Homework 4: Language Design and Scheme

This homework is due February 25, 1991, (by 5:00 PM).

Test your programs using the `scheme` command on Unix, which is the interpreter described in Appendix E. Programs that use `while` and programs that use `set` to modify variables will receive less credit than programs that do not use these features. Put your Scheme code for the standard problems in `hw4.l`. Hand in a print-out of the code along with your answers to other problems, and please put your name on both. (If you do extra-credit problems, please write the names of the file(s) where the source code for your solutions can be found on your print-out.)

1. (20 points) A compound data structure is a data structure that contains other data structures as elements. In LISP, there is only one compound data structure: s-expressions, which may contain symbols, numbers, or other s-expressions.

   Is having only one compound data structure in a language a good idea? Why or why not? (Consider as many issues as you think relevant.)

   Does APL have more than one compound data structure? How does APL compare with LISP for the issues you consider?

2. (15 points) John Backus, the leader of the group that designed and build FORTRAN, has criticized the use of "lambda" in programming languages. He says that in a language like APL, there are only a few combining forms (operators), whereas with lambda one can make as many kinds of operators as one wishes. For example, one can define the notion of "reduction" in Scheme, and use it to define APL's `+/` and `*/` operators. His criticism is that with a limited set of operators, one learns to know the laws that govern their use (for example, the equations we discussed in class), but with a general "tool making facility" like lambda, one never learns such laws, because one is always inventing new combining forms.

   Take a stand, either for or against Backus's position, and carefully state the arguments for your position. Use examples from APL and Scheme to illustrate your arguments.

3. (extra credit only) Make a "crib sheet" by answering all (or some) of the questions in the "review" handout (on Chapters 1, 2, and 3). Your "crib sheet" should be no more than 4 pages long (including the questions).

4. (35 points) In this problem you will apply your experience with APL to design a programming language to manipulate character strings. Your task is to design the datatypes "char" and "string" and the built-in operations to go with them. As a basis for your language design, use the language of chapter 1; that is, you should not re-think the basic syntax, numbers, or control structures from the chapter 1 language, but should include them unchanged. For the new parts of the language design, try to make as many analogies to APL as you can. (But do not include APL's vectors of numbers or matrices as part of the language.) For example, think of strings as vectors of characters. To summarize, you are to design an "APL for strings".

   Hint: try a few example programs before settling on details of your design.

   To present your design, follow the style of presentation found in Kamin's section 3.1 on APL. That is, define some terminology to talk about characters and strings, define the operators, and give a few simple examples.

5. (extra credit) Build an interpreter for your string language from the previous problem.

6. (12 points) Do problem 1, parts (c), (d) and (e) on page 148 of Kamin's book. Note that you are to do this using mapcar, mapc, or combine, and the answers you define.

7. (7 points) Define a function `iterate` such that `((iterate f) 0)` is the identity function and `((iterate f) k)` equals `(lambda (x) (f (((iterate f) (- k 1)) x))))` for `k` > 0. The argument to `iterate` is a function (`f`, which takes one argument), and `iterate` returns a function that takes a number (`k`), as an argument and returns a function. For example:

   ```
   (((iterate (lambda (x) (+ 1 x))) 3) 5)
   ```

   returns 8.

8. (10 points) Do problem 3 on page 149 of Kamin's book (infinite sets).

9. (10 points) Do problem 7 on page 150 of Kamin's book (dynamic scoping).

10. (extra credit only) Do problem 4 on page 149 of Kamin's book (continuations). Note that this exercise has a typo: the occurrence of `(gcd 3 5)` should be `(gcd 9 11)`.

11. (extra credit only) Do problem 5 on page 149 of Kamin's book (term rewriting).

12. (extra credit only) Do (one or more of) the problems 10, 12, or 15 in section 4.10.2 of Kamin's book. You will find the source for the scheme interpreter in

    ```
    /home/cs342/public/src/chap.4/scheme.p
    ```

    Make a copy using the Unix `cp` command and then edit your copy. Turn in differences (using the Unix command `diff`) as before.