**Algorithm 11: Conversion of NFA to DFA**

**Aim:** Convert a given NFA to an equivalent DFA.  
Step 1: Start.  
Step 2: Read the NFA states, input symbols, start state, and transition table.  
Step 3: Initialize the DFA state list with the NFA start state as the first DFA state.  
Step 4: Mark the first DFA state as unprocessed.  
Step 5: Select an unprocessed DFA state.  
Step 6: For each input symbol, compute the set of NFA next states reachable from any state in the selected DFA state.  
Step 7: If the computed set is new, add it to the DFA state list and mark it unprocessed.  
Step 8: Record the DFA transition from the selected state on the input symbol to the computed set.  
Step 9: Mark the selected DFA state as processed.  
Step 10: If any DFA state remains unprocessed, go to Step 5.  
Step 11: Identify DFA final states as any DFA state containing at least one NFA final state.  
Step 12: Print the DFA transition table.  
Step 13: Stop.

**Algorithm 12: Recursive Descent Parser**

**Aim:** Parse arithmetic expressions using recursive descent.  
Step 1: Start.  
Step 2: Define grammar E→T E′, E′→+ T E′ | ε, T→F T′, T′→\* F T′ | / F T′ | ε, F→(E) | id | num.  
Step 3: Read the input string and set index i = 0 and error = 0.  
Step 4: Call function E().  
Step 5: In E(), call T() then E′().  
Step 6: In E′(), if current symbol is ‘+’, advance i, call T(), then call E′(); otherwise return.  
Step 7: In T(), call F() then T′().  
Step 8: In T′(), if current symbol is ‘\*’ or ‘/’, advance i, call F(), then call T′(); otherwise return.  
Step 9: In F(), if ‘(’, advance i, call E(), require ‘)’, else if number, consume digits, else if identifier, consume letters/digits/‘\_’, else set error = 1.  
Step 10: After E() returns, if i equals input length and error = 0, accept; otherwise reject.  
Step 11: Stop.

**Algorithm 13A: FIRST of a Grammar**

**Aim:** Compute FIRST sets for all nonterminals.  
Step 1: Start.  
Step 2: Read the productions of the grammar.  
Step 3: Initialize FIRST(X) = ∅ for every nonterminal X.  
Step 4: For each production A→α, if α begins with a terminal a, add a to FIRST(A).  
Step 5: For each production A→X₁X₂…Xk, add FIRST(X₁) minus ε to FIRST(A).  
Step 6: If FIRST(X₁) contains ε, add FIRST(X₂) minus ε to FIRST(A), and continue this process until a symbol does not derive ε or all symbols are processed.  
Step 7: If all X₁…Xk derive ε, add ε to FIRST(A).  
Step 8: Repeat Steps 4–7 until no FIRST set changes.  
Step 9: Output FIRST sets.  
Step 10: Stop.

**Algorithm 13B: FOLLOW of a Grammar**

**Aim:** Compute FOLLOW sets for all nonterminals.

Step 1: Start.  
Step 2: Read the productions of the grammar and the start symbol S.  
Step 3: Compute FIRST sets for all grammar symbols.  
Step 4: Initialize FOLLOW(X) = ∅ for every nonterminal X.  
Step 5: Add $ to FOLLOW(S).  
Step 6: For each production A → αBβ, add FIRST(β) minus ε to FOLLOW(B).  
Step 7: For each production A → αB or A → αBβ where FIRST(β) contains ε, add FOLLOW(A) to FOLLOW(B).  
Step 8: Repeat Steps 6–7 until no FOLLOW set changes.  
Step 9: Output FOLLOW sets.  
Step 10: Stop.

**Algorithm 14: Shift–Reduce Parser**

**Aim:** Recognize strings using a shift–reduce parser.  
Step 1: Start.  
Step 2: Push $ onto the stack and append $ to the input.  
Step 3: If the handle exists on the stack top, perform a reduce operation by replacing the handle with its LHS.  
Step 4: Otherwise shift the next input symbol onto the stack.  
Step 5: If the stack becomes S$ and input is $, accept.  
Step 6: If no shift or reduce action applies, reject.  
Step 7: If input not exhausted or more reductions possible, go to Step 3.  
Step 8: Stop.

**Algorithm 15: Intermediate Code Generation**

**Aim:** Generate three-address code from an expression.  
Step 1: Start.  
Step 2: Read the input expression as a string.  
Step 3: Scan the expression to locate operators in decreasing precedence order.  
Step 4: For the next operator found, identify its left operand and right operand.  
Step 5: Create a new temporary Ti and generate the statement Ti := left op right.  
Step 6: Replace the subexpression in the string with Ti.  
Step 7: If more operators remain, go to Step 4.  
Step 8: Generate the final assignment target := lastTemp or target := expression as applicable.  
Step 9: Print the sequence of three-address statements.  
Step 10: Stop.

**Algorithm 16: Back End (Target Code Generation)**

**Aim:** Translate three-address code to simple 8086-like assembly.  
Step 1: Start.  
Step 2: Read each three-address statement of the form x = y op z.  
Step 3: Load y into a register Ri using MOV Ri, y.  
Step 4: Map op to ADD, SUB, MUL, or DIV.  
Step 5: Emit the instruction OP Ri, z.  
Step 6: Store the result using MOV x, Ri.  
Step 7: Repeat Steps 2–6 for all statements.  
Step 8: Stop.

**Algorithm 17: SLR Parser**

**Aim:** Parse input using SLR ACTION and GOTO tables.  
Step 1: Start.  
Step 2: Push state 0 onto the stack and append $ to the input.  
Step 3: Let s be the state on top of the stack and a be the current input symbol.  
Step 4: Read ACTION[s, a].  
Step 5: If ACTION is shift t, push a and state t onto the stack and advance input.  
Step 6: If ACTION is reduce A→β, pop 2|β| entries from the stack, let s′ be new top, push A, then push GOTO[s′, A].  
Step 7: If ACTION is accept, report success and stop.  
Step 8: If ACTION is error, report rejection and stop.  
Step 9: Go to Step 3.  
Step 10: Stop.