Fall, 1998 . Name: _____

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 2.2-3, 3, 4.5–6

This test has 6 questions and pages numbered 1 through 8.

Reminders

For this test, you can use one (1) page (8.5 by 11 inches, one (1) side, no less than 8pt font) of notes. No photo-reduction is permitted. These notes are to be handed in at the end of the test. Use of other notes or failure to follow these instructions will be considered cheating. Please put your name in the top right corner of your notes.

During the test, if you need more space for an answer, use the back of a page. Note when you do that on the front.

This test is timed. We will not grade your test if you try to take more than the time allowed. Therefore, before you begin, please take a moment to look over the entire test so that you can budget your time.

For programs, indentation is important to us for "clarity" points; if your code is sloppy or hard to read, you will lose points. We will take off a small amount if you do not give `TYPE` comments for recursive helping procedures. However, you do not have to write such comments for procedures for which the type is stated in the problem. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

Subset of Scheme You May Use.

Unless otherwise stated in a problem, when solving problems you may only use standard features of the language that we discussed in class, `define-record`, `variant-case`, and helping functions that you define yourself. The standard is defined by the *Revised*^{4} *Report on the Algorithmic Language Scheme*.

Parts of Scheme You May *Not* Use.

Unless otherwise stated in a problem, you are prohibited from defining your own macros, and from using internal defines, all the input and output facilities, and the following keywords and procedures. (Don't worry if you don't know what these are.)

```
call-with-current-continuation  do
```

1. (10 points) In each of the spaces provided ("_____") below, write, in set brackets, the entire set of the bound variables in the corresponding Scheme expression. For example, write $\{v, w\}$, if the bound variables are v and w. If there are no bound variables, write $\{\}$. (You're supposed to know what a "bound variable" is.)

   (a)
   ```
   (let ((x (plus a b))
         (y (foo x y)))
     (lambda (z) (foo q y z)))
   ```
   _____

   (b)
   ```
   (letrec ((sing (lambda (ls)
                    (if (null? ls) n (f (car ls) (cdr ls)))))
            (f (let ((a ffty))
                 (lambda (x) (notify (plus x a))))))
     (sing notes))
   ```
   _____

2. (5 points) In the space provided ("_____") below, write, in set brackets, the entire set of the free variables in the corresponding Scheme expression. For example, write $\{v, w\}$, if the free variables are v and w. If there are no free variables, write $\{\}$. (You're supposed to know what a "free variable" is.)

   ```
   (letrec ((sing (lambda (ls)
                    (if (null? ls) n (f (car ls) (cdr ls)))))
            (f (let ((a ffty))
                 (lambda (x) (notify (plus x a))))))
     (sing notes))
   ```

   _____

3. (15 points) Write a Scheme procedure,

```
last-dot-position : (-> (string) number)
```

that takes a string, `fname`, and returns the 0-based index of the last occurrence of the dot (.) character in `fname`, if there is such an occurrence, and otherwise returns -1. (This is useful in finding the suffix of a file name.)

The procedure you have to write, `last-dot-position` must be written without converting the string to a list.

The following are examples:

```
(last-dot-position "abc.txt") ==> 3
(last-dot-position "01.3456.89") ==> 7
(last-dot-position "my.file.scm") ==> 7
(last-dot-position "...") ==> 2
(last-dot-position "no-suffix") ==> -1
(last-dot-position "") ==> -1
```

Note that the dot character can be written as `#\.` in Scheme. You should use the built-in Scheme procedures

```
string-ref : (-> (string number) character)
string-length : (-> (string) number)
```

in your solution. The `string-ref` procedure returns the character in its string argument at the 0-based index. For example, `(string-ref "abc" 2) ==> #\c`. The `string-length` procedure returns the number of characters in its argument.

4. This problem is about mutation.

   (a) (5 points) Draw a box and pointer diagram for the state after executing the following top-level Scheme forms.

```
(define programmers (cons 'lovelace '()))
(define pioneers (cons 'turing (cons 'church programmers)))
(set! programmers (cons 'knuth programmers))
(set-car! (cdr programmers) 'ada)
```

   (b) (5 points) Continuing from the above, what is the current value of the following expression?

```
(list 'programmers-is programmers
      'pioneers-is pioneers)
```

   (c) (5 points) Continuing from the above, what is the value of the following expression? (You can draw another diagram, but please leave the one for part (a) alone.)

```
(begin
  (set! pioneers (cdr pioneers))
  (list 'programmers-is programmers
        'pioneers-is pioneers))
```

5. (30 points) Write a procedure, `strength-reduce` of the following type

```
(-> (parsed-exp) parsed-exp)
```

that desugars `square` expressions into `multiply` expressions.

In this problem, the variant record type `parsed-exp`, is defined by the union of the following record types, which forms an abstract syntax.

```
(define-record varref (var))
(define-record lit (datum))
(define-record app (rator rands))
(define-record square (arg-exp))
(define-record multiply (left-exp right-exp))
```

The types of the fields in the `varref`, `lit`, and `app`, records are as in the `lambda-multiple` grammar; that is, the the type of the `rands` field is `(list parsed-exp)`.

Your procedure is use the following formula for desugaring, where `E` is an arbitrary parsed-exp.

```
#(square E) ==> #(multiply E E)
```

There are some examples on the next page.

The following are examples for the problem on the previous page.

```
(strength-reduce (make-square (make-varref 'x)))
  ==> #(multiply #(varref x) #(varref x))

(strength-reduce (make-varref 'x))
  ==> #(varref x)

(strength-reduce (make-app
                  (make-varref 'h)
                  (list (make-square (make-lit 5))
                        (make-varref 'a))))
  ==> #(app #(varref h) (#(multiply #(lit 5) #(lit 5)) #(varref a)))

(strength-reduce (make-app
                  (make-varref 'g)
                  (list (make-square (make-lit 3))
                        (make-square
                         (make-app
                          (make-varref 'f)
                          (list (make-varref 'y) (make-lit 4)))))))
  ==> #(app #(varref g) (#(multiply #(lit 3) #(lit 3))
                         #(multiply #(app #(varref f)
                                          (#(varref y) #(lit 4)))
                                    #(app #(varref f)
                                          (#(varref y) #(lit 4))))))
```

6. (25 points) This problem is about transforming procedural to record representations.

   One can imagine an "infinite" image (like a picture or a photograph) as a mapping from x and y coordinates to a color for the pixel at that coordinate. This type will be called `image` below. Consider the following procedural representation of `image`. (We will use symbols for colors.)

```
(define image-generator ; TYPE: (-> ((-> (number number) symbol)) image)
  (lambda (f)
    (lambda (x y)
      (f x y))))

(define move-x    ; TYPE: (-> (image number) image)
  (lambda (image dx)
    (lambda (x y)
      (pixel-at image (+ x dx) y))))

(define move-y    ; TYPE: (-> (image number) image)
  (lambda (image dy)
    (lambda (x y)
      (pixel-at image x (+ y dy)))))

(define pixel-at  ; TYPE: (-> (image number number) symbol)
  (lambda (image x y)
    (image x y)))
```

   Examples aren't really going to help you, but if we define

```
(define red-at-10s  ; TYPE: image
  (image-generator
   (let ((multiple-of-10?   ; TYPE: (-> (number number) boolean)
          (lambda (n) (zero? (remainder n 10)))))
     (lambda (x y)
       (if (or (multiple-of-10? x) (multiple-of-10? y))
           'red
           'blue)))))

(define red-shifted ; TYPE: image
  (move-y (move-x red-at-10s 5) 5))
```

   then the following are examples of how the above code works:

```
(pixel-at red-at-10s 0 0) ==> red
(pixel-at red-at-10s 5 3) ==> blue
(pixel-at red-at-10s 10 0) ==> red
(pixel-at red-shifted 0 0) ==> blue
(pixel-at red-shifted 5 5) ==> red
(pixel-at red-shifted 15 5) ==> red
(pixel-at red-shifted 10 10) ==> blue
```

Your task is to transform the above representation into one that uses records. Do this by giving the `define-record` declarations needed and the bodies of the procedures in the spaces provided on the next page. You must use `variant-case` in your solution.

```
;;; Write the define-record declarations below




;;; Now fill in the code for the operations below

(define image-generator ; TYPE: (-> ((-> (number number) symbol)) image)


(define move-x    ; TYPE: (-> (image number) image)


(define move-y    ; TYPE: (-> (image number) image)


(define pixel-at  ; TYPE: (-> (image number number) symbol)
```