Homework 1: Lectures 1 – 4

CS 440: Programming Languages and Translators, Fall 2020

Due Sat Sep 26, 11:59 pm

9/18 p.1

How to Work; How to Submit

You'll be working in groups of 3, either self-formed or randomly-assigned. (We'll discuss details on Piazza.) Each group should submit only one copy of their answers, as a *.ocaml file, to Blackboard. Include the names and A-ids of all three members in a comment at the top of your *.ocaml file. There will be a separate work report; again, we'll discuss details on Piazza.

Problems [100 points]

Basic Syntax

1. [12 = 4 * 3 points]. Repair the syntax of the expressions below so that they calculate their intended values.
   
   a. \( \sin \sin 0.0 \) (* Should be 0.0 *)
   
   b. \( \cos -1 \) (* Should be \( \approx 0.54 \) *)
   
   c. let \( \pi = 2 \times \cos 0 \) in \( \sin \cos \pi \) (* Should be \( \approx -0.84... \) *)
   
   d. let \( f g x = f (g x) \) in \( \sqrt{(\%) f g x = f (g x)} \), \( \sqrt{16.0} \) (* Should be 2.0 *)

2. [4 points]. Enclosing an infix operator in parentheses gives us the prefix version of the operator. E.g, \( (+) x y \) means \( x + y \). (A special case is that you need the spaces in \( (\times) \) because otherwise it looks like you're writing a constant.) Rewrite the expression below to use (only) prefix operators. (Rearrange operators and manipulate parentheses but don't change order of the variables \( a, b, \ldots \).)

\[ ((a + b) \times c)/(d * (e \times f)) + x - y - z \]

3. [8 = 4*2 points]. The expression \( (((2) + 3) \times 4) \times 4) \) has too many parentheses: Some are redundant and some are just flat out bad. Removing the extra / bad parens gives us \( (2 + 3) \times 4 \), which is legal. For each of the expressions below, repeat this process: Remove any bad or redundant parentheses to get a legal expression (but don't add or remove any other symbols).

   a. \( ((\cos (\sqrt{2.5})) +. )((\sin (1.5))) \times (2.0) \) (* Should equal about 1.98 *)
   
   b. \( ((1 :: [[3 ; 4]]) :: [[[5]]] [[6]]) \) (* Should be an int list list *)
   
   c. \( [[[17]]] :: [[[][[]]]] \) (* Should be an int list list list [9/16] *)
   
   d. \( (((()) :: ((((); ())) \) \) \) \) (* Should be a unit list *)

CS Dept., Illinois Institute of Technology - 1 - © J. Sasaki 2020
List Manipulation

4. [6 points]. Complete the following definition of function `stutter n x` so that it returns a list of length `n` where each element is `x`. E.g., `stutter 3 5 = [5; 5; 5]`. If `n ≤ 0`, `stutter n x` is the empty list.

   ```ocaml
   let rec stutter n x = if ...
   ```

5. [7 points]. Write a function `last 'a list -> 'a` that returns the last element of a list or raises `Failure` with the message `Empty list` if the list is empty. Use pattern matching (not `List.hd`).

6. [8 points]. Write a function `evenly_divides k [v1; v2; ...]` (where `k` and the `v`s are integers) that returns the sublist of `v`s that are evenly divisible by `k`. E.g., `evenly_divides 3 [1; 2; 3; 4; 5; 6; 9; 2; -1]` should return `[3; 6; 9]`. (Keep the relative order of the numbers: `[9; 6; 3]` is wrong.)

Algebraic Datatypes

7. [10 = 3+7 points]. Here's a binary tree where polymorphic values appear on the leafs and nodes:

   ```ocaml
   type 'a tree = L of 'a | N of ('a * 'a tree * 'a tree)
   ```

   a. Write a function `top` so that `top` of a leaf is the value of the leaf and `top` of a node is the value attached to the node.

   b. Write a function `is_heap` that returns `true` iff a tree is a heap: Either it's a leaf, or it's a node whose subtrees are heaps and the node's value is ≥ than the tops of the subtrees. (In other words, the node value is ≥ than all the values in the node's subtrees).

Pattern Matching

8. [25 points total]. The function `twice` should take a list and return `true` iff some value occurs twice in the list. E.g., `twice` should return `true` for `[2; 2], [1; 2; 1], [1; 1; 2], and [1; 2; 2]; it should return `false` for `[]`, `[1], [1; 2], and [1; 2; 3].

   a. [3 points.] What is the type of `twice`? (Make it polymorphic.)

   b. [10 points]. Add comments to the code below to describe its bugs. (For example, the first line is at least missing the `rec` keyword.) For syntactic errors, don't just parrot the error messages; give a brief human-understandable description. For a missing case, add a comment line that gives an example of the computation that was missing. If the line is correct, write `(* None *)`.

   ```ocaml
   let twice = function ->
   [ ] = false
   | [ ] -> false
   | [x; x] -> true
   | [node1; node2; _] = node1 = node2
   | ( _ :: Tail) = twice Tail
   (* Add a line for each missing case (if any) .... *)
   ```
c. [4 points]. Write a corrected version of the `twice` function. Keep using definition by cases; feel free to add/change/delete cases as you see fit.

d. [4 points]. Change your definition from part c so that you have only two cases: one recursive, one not.

e. [4 points]. Change your definition from part d to use a `match` expression:
   
   ```ml
   let rec twice x = match x with ...
   ```

**Currying and Uncurrying**

9. [10 = 2*5 points]

   a. Write the definition for a function `uncurry2` so that if we `let mult = uncurry2 ( *)`,\(^1\) then `mult(3, 5) = 3 * 5`. Also, give the (polymorphic) type of `uncurry2`.

   b. Write the definition for a function `curry2` so that `curry2 mult 3 5` equals `3 * 5`. Also give the type of `curry2`.

**Unnamed Functions (= Lambda Functions)**

10. [10 = 4 + 6 points].

   a. Briefly, why does `let var = unnamed function` illustrate the concept of first-class functions?

   b. Rewrite `let f g x y = g x (y x) three ways, as`

   ```ml
   let f1 g x = unnamed function
   let f2 g = unnamed function
   let f3 = unnamed function.
   ```

   Note all the `f's should have the same type, `(a -> 'b -> 'c) -> 'a -> (a -> 'b) -> 'c`, and they should all behave identically.

---

\(^1\) Don't forget, you need extra parens around the `*` in `(*)` otherwise it looks like a comment.