Spring, 1996 Name: ____________________

. My Section Day and Time : ____________________

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 3, 4.5–6

This test has 7 questions and pages numbered 1 through 6.

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.

If you need more space, 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. (15 points) In each of the spaces provided ("__________") below, write, in set brackets, the entire set of the free variables in the preceding Scheme expression. For example, write $\{x, y\}$, if the free variables are x and y. If there are no free variables, write $\{\}$. (You're supposed to know what a "free variable" is.)

   (a)
   ```
   (let ((ls (cdr x))
         (tl (cdr ls)))
     (list (length ls) (length tl)))
   ```
   ____

   (b)
   ```
   (letrec ((o (lambda (n) (if (z? n) t (e (o (- n 1)))))))
     (o 3))
   ```
   ____

   (c)
   ```
   (let ((o (lambda (n) (if (z? n) t (e (o (- n 1)))))))
     (o 3))
   ```
   ____

2. (15 points) Desugar each of the following expressions. That is, rewrite them so that they do *not* use `cond`, `and`, `or`, and `let`, but so that they are equivalent in all contexts. (Assume that the free variables are globally defined.)

   (a)
```
(cond
  ((< x y) (foo 'less x))
  ((= x y) (bar 'equal x y))
  (else (baz 'greater x)))
```

   (b)
```
(and (not (null? ls))
     (or (equal? x (car ls))
         (recurse (cdr ls))))
```

   (c)
```
(let ((ls (cdr x))
      (tl (cdr ls)))
  (list (length ls) (length tl)))
```

3. (10 points) Briefly (a) describe when a program exhibits "representation independence", and (b) why it is important.

4. (5 points) Recall the Cell ADT from the textbook. Fill in the blanks in the following transcript.

```
> (define mycell (make-cell 5))
> (define c2 (make-cell 342))
> (cell-ref c2)

> (cell-set! mycell 4)
> (cell-ref mycell)

> (cell-swap! c2 mycell)
> (cell-ref c2)

> (cell-ref mycell)
```

5. (10 points) Consider the following concrete syntax, where ⟨number⟩ and ⟨symbol⟩ are as usual.

⟨cmd⟩ ::= `(begin` {⟨cmd⟩}*`)` | `(declare` ⟨var⟩`)` | `(assign` ⟨varref⟩ ⟨exp⟩`)` | ⟨call⟩
⟨call⟩ ::= `(`⟨exp⟩ {⟨exp⟩}*`)`

Define records to represent the abstract syntax of the nonterminal ⟨cmd⟩ in the above grammar. That is, write out the corresponding `define-record` declarations for each production. (Don't ask us what "abstract syntax" means, you're supposed to know that.)

6. (25 points) In this problem, we will use the following records to represent the type "nary-tree". That is, the type nary-tree is the union of the two record types below. (Note that this is different than in the text.)

   ```
   (define-record empty ())
   (define-record interior (number list-of-trees))
   ```

   The idea is that the `list-of-trees` field is a list of nary-trees.

   Write a procedure `nary-sub1`, that takes a nary-tree, `ntree`, and returns an nary-tree that is like `ntree`, but with each number in the tree decreased by 1. The following are examples.

   ```
   (nary-sub1 (make-empty)) = (make-empty)
   (nary-sub1 (make-interior 3 '())) = (make-interior 2 '())
   (nary-sub1 (make-interior 3 (list (make-empty) (make-empty))))
            = (make-interior 2 (list (make-empty) (make-empty)))
   (nary-sub1 (make-interior 2 (list (make-interior 3 '()) (make-empty))))
            = (make-interior 1 (list (make-interior 2 '()) (make-empty)))
   (nary-sub1 (make-interior
                11 (list (make-interior 4 (list (make-interior 5 '()))))))
            = (make-interior
                10 (list (make-interior 3 (list (make-interior 4 '())))))
   ```

   You *must* use `variant-case` in your solution.

7. (20 points) Consider the following procedural representation of infinite two-dimensional game boards. The type `(board T)` means a game board in which each position holds an object of type `T`.

```
(define fill-with ; TYPE: (-> (T) (board T))
  (lambda (contents)
    (lambda (i j) contents)))
(define add-at    ; TYPE: (-> ((board T) number number T) (board T))
  (lambda (old-board i j new-contents)
    (lambda (i2 j2)
      (if (and (= i2 i) (= j2 j))
          new-contents
          (value-at old-board i2 j2)))))
(define value-at   ; TYPE: (-> ((board T) number number) T)
  (lambda (board i j)
    (board i j)))
```

For example, if we define

```
(define wasteland (add-at (add-at (fill-with "empty") 34 2 "prize") 5 12 "pit"))
```

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

```
(value-at wasteland 7 -43925) ==> "empty"
(value-at wasteland 5 12) ==> "pit"
(value-at wasteland 34 2) ==> "prize"
```

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

```
;;; write the define-record declarations here



(define fill-with ; TYPE: (-> (T) (board T))
  (lambda (contents)



(define add-at    ; TYPE: (-> ((board T) number number T) (board T))
  (lambda (old-board i j new-contents)



(define value-at   ; TYPE: (-> ((board T) number number) T)
  (lambda (board i3 j3)
```