

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:

$E \rightarrow E + T$

$E \rightarrow E - T$

$E \rightarrow T$

$T \rightarrow T * F$

$T \rightarrow T / F$

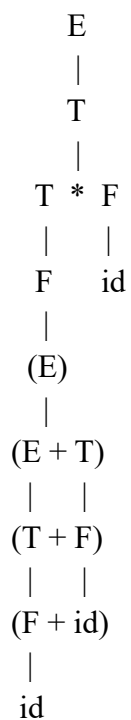
$T \rightarrow F$

$F \rightarrow (E)$

$F \rightarrow id$

1. $(a + b) * c$

Parse Tree:



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
  
```

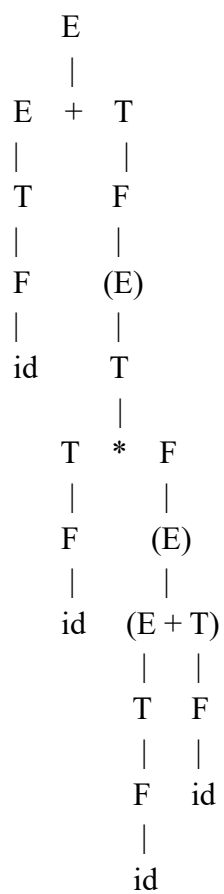
Semantic Actions Table

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:



Semantic Actions Table

Grammar	Actions
F \rightarrow id	id (a)
T \rightarrow F	No action
E \rightarrow T	No action
F \rightarrow id	Id(b)
T \rightarrow F	No action
F \rightarrow id	id (c)
T \rightarrow F	No action
E \rightarrow T	No action
F \rightarrow id	id (d)
E \rightarrow E + T	print ('+')
F \rightarrow (E)	No action
T \rightarrow T * F	print ('*')
E \rightarrow T	No action
F \rightarrow (E)	No action
T \rightarrow F	No action
E \rightarrow 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 \rightarrow B), (A \rightarrow C), (C \rightarrow D), (D \rightarrow E), (B \rightarrow E), (E \rightarrow 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 \rightarrow B), (A \rightarrow C), (C \rightarrow D), (D \rightarrow E), (B \rightarrow E), (E \rightarrow F)}
- **Meaning of edges:** A \rightarrow 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.
- $A \rightarrow B, C \rightarrow$ Tasks B and C depend on A
- $B \rightarrow E \rightarrow$ Task E depends on B
- $C \rightarrow D \rightarrow$ Task D depends on C
- $D \rightarrow E \rightarrow$ Task E depends on D
- $E \rightarrow F \rightarrow$ Task F depends on E
- $F \rightarrow - \rightarrow$ 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 $U \rightarrow V$, 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**