Compilers Solutions

1. (20 points)

   false. Every LR(2) LANGUAGE is also LR(1), but it may be necessary to modify an LR(2) CFG to get an equivalent LR(1) CFG.

   true. This is easily proved by sketching parse trees.

   true.

   true. If a leftmost derivation has $A \to \alpha \stackrel{*}{\Longrightarrow} \epsilon$, that can be replaced by $A \to \beta \stackrel{*}{\Longrightarrow} \epsilon$ to yield another leftmost derivation for the same terminal string.

   true. An unambiguous grammar cannot have more than one *leftmost* derivation for the same string, but there may be several derivations for the string, at most one of which is *leftmost*.

   true.

   true. Just delete the useless symbols and all productions that include them.

   false. The SLR(1) parser may perform more reductions before it "notices" that the next input doesn't work.

   false. Suppose you have a CFG with a lot of ϵ productions.

   true. There is an algorithm for eliminating all left recursion in the text.

2. (7 points) *a b a |a c |a b a |a a |b a |a*

3. (15 points)

   (a) row 3, col %, delete entry r1
   row 3, col #, delete entry s4
   row 3, col @, delete entry s7
   row 5, col %, delete entry s2
   row 5, col #, delete entry r2
   row 5, col @, delete entry s7

   (b)

Stack	Input
0	id # id % id $
0 6	# id % id $
0 1	# id % id $
0 1 4	id % id $
0 1 4 6	% id $
0 1 4 5	% id $

Stack	Input
0 1	% id $
0 1 2	id $
0 1 2 6	$
0 1 2 5	$
0 1	$
accept	$

4. (8 points)

```
expr    :       expr '+' expr   { printf("add"); }

        |       expr '-' expr   { printf("sub"); }

        |       expr '*' expr   { printf("mul"); }

        |       varname         { /* do nothing */ }

        |       '(' expr ')'        { /* do nothing */ }

        |       literal         { /* do nothing */ }
        ;
```

5. (10 points)

Here is what YACC would do if you included actions before and after the expression:

```
whileloop       :  WHILE M1 expr M4  DO stmt
                   {
                     printf("L%d: ", $4); /* end label */
                     printf("jump L%d\n", $2);
                   }
                ;

M1              : /* empty */  { printf("L%d: ", $$=newlabel()); }

M2              : /* empty */  { printf("br_false L%d\n", $$=newlabel()); }
```

The grammar can also be broken up in different ways.

It is important to keep the labels on a stack, so that labels for inner loops do not over-write labels for enclosing loops. In the solution above, we are using the value stack that YACC maintains (where it keeps **$2**, etc.).