1. Give the types of each of the following Ocaml functions:

```
(a) let alwaysfour x = 4
```

val alwaysfour : 'a -> int

```
(b) let add x y = x + y
```

val add : int -> int -> int

```
(c) let concat x y = x ^ y
```

val concat : string -> string -> string

```
(d) let addmult x y = (x + y, x * y)
```

val addmult : int -> int -> int * int

```
(e) let rec f x = if x=[] then [] else hd x @ f (tl x)
```

val f : 'a list list -> 'a list

```
(f) let rec copy x = if x=[] then [] else hd x :: copy (tl x)
```

val copy : 'a list -> 'a list

```
(g) let b (x,y) = x+y
```

val b : int * int -> int

```
(h) let c (x,y) = x
```

val c : 'a * 'b -> 'a

```
(i) let d x = match x with (a,b) -> a
```

val d : 'a * 'b -> 'a

```
(j) let e x = hd x + 1
```

val e : int list -> int

```
(k) let f x y = match x with
      [] -> 0 | a::b -> a+y
```

val f : int list -> int -> int

```
(l) let g (a,b) (c,d) = (a+d, b^c)
    (Recall that ^ is the string concatenation operation.)
```

val g : int * string -> string * int -> int * string

```
(m) let rec h x = match x with
        [] -> 0 | (a,b)::r -> a + (h r)
```

val h : (int * 'a) list -> int

2. Define the following OCaml functions:

(a) `contains : 'a -> 'a list -> bool` such that `contains x lst` returns true if and only if `lst` has `x` as one of its elements. Do not use any pre-existing functions. E.g.

```
contains 4 [3;4;5]  =  true
```

```
let rec contains x lst =
  match lst with
    []    -> false
  | y::ys -> x = y || contains x ys
```

(b) `evens: 'a list -> 'a list` returns the 2nd, 4th, etc. elements of its argument. E.g.

```
evens [13;5;9;0;7;8] = [5; 0; 8]
```

```
let rec evens lis = match lis with
    [] -> []
  | [a] -> []
  | (a::b::lis') -> b :: evens lis'
```

(c) Implement the Ocaml function `partition: int list -> (int list) list`, which divides a list into "runs" of the same integer, e.g.

```
partition [9;9;5;6;6;6;3] = [[9;9]; [5]; [6;6;6]; [3]]
```

(You may define auxiliary functions, but it is not actually necessary.)

```
let rec partition lis =
    if lis = [] then []
    else match partition (tl lis) with
        [] -> [[hd lis]]
      | x :: xs -> if hd lis = hd x
             then (hd lis :: x) :: xs
             else [hd lis] :: (x :: xs)
```

(d) `genlist m n = [m; m+1; ... ; n]  (or [] if m>n)`

```
let rec genlist m n = if m>n then [] else m :: (genlist (m+1) n)
```

(f) `compress: int list -> (int * int) list` replaces runs of the same integer with a pair giving the count and the number. E.g.

```
compress [1;1;5;6;6;6;3] = [(2,1); (1,5); (3,6); (1,3)]
```

```
let rec compress lis = if lis = [] then [] else
        match compress (tl lis) with
           [] -> [(1, hd lis)]
         | (n,x)::lis' -> if x = hd lis
                          then (n+1,x):: lis'
                          else (1, hd lis)::(n,x)::lis'
```

(g) `apply: string -> int list -> int` applies the operator described by the string argument to the elements in the int list. The string argument can be either “times” or “plus”.

```
apply “times” [2;3;4] = 24
```

Assume the int list argument is non-empty.

```
let rec apply s lis = match lis with
        [n] -> n
      | n::lis' -> let n' = apply s lis'
                   in if s="times" then n*n' else n+n'
```

3. Suppose we are given the following type definition:

```
type btree = Leaf of int | Node of int * btree * btree
```

Define the following functions in Ocaml:

(i) `preorder: btree -> int list` gives the preorder listing of the labels of the tree. E.g.

```
let t = node(1, node(2, leaf(4), leaf(5)), node(3, leaf(6), leaf(7)))
preorder t
=> [1; 2; 4; 5; 3; 6; 7]
```

```
let rec preorder bt = match bt with
     Leaf n -> [n]
   | Node(n,lt,rt) -> n :: (preorder lt @ preorder rt)
```

(ii) `followpath: btree -> boolean list -> int list` gives the list of integers in the tree on the path described by the boolean list, where "true" means follow the left child and "false" means follow the right child. You may assume that the path described by the boolean list actually exists in the tree. E.g.

```
followpath t [true; false]
=> [1; 2; 5]
```

```
let rec followpath bt blis = match bt with
     Leaf n -> [n]
   | Node(n,lt,rt) -> if blis = [] then [n]
                else if hd blis
                   then n::(followpath lt (tl blis))
                   else n::(followpath rt (tl blis))
```

(iii) `height: btree -> int` gives the height of a tree, defined as the length of the longest path from the root to a leaf node.

```
height t;;
```

```
=> 2
```

```
let max (x,y) = if x>y then x else y

let rec height bt = match bt with
     Leaf n -> 0
   | Node(n,lt,rt) -> 1 + max(height lt, height rt)
```

`(iv) balanced: btree -> boolean` returns true if for every internal node, the heights of its children differ by no more than 1.

```
balanced t
=> true
let t1 = node(1, leaf(2), node(3, leaf(6), leaf(7)))
=> true
let t2 = node(1, leaf(2), node(3, leaf(6), node(8, leaf(7), leaf(9)))
=> false
```

```
let rec balanced bt = match bt with
     Leaf n -> true
   | Node(n,lt,rt) -> let h1 = height lt
                 and h2 = height rt
                 in let p = if h1<h2 then h2-h1 else h1-h2
                   in (p=0 || p=1) && balanced lt
                                && balanced rt
```

4. Given this Java class definition:

```
class List { public int h; public List t;
   public int hd () { return h; }
   public List tl () { return t; }
   public List (int h, List t) {this.h = h; this.t = t; }
}
```

The empty list is represented by null. Define the Java function to concatenate two lists:

```
public static append (L1, L2) {
        if (L1 == null)
                return L2;
        return new List(L1.h, append(L1.t, L2));

    }
```

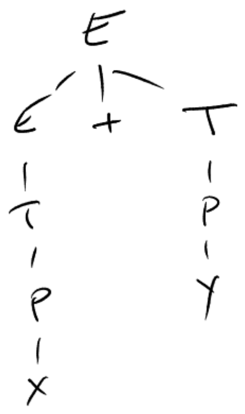
5. Given the grammar:

```
    E  ->  E + T  |  T
    T  ->  T * P  | P
    P  ->  id  |  ( E )
```

and an Ocaml type for trees:

```
    type tree = Node of string * tree list
```

we can represent concrete syntax trees. For example, the syntax tree



would correspond to the term

```
Node("E", [Node("E", [Node("T", [Node("P", [Node("id", [])])])]);
           Node(„+", []);
           Node("T", [Node("P", [Node("id", [])])])])
```

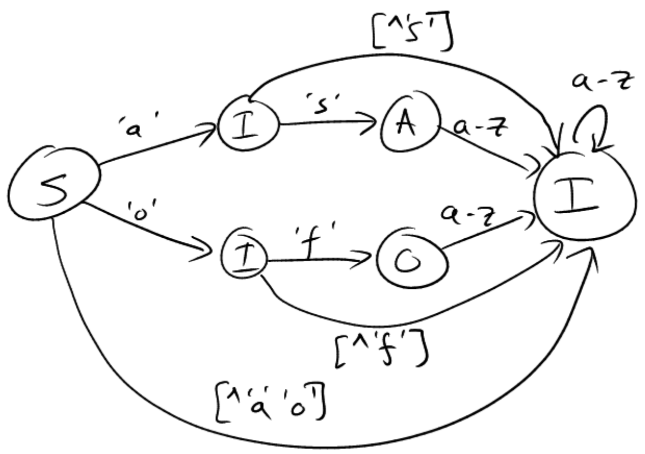
Here is a type for abstract syntax :

```
type exp = Id of string | Plus of Exp*Exp | Times of Exp*Exp
```

Write a function abstract: tree -> exp to transform a concrete syntax tree to an AST.

```
let rec abstract t = match t with
  Node(s, children) ->
        (match children with
          [] -> Id s
        | [ch] -> abstract ch
        | [Node(“(“,[]); ch; Node(“)”,[])] -> abstract ch
        | [ch1; Node(op, []); ch2] ->
                let ach1 = abstract ch1
                and ach2 = abstract ch2
                in if op = "+" then Plus(ach1, ach2)
                        else Times(ach1, ach2) )
```

6. An identifier is any sequence of one or more characters 'a' - 'z' that is not a keyword. Our language recognizes identifiers and the keywords 'of' and 'as'. Give a deterministic finite-state machine that recognizes our language. Each state should be labeled with either S (start state), O ('of' keyword), A ('as' keyword) or I (identifier).



7. Write an ocamllex specification for tokens of the following type:

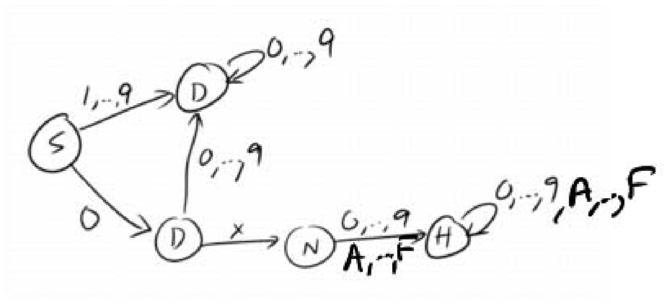
```
type token = PLUS | MINUS | INT of int
```

where these represent, respectively, the sequences “+”, “-“, and any string of one or more characters '0' - '9'. (You will want to use the function `int_of_string: string → int`.)

rule tokenize = parse
‘+’ { PLUS }
| ‘-‘ { MINUS }
| [‘0’-‘9’]+ as i { INT (int_of_string i) }

8. A decimal constant is any sequence of one or more decimal digits (including one starting with zero). A hexadecimal constant has the form 0xW where W is a sequence of one or more decimal digits or the letters A-F.

(a) Write a finite-state machine to recognize either decimal or hexadecimal constants. Label each state with either S (start state), D (decimal constant), H (hexadecimal constant), or N (neither).



(b) Write a regular expression to recognize either a decimal or hexadecimal constant.

‘0’ ‘x’ ([‘0’ – ‘9’] | [‘A’ – ‘F’])+ | [‘0’ – ‘9’]+

9. Define finite-state machines for the following regular expressions over the characters a and b. Label each node either S (start), A (accept the string) or R (reject it).

(a) a(a|b)*
(b) aa*b(b|a)*
(c) (b|a)*a*ab(b|a)*

(a)
$$\begin{tikzcd}
\boxed{S} \arrow[r, "a"] & \boxed{A} \arrow[loop right, "{a,b}"]
\end{tikzcd}$$

(b)
$$\begin{tikzcd}
\boxed{S} \arrow[r, "a"] & \boxed{R} \arrow[loop above, "a"] \arrow[r, "b"] & \boxed{A} \arrow[loop right, "{a,b}"]
\end{tikzcd}$$

(c)
$$\begin{tikzcd}
\boxed{S} \arrow[loop above, "b"] \arrow[r, "a"] & \boxed{R} \arrow[loop above, "a"] \arrow[r, "b"] & \boxed{A} \arrow[loop above, "{a,b}"]
\end{tikzcd}$$

10. (a) Write a regular expression for this language: semicolon-separated lists of a' and b's, in square brackets. Examples of strings in the language are [], [a], and [a;b;a;a].

'[' (('a' | 'b') (';' ('a' | 'b'))*)? ']'

(b) Then write a finite-state machine for the language. Each state should be labeled either as S (for the start state), A (for an accepting state), or E (for an error state).

(b)
$$\begin{tikzcd}
\boxed{S} \arrow[r, "{[}"] & \boxed{E} \arrow[d, "{a,b}"] & & \boxed{A} \\
& \boxed{E} \arrow[urr, "{]}"] \arrow[r, bend left, ";"] & \boxed{E} \arrow[l, bend left, "{a,b}"] &
\end{tikzcd}$$

11. Write an ocamllex specification for tokens of this type:

```
type token = LESSTHAN | LEFTSHIFT | LEFTSHIFTEQ | IDENT of string
```

where these represent, respectively, the sequence “<”, “<<”, “<<=”, and any string starting with a letter and consisting solely of letters and digits. Your specification should return the next token in the input, ignoring any character other than the ones that constitute tokens.

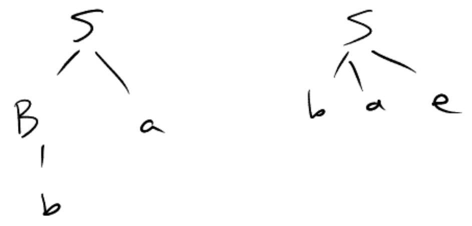
```
let letter = [‘A’ – ‘Z’] | [‘a’ – ‘z’]
let digit = [‘0’ – ‘9’]

rule tokenize = parse
      “<”   { LESSTHAN }
|     “<<”  { LEFTSHIFT }
|     “<<=” { LEFTSHIFTEQ }
|     (letter (letter | digit)*) as id { IDENT id }
|     _     { tokenize lexbuf }
```

12. The following grammar has a shift-reduce conflict:

$S \rightarrow B\ a \mid b\ a\ e$
$B \rightarrow b \mid c$

(a) Show two sentences that lead to the same stack configuration and lookahead symbol, but where the correct parsing action in one case is to shift and in the other is to reduce. Show the two parse trees, and the problematic stack configuration, and say whether to shift or reduce in either case.

ba and bae lead to stack configurations with b on the stack and a as the lookahead symbol. For ba, the correct action in this configuration is to Reduce using B -> b; for bae the correct action is Shift.



(b) Give an equivalent grammar – one with the same language – which does not have the conflict.

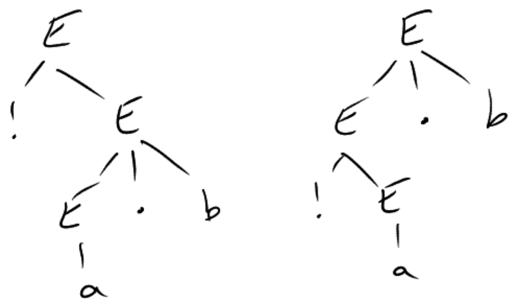
S -> b a | c a | b a e

13. Given the following ambiguous grammar

E → E . id | ! E | E [E] | id

a) Give a sentence that has two distinct parse trees. Show the parse trees.

! a . b (or !a[b], as well as others)



b) Explain how you might disambiguate the grammar using ocamlyacc precedence and associativity declarations. You are free to choose the precedence order and associativity of operators, but you must say what declarations you would use and what effect they would have. Assume the following declaration:

```
%token DOT IDENT BANG LBRACK RBRACK
```

%left DOT
%nonassoc BANG
%nonassoc LBRACK

These declarations give precedence to BANG over DOT, so the above sentence would be parsed as the tree on the right. It also gives precedence to LBRACK, which has an ambiguity as noted above.

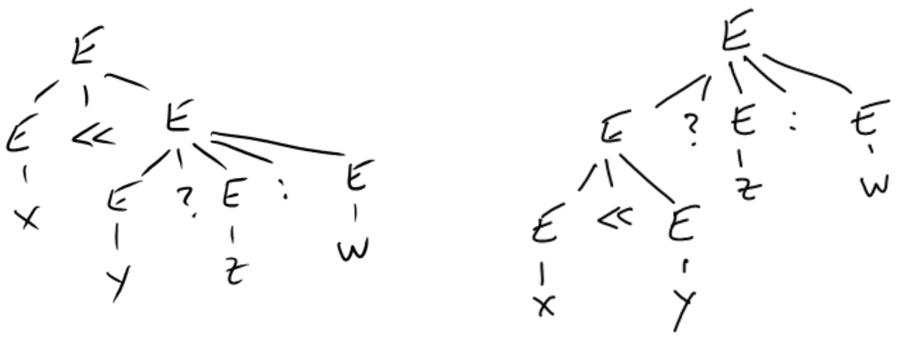
14. Here is a grammar for part of C expressions:

E -> id | E ? E : E | E << E

This grammar is ambiguous.

(a) Give a sentence that has two distinct parse trees, and show those parse trees.

x << y ? z : w



(b) Explain how you might use ocamlyacc associativity declarations to resolve the conflict caused by this ambiguity. Be specific about what declarations you would use and what their effect would be. (The actual rules for resolving this ambiguity in C or Java do not matter; you just need to use the rules to resolve it in some way.)

When we have << on the stack and ? in the input, if we reduce instead of shift, we'll get the tree on the right. We can do this by giving precedence of << over ?, which we do by putting an associativity declaration for ? before one for <<:

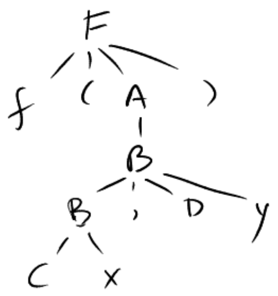
%nonassoc ?
%left <<

(Alternatively, we could reverse the order of these declarations and get the parse on the left.)

15. Consider this grammar:

F -> id (A)
A -> ε | B
B -> id id | B, id id

(a) Show a parse tree for the following sentence: f (C x, D y)



(b) Calculate FIRST(F), FIRST(A), and FIRST(B).

FIRST(F) = {id}   FIRST(A) = { id, • }   FIRST(B) = {id}

(c) Why is this not an LL(1) grammar?

It is left-recursive. (In addition, the two productions from B do not satisfy the LL(1) condition, in that FIRST(id id) and FIRST(B id id) are not disjoint.)

(d) Transform it to an equivalent LL(1) grammar, and write a recursive descent parser for it. Assume the token type is

type token = Id | LParen | RParen | Comma

and the parseF function has type

parseF: token list -> (token list) option where option is the type defined as: type ’a option = Some ’a | None.

```
F  ->  id ( A )
A  ->  ε  |  id id B
B  ->  ε  |  , id id B
```

```
let rec parseF lis = match lis with
          Id::LParen::lis’ -> (match parseA lis’ with
                                   Some (RParen :: lis”) -> Some lis”
                                 |  _  -> None)
    |     _ -> None

let rec parseA lis = match lis with
          Id::Id::lis’ -> parseB lis’
     |    _             -> Some lis   (* could check for RParen here *)

let rec parseB lis = match lis with
          Comma::Id::Id::lis’ -> parseB lis’
     |    _                    -> Some lis   (* could check for RParen here *)
```

16. Suppose we are given grammar that contains the following rules:

```
Expression  →  "do" Stmt "while" "(" Expression ")"
             | Stmt ";" Expression
             | Ident
```

Assume the token type is

```
type token = DO | WHILE | LPAREN | RPAREN
             | SCOLON | IDENT of string
```

the abstract syntax for the Expression non-terminal is

```
type exp = DoWhile of stmt * exp | Sequence of stmt * exp
         | Id of string
```

and you are provided with the `parseStmt` function of the following type.

```
parseStmt: token list -> stmt * token list
```

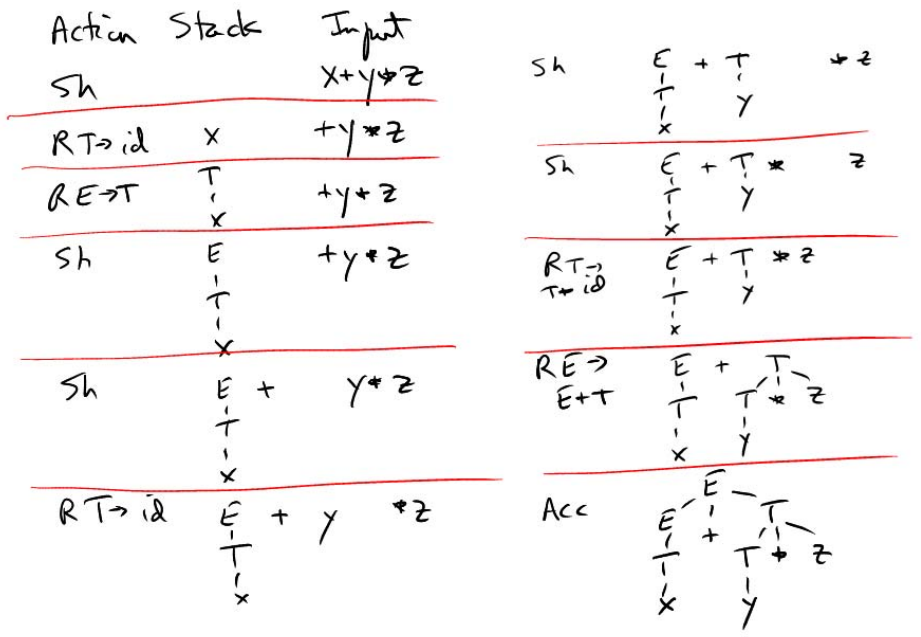
Implement the `parseExp: token list -> exp * token list` function. Ignore error cases. Also, assume that DO and IDENT are not in FIRST(Stmt).

```
let rec parseExp toks =
     match toks with
       DO::rest -> ( match parseStmt rest with
                       (st, WHILE::LPAREN::rest2) ->
                            match parseExp rest2 with
                                  (e, RPAREN::etc) -> (DoWhile(st, e), etc) )
     | IDENT s::rest  ->  (Id s, rest)
     | _  ->  ( match parseStmt toks with
                       (st, SCOLON::rest) -> match parseExp rest with
                                                  (e, etc) -> (Sequence(st, e), etc)  )
```

17. Consider this grammar:

E -> E + T | T
T -> T * id | id

Show a complete shift-reduce parse for the sentence x+y*z. Include columns Action, Stack, and Input, as we did in class. (Warning: this may take more than one piece of paper, depending upon how big you write. There are 11 steps, counting the final "Accept" step.)

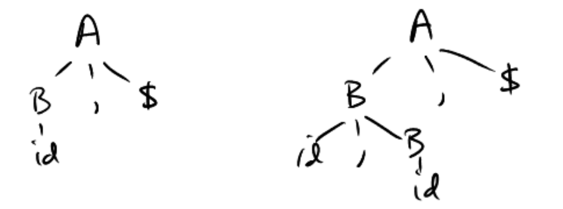


Action	Stack	Input
Sh		x+y*z
R T-> id	x	+y*z
R E->T	T	+y*z
Sh	E	+y*z
Sh	E +	y*z
R T-> id	E + y	*z
Sh	E + T	*z
Sh	E + T *	z
R T-> T* id	E + T * z	
R E-> E+T	E + T	
Acc	E	

18. This grammar is not LR(1). It has a shift-reduce conflict:

A -> B , $
B -> id | id , B

Show parse trees for two sentences with the following property: they lead to the same stack configuration (meaning, the roots of the trees on the stack are the same), with the same lookahead symbol, but in one case the correct action is the shift and in the other it is to reduce. Show that stack configuration.



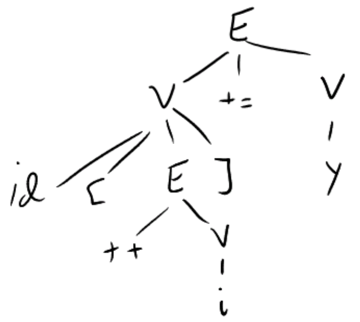
Stack Input
id ,$
Reduce

Stack Input
id ,id,$
Shift

19. Given this grammar:

E -> V ++ | ++ V | V += V
V -> id | id [E]

(a) Give the parse tree for input x[++i] += y.



(b) Show the shift/reduce parse for that input. We have shown the first line. You should show every shift and reduce action, indicating the production for reduce actions, until the final Accept action. On the stack, do not give the entire trees, but just the topmost node (as we did in class).

Action	**Stack**	**Input**
Sh		**x[++i] += y**
Sh	**x**	**[++i] += y**
Sh	**x [**	**++i] += y**
Sh	**x [++**	**i] += y**
R V-> id	**x [++ i**	**] += y**
R E -> ++ V	**x [++ V**	**] += y**
Sh	**x [E**	**] += y**
R V -> id [E]	**x [E]**	**+= y**
Sh	**V**	**+= y**
Sh	**V +=**	**y**
R V -> id	**V += y**	
R E -> V += V	**V += V**	
Acc	**E**	