Aisha Siddiqa - 22i-1281
Ayesha Ejaz - 22i-0899



# A brief report

## 1. Data Structures and Data Types Used

### 1.1 Terminals Storage - `vector<char>`

Data Type: `vector<char>`
Purpose: The vector `terminalSSS` stores all unique terminal symbols extracted from the grammar.
Usage: The function `extract_terminals()` identifies terminal symbols and adds them to

terminalSSS after checking for duplicates.

### 1.2 Non-Terminals Representation - class Non_Terminal

Data Type: class Non_Terminal
Purpose: This class represents a non-terminal symbol and its productions, along with the First and Follow sets.
Key Attributes:

- char name: Represents the non-terminal.
- *vector<string> Production:*
  - Stores the productions for the non-terminal.
- *set<char> First, Follow:*
  - Stores the computed First and Follow sets.
- map<char, string> ParsingTable: Maps terminals to their respective productions for LL(1) parsing.
- Non_Terminal* next_symbol: Forms a linked list of non-terminals.

Linked List Implementation:

- Non_Terminal* head: Static pointer to the first non-terminal.
- The linked list structure allows easy traversal and modification of non-terminals.

### 1.3 Parsing Table - map<char, string>

Data Type: map<char, string>

Purpose: The parsing table for each non-terminal stores terminal symbols as keys and their corresponding productions as values.

Example: A -> αB
iIn parsing table of A : ParsingTable['a'] = "aB"

This means that if terminal 'a' is encountered in the input, production "A -> αB" should be used.

## 2. Left Factoring

- Detects common prefixes among productions of a non-terminal.
- Extracts the common prefix and introduces a new non-terminal to represent the differing suffixes.
- Recursively applies left factoring until no common prefixes remain.

Example:
A → ab | ac
becomes:
A → aA'
A' → b | c

**Implementation**

- Function: `left_factored()`
- Steps:

  For each non_terminal it:

  - Iterate over all productions of a non-terminal.
  - Identify ALL common prefixes. ( for each production: count[p_size] )
  - Take the minimum common prefix and make a new non_terminal for that.
  - Update the original non-terminal to use the new non-terminal.
  - Keep doing it until sum(count) != 0, matlab no more common prefixes.

## 3. Left Recursion

Left recursion is removed to make the grammar LL(1)-compatible.

Types: Direct & Indirect

Direct Left Recursion:
A → Aα | β

A → βA' A' → αA' | ^
A → Bα   B → Aγ

```
Direct Recursion Removed CFG:
A  -> B d  | g
B  -> C f  | v
C  -> A v  | x

Applying Indirect Recursion:
A  -> C f d  | v d  | g
B  -> C f  | v
C  -> v d v a  | g v a  | x a
a  -> f d v a  | ^
```

Implementation

- Function: `direct_recursion()`
- Steps:

1. Identify productions where the left-hand side appears at the start.
2. Separate α (recursive part) and β (non-recursive part).
   - If alpha beta has issues : mam said nahi honge, but if yes then no recursion.
3. Introduce a new non-terminal A'.
4. Update the productions accordingly.

- Function: `indirect_recursion()`
- Steps:

1. FIRST Identify indirect recursion using depth-first search.
2. If found then apply *substitution* and then you'll end up with direct recursion, then call the `direct_recursion()`
3. If not found then no substitution.

# 4. First and Follow Sets Computation

The **First** and **Follow** sets determine which terminals can appear in the derivation of a non-terminal.

## First Set Computation

- **Function:** `helper_first()`
- **Steps:**

1. If the production starts with a terminal, add it to First.
2. If it starts with a non-terminal, add its First set.
3. If the non-terminal has "^" in First, continue checking the next symbol.

## Follow Set Computation

- **Function:** `helper_follow()`
- **Steps:**
  1. Start symbol always contains Follow(A) = {'$'}.
  2. If a non-terminal appears before another symbol in a production, copy the First set of the next symbol (excluding "^" ).
  3. If a non-terminal appears at the end, copy the Follow set of the current non-terminal.

```
First(A) = { (, id, num }
First(B) = { (, id, num }
First(C) = { (, id, num }
First(E) = { (, id, num }
First(K) = { ^, c }
First(L) = { d, e }
First(M) = { <, <= }
First(S) = { (, id, if, num }
First(a) = { ^, d, e }
First(b) = { *, ^ }
First(f) = { <, <=, ^ }
=============================================
Follow(A) = {  }
Follow(B) = { $, c, d, e, else }
Follow(C) = { $, ), *, <, <=, c, d, e, else, then }
Follow(E) = { ), then }
Follow(K) = { $, else }
Follow(M) = {  }
Follow(S) = { $, else }
Follow(a) = { $, c, else }
Follow(b) = { $, c, d, e, else }
Follow(f) = { ), then }
=============================================
```

# 5. LL(1) Parsing Table Construction

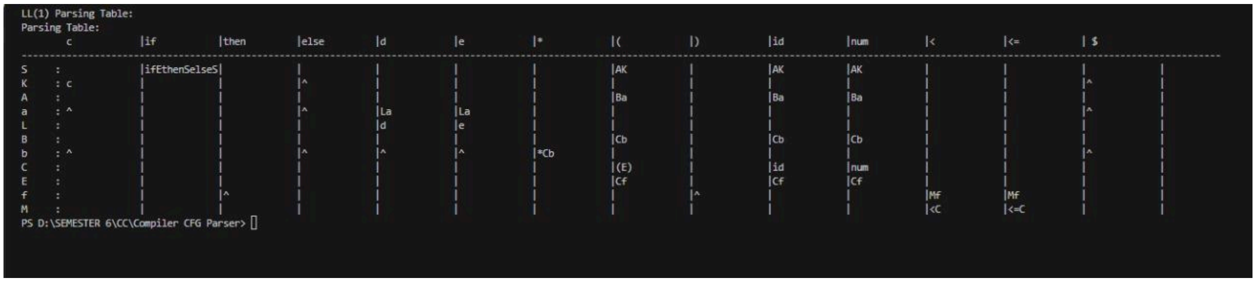
Approach:

- Function: `constructParsingTable()`
- Steps:

    1. Iterate over all productions of a non-terminal.
    2. For each production, determine its First set:
        - If it starts with a terminal, add it to the table.
        - If it starts with a non-terminal, add the First set of that non-terminal.
    3. If a production contains “^”, add `Follow(non-terminal)` to the table.

Handling LL(1) Conflicts:

*If a terminal already exists in the table for a non-terminal, the grammar* ***is not LL(1)***.



LL(1) Parsing Table:
Parsing Table:

| | c | if | then | else | d | e | * | ( | ) | id | num | < | <= | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| S : | | ifEthenSelseS | | | | | | AK | | AK | AK | | | |
| K : | c | | | ^ | | | | | | | | | | ^ |
| A : | | | | | | | | Ba | | Ba | Ba | | | |
| a : | ^ | | | ^ | La | La | | | | | | | | ^ |
| L : | | | | | d | e | | | | | | | | |
| B : | | | | | | | | Cb | | Cb | Cb | | | |
| b : | ^ | | | ^ | ^ | ^ | *Cb | | | | | | | ^ |
| C : | | | | | | | | (E) | | id | num | | | |
| E : | | | | | | | | Cf | | Cf | Cf | | | |
| f : | | | ^ | | | | | | ^ | | | Mf | Mf | |
| M : | | | | | | | | | | | | <C | <=C | |

PS D:\SEMESTER 6\CC\Compiler CFG Parser>

Process:

1. Initialize the table:
   - Each non-terminal gets an empty row.
   - Each terminal (including `$`) forms a column.
2. Populate table using First and Follow sets:
   - For each production, determine applicable terminal entries.
   - If "^" is in **First(non-terminal)**, add **Follow(non-terminal)**.

3. Handle conflicts:
   - If multiple productions exist for the same `(Non-Terminal, Terminal)` pair, the grammar is **not LL(1)**.

# challenges faced

## *Input Formatting*

Initially, we designed our code assuming that terminals would be lowercase characters, non-terminals would be uppercase characters, and there would be no spaces in the input. Since *we started early*, we completed most of the implementation before verifying the input format with Maám. Later, we learned that the input format required strings for terminals and non-terminals, with all values being space-separated. This required us to modify our approach significantly. Instead of rewriting the core CFG logic, which was functioning correctly, we decided to ***create an adaptor that maps input strings to unique characters using hashmaps***. This ensured our existing code remained unchanged while allowing it to handle the new input format. Implementing and debugging this adaptor *took 4-5 hours of additional effort*, but it ultimately allowed our code to work with any input format efficiently.

- The extra implementation is added in a separate file `"test.h"`

# how you verified the correctness of your program.

To verify the correctness of our program,

- ➢ We thoroughly tested it *at each step* by checking the output against expected results.
- ➢ We created our *own example inputs* and tested them, ensuring they produced the correct outputs.
- ➢ Additionally, my teammate searched for *examples from books*, and we tested those as well.
- ➢ We also found *examples online* for each step and verified that our program handled them correctly.
- ➢ Furthermore, we *asked our classmates* to provide test cases, adding more variety to our validation process.

All test results and outputs are saved in the file `"output.txt"` for reference

```
Applying Indirect Recursion:
S  -> A  | a
A  -> C f c  | x d c  | v f
B  -> C f  | x d
C  -> x d c s b  | v f s b  | x B c b
b  -> f c s b  | ^

Calculating First Sets:

First Sets:
First(S ) = {a ,v ,x }
First(A ) = {v ,x }
First(B ) = {v ,x }
First(C ) = {v ,x }
First(b ) = {^ ,f }

Calculating Follow Sets:

Follow Sets:
Follow(S ) = {$ }
Follow(A ) = {$ }
Follow(B ) = {c }
Follow(C ) = {f }
Follow(b ) = {f }

Constructing Parsing Table:
Grammar is not LL(1)! Conflict found at A for terminal x
Grammar is not LL(1)! Conflict found at A for terminal v
Grammar is not LL(1)! Conflict found at B for terminal x
Grammar is not LL(1)! Conflict found at C for terminal x
Grammar is not LL(1)! Conflict found at b for terminal f
```