Fall 2000 Name: _____

22C:54 — Programming Language Concepts

Test on *EOPL* Chapters 1 to 2.2

This test has 10 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, max, member, memq, memv, min, 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) Broadly speaking, the features of a programming language can be classified into 3 all-encompassing categories. We used these categories to help organize our design at the beginning of the course. Name these three categories, and give one example of each in Scheme.

2. (5 points) Given the following definition,

```
(define the-sports
  '(softball baseball (not gymnastics) volleyball womens-diving))
```

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

3. (5 points) Draw a box-and-pointer diagram for the following list.

```
((olympics) in sydney)
```

4. (5 points) Using only parentheses, the procedure `cons`, quoted symbols (such as `'this`), and the empty list, `'()`, write a Scheme expression that produces the following list.

```
((olympics) in sydney)
```

5. (10 points) Consider the following grammar.

⟨ml-type⟩ ::= ⟨basic-type⟩
| `(`⟨ml-type⟩ {× ⟨ml-type⟩}*`)`
| ⟨ml-type⟩ `->` ⟨ml-type⟩
⟨basic-type⟩ ::= `int` | `real` | `char` | `bool`

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

(a) _____ `real`

(b) _____ `(-> (int) real)`

(c) _____ `(int × real × int)`

(d) _____ `(int × real) -> bool`

(e) _____ `int -> int -> (int × int)`

6. (5 points) Define a curried version of the following procedure, call it `average3-c` and write the deftype for it as well.

```
(deftype average3 (-> (number number number) number))
(define average3
  (lambda (a b c)
    (/ (+ a b c) 3)))
```

7. (10 points) Write a procedure, `qualifies`, with type

```
(deftype qualifies (-> ((list number)) (list boolean)))
```

that takes a list of numbers, `lon`, and returns a list of booleans of the same length, but with each item in the result `#t` just when the corresponding item in the argument is strictly greater than 7.9. The following are examples.

```
(qualifies '(8.0 7.9 6.5 8.1 9.5 9.0)) ==> (#t #f #f #t #t #t)
(qualifies '(9.0)) ==> (#t)
(qualifies '()) ==> ()
(qualifies '(0.0 2.1 9.3 8.2)) ==> (#f #f #t #t)
```

8. (5 points) Write a procedure `qualifies*` with type

```
(deftype qualifies* (-> (number ...) (list boolean)))
```

that takes zero or more numbers as arguments, and returns a list of booleans of the same length, but with each item in the result `#t` just when the corresponding argument is strictly greater than 7.9. You may, of course, use `qualifies` as a helping procedure.

```
(qualifies* 8.0 7.9 6.5 8.1 9.5 9.0) ==> (#t #f #f #t #t #t)
(qualifies* 9.0) ==> (#t)
(qualifies* ) ==> ()
(qualifies* 0.0 2.1 9.3 8.2) ==> (#f #f #t #t)
```

9. (25 points) This is a problem about the type sym-exp. In your solution you may use the sym-exp helpers, as in the homework. Their types are summarized below.

```
(deftype s-list? (-> (datum) boolean))
(deftype sym-exp? (-> (datum) boolean))
(deftype symbol->sym-exp (-> (symbol) sym-exp))
(deftype s-list->sym-exp (-> ((list sym-exp)) sym-exp))
(deftype sym-exp-symbol? (-> (sym-exp) boolean))
(deftype sym-exp-s-list? (-> (sym-exp) boolean))
(deftype sym-exp->symbol (-> (sym-exp) symbol))
(deftype sym-exp->s-list (-> (sym-exp) (list sym-exp)))
(deftype parse-sym-exp (-> (datum) sym-exp))
(deftype parse-s-list (-> (datum) (list sym-exp)))
```

Write a procedure `wrap-sym-exp` with type

```
(deftype wrap-sym-exp (-> (sym-exp) sym-exp))
```

that takes a sym-exp, `se`, and returns a sym-exp that is just like `se`, except that each symbol in `se` is wrapped in a list. The following are examples.

```
(wrap-sym-exp (parse-sym-exp 'eh)) ==> (eh)
(wrap-sym-exp (parse-sym-exp '())) ==> ()
(wrap-sym-exp (parse-sym-exp '(and (swimming)))) ==> ((and) ((swimming)))
(wrap-sym-exp (parse-sym-exp '(water polo () (and (swimming)))))
                          ==> ((water) (polo) () ((and) ((swimming))))
```

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

10. (25 points) Consider the following grammar.

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

In this grammar, the nonterminals ⟨number⟩ and ⟨symbol⟩ have the same syntax as in Scheme. In your solution use the helping procedures starting on page 8, so that your code will type check.

In the space provided on the next page, write a procedure, `total-width`, with type:

```
(deftype total-width (-> (window-layout) number))
```

This procedure takes a ⟨window-layout⟩, `wl`, and returns the total width of the layout. The width of a ⟨window-layout⟩ of the form `(window` s n_1 n_2`)` is n_1. The width of a ⟨window-layout⟩ of the form `(horizontal` w_1 w_2 ... w_m`)` is the sum of the widths of w_1 through w_m (inclusive). The width of a ⟨window-layout⟩ of the form `(vertical` w_1 w_2 ... w_m`)` is the maximum of the widths of w_1 through w_m (inclusive). The following are examples.

```
(total-width (parse-window-layout '(window olympics 50 33))) ==> 50
(total-width
   (parse-window-layout '(horizontal (window olympics 80 33)
                                     (window local-news 20 10)))) ==> 100
(total-width
   (parse-window-layout '(vertical (window olympics 80 33)
                                   (window local-news 20 10)))) ==> 80
(total-width
   (parse-window-layout '(vertical (window star-trek 40 100)
                                   (window olympics 80 33)
                                   (window local-news 20 10)))) ==> 80
(total-width
   (parse-window-layout
      '(horizontal
          (vertical (window tempest 200 100)
                    (window othello 200 77)
                    (window hamlet 1000 600))
          (horizontal (window baseball 50 40)
                      (window track 100 60)
                      (window equesterian 70 30))
          (vertical (window star-trek 40 100)
                    (window olympics 80 33)
                    (window local-news 20 10))))) ==> 1300
```

Feel free to use Scheme's `map` and `max` procedures, as well as any others listed on the front page as well as the helpers starting on page 8. There is space for your answer on the next page.

;;; space for your answer, please write it below.

```
;;; <window-layout> ::= (window <symbol> <width> <height>)
;;;         | (horizontal {<window-layout>}*)
;;;         | (vertical {<window-layout>}*)
;;; <width> ::= <number>
;;; <height> ::= <number>

trustme!  ;; suppress the type checker's confusion about this file

(load-quietly-from-lib "all.scm")

(defrep window-layout (list datum))

(deftype window-layout? (-> (datum) boolean))
(define window-layout?
  (lambda (d)
    (or (window? d)
        (and (horizontal? d) (all window-layout? (cdr d)))
        (and (vertical? d) (all window-layout? (cdr d))))))

(deftype window? (-> (window-layout) boolean))
(define window?
  (lambda (wl)
    (and (list? wl) (= (length wl) 4)
         (eq? (car wl) 'window)
         (symbol? (cadr wl)) (number? (caddr wl)) (number? (cadddr wl)))))

(deftype horizontal? (-> (window-layout) boolean))
(define horizontal?
  (lambda (wl)
    (and (list? wl) (>= (length wl) 1)
         (eq? (car wl) 'horizontal))))

(deftype vertical? (-> (window-layout) boolean))
(define vertical?
  (lambda (wl)
    (and (list? wl) (>= (length wl) 1)
         (eq? (car wl) 'vertical))))

(deftype make-window (-> (symbol number number) window-layout))
(define make-window
  (lambda (name width height)
    (list 'window name width height)))

(deftype make-horizontal (-> ((list window-layout)) window-layout))
(define make-horizontal
  (lambda (lwl)
    (cons 'horizontal lwl)))
```

```
(deftype make-vertical (-> ((list window-layout)) window-layout))
(define make-vertical
  (lambda (lwl)
    (cons 'vertical lwl)))

(deftype extract-maker (-> ((-> (window-layout) boolean) string)
                           (-> (window-layout) (list datum))))
(define extract-maker
  (lambda (test? name)
    (lambda (x)
      (if (test? x)
          x
          (error (string-append "not a " name " window-layout:") x)))))

(deftype window->name (-> (window-layout) symbol))
(define window->name
  (lambda (wl)
    (cadr ((extract-maker window? "window") wl))))

(deftype window->width (-> (window-layout) number))
(define window->width
  (lambda (wl)
    (caddr ((extract-maker window? "window") wl))))

(deftype window->height (-> (window-layout) number))
(define window->height
  (lambda (wl)
    (cadddr ((extract-maker window? "window") wl))))

(deftype horizontal->subwindows (-> (window-layout) (list window-layout)))
(define horizontal->subwindows
  (lambda (wl)
    (cdr ((extract-maker horizontal? "horizontal") wl))))

(deftype vertical->subwindows (-> (window-layout) (list window-layout)))
(define vertical->subwindows
  (lambda (wl)
    (cdr ((extract-maker vertical? "vertical") wl))))

(deftype parse-window-layout (-> (datum) window-layout))
(define parse-window-layout
  (lambda (d)
    (cond
     ((window? d) (make-window (cadr d) (caddr d) (cadddr d)))
     ((horizontal? d) (make-horizontal (map parse-window-layout (cdr d))))
     ((vertical? d) (make-vertical (map parse-window-layout (cdr d))))
     (else (error "parse-window-layout: bad syntax:" d)))))
```