Fall 2006 Name: ______________________
Com S 342

Principles of Programming Languages

Exam 2 on Grammars and Recursion over Inductively-Specified Data

This test has 8 questions and pages numbered 1 through 12.

Special Instructions for this Test

Your code must properly use the appropriate helping procedures for each grammar. Do not use the parsing procedures, those named `parse-`..., in your solutions.

Reminders

For this test, you can use one (1) page (8.5 by 11 inches, one (1) side, no less than 9pt font) of notes. Handwriting is okay. No photo-reduction is permitted. Don't use anything with printing on the other side, please. These notes are to be handed in at the end of the test. Have your name in the top right corner. 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. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

You can use helping procedures whenever you like.

For Grading

Problem	Points	Score
1	5	
2	5	
3	10	
4	10	
5	15	
6	30	
7	15	
8	10	

1. (5 points) Write a curried version of the following procedure. (You don't have to write its type.)

```
(deftype force (-> (number number) number))
(define force
  (lambda (mass accel)
    (* mass accel)))
```

2. (5 points) Consider the Scheme expression:

```
(and (not (null? lst))
     (not (null? (cdr lst)))
     (> (car lst) 5)
     (= (cadr lst) 7))
```

Briefly describe the advantages of using `and` instead of the desugared equivalent form of the above expression (with `if`).

3. (10 points) Consider the following grammar.

$\langle$value$\rangle$ ::= $\langle$symbol$\rangle$
  | $\langle$number$\rangle$
  | $\langle$procedure$\rangle$
$\langle$procedure$\rangle$ ::= `proc ( $` $\langle$formals$\rangle$ `)` $\langle$value$\rangle$ `end`
$\langle$formals$\rangle$ ::= $\{\langle$symbol$\rangle\}^*$

where $\langle$symbol$\rangle$ stands for a Scheme symbol, such as `X`, $\langle$number$\rangle$ stands for a Scheme number, such as `72`, and in which `proc`, `end`, `$`, `(`, and `)` are terminals. Now consider the following input.

```
proc ( $ X Y ) X end
```

Either show how to derive the above string from the nonterminal $\langle$value$\rangle$, using the given grammar, or briefly explain why no derivation is possible.

Please show all steps, and don't replace more than one nonterminal in a step.

4. (10 points) Write a Scheme procedure,

```
itunes : (-> ((list-of (list-of symbol))) (list-of number))
```

such that `(itunes songs)` returns a list with the same length as `songs`, except that each element in `songs`, which is a list of symbols, is replaced with its price, which is 99 (cents). The following are examples.

```
(itunes '()) ==> ()
(itunes '((how to save a life) (smack that) (chasing cars)))
     ==> (99 99 99)
(itunes '((smack that) (chasing cars)))
     ==> (99 99)
(itunes '((chasing cars)))
     ==> (99)
(itunes '((white and nerdy) (lips of an angel) (chain hang low)
           (money maker) (sexyback) (call me when youre sober)))
     ==> (99 99 99 99 99 99)
```

5. (15 points) This is a problem about the homework's "window layout" grammar. As in the homework, the comments on the right are an aid to remembering the helping procedures.

```
⟨window-layout⟩ ::=
      (window ⟨symbol⟩ ⟨number⟩ ⟨number⟩)       "window (name width height)"
    | (horizontal {⟨window-layout⟩}*)           "horizontal (subwindows)"
    | (vertical {⟨window-layout⟩}*)             "vertical (subwindows)"
```

The following are the types of the helping procedures, from the library file `window-layout-mod.scm`.

```
window? : (-> (window-layout) boolean)
horizontal? : (-> (window-layout) boolean)
vertical? : (-> (window-layout) boolean)

window : (-> (symbol number number) window-layout)
horizontal : (-> ((list-of window-layout)) window-layout)
vertical : (-> ((list-of window-layout)) window-layout)

window->name : (-> (window-layout) symbol)
window->width : (-> (window-layout) number)
window->height : (-> (window-layout) number)
horizontal->subwindows : (-> (window-layout) (list-of window-layout))
vertical->subwindows : (-> (window-layout) (list-of window-layout))
```

Using the above helping procedures, write a Scheme procedure,

```
scale-layout : (-> (number window-layout) window-layout)
```

such that `(scale-layout factor wl)` returns a window layout that is just like `wl`, except that each window in `wl` has its height and width multiplied by `factor`. You can assume that the argument `factor` is non-negative. The following are examples that equate Scheme expressions.

```
(scale-layout 2 (window 'pbs 30 40))
            = (window 'pbs 60 80)
(scale-layout 1.5 (window 'orilley 1250 2000))
            = (window 'orilley 1875.0 3000.0)
(scale-layout 2 (horizontal
                 (list (window 'orilley 1250 2000) (window 'pbs 30 40))))
            = (horizontal
                (list (window 'orilley 2500 4000) (window 'pbs 60 80)))
(scale-layout 0.5
              (vertical
                (list (window 'prime 300 100) (window 'composite 500 600))))
            = (vertical
                (list (window 'prime 150.0 50.0) (window 'composite 250.0 300.0)))
```

```
 (scale-layout 0.4
              (vertical
                (list (vertical (list (window 'word 300 100)
                                      (window 'excel 500 600)
                                      (horizontal (list ))))
                      (horizontal (list (window 'aim 20 20)
                                        (window 'icq 20 20))))))
             = (vertical
                (list (vertical (list (window 'word 120.0 40.0)
                                      (window 'excel 200.0 240.0)
                                      (horizontal (list ))))
                      (horizontal (list (window 'aim 8.0 8.0)
                                        (window 'icq 8.0 8.0)))))

;;; Please write your answer below.

(require (lib "window-layout-mod.scm" "lib342"))
```

6. (30 points) This is a problem about the homework's statement and expression grammar.

⟨statement⟩ ::=	
⟨expression⟩	"`exp-stmt (exp)`"
| `(set!` ⟨identifier⟩ ⟨expression⟩`)`	"`set-stmt (id exp)`"
⟨expression⟩ ::=	
⟨identifier⟩	"`var-exp (id)`"
| ⟨number⟩	"`num-exp (num)`"
| `(begin` {⟨statement⟩}* ⟨expression⟩`)`	"`begin-exp (stmts exp)`"

In the above grammar the nonterminal ⟨identifier⟩ has the same syntax as a Scheme ⟨symbol⟩.

The following are the types of the helping procedures for the statement and expression grammar, from the library file `statement-expression.scm`.

```
exp-stmt? : (-> (statement) boolean)
set-stmt? : (-> (statement) boolean)
var-exp? : (-> (expression) boolean)
num-exp? : (-> (expression) boolean)
begin-exp? : (-> (expression) boolean)

exp-stmt : (-> (expression) statement)
set-stmt : (-> (symbol expression) statement)
var-exp : (-> (symbol) expression)
num-exp : (-> (number) expression)
begin-exp : (-> ((list-of statement) expression) expression)

exp-stmt->exp : (-> (statement) expression)
set-stmt->id : (-> (statement) symbol)
set-stmt->exp : (-> (statement) expression)
var-exp->id : (-> (expression) symbol)
num-exp->num : (-> (expression) number)
begin-exp->stmts : (-> (expression) (list-of statement))
begin-exp->exp : (-> (expression) expression)
```

Write a Scheme procedure,

```
inline-var : (-> (symbol number statement) statement)
```

such that `(inline-var name val statement)` that takes a symbol `name`, a number `val`, and a ⟨statement⟩, `stmt`, and returns a ⟨statement⟩ that is the same as `stmt` except that each expression of the form `(var-exp name)` is replaced by `val` an expression of the form `(num-exp val)`.

Your answer must properly use the helpers for the above grammar, whose types are given above.

The following are examples.

```
(inline-var 'plclass 541 (exp-stmt (var-exp 'plclass)))
   = (exp-stmt (num-exp 541))
(inline-var 'plclass 541 (exp-stmt (num-exp 342)))
   = (exp-stmt (num-exp 342))
(inline-var 'tft 342 (set-stmt 'tft (var-exp 'tft)))
   = (set-stmt 'tft (num-exp 342))
(inline-var 'zero 0 (exp-stmt (var-exp 'x)))
   = (exp-stmt (var-exp 'x))
(inline-var  'zero 0 (set-stmt 'val (begin-exp '() (var-exp 'val))))
   = (set-stmt 'val (begin-exp '() (var-exp 'val)))
(inline-var
 'zero 0
 (set-stmt 'val
           (begin-exp (list (set-stmt 'x (var-exp 'zero))) (var-exp 'x))))
   = (set-stmt 'val (begin-exp (list (set-stmt 'x (num-exp 0))) (var-exp 'x)))
(inline-var
   'zero 0
   (exp-stmt (begin-exp (list (set-stmt 'val (var-exp 'zero))
                              (set-stmt 'x (var-exp 'zero)))
                        (var-exp 'zero))))
 = (exp-stmt (begin-exp (list (set-stmt 'val (num-exp 0))
                              (set-stmt 'x (num-exp 0)))
                        (num-exp 0)))
(inline-var
   'zero 0
   (set-stmt 'z (begin-exp
                 (list
                  (exp-stmt (begin-exp (list (set-stmt 'val (var-exp 'zero))
                                             (set-stmt 'x (var-exp 'zero)))
                                       (var-exp 'zero))))
                 (begin-exp (list (set-stmt 'zero (var-exp 'zero)))
                            (num-exp 25)))))
 = (set-stmt 'z (begin-exp
                 (list
                  (exp-stmt (begin-exp (list (set-stmt 'val (num-exp 0))
                                             (set-stmt 'x (num-exp 0)))
                                       (num-exp 0))))
                 (begin-exp (list (set-stmt 'zero (num-exp 0)))
                            (num-exp 25))))
```

There is space for your answer on the next page.

```
;;; Please write your answer below.

(require (lib "statement-expression.scm" "lib342"))
```

7. (15 points) Consider the following grammar, for simple arithmetic and relational expressions; again this has comments on the right side enclosed in quotation marks as an aid to remembering the helping procedures.

⟨ntree⟩ ::=	
⟨number⟩	"`leaf (value)`"
| (⟨number⟩ ({⟨ntree⟩}*))	"`branch (value trees)`"

The types of the helping procedures for this grammar are as follows.

```
leaf? : (-> (ntree) boolean)
branch? : (-> (ntree) boolean)
leaf : (-> (number) ntree)
branch : (-> (number (list-of ntree)) ntree)
leaf->value : (-> (ntree) number)
branch->value : (-> (ntree) number)
branch->trees : (-> (ntree) (list-of ntree))
```

Using the above helping procedures, write a procedure,

```
sum-ntree : (-> (ntree) number)
```

such that `(sum-ntree ntr)` is the sum of all the numeric values in `ntr`.

The following are examples.

```
(sum-ntree (leaf 3)) ==> 3
(sum-ntree (leaf 7)) ==> 7
(sum-ntree (branch 22 (list))) ==> 22
(sum-ntree (branch 0 (list (leaf 7) (leaf 3)))) ==> 10
(sum-ntree (branch 0 (list (leaf 6) (leaf 7) (leaf 3)))) ==> 16
(sum-ntree (branch 200 (list (leaf 6) (leaf 7) (leaf 3)))) ==> 216
(sum-ntree (branch 10 (list (branch 20 (list))
                            (branch 5 (list))))) ==> 35
(sum-ntree
     (branch 0
             (list (branch 1 (list))
             (branch 3 (list (leaf 6) (leaf 7) (leaf 3)))))) ==> 20
(sum-ntree
     (branch 5
             (list
               (branch 1
                       (list (branch 1
                                     (list (branch 1 (list (leaf 1)))))
                             (leaf 10)))
               (branch 3 (list (leaf 6) (leaf 7) (leaf 3)))))) ==> 38
```

```
;;; Please write your answer below.

(require "ntree-mod.scm") ;; loads the helpers for this problem
```

8. (10 points) Without using `vector->list` or `list->vector`, write a procedure,

```
vector->list : (-> ((vector-of number)) (list-of number))
```

such that `(vector->list von)` returns a list with the same elements in the same order as von. The following are examples.

```
(vector->list (vector 3 4 2)) ==> (3 4 2)
(vector->list (vector 6 4 1 3 2)) ==> (6 4 1 3 2)
(vector->list (vector 8 1 2 3 3 2 1 89)) ==> (8 1 2 3 3 2 1 89)
(vector->list (vector )) ==> ()
```

Hints: Remember that Scheme vectors have indexes that start with zero (0). You can use

```
vector-length : (forall (T) (-> ((vector-of T)) number))
vector-ref    : (forall (T) (-> ((vector-of T) number) T))
```

to get the length and to access an element of a vector, respectively.