CSE 505 Fall 2015

Assignment 4: Higher-Order Functions and Advanced Control

Assigned: Mon, Nov 9, 2015

Due: Mon, Nov 23 (11:59 pm)

Note: This assignment may be done by a pair of students.

Problem 1. Consider the following function definition in the C programming language for carrying out the summation expressed by the operator ∑ discussed in Lecture 16:

```
int sigma(int *k, int low, int high, int expr()) {
    int sum = 0;
    for (*k=low; *k<=high; (*k)++) {
        sum = sum + expr();
    }
    return sum;
}
```

Write a `main` program in the C programming language that makes repeated use of `sigma` in order to compute and print out the value of the following expression:

$$\sum_{i=0}^{4} \left(i * \sum_{j=0}^{4} \left((i+j) * \sum_{k=0}^{4} (j*k - i) \right) \right)$$

Place the code for `sigma` and `main` in a file **sigma.c**. Use the gcc compiler for testing your code.

Problem 2. Lecture 14 showed an ML function `flatten: 'a list list → 'a list`, which takes a two-level list, say ll, and returns a single-level list by appending together all the sub-lists in ll. For example, `flatten([[1],[2,3],[4],[]]) = [1,2,3,4]`.

a. Why could we ***not*** write in ML a function that would flatten an input list with an arbitrary number of levels: 2-level list, 3-level list, etc.? That is, in addition to the example shown above, we would also like to have an example such as:

   `flatten([[[1],[2]],[[3]],[[4,5],[6]]]) = [1,2,3,4,5,6]`

b. Show how we can write a general `flatten` function using a **Python generator**. This generator should yield the values in a multi-level list one by one. Then, we can create a single-level list from any multi-level list, l, by executing: `[x for x in flatten(l)]`.

Create a single file**, flatten.py**, containing the definition of `flatten`, as well as your answer to part a.

Problem 3. Consider an infinite list of strings of the form:

```
"Lf.Lx.(f x)"
"Lf.Lx.(f (f x))"
"Lf.Lx.(f (f (f x)))"
"Lf.Lx.(f (f (f (f x))))"
          …
```

These strings represent the numbers 1, 2, 3, 4, … in the pure lambda-calculus. Here, L stands for λ. Each string is called a *Church numeral* – in honor of Alonzo Church who invented the λ-calculus.

Referring to the **infinite list** ML type discussed in Lecture 17:

```
datatype 'a inf_list = lcons of 'a * (unit -> 'a inf_list)
```

Define a function `church`: `string -> string inf_list` which generates an infinite list of Church numerals starting from 1. Test out `church` by executing the following "main" program:

```
fun take(0, _) = []
  | take(n, lcons(h, thk)) = h :: take(n-1, thk());

take(5,church("x"))
```

Create a file called **church.sml** with all relevant definitions.

Problem 4. Give a formal definition for a function *LI* which defines the **leftmost-innermost redex** of a lambda-term, along the lines of the substitution operation given in the notes on Lambda Calculus (pages 5-6). If there is no redex in the input term, *LI* should return ⊥, which stands for "undefined". Examples:

LI x = ⊥	*LI* λx.(a x) = λx.(a x)
LI λx.y = ⊥	*LI* (λx.y a) = (λx.y a)
LI λx.(x x) = ⊥	*LI* λx.(λy.y x) = (λy.y x)
LI (a b) = ⊥	*LI* ((a b) (c (λz.z b))) = (λz.z b)
LI ((a b) (c d)) = ⊥	*LI* ((λz.z a) (λz.z b)) = (λz.z a)

…

The definition of *LI* involves about 8 rules. Two of the rules to help you get started are:

LI *V* = ⊥
LI *λV.T* = *LI* *T*, *if LI T* ≠ ⊥

Important: The LHS of each of these rules must be mutually-exclusive of other rules.

You may use the variable *V* to stand for any variable and *T, T1, T2* for arbitrary lambda-terms. In addition to the standard *occurs_free_in* test, you may also use *is_etaredex(T)* and *is_betaredex(T)* to test whether *T* is an eta- or a beta-redex, respectively. Place your solution in a file called **lambda.pdf**.

***What to Submit**:* Prepare a top-level directory named *A4_UBITId1_UBITId2* if the assignment is done by two students; otherwise, name it as *A4_UBITId* if the assignment is done solo. (Order the *UBITId*'s in alphabetic order, in the former case.) In this directory, place **sigma.c, flatten.py, church.sml,** and **lambda.pdf**. Compress the directory and submit the resulting compressed file using the `submit_cse505` command. For more details regarding online submission, see `Resources` → Homeworks → `Online_Submission_2015.pdf`.

End of Assignment #4