Project 3
COMP301 Spring 2023

Deadline: May 5, 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 April 30, if you do not have any groups, then please write your name to *individuals* so that we can assign you to a group). You may either use the same groups as the previous project, or modify the assignments into new groups:

Link to Google Sheets for Choosing Group Members

There is a single code boilerplates provided to you: `Project3MyProc`. 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:

p3_member1IDno_member1username_member2IDno_member2username.zip
Example: *p3_0011221_baristopal20_0011222_akutuk21.zip*

The deadline for this project is May 5, 2023 - 23:59 (GMT+3 : Istanbul Time). **Read your task requirements carefully. Good luck!**

Table 1. Grade Breakdown for Project 3

Question	Grade Possible
Part A	7.5
Part B	7.5
Part C	75
Part D	10
Total	100

Problem Definition: As you have seen in the previous project, it is possible to implement a custom programming language using Scheme. A language can be created by specifying its syntax and behavior according to a given interface. In this project, you will extend the PROC language which you have seen in class to add more functionalities. The new language will be called MyProc. Its syntax is defined as follows:

Program ::= *Expression*

a-program (exp1)

Expression ::= *Number*

const-exp (num)

Expression ::= -(*Expression* , *Expression*)

diff-exp (exp1, exp2)

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 ::= *Identifier*

var-exp (var)

Expression ::= let *Identifier* = *Expression* in *Expression*

let-exp (var exp1 body)

Expression ::= proc (*Identifier*) *Expression*

proc-exp (var body)

Expression ::= (*Expression Expression*)

call-exp (rator rand)

Expression ::= empty-queue()

queue-exp(lst)

Expression ::= queue-push(*Expression*, *Expression*)

queue-push-exp (queue, exp)

Expression ::= queue-pop(*Expression*)

queue-pop-exp (exp)

Expression ::= queue-peek(*Expression*)

queue-peek-exp (exp)

Expression ::= queue-push-multi(*Expression*, {*Expression*}*)

queue-push-multi-exp (queue, exps)

Expression ::= queue-pop-multi(*Expression*, *Expression*)

queue-pop-multi-exp (queue, num)

Expression ::= queue-merge(*Expression*, *Expression*)

queue-merge-exp (queue1, queue2)

FIGURE 1. Syntax for the MyProc language

Part A (7.5 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 B (7.5 pts). Specify expressed and denoted values for `MyProc` language.

Part C (75 pts). This is the main part of the project where you implement the MyProc language given in Figure 1 by adding the missing expressions.

(1) (5 pts) Add *queue-exp* to the language. *queue-exp* should take no arguments and should return an empty queue. The queue can be represented as a list in the Scheme language.
(2) (10 pts) Add *queue-push-exp* to the language. For MyProc, we assume that the queue accepts only numbers. *queue-push-exp* should take a queue and a number that will be added to the beginning of the queue.
(3) (10 pts) Add *queue-pop-exp* to the language. *queue-pop-exp* should take a queue as input and return a queue also but with the element at the front of the queue removed. If the input queue is already empty, it should return an empty queue, but print an accompanying warning message.
(4) (10 pts) Add *queue-peek-exp* to the language. *queue-peek-exp* takes a queue as input and returns the element in the front of the queue. If the queue is empty, the expression must return the number 2000 which will denote a failed operation and also print a warning message.
(5) (15 pts) Add *queue-push-multi-exp* to the language. *queue-push-multi-exp* takes a queue as input, as well as an arbitrary number of expressions (which evaluate to a number each), and adds them all to the queue in the order in which they were given. For example, if the command `queue-push-multi(q, 1, 2, 3)` is evaluated, 1 will be added to the queue first, then 2, then 3 and so on.
(6) (15 pts) Add *queue-pop-multi-exp* to the language. *queue-pop-multi-exp* takes a queue and a number n as output. It returns a queue after removing n elements from the front of the queue. If n is greater than the size of the queue, all the elements are removed and a warning message must be printed.
(7) (10 pts) Add *queue-merge-exp* to the language. *queue-merge-exp* takes two queues as input and pops the elements of the second queue and pushes them one by one to the first queue. This means that the order in which the elements of the second queue are pushed to the first one, is the same order in which they are popped from the second one. It returns a single queue representing the merge operation of both queues.

    In the given code, we have provided the implementations for basic Proc that we have seen in the class, and your task is to implement the rest.

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.

Note 2: For the sake of passing the tests, it is important to implement the queue such that each new element is added to the leftmost side of the queue and popped from the right side.

Part D (10 pts). In this project, you implemented the queue data structure and its operations in MyProc. However, you delegated all operations in the defined language (MyProc) to the defining language (Scheme). In a more "proper" and "native" implementation, one would use low level memory operations at the CPU/assembly level, or low level resources available to the defined language to implement primitive data structures (such as pairs, arrays, queues), rather than delegating operations to data structures implemented in another high level language. Hence, the way you implemented the queue is not the best practice.
Can you think of a minimal set of resources, required properties or expressions that MyProc needs to have in order for us to be able to implement queues natively using MyProc expressions or MyProc resources rather than delegating everything to scheme ? Discuss.