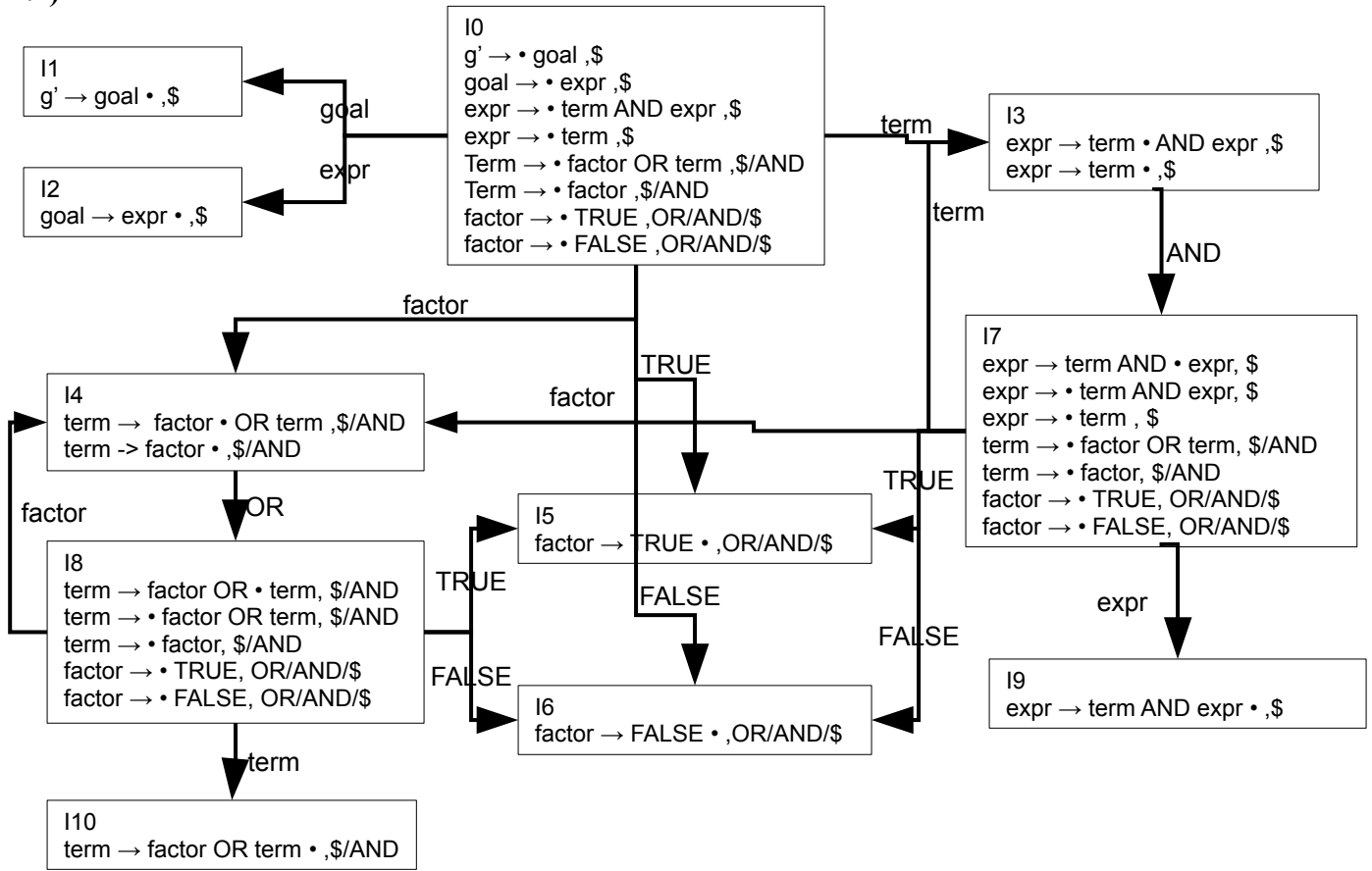


# CS 515 Homework 2 Sample Solution

## 1.1)



## 1.2)

State	AND	OR	TRUE	FALSE	\$	goal	expr	term	factor
0		s5	s6			1	2	3	4
1					acc				
2					r1				
3	s7				r3				
4	r5	s8			r5				
5	r6	r6			r6				
6	r7	r7			r7				
7			s5	s6			9	3	4
8			s5	s6				10	4
9					r2				
10	r4				r4				

**1.3)**

The grammar is LR(1) because, as the graph shows, for each state/lookahead pair there is only one possible shift or reduce action. This is also reflected in the table, as there is a maximum of one entry per cell.

**1.4)**

Stack	Remaining Input Tokens	Action
0	TRUE OR TRUE AND FALSE	s5
0 5	OR TRUE AND FALSE	r6 (pop 1)
0 "factor"	OR TRUE AND FALSE	goto 4
0 4	OR TRUE AND FALSE	s8
0 4 8	TRUE AND FALSE	s5
0 4 8 5	AND FALSE	r6 (pop 1)
0 4 8 "factor"	AND FALSE	goto 4
0 4 8 4	AND FALSE	r5 (pop 1)
0 4 8 "term"	AND FALSE	goto 10
0 4 8 10	AND FALSE	r4 (pop 3)
0 "term"	AND FALSE	goto 3
0 3	AND FALSE	s7
0 3 7	FALSE	s6
0 3 7 6	\$	r7 (pop 1)
0 3 7 "factor"	\$	goto 4
0 3 7 4	\$	r5 (pop 1)
0 3 7 "term"	\$	goto 3
0 3 7 3	\$	r3 (pop 1)
0 3 7 "expr"	\$	goto 9
0 3 7 9	\$	r2 (pop 3)
0 "expr"	\$	goto 2
0 2	\$	r1 (pop 1)
0 "goal"	\$	goto 1
0 1	\$	accept

2)

The grammar is not SLR(1) because of a shift-reduce conflict. The first state of the canonical collection contains the following items:

$S' \rightarrow \cdot S$   
 $S \rightarrow \cdot Aa$   
 $S \rightarrow \cdot bAc$   
 $S \rightarrow \cdot dc$   
 $S \rightarrow \cdot bda$   
 $A \rightarrow \cdot d$

Now note that on seeing a “d” on the input stream we shift into a state with the following items:

$S \rightarrow d \cdot c$   
 $A \rightarrow d \cdot$

This represents a shift-reduce conflict; we don’t know whether to shift the “c” or reduce on  $A \rightarrow D$ .

Adding the LR(1) lookahead solves this. Here is the initial state:

$S' \rightarrow \cdot S, \$$   
 $S \rightarrow \cdot Aa, \$$   
 $S \rightarrow \cdot bAc, \$$   
 $S \rightarrow \cdot dc, \$$   
 $S \rightarrow \cdot bda, \$$   
 $A \rightarrow \cdot d, a$

Note that the lookahead is different for  $A \rightarrow D$ . After seeing a “d”, we shift into the following state:

$S \rightarrow d \cdot c, \$$   
 $A \rightarrow d \cdot, a$

Now we know to reduce only if we see “a” in the lookahead. The remainder of the LR(1) construction proceeds similarly.

### 3) Dynamic reference counting with attribute grammars

For the following translation scheme, assume the grammar symbols PROGRAM, STMT\_LIST, STMT, FOR\_STMT, A\_STMT, READ\_STMT, WRITE\_STMT, and EXPR all have attribute reftabl, which is a table of (variable, count) pairs. Function NewRefTable() creates an empty reference table. Function addn(reftabl, var, n) adds n to the reference count for var to the table reftabl, inserting the variable if it doesn't exist. Function lookup(reftabl, var) returns the number of counts for id in the table. Function merge(reftabl, reftabl) merges two tables, adding together the reference counts, and returns a new table.

The final list is found in PROGRAM's reftabl.

```
PROGRAM ::= procedure STMT_LIST
        { $$reftabl = $2.reftabl; }
STMT_LIST ::= STMT ; STMT_LIST
        { $$reftabl = merge($1.reftabl, $3.reftabl); }
| STMT { $$reftabl = $1.reftabl; }
STMT ::= FOR_STMT { $$reftabl = $1.reftabl; }
| A_STMT { $$reftabl = $1.reftabl; }
| READ_STMT { $$reftabl = $1.reftabl; }
| WRITE_STMT { $$reftabl = $1.reftabl; }
FOR_STMT ::= for id := const to const begin STMT_LIST end
        { int iters = $6.val - $4.val + 1;
          // add each variable's count n-1 more times.
          for [id in $8.reftabl] {
            int newcounts = (iters - 1) * lookup($8.reftabl, id);
            addn($8.reftabl, id, newcounts);
          }
          $$reftabl = $8.reftabl;
        }
A_STMT ::= id := EXPR
        { addn($3.vt, $1.str, 1);
          $$reftabl = $3.vt; }
READ_STMT ::= read ( id )
        { $$reftabl = NewRefTable();
          addn($$.reftabl, $3.str, 1); }
WRITE_STMT ::= write ( EXPR )
        { $$reftabl = $3.reftabl; }
EXPR ::= EXPR + EXPR
        { $$reftabl = merge($1.reftabl, $3.reftabl); }
| EXPR * EXPR
        { $$reftabl = merge($1.reftabl, $3.reftabl); }
| id
        { $$reftabl = NewRefTable();
          addn(vt, $1.str, 1); }
| const { }
```

#### 4.1)

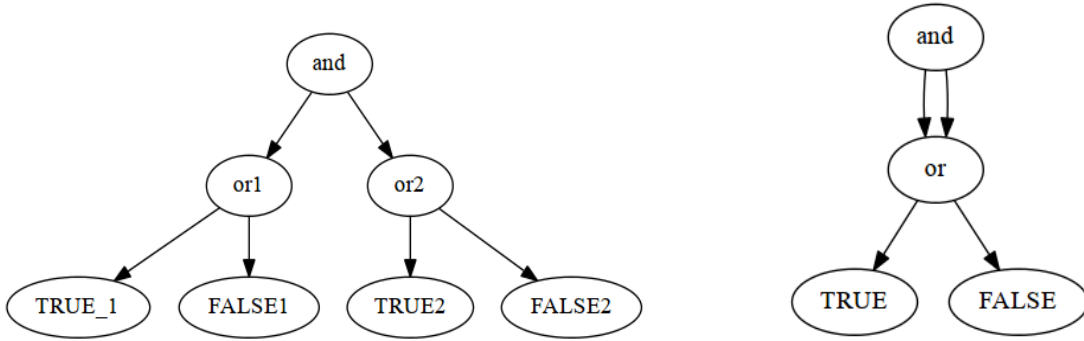
```
1 goal      ::= expr
  { goal.ptr = expr.ptr }
2 expr      ::= term AND expr1
  { expr.ptr = newOP(AND, term.ptr, expr1.ptr) }
3          | term
  { expr.ptr = term.ptr }
4 term      ::= factor OR term1
  { expr.ptr = newOP(OR, factor.ptr, term1.ptr) }
5          | factor
  { expr.ptr = factor.ptr }
6 factor    ::= TRUE
  { factor.ptr = newCONST(true) }
7          | FALSE
  { factor.ptr = newCONST(false) }
```

#### 4.2)

(newOPnode and newCONSTnode also add the newly created node to the DAG.)

```
1 goal      ::= expr
  { goal.ptr = expr.ptr }
2 expr      ::= term AND expr1
  { if (n = node(<AND, term.ptr, expr1.ptr>))
    then expr.ptr = n
    else term.ptr = newOP(AND, term.ptr, expr1.ptr)
      expr.ptr = newOPnode(AND, term.ptr, expr1.ptr) }
3          | term
  { expr.ptr = term.ptr }
4 term      ::= factor OR term1
  { if (n = node(<OR, expr.ptr, term1.ptr>))
    then term.ptr = n
    else term.ptr = newOPnode(OR, factor.ptr, term1.ptr) }
5          | factor
  { expr.ptr = factor.ptr }
6 factor    ::= TRUE
  { if (n = node(<true>)) then factor.ptr = n
    else factor.ptr = newCONSTnode(true) }
7          | FALSE
  { if (n = node(<false>)) then factor.ptr = n
    else factor.ptr = newCONSTnode(false) }
```

4.3)



4.4)

To capture commutativity, we would simply add an additional lookup for a node with the order of arguments to the binary operations swapped, for example:

```
if node(<AND, term.ptr, expr1.ptr>) || node(<AND, expr1.ptr, term.ptr>)
. . .
```

For associativity, we can add a lookup to see whether the left-associative version of the operation exists in the DAG, for example:

```
expr ::= term AND expr1
    { . . .
      if ((expr1.op = AND) &&
          (n1 = node(<AND, term.ptr, expr1.ptr.left>)) &&
          (n = node(<AND, n1, expr1.ptr.right>)))
      then expr.ptr = n
      . . . }
```