CS 421 Spring 2013 Midterm 1

Monday, February 18, 2013

Name	
NetID	

- You have **70 minutes** to complete this exam
- This is a **closed book** exam.
- Do not share anything with other students. Do not talk to other students. Do not look at another student's exam. Do not expose your exam to easy viewing by other students. Violation of any of these rules will count as cheating.
- If you believe there is an error, or an ambiguous question, seek clarification from one of the proctors. You must use a whisper, or write your question out.
- Including this cover sheet, there are eight pages to the exam. Please verify that you have all eight pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Value**	**Score**
1	15	
2	15	
3	15	
4	15	
5	15	
6	15	
7	10	
Total	**100**	

1. (15 pts) Write the following functions.

   (a) (5 pts) `keep_greater`: α list $\to \alpha$ list $\to \alpha$ list takes two lists of the same length. It compares the corresponding elements and returns the elements of the first list that are greater than the corresponding element in the second, e.g.: `keep_greater [10;20;30;40;50] [31;9;43;37;50]` = `[20;40]`. (You should not use any auxiliary functions.)

```
let rec keep_greater lis1 lis2 = match (lis1, lis2) with
      ([], []) -> []
    |
```

   (b) (5 pts) Given two lists of the same length, `zip` constructs a list of pairs: `zip` `[`$x_1, \ldots, x_n$`]` `[`$y_1, \ldots, y_n$`]` = `[(`x_1, y_1`); ...; (`x_n, y_n`)]`. Give the (polymorphic) *type* of `zip` (not its definition), including both argument and result types:

```
zip :
```

   (c) (5 pts) Fill in this definition to give an alternative definition of `keep_greater`:

```
let keep_greater lis1 lis2 =
    let rec aux_fun lis = match lis with
          []  ->  []
        |
```

```
    in aux_fun (zip lis1 lis2)
```

2. (15 pts) In this question, you will define a *dictionary* type — a mapping from strings to integers — using binary search trees. Define the type like this:

```
type dictionary = Node of string * int * dictionary * dictionary
                | Null
```

Here, `Null` represents an empty dictionary; leaf nodes are represented by terms of the form `Node(`s`, `i`, Null, Null)`.

Dictionaries are binary search trees — that is, given a node `Node(`s`, `i`, `t`, `t'`)`, every string occurring in t is less than s and every string in t' is greater than s (using OCaml's definition of $<$ on strings).

(a) (7 pts) Define `get`: `string` → `dictionary` → `int`. (No auxiliary functions should be used.)

```
let rec get str dict = match dict with
    Null -> raise NotInDictionaryException

  | Node(s,i,d,d') ->
```

(b) (8 pts) Define `add`: `string` → `int` → `dictionary` → `dictionary`, so as to preserve the search tree property. *You may assume that the string is not yet in the dictionary.*

```
let rec add str i dict = match dict with

    Null ->


  | Node(s,i',d,d') ->
```

3. (15 pts) Java has four types of *integer literals*:

   - Decimal: Single digit zero, or a non-zero digit followed by zero or more digits.
   - Octal: Zero followed by one or more octal digits (0–7).
   - Hex: Zero followed by `x` followed by one or more hex digits (0-9, a-f).
   - Binary: Zero followed by `b` followed by one or more binary digits (0 and 1).

   (a) (7 pts) Write a single DFA for all four types of integer literals. As we did in class, the states should be labelled either Start, Error, or one of the token types (Int, Octal, Hex, Binary). (The Discard label isn't needed here.) Hint: our solution has a total of eight states.

   (b) (8 pts) Write separate regular expressions, in ocamllex notation, for each of the four types of literals:

   - Decimal:
   - Octal:
   - Hex:
   - Binary:

4. (15 pts) This is an abstract syntax for part of OCaml, similar to the one discussed in lecture 4:

```
type exp = Integer of int | Var of string | Add of exp * exp | Let of def * exp
 and def = Defn of string * exp
```

Define `eval`: `exp` → `dictionary` → `int`, where `dictionary` is the type defined in question 2. Here is an example, using the abstract syntax for expression "`let a=4 in a+x`":

```
# let dict1 = add "x" 5 Null;;
# eval (Let(Defn("a", Integer 4), Add(Var "a", Var "x"))) dict1;;
- : int = 9
```

In the call `eval` *e dict*, you can assume any *free* variables in *e* — that is, variables not defined in a `let` expression inside *e* — are defined in *dict*. (In the example just given, `x` is a free variable — so it is given in the dictionary — but `a` is not.)

The clause for `Let` should call auxiliary function `evaldef`: `def` → `dictionary` → `dictionary`. `evaldef` adds the value of an expression to a dictionary; that is `evaldef (Defn(s,e)) dict` returns a dictionary that is the same as `dict` except the value of `e` has been bound to `s`.

You can use `get` and `add` from problem 2 in your solution.

```
let rec eval e dict = match e with

    Integer i ->


  | Var s ->


  | Add(e1, e2) ->



  | Let(d, e1) ->



and evaldef (Defn(s,e)) dict =
```

5. (15 pts) Here is a concrete syntax for the same part of OCaml:

   $E \rightarrow$ `let` D `in` E | $E + E$ | `int` | `string` | `(` E `)`
   $D \rightarrow$ `string =` E

   (a) (6 pts) Give the parse tree for the sentence `let x=10 in x`.

   (b) (9 pts) Give the shift-reduce parse for that sentence. To reduce the amount of writing, just use "S" for shift and "R" for reduce; you do not need to write the production you use. (Hint: it has 11 steps.)

Action	**Stack**	**Input**
Shift		`let x=10 in x`
Accept	E	eof

6. (15 pts) The OCaml grammar in the last problem is not LL(1) because it is ambiguous and also uses left-recursion. This is a version that is unambiguous, and doesn't use left-recursion:

   $E \rightarrow$ `let` D `in` E | F
   $F \rightarrow G$ `+` F | G
   $G \rightarrow$ `int` | `string` | `(` E `)`
   $D \rightarrow$ `string =` E

   (a) (3 pts) However, this grammar is still not LL(1) because the rules for F need to be left-factored. Write the left-factored version of the rules for F (and just those; the others don't need to be changed):

   (b) (12 pts) Write a recursive-descent recognizer for this grammar. You can assume `parseF` and `parseD` are already done; you have to write `parseE` and `parseG`. Recall that each of the functions has type `token list` → `token list`. You should write `raise SyntaxError` when you detect a syntax error. Here is the list of tokens:

```
type token = LetKW | InKW | Plus | LParen | RParen | Eq
           | Int of int | Id of string | EOF

let rec parseE toklis = match toklis with




and parseG toklis = match toklis with




and parseF toklis = (* consider this one done *)
and parseD toklis = (* consider this one done *)
```

7. (10 pts) In lecture 10, we introduced type judgments for expressions and statements:

$\pi,\ \Gamma \vdash S$	S is a type-correct statement, where Γ gives all variable declarations surrounding S (fields, parameters, locals)
$\pi,\ \Gamma \vdash e : \tau$	e is a type-correct expression of type τ, where Γ gives the types of any variables occurring in e.

Fill in the missing lines for these inference rules:

(a) $\pi, \Gamma \vdash e_1 + e_2 :$ int

_____

_____

(b) $\pi, \Gamma \vdash x$ `=` e`;`

_____

(c) $\pi, \Gamma \vdash$ `if (`e`)` S_1 `else` S_2

_____

_____

_____