# Syntax Analysis

Koya University
Dr. Akar Hawree Taher

gar code'aka nwsrabw :
fi x = 2;

tokenize:
<identifier, fi>
<identifier, x>
<operator, =>
<operand, 2>
<symbol, ;>

wshakan wak wsha tawawn wata la lexical analysis be kesha darwata syntax analysis, lawe ba pey CFG'i zmanaka check dakretawa ka ayaa tawawa ya na. katekish la lexical analysis kesha drwstabe ka aw code'ay nwsrawa lexeme(wsha)'ayaki teda be ka unknown token class bet

# Syntax Analysis

- Syntax analysis or parsing is the second phase of a compiler.
- We have seen that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata. CFG = push-down automata
- CFG, on the other hand, is a superset of Regular Grammar, as depicted below:

wenaa krawa


Context Free
Grammar
Regular
Grammar

# Syntax Analysis (cont'd)

awash awa dagayanet ka

- It implies that every Regular Grammar is also context-free, but there exists some problems, which are beyond the scope of Regular Grammar. CFG is a helpful tool in describing the syntax of programming languages.

ka la daraway baznay rezmani asaiyn

# Context-Free Grammar

- A context-free grammar has four components:
- A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.

rstasazi

# Context-Free Grammar (cont'd)

- A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **non- terminals**, called the right side of the production.

shewaz

production = equation : pet dalet non-terminal'akan w terminal'akan chi drwst dakan

- One of the non-terminals is designated as the start symbol (S); from where the production begins.

# Context-Free Grammar (cont'd)

- The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

# Syntax Analyzers

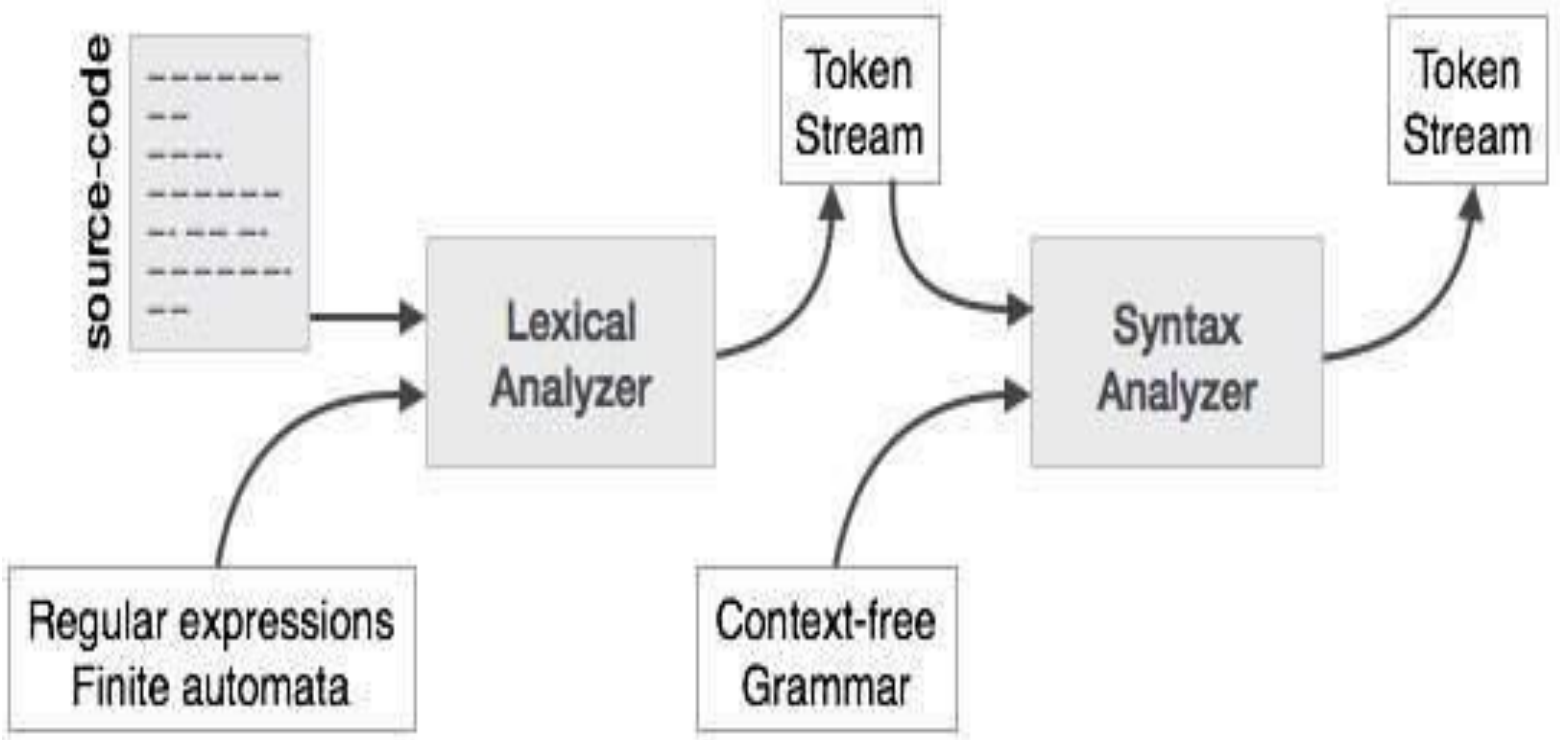
- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a **parse tree**.
- The parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.

ba anjam dagayanet

that is

- Parsers are expected to parse the whole code even if some errors exist in the program.

# Syntax Analyzers (Cont'd)



# Derivation

- A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:

rstayy

bryardan la sar aw non-terminal'ay ka bryara bgordret

- Deciding the non-terminal which is to be replaced.

bryardan la sar aw production rule'ay ka ba hoyawa non-terminal'aka dagordret

- Deciding the production rule, by which, the non-terminal will be replaced.

- To decide which non-terminal to be replaced with production rule, we can have two options.

# Derivation

- Left-most Derivation
  - If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.
- Right-most Derivation
  - If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

# Derivation (Cont'd)

- **Example**
- Production rules:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

Input string: id + id * id

# Derivation (Cont'd)

The left-most derivation is:

$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$

We take the left-most derivation of a + b * c

# Derivation (Cont'd)

The right-most derivation is:

$E \rightarrow E + E$

$E \rightarrow E + E * E$

$E \rightarrow E + E * id$

$E \rightarrow E + id * id$

$E \rightarrow id + id * id$

# Parse Tree

- A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Let us see this by an example from the last topic.

# Parse Tree (Cont'd)

The left-most derivation is:

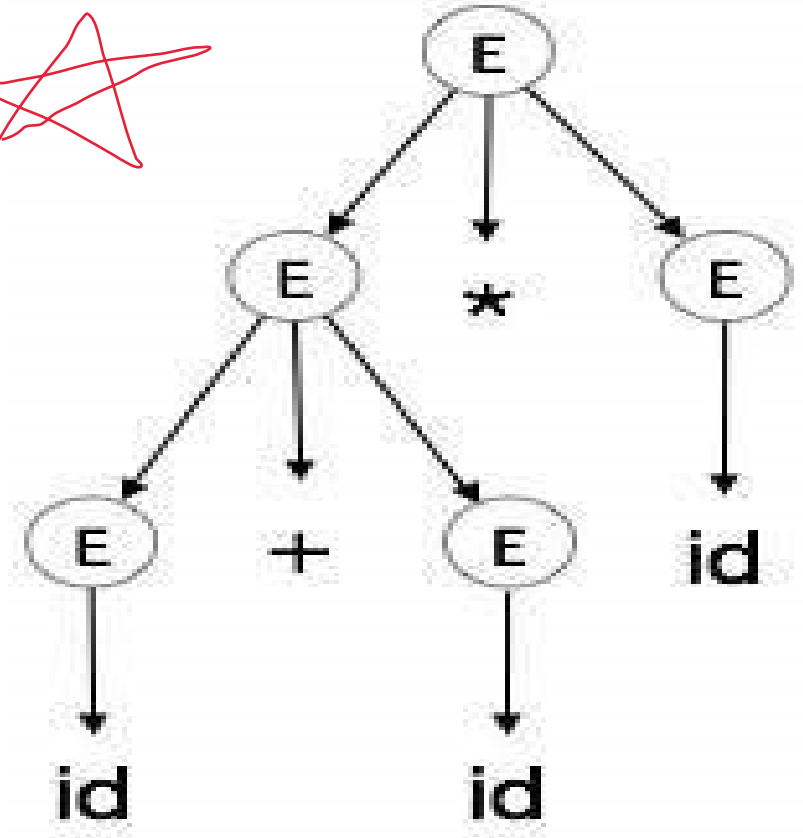
$E \rightarrow E * E$

$E \rightarrow E + E * E$

$E \rightarrow id + E * E$

$E \rightarrow id + id * E$

$E \rightarrow id + id * id$



# Parse Tree (Cont'd)

- In a parse tree:
  - All leaf nodes are terminals.
  - All interior nodes are non-terminals.
  - In-order traversal gives original input string.

# Ambiguity

- A grammar G is said to be ambiguous if it has more than one parse tree (left or right derivation) for at least one string.
- **Example**

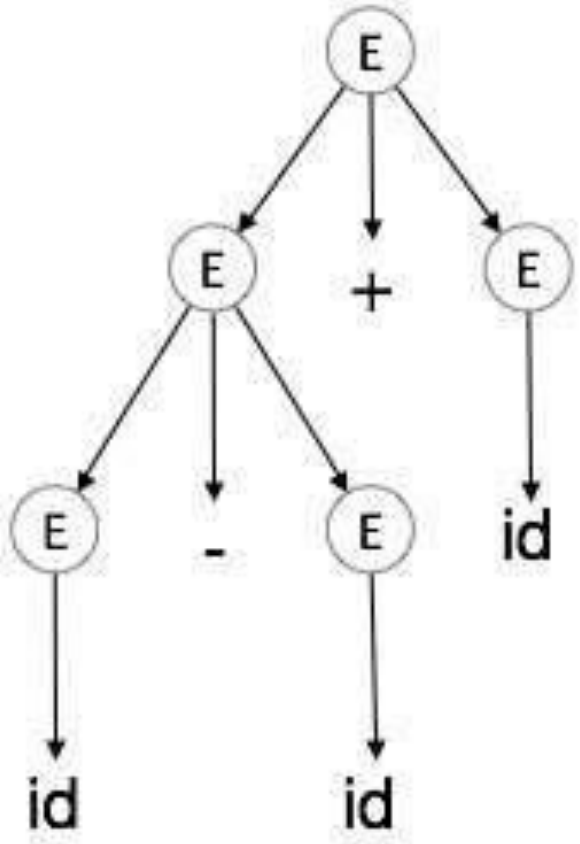
$E \rightarrow E + E$

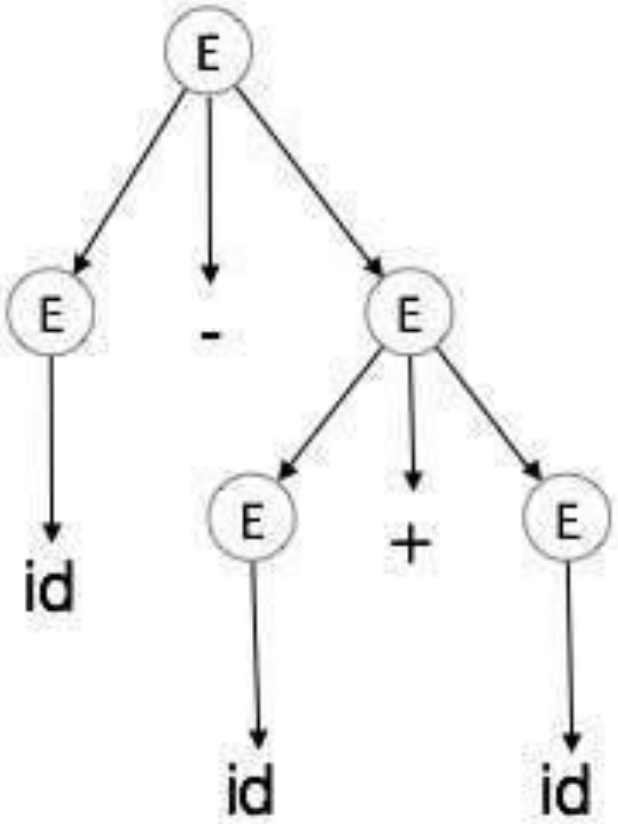
$E \rightarrow E - E$

$E \rightarrow id$

For the string id + id – id, the above grammar generates two parse trees:

# Ambiguity (Cont'd)



# Ambiguity (Cont'd)

- The language generated by an ambiguous grammar is said to be **inherently ambiguous**. Ambiguity in grammar is not good for a compiler construction. No method can detect and remove ambiguity automatically, but it can be removed by either re-writing the whole grammar without ambiguity, or by setting and following associativity and precedence constraints.

# Associativity

- If an operand has operators on both sides, the side on which the operator takes this operand is decided by the associativity of those operators. If the operation is left-associative, then the operand will be taken by the left operator or if the operation is right-associative, the right operator will take the operand.
- **Example**
- Operations such as Addition, Multiplication, Subtraction, and Division are left associative. If the expression contains:

  op id op id

# Associativity (Cont'd)

- op id op id: will be evaluated as:

(id op id) op id

For example, (id + id) + id

Operations like Exponentiation are right associative, i.e., the order of evaluation in the same expression will be:

id op (id op id)

For example, id ^ (id ^ id)

# Precedence

- If two different operators share a common operand, the precedence of operators decides which will take the operand.
- That is, 2+3*4 can have two different parse trees, one corresponding to (2+3)*4 and another corresponding to 2+(3*4). By setting precedence among operators, this problem can be easily removed.

# Precedence (Cont'd)

- As in the previous example, mathematically * (multiplication) has precedence over + (addition), so the expression 2+3*4 will always be interpreted as: 2+(3*4)
- These methods decrease the chances of ambiguity in a language or its grammar

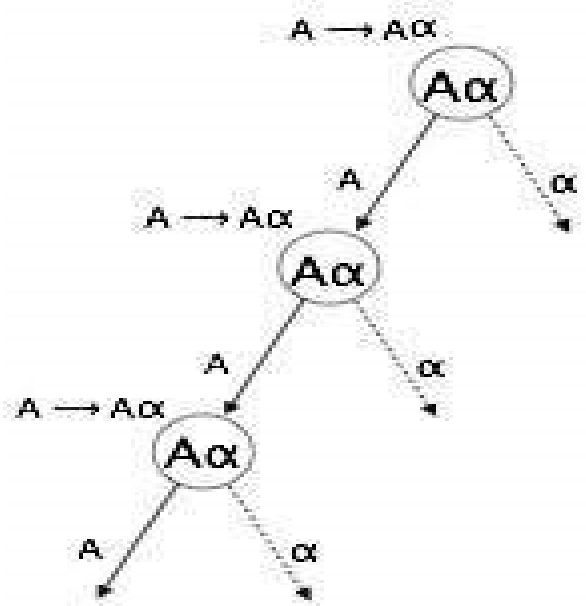
# Left Recursion

- A grammar becomes left-recursive if it has any non-terminal 'A' whose derivation contains 'A' itself as the left-most symbol.
- Left-recursive grammar is a problematic situation for top-down parsers.
- Top-down parsers start parsing from the Start symbol, which is non-terminal.
- So, when the parser encounters the same non-terminal in its derivation, it becomes hard for it to judge when to stop parsing the left non-terminal and it goes into an infinite loop.

# Left Recursion (Cont'd)

- **Example:**

(1) A => Aα | β




(2) S => Aα | β

A => Sd

- A top-down parser will first parse the A, which in-turn will yield a string consisting of A itself and the parser may go into a loop forever.

# Removal of Left Recursion

- To remove left recursion is to use the following technique:

  The production

  $A \Rightarrow A\alpha \mid \beta$

  is converted into following productions

  $A \Rightarrow \beta A'$

  $A' \Rightarrow \alpha A' \mid \varepsilon$ A' = A prime

- This does not impact the strings derived from the grammar, but it removes immediate left recursion.

# Removal of Left Recursion (cont'd)

- Example: L➔L,S|S

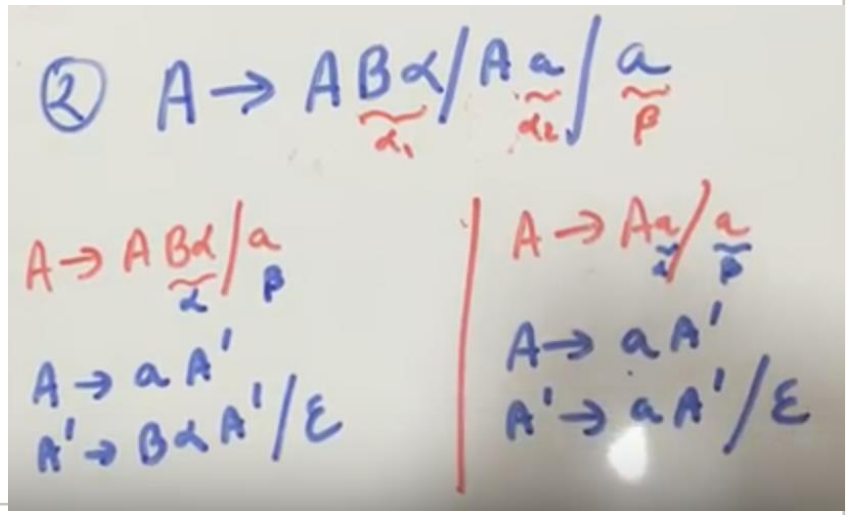
L➔SL'

L'➔,SL'|ε

- Example: A➔ABα|Aa|a

Example:
A --> AB@ | a@ | Aa | a

Solution:
A --> aA' | a@A'
A' --> B@A' | aA' | €



# Left Factoring

- If more than one grammar production rules has a common prefix string, then the top-down parser cannot make a choice as to which of the production it should take to parse the string in hand.
- **Example**

If a top-down parser encounters a production like

$A \Longrightarrow \alpha\beta \mid \alpha\gamma \mid \ldots$

# Left Factoring (Cont'd)

- Then it cannot determine which production to follow to parse the string as both productions are starting from the same terminal (or non-terminal). To remove this confusion, we use a technique called left factoring.
- Left factoring transforms the grammar to make it useful for top-down parsers. In this technique, we make one production for each common prefixes and the rest of the derivation is added by new productions.

# Left Factoring (Cont'd)

- **Example**

The above productions can be written as

$A \Rightarrow \alpha A'$

$A' \Rightarrow \beta \mid \gamma \mid \ldots$

Solve these:
1) A --> @E | @K | M
2) A --> @R | @W | bA | F

- Now the parser has only one production per prefix which makes it easier to take decisions.