Spring 2005
Com S 342

Name: _____
Section: _____

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	10	
3	10	
4	5	
5	15	
6	30	
7	15	
8	10	

1. (5 points) Write a curried version of the following procedure.

```
(deftype apologize (-> (string string) string))
(define apologize
  (lambda (name deed)
    (string-append name
                   ", we're sorry that we"
                   deed
                   ".  It won't happen again.")))
```

2. (10 points) Write "yes" underneath each of the following that is a true statements about syntactic sugars (syntax abstractions). Write "no" underneath each of the following that is false.

   (a) Syntactic sugars make programming languages more difficult to use.

   (b) The `for` loop in C++ and Java is a syntactic sugar for a `while` loop.

   (c) Scheme's `cond` form is a syntactic sugar for nested `if`-expressions.

   (d) The `define` special form in Scheme is syntactic sugar for a procedure call.

   (e) The use of a vector in Scheme is syntactic sugar for the use of a list.

3. (10 points) Consider the following grammar.

⟨require⟩ ::= `( require` {⟨mod-desig⟩}* `)`
⟨mod-desig⟩ ::= ⟨symbol⟩
| ⟨string⟩
| `( file` ⟨string⟩ `)`
| `( lib` ⟨string⟩ {⟨string⟩}* `)`

where ⟨symbol⟩ stands for a Scheme symbol, such as `x`, and ⟨string⟩ stands for a Scheme string literal, such as `"a str"`. Now consider the following input.

```
( require (lib "maybe.scm" "lib342" ) )
```

Either show how to derive the above string from the nonterminal ⟨require⟩, 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. (5 points) Write a Scheme procedure,

```
greet-all : (-> (symbol (list-of symbol)) (list-of (list-of symbol)))
```

such that `(greet-all greeting names` returns a list that is like `names`, except that each element in `names` is made into a two-element list in the result containing the greeting followed by that element. The following are examples.

```
(greet-all 'hi '()) ==> ()
(greet-all 'hi '(marie don sally fred))
    ==> ((hi marie) (hi don) (hi sally) (hi fred))
(greet-all 'hi '(don sally fred))
    ==> ((hi don) (hi sally) (hi fred))
(greet-all 'hello '(sara james witherspoon
                    humperdinck balderdash))
    ==> ((hello sara) (hello james) (hello witherspoon)
         (hello humperdinck) (hello balderdash))
```

5. (15 points) This is a problem involving the following Boolean expression grammar. As in the homework, the comments on the right are an aid to remembering the helping procedures.

⟨bexp⟩ ::=	
⟨varref⟩	"`var-exp (varref)`"
| `(and` ⟨bexp⟩ ⟨bexp⟩`)`	"`and-exp (left right)`"
| `(or` ⟨bexp⟩ ⟨bexp⟩`)`	"`or-exp (left right)`"
| `(not` ⟨bexp⟩`)`	"`not-exp (arg)`"
⟨varref⟩ ::=	
`P`	"`P ()`"
| `Q`	"`Q ()`"

The following are the types of the ⟨bexp⟩ helping procedures, from the library file `bexp.scm`.

```
var-exp? : (-> (bexp) boolean)
and-exp? : (-> (bexp) boolean)
or-exp? : (-> (bexp) boolean)
not-exp? : (-> (bexp) boolean)
P? : (-> (varref) boolean)
Q? : (-> (varref) boolean)

var-exp : (-> (varref) bexp)
and-exp : (-> (bexp bexp) bexp)
or-exp : (-> (bexp bexp) bexp)
not-exp : (-> (bexp) bexp)
P : (-> () varref)
Q : (-> () varref)

var-exp->varref : (-> (bexp) varref)
and-exp->left : (-> (bexp) bexp)
and-exp->right : (-> (bexp) bexp)
or-exp->left : (-> (bexp) bexp)
or-exp->right : (-> (bexp) bexp)
not-exp->arg : (-> (bexp) bexp)
```

(The problem continues on the next page.)

Using these helping procedures on the previous page, write a Scheme procedure,

```
count : (-> (bexp symbol) number)
```

such that `(count be sym)` is the number of times that `sym` appears in `be`. You can assume that the argument symbol, `sym`, will always be either `P` or `Q`. The following are examples.

```
(count (var-exp (P)) 'P) ==> 1
(count (var-exp (P)) 'Q) ==> 0
(count (var-exp (Q)) 'P) ==> 0
(count (var-exp (Q)) 'Q) ==> 1
(count (not-exp (var-exp (P))) 'P) ==> 1
(count (and-exp (not-exp (var-exp (Q))) (var-exp (Q))) 'Q) ==> 2
(count (or-exp (and-exp (not-exp (var-exp (Q))) (var-exp (Q)))
               (not-exp (and-exp (var-exp (P)) (var-exp (Q))))) 'Q) ==> 3
(count (and-exp
        (or-exp (and-exp (not-exp (var-exp (Q))) (var-exp (Q)))
                (not-exp (and-exp (var-exp (P)) (var-exp (Q)))))
        (or-exp (and-exp (not-exp (var-exp (Q))) (var-exp (Q)))
                (not-exp (and-exp (var-exp (P)) (var-exp (Q))))))
       'P) ==> 2
```

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,

```
elim-useless : (-> (statement) statement)
```

such that `(elim-useless stmt)` that takes a ⟨statement⟩, `stmt`, and returns a ⟨statement⟩ that is the same except that useless set statements that have exactly the form produced by `(set-stmt` x `(var-exp` x`))`, for some symbol x, are replaced with the expression-statement `(exp-stmt (var-exp` x`))`. Your answer must properly use the helpers for the above grammar, whose types are given above.

```
(elim-useless (set-stmt 'y (var-exp 'y)))
   = (exp-stmt (var-exp 'y))
(elim-useless (set-stmt 'val (var-exp 'val)))
```

```
   = (exp-stmt (var-exp 'val))
(elim-useless (set-stmt 'i (num-exp 541)))
   = (set-stmt 'i (num-exp 541))
(elim-useless (set-stmt 'val (begin-exp '() (var-exp 'val))))
   = (set-stmt 'val (begin-exp '() (var-exp 'val)))
(elim-useless (exp-stmt (var-exp 'm)))
   = (exp-stmt (var-exp 'm))
(elim-useless (exp-stmt (num-exp 1)))
   = (exp-stmt (num-exp 1))
(elim-useless
   (set-stmt 'val (begin-exp (list (set-stmt 'val (var-exp 'val)))
                             (var-exp 'val))))
 = (set-stmt 'val (begin-exp (list (exp-stmt (var-exp 'val)))
                             (var-exp 'val)))
(elim-useless
   (exp-stmt (begin-exp (list (set-stmt 'val (var-exp 'val))
                              (set-stmt 'x (var-exp 'y)))
                        (num-exp 342))))
 = (exp-stmt (begin-exp (list (exp-stmt (var-exp 'val))
                              (set-stmt 'x (var-exp 'y)))
                        (num-exp 342)))
(elim-useless
   (exp-stmt (begin-exp (list (set-stmt 'val (var-exp 'y))
                              (set-stmt 'x (var-exp 'x)))
                        (num-exp 342))))
 = (exp-stmt (begin-exp (list (set-stmt 'val (var-exp 'y))
                              (exp-stmt (var-exp 'x)))
                        (num-exp 342)))
(elim-useless
   (set-stmt 'z (begin-exp
                 (list
                  (exp-stmt (begin-exp (list (set-stmt 'val (var-exp 'y))
                                             (set-stmt 'x (var-exp 'x)))
                                       (num-exp 342))))
                 (begin-exp (list (set-stmt 'x (var-exp 'x)))
                            (var-exp 'val)))))
 = (set-stmt 'z (begin-exp
                 (list
                  (exp-stmt (begin-exp (list (set-stmt 'val (var-exp 'y))
                                             (exp-stmt (var-exp 'x)))
                                       (num-exp 342))))
                 (begin-exp (list (exp-stmt (var-exp 'x)))
                            (var-exp 'val)))))
```

There is space for your answer on the next page.

```
;; Write your answer below.

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

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

⟨type-expr⟩ ::=

	⟨symbol⟩	"`basic-type (name)`"
|	`(->` `(`{⟨type-expr⟩}*`)` ⟨type-expr⟩`)`	"`proc-type (arg-types result-type)`"

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

```
basic-type? : (-> (type-expr) boolean)
proc-type? : (-> (type-expr) boolean)
basic-type : (-> (symbol) type-expr)
proc-type : (-> ((list type-expr) type-expr) type-expr)
basic-type->name : (-> (type-expr) symbol)
proc-type->arg-types : (-> (type-expr) (list type-expr))
proc-type->result-type : (-> (type-expr) type-expr)
```

Assuming you have above helpers, write a procedure,

```
subst : (-> (symbol symbol type-expr) type-expr))
```

such that `subst new old typ` returns a type that is just like `typ`, except that all uses of `old` as a ⟨basic-type⟩ within the argument `typ` are replaced by `new`. The following are examples.

```
(subst 'y 'g (basic-type 'g))
           = (basic-type 'y)
(subst 'y 'g (basic-type 'foo))
           = (basic-type 'foo)
(subst 'z 'number (proc-type (list (basic-type 'number)) (basic-type 'number)))
                 = (proc-type (list (basic-type 'z)) (basic-type 'z))
(subst 'n 'o
      (proc-type
        (list (proc-type
                 (list (proc-type (list (basic-type 'o)) (basic-type 'o))
                       (proc-type (list (basic-type 'x)) (basic-type 'o))
                       (proc-type (list (basic-type 'o)) (basic-type 'x)))
                 (proc-type (list (basic-type 'o) (basic-type 'o))
                            (basic-type 'o))))
        (proc-type (list (basic-type 'o)) (basic-type 'o))))
    = (proc-type
        (list (proc-type
                 (list (proc-type (list (basic-type 'n)) (basic-type 'n))
                       (proc-type (list (basic-type 'x)) (basic-type 'n))
                       (proc-type (list (basic-type 'n)) (basic-type 'x)))
                 (proc-type (list (basic-type 'n) (basic-type 'n))
                            (basic-type 'n))))
        (proc-type (list (basic-type 'n)) (basic-type 'n)))
```

Please write your answer below.

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

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

```
any-equals? : (-> (number (vector-of number)) boolean)
```

such that `(any-equals? n von)` returns `#t` if there is some index, $0 \leq i < len$, where *len* is the length of `von`, such that element i of `von` is equal to `n`. The following are examples.

```
(any-equals? 5 (vector 3 4 2)) ==> #f
(any-equals? 4 (vector 3 4 2)) ==> #t
(any-equals? 4 (vector 3 4 2 2 4 3 5 10 4)) ==> #t
(any-equals? 4 (vector 3 3 2)) ==> #f
(any-equals? 4 (vector )) ==> #f
(any-equals? 7 '#(17 10 1000 27 541 342 8 95 3 9 7 22 5)) ==> #t
(any-equals? 7 '#(17 10 1000 27 541 342 8 95 3 9 22 5)) ==> #f
```

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.