Spring 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.

```
(deftype extend
  (-> ((vector-of number) (list-of number)) (vector-of number)))
(define extend
  (lambda (von lon)
    (list->vector (append (vector->list von) lon))))
```

2. (5 points) Name a syntactic form in either C++ or Java that is syntactic sugar, and briefly describe what it desugars into. Be sure to say what language you are describing in your answer.

3. (10 points) Consider the following grammar.

   $\langle$rule$\rangle$ ::= $\langle$atom$\rangle$ `.`
   $\quad$ | $\langle$atom$\rangle$ `:-` $\langle$atom-list$\rangle$ `.`
   $\langle$atom-list$\rangle$ ::= $\langle$atom$\rangle$
   $\quad$ | $\langle$atom-list$\rangle$ `,` $\langle$atom$\rangle$
   $\langle$atom$\rangle$ ::= $\langle$symbol$\rangle$

   where $\langle$symbol$\rangle$ stands for a Scheme symbol, such as `x`, and in which the `:-`, the `.` and the `,` are terminals. Now consider the following input.

   ```
   good :- logical , sensible .
   ```

   Either show how to derive the above string from the nonterminal $\langle$rule$\rangle$, 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,

```
slalom : (-> ((list-of symbol)) (list-of symbol))
```

such that `(slalom gates)` returns a list with the same length as `gates`, except that each element in `gates` that is the symbol `red` is turned into the symbol `right`, and each element that is the symbol `blue` is turned into the symbol `left`. The following are examples.

```
(slalom '()) ==> ()
(slalom '(blue red red))
     ==> (left right right)
(slalom '(red orange blue red red))
     ==> (right orange left right right)
(slalom '(blue blue red red red blue red red pink))
     ==> (left left right right right left right right pink)
```

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,

```
stretch-width : (-> (window-layout number) window-layout)
```

such that `(stretch-width wl amt)` returns a window layout that is just like `wl`, except that each window in `wl` has `amt` added to its width. You can assume that the argument `amt` is non-negative. The following are examples that equate Scheme expressions.

```
(stretch-width (window 'pbs 30 40) 5)
             = (window 'pbs 35 40)
(stretch-width (window 'pella 1250 2000) 33)
             = (window 'pella 1283 2000)
(stretch-width (horizontal
                 (list (window 'pella 1250 2000) (window 'pbs 30 40)))
               5)
             = (horizontal
                 (list (window 'pella 1255 2000) (window 'pbs 35 40)))
(stretch-width (vertical
                 (list (window 'word 300 100) (window 'excel 500 600)))
               8)
             = (vertical
                 (list (window 'word 308 100) (window 'excel 508 600)))
```

```
 (stretch-width (vertical
                 (list (vertical (list (window 'word 300 100)
                                       (window 'excel 500 600)
                                       (horizontal (list ))))
                       (horizontal (list (window 'aim 20 20)
                                         (window 'icq 20 20)))))
               8)
             = (vertical
                 (list (vertical (list (window 'word 308 100)
                                       (window 'excel 508 600)
                                       (horizontal (list ))))
                       (horizontal (list (window 'aim 28 20)
                                         (window 'icq 28 20)))))

;;; 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,

```
replace-constant : (-> (statement symbol number) statement)
```

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

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

The following are examples.

```
(replace-constant (exp-stmt (num-exp 541)) 'plclass 541)
   = (exp-stmt (var-exp 'plclass))
(replace-constant (exp-stmt (num-exp 342)) 'plclass 541)
   = (exp-stmt (num-exp 342))
(replace-constant (set-stmt 'i (num-exp 342)) 'tft 342)
   = (set-stmt 'i (var-exp 'tft))
(replace-constant (set-stmt 'val (begin-exp '() (var-exp 'val))) 'zero 0)
   = (set-stmt 'val (begin-exp '() (var-exp 'val)))
(replace-constant
 (set-stmt 'val (begin-exp (list (set-stmt 'x (num-exp 0))) (var-exp 'x)))
 'zero 0)
   = (set-stmt 'val
               (begin-exp (list (set-stmt 'x (var-exp 'zero))) (var-exp 'x)))
(replace-constant (exp-stmt (var-exp 'x)) 'zero 0)
   = (exp-stmt (var-exp 'x))
(replace-constant
   (exp-stmt (begin-exp (list (set-stmt 'val (num-exp 0))
                              (set-stmt 'x (num-exp 0)))
                        (num-exp 0)))
   'zero 0)
 = (exp-stmt (begin-exp (list (set-stmt 'val (var-exp 'zero))
                              (set-stmt 'x (var-exp 'zero)))
                        (var-exp 'zero)))
(replace-constant
   (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))))
   '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))))
```

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 expressions; again this has comments on the right side enclosed in quotation marks as an aid to remembering the helping procedures.

⟨expression⟩ ::=	
⟨number⟩	"`lit-exp (datum)`"
| (⟨expression⟩ ⟨op⟩ ⟨expression⟩)	"`op-exp (left-arg op right-arg)`"
⟨op⟩ ::=	
`+`	"`plus-op ()`"
| `-`	"`minus-op ()`"
| `*`	"`times-op ()`"

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

```
lit-exp? : (-> (expression) boolean)
op-exp? : (-> (expression) boolean)
plus-op? : (-> (op) boolean)
minus-op? : (-> (op) boolean)
times-op? : (-> (op) boolean)
lit-exp : (-> (number) expression)
op-exp : (-> (expression op expression) expression)
plus-op : (-> () op)
minus-op : (-> () op)
times-op : (-> () op)
lit-exp->datum : (-> (expression) number)
op-exp->left-arg : (-> (expression) expression)
op-exp->op : (-> (expression) op)
op-exp->right-arg : (-> (expression) expression)
```

Just to be clear about how these work, note that for all expressions E_1, E_2 and ops O:

`(op-exp->left-arg (op-exp` E_1 O E_2`))` $= E_1$
`(op-exp->right-arg (op-exp` E_1 O E_2`))` $= E_2$

Using the above helping procedures, and without using Scheme's built in `eval` procedure, write a procedure,

```
interpret : (-> (expression) number)
```

such that `(interpret exp)` returns the arithmetic value of `exp`.

The following are examples.

```
(interpret (lit-exp 342)) ==> 342
(interpret (lit-exp 0)) ==> 0
(interpret (op-exp (lit-exp 5) (minus-op) (lit-exp 3))) ==> 2
(interpret (op-exp (lit-exp 3) (plus-op) (lit-exp 5))) ==> 8
(interpret (op-exp (lit-exp 2) (times-op) (lit-exp 7))) ==> 14
```

```
  (interpret (op-exp (lit-exp 2)
                     (plus-op)
                     (op-exp (lit-exp 3) (times-op) (lit-exp 7))))
    ==> 23
  (interpret (op-exp (lit-exp 100)
                     (minus-op)
                     (op-exp (lit-exp 3) (plus-op) (lit-exp 7))))
    ==> 90
  (interpret
   (op-exp (op-exp (lit-exp 2) (times-op) (lit-exp 9))
           (plus-op)
           (op-exp (op-exp (lit-exp 3) (times-op) (lit-exp 7))
                   (plus-op)
                   (lit-exp 0))))
    ==> 39

;;; Please write your answer below.

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

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

```
all-zero? : (-> ((vector-of number)) boolean)
```

such that `(all-zero?  von)` returns `#t` if, for all indexes, $0 \leq i < len$, where *len* is the length of `von`, each element i of `von` is 0. The following are examples.

```
(all-zero? (vector 3 4 2)) ==> #f
(all-zero? (vector 0 0 0)) ==> #t
(all-zero? (vector 0 0 0 0 0 1 0 0 0 0)) ==> #f
(all-zero? (vector 0 0 0 0 0 0 0 0 0 0)) ==> #t
(all-zero? (vector )) ==> #t
```

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.