals" column.

Link to Google Sheets for Choosing Group Members

In this project, two code boiler-plates are provided to you: use `Project4Part1` for the first part, `Project4Part2` for the second part. Submit a report containing a brief explanation of your approach to problems and your team's workload breakdown in a PDF file and Racket files for the coding questions to Blackboard as a zip. Include `Project4Part1`, `Project4Part2` folders separately. Name your submission files as

p3_member1IDno_member1username_member2IDno_member2username.zip
Example: *p3_0022222_nnayal21_0011111_edemirbas17.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 throw errors and indicate both which parts work and which parts do not work in your report explicitly.

Testing: You are provided some test cases under `tests.scm`. Please, check them to understand how your implementation should work. You can run all tests by using `(run-all)` command on the `top.scm` file. We will test your program with additional cases but your submission should pass all provided test cases.

Please use the TA emailing list for all your questions.

The deadline for this project is May 21, 2023 - 23:59 (GMT+3 : Istanbul Time). **Read your task requirements carefully. Good luck!**

Part 1: In this part, you will implement a translator for LETREC language. This translator will modify the expressions of LETREC to include an extra parameter that keeps track of how often the procedure is called recursively.

In the translated version, when a procedure is called the name of the procedure, you should print how many times it has been called along with the name of the procedure. If the procedure is assigned with no name, it should be named as 'anonym. Below, you can see some examples with their translated forms from the updated LETREC language:

```
; Note: n keeps the number of calls to the function,
;;;;;;;; 0 is given in the call-nested-exp since the initial call
;;;;;;;; is the zero'th call


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Expression:
let func = proc (x) -(x, 1) in (func 30)

;; Translated to:
let func = proc-nested (x, n, func) -(x, 1) in call-nested(func 30, 0)

;; Prints:
;; func --> 1


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Expression:
(proc (x) -(x, 1) 30)

;; Translated to:
call-nested(proc-nested (x, n, anonym) -(x, 1) 30, 0)

;; Prints:
;; anonym --> 1


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Expression:
letrec double(x) = if zero?(x)
      then 0 else -((double -(x, 1)), -2) in (double 4)

;; Translated to:
letrec-nested double(x, n) = if zero?(x)
      then 0 else -(call-nested(double -(x, 1), -(n, -1)), -2)
   in call-nested(double 4, 0)

;; Prints:
;; double --> 1
;; .... double --> 2
;; ........ double --> 3
;; ............ double --> 4
;; ................ double --> 5
```

Although the syntax of the input is the same with the original LETREC language, you need to translate the `call-exp`, `proc-exp`, and `letrec-exp` to the structure given below:

```
Expression ::= proc-nested(Identifier, Identifier, Identifier)
Expression
```
> proc-nested-exp (var count name body)

```
Expression ::= call-nested(Expression Expression, Expression)
```
> call-nested-exp (rator rand count)

```
Expression ::= letrec-nested Identifier (Identifier, Identifier) =
Expression in Expression
```
> letrec-nested-exp (p-name b-var b-count p-body letrec-body)

FIGURE 1. Syntax for the translated verison of LETREC language

To receive full points from this part, your code must evaluate all the test cases given in the `tests.scm` file correctly, AND YOUR CODE NEEDS TO PRINT THE NUMBER OF RE-CURSIVE CALLS CORRECTLY ALONG WITH THE PROCEDURE NAMES. Additionally, you have to provide three additional test cases different from the ones we provided to you, which are different (**not just slightly**). Please consider that we will test your code with additional test cases that will not be publicly shared.

Under each test case text in `test.scm`, the correct recursive print outputs are provided for you to check your solution. The areas you should update are also highlighted in the source code with comments. To print your recursive calls, you can use the `recursive-displayer` procedure defined in the `interp.scm` file.

Note 1: If you need to call multiple procedures one after another, you can use the `begin` command.

Note 2: The parts you need to change are marked in our provided code. Although you are not limited to these spaces, the problem is solvable by only changing these parts. If you need to change another part, please ask TAs if it is allowed, and if it is, please mark that piece of code of yours with the comment ***Extra Change***.

Note 3: You need to update the following files:
- `data-structures.scm`
- `environments.scm`
- `interp.scm`
- `lang.scm`
- `translator.scm`

Part 2: In this part, you will modify the translator for LEXADDR language. This translator will modify the expressions of LEXADDR so that variables will be renamed as their name + their number of occurrences. If the variable exists in the environment, display a message. Explicitly state which variable is shadowed by the reinitialized variable.

Example: the first appearance of x will be translated into x1

Below, you can see some examples and their translated forms from the updated LEXADDR language:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Expression:
let x = 10 in let x = 10 in (proc (x) -(x,3) 4)

;; Translated to:
#(struct:a-program
  #(struct:let-exp
    x1
    #(struct:const-exp 10)
    #(struct:let-exp
      x2
      x has been reinitialized. x2 is created and shadows x1
      #(struct:const-exp 10)
      #(struct:call-exp
        #(struct:proc-exp
          x3
          x has been reinitialized. x3 is created and shadows x2.
          #(struct:diff-exp #(struct:var-exp x3) #(struct:const-exp 3)))
        #(struct:const-exp 4)))))
>

   ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Expression:
let f = proc (x) -(x,1) in (f 30)

;; Translated to:
#(struct:a-program
  #(struct:let-exp
    f1
    #(struct:proc-exp
      x1
      #(struct:diff-exp #(struct:var-exp x1) #(struct:const-exp 1)))
    #(struct:call-exp
      #(struct:var-exp f1)
      #(struct:const-exp 30))))


  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Expression:
let a = 2 in let b = 3 in f = proc (a) -(a,b) in (f 10)
```

```scheme
;; Translated to:
#(struct:a-program
  #(struct:let-exp
    a1
    #(struct:const-exp 2)
    #(struct:let-exp
      b1
      #(struct:const-exp 3)
      #(struct:let-exp
          f1
          #(struct:proc-exp
              a2
              a has been reinitialized. a2 is created and shadows a1.
              #(struct:diff-exp #(struct:var-exp a2) #(struct:const-exp b1)))
          #(struct:call-exp
            #(struct:var-exp f1)
            #(struct:const-exp 10))))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Under each test case text in `test.scm`, the correct translated versions print outputs are provided for you to check your solution. The areas you should update are also highlighted in the source code with comments. You can use the built-in `string->symbol, string-append, number->string` procedures to manipulate strings.

To receive full points from this part, your code needs to print exact same outputs given for each test case provided in the `tests.scm` file. Please consider that we will test your code with some additional test cases, which will not be shared publicly.

Note: You need to update the following file:
  - `translator.scm`

Note2: You are expected to translate the expression but you do not need to evaluate them.