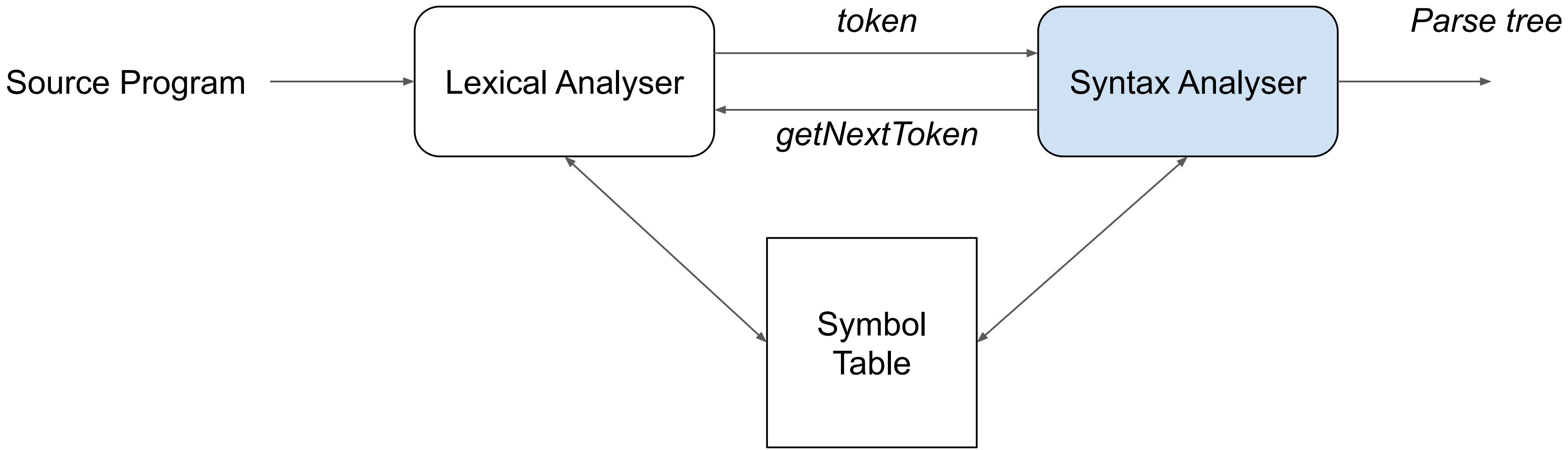
Syntax Analysis

Md Shad Akhtar
Assistant Professor
IIIT Dharwad

Syntax Analyser (Parser)

- Define the syntactic structure for a programming language
- Reads the sequence of tokens from lexical analysis and create|validate the syntactic structure (parse tree) for the sequence of tokens.



Syntactic structure and grammar

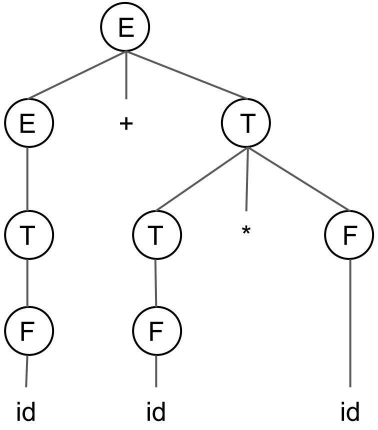
- Syntactic structure is defined by the context-free grammar (CFG)
- Steps to create parse tree
  - Parser checks whether a given source program satisfies the rules implied by a CFG or not
  - If it satisfies, the parser creates the parse tree of that program
  - Otherwise, the parser gives the error messages

Grammar (G)

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

Token sequence

id + id * id



Syntax Errors

- Role of error handler in parser
  - Report the presence of errors clearly and accurately
  - Recover from each error quickly enough to detect subsequent errors
  - Add minimal overhead to the processing of correct programs.

- Error Detection:
  - Sequence of tokens that can not be accepted by any grammar rule.
  - E.g.:
    - A switch statement without a case statement
    - Missing closing braces
    - Operator without operands   c = a +
    - Operands without operator   c = a   b

Syntax Error Recovery

- Panic-mode:
  - On discovering an error, discards input symbols one at a time until one of a designated set of synchronizing tokens is found, e.g., semicolon, closing brace, etc.
- Phrase-level recovery:
  - Perform local correction on the remaining input to continue
    - Replace the prefix of the remaining input by some string that allows the parser to continue. E.g., Replace comma by semicolon, delete|insert an extra|missing semicolon.
- Error Production
  - For common errors, add special production rules to handle such scenario
- Global correction
  - Ideally, we want as few changes as possible to process incorrect inputs.
  - We can design an algorithm for choosing a minimal sequence of changes to obtain a globally least-cost correction.
    - Given incorrect input x and grammar G, find a correct related input y with as less changes as possible.

Types of parsers

- In general three types of parsers
  - Universal
    - Capable to parse any grammar but too complex to use in compiler
    - E.g.: Cocke-Younger-Kasami (CYK) parser, Earley’s parser
  - Top-down
    - Build parse tree from root to leaf
  - Bottom-Up
    - Build parse tree from leaf to root

Context-free Grammar (CFG)

- Provides a precise syntactic specification of a programming language
- A CFG G = <N, T, P, S>
  - **N**on-terminals:
    - A finite set of non-terminals (variables) [usually in capital letters]
  - **T**erminals:
    - A finite set of terminals (input symbols|tokens) [usually in small letters]
  - **P**roduction:
    - A finite set of productions rules in the following form $A \rightarrow \alpha$ where A is a non-terminal and α is a string of terminals and non-terminals (including the empty string); $|A| <= |\alpha|$
  - **S**tart symbol:
    - One of the non-terminal symbols

CFG: An example

- CFG G = <N, T, P, S>
  - **N**on-terminal = {E}
  - **T**erminals = {+, -, *, |, (,), id}
  - **S**tart symbol = {E}
  - **P**roduction

$$E \rightarrow E + E \mid E - E \mid E * E \mid E \mid E \mid - E$$
$$E \rightarrow (E)$$
$$E \rightarrow id$$

Derivations

- Starting with the start symbol, replace each non-terminals with the body of one of its production rules till all non-terminals are replaced by terminal symbols.
    - $E \Rightarrow E+E \Rightarrow id + E \Rightarrow id + id$
- In general a derivation step is

$$\alpha A\beta \Rightarrow \alpha\gamma\beta$$

  if there is a production rule $A \rightarrow \gamma$ in our grammar, where α and β are arbitrary strings of terminal and non-terminal symbols.
- $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n$ (α_n derives from α_1 or α_1 derives α_n)
- $\Rightarrow$ drives in one step
- $\Rightarrow^*$ drives in zero or more steps
- $\Rightarrow^+$ drives in zero or one step

Derivations

- $S \Rightarrow^* \alpha$
  - If α contains non-terminals, it is called as a sentential form of G
  - If α does not contain non-terminals, it is called as a sentence of G
- **Left-most derivation:** Always chooses the left-most non-terminal in each derivation step

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(id+E) \Rightarrow -(id+id)$$

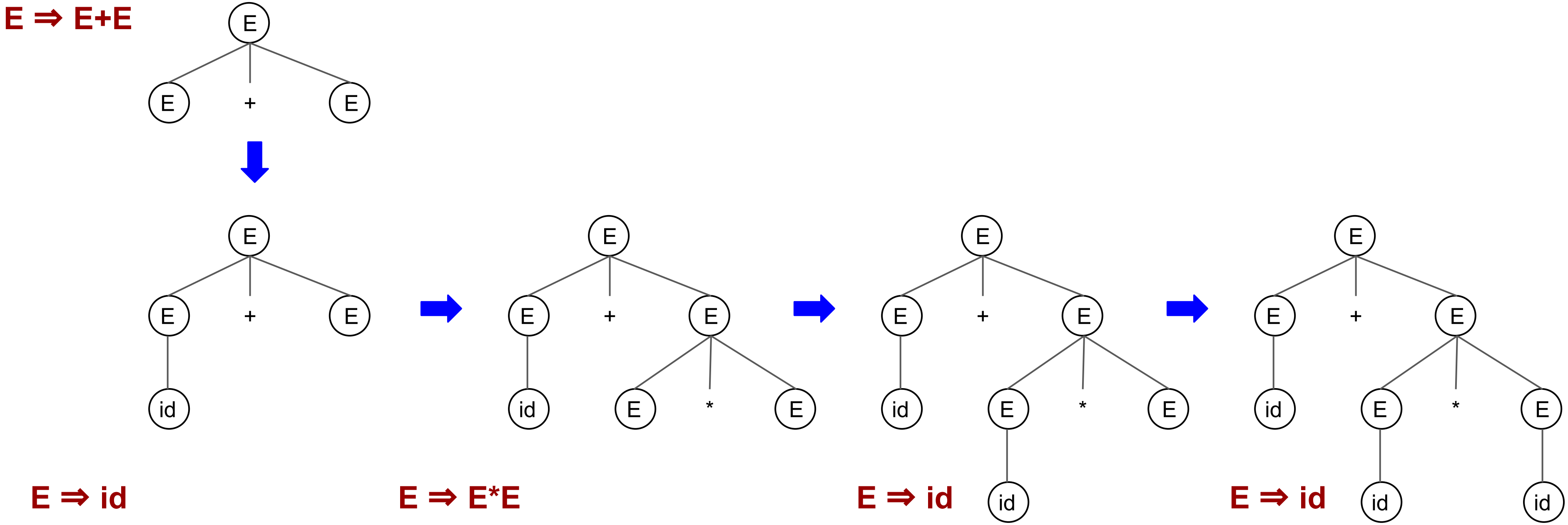
- **Right-most derivation:** Always chooses the right-most non-terminal in each derivation step

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+id) \Rightarrow -(id+id)$$

- **Top-down parsers:** Finds the left-most derivation of the given source program
- **Bottom-up parsers:** Finds the right-most derivation of the given source program in the reverse order

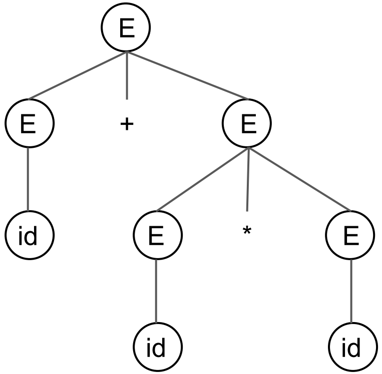
Parse Tree

- A graphical representation of a derivation
- Intermediate nodes: Inner nodes of a parse tree
- Leaves: Terminal symbols



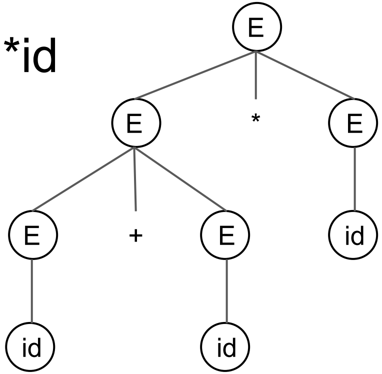
Ambiguity

- A grammar that produces more than one parse tree for a sentence is called as an ambiguous grammar



- $E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E \Rightarrow^* id+id*E \Rightarrow id+id*id$

- $E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow id+E*E \Rightarrow id+id*E \Rightarrow id+id*id$



Ambiguity and Parser

- For the most parsers, the grammar must be unambiguous.
  - unique selection of the parse tree for a sentence

- Disambiguation of an ambiguous grammar
  - Necessary to eliminate the ambiguity in the grammar during the design phase of the compiler
  - Choose one of the parse trees of a sentence to restrict to this choice

Ambiguity disambiguation

- Stmt → if Expr then Stmt | if Expr then Stmt else Stmt | other_stmts
- Input string: if E_1 then if E_2 then S_1 else S_2
- **Interpretation 1:** S_2 being executed when E_1 is false (thus attaching the else to the first if)
    - if E_1 then (if E_2 then S_1) else S_2
- **Interpretation 2:** S_2 being executed when E_1 is true and E_2 is false (thus attaching the else to the second if)
    - if E_1 then (if E_2 then S_1 else S_2)

Ambiguity disambiguation

- In general, we prefer the second parse tree (else matches with closest if)
- So, we have to disambiguate our grammar to reflect this choice
- Unambiguous grammar:

Stmt	→	matchedStmt | unmatchedStmt
matchedStmt	→	if Expr then matchedStmt else matchedStmt |
		Otherstmts
unmatchedStmt	→	if Expr then Stmt |
		if Expr then matchedStmt else unmatchedStmt

Ambiguity disambiguation

- Operator precedence grammar:

```
E → E+E | E*E | E^E | id | (E)
```

- Unambiguous grammar

```
E → E+T | T
T → T*F | F
F → G^F | G
G → id | (E)
```

Precedence

^ (right to left)
* (left to right)
+ (left to right)

Left Recursion

- A grammar is left recursive if it has a non-terminal A such that there is a derivation
    - $A \Rightarrow^{+} A\alpha$ for some string α

- Top-down parsing techniques ***cannot handle*** left-recursive grammars
    - Conversion of left-recursive grammar into an equivalent non-recursive grammar is ***mandatory***.
- Possible ways of left-recursion
    - It may appear in a single step of the derivation (immediate left-recursion)
    - It may appear in more than one step of the derivation

Removing Left Recursion

In general,

$A \to A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ Where $\beta_1 \ \ldots \ \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \to \beta_1 A' \mid \ldots \mid \beta_n A'$

$A' \to \alpha_1 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$ an equivalent grammar

Removing Left Recursion: An example

$$
\begin{aligned}
E &\rightarrow E+T \mid T \\
T &\rightarrow T*F \mid F \\
F &\rightarrow id \mid (E)
\end{aligned}
$$

$\Downarrow$ eliminate immediate left recursion

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +T\ E' \mid \varepsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *F\ T' \mid \varepsilon \\
F &\rightarrow id \mid (E)
\end{aligned}
$$

Why left-recursion is a problem?

- Given
    - `A → Aa | b`

  generate a top-down parse tree from input string ‘aaaaaa’

- On first input symbol ‘a’, you appy first production since second production expect first character to be ‘b’. [Note that you don’t know what’s your second input]
    - $A \Rightarrow Aa \Rightarrow Aaa \Rightarrow \quad \Rightarrow Aaaaaaa$
    - We are waiting to reduce ‘A’ to ‘a’
    - After infinite/many steps, we may get to know that the path we chose was not correct.

Non-immediate Left-recursion

- A grammar cannot be immediately left-recursive, but it still can be left-recursive
- Just elimination of the immediate left-recursion does not guarantee a grammar which is not left-recursive

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Sc \mid d$$

- This grammar is not immediately left-recursive, but it is still left-recursive

$$S \Rightarrow Aa \Rightarrow Sca$$

Or

$$A \Rightarrow Sc \Rightarrow Aac$$

Elimination of left-recursion: Algorithm

Input: A grammar G without *e-moves* or *cycle*
Output: An equivalent grammar without left recursion

1. Arrange non-terminals in some order: $A_1 \dots A_n$
2. for i = 1 to n
   a. for j = 1 to i-1
      i. replace each production of the form

$$A_i \rightarrow A_j\gamma \qquad \Rightarrow \qquad A_i \rightarrow \alpha_1\gamma \mid \alpha_2\gamma \mid \dots \mid \alpha_k\gamma, \quad A_j \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_k$$

   b. eliminate the immediate left-recursions among A_i productions

If there are e-moves, the algorithm does not guarantee to work.

Elimination of left-recursion: Example

- Let grammar G: $S \to Aa \mid b$
  $A \to Ac \mid Sd \mid f$
- Order of non-terminals: S, A
- For S: There is no immediate left recursion in S.
- For A: Replace $A \to Sd$ with $A \to Aad \mid bd \Rightarrow A \to Ac \mid Aad \mid bd \mid f$

  Eliminate the immediate left-recursion in A

  $$A \to bdA' \mid fA'$$
  $$A' \to cA' \mid adA' \mid \varepsilon$$
- So, the resulting equivalent grammar which is not left-recursive is:

  $$S \to Aa \mid b$$
  $$A \to bdA' \mid fA'$$
  $$A' \to cA' \mid adA' \mid \varepsilon$$

Elimination of left-recursion: Exercises

1. $A \rightarrow ABd \mid Aa \mid a$

   $B \rightarrow Be \mid b$

2. $A \rightarrow Ba \mid Aa \mid c$

   $B \rightarrow Bb \mid Ab \mid d$

3. $X \rightarrow XSb \mid Sa \mid b$

   $S \rightarrow Sb \mid Xa \mid a$

Elimination of left-recursion: Solutions

1.

$$A \to aA'$$
$$A' \to BdA' \mid aA' \mid \varepsilon$$
$$B \to bB'$$
$$B' \to eB' \mid \varepsilon$$

2.

$$A \to BaA' \mid cA'$$
$$A' \to aA' \mid \varepsilon$$
$$B \to cA'bB' \mid dB'$$
$$B' \to bB' \mid aA'bB' \mid \varepsilon$$

3.

$$X \to SaX' \mid bX'$$
$$X' \to SbX' \mid \varepsilon$$
$$S \to bX'aS' \mid aS'$$
$$S' \to bS' \mid aX'aS' \mid \varepsilon$$

Left-factoring

- Top-down parser without backtracking (predictive parser) insists that the grammar must be left left-factored

```
stmt   →   if expr then stmt else stmt |
           if expr then stmt
```

- After seeing `if`, we cannot decide which production rule to choose to re-write stmt in the derivation

Left-factoring

- In general,

  $$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

  where α is non-empty and the first symbols of β_1 and β_2 (if they have one) are different
- Choice involved when processing α

  A to $\alpha\beta_1$ or
  A to $\alpha\beta_2$

- Rewrite the grammar as follows:

  $$A \rightarrow \alpha A'$$
  $$A' \rightarrow \beta_1 \mid \beta_2$$

  so, we can immediately expand $A \rightarrow \alpha A'$

Elimination of Left-factoring: Algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix,

$$A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma_1 \mid \ldots \mid \gamma_m$$

Convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid \ldots \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$$

Elimination of Left-factoring: Example

Example 1:

$A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid cdg \mid cdeB \mid cdfB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Example 2:

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$\Downarrow$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

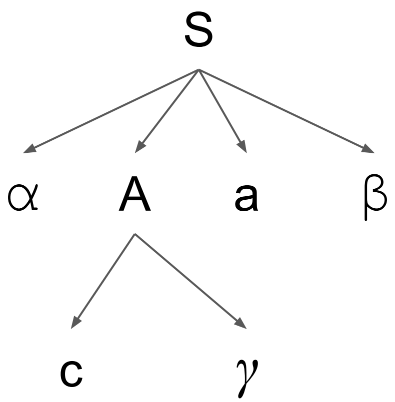
$A'' \rightarrow \varepsilon \mid c$

Lookahead symbols

FIRST() and FOLLOW()

- The construction of top-down and bottom-up parsing is aided by two functions on grammar G
  - FIRST(α): The set of *first character* that can be derived from α
  - FOLLOW(A): The set of *character that can come immediately after* the non-terminal A.

$$S \rightarrow \alpha \ A \ a \ \beta$$
$$A \rightarrow c \ \gamma$$



FIRST(a) = {a}
FIRST(A) = FIRST(c) = {c}
FIRST(S) = FIRST(α) = {...}
FIRST(β) = {...}
FIRST(γ) = {...}

FOLLOW(A) = {a}
FOLLOW(S) = {$}

$: A special symbol for the end marker.

FIRST()

- FIRST(α)

  a. If α is a terminal

     - FIRST(α) = {α}

  b. If α is a non-terminal and $\alpha \longrightarrow \beta_1 \beta_2 \beta_3 \ldots \beta_k$

     - FIRST(α) = FIRST(α) U FIRST(β_i) if $\beta_1 \beta_2 \ldots \beta_{i-1} \Rightarrow^* \varepsilon$

  c. If $\alpha \longrightarrow \varepsilon$

     - FIRST(α) = FIRST(α) U ε

FOLLOW()

- FOLLOW(A)
  a. If A is the start symbol and $ is the special end marker
     - FOLLOW(A) = {$}

  b. If $A \rightarrow \alpha B \beta$
     - FOLLOW(B) = FOLLOW(B) U { FIRST(β) - ε }

  c. If $A \rightarrow \alpha B$ OR $A \rightarrow \alpha B \beta$ with FIRST(β) has ε
     - FOLLOW(B) = FOLLOW(B) U FOLLOW(A)

FIRST() and FOLLOW()

1. G: $A \rightarrow aBe \mid cBd \mid C$
   $B \rightarrow bB \mid \varepsilon$
   $C \rightarrow f$

FIRST(a) = {a}
FIRST(b) = {b}
FIRST(c) = {c}
FIRST(d) = {d}
FIRST(e) = {e}
FIRST(f) = {f}
FIRST(A) = {a, c, f}
FIRST(B) = {b, ε}
FIRST(C) = {f}

FOLLOW(A) = {$}
FOLLOW(B) = {e,d}
FOLLOW(C) = {$}

2. G: $A \rightarrow aBc$
   $B \rightarrow bC$
   $C \rightarrow c \mid \varepsilon$

FIRST(a) = {a}
FIRST(b) = {b}
FIRST(c) = {c}
FIRST(A) = {a}
FIRST(B) = {b}
FIRST(C) = {c, ε}
FOLLOW(A) = {$}
FOLLOW(B) = {c}
FOLLOW(C) = {c}

FIRST() and FOLLOW()

3. G: E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → id | (E)

FIRST(+) = {+}, FIRST(*) = {*}, FIRST(id) = {id}, FIRST('(') = { (}, FIRST(')') = {) }

FIRST(E) = FIRST(T) = FIRST(F) = { id, (}

FIRST(T') = {*, ε}

FIRST(E') = {+, ε}

FOLLOW(E) = FOLLOW(E') = {), $}

FOLLOW(T) = FOLLOW(T') = {+,), $}

FOLLOW(F) = {+, *,), $}