COMP301: Project 3

Project completed individually by Altun Hasanli.

Part A.

The initial environment is defined in `init-env` in `environments.rkt`.

Using the syntax in EOPL, p.61,

```
(init-env) => ρ
= [z=[304]]([y=[302]([x=[10]][]))
= [z=[304]]([y=[302][x=[10]])
= [z=[304]][y=[302], x=[10]]
= [z=[304], y=[302], x=[10]]
```

Part B.

In MyProc, expressed and denoted values are the same, and are as follows:

`ExpVal` = `DenVal` = `Int` + `Bool` + `Proc` + `List<Int>`

Defined in `data-structures.rkt`:

`num-val`, `bool-val`, `proc-val`, `queue-val`

Part C.

Implemented all queue expressions in MyProc. Passes all tests.

Part D.

To implement queues natively in MyProc without relying on Scheme or any other high-level language, here is a set of **minimal features** needed to implement queues natively:

Arrays

We can create a standard library for MyProc to support native arrays, and provide basic operations for arrays that rely on functions written in a more low-level language such as C, instead of using Scheme's pairs and lists. We could then, for example, port them over to `eopl`, using Racket's Foreign Function Interface, and make them available to the user in MyProc.

Here is an example of how it would look creating and using an array in MyProc. Assuming we have a dynamically resizing array implemented in C, and added these as library functions to the environment, so that we don't have to worry about running out of space.

```
; create an empty array
let arr = empty-num-array() in
  ; push two numbers to the array
  ; discard the result of num-array-push
```

```
  let _ = num-array-push(arr, 10, 12) in
    ; reference the array items and do subtraction on them
    -(num-array-ref(0), num-array-ref(1))
```

Comparison Operations

We need a way to compare numbers: `>`, `<`, `==`.

More ways to modify the environment

The only way to modify the environment in MyProc is defining a variable using `let`. We need a way to be able to modify existing variables, instead of creating new variables that shadow the previous ones.

```
let a = 12 in
  let _ = set!(a, 15) in
    print(a) ; => 15
```

More Control Flow

MyProc doesn't support recursion, so it would be impossible to iterate over arrays using recursion, for example if we implemented a library function such as `array-slice` and passed the slice to the next call of the function. We need a way to iterate over array elements:

```
; find the minimum element of an array
let find_min = proc(arr)
let min = -999999 in
  for num of arr in
    if <(num, min)
      then set!(min, num)
```

Native Queues at last

```
let empty-queue = proc () ...
    queue-push = proc (arr, num) ...
    queue-pop = proc (arr) ...
    queue-push-multi = proc (arr, nums) ... in
  let queue = empty-queue() in
    queue-pop-multi(queue-push-multi(queue, num-array(1,2,3,4,5)), 2)
```

Demonstration

To demonstrate how we could implement a standard library in a low-level language like C, and then port over to `eopl`, I implemented a queue in C, in the file `clib/queue.c`. Then the C file is compiled as a shared object in `clib/bin/queue.so`, as follows:

```
gcc -shared -o bin/queue.so queue.c
```

In `cproc.rkt`, the library functions are imported using Racket's FFI, as follows:

```
;; clib/queue library
(define clib/queue (ffi-lib "clib/bin/queue.so"))
```

```
;; clib/queue types
(define _Queue (_cpointer 'void))
(define _queue-pointer (_cpointer _Queue))

;; clib/queue/new : Int -> Ptr
(define clib/queue/new
  (get-ffi-obj "queue_new" clib/queue (_fun _int -> _queue-pointer)))
...
```

These functions operate on the queue pointer that point to the `Queue` struct in C. It's implemented as a circular buffer that automatically resizes based on capacity.

```
typedef struct {
    int *data;
    int front;
    int rear;
    int size;
    int capacity;
} Queue;
```

Then, I wrote a datatype wrapper around the queue pointer:

```
(define-datatype queue queue?
  (cqueue (ptr cpointer?)))
```

This type is then used in MyProc, as an `ExpVal` in a `queue-val` struct

```
(define-datatype expval expval?
  (num-val (num number?))
  (bool-val (bool boolean?))
  (proc-val (proc proc?))
  (queue-val (queue queue?)))
```

Then, I defined functions that operate on the struct using the C library functions:

```
;; cqueue:new : Void -> CQueue
(define (cqueue:new)
  (cqueue:new-with-capacity 10))

;; cqueue:empty? : CQueue -> Bool
(define (cqueue:empty? queue)
  (if (= (clib/queue/empty? (queue->ptr queue))
         0)
      #f #t))

;; cqueue:push : CQueue * Int -> CQueue
(define (cqueue:push queue num)
  (clib/queue/push (queue->ptr queue) num)
  queue)

;; cqueue:push-multi : CQueue * List<Int> -> CQueue
(define (cqueue:push-multi queue nums)
  (clib/queue/push-multi (queue->ptr queue)
                         (int-list->cvector-ptr nums)
                         (length nums))
  queue)

;; cqueue:merge : CQueue * CQueue -> CQueue
(define (cqueue:merge q1 q2)
```

```
  (cqueue (clib/queue/merge (queue->ptr q1)
                            (queue->ptr q2))))

...
```

The file `interp.rkt` is modified to use the above functions for operations on queues too. For the sake of passing the tests, I modified `sloppy->expval` and `equal-answer?` to retrieve reversed the list of elements from the `Queue` struct:

```
(define equal-answer?
  (lambda (ans correct-ans)
    (cases expval ans
      (queue-val (queue) (equal? (reverse (queue->list queue)) correct-ans))
      (else (equal? ans (sloppy->expval correct-ans))))))
```

At last, it passes the tests.

Summary

Since we are concerned about the minimal set of features, extending MyProc to support more operations, more control flow, and more ways to mutate the environment is an important first step. To have fast primitive types that rely on low-level memory operations, we can write the types and their methods in C, and add them to the environment of MyProc as standard library. This way, we can have arrays or queues readily available to the user. But, obviously, it would be nice to have more control in MyProc by providing more memory management features, allowing the user to define their own types in MyProc, wrap arrays and implement custom structures like queues, stacks, buffered lists or anything really.