Practice questions for midterm 1

Questions 1-3 use this definition of type exp:

```
type exp = Plus of exp*exp | Times of exp*exp | Minus of exp*exp
         | Negate of exp | Var of string | Int of int
```

1. Write removeMinus: exp -> exp which replaces every occurrence of an expression of the form Minus(e,e') by Plus(e, Negate(e')). For example:

```
removeMinus (Minus (Minus(Int 3, Var "x"), Times(Int 3, Var "a")))
  = Plus (Plus(Int 3, Negate (Var "x")),
          Negate (Times(Int 3, Var "a")))
```

SOLN:

```
let rec removeMinus e = match e with
   Plus(e1,e2) -> Plus(removeMinus e1, removeMinus e2)
 | Times(e1,e2) -> Times(removeMinus e1, removeMinus e2)
 | Minus(e1,e2) -> Plus(removeMinus e1, Negate (removeMinus e2))
 | Negate e1 -> Negate (removeMinus e1)
 | e -> e
```

2. Write simplify: exp -> exp which simplifies occurrences of multiplication by zero to zero, multiplication by one to its other argument, and additions or subtractions of zero: For example:

```
simplify (Mult(Int 1, Plus(Var "x", Int 0))) = Var "x"
```

SOLN:

```
let rec simplify e = match e with
   Times(e1, e2) -> (match (simplify e1, simplify e2) with
                       (e', Int 1) -> e'
                     | (Int 1, e') -> e'
                     | (e', Int 0) -> Int 0
                     | (Int 0, e') -> Int 0
                     | (e', e'') -> Times(e', e''))
 | Plus(e1, e2) -> (match (simplify e1, simplify e2) with
                       (e', Int 0) -> e'
                     | (Int 0, e') -> e'
                     | (e', e'') -> Plus(e', e''))
 | Minus(e1, e2) -> (match (simplify e1, simplify e2) with
                       (e', Int 0) -> e'
                     | (e', e'') -> Minus(e', e''))
 | Negate e1 -> Negate (simplify e1)
 | e -> e
```

3. Write pp: exp -> string which ``pretty-prints'' its argument,

i.e. renders it in concrete syntax. For example:

```
pp (Mult(Var "x", Plus(Int 3, Var "y"))) = "(x*(3+y))"
```

SOLN:

```
let rec pp e = match e with
   Plus(e1,e2) -> "(" ^ pp e1 ^ "+" ^ pp e2 ^ ")"
 | Times(e1,e2) -> "(" ^ pp e1 ^ "*" ^ pp e2 ^ ")"
 | Minus(e1,e2) -> "(" ^ pp e1 ^ "-" ^ pp e2 ^ ")"
 | Negate e1 -> "-" ^ pp e1
 | Var s -> s
 | Int i -> string_of_int i
```

4. Write an ocamllex specification that extracts the next integer (sequence of digits) from an input stream and returns it as an integer value; if there are not digits in the input, it should return -1.

SOLN:

```
rule main = parse
   ['0'-'9']+ as i   { int_of_string i }
   EOF               { -1 }
   _                 { main lexbuf }
```

5. For this grammar:

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

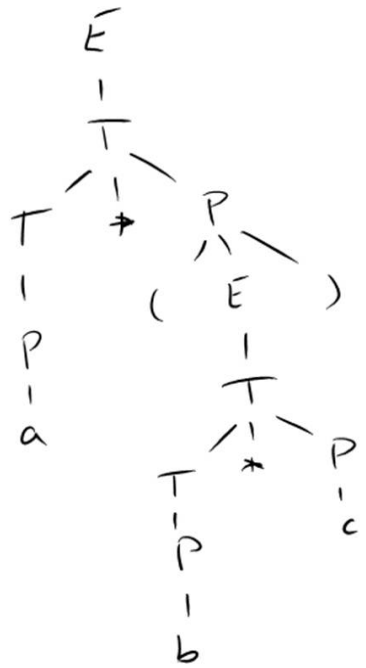
(a) Give an example of an E-sentence, and an E-form that is not a sentence;
similarly for T and P.

SOLN:

E-sentence: x*(y+z)
E-form: E+T
F-sentence: x*(y+z)
F-form: T*P
P-sentence: (y+z)
P-form: (E)

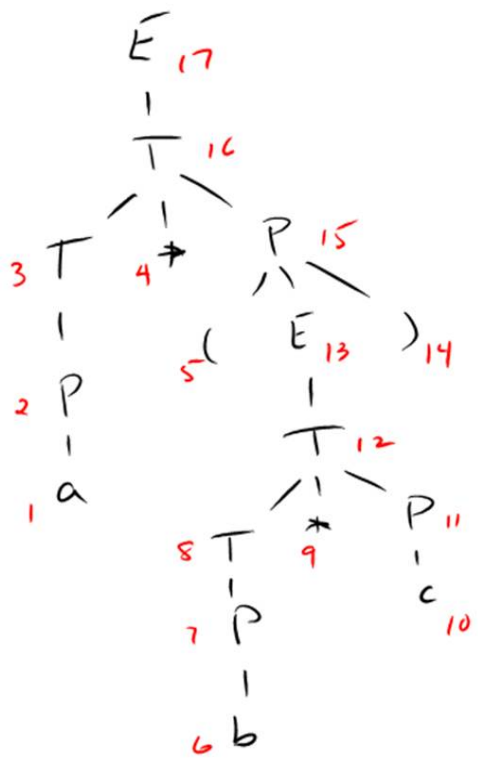
(b) Give a parse tree for "a*(b*c)".

SOLN:



(c) Give a numbering of that tree in a post-order traversal. Then show the shift-reduce parse for the tree.

SOLN:



```
Action          Stack             Input
Shift                             a*(b*c)
Red              a                 *(b*c)
Red              P                 *(b*c)
Shift            T                 *(b*c)
Shift            T *               (b*c)
Shift            T * (             b*c)
Red              T * ( b           *c)
Red              T * ( P           *c)
Shift            T * ( T           *c)
Shift            T * ( T *         c)
Red              T * ( T * c       )
Red              T * ( T * P       )
Red              T * ( T           )
Shift            T * ( E           )
Red              T * ( E )
Red              T * P
Red              T
Acc              E
```

For the next two questions, suppose we use this type to represent the *concrete* syntax given above:

```
    type token = PlusT | TimesT | Id of string | LParen | RParen

    type etree = E_prod1 of etree * token * ttree | E_prod2 of ttree
     and ttree = T_prod1 of ttree * token * ptree | T_prod2 of ptree
     and ptree = P_prod1 of token | P_prod2 of token * etree * token
```

For example, the parse tree for "x+y" would be represented by this value of type etree:

```
       E_prod1(E_prod2(T_prod2(P_prod1 (Id "x"))),
               PlusT,
               T_prod2(P_prod1 (Id "y")))
```

(d) Define legaltree: etree -> bool, to check if it is legal in the sense that the token in eprod1 is PlusT, the token in tprod1 is TimesT, etc. You will need to define auxiliary functions legaltreeT and legaltreeP.

SOLN:

```
   let rec legaltree e = match e with
        E_prod1(e1,tok,t1) -> legaltree e1 && tok=PlusT && legaltreeT t1
      | E_prod2 t1 -> legaltreeT t1
   and legaltreeT t = match t with
        T_prod1(t1,tok,p1) -> legaltreeT t1 && tok=TimesT && legaltreeP
p1
      | T_prod2 p1 -> legaltreeP p1
   and legaltreeP p = match p with
        P_prod1(tok) -> (match tok with Id _ -> true | _ -> false)
```

```
      | P_prod2(tok1,e1,tok2) -> tok1=LParen && legaltree e1 &&
tok2=RParen
```

(e) Define frontier: etree -> token list. Again, you will need auxiliary functions for values of type ttree and ptree.

SOLN:

```
 let rec frontier e = match e with
        E_prod1(e1,tok,t1) -> frontier e1 @ [tok] @ frontierT t1
      | E_prod2 t1 -> frontierT t1
 and frontierT t = match t with
        T_prod1(t1,tok,p1) -> frontierT t1 @ [tok] @ frontierP p1
      | T_prod2 p1 -> frontierP p1
 and frontierP p = match p with
        P_prod1(tok) -> [tok]
      | P_prod2(tok1,e1,tok2) -> [tok1] @ frontier e1 @ [tok2]
```

(f) Define absE: etree -> exp, where exp is the type defined above:

```
   type exp = Plus of exp*exp | Times of exp*exp | Minus of exp*exp
            | Negate of exp | Var of string | Int of int
```

For example, if e is the value of type etree shown above (representing the parse tree of "x+y"), then absE e = Plus(Var "x", Var "y"). You will need to define auxiliary functions absT and absP.

SOLN:

```
 let rec absE e = match e with
        E_prod1(e1,_,t1) -> Plus(absE e1, absT t1)
      | E_prod2 t1 -> absT t1
 and absT t = match t with
        T_prod1(t1,_,p1) -> Times(absT t1, absP p1)
      | T_prod2 p1 -> absP p1
 and absP p = match p with
        P_prod1(Id x) -> Var x
      | P_prod2(_,e1,_) -> absE e1
```

6. For each of the following grammars, calculate FIRST and FOLLOW sets (showing the tables calculated at each iteration), and say whether the grammar is LL(1).

(a)
```
S -> aSe  |  B
B -> bBe  |  C
C -> cCe  |  d
```

SOLN:

	fsts_0	fsts_1	fsts_2	fsts_3	fsts_4
S		a	a,b	a,b,c,d	a,b,c,d
B		b	b,c,d	b,c,d	b,c,d

C	c,d	c,d	c,d	c,d

	flws_0	flws_1	flws_2	flws_3
S	eof	eof, e	eof, e	eof, e
B		eof, e	eof, e	eof, e
C		e	eof, e	eof, e

Is LL(1): No FIRST sets of right-hand sides from any non-terminal overlap. (Since there are no epsilon productions, this calculation doesn't involve FOLLOW sets.)

(b) S -> I | b
I -> aESF
F -> eS | epsilon
E -> t | f

SOLN:

	fsts_0	fsts_1	fsts_2	fsts_3
S		b	a,b	a,b
I		a	a	a
F		e, bullet	e, bullet	e, bullet
E		t, f	t, f	t, f

	flws_0	flws_1	flws_2	flws_3
S	eof	eof, e	eof, e	eof, e
I		eof	eof, e	eof, e
F			eof, e	eof, e
E		a, b	a, b	a, b

Not LL(1): F_{F,1} = FIRST(eS) = {e}
F_{F,2} = FIRST(epsilon) union FOLLOW(F) = {bullet, eof, e}
These overlap, so the grammar is not LL(1).

(c) A -> BCe | gDB
B -> bCDE | epsilon
C -> DaB | ca
D -> dD | epsilon
E -> gAf | c

SOLN:

	fsts_0	fsts_1	fsts_2	fsts_3	fsts_4
A		g	b,c,g	a,b,c,d,g	a,b,c,d,g
B		b,bullet	b,bullet	b,bullet	b,bullet
C		c	d,a,c	d,a,c	d,a,c
D		d,bullet	d,bullet	d,bullet	d,bullet
E		g,c	g,c	g,c	g,c

	flws_0	flws_1	flws_2	flws_3	flws_4
A	eof	eof, f	eof,f	eof,f	eof,f
B		d,a,c	d,a,c,g	d,a,c,g	d,a,c,g
C		c,d,g	c,d,g	c,d,g	c,d,g
D		a,b,c,g,eof	a,b,c,g,eof	a,b,c,g,eof	a,b,c,g,eof

E	d,a,c	d,a,c,g	d,a,c,g

Is LL(1):

A: FIRST(BCe) = {b,d,a,c}, FIRST(gDB) = {g}

B: FIRST(bCDE) = {b}, FIRST(epsilon) u FOLLOW(B) = {bullet,d,a,c,g}

C: FIRST(DaB) = {d,a}, FIRST(ca) = {c}

D: FIRST(dD) = {d}, FIRST(epsilon) u FOLLOW(D) = {bullet,a,b,c,g,eof}

E: FIRST(gAf) = {g}, FIRST(c) = {c}