CS 421 Spring 2010 Midterm 1

Wednesday, February 24, 2010

Name	
NetID	

- You have **75 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 17 pages to the exam. Please verify that you have all 17 pages.
- Please write your name and NetID in the spaces above, and at the top of every page.

Question	**Value**	**Score**
1	10	
2	10	
3	10	
4	10	
5	10	
6	10	
7	10	
8	10 + 5 XC	
9	10	
10	10	
11	10 XC	
Total	**100 + 15**	

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

   (a) (1pt) `let f = 3.4`

   `float`

   (b) (1pt) `let f x = x + 2`

   `int → int`

   (c) (1pt) `let f x y = if x then y else y *. 2.0`

   `bool → float → float`

   (d) (1pt) `let f (x, y) z = (x, y, z)`

   `'a * 'b → 'c → 'a * 'b * 'c`

   (e) (1pt) `let f (x, y) (a, b) = if x = a then y else b`

   `'a * 'b → 'a * 'b → 'b`

   (f) (2pt)

   ```
   let rec f x y =
     match x with
       []   -> y
     | a::b -> a::f b y
   ```

   `'a list → 'a list → 'a list`

   (g) (3pt)

   ```
   let f (x, y) =
     match x with
       []   -> [x; y]
     | _::_ -> [y; x]
   ```

   `'a list * 'a list → 'a list list`

   ---

2. (10pts) Consider the following function:

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

(a) (4pts) What do the following expressions evaluate to?

i. `f [] []`

`[]`

---

ii. `f [] [1; 2; 3]`

`[1; 2; 3]`

---

iii. `f [3; 4] [2; 1]`

`[4; 3; 2; 1]`

---

iv. `f ["reversed"; "am"; "I"] []`

`["I"; "am"; "reversed"]`

(b) (6pts) Write a function `rev : 'a list -> int -> 'a list` such that `rev xl n` returns a list of the first `n` elements of `xl` in reverse order, followed by the remaining elements in their original positions. You can assume that n is between 0 and the length of `xl`.

Examples:

```
# rev [1; 2; 3; 4] 2;;
- : int list = [2; 1; 3; 4]
# rev [1; 2; 4] 0;;
- : int list = [1; 2; 4]
# rev [3; 5; 7] 3;;
- : int list = [7; 5; 3]
# rev [3; 8; 9; 2] 3;;
- : int list = [9; 8; 3; 2]
```

```
let rec rev_aux xl rl n =
  if n <= 0 then rl@xl else
  match xl with
    []    -> rl
  | x::xs -> rev_aux xs (x::rl) (n - 1)

let rev xl n = rev_aux xl [] n
```

3. (10pts) Write `is_square: int -> bool` such that `is_square n` returns `true` if and only if n is a square number, that is, there exists an integer m such that $n = m * m$.

   Hint: do not worry about efficiency; only correctness counts. You can search for m among all of the integers from 1 to n.

```
let rec is_square_aux n m =
  if m > n then false
  else if m * m = n then true
  else
    is_square_aux n (m + 1)

let is_square n = is_square_aux n 0
```

4. (10pts) Write `flatten: 'a list list -> 'a list` that appends all of the lists in the given list of lists. You may not use `@`, although you may write any helper function you choose (even one that mimics the functionality of `@`).

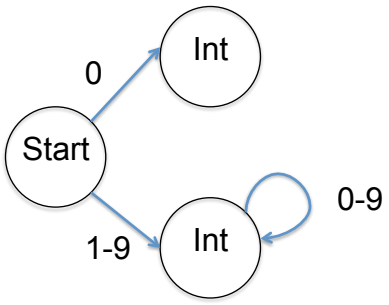
   Example:

```
# flatten [[1;2;3]; [4;5]; [8;2;3;4]];;
- : int list = [1;2;3;4;5;8;2;3;4]
```

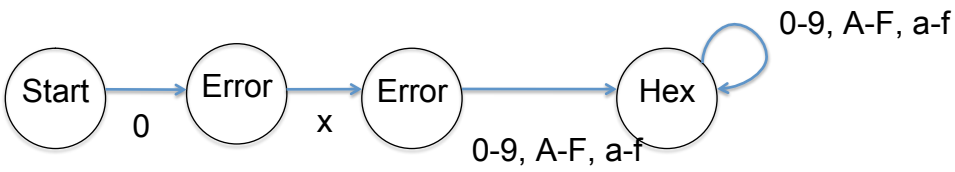
```
let rec flatten xl =
  match xl with
    []          -> []
  | []::xs      -> flatten xs
  | (y::ys)::xs -> y::flatten (ys::xs)
```

5. (10pts) In this question, you will write separate DFAs for lexing integer, hexadecimal, and octal constants, and then combine them. In each case, you should give a DFA with a start state, and with every other state labeled either "Error" or the type of the particular token the DFA is recognizing.

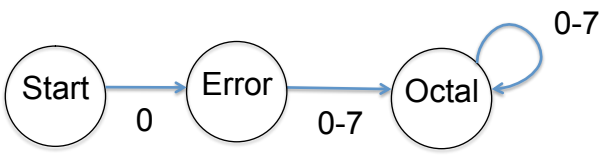
   (a) An *int* is a either the digit '0' by itself, or a digit '1' - '9' followed by zero or more digits '0' - '9'. Your states should be labeled Start, Error, or Int.



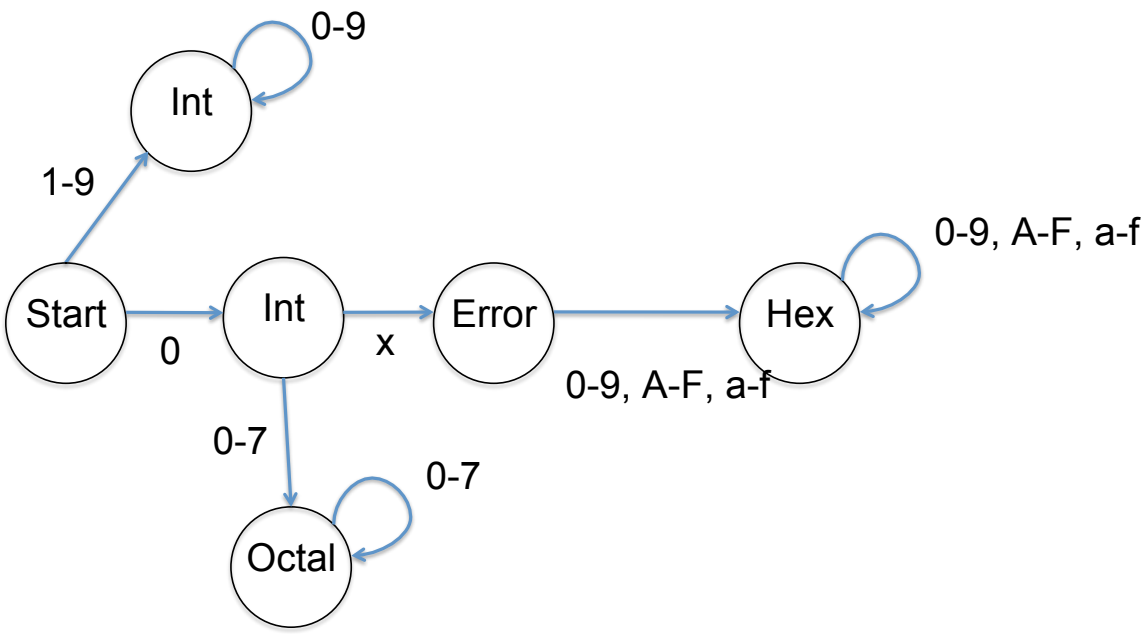
   (b) A *hexadecimal constant*, or *hex*, is "0x" followed by a sequence of one or more hexadecimal digits ('0' - '9', 'A' - 'F', 'a' - 'f'). Your states should be labeled Start, Error, or Hex.



   (c) An *octal constant* is '0' followed by a sequence of one or more octal digits ('0' - '7'). (Note that a single '0' is not a valid octal constant.) Your states should be labeled Start, Error, or Octal.



(d) Give a DFA that recognizes *ints*, *hexes* and *octals*. Your states should be labeled Start, Error, Int, Hex, or Octal.



6. (10pts) Consider the OCaml type **token** of tokens representing:

   - *ints*
   - *hexes*
   - *octals*
   - the operator '+'
   - the operator '*'

   given by:

```
type token = PLUS | TIMES | INT of int | HEX of int | OCTAL of int
```

   Give an ocamllex specification taking strings of ints, floats, hexes, '+'s and '*'s to lists of **token**s, while ignoring all other characters. You have access to the following functions:

   - `int_of_string : string -> int`
   - `hex_of_string : string -> int`
   - `octal_of_string : string -> int`

   Note that `hex_of_string` takes in a string in hex form and converts it to an int; likewise for `octal_of_string`.

   Example:

```
3 + 051 * 0x3F
```

   should lex to:

```
[INTEGER 3; PLUS; OCTAL 41; TIMES; HEX 63]
```

Please complete the solution started on the following page

Please write your solution to the problem from the previous page here:

```
let digit = ['0' - '9']
let hexdigit = ['0' - '9', 'A' - 'F', 'a' - 'f']
let octdigit = ['0' - '7']

rule tokenize = parse

  (* add your rules below *)

| '+'                              { PLUS }
| '*'                              { TIMES }
| ('0' | ['1'-'9'] digit*) as s     { INT (int_of_string s) }
| '0' octdigit+ as s               { OCTAL (octal_of_string s) }
| "0x" hexdigit+ as s              { HEX (hex_of_string s) }
```

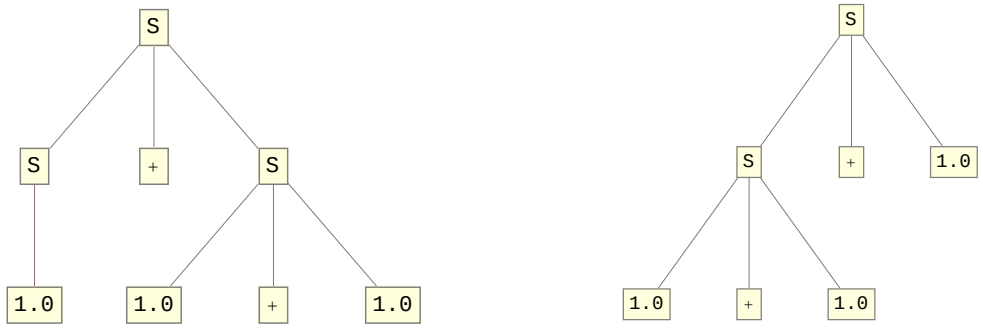
7. (10pts) Consider the following ambiguous grammar:

   S → <float> | S + S | S * S

   (a) (1pt) Give a sentence that has two parse trees.

   ```
   1.0 + 1.0 + 1.0
   ```

   (b) (1pt) Show the two parse trees.



   (c) (3pts) Based on those parse trees, show a stack/lookahead configuration where there are two different actions - either a shift and a reduce, or two different reduces - that would lead to the two parse trees shown, and say which action leads to which tree.

   ```
    S   + 1.0   .   + 1.0
    |
   1.0
   ```

   Here the central . separates the stack from the remaining input. Shifting results in the first parse tree; reducing results in the second.

(d) (5pts) Give a grammar recognizing the same language as the given grammar, but that is unambiguous and enforces the following:

- * has higher precedence than +
- * is left associative
- + is left associative

```
S → S + T       | T
T → T * <float> | <float>
```

8. (10pts + 5pts XC) Consider the following definitions:

```
exception Parse_failure
type token = LBRACKET | RBRACKET | LPAREN | RPAREN | COMMA | SEMICOLON | INT of int
```

We want to write a recursive descent parser recognizing the language of lists of pairs of integers (e.g. `[]`, `[(1, 2)]`, `[(1, 2); (3, 4); (0, 1)]`.)

Note that pair items are separated by a *comma* and list items are separately by *semicolons*.

Your parser should have the following signature:

```
parse : token list -> bool
```

`parse` takes in a list of `token`s and returns `true` if the corresponding string is in the language. If the string is not in the language, `parse` raises the exception `Parse_failure`.

(a) (5 pts.) Write a grammar for this language. Make sure it is not left-recursive and does not require an "obvious" left-factoring (i.e. there are no two productions for any non-terminal that begin exactly the same).

S → [T]
T → ϵ | P U
P → (<int> , <int>)
U → ϵ | ; P U

Note: Alternatively, you can use tokens (LBRACKET, etc...) in the above grammar.

(b) (5 pts XC) Argue that your grammar is LL(1) by showing that the FIRST sets of right-hand sides do not overlap. (You can ignore FOLLOW sets, even if your grammar has ϵ-productions.)

In regards to T: First(ϵ) = $\{\bullet\}$, whereas First(PU) =First(P) = $\{($, so there is no intersection

In regards to U: First(ϵ) = $\{\bullet\}$, whereas First($;PU$) = $\{;\}$, so there is no intersection.

(c) (5 pts) Write a recursive-descent parser based on this grammar.

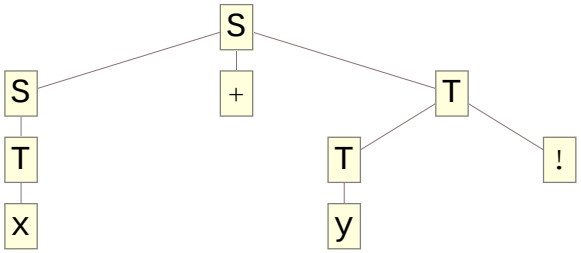
```
let rec parse_S toks =
  match toks with
    LBRACKET::toks' ->
    (
      match parse_T toks' with
        RBRACKET::toks'' -> toks''
      | _                -> raise Parse_failure
    )
  | _ -> raise Parse_failure
and parse_T toks =
  match toks with
    LPAREN::_ -> parse_U (parse_P toks)
  | _         -> toks
and parse_P toks =
  match toks with
    LPAREN::INT _::COMMA::INT _::RPAREN::toks' -> toks'
  | _                                          -> raise Parse_failure
and parse_U toks =
  match toks with
    SEMICOLON::toks' -> parse_U (parse_P toks')
  | _                -> toks

let parse toks =
  match parse_S toks with
    [] -> true
  | _  -> raise Parse_failure
```

9. (10pts) Consider the following grammar:

```
S → S + T | T
T → <id>  | T !
```

(a) (3pts) Give the parse tree for `x+y!`



(b) (7pts) Give the entire shift-reduce parse of `x+y!`, showing every shift and reduce action. For each reduce action, also give the production being reduced. On the stack, show only the top node of each tree. We have partially filled in the outline of the parse below, using exactly as many lines as there are steps:

Action	**Stack**	**Input**
Shift		x+y!
Reduce (T $\rightarrow <id>$)	x	+y!
Reduce (S $\rightarrow T$)	T	+y!
Shift	S	+y!
Shift	S +	y!
Reduce (T $\rightarrow <id>$)	S + y	!
Shift	S + T	!
Reduce (T $\rightarrow T!$)	S + T!	
Reduce (S $\rightarrow S + T$)	S + T	
Acc	S	

10. (10pts) True or False

(a) **True** **False** OCaml is a **statically** typed language

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

(c) **True** **False** Any program written in OCaml is guaranteed to terminate, due to OCaml's strong type system

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

(e) **True** **False** Every LL(1) grammar is unambiguous

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

(g) **True** **False** Nested comments can be recognized by a regular expression

(h) **True** **False** LR(1) parsers are capable of recognizing any context-free language

(i) **True** **False** LR(1) (bottom-up) parsers cannot handle **left recursion**.

(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.

11. (10pts) Extra Credit

Write a function `counts : 'a list -> ('a * int) list`.

`counts xl` returns a list of n pairs, where n is the number of distinct elements in xl. The i_{th} pair is (x, j) where x is the i_{th} distinct element of `xl` (ordering distinct elements of `xl` by first occurrence in `xl`), and j is the total number of times x occurs in `xl`.

Example:

```
# counts [1; 2; 3; 1; 3; 2; 1; 4];;
- : (int * int) list = [(1, 3); (2, 2); (3, 2); (4, 1)]
```

```
let rec count_instance a xl =
  match xl with
    []         -> [(a, 1)]
  | (b, n)::xs -> if b=a then (a, n + 1)::xs
                  else (b, n)::count_instance a xs

let rec counts_aux xl rl =
  match xl with
    []    -> rl
  | x::xs -> counts_aux xs (count_instance x rl)

let counts xl = counts_aux xl []
```