Fall 2001 Name: _____
Com S 342 TA (or Section): _____

Principles of Programming Languages

Exam 2 on Induction and Recursion

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

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.

1. (10 points) Consider the following BNF grammar.

⟨type⟩ ::= ⟨identifier⟩
    | `(` `->` `(` ⟨type-list⟩ `)` ⟨type⟩ `)`
    | `(` `list-of` ⟨type⟩ `)`
⟨identifier⟩ ::= `S` | `T` | `U`
⟨type-list⟩ ::= ⟨empty⟩ | ⟨type⟩ ⟨type-list⟩
⟨empty⟩ ::=

Now consider the following string.

```
( -> ( ( list-of S ) ) T )
```

Either show how to derive the above string from the nonterminal ⟨type⟩ using the given grammar, or briefly explain why no derivation is possible.

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

2. (15 points) Write a Scheme procedure, `range`,

```
(deftype range (forall (T) (-> ((list-of (list-of T))) (list-of T))))
```

that takes a list, `rel`, of two-element lists, and produces a list consisting of the second element in each list of `rel`. The following are examples.

```
(range '()) ==> ()
(range '((2 3))) ==> (3)
(range '((2 3) (4 5) (6 7) (99 105))) ==> (3 5 7 105)
(range '((x xv) (y yv) (z zv) (x xv) (z zv) (c cv)))
    ==> (xv yv zv xv zv cv)
(range '((x xv) (y yv) (z zv) (x xv) (z zv) (c cv) (q qv) (r rv)))
    ==> (xv yv zv xv zv cv qv rv)
```

Assume that the argument is a list of two-element lists.

3. (10 points) Write a Scheme procedure, `range*`,

```
(deftype range* (forall (T) (-> ((list-of T) ...) (list-of T))))
```

that takes any number of two-element lists, and produces a list consisting of the second element of each of the two element lists given as arguments. (You may use the solution for the previous problem as a helping procedure.) The following are examples.

```
(range* ) ==> ()
(range* '(2 3)) ==> (3)
(range* '(2 3) '(4 5) '(6 7) '(99 105)) ==> (3 5 7 105)
(range* '(x xv) '(y yv) '(z zv) '(x xv) '(z zv) '(c cv))
    ==> (xv yv zv xv zv cv)
```

Assume that the arguments are all two-element lists.

4. (25 points) Write a Scheme procedure, `update`,

```
(deftype update (forall (T) (-> (T T (list-of (list-of T)))
                                (list-of (list-of T)))))
```

that takes two values, `key` and `new-val`, and a list, `rel`, of two-element lists, and produces a list of two-element lists that is like `rel` except that the first two-element list in `rel` that has `key` as its first element is replaced by a two-element list whose first element is `key` and whose second element is `new-val`. (Use `equal?` to compare `key` and the first elements in the two-element lists.) The following are examples.

```
(update 'k 'v '()) ==> ()
(update 'k 'v '((k y))) ==> ((k v))
(update 'k 'v '((k x) (k y))) ==> ((k v) (k y))
(update 3 4 '((1 2) (2 3) (1 5) (2 7)))
         ==> ((1 2) (2 3) (1 5) (2 7))
(update 3 4 '((3 9) (1 2) (2 3) (1 5) (2 7)))
         ==> ((3 4) (1 2) (2 3) (1 5) (2 7))
(update 9 3 '((3 9) (1 2) (2 3) (1 5) (2 7)))
         ==> ((3 9) (1 2) (2 3) (1 5) (2 7))
(update 9 3 '((9 0) (9 0) (9 0) (9 0) (9 0) (9 0) (9 0) (9 0) (9 0)))
         ==> ((9 3) (9 0) (9 0) (9 0) (9 0) (9 0) (9 0) (9 0) (9 0))
(update 9 3 '((3 0) (3 0) (2 0) (9 0) (3 0)))
         ==> ((3 0) (3 0) (2 0) (9 3) (3 0))
```

Assume that the argument is a list of two-element lists.

5. (25 points) This is a problem about the type `sym-exp`. In your solution you may use the sym-exp helpers whose types are listed below, except that you may not use `parse-sym-exp`.

```
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))
parse-sym-exp   : (-> (datum) sym-exp)
```

Write a Scheme procedure, `graph-sym-exp`,

```
(deftype graph-sym-exp (-> ((-> (symbol) symbol) sym-exp) sym-exp))
```

that takes a procedure, `f`, and a sym-exp, `se`, and returns a sym-exp that has the same shape as `se`, but with each symbol, x, in `se` replaced by two element s-list whose first element is x and whose second element is the result of applying `f` to x. For example:

```
(graph-sym-exp (lambda (x) 'y) (parse-sym-exp 'x)) ==> (x y)
(graph-sym-exp (lambda (x) 'y) (parse-sym-exp '())) ==> ()
(graph-sym-exp (lambda (x) 'y) (parse-sym-exp '(a () b))) ==> ((a y) () (b y))
(graph-sym-exp (lambda (x) (if (eq? x 'coms) 'cpre 'mis))
               (parse-sym-exp '((a dept) ((of))
                                (coms (() or math)) ()))
                           ==> (((a mis) (dept mis)) (((of mis)))
                                ((coms cpre) (() (or mis) (math mis))) ())
(graph-sym-exp (lambda (x) (if (eq? x 'coms) 'cpre 'mis))
               (parse-sym-exp '(((of))
                                (coms (() or math)) ()))
                           ==> ((((of mis)))
                                ((coms cpre) (() (or mis) (math mis))) ())
```

We will take off points for procedures that do not type check, so use the sym-exp helpers.

There is more space for your answer on the next page.

```
;; space for your answer to the problem on the previous page
```

6. (15 points) Without using `vector->list`, write a procedure, `vector-increasing?`,

```
(deftype vector-increasing? (-> ((vector-of number)) boolean))
```

that takes a vector of numbers, `vec`, and returns true just when the elements of `vec` are in strictly increasing order. That is, the result is true if and only if for all $0 \leq i < j < len$, where *len* is the length of `vec`, the ith element of `vec` is strictly less than the jth element. The following are examples. (Recall that `(vector 3) = '#(3)`.)

```
(vector-increasing? (vector )) ==> #t
(vector-increasing? (vector 99)) ==> #t
(vector-increasing? (vector 99 99)) ==> #f
(vector-increasing? '#(1 5 99)) ==> #t
(vector-increasing? '#(99 3 1 2 99)) ==> #f
(vector-increasing? '#(1 2 3 4 5 7 99 100 101 1000 9999)) ==> #t
(vector-increasing? '#(1 2 3 4 5 7 99 99 100 101 1000 9999)) ==> #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. However, you may *not* use `vector->list`. Your code must be in Scheme, of course.