Spring, 1999 Name: _____

My Section Letter: _____ My Section Day and Time : _____

Com S 342 — Principles of Programming Languages

Test on *EOPL* Chapters 1 to 2.2

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

Reminders

This test is closed book and notes.

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. We may take off a small amount if you do not give `TYPE` comments for recursive helping procedures. However, you do not have to write such comments for procedures for which the type is stated in the problem. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

Subset of Scheme You May Use

Unless otherwise stated, when defining procedures you may only use: helping procedures that you define yourself, comments, and the procedures and keywords that are included in the following list. (That is, on this test do *not* use Scheme procedures and keywords that are absent from the following list.) The notation `c...r` means `caar`, `cadr`, `cddr`, `cdar`, `caaar`, etc. You may only use `define` at the top level.

```
', #t, #f, *, +, -, /, <, <=, =, >=, >,
and, append, apply, boolean?, car, cdr,  c...r, char?,
cond, cons, define, display, else, eq?, equal?, eqv?, error,
if, let, letrec, lambda, list, length, list?, map, member, memq, memv, newline,
not, null?, number?, or, pair?, procedure?, quote, string,
string?, string=?, string-append, string-ci=?, string-length,
string-ref, string->list, string->number, string->symbol,
substring, symbol?, vector, vector?, vector-length,
vector->list, vector-ref, zero?
```

1. (5 points) Write a Scheme expression that produces the following value. Your expression should use only parentheses, the procedure `cons`, quoted symbols (such as `'this`), and the empty list, `'()`.

```
(thats (wrapped))
```

2. (5 points) Given the following definition,

```
(define the-datum
  '(a problem (like this-one) may be too easy for you))
```

write a Scheme expression using procedures like `car`, `cdr`, etc., that extracts the symbol `this-one` from `the-datum`. (Note: the expression you write should depend on the value of `the-datum`, so `'this-one` is not correct.)

3. (10 points) Consider the following grammar.

⟨decl⟩ ::= `var` ⟨name⟩ `:=` ⟨expr⟩
| `const` ⟨name⟩ `=` ⟨expr⟩
| ⟨decl⟩ `;` ⟨decl⟩
⟨name⟩ ::= `a` | `b` | `c` | `x` | `y` | `z`
⟨expr⟩ ::= ⟨name⟩ | ⟨expr⟩ ⟨op⟩ ⟨expr⟩
⟨op⟩ ::= `+` | `-` | `*` | `/`

In each of the spaces provided ("_____") below, write "yes" if the text is an example of a ⟨decl⟩ in the above grammar, and "no" if it is not.

(a) _____ `( a - b )`

(b) _____ `var x := a - b`

(c) _____ `const y : integer = x + y`

(d) _____ `const x = b; var y := c / a`

(e) _____ `var x := x`

4. (10 points) Define a curried version of the following procedure.

```
(define eval-quadratic
  ;; TYPE: (-> (number number number number) number)
  (lambda (a b c x)
    (+ (* a (square x)) (* b x) c)))
```

5. (15 points) Write a procedure, `square-each`, with type

```
(-> ((list number)) (list number))
```

that takes a list of numbers, `lon`, and returns a list of numbers of the same length, but with each item in the result being the square of the corresponding item in the argument. The following are examples.

```
(square-each '(3 10 5 3)) ==> (9 100 25 9)
(square-each '()) ==> ()
(square-each '(5 3)) ==> (25 9)
(square-each '(88 512 92 400 35)) ==> (7744 262144 8464 160000 1225)
```

6. (10 points) Write a procedure `square-each*` with type

```
(-> (number ...) (list number))
```

that takes zero or more numbers as arguments, and returns a list of their squares. The following are examples.

```
(square-each* ) ==> ()
(square-each* 5 1) ==> (25 1)
(square-each* 10 5 1) ==> (100 25 1)
(square-each* 19 24 3 4 2 8 241 541) ==> (361 576 9 16 4 64 58081 292681)
```

7. (25 points) Write a procedure, `list-diff`, with type

```
(-> ((list T) (list T)) (list T))
```

that takes two lists, `ls1` and `ls2`, and returns a list that is the same as `ls1`, except that the result contains no item that is found in `ls2`. The following are examples.

```
(list-diff '() '(a b)) ==> ()
(list-diff '(a a b a) '(a b)) ==> ()
(list-diff '(c a a b a) '(a b)) ==> (c)
(list-diff '(c a d w h y) '(a x d)) ==> (c w h y)
(list-diff '(c a d w h y) '(x d)) ==> (c a w h y)
(list-diff '(5 2 3 1 7) '()) ==> (5 2 3 1 7)
(list-diff '(5 2 2 5) '()) ==> (5 2 2 5)
```

8. (30 points) This is a problem about s-lists. You may in your solution the sym-exp helpers as in the homework. The types of the ones you will find useful in your solution are as follows:

```
symbol->sym-exp : (-> (symbol) sym-exp)
s-list->sym-exp : (-> ((list 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 sym-exp))
```

Write a procedure `insert-left-of` with type

```
(-> (symbol symbol (list sym-exp)) (list sym-exp))
```

that takes two symbols, `what` and `where`, and a list of sym-exps, `slist`, and returns a list of sym-exps that is just like `slist`, except that `what` is found just to the left of each occurrence of `where`. The following are examples.

```
(insert-left-of 'x 'here (parse-s-list '(an here goes)))
                                    ==> (an x here goes)
(insert-left-of 'x 'here (parse-s-list '(here goes))) ==> (x here goes)
(insert-left-of 'x 'here (parse-s-list '(here here))) ==> (x here x here)
(insert-left-of 'ah 'here (parse-s-list '((here) um () (here))))
                                     ==> ((ah here) um () (ah here))
(insert-left-of 'a 'now (parse-s-list '(now (is now) (not then (now)))))
                                   ==> (a now (is a now) (not then (a now)))
```

We will take off a small number of points for procedures that do not type check.

9. (30 points) Consider the following grammar.

⟨v-exp⟩ ::= ⟨varref⟩ | ⟨number⟩ | `(-` ⟨v-exp⟩ ⟨v-exp⟩`)`
| `(vector-length` ⟨v-exp⟩`)` | `(vector-ref` ⟨v-exp⟩ ⟨v-exp⟩`)`
⟨varref⟩ ::= ⟨symbol⟩

In this grammar, the nonterminals ⟨number⟩ and ⟨symbol⟩ have the same syntax as in Scheme. In your solution use the helping procedures for this grammar found on the following pages, so that your code will type check.

Write a procedure, `sub1-ref`, with the following type.

```
(-> (v-exp) v-exp)
```

This procedure takes an ⟨v-exp⟩, `e`, and returns an ⟨v-exp⟩ that is the same as `e`, except for the following change. The change is that each ⟨v-exp⟩ of the form `(vector-ref` e_1 e_2`)` is changed to the form `(vector-ref` e_1 `(-` e_2 `1))` The following are examples. (Note that these are written as equations, using the helpers on the following pages.)

```
(sub1-ref (parse-v-exp 'x)) = (make-varref 'x)
(sub1-ref (parse-v-exp 3)) = (make-number 3)
(sub1-ref (parse-v-exp '(- x 3))) = (make-minus (make-varref 'x) (make-number 3))
(sub1-ref (parse-v-exp '(vector-length y)))
  = (make-vector-length (make-varref 'y))
(sub1-ref (parse-v-exp '(vector-ref z 3)))
  = (make-vector-ref (make-varref 'z)
                     (make-minus (make-number 3) (make-number 1)))
(sub1-ref (parse-v-exp '(vector-ref y (vector-ref z 3))))
  = (make-vector-ref (make-varref 'y)
                     (make-minus
                      (make-vector-ref (make-varref 'z)
                                       (make-minus (make-number 3)
                                                   (make-number 1)))
                      (make-number 1)))
```

```
;; <v-exp> ::= <varref> | <number> | (- <v-exp> <v-exp>)
;;         | (vector-length <v-exp>) | (vector-ref <v-exp> <v-exp>)

(define v-exp?  ; TYPE: (-> (datum) boolean)
  (lambda (d)
    (or (varref? d)  (number? d)  (minus? d)
        (vector-length? d)
        (vector-ref? d))))

(define varref?        ; TYPE: (-> (v-exp) boolean)
  symbol?)

; you can also use number? with type (-> (v-exp) boolean)

(define keyword-match?-maker
  ;; TYPE: (-> (symbol number) (-> (v-exp) boolean))
  (lambda (keyword len)
    (lambda (e)
      (and (list? e)
           (= (length e) len)
           (eq? keyword (car e))))))

(define minus?         ; TYPE: (-> (v-exp) boolean)
  (keyword-match?-maker '- 3))

(define vector-length? ; TYPE: (-> (v-exp) boolean)
  (keyword-match?-maker 'vector-length 2))

(define vector-ref?    ; TYPE: (-> (v-exp) boolean)
  (keyword-match?-maker 'vector-ref 3))

(define make-varref    ; TYPE: (-> (symbol) v-exp)
  (lambda (symbol)
    symbol))

(define make-number    ; TYPE: (-> (number) v-exp)
  (lambda (value)
    value))

(define make-minus     ; TYPE: (-> (v-exp v-exp) v-exp)
  (lambda (left right)
    (list '- left right)))

(define make-vector-length ; TYPE: (-> (v-exp) v-exp)
  (lambda (exp)
    (list 'vector-length exp)))

(define make-vector-ref ; TYPE: (-> (v-exp v-exp) v-exp)
```

```
  (lambda (vec index)
    (list 'vector-ref vec index)))

(define extract-maker
  ;; TYPE: (-> ((-> (datum) boolean) string) (-> (T) T))
  (lambda (test? name)
    (lambda (x)
      (if (test? x)
          x
          (error (string-append "not a " name " v-exp:") x)))))

(define varref->symbol  ; TYPE: (-> (v-exp) symbol)
  (extract-maker varref? "varref"))

(define number->value   ; TYPE: (-> (v-exp) number)
  (extract-maker number? "number"))

(define minus->left     ; TYPE: (-> (v-exp) v-exp)
  (lambda (e)
    (cadr ((extract-maker minus? "minus") e))))

(define minus->right    ; TYPE: (-> (v-exp) v-exp)
  (lambda (e)
    (caddr ((extract-maker minus? "minus") e))))

(define vector-length->exp ; TYPE: (-> (v-exp) v-exp)
  (lambda (e)
    (cadr ((extract-maker vector-length? "vector-length") e))))

(define vector-ref->vec ; TYPE: (-> (v-exp) v-exp)
  (lambda (e)
    (cadr ((extract-maker vector-ref? "vector-ref") e))))

(define vector-ref->index ; TYPE: (-> (v-exp) v-exp)
  (lambda (e)
    (caddr ((extract-maker vector-ref? "vector-ref") e))))

(define parse-v-exp       ; TYPE: (-> (datum) v-exp)
  (lambda (d)
    (cond
     ((varref? d) (make-varref d))
     ((number? d) (make-number d))
     ((minus? d) (make-minus (cadr d) (caddr d)))
     ((vector-length? d) (make-vector-length (cadr d)))
     ((vector-ref? d) (make-vector-ref (cadr d) (caddr d)))
     (else (error "parse-v-exp: bad syntax:" d)))))
```