The University of British Columbia
Department of Computer Science

Computer Science 124
Practice examination

Do not open this paper until you are ready to write this exam!

This is a practice examination, which is intended to give you a chance to try some problems which might be found on a CPSC 124 final examination. You should attempt this under exam conditions, i.e., give yourself a quiet, undisturbed place to work, use no notes or books (except for one sheet of notes), and stop after the time allotted.

NO WARRANTY IS GIVEN THAT THE FINAL EXAMINATION IS GOING TO BE ANYTHING LIKE THIS ONE. IN FACT, IT PROBABLY WON'T BE. HOWEVER, DOING THIS PRACTICE EXAM, UNDER EXAMINATION CONDITIONS, CAN BE A VALUABLE PART OF A COMPREHENSIVE STUDY AND REVIEW PROCESS FOR THE FINAL EXAMINATION.

There are 120 marks; give yourself two hours, allowing for one mark per minute. The actual exam will be two and one-half hours (and will be slightly longer than this).

1) (20 points) There are five independent parts (plus one strictly optional extra) to this problem. No explanation of your work is required. Determine your answer carefully: no partial credit will be given. For each part, you are given a sequence of expressions. Assume that each sequence is presented to Scheme in the order given. Write down the value that will be printed in response to the last expression in the sequence.

a)

```
(define a (list 'a 'b 'c))
(define b (cons 'b (quote (a b))))
b
```

b)

```
(define a (list 'a 'b))
(define b (cons a a))
(set-car! b 'b)
b
```

c)

```
(define f (lambda (n)
            (if (= n 0)
                3
                (if (= n 1)
                    1
                    (+ (f (- n 1))
                       (f (- n 2)))))))
(f 4)
```

d)

```
(define word (string-append "down" "upside"))
(if (string=? word "upsidedown")
      (substring word 1 5)
      (substring word 3 8))
```

e)

```
(define a (list 'a 'b 'c))
(define b (list 'a 'b 'c))
(if (equal? a b)
     (if (eq? a b) 0 1)
     (if (eq? a b) 2 3))
```

f) (Here's an extra question for practice!)

```
(((lambda (f g) (f g))
  (lambda (f) (f))
    (lambda () (lambda (f g) (g f))))
      (lambda (f g) (g (f 3)))
(lambda (f) (f (lambda (g) (* g g)) 1+)))
```

(This one is a quite confusing. Don't worry too much if you can't work it out! We wouldn't put this on a real exam — but it is a good test of your skills!)

2) (15 points) Suppose we have typed the following to Scheme:

```
(define p
  (list (quote a)
    (list 'b 'c)
      '(d (e))))
```

a) Draw the box and arrow diagram for `p`.

b) What will Scheme print in response to:

```
p
```

c) Now suppose we had typed:

```
(define q '(x (2 3 4 5) (a (b c) d) 27))
```

What will Scheme print in response to:

```
(car (car (cdr q)))
```

3) (15 points) Consider the procedure `count-foos` which takes a list as an argument and counts the number of occurrences of the symbol `foo` in it, e.g.,

```
> (count-foos '(1 foo 3 foo))
2
> (count-foos '(1 2 3 4 5))
0
> ; the list (foo 3) is not the symbol foo
> (count-foos '(foo (foo 3) 4))
1
```

Here is a definition of `count-foos`:

```
(define count-foos
  (lambda (lst)
    (if (null? lst)
        0
        (if (eq? (car lst)  'foo)
            (+ 1 (count-foos (cdr lst)))
            (+ 0 (count-foos (cdr lst)))))))
```

a) Suppose that `symb-list` is a list of n symbols. How many times is `count-foos` called during the evaluation of

```
(count-foos symb-list)
```

b) Using $O(\ldots)$ notation, express the time complexity of the procedure `count-foos`, as a function of the length of its argument.

c) Using $O(\ldots)$ notation, express the space complexity of the procedure `count-foos`, as a function of the length of its argument.

4) (20 points) A *stack* is a set of objects conceptually arranged like stacks of trays or plates at your favorite dining hall. An object that is PUSHed onto the stack becomes the TOP element: POPping a stack removes the top element. This behavior means that stacks obey a Last In First Out (LIFO) discipline.

A description of the set of operations defined for stacks appears below:

`(stack-create)` ⇒ *stack*
Returns an empty stack.

`(stack-empty?` *stack*`)` ⇒ *boolean*
Returns #t if the stack has no elements and #f otherwise.

`(stack-push` *obj stack*`)` ⇒ *stack*
Returns a new stack has that has *obj* at its top. Note that pushing an item onto the stack *doesn't mutate* the original stack, but creates a new stack.

`(stack-pop` *stack*`)` ⇒ *stack*
Returns a new stack identical to *stack* with the top element removed. Generates an error if *stack* is empty. Doesn't mutate the original stack, but creates a new stack.

`(stack-top` *stack*`)` ⇒ *obj*
Returns the top object on *stack*, but doesn't remove it. Generates an error if *stack* is empty.

`(stack-elements` *stack*`)` ⇒ *list of objects*
Returns a list of the objects on *stack*, without modifying the stack itself.

The following interactions illustrate uses of the above operations.

```
> (define a
    (stack-push 1
       (stack-push 2
         (stack-push 3
                     (stack-create)))))
> (stack-elements a)
(1 2 3 )
> (stack-top a)
1
> (define b (stack-pop a))
> (stack-elements b)
( 2 3 )
> (stack-elements a)
( 1 2 3 )
> (stack-empty? b)
#f
> (stack-empty? (stack-pop (stack-pop b)))
#t
> (stack-top (stack-pop (stack-pop b)))
<<error message of some sort, doesn't really matter>>
```

Implement the above operations for a stack assuming a stack is represented as a list whose first element is the top of the stack. Is this the only possible representation? Explain your answer.

5) (30 marks total) Using a Scheme program we wish to model a music CD collection consisting of a large number of CDs categorized by the name of the artist. In order to keep track of the changing number of CDs in the collection by each artist, we want to develop an abstract data type (ADT) called *CD collection.* Each *CD collection* consists of two pieces of information: the total number of CDs in the collection and an *artist list.* Each element in the artist list consists of an artist's name and the number of CDs by that artist. We have decided to store the information for a CD collection in a pair: the `car` of the pair is the total number of CDs in the collection, while the `cdr` of the pair is the artist list. The elements of the artist list are themselves pairs, with the `car` of each being the artist name and the `cdr` of each being the number of CDs by the artist.

We provide a constructor for this ADT as follows:

```
;;; Constructor for CD collection
(define make-CD-coll
  (lambda ()
    (cons 0 '()) ))
```

Thus, initially every CD collection has no CDs. We assume that the accessors `total-CDs` and `CD-list` are defined as follows:

```
;;; Accessors for CD collection
(define total-CDs car)
(define artist-list cdr)
```

You are to write a *mutator* called `add-CDs!` and an *accessor* called `surplus-CDs` as described below.

(A) (15 marks) Write a mutating procedure `add-CDs!` that accepts three arguments: a CD collection, a symbol representing an artist (e.g., `'lopez`), and the number of new CDs (e.g., 3) for that artist. The procedure should **mutate** the CD collection to update the total number of CDs in the collection and the artist list. If there is an entry for this artist in the artist list, then the procedure should add the number of new CDs to the artist's total. Otherwise the procedure should add an appropriate element to the end of the artist list. For example, we expect the following behavior:

```
>(define Coll (make-CD-coll))
>(add-CDs! Coll 'lopez 5)
>(add-CDs! Coll 'spears 2)
>(add-CDs! Coll 'lopez 3)
>(add-CDs! Coll 'beatles 10)
>(total-CDs Coll)
 20
>(artist-list Coll)
 ((lopez . 8) (spears . 2) (beatles . 10))
```

(B) (15 marks) If the number of CDs (`T`) by a particular artist exceeds 15% of the collection's total number of CDs (`C`), then there are `(T - floor(0.15 * C))` *surplus* CDs by this artist in the collection. Write a procedure `surplus-CDs` that accepts as input a CD collection and returns an *surplus artist list.* Each element of the surplus artist list should be a pair with the symbol of a surplus artist in its `car` and the number of surplus CDs in its `cdr`. Your procedure may arrange the elements of the returned surplus list in any order you wish. (Note that the Scheme form\ `(floor x)` returns the largest integer less than or equal to the real number `x` \(e.g., `(floor 3.51)` evaluates to `3`).

As an example, if `surplus-CDs` is applied to the CD collection `Coll` from the previous page, then we expect the following behavior:

```
> (surplus-CDs Coll)
((lopez . 5.0) (beatles . 7.0))
```

In this example, there are surplus CDs for any artist with more than 3 CDs in the collection, since there are a total of 20 CDs and `floor(0.15 * 20)` is 3. Consequently, the collection has 5 surplus Jennifer Lopez CDs (since the collection has 8 Lopez CDs) and 7 surplus Beatles CDs.

6) (20 points) Write a procedure `make-sequence`, with two arguments `a` and `b`, that returns a procedure with no arguments. When that procedure is applied, it returns the next number in the sequence of numbers $ai+b$, starting with $i = 1$. The following interactions illustrate the use of this procedure.

```
> (define make-sequence (lambda (a b) .......))
> (define plus1 (make-sequence 1 1))
> (plus1)
2
> (plus1)
3
> (plus1)
4
> (define evens (make-sequence 2 0))
> (evens)
2
> (evens)
4
```

Draw the environment diagram after evaluating

```
> (define make-sequence (lambda (a b) .......))
> (define plus1 (make-sequence 1 1))
> (plus1)
2
```