CS355: Paradigms for Programming Lab

Midsem Exam (1.5 hours; 30 marks)

September 9^{th}, 2024

Instructions:

1. Use the system assigned to you for the exam.
2. You can refer to the slides that are present on your Desktop.
3. Internet and phone access are prohibited.
4. **The evaluation will be automated, so follow the next two submission instructions strictly.**
5. Save each solution in a file `Qi.scm`, where i is the question number. Enclose all your solution files in a folder `submission_rollnum`, where `rollnum` is your roll number (all small letters). Let this folder be there on the Desktop of your system.
6. Each file should be self sufficient, with `#lang sicp` at the top. Also, remove (or comment out) any tests and additional displays that you use for debugging.
7. Your submissions will be captured using a check-submit script, which is password protected. Do not leave until a TA submits your solutions.
8. We would have a few testcases for each question; you would get marks in proportion to the testcases passed. There is no manual evaluation for this exam.

LISTY LISPY

Q1 [3]. Diptanshu tries to write an "iterative" procedure `square-list` that squares all the elements of a list. That is:

```
> (square-list (list 1 2 3 4))
(1 4 9 16)
```

His attempt is given below:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things) (cons (square (car things)) answer))))
  (iter items nil))
```

Unfortunately, this produces the answer list in the reverse order of the one desired.

Then, Divyanshu tries to fix the bug by interchanging the arguments to `cons`:

```
(define (square-list items)
  (define (iter things answer)
    (if (null? things)
        answer
        (iter (cdr things) (cons answer (square (car things))))))
  (iter items nil))
```

This doesn't work either.

Write the correct definition of `square-list`.

Q2 [8]. First define a procedure `merge` that takes two sorted lists and merges them while preserving the order and removing the duplicates:

```
> (merge (list 8 10 22 45) (list 20 34 43 67))
(8 10 20 22 34 43 45 67)
```

Now define a procedure `merge-sort` that takes a list as input and uses your `merge` procedure to sort a given list:

```
> (merge-sort (list 45 8 22 10 89))
(8 10 22 45 89)
```

ORDER ORDER

Q3 [4]. The higher order procedure `for-each` is similar to `map`. It takes as arguments a procedure and a list of elements. However, rather than forming a list of the results, `for-each` just applies the procedure to each of the elements in turn, from left to right. For example:

```
> (for-each (lambda (x)
              (begin
                (newline)
                (display x)))
            (list 57 321 88))
57
321
88
```

The value returned by a call to `for-each` can be arbitrary, say we use `#t`. Give an implementation of `for-each`.

Q4 [4]. Use the higher order functions `map`, `filter` and `foldr` to write a procedure `odd-fibs` that gives the sum of all the odd Fibonacci numbers *fib(k)*, where *k* is less than or equal to a given integer *n*. That is:

```
> (odd-fibs 10)
99     ; (0 1 1 2 3 5 8 13 21 34 55)
```

Paste and use the definitions of `map`, `filter` and `foldr` from the file `hof.scm` present on your Desktop, in your solution. (Hint: There is a predefined procedure `odd?` in the `sicp` package.)

OOOOOOOOOO

Q5 [8]. Define a class `Queue` that supports the following operations:

- A zero-argument constructor that returns a `Queue` object.
- A one-argument procedure `enque` that adds an element into the rear end of the queue.
- A zero-argument procedure `deque` that removes and returns an element from the front end of the queue.
- A zero-argument procedure `isEmpty` that returns `#t` or `#f` denoting whether the queue is empty or not, respectively.

- A zero-argument procedure `printQ` that returns all the elements of the queue as a list, sequentially from the front to the rear.

An example interaction is given below:

```
> (define q (Queue))
> ((q 'enque) 10)
(10)
> ((q 'enque) 20)
(10 20)
> (q 'printQ)
(10 20)
> (q 'isEmpty)
#f
> (q 'deque)
(20)
> (q 'deque)
()
> (q 'isEmpty)
#t
```

Your queue operations should not throw any unhandled errors; whenever there is one, just `(display "error")`. Also, your queue implementation should be hidden from external users (*aka* "private" to the `Queue` class).

Q6 [3]. Inherit `Queue` to define another class `SetQueue` that does not allow duplicates. That is, trying to enque an element that already exists in the queue should leave the queue as is. Remaining operations on `SetQueue` objects should simply get passed on to the parent `Queue` instance.