Final Project

Matías Martínez Moreno and Samuel Orozco

Universidad Eafit, Medellín

Formal languages

Adolfo Andrés Castro Sánchez

Medellín

May 2025

Introduction

This project analyze if a text string follows the rules of a grammar (syntactic analysis or parsing).

First, we calculate the First and Follow sets for the grammar.

Then, we use these sets to build two types of parsers:

LL(1): A Top-Down parser.

SLR(1): A Bottom-Up parser.

The final goal is to have a tool that outputs "yes" or "no" indicating whether an input string belongs to the language defined by the grammar.

Tools used:

Python (versión 3.13.3)

Windows 11

Visual Studio Code

“Compilers: Principles, Techniques, and Tools“ book

Important Theory we read before implementation

Syntactic analysis is the compiler phase that checks if the input token sequence conforms to the structure defined by a CFG. Its main function is to validate syntax and build a parse tree. The most common parsing methods are Top-Down, suitable for LL grammars (like LL(1)), and Bottom-Up, suitable for LR grammars (like SLR(1)).

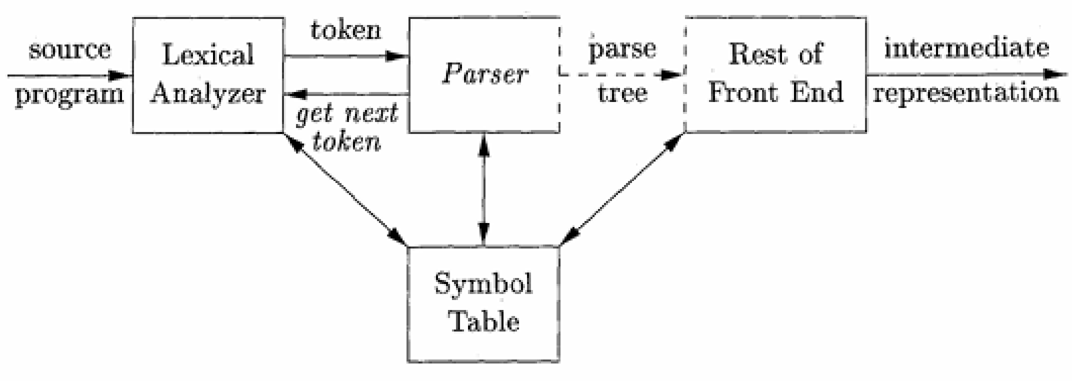
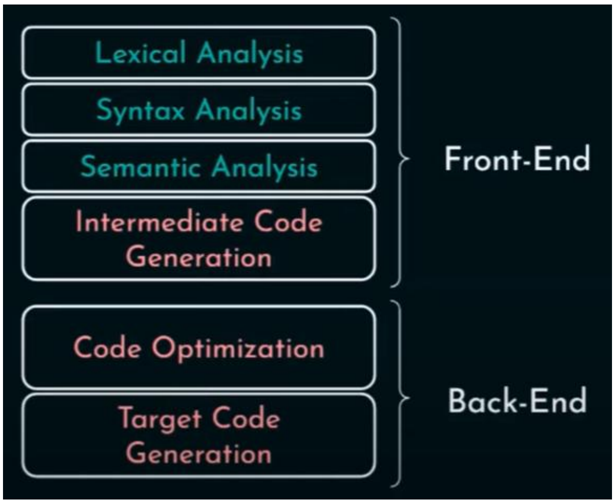


Figure 4.1: Position of parser in compiler model



Context-Free Grammars (CFG)

CFGs are the standard way to formally define syntax. They have 4 parts: terminals (basic tokens), nonterminals (variables representing sets of strings), a start symbol (where derivations begin), and productions (rules A -> α defining how to form strings).

Derivations

It is the formal process of generating strings from a CFG by applying productions starting from the start symbol.

Leftmost Derivation (⇒lm): Always replaces the leftmost nonterminal. Related to Top-Down (LL) parsing.

Rightmost Derivation (⇒rm): Always replaces the rightmost nonterminal. Related (in reverse) to Bottom-Up (SLR) parsing.

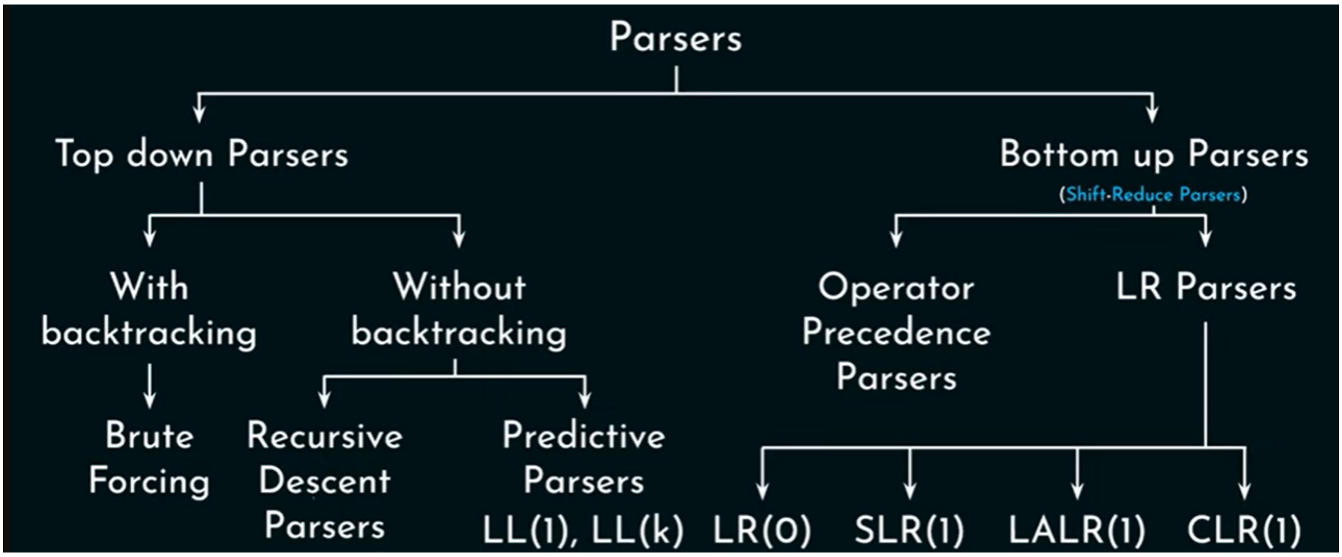
Parse Trees and Ambiguity

A parse tree graphically represents a derivation, serving as the implicit goal of parsing algorithms. If a grammar generates more than one tree for the same sentence, it **is ambiguous**. Also grammars sometimes need transformations, like:

Eliminate left recursion: Rules like A -> Aα | β are changed to A -> βA', A' -> αA' | ε.

Apply left factoring: Rules like A -> αβ1 | αβ2 are changed to A -> αA', A' -> β1 | β2.

This allows deciding which rule to use by looking at the next input symbol (lookahead).



Top-Down Parsing

tries to build the parse tree from the root (start symbol) down to the leaves (tokens), seeking a leftmost derivation. The challenge is choosing the correct production to expand a nonterminal. Predictive parsing (LL(1)) does this efficiently without backtracking, using one lookahead symbol to decide. It works for LL(1) grammars and requires calculating *FIRST and FOLLOW sets* to build a predictive parsing table.

FIRST and FOLLOW sets

FIRST(α): The set of terminals that can start strings derived from α. Includes ε if α can derive the empty string. Calculated iteratively based on productions.

FOLLOW(A): The set of terminals that can appear immediately after the nonterminal A in some derivation. Includes $ if A can be the last symbol. Calculated iteratively using productions and FIRST sets.

LL(1) Parsing Table

The predictive parsing table M[A, a] is built using FIRST and FOLLOW sets. M[A, a] tells the parser which production to use when A is on the stack top and a is the lookahead. If any cell M[A, a] contains more than one production, the grammar is not LL(1).

Bottom-Up Parsing (SLR(1))

Bottom-Up Parsing builds the parse tree from the leaves (tokens) up to the root (start symbol), equivalent to performing a rightmost derivation in reverse. Its key operation is reduction: finding a substring matching a production body (A -> α) and replacing it with the head (A). This repeats until the start symbol is reached. The most common style is shift-reduce parsing, efficient for LR grammars (like SLR(1))

Shift-Reduce and Handles

Shift-reduce parsing uses a stack and the input buffer. Its actions are: shift input symbols onto the stack, or reduce.

LR Parsing and SLR(1):

LR(k) parsing is a general shift-reduce technique (L: Left scan, R: Rightmost derivation reverse, k: lookahead). LR grammars are a broader class than LL, covering most programming languages. LR parsers are efficient and detect errors early but are complex.SLR(1) is the simplest LR(1) variant

LR parsers work using a stack and ACTION/GOTO tables. The parser looks at the state on top of the stack and the next input symbol (lookahead).

It checks

SHIFT: Push the lookahead and a new state onto the stack, then read the next input.

REDUCE: Pop symbols/states corresponding to a grammar rule (A -> β) off the stack. Then, use the GOTO table with the state now on top and the rule's left side (A) to find the next state to push.

ACCEPT-ERROR

To build the SLR(1) tables:

First, create the LR(0) states (sets of items like [A -> α . β]).

SHIFT actions go into ACTION[state, terminal] based on GOTO transitions between states on terminals.

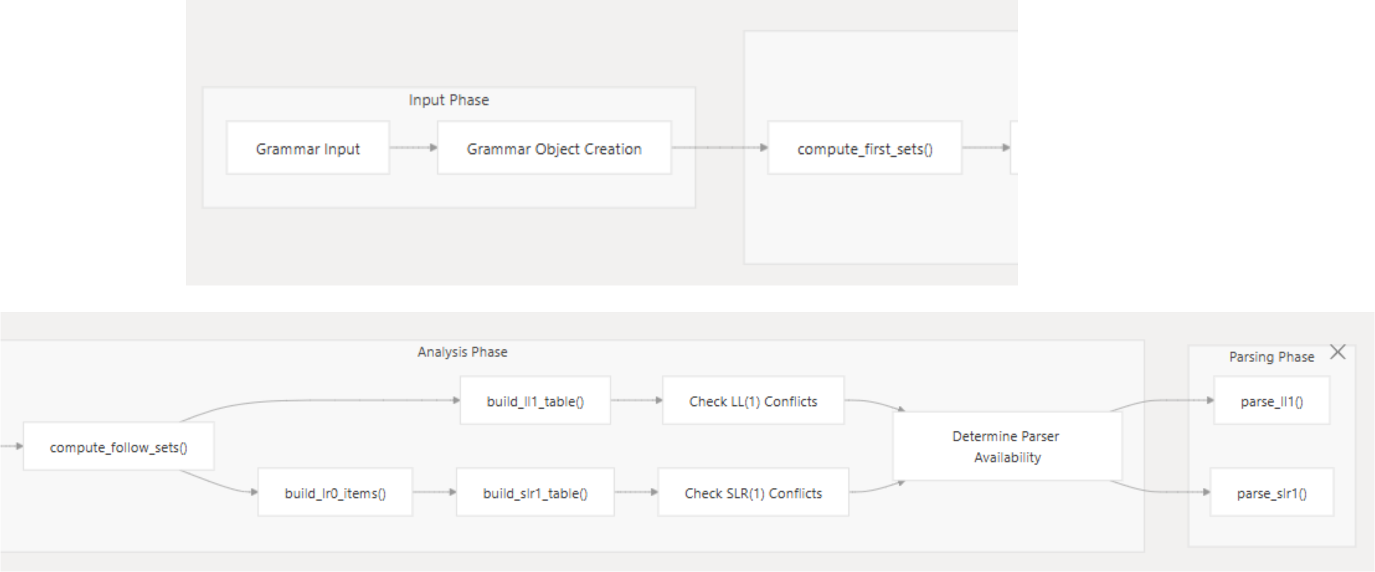
REDUCE actions for A -> α go into ACTION[state, terminal] if the state contains the completed item [A -> α .] and the terminal is in FOLLOW(A).

The GOTO table stores the state transitions for nonterminals.

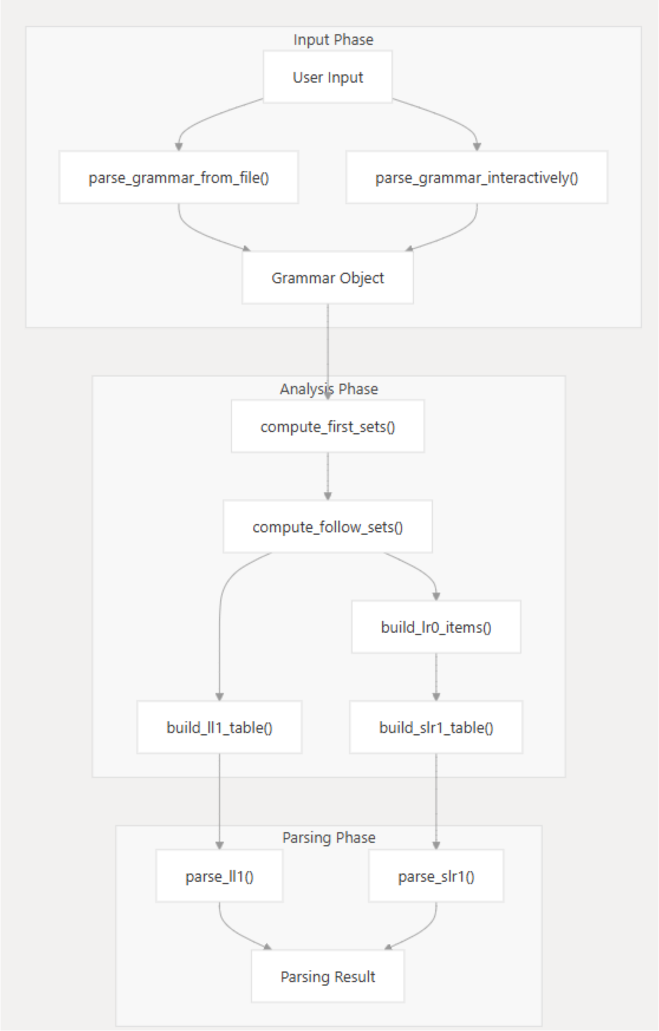
If any ACTION table cell ends up with more than one action (a conflict), the grammar is not SLR(1).

Our implementation

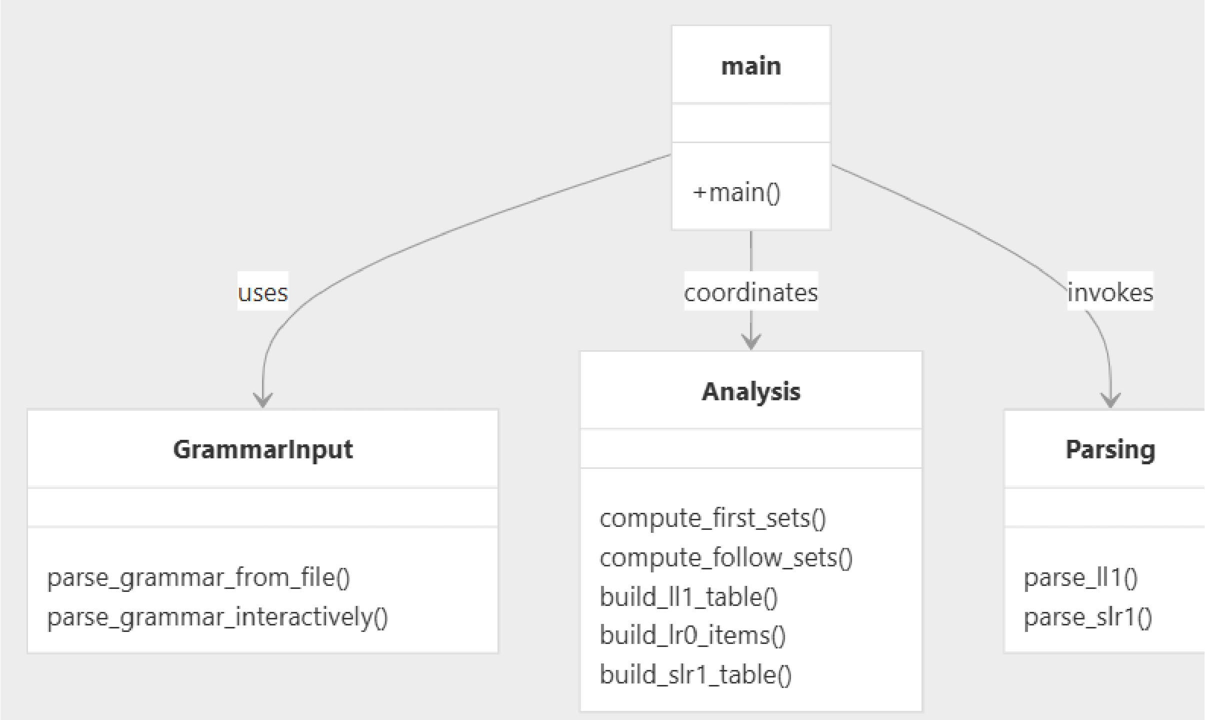
Processing pipeline



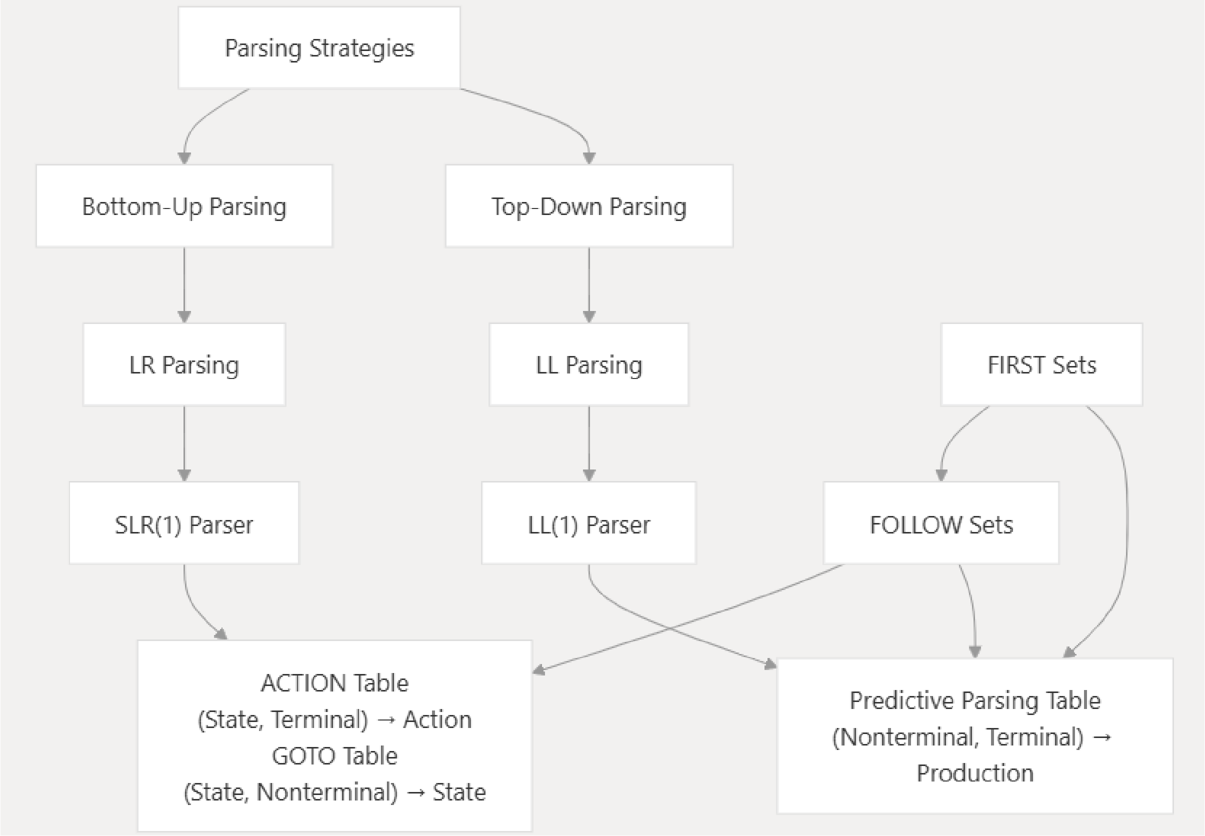
System integration



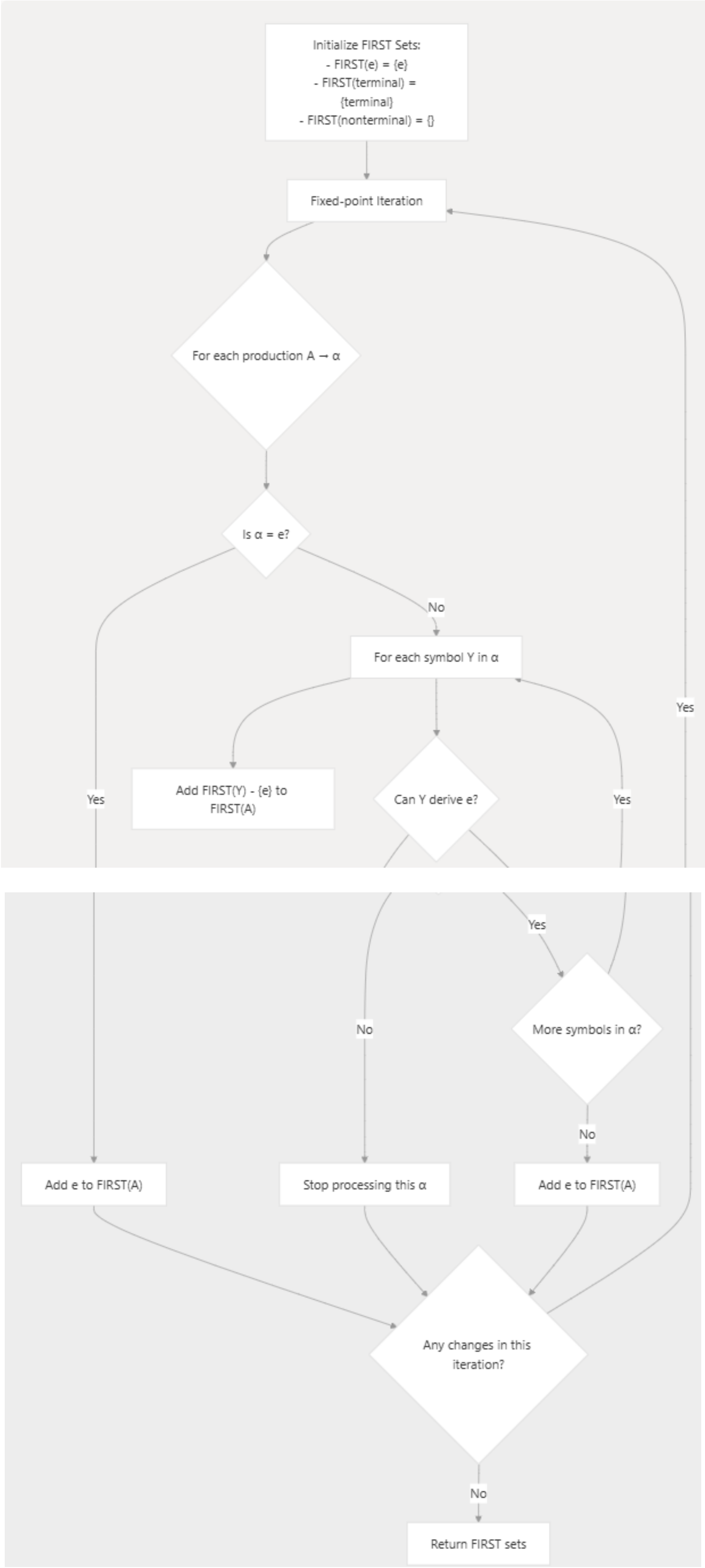
System overview



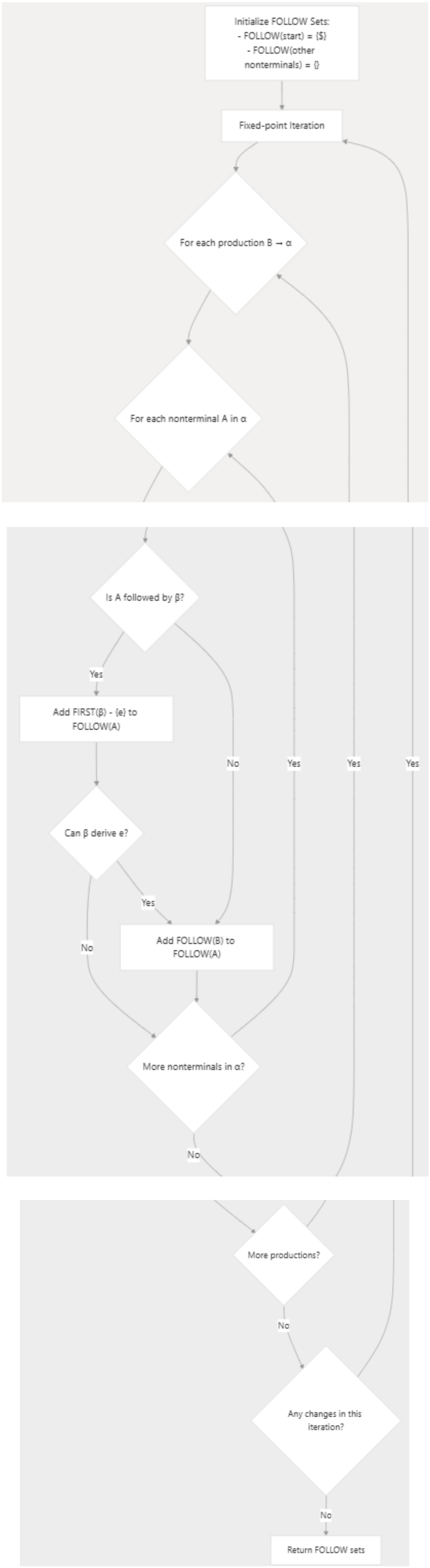
Parsing



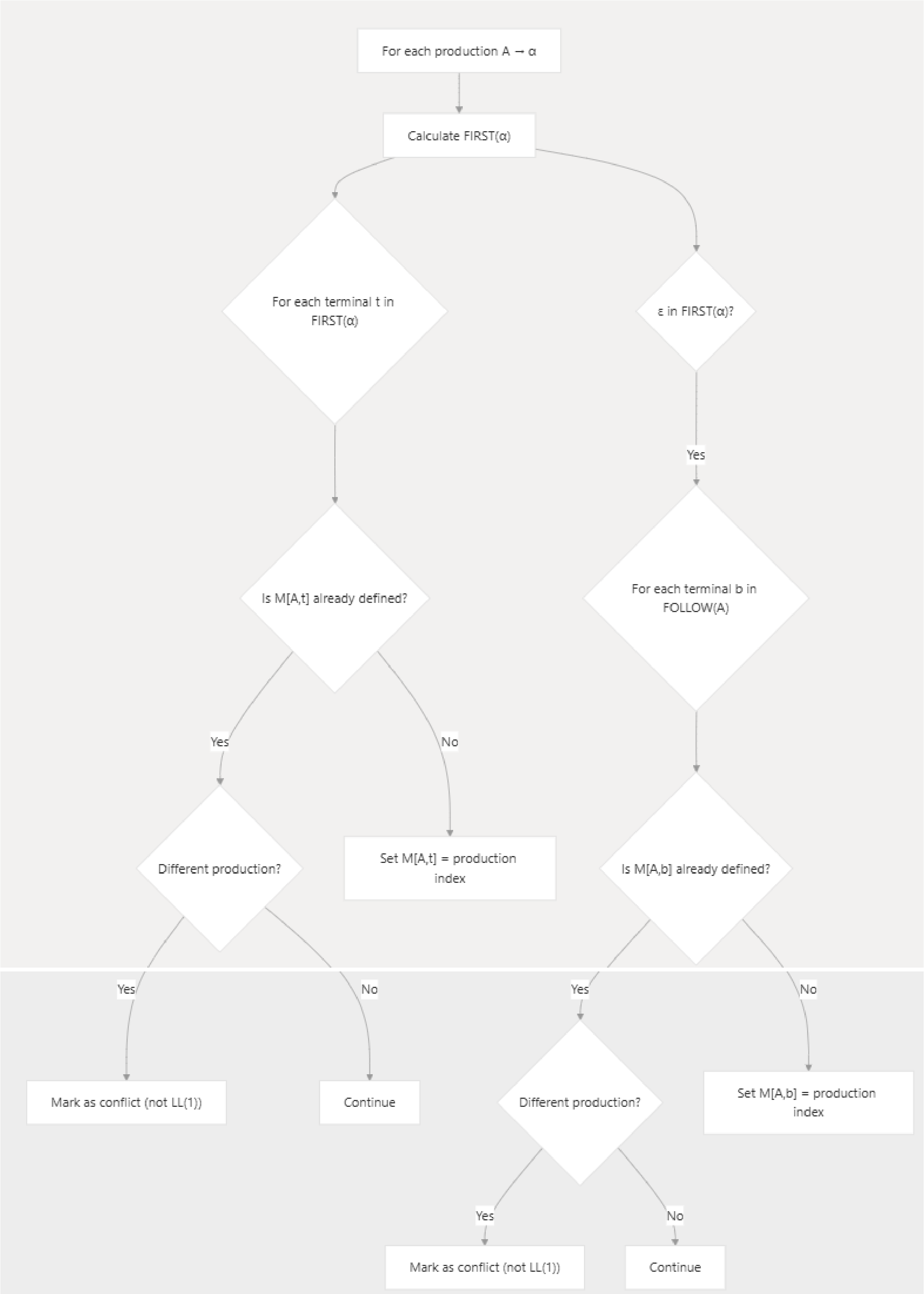
FIRST Sets Computation Algorithm



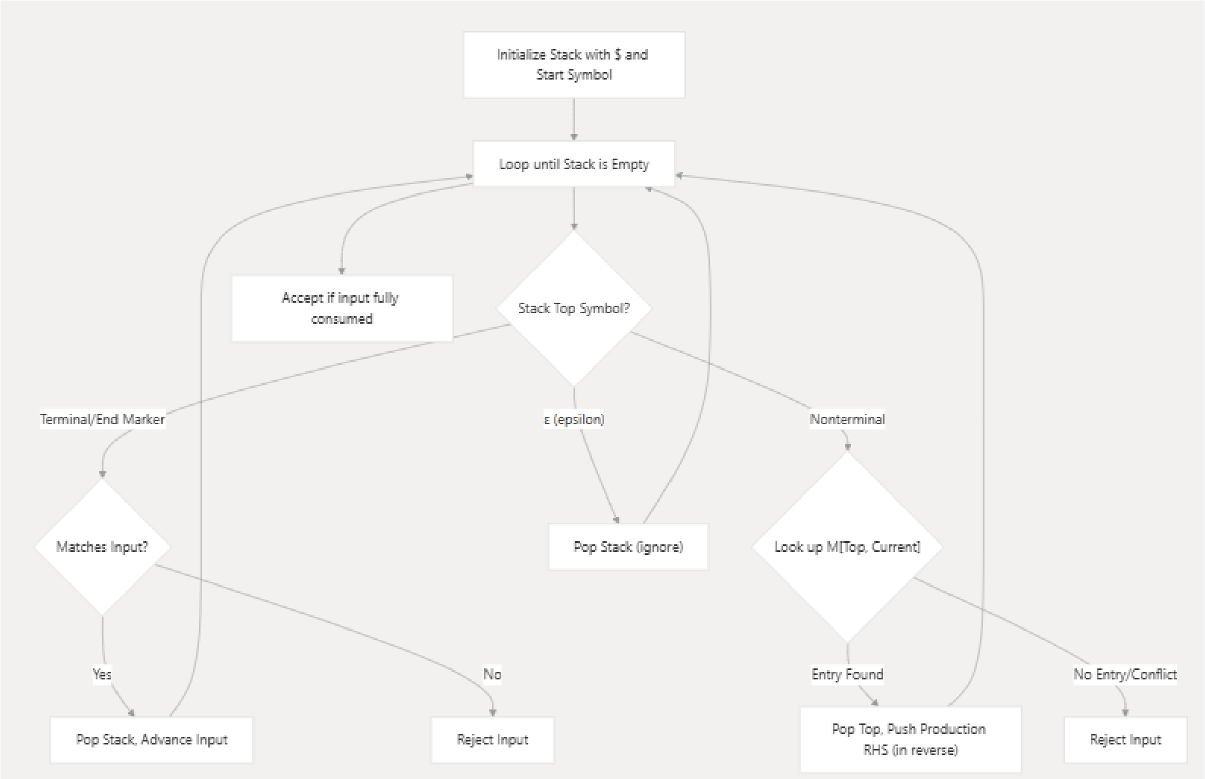
FOLLOW Sets Computation Algorithm



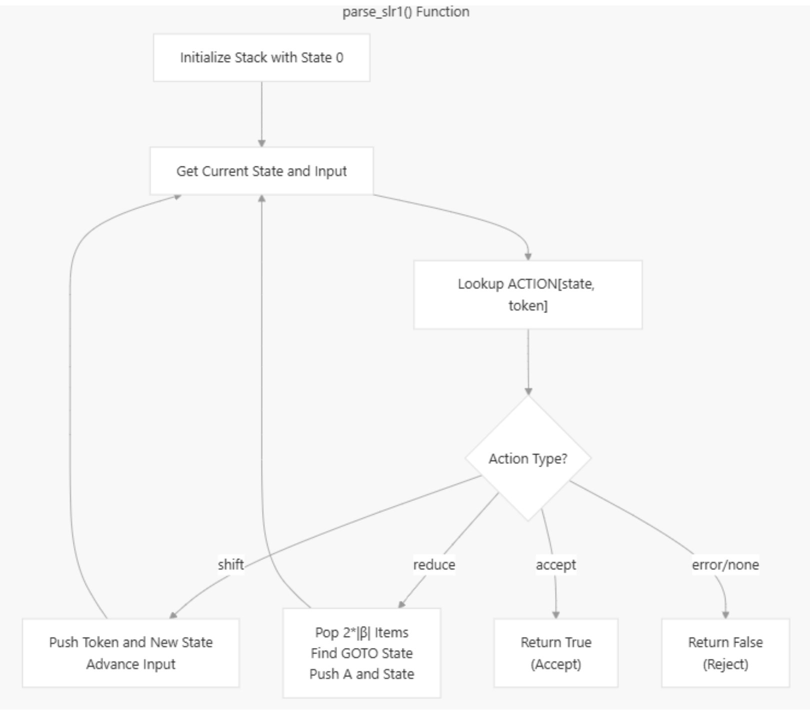
Conflict detection ll1


For each production A → α
Calculate FIRST(α)
For each terminal t in FIRST(α)
ε in FIRST(α)?
Yes
Is M[A,t] already defined?
For each terminal b in FOLLOW(A)
Yes
No
Different production?
Set M[A,t] = production index
Is M[A,b] already defined?
Yes
No
Yes
No
Mark as conflict (not LL(1))
Continue
Different production?
Set M[A,b] = production index
Yes
No
Mark as conflict (not LL(1))
Continue

LL(1) parsing



SLR(1)


parse_slr1() Function
Initialize Stack with State 0
Get Current State and Input
Lookup ACTION[state, token]
Action Type?
shift
reduce
accept
error/none
Push Token and New State Advance Input
Pop 2*|β| Items Find GOTO State Push A and State
Return True (Accept)
Return False (Reject)

Problems found and solutions

First and Follow

-Initially, it was hard to see how First and Follow were calculated if one seemed to depend on the other. The key was understanding they are computed in rounds, repeating until they no longer change.

-Ensuring the code calculating First and Follow repeated the necessary number of times without entering infinite loops.

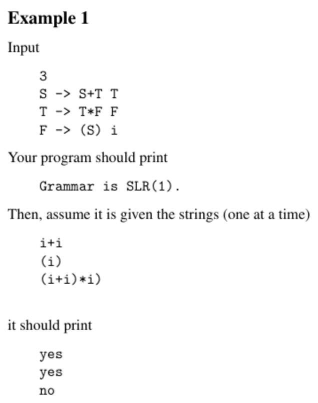
Reading the Grammar: Getting the program to correctly read the grammar from the user or a file, ignoring extra spaces but detecting format errors. strip() was used, and the expected format was checked.

handling Épsilon ('e'): Correctly using 'e' when calculating First, deciding whether to use Follow, and in the parsing tables was hard. It was treated as a special case in the calculation functions and when building the tables. Also $ was included incorrectly in first sets buy it was solved doing it explicit in a condition that excluded it.

Why Follow in SLR(1): It wasn't obvious why SLR(1) uses Follow sets. It became clear upon realizing that Follow limits when a reductiion can occur, restricting it only to cases where the next input symbol is valid according to Follow, thus preventing errors.

Detecting SLR(1) Conflicts: Knowing when a conflict (shift/reduce or reduce/reduce) occurred while filling the ACTION table. This was solved by checking if the table cell already held a different action before attempting to write a new one.

Testing



Example 1

Input

```
3
S -> S+T T
T -> T*F F
F -> (S) i
```

Your program should print

```
Grammar is SLR(1).
```

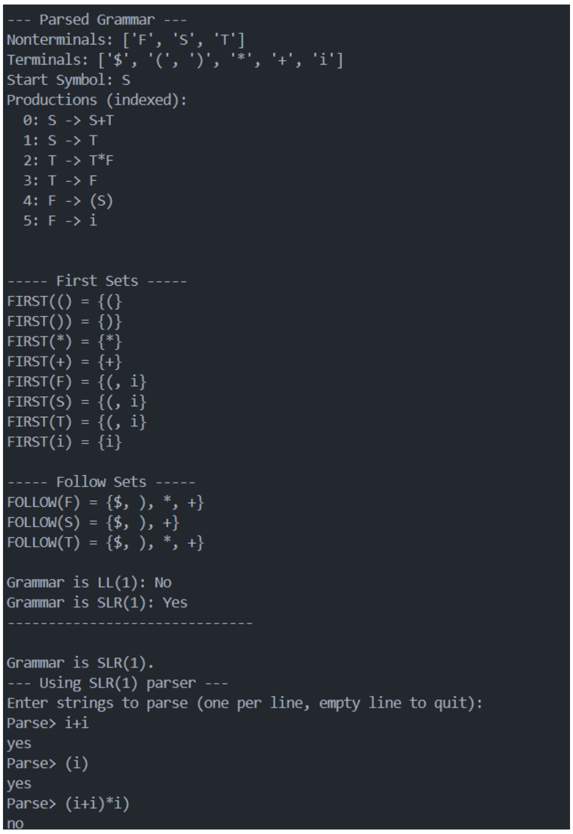
Then, assume it is given the strings (one at a time)

```
i+i
(i)
(i+i)*i)
```

it should print

```
yes
yes
no
```

Then



```
--- Parsed Grammar ---
Nonterminals: ['F', 'S', 'T']
Terminals: ['$', '(', ')', '*', '+', 'i']
Start Symbol: S
Productions (indexed):
  0: S -> S+T
  1: S -> T
  2: T -> T*F
  3: T -> F
  4: F -> (S)
  5: F -> i


----- First Sets -----
FIRST(() = {(}
FIRST()) = {)}
FIRST(*) = {*}
FIRST(+) = {+}
FIRST(F) = {(, i}
FIRST(S) = {(, i}
FIRST(T) = {(, i}
FIRST(i) = {i}

----- Follow Sets -----
FOLLOW(F) = {$, ), *, +}
FOLLOW(S) = {$, ), +}
FOLLOW(T) = {$, ), *, +}

Grammar is LL(1): No
Grammar is SLR(1): Yes
------------------------------

Grammar is SLR(1).
--- Using SLR(1) parser ---
Enter strings to parse (one per line, empty line to quit):
Parse> i+i
yes
Parse> (i)
yes
Parse> (i+i)*i)
no
```

Example 2

Input

```
3
S -> AB
A -> aA d
B -> bBc e
```

Your program should print

```
Select a parser (T: for LL(1), B: for SLR(1), Q: quit):
```

Assume `T` is given. Then, assume it is given the strings (one at a time)

```
d
adbc
a
```

it should print

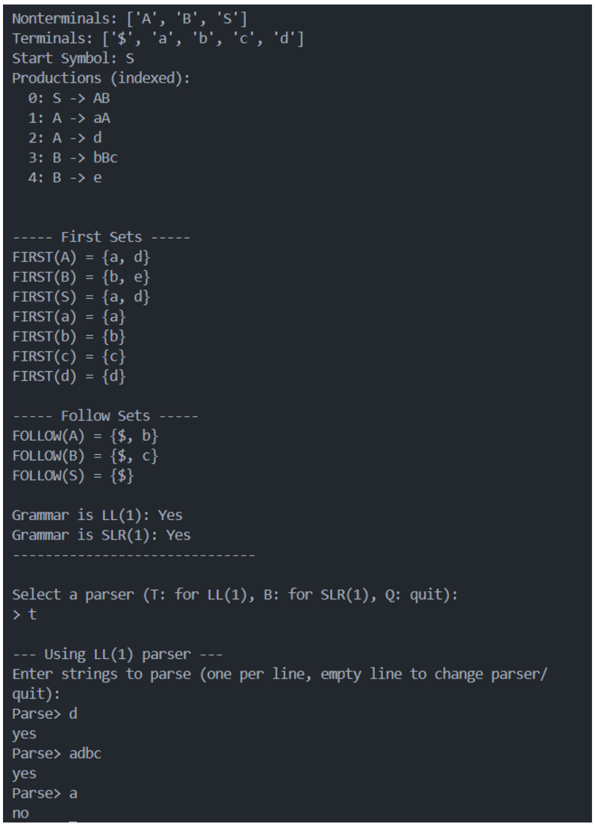
```
yes
yes
no
```

Then

```
Select a parser (T: for LL(1), B: for SLR(1), Q: quit):
```

Assume `Q` is given.

Then



```
Nonterminals: ['A', 'B', 'S']
Terminals: ['$', 'a', 'b', 'c', 'd']
Start Symbol: S
Productions (indexed):
  0: S -> AB
  1: A -> aA
  2: A -> d
  3: B -> bBc
  4: B -> e


----- First Sets -----
FIRST(A) = {a, d}
FIRST(B) = {b, e}
FIRST(S) = {a, d}
FIRST(a) = {a}
FIRST(b) = {b}
FIRST(c) = {c}
FIRST(d) = {d}

----- Follow Sets -----
FOLLOW(A) = {$, b}
FOLLOW(B) = {$, c}
FOLLOW(S) = {$}

Grammar is LL(1): Yes
Grammar is SLR(1): Yes
------------------------------

Select a parser (T: for LL(1), B: for SLR(1), Q: quit):
> t

--- Using LL(1) parser ---
Enter strings to parse (one per line, empty line to change parser/
quit):
Parse> d
yes
Parse> adbc
yes
Parse> a
no
```

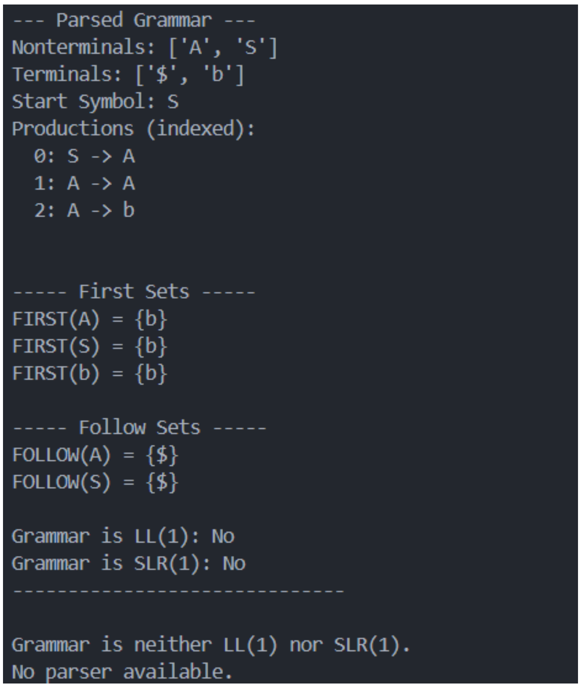
Example 3

Input

```
2
S -> A
A -> A b
```

Your program should print

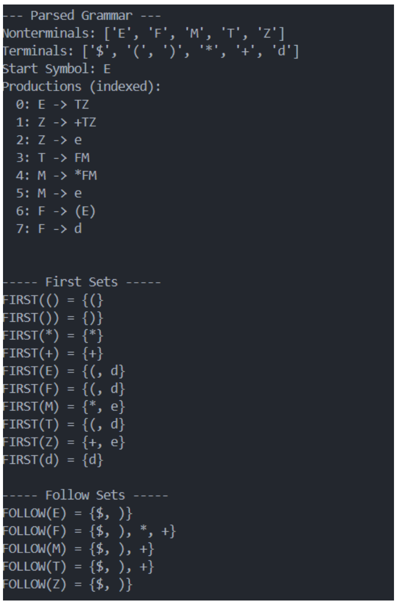
```
Grammar is neither LL(1) nor SLR(1).
```



```
--- Parsed Grammar ---
Nonterminals: ['A', 'S']
Terminals: ['$', 'b']
Start Symbol: S
Productions (indexed):
  0: S -> A
  1: A -> A
  2: A -> b


----- First Sets -----
FIRST(A) = {b}
FIRST(S) = {b}
FIRST(b) = {b}

----- Follow Sets -----
FOLLOW(A) = {$}
FOLLOW(S) = {$}

Grammar is LL(1): No
Grammar is SLR(1): No
------------------------------

Grammar is neither LL(1) nor SLR(1).
No parser available.
```

Additionaly, using an example of the book:

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \ \mid\ \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \ \mid\ \epsilon \\ F & \rightarrow & (\ E\ )\ \mid\ \mathbf{id} \end{array} \tag{4.28}$$



```
--- Parsed Grammar ---
Nonterminals: ['E', 'F', 'M', 'T', 'Z']
Terminals: ['$', '(', ')', '*', '+', 'd']
Start Symbol: E
Productions (indexed):
  0: E -> TZ
  1: Z -> +TZ
  2: Z -> e
  3: T -> FM
  4: M -> *FM
  5: M -> e
  6: F -> (E)
  7: F -> d


----- First Sets -----
FIRST(() = {(}
FIRST()) = {)}
FIRST(*) = {*}
FIRST(+) = {+}
FIRST(E) = {(, d}
FIRST(F) = {(, d}
FIRST(M) = {*, e}
FIRST(T) = {(, d}
FIRST(Z) = {+, e}
FIRST(d) = {d}

----- Follow Sets -----
FOLLOW(E) = {$, )}
FOLLOW(F) = {$, ), *, +}
FOLLOW(M) = {$, ), +}
FOLLOW(T) = {$, ), +}
FOLLOW(Z) = {$, )}
```

Conclusions:

Algorithms to compute First and Follow sets, and to build and operate LL(1) (top-down) and SLR(1) (bottom-up) parsers were successfully implemented

References:

Aho, Alfred V. et al. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley, 2006.