Spring, 1992 Name: ______________________

CS 342 — Principles of Programming Languages

CLU Test (Problem Solving, Version 1)

Before you begin, please take a moment to look over the entire test so that you can budget your time.
For programming problems, there is usually more than one right answer, hence it is important to present with your program evidence that it is correct. Answers to programming questions will be scored on the following points:

- elegance and clarity (conciseness, using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution, and how clear and convincing your argument is that the code is correct)
- correctness (whether your program solves the given problem correctly in the given programming language)

You should try to eliminate all syntax errors from your programs, as they will affect both the correctness and clarity of your solution; you may receive no points for programs with major syntax errors. Run-time space and time efficiency of programs is not important, unless the problem says so. Also, checking of external inputs, prompting, labeling output, and other conventions for dealing with humans are not important (for this class), unless the problem says so.
The breakdown of points between these categories is roughly as follows.

- 60% for elegance and clarity (using relevant techniques to modularize the solution, making good use of language features, the creativity of your solution),
- 40% for correctness (whether your program solves the given problem correctly in the given subset of the given language)

You are encouraged to use auxiliary subroutines whenever they are helpful.
In your solutions you may use the `List`, `Array`, and `Pair` clusters from the textbook (pages 216, 218, and 218). You may also use any of the procedures below. Any other clusters or procedures that you wish to use must be written in your solution. If you wish to use functions from your solutions in other parts of the exam, you must clearly state where we can find those function definitions.

```
(define abs (x) (if (< x 0) (- 0 x) x))
(define and (x y) (if x y x))
(define or (x y) (if x x y))
(define not (x) (if x 0 1))
(define <> (x y) (not (= x y)))
(define >= (x y) (or (> x y) (= x y)))
(define <= (x y) (or (< x y) (= x y)))
(define mod (m n) (- m (* n (/ m n))))
(define min (x y) (if (< x y) x y))
(define max (x y) (if (> x y) x y))
;
(define length (l)
    (if (List$null? l) 0 (+1 (length (List$cdr l)))))
(define nth (n l)
    (if (= n 0) (List$car l) (nth (- n 1) (List$cdr l))))
(define changenth (n x l)
    (if (= n 0) (List$rplaca l x) (changenth (- n 1) x (List$cdr l))))
;
(define assoc (i l)
```

```
(if (List$null? l) l
     (if (= (Pair$fst (List$car l)) i)
          l
          (assoc i (List$cdr l)))))
```

1. (7 points) Write a function `(equal lst1 lst2)` that takes two lists of integers and returns 1 (true) if and only if they have the same elements in the same order, and 0 (false) otherwise. That is, `(equal (List$nil) (List$nil))` is true; if both $w = x$ and `(equal` y z`)`, then `(equal (List$cons` w y`)` `(List$cons` x z`)` is true; and in all other cases the result is false. You should assume that each top-level element in the list arguments is an integer (there are no nested sublists).

2. (20 points) Harry Horsewise of Altoona is a former graduate of CS 342 and a bookie. He takes bets on horse races and needs to record his customers bets on the computer. Because he really liked CLU in 342, he has commissioned you to write a simple program in CLU for him. Since off track betting is illegal in Iowa, Harry doesn't want you to know the names of his customers, so he refers to all his customers by number. To keep things simple, we'll assume that all that has to be recorded is how much money each customer has bet.

   Harry specifies that he wants to be able to run the following procedures.

```
(define start ()
  (set db (Bets$new)))
(define place-bet (customer-number amount)
  (Bets$credit db customer-number amount))
(define debit (customer-number amount)
  (Bets$debit db customer-number amount))
(define balance (customer-number)
  (Bets$balance db customer-number))
```

   Harry has specified how the procedures should work by noting that `Bets$new` should produce an object of type `Bets`, that `Bets$credit` should add the amount to the customer's account; that `Bets$debit` should subtract the amount from the customer's account; and that `Bets$balance` should return an integer that tells how much remains in the customer's account. He's also provided the following sample interaction with the program (running in the CLU interpreter), where the results that are user values (`<userval>`) or not specified are simply not shown.

```
-> (start)
-> (place-bet 27 300)
-> (place-bet 2 1000)
-> (place-bet 27 400)
-> (balance 27)
700

-> (debit 27 100)
-> (balance 27)
600

-> (debit 2 1000)
-> (balance 2)
0
```

   Your task is to finish the program by implementing the cluster `Bets`. You may assume that Harry's customer numbers run from 1 to the value of the global variable `max-cust-num`. However, you do not need to check that the customer numbers are legal. You do not need to do sanity checking on how much money is credited or debited from a customer's account. You may assume that every session with the program starts with a call to `start`.

   (You may want to write your answer on the next page.)

3. (20 points) Fill in the missing code in cluster below to implement the type `PointOrInt` in the following CLU cluster definition. An object of type `PointOrInt` can be thought of, abstractly, as consisting of a tag and a value. The tag is either POINT or INT; when it is POINT, the value must be a `Point`, and when it is INT, the value must be an integer. For those who have heard the terms before, this is a tagged, discriminated union. This is like a variant record in Pascal.

```
(cluster PointOrInt
    ; Export: makePoint, makeInt, valuePoint, valueInt, isPoint, isInt,
    ;         changeToPoint, changeToInt
    (rep                                )  ; you must fill this in

    (define makePoint (p)
        ; TYPE:  Point -> PointOrInt (i.e. takes a Point, returns a PointOrInt)
        ; EFFECT: return a PointOrInt with tag POINT that contains p



    )
    (define makeInt (i)
        ; TYPE:  Int -> PointOrInt (i.e. takes an Int, returns a PointOrInt)
        ; EFFECT: return a PointOrInt with tag INT that contains i



    )
    (define valuePoint (pOri)
        ; TYPE: PointOrInt -> Point
        ; REQUIRES: pOri has tag POINT.
        ; EFFECT: returns the Point that is contained in pOri



    )
    (define valueInt (pOri)
        ; TYPE: PointOrInt -> Int
        ; REQUIRES: pOri has tag INT
        ; EFFECT: returns the integer that is contained in pOri



    )
    (define isPoint (pOri)
        ; TYPE: PointOrInt -> bool
        ; EFFECT: if p has tag POINT, return 1, else return 0



    )
    (define isInt (pOri)
        ; TYPE: PointOrInt -> bool
        ; EFFECT: if p has tag Int, return 1, else return 0


    )
```

```
    (define ChangeToPoint (pOri p)
        ; TYPE: PointOrInt * Point -> ()
        ;  (i.e., takes a PointOrInt and a Point and has no return value)
        ; MODIFIES: pOri
        ; EFFECT: make the tag of pOri POINT and the value be p



    )
    (define ChangeToInt (pOri i)
        ; TYPE: PointOrInt * Int -> ()
        ;  (i.e., takes a PointOrInt and an Int and has no return value)
        ; MODIFIES: pOri
        ; EFFECT: make the tag of pOri INT and the value be i



    )






)
```