Chapter 3 & 4

1) (10) Given the grammar below, identify which sentences are in the language (which are valid sentence).
   - **a. baab**
   - b. bbbab
   - c. bbaaaaaa
   - **d. bbaab**

<S> → <A> a <B> b
<A> → <A> b | b
<B> → a <B> | a

a. <S> → <A>a<B>b
→ ba<B>b
→ baab ✓

b. <S> → <A>a<B>b
→ <A>ba<A>b
→ <A>bba<B>b
→ bbba<B>b ✗ (still have <B> which results in a's)

c. <S> → <A>a<B>b
→ <A>ba<B>b
→ bba<B>b
→ bbaa<B>b
→ bbaaa<B>b
→ bbaaaa<B>b
→ bbaaaaa<B>b
→ bbaaaaaab ✗ (will always end with b)

d. <S> → <A>a<B>b
→ <A>ba<B>b
→ bba<B>b
→ bbaab ✓

2) (10) Identify all of the tokens (categories of lexemes) in the grammar below, and which lexemes they categorize. Put them in a table.

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <id> + <expr>

| <id> * <expr>

| (<expr>)

| <id>

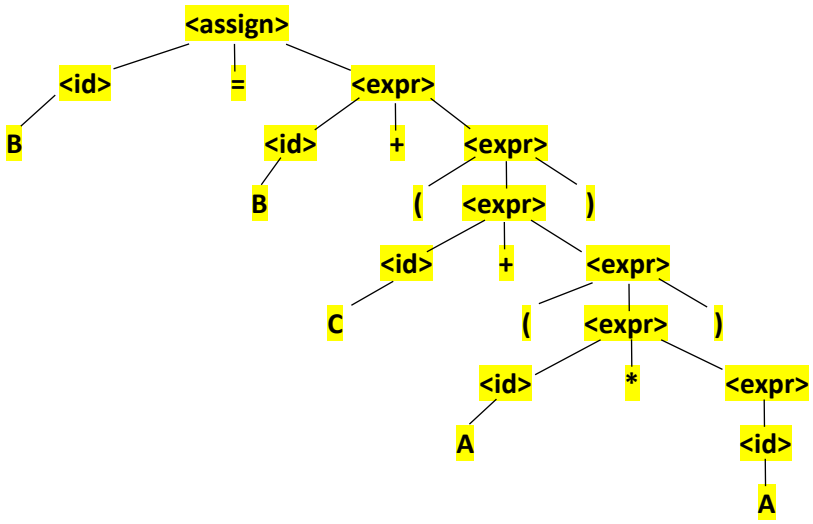
TOKENS	LEXEMES
Equal_op	=
Add_op	+
Mult_op	*
Identifiers	A, B, C
Left_paren	(
Right_paren	)

3) (10) Given the grammar from question 2, show a left-most derivation and draw the parse tree for the following statement.
   a. B = B + (C + (A * A))

DERIVATION:

<assign> → <id> = <expr>
→ B = <expr>
→ B = <id> + <expr>
→ B = B + <expr>
→ B = B + (<expr>)
→ B = B + (<id> + <expr>)
→ B = B + (C + <expr>)
→ B = B + (C + (<expr>))
→ B = B + (C + (<id> * <expr>))
→ B = B + (C + (A * <expr>))
→ B = B + (C + (A * <id>))
→ B = B + (C + (A * A))

PARSE TREE:



4) (10) Remove all of the recursion from the following grammar:

S -> Aa | Bb
A -> Aa | AbC | C
B -> S | bb
C -> c

First remove the direct recursion in A:

A -> CA'
A' -> aA' | bCA' | ∈

Now remove the indirect recursion with S and B:

B -> Aa | Bb | bb
B -> CA'a | Bb |bb
B -> CA'aB' | bbB'
B' -> bB' | ∈

Equalities:
S -> Aa
S -> Bb
A -> CA'
B -> S
B -> bb

Final Grammar:

S -> Aa | Bb
A -> CA'
A' -> aA' | bCA' | ∈
B -> CA'aB' | bbB'
B' -> bB' | ∈
C -> c

5) (10) Use left factoring to resolve the pairwise disjointness problems in the following grammar:

A -> aBc | ac | a

B -> b | aB

A -> aA'

A' -> Bc | c | ∈

B -> b | aB

6) (20 pts) Create an LR(0) parse table for the following grammar. Show all steps (creating closures, the DFA, the transition table, and finally the parse table):

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

T -> (E) | id

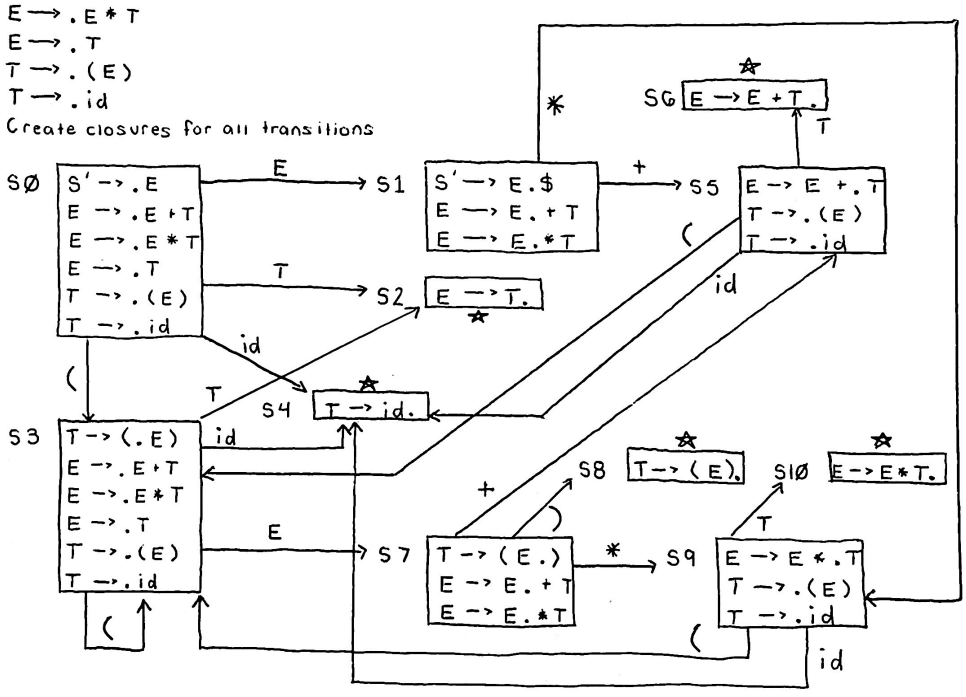
E → E + T | E * T | T
T → (E) | id
Rewrite our rules (no |'s) and add a starting rule

	Number the rules:
S' → E $	r0 : S' → E $
E → E + T	r1 : E → E + T
E → E * T	r2 : E → E * T
E → T	r3 : E → T
T → (E)	r4 : T → (E)
T → id	r5 : T → id

Create closures for rules

S' → .E
E → .E + T
E → .E * T
E → .T
T → .(E)
T → .id

Create closures for all transitions



(☆ indicates a reduction state)

	+	*	(	)	id	E	T
S Ø			3		4	1	2
S 1	5	9					
S 2							
S 3			3		4	7	2
S 4							
S 5			3		4		6
S 6							
S 7	5	9		8			
S 8							
S 9			3		4		10
S 10							

	Action						Goto	
State								
	+	*	(	)	id	$	E	T
Ø			S3		S4		1	2
1	S5	S9				acc		
2	r3	r3	r3	r3	r3	r3		
3			S3		S4		7	2
4	r5	r5	r5	r6	r5	r5		
5			S3		S4			6
6	r1	r1	r1	r1	r1	r1		
7	S5	S9		S8				
8	r4	r4	r4	r4	r4	r4		
9			S3		S4			10
10	r2	r2	r2	r2	r2	r2		

r Ø : S' → E $
r 1 : E → E + T
r 2 : E → E * T
r 3 : E → T
r 4 : T → (E)
r 5 : T → id

7) (20 pts) Show a complete bottom-up parse, including the parse stack contents, input string, and action for the string below using the parse table you created in step 6. Think about how I went through this in class.

(id + id) * id

STACK	INPUT	ACTION
0	.(id + id) * id $	Shift 3
0(3	(.id + id) * id $	Shift 4
0(3id4	(id. + id) * id $	Reduce 5
0(3T2	(id. + id) * id $	Reduce 3
0(3E7	(id. + id) * id $	Shift 5
0(3E7+5	(id +. id) * id $	Shift 4
0(3E7+5id4	(id + id.) * id $	Reduce 5
0(3E7+5T6	(id + id.) * id $	Reduce 1
0(3E7	(id + id.) * id $	Shift 8
0(3E7)8	(id + id). * id $	Reduce 4
0T2	(id + id). * id $	Reduce 3
0E1	(id + id). * id $	Shift 9
0E1*9	(id + id) *. id $	Shift 4
0E1*9id4	(id + id) * id. $	Reduce 5
0E1*9T10	(id + id) * id. $	Reduce 2
0E1	(id + id) * id. $	Shift 1
0E1$	(id + id) * id $.	Accept

8) (10 pts) Show a rightmost derivation for the string above, and show how the bottom-up parse you completed in step 7 correctly finds all of the handles for the input string above.

E -> E * T (prune id)

-> E * id (prune T) (handle is id)

-> T * id (prune (E)) (handle is T)

-> (E) * id (prune E + T) (handle is (E))

-> (E + T) * id (prune id) (handle is E + T)

-> (E + id) * id (prune T) (handle is id)

-> (T + id) * id (prune id) (handle is T)

-> (id + id) * id (handle is id)