CS 421 Spring 2012 Midterm 1

Tuesday, February 21, 2012

Name	**Answer sheet**
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 TAs. 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	18	
2	18	
3	8	
4a	6	
4b	15	
5	15	
6	10	
7	10	
Total	**100**	

1. (18 pts) Fill in the types of these expressions and functions (some are polymorphic), or write "type error." For function definitions, give the type of the function (both argument and result types):

   (a) `(4, "4", '4')` **int * string * char**

   (b) `[3; [3]; 4; [4]]` **type error**

   (c) `let f (a,b,c) = a+b` **int * int * α → int**

   (d) `let intvals p = let (a,b) = p in [a+1; hd b]` **int * int list → int list**

2. (18 pts) Write the following functions. (You will not need, and should not use, auxiliary functions.)

   (a) `revpairs`: (α*β) list → (β*α) list reverses each pair in its argument: `revpairs [(1,2); (3,4)] = [(2,1); (4,3)]`.

   ```
   let rec revpairs lis = match lis with
        [] -> []
     | (a,b)::t -> (b,a) :: revpairs t
   ```

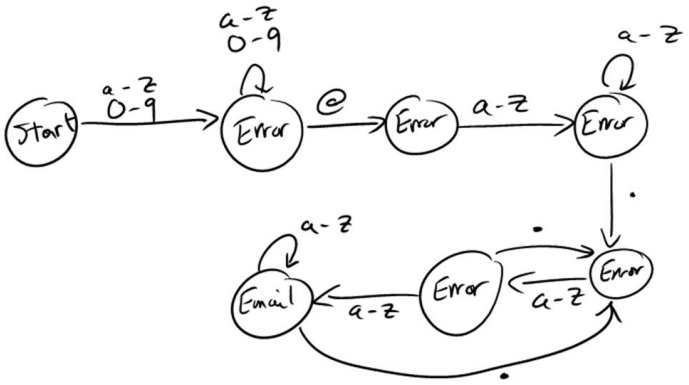
   (b) `lookup`: string → (string * α) list → α returns the value in the second argument that is associated with the first argument. Think of the second argument as a "dictionary" mapping names to values. E.g. `lookup "i" [("a", 3); ("i", 5)] = 5`.

   ```
   let rec lookup s dict = match dict with
     (a,v) :: t -> if s=a then v else lookup s t
   ```

   (c) `partition`: int list → int → (int list * int list) divides its first argument into two lists, one containing all elements less than its second argument, and the other all the elements greater than or equal to its second argument. `partition [5; 2; 10; 4] 4 = ([2], [5; 10; 4])`.

   ```
   let rec partition lis pivot = match lis with
        [] -> ([], [])
     | h::t -> let (lis1, lis2) = partition t pivot
               in if h<pivot then (h::lis1, lis2)
                             else (lis1, h::lis2)
   ```

3. (8 pts) Write a DFA to recognize a simplified form of email addresses. The username part is a non-empty sequence of lower-case letters and digits. This is followed by the customary @. The host or domain part consists of lower-case letters separated by periods. Specifically, it has at least one period, can not have two consecutive periods, can not begin or end with a period, and has at least two lower-case letters after the last period. Each state should be labelled as either *Email* or *Error*; unlike our examples in class, most of the states will be labelled *Error*. (Hint: our solution has a total of seven states.)



4. (20 pts) This is a simplified portion of the abstract syntax of MiniJava:

```
type statement = Block of (statement list)
               | While of exp * statement
               | Assignment of id * exp
and exp = Operation of exp * binary_operation * exp
               | Id of string | Integer of int
and binary_operation = Equal | LessThan | Plus
```

   (a) (6 pts) Write the expression of type `statement` representing the abstract syntax of this statement:

   ```
   while(x < 5) y = a + b;
   ```

   ```
   While(Operation(Id "x", LessThan, Integer 5),
         Assignment("y", Operation(Id "a", Plus, Id "b")));;
   ```

(b) (15 pts) Write the function `eval : exp → (string * value) list → value`, which evaluates an expression, given that the values of any variables occurring in the expression are given by the second argument (the "dictionary"). The type `value` is

```
type value = Int of int | Bool of bool
```

For example:

```
eval (Operation(Id "x", Plus, Integer 10)) [("a", Int 5), ("x", Int 7)]
 ==> Int 17
```

You will want to use the function `lookup` defined in problem 2 above. You can assume that any variables occurring in the expression occur in the dictionary, and that the expression uses all values in a type-correct way — don't worry about type errors. Specifically: `<` and `+` only apply to integers; `=` applies to two bools or two integers.

You must use auxiliary function `apply : binary_operation → value → value → value` that applies an operation to its arguments (again, assuming the arguments have the correct type). E.g. `apply Plus (Int 3) (Int 7) = Int 10`.

```
let rec eval e dict = match e with
      Operation(e1, bop, e2) -> apply bop (eval e1 dict) (eval e2 dict)
    | Id x -> lookup x dict
    | Integer i -> Int i

and apply bop v1 v2 = match bop with
      Equal -> (match (v1, v2) with
                  (Int i1, Int i2) -> Bool (i1 = i2)
                | (Bool b1, Bool b2) -> Bool (b1 = b2))
    | LessThan -> (match (v1, v2) with (Int i1, Int i2) -> Bool (i1<i2))
    | Plus -> (match (v1, v2) with (Int i1, Int i2) -> Int (i1+i2)) ;;
```

or:

```
and apply bop v1 v2 = match (v1, v2) with
      (Int i1, Int i2) -> (match bop with
                             Equal -> Bool (i1=i2)
                           | LessThan -> Bool (i1<i2)
                           | Plus -> Int (i1+i2))
    | (Bool b1, Bool b2) -> (match bop with
                             Equal -> Bool (b1=b2)) ;;
```

or:

```
and apply bop v1 v2 = match (bop, v1, v2) with
      (Equal, Int i1, Int i2) -> Bool (i1=i2)
      (Equal, Bool b1, Bool b2) -> Bool (b1=b2)
      (LessThan, Int i1, Int i2) -> Bool (i1<i2)
      (Plus, Int i1, Int i2) -> Bool (i1+i2)
```

5. (15 pts) Consider these two expression grammars:

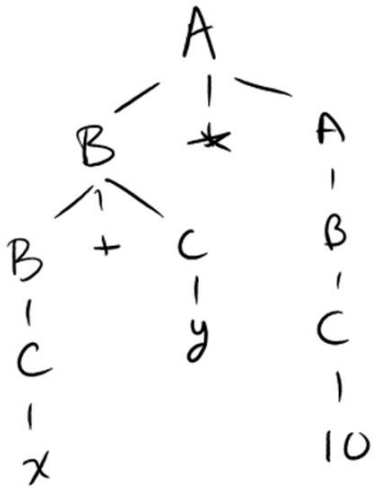
G1

$A \rightarrow B \mid B * A \mid B \ / \ A$
$B \rightarrow C \mid B + C \mid B \text{ - } C$
$C \rightarrow \text{id} \mid \text{int}$

G2

$A \rightarrow \text{id} \mid \text{int} \mid A{+}A \mid A{-}A \mid A{*}A \mid A/A$

(a) (8 pts) Draw the parse tree for `x+y*10` in G1:



(b) (6 pts) What *precedences* and *associativities* are enforced by G1, if any?

Plus and minus have precedence over Multiplication and Division.
Plus and minus are left-associative.
Multiplication and division are right-associative.

(c) (6 pts) Provide ocamlyacc precedence declarations for G2 so that the precedence and associativity of all operators is the same as those enforced by G1. (For tokens, use: `Star`, `Slash`, `Plus`, `Minus`, `Id of string`, and `Int of int`.)

```
%right Star Slash
%left Plus Minus
```

6. (10 pts) For this problem, the algorithms for computing FIRST and FOLLOW sets are copied on the last page of the exam (so you can tear it off).

G3:
$S \rightarrow A\ C\ B \mid C$ b $B \mid B$ a
$A \rightarrow$ d a $\mid B\ C$
$B \rightarrow$ g $\mid \epsilon$
$C \rightarrow$ h $\mid \epsilon$

(a) (5 pts) Perform the FIRST sets calculation on G3. (As usual, $fsts_0$ should be left blank.) We have filled in the final table (you just have to fill in the intermediate steps).

$fsts_0$:

S	
A	
B	
C	

$fsts_1$:

S	
A	**d**
B	**g, •**
C	**h, •**

$fsts_2$:

S	**d, h, g, a, b**
A	**d, g, h, •**
B	**g, •**
C	**h, •**

$fsts_3$:

S	**d, h, g, a, b, •**
A	**d, g, h, •**
B	**g, •**
C	**h, •**

$fsts_4$:

S	a, b, d, g, h, •
A	d, g, h, •
B	g, •
C	h, •

(b) (5 pts) Perform the FOLLOW sets calculation on G3. As usual, $flws_0$ is empty except that the start symbol contains eof.

$flws_0$:

S	eof
A	
B	
C	

$flws_1$:

S	**eof**
A	**h, g, eof**
B	**eof, a, h**
C	**g, eof, b**

$flws_2$:

S	**eof**
A	**h, g, eof**
B	**eof, a, h, g**
C	**g, eof, b, h**

$flws_3$:

S	**eof**
A	**h, g, eof**
B	**eof, a, h, g**
C	**g, eof, b, h**

7. (10 pts) Write a recursive-descent recognizer for the following grammar. We have provided the FIRST sets; since the grammar has no ϵ-productions, you do not need FOLLOW sets.

   $S \rightarrow$ id $=$ E | if (E) T FIRST(S) = { id, if }
   $T \rightarrow S$ | else S FIRST(T) = { id, if, else }
   $E \rightarrow$ id | int FIRST(E) = { id, int }

   Use the following tokens:

```
type token = Id of string | Int of int | Equal | LParen | RParen | If | Else
```

   As usual, raise `SyntaxError` when appropriate.

```
let rec parseS toklis = match toklis with
       Id _ :: Equal :: t -> parseE t
     | If :: LParen :: t -> (match parseE t with
                               RParen :: t' -> parseT t'
                             | _ -> raise SyntaxError)
     | _ -> raise SyntaxError

and parseT toklis = match toklis with
       Else :: t -> parseS t
     | _ -> parseS toklis

and parseE toklis = match toklis with
       Id _ :: t -> t
     | Int _ :: t -> t
     | _ -> raise SyntaxError;;
```

FIRST(G) =
    fsts_0 = empty table (i.e. maps every $A \in N$ to {})
    $i = 0$
    repeat { $i = i{+}1$; fsts_i = empty table
        for every production $A \to \alpha$ in G:
            $\mathit{fsts}_i(A) = \mathit{fsts}_i(A) \cup \mathit{RHSFirst}(\alpha, \mathit{fsts}_{i-1})$
    } until $\mathit{fsts}_i = \mathit{fsts}_{i-1}$
    return fsts_i

RHSFirst($X_1 X_2 \ldots X_n$, *fsts*) =
    if $n{=}0$ return {$\bullet$}
    else if $X_1 \in T$ return { X_1 }
    else if $\bullet \notin \mathit{fsts}[X_1]$ return $\mathit{fsts}[X_1]$
        else return $(\mathit{fsts}[X_1] - \{\bullet\}) \cup \mathit{RHSFirst}(X_2 \ldots X_n, \mathit{fsts})$

FOLLOW(G) =
    flws_0 = table mapping S to { eof }, and every other $A \in N$ to {}
    $i = 0$
    repeat {
        $i = i{+}1$; $\mathit{flws}_i = \mathit{flws}_0$
        for every $B \in N$:
            for every occurrence of B in a production $A \to \alpha B \beta$:
                $\mathit{flws}_i[B] = \mathit{flws}_i[B] \cup (\mathit{FIRST}(\beta) - \{\bullet\})$
                if $\bullet \in \mathit{FIRST}(\beta)$ then $\mathit{flws}_i[B] = \mathit{flws}_i[B] \cup \mathit{flws}_{i-1}[A]$
    } until $\mathit{flws}_i = \mathit{flws}_{i-1}$
    return flws_i