Compiler Construction

Syntax Analysis

Department of CSE

Outline

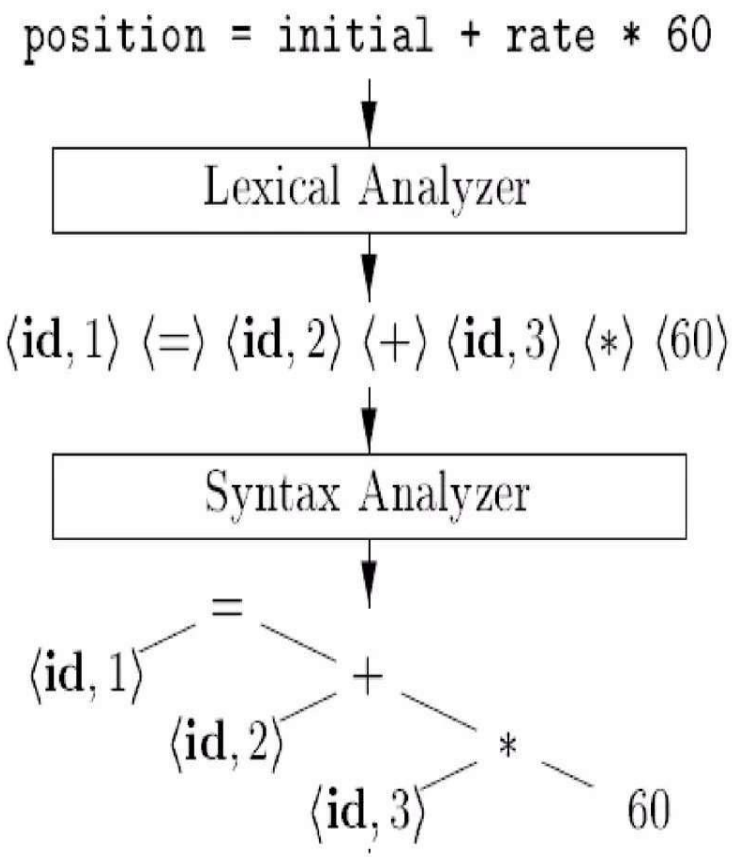
- ✓ Definition - Role of parser
- ✓ Lexical versus Syntactic Analysis
- ✓ Representative Grammars
- ✓ Syntax Error Handling
- ✓ Error recovery strategies
- ✓ Derivations

What is Syntax Analysis?

- **Syntax Analysis** is a second phase of the compiler design process after lexical analysis in which the given input string is checked for the confirmation of rules and structure of the formal grammar.
- **Syntax Analyzer or Parser** analyses the syntactical structure and checks if the given input is in the correct syntax of the programming language or not.
- It does so by getting the input from the tokens and building a data structure, called a **Syntax tree or Parse tree.**

Cont.

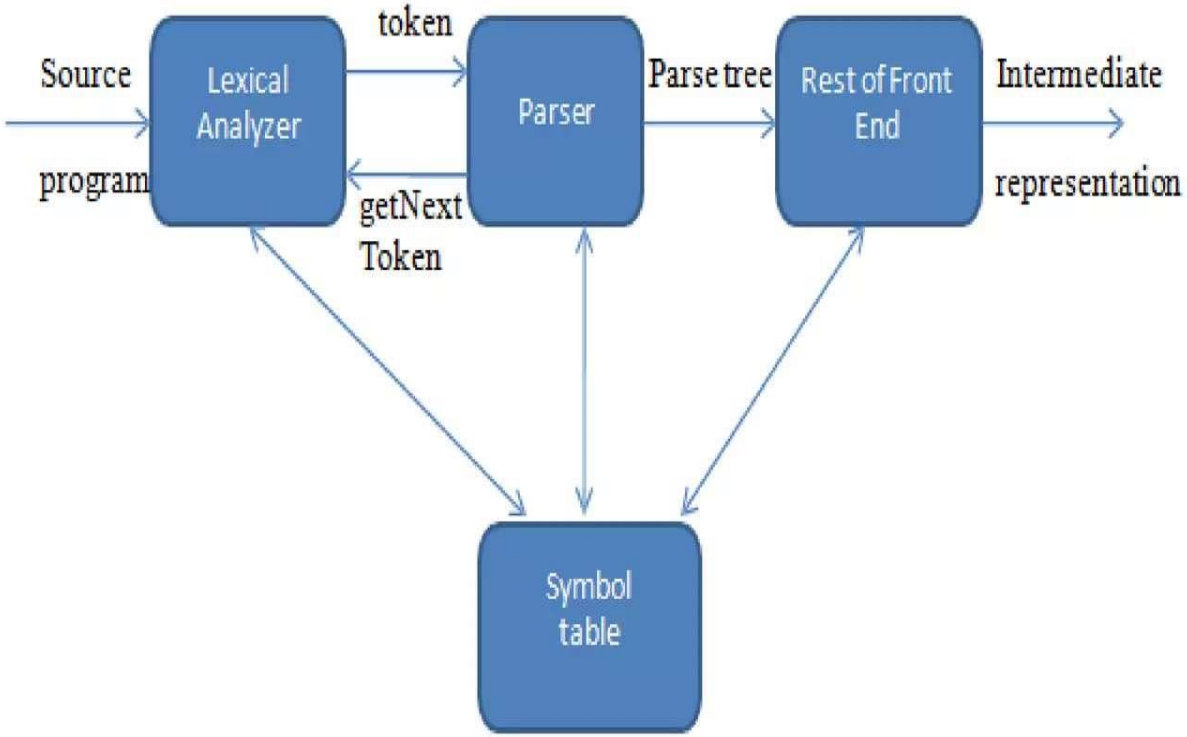
Parse Tree



Cont.

- The parse tree is constructed by using the pre-defined Grammar of the language and the input string.
- If the given input string can be produced with the help of the syntax tree, then the input string is found to be in the correct syntax.
- Otherwise, *error* is reported by syntax analyzer.

Role of the parser



Tasks performed by Parser

- The parser helps to apply rules to the code
- Helps to make sure that each opening brace has a corresponding closing balance
- Helps to detect all types of Syntax errors
- Find the position at which error has occurred
- Clear and accurate description of the error
- Recovery from an error to continue and find further errors in the code
- Should not affect compilation of “correct” programs
- The parse must reject invalid texts by reporting syntax errors

Types of Parsers

- Three types:
  - Universal Parser
  - Top-down Parser
  - Bottom-up Parser
- *Universal Parsers* like CYK (Cocke-Younger-Kasami) algorithm and Earley's Algorithm can parse any grammar but they are inefficient to use in production compilers.
- As implied by their names, *top-down parsers* build parse trees from the top (root) to the bottom (leaves). Eg. LL parser
- *Bottom-up parsers* start from the leaves and work their way up to the root. Eg. LR parser

Syntax Error Handling

Types of Errors

1) Lexical

2) Syntactic

3) Semantic

4) Logical

- **Lexical errors** include misspellings of identiers, keywords, or operators.
  - e.g., the use of an identier *elipseSize* instead of *ellipseSize*
  - missing quotes around text intended as a string
- **Syntactic errors** include misplaced semicolons or extra or missing braces; that is, "{"or "}"
  - Example: In C or Java, the appearance of a *case* statement without an enclosing *switch* is a syntactic error

Syntax Error Handling

Types of Errors

1) Lexical

2) Syntactic

3) Semantic

4) Logical

- **Semantic errors** include type mismatches between operators and operands,
  - e.g., the return of a value in a Java method with result type void.
- **Logical errors** occur when executed code does not produce the expected result.
  - incorrect reasoning on the part of the programmer
  - The use in a C program of the assignment operator = instead of the comparison operator = =

Syntax Error Handling

- The error handler in a parser has goals that are simple to state but challenging to realize:
  - 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-Recovery Strategies

1. Panic-Mode Recovery
2. Phrase-Level Recovery
3. Error Productions
4. Global Correction

Panic-Mode Recovery

- Once an error is found, the parser intends to find designated set of synchronizing tokens by discarding input symbols one at a time.
- Synchronizing tokens are *delimiters, semicolon or }* whose role in source program is clear.
- When parser finds an error in the statement, it ignores the rest of the statement by not processing the input.
- *Advantage:*
  - Simplicity
  - Never get into infinite loop
- *Disadvantage:*
  - Additional errors cannot be checked as some of the input symbols will be skipped.

Phrase-Level Recovery

- When a parser finds an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. The corrections may be
  - Replacing a prefix by some string.
  - Replacing comma by semicolon.
  - Deleting extraneous semicolon.
  - Inserting missing semicolon.
- *Advantage:*
  - It can correct any input string.
- *Disadvantage:*
  - It is difficult to cope up with actual error if it has occurred before the point of detection.

Error Productions

- The use of the error production method can be incorporated if the user is aware of common mistakes that are encountered in grammar in conjunction with errors that produce erroneous constructs.
  - Example: write 5x instead of 5*x
- *Advantage:*
  - If this is used then, during parsing appropriate error messages can be generated and parsing can be continued.
- *Disadvantage:*
  - The disadvantage is that it's difficult to maintain.

Global Correction

- The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free.
- When an erroneous input statement X is fed, it creates a parse tree for some closest error-free statement Y.
- *Advantage:*
  - This may allow the parser to make minimal changes in the source code.
- *Disadvantage:*
  - Due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.

Lexical Vs Syntactic analysis

Lexical analysis	Syntax analysis
It is responsible for converting a sequence of characters into a pattern of tokens.	Process of analyzing a string of symbols either in natural language or computer languages that satisfies the rules of a formal grammar.
Reads the program one character at a time, the output is meaningful lexemes.	Tokens are taken as input and a parse tree is generated as output.
It is the first phase of the compilation process.	It is the second phase of the compilation process.

Lexical Vs Syntactic analysis

Lexical analysis	Syntax analysis
It can also be referred to as lexing and tokenization.	It can also be referred to as syntactic analysis and parsing.
A lexical analyser is a pattern matcher.	A syntax analysis involves forming a tree to identify deformities in the syntax of the program.
Less complex approaches are often used for lexical analysis.	Syntax analysis requires a much more complex approach.

Representative Grammars - CFG

- Context-free grammars are named as such because **any of the production rules in the grammar can be applied regardless of context**—it does not depend on any other symbols that may or may not be around a given symbol that is having a rule applied to it.
- A context free grammar G is defined by four tuple format as

$$G = (V, T, P, S)$$

where,

G − Grammar

V − Set of variables

T − Set of terminals

P − Set of productions

S − Start symbol

Context Free Grammar

- ***Terminals*** are symbols from which strings are formed.
  - **Lowercase letters, i.e., *a, b, c.***
  - **Operators, i.e., +,−, *.**
  - **Punctuation symbols, i.e., comma, parenthesis.**
  - Digits, i.e., 0, 1, 2, · · · ,9.
  - **Boldface letters, i.e., id, if.**
- ***Non-terminals*** are syntactic variables that denote a set of strings.
  - **Uppercase letters, i.e., A, B, C.**
  - Lowercase italic names, i.e., expr, stmt.
- ***Start symbol*** is the head of the production stated first in the grammar.
- ***Production*** is of the form LHS → RHS or *head* → *body*, where head contains only one non-terminal and body contains a collection of terminals and non-terminals.

Context Free Grammar

- Example

E → E + T | E − T | T

T → T * F | T / F | F

F → (E) | *id*

V = {E, T, F}

T = {+ , −, *, /, (,), *id*}

S = {E}

P :

E → E+T

E → E−T

E →T

T → T* F

T→ T / F

T→ F

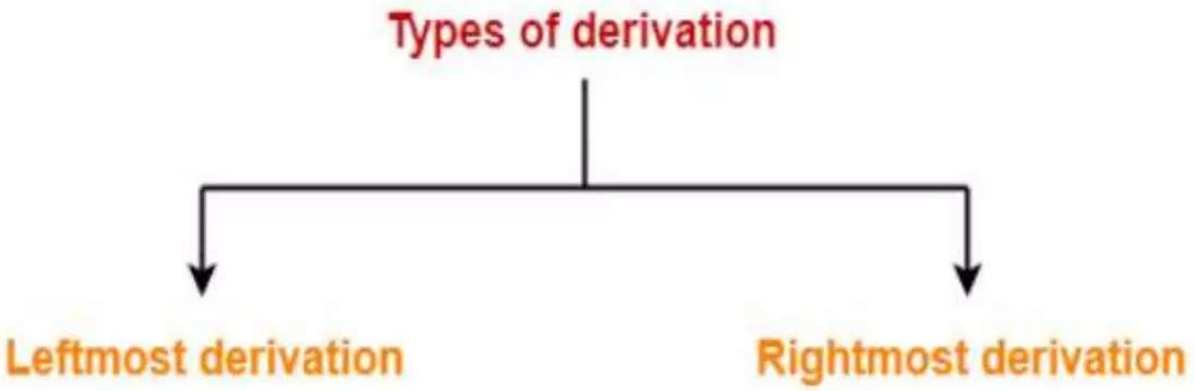
F→ (E)

F→ *id*

Derivations

Parse Tree-

- The process of deriving a string is called as **derivation**.
- The geometrical representation of a derivation is called as a **parse tree** or **derivation tree.**



Leftmost Derivation

- At each and every step the leftmost non-terminal is expanded by substituting its corresponding production to derive a string.

$$\mathrm{E} \rightarrow \mathrm{E} + \mathrm{E} \mid \mathrm{E} * \mathrm{E} \mid \mathbf{id}$$

$$w = \mathbf{id} + \mathbf{id} * \mathbf{id}$$

$$\mathrm{E} \underset{\mathrm{lm}}{\rightarrow} \mathrm{E} + \mathrm{E}$$

$$\mathrm{E} \underset{\mathrm{lm}}{\rightarrow} \mathrm{id} + \mathrm{E} \qquad [\mathrm{E} \rightarrow \mathrm{id}]$$

$$\mathrm{E} \underset{\mathrm{lm}}{\rightarrow} \mathrm{id} + \mathrm{E} * \mathrm{E} \qquad [\mathrm{E} \rightarrow \mathrm{E} * \mathrm{E}]$$

$$\mathrm{E} \underset{\mathrm{lm}}{\rightarrow} \mathrm{id} + \mathrm{id} * \mathrm{E} \qquad [\mathrm{E} \rightarrow \mathrm{id}]$$

$$\mathrm{E} \underset{\mathrm{lm}}{\rightarrow} \mathrm{id} + \mathrm{id} * \mathrm{id} \qquad [\mathrm{E} \rightarrow \mathrm{id}]$$

Rightmost Derivation

- At each and every step the rightmost non-terminal is expanded by substituting its corresponding production to derive a string.

$$\mathrm{E} \rightarrow \mathrm{E}+\mathrm{E} \mid \mathrm{E} * \mathrm{E} \mid \mathbf{id}$$

$$w = \mathbf{id} + \mathbf{id} * \mathbf{id}$$

$$\mathrm{E} \underset{\mathrm{rm}}{\rightarrow} \mathrm{E}+\mathrm{E}$$

$$\mathrm{E} \underset{\mathrm{rm}}{\rightarrow} \mathrm{E}+\mathrm{E} * \mathrm{E} \qquad [\mathrm{E} \rightarrow \mathrm{E} * \mathrm{E}]$$

$$\mathrm{E} \underset{\mathrm{rm}}{\rightarrow} \mathrm{E}+\mathrm{E} * \mathbf{id} \qquad [\mathrm{E} \rightarrow \mathrm{id}]$$

$$\mathrm{E} \underset{\mathrm{rm}}{\rightarrow} \mathrm{E}+\mathbf{id} * \mathbf{id} \qquad [\mathrm{E} \rightarrow \mathrm{id}]$$

$$\mathrm{E} \rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \qquad [\mathrm{E} \rightarrow \mathrm{id}]$$

Derivation Examples

- Leftmost

$S \rightarrow SS+ \mid SS* \mid a$

$$
\begin{aligned}
w &= aa+a* \\
S &\underset{lm}{\rightarrow} SS* \\
S &\underset{lm}{\rightarrow} SS+S* && [S \rightarrow SS+] \\
S &\underset{lm}{\rightarrow} aS+S* && [S \rightarrow a] \\
S &\underset{lm}{\rightarrow} aa+S* && [S \rightarrow a] \\
S &\underset{lm}{\rightarrow} aa+a* && [S \rightarrow a]
\end{aligned}
$$

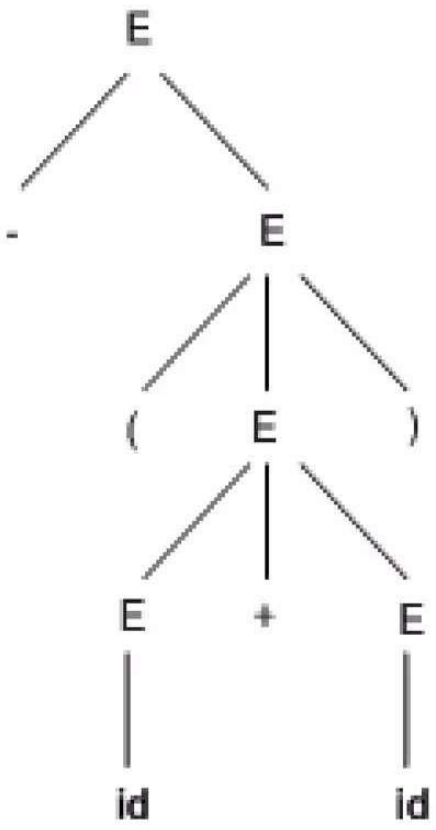
- Rightmost

$S \rightarrow SS+ \mid SS* \mid a$

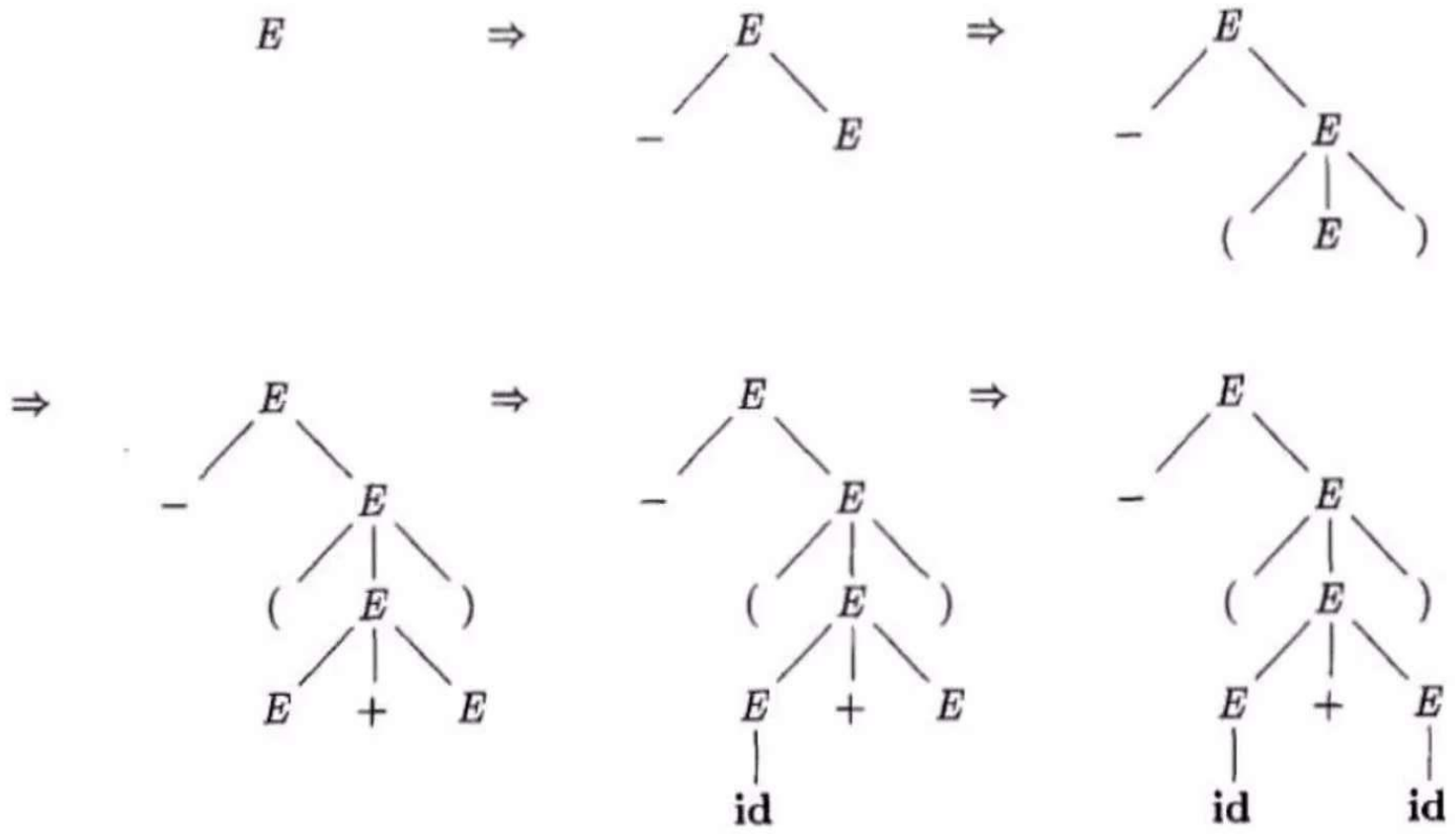
$$
\begin{aligned}
w &= aa+a* \\
S &\underset{rm}{\rightarrow} SS* \\
S &\underset{rm}{\rightarrow} Sa* && [S \rightarrow a] \\
S &\underset{rm}{\rightarrow} SS+a* && [S \rightarrow SS+] \\
S &\underset{rm}{\rightarrow} Sa+a* && [S \rightarrow a] \\
S &\underset{rm}{\rightarrow} aa+a* && [S \rightarrow a]
\end{aligned}
$$

Parse Tree

- Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings.
- Root node of parse tree has the start symbol of the given grammar from where the derivation proceeds.
- Leaves of parse tree represent terminals.
- Each interior node represents productions of grammar.
- If G : E → E+E | E*E | (E) | - E | id is the grammar, then Parse tree for the input string - **(id + id)** is shown.



Parse Tree

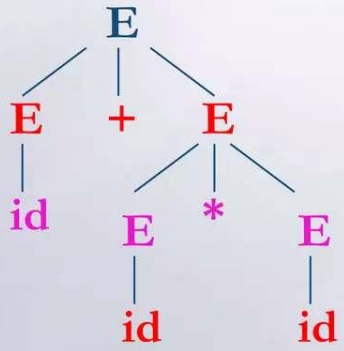


Ambiguity

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

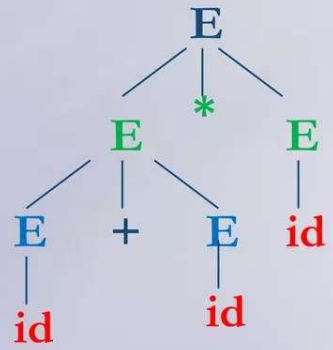
$E \Rightarrow E+E \Rightarrow id+E \Rightarrow id+E*E$

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



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

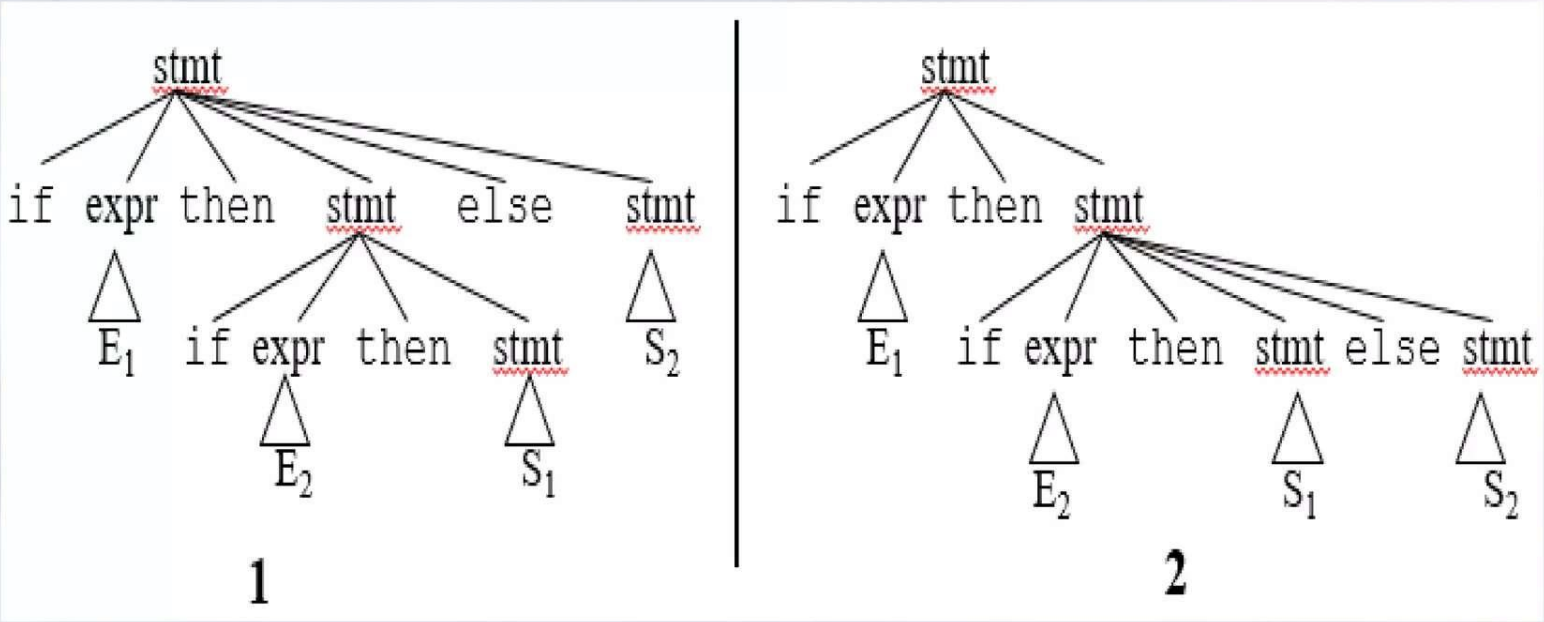
$\Rightarrow id+id*E \Rightarrow id+id*id$



Ambiguity (cont.)

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

`if` E_1 `then` `if` E_2 `then` S_1 `else` S_2



Left Recursion

- A grammar is **left recursive** if it has a **non-terminal A** such that there is a derivation.

$$A \overset{+}{\Rightarrow} A\alpha \quad \text{for some string } \alpha$$

- **Top-down** parsing techniques **cannot** handle **left-recursive** grammars.
- So, we have to **convert** our **left-recursive** grammar into an **equivalent** grammar which is **not** **left-recursive**.
- The left-recursion may **appear** in a single **step** of the derivation (***immediate left-recursion***), or may **appear** in more **than one** step of the derivation.

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\,A' \mid \ldots \mid \beta_n\,A'$

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

Immediate Left-Recursion -- Example

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

$\Downarrow$ **eliminate** immediate left recursion

$E \rightarrow T\ E'$

$E' \rightarrow +T\ E' \mid \varepsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *F\ T' \mid \varepsilon$

$F \rightarrow id \mid (E)$

Left-Recursion -- Problem

- A grammar **cannot** be **immediately** left-recursive, but it **still** can be **left-recursive**.
- By just **eliminating** the immediate left-recursion, we may **not** get 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

$A \Rightarrow Sc \Rightarrow Aac$ causes to a left-recursion

- So, we have to **eliminate** all left-recursions from our grammar

Eliminate Left-Recursion -- Example

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: S, A

<u>for S</u>:
- we do not enter the inner loop.
- there is no immediate left recursion in S.

<u>for A</u>:
- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$
  So, we will have $A \rightarrow Ac \mid Aad \mid bd \mid f$
- Eliminate the immediate left-recursion in A
  $A \rightarrow bdA' \mid fA'$
  $A' \rightarrow cA' \mid adA' \mid \varepsilon$

Cont.

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow Aa \mid b$

$A \rightarrow bdA' \mid fA'$

$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Left-Factoring

- A predictive parser (a top-down parser without backtracking) **insists** that the grammar must be **left-factored**.

  grammar ➔ a new equivalent grammar suitable for predictive parsing

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

- when we see **if**, we **cannot** now which **production** rule to **choose** to re-write **stmt** in the derivation.

Left-Factoring (cont.)

- 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.

- when processing α we cannot know whether expand

  A to $\alpha\beta_1$ or

  A to $\alpha\beta_2$

- But, if we re-write the grammar as follows

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring -- Algorithm

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

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

convert it into

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

Left-Factoring – Example1

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

$\Downarrow$

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

$A' \rightarrow bB \mid B$

$\Downarrow$

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

$A' \rightarrow bB \mid B$

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

Left-Factoring – Example2

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

$\Downarrow$

$A \rightarrow aA' \mid b$

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

$\Downarrow$

$A \rightarrow aA' \mid b$

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

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

References

- *Alfred V Aho, Jeffery D Ullman, Ravi Sethi, "Compilers, Principles techniques and tools", Pearson Education 2011*
- https://www.gatevidyalay.com
- https://electricalvoice.com

THANK YOU