Fall, 1998 Name: ______________________

. My Section Time : ______________________

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 1 to 2.2

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

Reminders

This test is closed book and notes.

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 may 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, when defining procedures you may only use: helping procedures that you define yourself, comments, and the procedures and keywords that are included in the following list. (That is, on this test do *not* use Scheme procedures and keywords that are absent from the following list.) The notation `c...r` means `caar`, `cadr`, `cddr`, `cdar`, `caaar`, etc. You may only use `define` at the top level.

```
', #t, #f, *, +, -, /, <, <=, =, >=, >,
and, append, apply, boolean?, car, cdr,  c...r, char?,
cond, cons, define, display, else, eq?, equal?, eqv?, error,
if, let, letrec, lambda, list, length, list?, map, newline,
not, null?, number?, or, pair?, procedure?, quote, string,
string?, string=?, string-append, string-ci=?, string-length,
string-ref, string->list, string->number, string->symbol,
substring, symbol?, vector, vector?, vector-length,
vector->list, vector-ref, zero?
```

1. (10 points) Consider the following grammar.

   $\langle$clause$\rangle$ ::= $\langle$exp$\rangle$ `.` | $\langle$exp$\rangle$ `:-` $\langle$exp$\rangle$ `.`
   $\langle$exp$\rangle$ ::= $\langle$atom$\rangle$ | $\langle$name$\rangle$ `(` $\langle$atom$\rangle$ `{,` $\langle$atom$\rangle$`}`* `)`
   $\langle$atom$\rangle$ ::= $\langle$name$\rangle$ | $\langle$var$\rangle$
   $\langle$name$\rangle$ ::= `f` | `g` | `h` | `i` | `j`
   $\langle$var$\rangle$ ::= `X` | `Y` | `Z`

   In each of the spaces provided ("______") below, write "yes" if the text is an example of a $\langle$clause$\rangle$ in the above grammar, and "no" if it is not.

   (a) ______ `f`
   (b) ______ `f(X) :- X(h).`
   (c) ______ `f(i, j).`
   (d) ______ `f(X) :- g(Y, Z, i).`
   (e) ______ `f(X, i) :- g(Z, X), h(X).`

2. (5 points) Using Scheme, write an uncurried version of the following curried procedure. (Don't ask us what a "curried procedure" means, you're supposed to know that.)

```
(define s-combinator-c
  (lambda (x)
    (lambda (y)
      (lambda (z)
        ((x z) (y z))))))
```

3. (20 points) Write a procedure, `downcase-firsts`, with type

   ```
   (-> ((list (list string))) (list (list string)))
   ```

   that takes a list of non-empty lists of strings, and returns the same list but with the first string in each sublist changing its first letter to a lower case letter. To make this easier, use the following helping procedure, which changes the first letter of a string to a lower case letter.

   ```
   (define downcase-word  ;; TYPE: (-> (string) string)
     (lambda (word)
       (if (= (string-length word) 0)
           word
           (string-append (string (char-downcase (string-ref word 0)))
                          (substring word 1 (string-length word))))))
   ```

   The following are examples of the procedure you are to write.

   ```
   (downcase-firsts '())
                 ==> ()
   (downcase-firsts '(("It" "was" "the" "best")
                      ("Did" "we" "forget:" "of" "times?")))
                 ==> (("it" "was" "the" "best")
                      ("did" "we" "forget:" "of" "times?"))
   (downcase-firsts '(("Cain") ("Able") ("Adam") ("Eve")))
                 ==> (("cain") ("able") ("adam") ("eve"))
   (downcase-firsts '(("Computer" "science")))
                 ==> (("computer" "science"))
   ```

4. (10 points) Write a Scheme procedure `downcase-firsts*` with the type

   ```
   (-> ((list string) ...) (list (list string)))
   ```

   that takes 0 or more arguments that are non-empty lists of strings, and returns a list of these sublists with the first string in each sublist having its first letter changed to a lower case letter. You may use the `downcase-firsts` procedure from the previous problem above if you wish. The following are examples.

   ```
   (downcase-firsts* )
                 ==> ()
   (downcase-firsts* '("It" "was" "the" "best")
                     '("Did" "we" "forget:" "of" "times?"))
                 ==> (("it" "was" "the" "best")
                      ("did" "we" "forget:" "of" "times?"))
   (downcase-firsts* '("Cain") '("Able") '("Adam") '("Eve"))
                 ==> (("cain") ("able") ("adam") ("eve"))
   (downcase-firsts* '("Computer" "science"))
                 ==> (("computer" "science"))
   ```

5. (20 points) Write a procedure `preceed-each` with the following type.

```
(-> (T (list T)) (list T))
```

that takes an element, `pre`, and a list, `ls`, of the same type, and returns a list that is like `ls`, but with each element of `ls` preceeded by `pre`. The following are examples.

```
(preceed-each 'x '())
              ==> ()
(preceed-each 'x '(a b c))
              ==> (x a x b x c)
(preceed-each 'uh '(a fine time for this eh))
              ==> (uh a uh fine uh time uh for uh this uh eh)
```

6. (30 points) Consider the following grammar.

   ⟨line⟩ ::= (⟨third⟩ ⟨third⟩ ⟨third⟩)
   ⟨third⟩ ::= ⟨symbol⟩ | ⟨line⟩

   In this grammar, the nonterminal ⟨symbol⟩ has the same syntax as in Scheme. We will always use this grammar with the context-sensitive constraint that the each ⟨third⟩ of a ⟨line⟩ has the same shape; for example, they will either all be symbols or not. Write a procedure, `change-middle-right`, with the following type

```
  (-> (symbol line) line)
```

   that takes a `symbol`, `sym`, and a ⟨line⟩, `ln`, and returns a line that is like `ln`, except that the middle and rightmost symbols in each middle and rightmost line are changed to `sym`.

   Assume that all the ⟨third⟩s in a ⟨line⟩ have the same shape; that is, if the first third is a symbol, then all are, etc.

   The following are examples.

```
(change-middle-right '* '(a b c))
                     ==> (a * *)
(change-middle-right '* '((a b c) (d e f) (g h i)))
                     ==> ((a b c) (d * *) (g * *))
(change-middle-right '* '(((a b c) (d e f) (g h i))
                          ((j k l) (m n o) (p q r))
                          ((s t u) (v w x) (y z a1))))
                     ==> (((a b c) (d e f) (g h i))
                          ((j k l) (m * *) (p * *))
                          ((s t u) (v * *) (y * *)))
(change-middle-right '+ '((((a0 b0 c0) (d0 e0 f0) (g0 h0 i0))
                           ((j0 k0 l0) (m0 n0 o0) (p0 q0 r0))
                           ((s0 t0 u0) (v0 w0 x0) (y0 z0 a1)))
                          (((a1 b1 c1) (d1 e1 f1) (g1 h1 i1))
                           ((j1 k1 l1) (m1 n1 o1) (p1 q1 r1))
                           ((s1 t1 u1) (v1 w1 x1) (y1 z1 a2)))
                          (((a2 b2 c2) (d2 e2 f2) (g2 h2 i2))
                           ((j2 k2 l2) (m2 n2 o2) (p2 q2 r2))
                           ((s2 t2 u2) (v2 w2 x2) (y2 z2 a3)))))
                     ==> ((((a0 b0 c0) (d0 e0 f0) (g0 h0 i0))
                           ((j0 k0 l0) (m0 n0 o0) (p0 q0 r0))
                           ((s0 t0 u0) (v0 w0 x0) (y0 z0 a1)))
                          (((a1 b1 c1) (d1 e1 f1) (g1 h1 i1))
                           ((j1 k1 l1) (m1 + +) (p1 + +))
                           ((s1 t1 u1) (v1 + +) (y1 + +)))
                          (((a2 b2 c2) (d2 e2 f2) (g2 h2 i2))
                           ((j2 k2 l2) (m2 + +) (p2 + +))
                           ((s2 t2 u2) (v2 + +) (y2 + +))))
```

   Hint: don't hesitate to write helping procedures. There is more space on the next page.

(space for answer to the problem on the previous page)