CS 538

Project #1

Programming in Scheme

Due: Friday, March 22, 2002

(Not accepted after Monday, April 1, 2002)

Handin Instructions. Place the Scheme code you write to solve this assignment in `~cs538-1/public/handin/proj1/login` where `login` is your login name. Create files `1`, `2a`, `2b`, `2c`, `2d`, `3a`, `3b`, `4a`, `4b`, `4c`, `5a`, `5b` and `5c` in your handin directory.

1. You are to write a Scheme function `make-queue` that generates a function that implements a queue data structure. Recall that a queue operates like a "waiting line." New entries are added to the rear end of the queue, and entries are removed from the front of the queue.
   The call `(make-queue)` should return a function that represents an initially empty queue. If `q` is a queue function, it should accept the following calls:

   `(q enter! val`$_1$ `val`$_2$ `...)`
   `q` adds `val`$_1$, `val`$_2$, ... to the end of the queue it represents, with `val`$_1$ being the last value entered (and thus at the very end of the queue). The list `(val`$_1$ `val`$_2$ ... `)` is returned.

   `(q remove! cnt)`

   `cnt` is a non-negative integer. `cnt` values are removed from the queue `q` represents. A list containing the values removed is returned (with the first value removed at the right end of the list). If the queue does not contain `cnt` values, then `#f` is returned (as an error indication).

   `(q contents)`

   Returns a list representing the contents of `q`. The last element of the queue appears at the left end of the list; the first element of the queue appears at the right end of the list

   `(q clone)`

   Returns a new queue function whose contents are (initially) identical to the contents of `q`.

   The following calls illustrate how queue functions are expected to behave:
   ```
   (define my-queue (make-queue))
   (my-queue 'enter! 4 5 6) ⇒  (4 5 6)
   (my-queue 'enter! 1 2 3) ⇒  (1 2 3)
   (my-queue 'contents) ⇒  (1 2 3 4 5 6)
   (my-queue 'remove! 2) ⇒  (5 6)
   ```

```
(my-queue 'contents) ⇒  (1 2 3 4)
(define your-queue (my-queue 'clone))
(your-queue 'contents) ⇒  (1 2 3 4)
(my-queue 'remove! 1) ⇒  (4)
(my-queue 'contents) ⇒  (1 2 3)
(your-queue 'contents) ⇒  (1 2 3 4)
```

2. (a) In Scheme sets can be represented as lists. However, unlike lists, the order of values in a set is not significant. Thus both `(1 2 3)` and `(3 2 1)` represent the same set. Moreover, the same value may not appear more than once within a list that represents a set. Thus `(1 2 3)` is a valid set, but `(1 2 1)` is not.

   Write a Scheme function `(valid-set? S)` that tests whether set `S` is valid. A set is valid if no value in the list appears more than once. In this part you may assume that sets contain only atomic values (numbers, strings, symbols, etc.). For example

```
(valid-set?  '(1 2 3)) ⇒  #t
(valid-set?  '(1 2 1)) ⇒  #f
(valid set?  '(1 1.0 "one")) ⇒  #t
```

   (b) Write a Scheme function `(equal-sets? S1 S2)` that tests whether sets `S1` and `S2` (represented as lists) are equal. Two sets are equal if they contain exactly the same members, ignoring ordering. In this part you may assume that sets contain only atomic values (numbers, strings, symbols, etc.). If either `S1` or `S2` is invalid, return `#f` (as an error indication). For example

```
(equal-sets?  '(1 2 3)  '(3 2 1)) ⇒  #t
(equal-sets?  '(1 2)  '(3 2 1)) ⇒  #f
(equal-sets?  '(curly larry moe)  '(moe larry curly)) ⇒  #t
(equal-sets?  '(1 1)  '(1 1)) ⇒  #f
```

   (c) Two common operations on sets are **union** and **intersection**. The union of two sets is the set of all elements that appear in either set (with no repetitions). The intersection of two sets is the set of elements that appear in both sets.

   Write Scheme functions `(union S1 S2)` and `(intersect S1 S2)` that implement set union and set intersection. You may again assume that set elements are always atomic. If either `S1` or `S2` is invalid, return `#f` (as an error indication). For example

```
(union  '(1 2 3)  '(3 2 1)) ⇒  (1 2 3)
(union  '(1 2 3)  '(3 4 5)) ⇒  (1 2 3 4 5)
(union  '(a b c)  '(3 2 1)) ⇒  (a b c 1 2 3)
(union  '(1 1)  '(2 2)) ⇒  #f
(intersect  '(1 2 3)  '(3 2 1)) ⇒  (1 2 3)
(intersect  '(1 2 3)  '(4 5 6)) ⇒  ()
(intersect  '(1 2 3)  '(2 3 4 5 6)) ⇒  (2 3)
(intersect  '(1 1)  '(2 2)) ⇒  #f
```

(d) In general sets can contain other sets. Extend your solution to parts (a), (b) and (c) to allow sets to contain other sets. For example,

```
(valid-set?  '(1 (1) 2)) ⇒  #t
(valid-set?  '((1) 2 (1))) ⇒  #f
(equal-sets?  '(1 (2 3))  '((3 2) 1)) ⇒  #t
(equal-sets?  '(1 2 3)  '((3 2) 1)) ⇒  #f
(equal-sets?  '(1 2 3)  '((1 2 3)) ⇒  #f
(union  '(1 (2) 3)  '(3 2 1)) ⇒  (1 2 3 (2))
(union  '((1 2 3))  '((3 4 5))) ⇒  ((1 2 3) (3 4 5))
(union  '((1 2 3))  '((3 2 1))) ⇒  ((1 2 3))
(intersect  '((1 2 3))  '((3 2 1))) ⇒  ((1 2 3))
(intersect  '((1 2 3))  '((4 5 6))) ⇒  ()
(intersect  '((1) (2) (3))  '((2) (3) (4))) ⇒  ((2) (3))
```

3. (a) Write a pair of Scheme functions, `(gen-list start stop step)` and `(pair-prod? list val)`. The function `gen-list` will generate a list of integers, from `start` to `stop`, with consecutive values incremented by `step`. (If `start` > `stop` then the empty list is generated). For example `(gen-list 1 9 2)` ⇒ `(1 3 5 7 9)`.

The predicate `pair-prod?` tests whether any two adjacent values in `list` have a product equal to `val`. For example, `(pair-prod? '(1 2 3) 6)` ⇒ `#t` since 2*3=6. Similarly, `(pair-prod? (gen-list 1 100 1) 10)` ⇒ `#f` since no two adjacent integers in the range 1 to 100 have a product of 10.

(b) A problem with a function like `pair-prod?` is the fact that its `list` parameter must be fully computed in advance, even if all the values on the list are not needed. For example, `(pair-prod? (gen-list 1 100000 1) 2)` will spend a lot of time and space building up a list of 100000 values, even though only the first two are needed!

An alternative to completely building a complex data structure is **lazy evaluation**. That is, part of the structure is built along with a **suspension**—a function that will supply a few more values upon request. The following two Scheme functions produce a sequence of integers in a lazy manner.

```
(define (return-one-val start stop step)
   (if (> (+ start step) stop)
       (cons start #f)
       (cons start
            (lambda () (return-one-val (+ start step) stop step)))
   )
)

(define (int-seq start stop step)
     (if (> start stop)
         (cons #f #f)
         (return-one-val start stop step)
     )
)
```

When called, `int-seq` returns a pair of values. The *car* is the first integer in the sequence, or `#f` if the sequence is empty. The *cdr* is a function that, when called, will return another pair. That pair consists of the next integer in the sequence plus another suspension function. When the end of the sequence is reached, the *cdr* of the pair is `#f` (rather than a function), indicating that no more values can be produced.

Create a new version of `pair-prod?` called `pair-prod-seq?` that takes a lazy integer sequence (as defined by `int-seq`) rather than a list as a parameter. You can use the Scheme function `(time (f args))` to time the evaluation of `(f args)`. Compare the execution times of `(pair-prod? (gen-list 1 100000 1) 90)` and `(pair-prod-seq? (int-seq 1 100000 1) 90)`. Which is faster? Why? Now compare the execution times of `(pair-prod? (gen-list 1 100000 1) 91)` and `(pair-prod-seq? (int-seq 1 100000 1) 91)`. Again, which is faster, and why?

4. (a) Scheme can represent numbers in a variety of forms, including integer, rational, real and complex. Let's add a new form—*English*. That is, we will allow numbers to be represented as a list of symbols representing English language words corresponding to numbers. Thus `0` would be represented as `(zero)`, `123` as `(one hundred twenty three)` and `-456789` as `(minus four hundred fifty six thousand seven hundred eighty nine)`.

   Assume that we represent numbers (in English) using the symbols `minus`, `zero`, `one`, `two`, ... , `ten`, `eleven`, `twelve`, ..., `nineteen`, `twenty`, `thirty`, ..., `ninety`, `hundred`, `thousand`, `million`, ... For simplicity, numbers that are normally hypenated (like twenty-three) won't be.

   Write a Scheme function `(integer->english int-val)` that converts an integer `int-val` into `english` form (that is, a list of symbols representing the number in English). Your function should handle numbers at least as large as one trillion (handling even larger numbers is fairly easy). For numbers larger than your implementation limit (of one trillion or more) you may return the integer unconverted.

   (b) Now write an inverse function `(english->integer L)` that converts a number in `english` form back to `integer` form. Of course, it should be the case that `(english->integer (integer->english int-val)) = int-val` for all integers in the range you handle.

   (c) Finally, you are to redefine the `+`, `-`, `*` and `/` functions so that they accept numbers in `english` form as well as `integer` form (you may ignore numbers in rational, real and complex forms). You may assume that your extended operators will be called with only two operands. If both operands are in `integer` form, then you will return an `integer` result. If one or both are in `english` form, then you will return a result in `english` form. For `/`, you may use `truncate` to convert fractional quotients to integers; that is, you need not worry about divisions that have a non-zero remainder.

The following are sample calls that you should handle:

`(+ 123 456)` ⇒ `579`

`(- 111111 '(ten))` ⇒ `(one hundred eleven thousand one hundred one)`

`(* '(two thousand fifty five) '(one million one))` ⇒ `(two billion fifty five million two thousand fifty five)`

`(/ 123456789 '(three))` ⇒ `(forty one million one hundred fifty two thousand two hundred sixty three)`

5. (a) Assume we have a list `L` of integers. Define a function `(listify L)` that divides `L` into one or more sublists so that each sublist contains integers in non-decreasing (sorted) order. That is, if `v1` and `v2` are adjacent in `L` and `v1` ≤ `v2` then `v1` and `v2` are adjacent in the same sublist. However if `v1` > `v2` then `v2` ends one sublist and `v2` begins the next sublist. For example,

```
(listify '(3 5 1 8 9 2 1 0)) ⇒ ((3 5) (1 8 9) (2) (1) (0))
(listify '(1 2 3 4 5 6)) ⇒ ((1 2 3 4 5 6))
(listify '(5 4 3 2 1)) ⇒ ((5) (4) (3) (2) (1))
```

(b) If the output of `listify` contains a single sublist, then we know the input list `L` was in sorted order. This makes it easy to test for duplicates within L—we just compare adjacent values. If listify returns more than one sublist, we can merge the first two lists into one sorted list, and repeat the process until we have one sorted list. Then testing for duplicates is easy.
For example, if `listify` initially produces
`((3 5) (1 8 9) (2) (1) (0))` we reduce it to `((1 3 5 8 9) (2) (1) (0))`, then to `((1 2 3 5 8 9) (1) (0))`, then to `((1 1 2 3 5 8 9) (0))`, and finally to `((0 1 1 2 3 5 8 9))`.

Implement `(duplicates? L)`, which sorts the values in `L` using `listify` as a subroutine and tests for duplicates.

(c) Our algorithm for testing for duplicates is somewhat inefficient. If a duplicate appears in `L` it can readily be detected when `listify` initially partitions `L` or when the sublists produced by `listify` are merged.

Create a function `duplicates-cc?` (and whatever auxiliary functions you require) that make duplicate checking *integral* to the implementation of `listify` and the merge component of the sort. As soon as a duplicate is seen, `#t` should be immediately returned, without any further processing of `L`. You should use `call-with-current-continuation` to implement the exception mechanism you will need to return when you see a duplicate value.