CSCI 2041: Advanced Programming Principles
First Mid-Term Exam
Feb 23, 2015, 13:25 - 14:15 p.m.

Before doing anything else, please put aside all notes, books and electronic devices. This is a **closed book** exam. Also, please turn off your cell phones if you have them with you in class.

There are six questions in this exam all of which must be answered for full credit. The exam paper consists of 10 pages. Before reading any further, please make sure you have all of the pages.

Before you begin to write anything, please print your name and university id (i.e. your university email address) legibly in the space provided below. Please also have your student id ready for inspection during the exam and as you turn in your exam paper.

Name:

University Id:

Now, here are some guidelines that you should follow in doing the exam:

- Read all the questions through first to decide on a preferred order for doing them in.
- *Write your answers to the questions in the space provided in the exam paper.* Typically, you should not need more space. However, if you do need extra space
  - continue your work into the extra sheet at the end of the exam,
  - indicate under the problem that you are continuing the answer onto the extra space, and
  - in the extra space at the end, indicate *clearly* each problem that you are continuing there.
- Use only one side of each sheet, i.e., *do not write on the back of any page.*
- Use a number 2 pencil or pen for writing your answers. Also, write legibly—we cannot grade what we cannot read.
- Where you are asked to explain the reasons for your answer, note that the explanation is *crucial*; you will get very little credit without the explanation.

Good luck in the exam!

Problem 1. *(7 points each, 21 points in all)*

For each of the expressions below, indicate whether or not OCaml will accept it without pointing out errors. If you say it will be accepted without error, indicate what value will be returned for it. If you say that OCaml will indcate an error for it, explain what the error will be. *Assume that there are no let declarations before the expression is presented to OCaml.*

1. `let x = 5 in let y = 7 in x + y`

2. `let x = (let y = " world" in "hello" ^ y) in x ^ y`
   In case you are wondering about this, `^` is the string concatenation operator that is written in infix form in OCaml.

3. `fun y -> let x = 3 in x + y`

Problem 2. *(7 points each, 21 points in all)*

For each of the expressions below, indicate if they are well-typed. If they are well-typed, indicate their values and their types. If they are not well-typed, explain why not.

1. `let id x = x in if (3 < 7) then 5 else (id "hello")`

2.
```
match [1;2;3] with
  | [] -> 0
  | a::b::c::d -> a + c
  | a::b::c -> b
```

3.
```
match [(1,2); (3,4,5)] with
  | [] -> 0
  | (f,s)::t -> f + s
```

Problem 3 *(10 points)*

Suppose that OCaml has been presented with the following collection of let declarations:

```
let z = 5
let addz x = z + x
let z = 7
let whatisz x = let z = 3 in addz x
```

What will be the result of evaluating `(whatisz 10)` after these declarations? Explain your answer. Note that the explanation will carry the most credit.

Problem 4 *(9 points each, 18 points in all)*

For each of the function definitions below

- explicitly follow the process that was explained in connection with the definition of `append` in class to determine if the following function definitions are type correct, and
- at the end of it, either conclude that the definition is not type correct or show the type that would be inferred for the function.

You must show your work for the first of the items above for credit in this problem.

1. ```
   let rec f (a,b,n) =
       match n with
       | 0 -> a
       | n -> if b then a else a * f(a,b,n-1)
   ```
   ```

2.
```
let rec repeatA a n =
      match n with
      | 0 -> a
      | n -> (a, repeatA a (n-1))
```

Problem 5 *(15 points)*

Recall the following type from class that allows us to represent both integer and string values:

```
type intorstr = Int of int | Str of string
```

Define a function

```
separate : intorstr list -> (int list) * (string list)
```

that takes a list of `intorstr` items and separates them into two lists, one containing the integers in the list and the other containing the strings in the list. The following interaction illustrates the use of this function:

```
# separate [(Int 5);(Str "foo");(Str "bar");(Int 7);(Int 3);(Str "baz")];;
- : int list * string list = ([5; 7; 3], ["foo"; "bar"; "baz"])
# separate [(Int 5);(Int 7);(Int 3);];;
- : int list * string list = ([5; 7; 3], [])
# separate [(Str "foo");(Str "bar");(Str "baz")];;
- : int list * string list = ([], ["foo"; "bar"; "baz"])
# # separate [];;
- : int list * string list = ([], [])
```

Problem 6 *(15 points)*

Consider the following pruned down version of the `expr` type that we considered in class:

```
type  expr =
  Id of string | Int of int
| Plus of expr * expr | Minus of expr * expr
```

Define a function

```
   subst : expr -> string -> expr -> expr
```

that takes an expression, the name of an identifier and an expression to replace it with and returns an expression that is obtained by replacing all occurrences of the identifier in the first expression by the second one. The following interaction exemplifies the behaviour that is required of `subst`:

```
# subst (Plus (Int 5, Id "x")) "x" (Plus (Int 3,Int 4));;
- : expr = Plus (Int 5, Plus (Int 3, Int 4))
# subst (Plus (Int 5, Id "y")) "x" (Plus (Int 3,Int 4));;
- : expr = Plus (Int 5, Id "y")
# subst (Plus (Id "x", Minus (Id "x", Int 5))) "x" (Int 7);;
- : expr = Plus (Int 7, Minus (Int 7, Int 5))
#
```

Continue your answer onto this page if the space provided below some question is not sufficient for your answer. *Make sure to indicate clearly which answer is being continued here.*

This additional page has been provided as scratch area