

Project 2

COMP301 Fall 2023

Deadline: November 24, 2023 - 23:59 (GMT+3 : Istanbul Time)

In this project, you will work in groups of three. To create your group, use the Google Sheet file in the following link (specify your group until November 16th, 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](#)

There is a single code boilerplates provided to you: `Project2MYLET`. 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:

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 November 24, 2023 - 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

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

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

Expression ::= create-new-list ()
            list-exp (lst)

Expression ::= cons Expression to Expression
            cons-exp (exp1 lst)

Expression ::= sum (Expression)
            sum-exp (lst)

Expression ::= max (Expression)
            max-exp (lst)

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

Expression ::= Identifier
            var-exp (var)

Expression ::= let Identifier = 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 language¹:
- (2) For each component, specify where or which racket file (if it applies) we define and handle them.

¹Hint: review Lecture 10 slides

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 the environment changes at each variable addition.

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

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

- (1) (10 pts) Add *list-exp* to the language. *list-exp* should take no arguments and should return an empty list.
- (2) (10 pts) Add *cons-exp* to the language. For our language, lists may only contain numbers. *cons-exp* should take a list and a number to add to the list.
- (3) (7 pts) Add *sum-exp* to the language. *sum-exp* should take a list and sum all its elements. An empty list is assumed to have a sum of 0.
- (4) (8 pts) Add *max-exp* to the language. *max-exp* should take a list as input and find the maximum number inside this list. An empty list should return -1.
- (5) (10 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 *cons-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 rational numbers. Additionally, you have to check if the denominator part of the rational number is zero and raise an error in that case. Lastly, use "*cons*" to construct the pairs instead of "*list*".
- (6) (15 pts) Support additional operations in *op-exp*. *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 5 arithmetic operations via its third input (*Number*) when third input is:
 - 1: perform addition ($\text{exp1} + \text{exp2}$)
 - 2: perform multiplication ($\text{exp1} * \text{exp2}$)
 - 3: perform division ($\text{exp1} / \text{exp2}$)
 - any other number: perform subtraction ($\text{exp1} - \text{exp2}$)

All of these operations should support both integers and rational numbers. In the given code, we have provided the implementations for addition and multiplication, and your task is to implement the rest. For all operations except the simplification operation, you are not expected to simplify the resulting rational number (e.g., $5/3 + 9/6$ will output $57/18$, not $19/6$).

- (7) (10 pts) Support the rational simplification operation *simpl-exp*. *simpl-exp* takes either a number or a rational and simplifies it. In the case of rationals, this is done by dividing both the numerator and the denominator by the greatest common divisor. This is trivial for numbers.

Note 1: We provided several test cases for you to try your implementation. Uncomment corresponding test cases and run `tests.rkt` to test your implementation.