Spring 2004 Name: ______________________
Com S 342 Section: ______________________

Principles of Programming Languages

Makeup for Exam 2 on Recursion over Grammars

This test has 4 questions and pages numbered 1 through 9.

Special Instructions for this Test

Your code should type check for full credit. You may not use `trustme!` or `has-type-trusted` in your code. You are not permitted to use the helping procedures named `parse-...` in your code. Do not use `set!`.

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. If you write recursive helping procedures, please give a deftype declaration for them.

For Grading

Problem	Points
1	
2	
3	
4	

1. (25 points) This is a problem about the ⟨sym-exp⟩ grammar below.

   ⟨sym-exp⟩ ::= ⟨symbol⟩ | ⟨s-list⟩
   ⟨s-list⟩ ::= `(` {⟨sym-exp⟩}* `)`

   For this grammar you can use the following helping procedures.

```
symbol->sym-exp : (-> (symbol) sym-exp)
s-list->sym-exp : (-> ((list-of sym-exp)) sym-exp)
sym-exp-symbol? : (-> (sym-exp) boolean)
sym-exp-s-list? : (-> (sym-exp) boolean)
sym-exp->symbol : (-> (sym-exp) symbol)
sym-exp->s-list : (-> (sym-exp) (list-of sym-exp))
```

   Write a Scheme procedure,
   `deepen : (-> (sym-exp) sym-exp)`
   that takes a ⟨sym-exp⟩, `symexp`, and returns a ⟨sym-exp⟩ that is just like `symexp`, except that every each place where a symbol, say *x*, occurs in `symexp`, it is replaced by `(`*x*`)`. Your answer must use the helpers for the ⟨sym-exp⟩ grammar, and must type check for full credit. If you write a helping procedure, be sure to give a `deftype` for it. Here are some examples:

```
(deepen (parse-sym-exp 'x)) ==> (x)
(deepen (parse-sym-exp 'y)) ==> (y)
(deepen (parse-sym-exp '())) ==> ()
(deepen (parse-sym-exp '(x + (y + z)))) ==> ((x) (+) ((y) (+) (z)))
(deepen (parse-sym-exp '(x () ((y)) () ((((q)))) z a b)))
                    ==> ((x) () (((y))) () (((((q))))) (z) (a) (b))
(deepen (parse-sym-exp '(olive (((((((() () x + y () y ()))))))) y))))
           ==> ((olive) (((((((() () (x) (+) (y) () (y) ()))))))) (y)))
```

2. (20 points) Consider the following grammar, with comments on the right side enclosed in quotations (" and "). The comments are an aid to remembering the helping procedures, whose types are shown on the next page.

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

where the nonterminals ⟨number⟩ and ⟨symbol⟩ have the same syntax as in Scheme.

Using the helpers whose types are shown on the next page, write a procedure,

```
total-area : (-> (window-layout) number)
```

that takes a ⟨window-layout⟩, `wl`, and returns the area represented by that layout. There are examples on the next page. Please write your answer below.

The following are for the problem on the previous page.

The helping procedures for the `window-layout` type are as follows.

```
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))
```

The following are examples of the problem.

```
(total-area (window 'a 5 10)) ==> 50
(total-area (window 'b 3 4)) ==> 12
(total-area (horizontal '())) ==> 0
(total-area (horizontal (list (window 'a 5 10) (window 'b 3 4)))) ==> 62
(total-area (vertical '())) ==> 0
(total-area (vertical (list (window 'a 1 2) (window 'b 8 8)))) ==> 66
(total-area (horizontal (list (window 'pane 3 4) (vertical '())))) ==> 12
(total-area (vertical (list (window 'title 10 10)
                            (horizontal
                             (list (window 'a 5 10)
                                   (window 'b 3 4)))))) ==> 162
(total-area (horizontal
             (list (window 'title 2 100)
                   (vertical (list (window 'panel-a 10 10)
                                   (window 'panel-b 20 20)))
                   (horizontal (list (window 'left 3 50)
                                     (window 'right 3 50)))
                   (vertical
                    (list
                     (vertical
                      (list
                       (horizontal
                        (list
                         (vertical
                          (list
                           (vertical '())
                           (window 'deep 5 10)))))))))))) ==> 1050
```

3. (20 points) Consider the following grammar, with comments on the right side enclosed in quotations (“ and ”). The comments are an aid to remembering the helping procedures, whose types are shown on the next page.

⟨fraction-expression⟩ ::= ⟨symbol⟩	“var-exp (id)”
| ⟨number⟩	“num-exp (num)”
| `(/` ⟨fraction-expression⟩ ⟨fraction-expression⟩`)`	“div-exp (numer denom)”

where the nonterminals ⟨number⟩ and ⟨symbol⟩ have the same syntax as in Scheme.

Using the helpers whose types are shown on the next page, write a procedure,

```
eval-fe : (-> ((-> (symbol) number) fraction-expression) number)
```

that takes a mapping from symbols to numbers, `env`, and a ⟨fraction-expression⟩, `fe`, and returns the value of `fe`, with each identifier, x, that occurs in `fe` replaced by the result of applying `env` to x. Assume that every symbol that appears in `fe` is given a value by `env`. For example, if the symbol `foo` occurs in `fe`, then you should assume that `(env 'foo)` is defined. There are examples on the next page.

Please write your answer below. You may not use Scheme's `eval` procedure in your answer.

The following are for the problem on the previous page.

The helping procedures for the `fraction-expression` type are as follows.

```
var-exp?       : (-> (fraction-expression) boolean)
num-exp?       : (-> (fraction-expression) boolean)
div-exp?       : (-> (fraction-expression) boolean)
var-exp        : (-> (symbol) fraction-expression)
num-exp        : (-> (number) fraction-expression)
div-exp        : (-> (fraction-expression fraction-expression) fraction-expression)
var-exp->id    : (-> (fraction-expression) symbol)
num-exp->num   : (-> (fraction-expression) number)
div-exp->numer : (-> (fraction-expression) fraction-expression)
div-exp->denom : (-> (fraction-expression) fraction-expression)
parse-fraction-expression : (-> (datum) fraction-expression)
```

The following are a few very simple examples of the problem.

```
(eval-fe (lambda (x) 999) (var-exp 'x)) ==> 999
(eval-fe (lambda (y) 342) (num-exp 4)) ==> 4
(eval-fe (lambda (z) 342) (div-exp (num-exp 1) (var-exp 'z)) ==> 1/342
```

The following examples use the procedure `env1`, defined below.

```
(deftype env1 (-> (symbol) number))
(define env1
  (lambda (sym)
    (cond
     ((eq? sym 'one) 1)
     ((eq? sym 'three) 3)
     ((eq? sym 'five) 5)
     ((eq? sym 'ten) 10)
     (else (error "unknown symbol: " sym)))))

(eval-fe env1 (div-exp (var-exp 'one) (num-exp 4))) ==> 1/4
(eval-fe env1 (parse-fraction-expression '(/ five 6))) ==> 5/6
(eval-fe env1 (parse-fraction-expression '(/ (/ five ten) ten))) ==> 1/20
(eval-fe env1 (parse-fraction-expression '(/ one (/ ten 2)))) ==> 1/5
(eval-fe env1 (parse-fraction-expression '(/ (/ ten 2) (/ ten 2)))) ==> 1
(eval-fe env1
         (parse-fraction-expression
          '(/ (/ ten three) (/ ten (/ 1 five))))) ==> 1/15
(eval-fe env1
         (parse-fraction-expression
          '(/ 2 (/ 1 (/ (/ 1000 (/ ten three)) (/ ten (/ 1 five))))))) ==> 12
```

4. (35 points) Consider the following grammar, with comments on the right side enclosed in quotations (" and "). The comments are an aid to remembering the helping procedures, which are shown below.

```
⟨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)"
```

where an ⟨identifier⟩ has the same syntax as a Scheme ⟨symbol⟩.

The following are the types of the helping procedures for this grammar.

```
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)
parse-statement   : (-> (datum) statement)
parse-expression  : (-> (datum) expression)
```

Using these helpers, write a procedure,
`replace-unbound :  (-> ((list-of symbol) statement) statement)`
that takes a list of symbols, `bound`, and a statement, `stmt`, and returns a statment that is just like `stmt`, except that every place where an ⟨identifier⟩ that is not in the list `bound` occurs, that identifier is replaced by the symbol `unbound!`.

Hint: use `memq : (forall (T) (-> (T (list-of T)) boolean))`, which is a built-in procedure in Scheme, to test if a symbol occurs in a list of symbols.

Examples are on the next page. Space for your answer is on the page after that.

The following are examples for the problem on the previous page. Note that each example is an equation; in each equation, both sides are expressions in Scheme that evaluate to a value of type `statement`.

```
(replace-unbound '(x y z)
                 (exp-stmt (var-exp 'y)))
               = (exp-stmt (var-exp 'y))
(replace-unbound '(x y z)
                 (exp-stmt (var-exp 'a)))
               = (exp-stmt (var-exp 'unbound!))
(replace-unbound '(a b c)
                 (exp-stmt (var-exp 'a)))
               = (exp-stmt (var-exp 'a))
(replace-unbound '(a b c)
                 (exp-stmt (num-exp 3)))
               = (exp-stmt (num-exp 3))
(replace-unbound '(a b c)
                 (exp-stmt (begin-exp (list (set-stmt 'x (var-exp 'a)))
                                      (var-exp 'x))))
               = (exp-stmt (begin-exp (list (set-stmt 'unbound! (var-exp 'a)))
                                      (var-exp 'unbound!)))
(replace-unbound '(x y z)
                 (exp-stmt (begin-exp (list (set-stmt 'x (var-exp 'a)))
                                      (var-exp 'x))))
               = (exp-stmt (begin-exp (list (set-stmt 'x (var-exp 'unbound!)))
                                      (var-exp 'x)))
(replace-unbound '(x y z)
                 (set-stmt 'x (begin-exp (list (set-stmt 'x (var-exp 'a)))
                                         (var-exp 'x))))
               = (set-stmt 'x (begin-exp (list (set-stmt 'x (var-exp 'unbound!)))
                                         (var-exp 'x)))
(replace-unbound
   '(a b c)
   (parse-statement '(set! x (begin (set! x a) x))))
 = (parse-statement '(set! unbound! (begin (set! unbound! a) unbound!)))
(replace-unbound
   '(a b c)
   (parse-statement
    '(set! a (begin (begin (set! b (begin (set! x a) x)) b)))))
 = (parse-statement
    '(set! a (begin (begin (set! b (begin (set! unbound! a) unbound!)) b))))
(replace-unbound
   '(a b c)
   (parse-statement '(set! a (begin x))))
 = (parse-statement '(set! a (begin unbound!)))
```

Fill in your answer below.