Project 2
COMP301 Spring 2022

Deadline: April 17, 2022 - 23:59 (GMT+3 : Istanbul Time)

In this project, you will work in groups of two. To create your group, use the Google Sheet file in the following link (specify your group until April 3, if you do not have any groups, then please write your name to *individuals* so that we can assign you to a group):

Link to Google Sheets for Choosing Group Members

This project contains a bonus component specified at the end and there are two code boilerplates provided to you: use `Project2MYLET` for the project and `Project2BONUS` for the bonus. 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. If you attempt to solve the bonus question, make sure that your zip includes both `Project2MYLET` and `Project2BONUS` folders separately. Name your submission files as:

p2_ member1IDno_ member1username_ member2IDno_ member2username.zip
Example: *p2_ 0011221_ baristopal20_ 0011222_ akutuk21.zip*

Please use *Project 2 Discussion Forum* on Blackboard for all your questions. The deadline for this project is April 17, 2022 - 23:59 (GMT+3 : Istanbul Time). **Read your task requirements carefully. Good luck!**

Table 1. Grade Breakdown for Project 2

Question	**Grade Possible**
Part A	15
Part B	10
Part C	5
Part D	70
Total	100
Bonus	2 pts to overall course grade

Problem Definition: To evaluate the programs, you need to understand the expressions of the language. It is the same for computers; therefore, you saw in the lecture how you can invent a language and define it for the computer to understand and evaluate. In this project, you will define a language named MYLET that is similar to the simple LET language covered in the class. The syntax for the MYLET language is given below.

```
Program ::= Expression
            a-program (exp1)

Expression ::= Number
            const-exp (num)

Expression ::= (num1 / num2)
            rational-exp (num1, num2)

Expression ::= String
            str-exp (str)

Expression ::= op(Expression, Expression, Number)
            op-exp (exp1, exp2, num)

Expression ::= zero?  (Expression)
            zero?-exp (exp1)

Expression ::= if Expression then Expression
                 {elif Expression then Expression}*
                 else Expression
            if-exp (exp1 exp2 conds exps exp3)

Expression ::= Indetifier
            var-exp (var)

Expression ::= let Indetifier = Expression in Expression
            let-exp (var exp1 body)
```

FIGURE 1. Syntax for the MYLET language

Part A (15 pts). This part will prepare you for the following parts of the project.

(1) Write the 5 components of the language[1]:
(2) For each component, specify where or which racket file (if it applies) we define and handle them.

Part B (10 pts). In this part, you will create an initial environment for programs to run.

(1) Create an initial environment that contains 3 different variables (`x`, `y`, and `z`).
(2) Using the environment abbreviation shown in the lectures, write how environment changes at each variable addition.

Part C (5 pts). Specify expressed and denoted values for `MYLET` language.

[1]Hint: review Lecture 10 slides

Part D (70 pts). This is the main part of the project where you implement the MYLET language given in the Figure 1 by adding the missing expressions.

(1) (15 pts) Add *str-exp* to the language. Strings are defined as any text starting and ending with ', e.g. 'comp301', 'program'; strings are stored with ' symbols.
Hint: *String* is an expression that is similar to *Number*, understanding the addition and implementation of *Number* may be helpful to complete this step.

(2) (20 pts) Add *rational-exp* to the language. In this design, rational numbers are kept as pairs, where the first element is the numerator and the second one is the denominator (e.g., 5/3 is stored as (5 . 3)). Like *const-exp*, you will add an additional structure where you can keep rational numbers as explained above. Please note that, *op-exp* and *zero?-exp* should also be implemented/changed for supporting also rational numbers. Additionally, you have to check if the denominator part of the rational number is zero and raise error in that case. Lastly, use *"cons"* to construct the pairs instead of *"list"*.

(3) (20 pts) Add *op-exp* to the language. *op-exp* is similar to the *diff-exp* of the LET language; however, in LET language, the only possible operation was subtraction. *op-exp* enables you to do 4 arithmetic operations via its third input (*Number*), when third input is:
- 1: perform addition (exp1 + exp2)
- 2: perform multiplication (exp1 * exp2)
- 3: perform division (exp1 / exp2)
- any other number: perform subtraction (exp1 - exp2)

All of these operations should support both integers and rational numbers. Still, you are not expected to simplify the resulting rational number (e.g., 5/3 + 9/6 will output 57/18, not 19/6).

(4) (15 pts) Add *if-exp* to the language. Unlike *if-exp* of the LET language, you can add multiple conditions to be checked through *elif-then* extension. Starting from the condition of if, conditions will be checked until a true condition is found, and expression corresponding to the true condition will be evaluated as a result. If none of the if/elif conditions are correct, the expression in the else statement will be evaluated.

Note 1: Note that the implementation of the other expressions, that are same with the LET language, are already given in the .rkt file provided. We deleted the former implementations of `if` and `diff-exp`.

Note 2: We provided several test cases for you to try your implementation. Uncomment corresponding test cases and run `tests.rkt` to test your implementation.

Bonus. Here is an alternative datatype *ropes* that allows manipulation of sequence of characters instead of the most commonly used `strings`. You can try to implement *rope*s instead of *string*s as a bonus challenge.

Note 3: The bonus question is worth 2 points in your overall final grade and no partial credits will be awarded. To get full credit, please implement this problem using the second code boilerplate (`Project2BONUS`) provided and write at least 6 test cases (two for each: fetch ith character, concatenate, substring) in a clear way to your `tests.rkt` for us to run. Please make sure that your test cases are clear and `tests.rkt` doesn't give any errors, otherwise you won't be able to receive any credits for this question. Add your code for the bonus problem to your submission as specified in the instructions.

Hint: Define your *rope* datatype similar to the way you did in the project, clearly define your grammar and feel free to use any helper procedures.