Project 5
COMP301 Spring 2024

Deadline: May 24, 2024 - 23:59 (GMT+3 : Istanbul Time)

This is a practice project aimed at providing more chance for you to practice the concepts of the course.

This project contains a boilerplate provided to you use `project5_code.zip` for the project.

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 running `top.scm`.

Project Definition: In this project, you will implement common data structures such as vector and queue to `EREF`. Please, read each part carefully, and pay attention to *Assumptions and Constraints* section.

Part A. In this part, you will add vector to `EREF`. Introduce new operators newvector, update-vector, read-vector, length-vector, and swap-vector with the following definitions: (50 pts)

```
newvector:  ExpVal x ExpVal -> VecVal
update-vector:  VecVal x ExpVal x ExpVal -> Unspecified
read-vector:  VecVal x ExpVal -> ExpVal
length-vector:  VecVal -> ExpVal
swap-vector:  VecVal x ExpVal x ExpVal-> Unspecified
copy-vector:  VecVal -> VecVal
```

This leads us to define value types of `EREF` as:

```
VecVal = (Ref(ExpVal))*
ExpVal = Int + Bool + Proc + VecVal + Ref(ExpVal)
DenVal = ExpVal
```

Operators of vectors is defined as follows;

`newvector(length, value)` initializes a vector of size `length` with the value `value`.
`update-vector(vec, index, value)` updates the value of the vector `vec` at index `index` by value `value`.
`read-vector(vec, index)` returns the element of the vector `vec` at index `index`.
`length-vector(vec)` returns the length of the vector `vec`.
`swap-vector(vec, index, index)` swaps the values of the indexes in the vector `vec`.
`copy-vector(vec)` initializes an new vector with the same values of the given vector `vec`. (Creates a deep copy of the given vector.)

Part B. In this part, you will implement a Queue using vectors that you implemented in Part A.

Queue is a data structure that follows the FIFO (First-In-First-Out) principle and allows insertion operation from the rear end and deletion operation from the front end of the queue data structure. In other words, whenever dequeue is called, the element that is on the front of the queue is removed and returned from the queue. You will implement the following operators of Queue with the given grammar:

`newqueue(L)` returns an empty queue with max-size L.
`enqueue(q, val)` adds the element `val` to the queue `q`. If the queue is full it throws a stack overflow error.
`dequeue(q)` removes the first element of the queue `q` and returns its value.
`queue-size(q)` returns the number of elements in the `q`.
`peek-queue(q)` returns the value of the first element in the queue `q` without removal.
`queue-empty?(q)` returns true if there is no element inside the queue `q` and false otherwise.
`print-queue(q)` prints the elements in the queue `q`.

Part C (bonus). In this part, you will implement multiplication function for vectors, in order to support interpreter level vector multiplication, which is more efficient:

vec-mult takes two vectors, calculates their pairwise multiplication and outputs a new vector. If the sizes of the vectors are not equal, it throws an error

```
Expression ::= vec-mult(Expression, Expression)
               vec-mult-exp (vec1, vec1)
```

Example:

```
let x = newvector(3, 0) in
   let y = newvector(3, 0) in
      begin
         update-vector(x, 0, 1);
         update-vector(x, 1, 2);
         update-vector(x, 2, 3);
         update-vector(y, 0, 4);
         update-vector(y, 1, 5);
         update-vector(y, 2, 6);
         vec-mult(x, y)
      end
;;; [ 4 10 18 ]
```

FIGURE 1. Syntax for vec-mult Expression

Assumptions and Constraints. Read the following assumptions and constraints carefully. You may not consider the edge cases related to the assumptions.

(1) For queue, you may assume print-queue will only be used for queue of integers.
(2) Queue does not have to be new defined data types, you can utilize the vector implementation from Part A.
(3) It is guaranteed that the correct type of parameters will be passed to the operators. For example, in dequeue(q), q always be a queue.
(4) If queue is empty, dequeue operation must return -1.
(5) You CANNOT define global variables to keep track of the size or front and rear elements of a queue. The reason is we may create multiple queue and each of them may have different sizes and front and rear elements.
(6) If you consider using list of references for vector and queue, find an efficient way to reach to an elements, you are NOT allowed to iterate over list.