Assignment No. 3

Name:

M.Zain Ul Abideen

Reg No:

FA22-BCS-090

Section:

FA22-BCS-7-B

Subject:

Compiler Construction

Date:

Dec 05, 2025

Submitted to:

Mr. Nasir Mehdi

Q.No.1

Apply an SDT scheme to implement the translation and demonstrate it on the following inputs:

Grammer:

E -> E + T

E -> E - T

E -> T

T -> T * F

T -> T / F

T - > F

F -> (E)

F -> id

1. **(a + b) * c**
   Parse Tree:

```
       E
       |
       T
       |
   T  *  F
   |     |
   F     id
   |
  (E)
   |
 (E + T)
  |    |
 (T + F)
  |    |
 (F + id)
  |
  id
```

Semantic Actions Table

Grammar	Actions
F -> id	id (a)
T -> F	No action
F -> id	id (b)
T -> T * F	print (‘*’)
E -> T	No action
F -> id	id (c)
T -> F	No action
F -> id	id (d)
T -> T/F	print (‘/’)
E -> E + T	print (‘+’)

Postfix: a b + c *

2. `a * b + c / d`

Parse Tree:

```
        E
        |
    E   +   T
    |       |
    T      T / F
    |      |   |
  T * F    F   id
  |   |    |
  F   id   id
  |
  id
```

Semantic Actions Table

Grammar	**Actions**
F -> id	id (a)
T -> F	No action
F -> id	id (b)
T -> T * F	print (‘*’)
E -> T	No action
F -> id	id (c)
T -> F	No action
F -> id	id (d)
T -> T/F	print (‘/’)
E -> E + T	print (‘+’)

postfix: a b * c d / +

3. **a + (b * (c + d))**

Parse Tree:

```
      E
      |
  E   +   T
  |        |
  T       F
  |        |
  F      (E)
  |        |
  id       T
           |
      T   *   F
      |       |
      F      (E)
      |       |
      id    (E + T)
              |   |
              T   F
              |   |
              F   id
              |
              id
```

Semantic Actions Table

Grammer	Actions
F -> id	id (a)
T -> F	No action
E -> T	No action
F -> id	Id(b)
T -> F	No action
F -> id	id (c)
T -> F	No action
E -> T	No action
F -> id	id (d)
E -> E + T	print ('+')
F -> (E)	No action
T -> T * F	print ('*')
E -> T	No action
F -> (E)	No action
T -> F	No action
E -> E + T	print ('+')

postfix: a b c d + * +

Q.No.2

Given, a Directed Acyclic Graph (DAG) representing tasks and their dependencies:
Vertices (tasks): {A, B, C, D, E, F}
Edges (dependencies): {(A → B), (A → C), (C → D), (D → E), (B → E), (E → F)}

Perform a topological sort of the graph and print the order of tasks using the following steps:

1. **Construct Adjacency Lists:** Represent the graph using adjacency lists.
2. **Use a Recursion Stack:** Use a Depth-First Search (DFS)-based approach to explore the graph. As each node's dependencies are resolved, add the node to a recursion stack.

Output the Recursion Stack: Once all nodes are processed, the recursion stack will contain the topological order in reverse. Print the reversed stack.

Given DAG:

- **Vertices (Tasks):** {A, B, C, D, E, F}
- **Edges (Dependencies):** {(A → B), (A → C), (C → D), (D → E), (B → E), (E → F)}
- **Meaning of edges:** A → B means **Task B cannot start until Task A is completed**.

Adjacency Lists

- The adjacency list represents all tasks that depend on a given task.
- It helps in quickly finding all dependent tasks during traversal.
- **A → B, C** → Tasks B and C depend on A
- **B → E** → Task E depends on B
- **C → D** → Task D depends on C
- **D → E** → Task E depends on D
- **E → F** → Task F depends on E
- **F** → – → Task F has no dependent tasks

Using DFS for Topological Sorting

- **Topological Sorting** gives a linear ordering of tasks such that **for every directed edge U → V, U appears before V** in the ordering.
- **Depth-First Search (DFS)** is commonly used because it allows us to **process all dependencies of a task before the task itself**.
- **Recursion Stack Concept:**
  - As we perform DFS, we **push a task onto a stack only after visiting all its dependent tasks**.
  - Reversing this stack gives the topological order.

DFS Process

Start with node A:

1. **Visit A**
   → Go to neighbor **B**
2. **Visit B**
   → Go to neighbor **E**
3. **Visit E**
   → Go to neighbor **F**
4. **Visit F**
   - F has no outgoing edges
     → **F finishes** → push F onto stack
     Stack: [F]
5. **Back to E**

    - All dependencies done
      → **E finishes** → push E onto stack
      Stack: [F, E]

6. **Back to B**

    - All dependencies done
      → **B finishes** → push B onto stack
      Stack: [F, E, B]

7. **Return to A**
   → Visit next neighbor **C**

8. **Visit C**
   → Go to neighbor **D**

9. **Visit D**
   → Go to neighbor **E** (already visited & processed)
   → **D finishes** → push D onto stack
   Stack: [F, E, B, D]

10. **Back to C**
    → **C finishes** → push C onto stack
    Stack: [F, E, B, D, C]

11. **Back to A**
    → **A finishes** → push A onto stack
    Stack: [F, E, B, D, C, A]

Generating Topological Order

- **Reverse the recursion stack** to get the order in which tasks can be executed:
- **Topological Order: A, C, D, B, E, F**