Problem 1: Finite Automata and Regular Expressions [30 points]

Given the regular expression $RE = a.(b^*).(aa \mid a)$ over the alphabet = \{a,b\} answer the following questions:

(a) [10 points] Derive the NFA using the Thompson construction that corresponds to this RE.
(b) [10 points] Convert this NFA to a DFA using the closure computation.
(c) [10 points] Minimize the resulting DFA and derive the complement DFA.

Answer:
(a) See NFA below.

(b) The subset construction yields the DFA on below where we have noted the subset each state stands for.

(c) Using the iterative partition refinement we get the same DFA as above. In this case the DFA resulting from the subset construction was already minimal.
**Problem 2: LR Bottom-Up Parsing Algorithms [40 points]**

Consider the following grammar $G$ for expressions and lists of statements (StatList) using assignment statements (Assign) and basic expressions (Expr) with the productions presented below and already augmented by the initial production $G \rightarrow \text{StatList }$.

1. $G \rightarrow \text{StatList }$
2. $\text{StatList } \rightarrow \text{Stat } \cdot \text{; } \text{StatList }$
3. $\text{StatList } \rightarrow \text{Stat }$
4. $\text{Stat } \rightarrow \text{Assign }$
5. $\text{Assign } \rightarrow \text{id } \cdot \text{=} \text{ Expr }$
6. $\text{Expr } \rightarrow \text{id }$
7. $\text{Expr } \rightarrow \text{const }$

For the grammar presented above determine the following:

(a) [10 points] Compute the DFA that recognizes the set of LR(0) items.
(b) [10 points] Construct the LR(0) parsing table.
(c) [10 points] Identify the nature of and explain how to resolve any conflicts in the table found in (b).
(d) [10 points] For the erroneous input “id = 2; id” describe the possible actions of the parser to allow it to recover from this input sentence.

**Answer:**

(a) The set of LR(0) items and the corresponding DFA that identifies them is depicted below where for each state (labeled at the upper right corner) we indicate the items and the transition on terminals and non-terminals.

(b) The LR(0) parsing table is shown below.
(c) As can be seen there is a shift/reduce conflict in state 2 resulting from the fact that on a terminal ‘;’ and having seen a Statement the parser could reduce immediately by the production rule (2). But because there might be a valid statement list coming next, there is also a possibility of executing a shift operation and moving to state 3. Typically, as one tries to parse the largest possible sentences preference is given to a shift operation.

(d) For this particular input the parser will have executed the following operations: shift 6, shift7, shift9, reduce (6), goto 10, reduce (4), goto 5, reduce (3), goto 2, shift3, shift6 and the end of which a $ terminal is seen leading to an error condition. To recover from this situation a possible sequence is to pop the “id” and ‘;’ tokens from the stack (along with the corresponding states 5 and 3) and in state 2 instead of performing a shift operation execute a reduce (2) which will lead to a goto 1 and an accept.
Problem 3: Attributive Grammar for Code Generation [30 points]
Consider the following grammar G basic expressions (Expr) and assignment statements (Assign) using the productions presented below. Here const stands for an integer constant and id stands for a scalar variable. The input program is assumed to be syntactically correct.

(1) Expr → Expr + Expr
(2) Expr → const
(3) Expr → id
(4) Assign → id = Expr
(5) Assign → Expr += Expr

Using a three-address instruction target machine write a syntax-directed definition to generate code for Assign and Expr implementing an optimization called constant folding. In constant folding whenever two sub-expressions of an arithmetic operation are denoted by a constant the compiler performs the operator at compile time and replaces the sub-expressions with the result. For instance, if the input program includes the expression (1 + 2) the compiler can replace it with the value 3. Focus only on the addition arithmetic operator as this optimization can also be extended to other binary operator. Describe briefly the attribute of your translation and auxiliary functions used. You should also try to minimize the number of temporary variables used.

Answer:
A possible solution, not necessarily unique, is to have an attribute setting the type of an expression as a constant. This attribute is used to determine if the arithmetic operator should be done at compile time whenever the two sub-expressions are integer.

Expr → id { Expr.place = nil; Expr.kind = var; Expr.value = id.value }
Expr → const { Expr.place = nil; Expr.kind = const; Expr.val = const.value; }
Expr0 → Expr1 + Expr2 { if(Expr1.kind == const && Expr2.kind == const) {
    Expr0.kind = const; Expr0.place = nil; Expr0.value = Expr1.value + Expr2.value;
} if(Expr1.kind == const && Expr2.kind == var) { /* repeat for (var, const) and (var, var) */
    t = newTemp();
    Expr0.kind = temp; Expr0.place = t; emit('t = Expr1.value + IdString(Expr2.value));
} if(Expr1.kind == temp && Expr2.kind == var) { /* no need for new temp – reuse */
    Expr0.kind = temp; Expr0.place = Expr1.place;
    emit('Expr1.place=Expr1.place+ IdString(Expr2.value));
} /* repeat this for (var, temp) and (temp, temp) reusing the first temp when possible */
/* release the second temp in the case of (temp, temp) */
} }
Assign → id = Expr { if(Expr.kind == const) emit('IdString(id.value) = Expr.value');
if(Expr.kind == var) emit('IdString(id.value) = IdString(Expr.value)'); /* could check if diff */
if(Expr.kind == temp) {
    emit('IdString(id.value) = Expr.place'); releaseTemp(Expr.place); }
}
Assign → Expr1 += Expr2 { if(Expr1.kind != var) error();
if(Expr2.kind == var) /* could check if diff */
    emit('IdString(Expr1.value) = IdString(Expr1.value) + IdString(Expr2.value)');
if(Expr2.kind == const)
    emit('IdString(Expr1.value) = IdString(Expr1.value) + Expr2.value');
if(Expr2.kind == temp)
    emit('IdString(Expr1.value) = IdString(Expr1.value) + Expr2.place');
    releaseTemp(Expr2.place);
}