Spring 2004 Name: _____
Com S 342 Section: _____

Principles of Programming Languages

Exam 1 on Language Design and Scheme Basics

This test has 11 questions and pages numbered 1 through 7.

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. Correct syntax also matters. Check your code over for syntax errors. You will lose points if your code has syntax errors.

1. (5 points) Give an example of a "means of computation" in Scheme. Your example should consist of a short English description and a bit of Scheme code.

2. (5 points) Give an example of a "means of combination" in Scheme. Your example should consist of a short English description and a bit of Scheme code.

3. (5 points) Give an example of a "means of abstraction" in Scheme. Your example should consist of a short English description and a bit of Scheme code.

4. (10 points) In our class discussion about the design of Simple, what goals made us decide that we could do without a built-in data type of "character string"? Explain briefly.

5. (5 points) Unlike C++, Java features garbage collection, which returns unreachable dynamically allocated storage to the free space in the heap. What design goals of Java influenced this feature in Java? Briefly explain.

6. (10 points) Given the following Scheme definition,

```
(define usa-states
  '((north (alaska (north dakota) michigan))
    (middle (iowa nebraska illinois))
    (east ((rhode island) (new jersey)))))
```

write a Scheme expression using procedures like `car`, `cdr`, `caar`, etc., that extracts the symbol `iowa` from `usa-states`. (Note: the expression you write should depend on the value of `usa-states`, so `'iowa` is not correct. You may not use `list-ref` in your answer.)

7. (10 points) Draw a box-and-pointer diagram for the following list.

```
((north carolina) florida louisiana)
```

8. (10 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.

```
((north carolina) florida louisiana)
```

9. (10 points) Write a Scheme procedure, `make-question`, with type

```
(-> ((list-of symbol)) (list-of symbol))
```

which takes a list of exactly three symbols, and returns a list consisting of the second, first, and third symbols in that order, as shown in the following examples. (Hint: this is *not* a problem about recursion.)

```
(make-question '(scheme is good)) ==> (is scheme good)
(make-question '(charlene is careful)) ==> (is charlene careful)
(make-question '(bill fleeces customers)) ==> (fleeces bill customers)
```

10. (10 points) Write a recursive Scheme procedure `all-pork-dishes?`, of type

```
(-> ((list-of (list-of-symbol))) boolean)
```

that takes a list of non-empty lists of symbols, `lls`, and returns `#t` just when all the top-level elements of `lls` are lists that start with the symbol `pork`, and `#f` otherwise. You can assume that each inner list has at least one element. For example,

```
(all-pork-dishes? '()) ==> #t
(all-pork-dishes? '((pork chops))) ==> #t
(all-pork-dishes? '((shredded beef) (mongolian beef))) ==> #f
(all-pork-dishes? '((pork fried rice) (pork chou mein) (pork))) ==> #t
(all-pork-dishes? '((pork chou mein) (pork))) ==> #t
(all-pork-dishes? '((pork))) ==> #t
(all-pork-dishes? '((ham) (bacon))) ==> #f
(all-pork-dishes? '((mushu pork) (broccoli and pork))) ==> #f
(all-pork-dishes? '((pork roast) (tofu) (wine) (pork and beans))) ==> #f
```

11. (20 points) Write a recursive Scheme procedure `all-bigger-than-first?`, with type

```
(-> ((list-of number)) boolean)
```

that takes a list of numbers, `lon`, and returns `#t` just when all the elements of `lon` are strictly greater than `lon`'s first element, if any, and `#f` otherwise. For example,

```
(all-bigger-than-first? '()) ==> #t
(all-bigger-than-first? '(1 16 3 4 16)) ==> #t
(all-bigger-than-first? '(1 3 4 16)) ==> #t
(all-bigger-than-first? '(1 4 16)) ==> #t
(all-bigger-than-first? '(1 16)) ==> #t
(all-bigger-than-first? '(1)) ==> #t
(all-bigger-than-first? '(17 16 3 4 16)) ==> #f
(all-bigger-than-first? '(17 3 4 16)) ==> #f
(all-bigger-than-first? '(17 4 16)) ==> #f
(all-bigger-than-first? '(0 0)) ==> #f
(all-bigger-than-first? '(0 15 0)) ==> #f
(all-bigger-than-first? '(-4.3 3.14159 2.73 342.512 10e-20)) ==> #t
```

Hint: you may want to use a helping procedure.