CS61A – Homework 1.1

Kurt Meinz
University of California, Berkeley
Summer 2003

Topic: Functional programming

Lectures: Monday June 23, Tuesday June 24

Reading: Abelson & Sussman, Section 1.1

In this assignment you'll write simple recursive programs to manipulate words and sentences, as well as explore what makes special forms special. You should use the functions `sentence`, `first`, `butfirst`, `last` and `butlast` presented in lecture to operate on words and sentences. These functions are not discussed in the book. If you have taken CS3 and know about higher-order procedures such as `every`, please do not use them; use explicit recursion.

This homework is due at **8 PM on Sunday, June 29**. Please put your answers into a file called `hw1-1.scm` and submit it electronically by typing `submit hw1-1` in the directory where the file is located. You will probably find it convenient to make a new directory (folder) for every week of the course and store the associated labs and homeworks in it; to create a folder called `week1` type `mkdir week1` at the Unix prompt. We understand that many of you have never used Unix before and will be struggling to find your way around. If you run into problems submitting the homework electronically don't freak out. We'll be quite lenient the first time around. Get your TA to help you submit on Monday. In subsequent weeks, we expect you to have mastered the online submission process.

Be sure to test each function you write; the sample calls given here do not guarantee your code is bug-free. Include your test cases in your submission, but make sure to comment them out (the semicolon character begins a one-line comment in Scheme) so the file loads smoothly. Unless explicitly disallowed, you may always write helper procedures.

One final note: Please ensure that your submitted `.scm` file loads into STk via the `(load "hw1-1.scm")` command smoothly. **Submissions that cause errors on loading may lose points.**

Question 1. Write a procedure `scale` that takes two arguments: a number n and a sentence of numbers. It should multiply each number in the sentence by n and return a sentence of the results:

```
STk> (scale 5 (se 1 2 0 10))
(5 10 0 50)
```

Question 2. Write a procedure `ends-e` that takes a sentence as its argument and returns a sentence containing only those words of the argument whose last letter is "e":

```
STk> (ends-e '(please put the salami above the blue elephant))
(please the above the blue)
STk> (ends-e '(absolutely nothing))
()
```

Question 3. Write a procedure `reverse` which reverses a sentence:

```
STk> (reverse '(the matrix cannot tell you who you are))
(are you who you tell cannot matrix the)
STk> (reverse '(kurt alex greg carolen))
(carolen greg alex kurt)
```

The adventure continues on the next page.

Question 4. Write a predicate `decreasing?` that takes a **non-empty** sentence of numbers as its argument. It should return a true value if the numbers are in strictly decreasing order and a false value otherwise:

```
STk> (decreasing? '(10 9 8 -2))
#t
STk> (decreasing? '(5 5 4))
#f
STk> (decreasing? '(17))
#t
```

Question 5. This question concerns special forms.

A. Most versions of Lisp provide `and` and `or` procedures like the ones described on Page 19 of the book. In principle there is no reason why these can't be ordinary procedures, but some versions of Lisp make them special forms. Suppose we evaluate:

```
STk> (or (= x 0) (= y 0) (= z 0))
```

If `or` is an ordinary procedure, all three argument expressions will be evaluated when `or` is invoked. But if the variable `x` has the value `0`, we know that `or` should return true regardless of the values of `y` and `z`. There is no reason to evaluate the other two expressions! A Lisp interpreter in which `or` is a special form can evaluate the arguments one by one until either a true one is found or it runs out of arguments. (This is called *short-circuit* evaluation.)

Devise a test that will determine whether Scheme's `and` and `or` are a short-circuiting special forms or ordinary functions. That is, do `and` and `or` evaluate all their arguments all the time or do they stop as soon as they know the correct value to return?

B. Scheme has two special forms for making choices, `cond` and `if`. Is it possible to define one in terms of the other? Specifically, say we attempt to define our own `if` procedure:

```
STk> (define (my-if predicate consequent alternative)
         (cond (predicate consequent)
               (else alternative)))
```

Let's take it out for a spin:

```
STk> (my-if (= 5 6) 'yes 'no)
no
```

It seems to work, so try something more interesting:

```
STk> (define (my-factorial n)
         (my-if (= n 0)
                1
                (* n (my-factorial (- n 1)))))
```

What happens when you attempt to use `my-factorial`? Why?