Spring, 1992 Name: _____

CS 342 — Principles of Programming Languages

CLU Test (Problem Solving, Version 2)

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 `(pair-up lst1 lst2)` that takes two lists of integers, both of which have the same length, and returns a single list of pairs; the list of pairs contains a pair of the first elements of `lst1` and `lst2`, then the pair of the second, etc. That is, `(pair-up (List$nil) (List$nil))` returns an empty list, and for non-empty lists, the following equations should be satisfied.

   `(Pair$fst (List$car (pair-up (List$cons` x l_1`) (List$cons` y l_2`)))) =` x
   `(Pair$snd (List$car (pair-up (List$cons` x l_1`) (List$cons` y l_2`)))) =` y
   `(List$cdr (pair-up (List$cons` x l_1`) (List$cons` y l_2`))) = (pair-up` l_1 l_2`)`

   You must use an operation of the cluster `Pair`, defined on page 218 of the book, in your solution. You may assume that the arguments `lst1` and `lst2` have the same length.

2. (20 points) For this problem tags are represented by the following cluster.

```
(cluster Tag
   (rep tagValue)
   (define EMPTY () (Tag 0))
   (define INT () (Tag 1))
   (define equal (t1 t2) (= (tagValue t1) (tagValue t2)))
)
```

Fill in the missing code in cluster below to implement the type `MaybeInt` in the following CLU cluster definition. An object of type `MaybeInt` can be thought of, abstractly, as consisting of a tag and a value. The tag is either INT or EMPTY; when it is INT, the value must be an integer, and when it is EMPTY, there is no value. You must use the cluster `Tag` above in your solution.

```
(cluster MaybeInt
    ; Export: makeInt, makeEmpty, hasTag?, value, changeToInt
    (rep                                )  ; you must fill this in
    (define makeInt (i)
        ; TYPE:  Int -> MaybeInt (i.e., takes an Int, returns a MaybeInt)
        ; EFFECT: return a MaybeInt with tag INT that contains i


    )
    (define makeEmpty ()
        ; TYPE:  -> MaybeInt (i.e., no arguments, returns a MaybeInt)
        ; EFFECT: return a MaybeInt with tag EMPTY


    )
    (define hasTag? (mi t)
        ; TYPE: MaybeInt, Tag -> Bool
        ; EFFECT: returns 1 if mi has the same tag as t



    )
    (define value (mi)
        ; TYPE: MaybeInt -> Int
        ; REQUIRES: mi has tag INT
        ; EFFECT: returns the integer that is contained in mi



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



    )
)
```

3. (20 points) On the planet Skyron in the galaxy of Andromena, the inhabitants use polynomials instead of numbers to count how much money they have.

   Through a curious coincidence, a former 342 student, Sally Starwise, finds herself on the planet Skyron. As the author of the best-seller *Earth*, Sally has become incredibly wealthy, and has therefore commissioned you to write a program to keep track of her savings and checking accounts.

   Sally's checking accounts has overdraft protection; that is, if the withdrawal of a polynomial from her checking account would cause it to be negative, the balance is made up from the savings account. Polynomial money is considered to be negative if the coefficient of the highest power of x in the polynomial is negative; for example $-200x + 3$ is negative.

   Sally specifies that she wants to be able to run the following procedures.

```
(define start ()
  (set accounts (Accts$new)))
(define deposit-checking (poly-money)
  (Accts$deposit-checking accounts poly-money))
(define deposit-savings (poly-money)
  (Accts$deposit-savings accounts poly-money))
(define withdraw-checking (poly-money)
  (Accts$withdraw-checking accounts poly-money))
(define balance-checking ()
  (Accts$balance-checking accounts))
(define balance-savings ()
  (Accts$balance-savings accounts))
```

   Sally has specified how the procedures should work by noting that `Accts$new` should produce an new object of type `Accts`, that `Accts$deposit-checking` and `Accts$deposit-savings` should add the given polynomial to the appropriate account. `Accts$withdraw-checking` should subtract the given polynomial from the her checking account, taking enough from the savings account to ensure that the checking account is not negative, if there is enough in the savings and checking accounts to cover the withdrawal; since Sally is so wealthy, you may assume that there is enough money in the savings account so that it will never be made negative by taking out enough to cover the checking withdrawal. Finally, `Accts$balance-checking` and `Accts$balance-savings` should print the polynomial that is her balance in the appropriate account. Sally has 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. See page 228 for how polynomials print.

```
-> (start)
-> (balance-savings)
0
0
-> (balance-checking)
0
0
-> (deposit-checking (Poly$create 1000 0))
-> (deposit-checking (Poly$create 300 1))
-> (balance-checking)
300
1
1000
0
-> (Poly$prnt (Poly$add (Poly$create 1000 0) (Poly$create 300 1)))
300
1
```

```
1000
0
-> (withdraw-checking (Poly$create 100 1))
-> (balance-checking)
200
1
1000
0
-> (deposit-savings (Poly$create 5000 2))
-> (balance-savings)
5000
2
0
0
0
0
-> (withdraw-checking (Poly$create 400 1))
-> (balance-checking)
1000
0
-> (balance-savings)
5000
2
-400
1
0
0
```

Note that at the end Sally's checking account balance is $0x + 1000$; and her savings balance is $5000x^2 + -400x + 0$. Your task is to finish the program by implementing the cluster `Accts`. You do not need to do sanity checking on how much money is credited or debited from Sally'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.)