PRINCIPLES OF COMPILER DESIGN - SYNTAX ANALYSIS

Individual Assignment 01

Student Number: 41

Name: Abrham Abebaw
Course Code: SEng4031
Id: BDU1504862
Section: B

Table of Contents

Question 1: Theory

Explain FIRST and FOLLOW Sets in Context-Free Grammars.

FIRST Set in Context-Free Grammars

Definition of FIRST Set

In **syntax analysis**, the **FIRST set** of a grammar symbol **X** is defined as the set of all terminal symbols that can appear as the first symbol in any string derived from X.

Formally:

FIRST(X) = {a | X ⇒* aα, where *a* is a terminal and α is any string of grammar symbols}

If **X can derive the empty string (ε)**, then **ε is also included in FIRST(X)**.

The FIRST set is a fundamental concept used in **top-down parsing**, especially in **LL (1) parsers**, to determine which production rule should be applied when parsing an input string.

Rules to Compute FIRST Sets

The FIRST set is computed by systematically analyzing the grammar productions using the following rules:

Rule 1: FIRST of a Terminal

If **X is a terminal symbol**, then the FIRST set contains only that terminal itself.

$$\mathrm{FIRST}(X) = \{X\}$$

Explanation:
A terminal symbol cannot derive any other symbol; therefore, it must appear first in any string derived from itself.

Example:
If X = a, then:

FIRST(a) = {a}

Rule 2: FIRST of ε (Empty String)

If a non-terminal **X has a production that directly derives ε**, then ε is included in FIRST(X).

$$X \to \varepsilon \Rightarrow \varepsilon \in \mathrm{FIRST}(X)$$

Explanation:
This indicates that the non-terminal can generate an empty string and may not contribute any terminal symbol to the beginning of a derived string.

Rule 3: FIRST of a Non-Terminal with Multiple Symbols on the Right-Hand Side

If $\mathbf{X \rightarrow Y_1Y_2\ldots Y_n}$, then FIRST(X) is determined as follows:

1. Add **FIRST(Y_1)** (excluding ε) to **FIRST(X)**
2. If **FIRST(Y_1)** contains ε, then:
    - Add **FIRST(Y_2)** (excluding ε)
3. Continue this process for $\mathbf{Y_3, Y_4, \ldots}$
4. If **all $Y_1, Y_2, \ldots, Y_n$ can derive ε**, then:
    - Add **ε to FIRST(X)**

Explanation:
Since $\mathbf{Y_1}$ appears first in the production, its FIRST set determines what symbols can start strings derived from **X**.
However, if **Y_1 can disappear (derive ε)**, then the parser must look at $\mathbf{Y_2}$, and so on.

Example

Consider the grammar:

S → ABC

A → ε

B → b

C → c

- FIRST(A) = {ε}
- FIRST(B) = {b}
- FIRST(C) = {c}

Now computing FIRST(S):

- FIRST(A) contains ε → look at FIRST(B)
- FIRST(B) = {b}

So:

FIRST(S) = {b}

Purpose and Importance of FIRST Sets

1. Selecting the Correct Production Rule

In **predictive (LL) parsing**, the parser uses the FIRST set to decide **which production rule to apply** based on the **next input symbol**.

2. Construction of LL (1) Parsing Tables

FIRST sets are a **core component** in building **LL (1) parse tables**.
Each table entry is filled using FIRST sets to ensure deterministic parsing with one symbol lookahead.

3. Handling ε-Productions

When a grammar contains ε-productions, FIRST sets help determine:

- Whether a non-terminal can be skipped
- When FOLLOW sets must be consulted

4. Detection of Grammar Ambiguity and Conflicts

If two different productions for the same non-terminal have **overlapping FIRST sets**, the grammar:

- Is **not LL (1)**
- May be **ambiguous** or unsuitable for predictive parsing

FOLLOW Set in Context-Free Grammars

Definition of FOLLOW Set

In **syntax analysis**, the **FOLLOW set** of a non-terminal symbol **A** is defined as the **set of all terminal symbols that can appear immediately to the right of A** in **some sentential form** derived from the grammar.

Formally:

FOLLOW(A) = $\{a \mid S \Rightarrow^* \alpha A\, a\beta$, where a is a terminal$\}$

If **A appears at the end of a derived string**, the **end-of-input marker ($)** may also belong to FOLLOW(A).

The FOLLOW set is mainly used in **top-down parsing**, especially in **LL (1) parsers**, to determine **when a non-terminal has completed its derivation**.

Rules to Compute FOLLOW Sets

The FOLLOW set is computed using the following systematic rules:

Rule 1: End Marker Rule

Place the **end-of-input marker $** in the FOLLOW set of the **start symbol S**.

$$\$ \in \text{FOLLOW}(S)$$

Explanation:
The start symbol represents the entire input. When parsing is complete, the parser expects the **end of input**, which is represented by $.

Rule 2: Non-Terminal Followed by Symbols

If there is a production of the form:

$A \rightarrow \alpha\ B\ \beta$

then:

$$\text{FIRST}(\beta) - \{\varepsilon\} \subseteq \text{FOLLOW}(B)$$

Explanation:

- **β** appears immediately after **B**
- Any terminal that can begin strings derived from **β** can also appear immediately after **B**
- ε is excluded because it does not produce a visible terminal

Rule 3: Non-Terminal at the End or Followed by ε-Producing Symbols

If:

$A \rightarrow \alpha\ B$

or

$A \rightarrow \alpha\ B\ \beta$ where FIRST(β) contains ε

then:

$$\text{FOLLOW}(A) \subseteq \text{FOLLOW}(B)$$

Explanation:

- If **B is at the end**, nothing follows it directly
- If **β can disappear (derive ε)**, then what follows **A** can also follow **B**
- Therefore, FOLLOW(A) is propagated to FOLLOW(B)

Illustrative Example

Consider the grammar:

S → A B

A → a | ε

B → b

Step-by-step FOLLOW computation:

1. Start symbol:

FOLLOW(S) = {$}

2. From S → A B:

- B is at the end → FOLLOW(S) ⊆ FOLLOW(B)

FOLLOW(B) = {$}

3. A is followed by B:

- FIRST(B) = {b}

FOLLOW(A) = {b}

Purpose and Importance of FOLLOW Sets

1. Handling ε-Productions

FOLLOW sets are essential when a non-terminal can derive **ε**, because they tell the parser **what symbols may appear next** after skipping that non-terminal.

2. Determining When to Apply ε-Productions

In LL (1) parsing:

- If the current input symbol is in **FOLLOW(A)** and

- **A → ε** exists
  then the parser can safely apply the ε-production.

3. Construction of Predictive (LL (1)) Parsing Tables

FOLLOW sets are used:

- Along with FIRST sets
- To fill parse table entries for ε-productions
- To ensure **unambiguous, deterministic parsing**

4. Ensuring Correct Parsing Termination

FOLLOW sets help the parser identify:

- When a non-terminal’s derivation is complete
- When to return control to higher-level grammar symbols

Relationship Between FIRST and FOLLOW Sets

FIRST Set	**FOLLOW Set**
Describes what can appear **at the beginning** of a derivation	Describes what can appear **after** a non-terminal
Used to choose productions	Used to decide when ε is applied
Based on RHS symbols	Based on surrounding context

Question 2: C++ Programming

Write a C++ Program to Count the Number of Digits in an Input String

Explanation

- The program reads a string from the user
- It checks each character
- If the character is between '0' and '9', it is counted as a digit.

```cpp
#include <iostream>
#include <string>
using namespace std;
int main() {
    string input;
    int count = 0;
    cout << "Enter a string: ";
    getline(cin, input);

    for (char c : input) {
        if (c >= '0' && c <= '9') {
            count++;
        }
    }
    cout << "Number of digits in the string: " << count << endl;
    return 0;
}
```

Sample Input

Hello123World45

Sample Output

Number of digits in the string: 5

Question 3: Problem Solving – Parse Tree Construction

Given Grammar

S → aSb | ε

Input String

aaabbb

Understanding Grammar

The grammar consists of:

- One **non-terminal**: S
- Two **terminals**: a and b
- One **recursive production**: S → aSb
- One **base case**: S → ε

This grammar generates strings that:

- Contain an **equal number of as and bs**
- Have all as **before** all bs
- Are **symmetrical**, meaning every a added at the beginning has a matching b at the end

How the Grammar Generates aaabbb

Each application of the rule S → aSb:

- Adds one a to the **left**
- Adds one b to the **right**
- Keeps S in the middle to allow further expansion

The recursion continues until the required number of as and bs is produced.
Finally, the rule S → ε is applied to **stop the recursion**.

Step-by-Step Derivation

Starting from the start symbol S:

S

→ aSb

→ aaSbb

→ aaaSbbb

→ aaaεbbb

→ aaabbb

This derivation shows:

- Three applications of S → aSb produce three as and three bs
- The ε-production terminates the recursive expansion

Steps to Sketch the Parse Tree

Follow these steps carefully when drawing the parse tree:

Step 1: Draw the Root

- Start with the start symbol S at the top of the tree

Step 2: Apply the Production S → aSb

- Draw three branches from S:
    - Left child: a
    - Middle child: S
    - Right child: b

Step 3: Expand the Middle S

- Apply S → aSb again
- Repeat the same branching structure (a, S, b)

Step 4: Continue Until the Required Length Is Reached

- Apply S → aSb **three times** to match the three as and bs in aaabbb

Step 5: Apply the Base Case

- Replace the deepest S with ε
- This indicates the end of recursion

Step 6: Verify the Leaves

- Read the leaf nodes from **left to right**
- The result should be:

aaabbb

If it matches the input string, the parse tree is correct.