Fundamentals of Computer Science I (CS151.01 2006F)

# Homework 7: Evaluating Expressions

Assigned: Tuesday, 19 September 2006
Due: Friday, 22 September 2006
*No extensions!*

**Summary:** In this assignment, you will further explore the evaluation of expressions in Scheme.

**Purposes:** To reinforce Scheme's evaluation strategy. To give you experience writing recursive procedures. To raise the question of how to implement a language.

**Expected Time:** One to two hours.

**Collaboration:** You may work in a group of any size between two and four, inclusive. You may consult others outside your group, provided you cite those others. You need only submit one assignment per group.

**Submitting:** Email me your work, using a subject of *CSC151 Homework 7*.

**Warning:** So that this exercise is a learning assignment for everyone, I may spend class time publicly critiquing your work.

## Background: Evaluating Expressions

As you may recall from our discussion in class, Scheme uses a straightforward strategy for evaluating most expressions:

- If the expression is a basic value, return that value.
- If the expression is a procedure application, evaluate the parameters to the procedure and then apply the procedure.
  - To apply a built-in procedure, follow the internal instructions.
  - To apply a user-defined procedure, substitute the actual parameters for the formal parameters in the body, and then evaluate the body.

You may have also observed that procedure applications look remarkably like lists. For example,

```
(+ 2 3)
```

might represent "apply the procedure + to the values 2 and 3" or it may be a list with elements +, 2, and 3. Similarly,

```
(* (+ 2 a) (+ 3 b))
```

might be the product of two sums, or it may be a list with two sublists.

The close ties between lists and general expressions is intentional. In fact, behind the scenes, Scheme represents expressions as lists and has an internal procedure that evaluates them using the algorithm above.

In this assignment, you will build a simple version of that algorithm.

## Assignment

Consider a Scheme-like language for building strings, which we'll call `Streme`. In Streme, the basic values are all strings (things surrounded by quotation marks). The valid operations on strings are

- `(first `*str*`)`, which extracts the first letter of *str* as a string;
- `(last `*str*`)`, which extracts the last letter of *str* as a string;
- `(remove-first `*str*`)`, which returns a string that corresponds to all but the first letter of *str*;
- `(remove-last `*str*`)`, which returns a string that corresponds to all but the last letter of *str*;
- `(reverse `*str*`)`, which reverses the letters in the string;
- `(alphabetically-first `*str1*` `*str2*`)`, which returns the alphabetically first of *str1* and *str2*;
- `(longer-string `*str1*` `*str2*`)`, which returns the longer of the two strings; and
- `(join `*str1*` `*str2*`)`, which joins the two strings together into a new string.

Your goal in this assignment is to write three related procedures, `(streme-eval `*str-exp*`)`, `(streme-apply-unary `*unary-function*` `*str*`)`, and `(streme-apply-binary `*binary-function*` `*str1*` `*str2*`)`, that we can use to evaluate Streme expressions.

The `streme-eval` procedure should evaluate its argument using a modified version of the technique discussed above. (That is, determine whether `str-exp` is a string or a procedure application. In the latter case, evaluate the arguments and then apply the procedure.)

In particular, `streme-eval` should look something like the following.

```
(define streme-eval
  (lambda (str-exp)
    (cond
      ((string? str-exp) str-exp)
      ((procedure-call-with-one-parameter? str-exp)
       (streme-apply-unary (car str-exp)
                           (streme-eval (cadr str-exp))))
      ((procedure-call-with-two-parameters? str-exp)
       ...)
      (else #f))))
```

You will, of course, have to fill in the ellipses and the definitions of the new predicates. Note that the correspondence between lists and expressions suggests that a procedure call with one parameter is simply a list of two elements (the procedure and the expression to which to apply the procedure).

```
(define procedure-call-with-one-parameter?
  (lambda (str-exp)
    (and (list? str-exp)
         (= (length str-exp) 2))))
```

As the code above sugests, the `streme-apply-unary` and `streme-apply-binary` procedures should give the instructions necessary to apply the given procedure to one string or two strings, respectively.

The `streme-apply-unary` should look something like the following.

```
(define streme-apply-unary
  (lambda (proc str)
    (cond
      ((eq? proc 'first) (substring str 0 1))
      ...
      (else #f))))
```

# Practica

Here are some sample sessions with our Streme interpreter.

```
> (define sample1 (list 'first "Hello"))
> sample1
(first "Hello")
> (streme-eval sample1)
"H"
> (define sample2 (list 'remove-first (list 'remove-first "Hello")))
> sample2
(remove-first (remove-first "Hello"))
> (streme-eval sample2)
"llo"
> (define sample3 (list 'remove-first (list 'remove-last "Hello")))
> sample3
(remove-first (remove-last "Hello"))
> (streme-eval sample3)
"ell"
> (define sample4 (list 'join (list 'first "Joy") (list 'remove-first "Hello")))
> sample4
(join (first "Joy") (remove-first "Hello"))
> (streme-eval sample4)
"Jello"
> (define sample5 '(join (reverse "Hello") (remove-first "Hello")))
> sample5
(join (reverse "Hello") (remove-first "Hello"))
> (streme-eval sample5)
"olleHello"
```

As the last example suggests, when testing it is okay if you build your "expressions" using quote. (In general, I prefer that you not build lists with quote, but here, it's clearly more natural.) Just be sure not to use quotes anywhere within the expressions.

## Approaching the Problem

This problem is a bit bigger than the other work you've done so far, but it is managable if you break it up into pieces.

- Review the reading on strings or the Scheme manual to remind yourself of the procedures available for manipulating strings. (In particular, you may need to use `string-length`, `substring`, `string->list`, `list->string`, and `string-ci<=?`.)
- Implement one "built-in" procedure at a time and test it to be sure that it works as expected.
  - Also test to be sure that recursive evaluation works as expected (e.g., as in `sample2`).

We'd recommend that you try implementing the procedures in the order given above (`first`, `last`, `remove-first`, `remove-last`, `reverse`, `alphabetically-first`, `longer-string`, and `join`).

## Important Evaluation Criteria

As has been the case in all recent assignments, my primary evaluation criteria will be correctness, layout, and elegance.

---