Solution for Midterm

1.a) (8 Points)

First(Expr) = { -, (, **id** }; Follow(Expr) = { $,)}, First(Var) = { **id** }; Follow(Var) = { $,), -}
First(ExprTail) = { -, ϵ }; Follow(ExprTail) = { $,)}; First(VarTail) = {), ϵ } Follow(VarTail) = { $,), -}

1.b (11 Points) The predictive parser table for the grammar:

	id	–	(	)	$
Expr	Var ExprTail	- Expr	(Expr)		
ExprTail		- Expr		ϵ	ϵ
Var	**id** VarTail				
VarTail		ϵ	(Expr)	ϵ	ϵ

1.c (3+5 Points) The grammar is not LL(1). $First(B)$ and $Follow(B)$ both contain symbol b. Hence, on input b, B can either be expanded by nCfgRuleBb, or by $B \rightarrow \epsilon$.

1.d (3+5 Points) The grammar is LL(1) because the rules of the nonterminals can be distinguished by their different starting terminal symbols. Hence, it is possible to look at an input symbol and decide which rule to use for each nonterminal.

2.a (13 Points) Start by adding a new nonterminal S' and rule $S' \rightarrow SL$. The states are:

```
s0:
  [S' -> .SL]
  [SL -> .SL; S]
  [SL -> .]                goto(s0, SL) = s1;
s1:
  [S' -> SL.]
  [SL -> SL.;S]            goto(s1, ;) = s2;
s2:
  [SL -> SL;.S]
  [S -> .stmt]             goto(s2, S) = s3;
s3:
  [SL-> SL; S.]            goto(s2, stmt) = s4
s4:
  [S -> stmt.]
```

2.b (12 Points) First we calculate the follow for nonterminals:

```
Follow(S') = { $ }, Follow(SL) = {; $}, Follow(S) = { ;, $}
    |   ;    |    stmt    |   $       | SL   | S   |
----- ---------------------------------------------
s0  |   r2   |            |   r2      |  1   |     |
-----------------------------------------------------
s1  |   s2   |            |   accept  |      |     |
-----------------------------------------------------
s2  |        |    s4      |           |      |  3  |
-----------------------------------------------------
s3  |   r1   |            |   r1      |      |     |
-----------------------------------------------------
s4  |   r3   |            |   r3      |      |     |
-----------------------------------------------------
```

2.c (3+7 Points) The grammar is LR(1) (In fact, it is SLR(1)). There are no shift/reduce or reduce/reduce conflicts. This can be discerned by looking at the grammar carefully and examining rules that can cause conflicts. The first possibility appears to be in the following rules:

$S \rightarrow$ **id** $:= A;$
$A \rightarrow$ **id** $:= A$

They are very similar so they cause reduce/reduce error. However, in order for them to cause reduce/reduce or shift/reduce conflict, they must occur in the same state. However, if you creates SLR(1) states, they will never appear in the same state. If the grammar has an additional rule:

$$S \rightarrow \textbf{id} := A; \mid A$$

A shift/reduce conflict may arise. Another possibility may arise in the following state:

```
[ A -> id := .A]
[ A -> .id := A]
[ A ->.E]
[ A -> .P]
[ P ->.id]
[ P -> .(A)]
```

However, there are no possibilities of shift/reduce or reduce/reduce here either because '.' is not at the end of any of the items. On seeing an id, we will do a shift to state:

```
[ A -> id. := A]
[ P ->id.]
```

Shift/reduce error can occur here if follow(P) includes ':=' which it does not. There are no other possible states where conflict situations can arise.

3.a (15 Points) A tree $(numt_1t_2)$ is balanced if $(num > t_1.max)$ and $(num < t2.min)$ Here $t_1.max$ denotes the maximum number in tree t_1 and $t_2.min$ stores the minimum number in tree t_2. Hence, we will need to keep track of both minimum and maximum number in a binary tree.

```
BinTree -> num { BinTree.num = num; }
           BinTree1
           { if (BinTree1.tree == null)  BinTree.min = num;
             else
                   BinTree.balanced = BinTree1.balanced && binTree.num > BinTree1.max;
                   BinTree.min = BinTree1.min;
             BinTree.tree = nonnull;
           }
           BinTree2
           { if (BinTree1.tree = null) BinTree.max = num;
             else
                 BinTree.balanced = BinTree2.balanced && binTree.num < BinTree2.min;
                 BinTree.max = BinTree2.max
           }
BinTree -> empty {BinTree.balanced = true; BinTree.tree = null}
```

3.b (15 Points) Associate the following attributes with C and D: i) i) D.count: inherited attribute of D Keeps track of number of a's and b's generated. ii) C.acount: Inherited attribute of C. Keeps track of number of a's that were generated. iii) C.bcount: Inherited attribute of C. Keeps track of number of b's that were generated.

```
DS -> {D.count = 0} D
D  -> a               { D1.count =  D.count + 1; cout << 'a'; }
      D1
      b               { cout << 'b';}
D  -> { C.acount = D.count; C.bcount = D.count}
      C
C  -> c {C1.acount = C.acount - 1; if (C1.acount > 0) count << 'a' else count << 'c';}
      C1 {C1.bcount = C.bcount-1; if (C1.bcount > 0) count << 'b' else count << 'd';}
      d
```