CS 421 Spring 2011 Midterm 1

Thursday, February 24, 2010

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 TAs. You must use a whisper, or write your question out.
- Including this cover sheet, there are 14 pages to the exam. Please verify that you have all 14 pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Value**	**Score**
1	10	
2	4	
3	4	
4	15	
5	6	
6	7	
7a	5	
7b	7	
7c	5	
7d	12	
8a	5	
8b, 8c	10 + 5XC	
9	10	
Total	**100 + 5**	

1. (10 pts) Immediately after each of the following declarations, what is the most general type of `f` (use type variables where necessary)?

   (a) (1pt) `let f = "asd"`

   (b) (1pt) `let f b c = b`

   (c) (1pt) `let f b z = if b then z else z +. 1.4`

   (d) (1pt) `let f (b, c) = b ^ c`

   (e) (1pt) `let f z (a, b) = match z with (x, y) -> if x = y then a else b`

   (f) (2pt)

```
let f l x = match l with
     (y::ys)::zs -> if x = y then (x::y::ys)::zs else [x]::((y::ys)::zs)
   | [] -> [[x]]
```

   (g) (3pt)

```
let rec f a b = match a with
     (y :: ys) :: xs -> if y = b then (true, ys) else f (ys :: xs) b
   | _ -> (false, [])
```

   ---

2. (4 pts) Consider the following function:

```
let g l = match l with
      x :: y :: zs -> (x = y) :: zs
    | [] -> [false]
```

What do the following expressions evaluate to? (Note that some may cause a type error or a run-time error; write "type error" or "run-time error" for your answer.)

(a) `g [1;2]`

_____

(b) `g [true]`

_____

(c) `g [false; true]`

_____

(d) `g [false; true; true]`

_____

3. (4 pts) Consider the following function:

```
let rec f l = match l with
        x :: y :: xs -> f (y :: xs)
     | [x :: xs] -> x
```

What do the following expressions evaluate to? (Again, type or run-time errors are possible.)

(a) `f [2]`

(b) `f []`

(c) `f [[]]`

(d) `f [[]; [2;3]]`

---

4. (15 pts) Recursive functions

   (a) (5 pts) Write a function `positive : int list` → `bool list` that returns a list where position i is `true` if ith element of the input is nonnegative and `false` otherwise. `positive` should return an empty list on the empty input.

   Examples:

```
# positive [1;0;-2;3];;
- : bool list = [true; false; false; true]
# positive [];;
- : bool list = []
```

   (b) (10 pts) Write `from_to : 'a list` → `int` → `-> int -> 'a list` such that `from_to l n m` returns a sublist of l that starts at the nth element (indexing from zero) and ends at the mth element. You may assume that $0 <= n <= m <$ (`List.length` l)
   Examples:

```
# from_to [0;1;2;3;4;5;6] 2 4;;
- : int list = [2; 3; 4]
# from_to [0;1;2;3;4;5;6] 5 5;;
- : int list = [5]
```

5. (6 pts) Consider the following independent inputs:

   (a) `1.23`

   (b) `1.`

   (c) `1.23e`

   (d) `1.23e4`

   (e) `1.23e+4`

   After each of the following rules written in ocamllex, choose every input that the rule accepts as a whole:

   (a) (2pts) `['0'-'9']+ '.' ['0'-'9']*`

   a b c d e

   (b) (2pts) `['0'-'9']+ '.' ['0'-'9']* ['e' 'E'] ['+' '-'] ['0'-'9']*`

   a b c d e

   (c) (2pts) `['0'-'9']+ '.' ['0'-'9']+ ( ['e' 'E'] ['+' '-']? ['0'-'9']* )?`

   a b c d e

6. (7 pts) Write a DFA to recognize a simplified form of XML document nodes. These nodes have the form: $<$*tag attr*=”*value*” ... $>$, where we will restrict attributes and values to consist solely of lower-case letters. To be precise, here is a regular expression describing in the syntax, in ocamllex format:

```
'<' ['a'-'z']+ (' '+ ['a'-'z']+ '=' '"' ['a'-'z']* '"')* ' '* '>'
```

Write a DFA to recognize an entire node. This DFA will have one start state and one or more “accept” states, and all the other states will be error states. (Hint: our solution has a total of nine states.)

7. (29 points) This multi-part question concerns a highly simplified version of OCaml, which conforms to the following syntax:

   $$\textit{Expr} \rightarrow \text{Id} \mid \text{let Id} = \textit{Expr} \text{ in } \textit{Expr} \mid \textit{Expr Expr}$$

   (a) (5 pts) That grammar is ambiguous. Give a sentence and two distinct parse trees to prove this.

   The following datatype gives an abstract syntax for expressions (which closely follows the ambiguous grammar):

```
type expr =  Id of string
           | Let of string * expr * expr
           | App of expr * expr
```

   For example, the expression:

```
let x = let y = z
        in y w
in let x = f x
   in g x
```

   corresponds to this expression:

```
let ex1 = Let("x", Let("y", Id "z", App (Id "y", Id "w")),
              Let("x", App(Id "f", Id "x"),
                  App(Id "g", Id "x")))
```

(b) (7 pts) Write a function `varsDeclared`: `expr` → `string list` that gives a list of all the variables bound in let expressions, in the order of a depth-first, left-to-right traversal.

```
# varsDeclared ex1;;
- : string list = ["x"; "y"; "x"]
```

For reference, here is the abstract syntax again:

```
type expr =  Id of string
           | Let of string * expr * expr
           | App of expr * expr
```

(c) (5 pts) Write a function rename: expr → expr that renames the let-bound variables in an expression to correctly reflect the scope; that is, it renames them so that if a variable is declared in a let that is inside another declaration of a variable with the same name, the two will have distinct names.

```
# rename ex1;;
- : expr = Let ("x1", Let ("y1", Id "z", App (Id "y1", Id "w")),
              Let ("x2", App (Id "f", Id "x1"), App (Id "g", Id "x2")))
```

To help with this problem, you will be provided with a dictionary type with one constant and two operations: emptyDict: dict, lookup: dict → string → string, and add: dict → string → dict. Dictionaries work like this: suppose `d = add emptyDict "x"`; then `lookup d "x"` will return `"x1"`; then, if `d1 = add d "x"`, `lookup d1 "x"` will return `"x2`; and so on. The idea is that every time you see a variable being bound in a let expression, add it to the dictionary and then use the name it returns wherever that variable occurs.

You actually need to write `rename1`: `dict` → `expr` → `expr`. `rename` is defined as follows:

```
let rec rename1 dict e = ...
let rename e = rename1 emptyDict e;;
```

Define `rename1`:

```
let rec rename1 dict e =
```

(d) (12 pts) The following is an unambiguous, LL(1) grammar for this subset of OCaml:

exp → Let Ident Equal *exp* In *exp* | *term*
term → Ident *term1*
term1 → Ident *exp* | ϵ

Write a recursive-descent parser for the above grammar that produces ASTs. Specifically, write `parseexp`: `token list` → `expr` * `token list`. `parseexp` takes as input a token list and returns the remaining tokens and the AST. (Technically, you need to calculate the FIRST and FOLLOW sets, but for this grammar, you can do it by inspection.) The tokens are:

```
type token = IDENT of string | LET | EQUAL | IN
```

As a reminder, here is the abstract syntax:

```
type expr =  Id of string | Let of string * expr * expr | App of expr * expr
```

```
let rec parseexp tl =
```

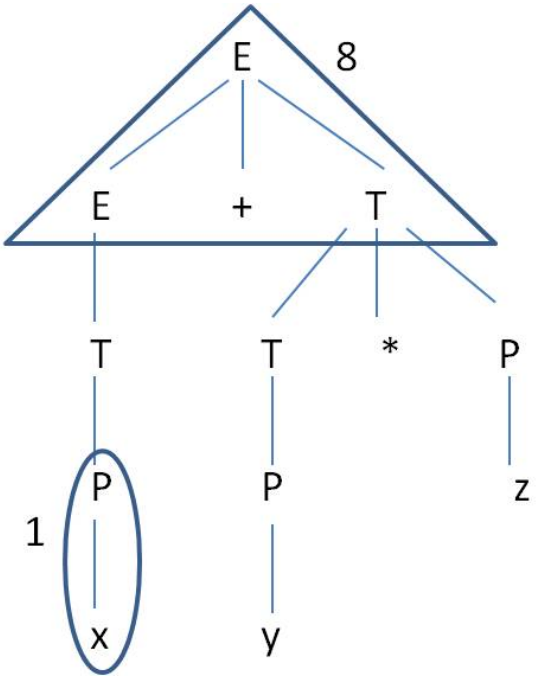
```
and parseterm tl =
```

```
and parseterm1 tl =
```

8. (15 pts) Here is a standard, left-recursive, stratified expression grammar:

$$
\begin{aligned}
E &\rightarrow T \mid E + T \\
T &\rightarrow P \mid T * P \\
P &\rightarrow \text{id}
\end{aligned}
$$

(a) (5 pts) Following is the parse tree for sentence x+y*z. Circle and number the handles in the order in which they are pruned in a shift-reduce parse. We have filled in the first and last as a hint.



(b) (10 pts) Based on the tree above, fill in the following table, giving parse actions under various combinations of stack and lookahead symbol; the actions can be *shift*, *reduce by a given production*, and *accept*. There are no *reject* actions here; we filled in the first row.

When the stack is,	and the lookahead symbol is,	the action is
empty	id	**shift**
P	+	
T	+	
E	+	
E+	id	
E+id	*	
E+P	*	
E+T	*	
E+T*	id	
E+T*id	eof	
E+T*P	eof	
E+T	eof	
E	eof	

(c) **Extra credit** (5 pts) It turns out that the grammar above has a very limited number of possible stack configurations. Fill in the actions for these other stack/lookahead configurations (here, *reject* actions are possible):

When the stack is,	and the lookahead symbol is,	the action is
empty	id	
P	*	
T	*	
E	*	
E	id	
T*	id	
T*P	id	
T*P	+	

9. (10pts) True or False (circle one):

   (a) **True** **False** OCaml is dynamically typed (because it has no variable declarations)

   (b) **True** **False** OCaml employs automatic memory management

   (c) **True** **False** Java employs automatic memory management

   (d) **True** **False** In OCaml, every element of a list must have the same type

   (e) **True** **False** In OCaml, every element of a tuple must have the same type

   (f) **True** **False** Every LALR(1) grammar is unambiguous

   (g) **True** **False** Top-down (LL(1)) grammars cannot be left-recursive.

   (h) **True** **False** Regular expressios are more powerful than DFAs — that is, some tokens can be described by a regular expression by not by a DFA.

   (i) **True** **False** In the OCaml definition "`let rec f x = e1 in e2`", the scope of `f` includes `e1`.

   (j) **True** **False** The use of `let rec` instead of `let` provides a note to the programmer that a function is recursive, but has no significance to the OCaml compiler.