CS335: Syntax Analysis

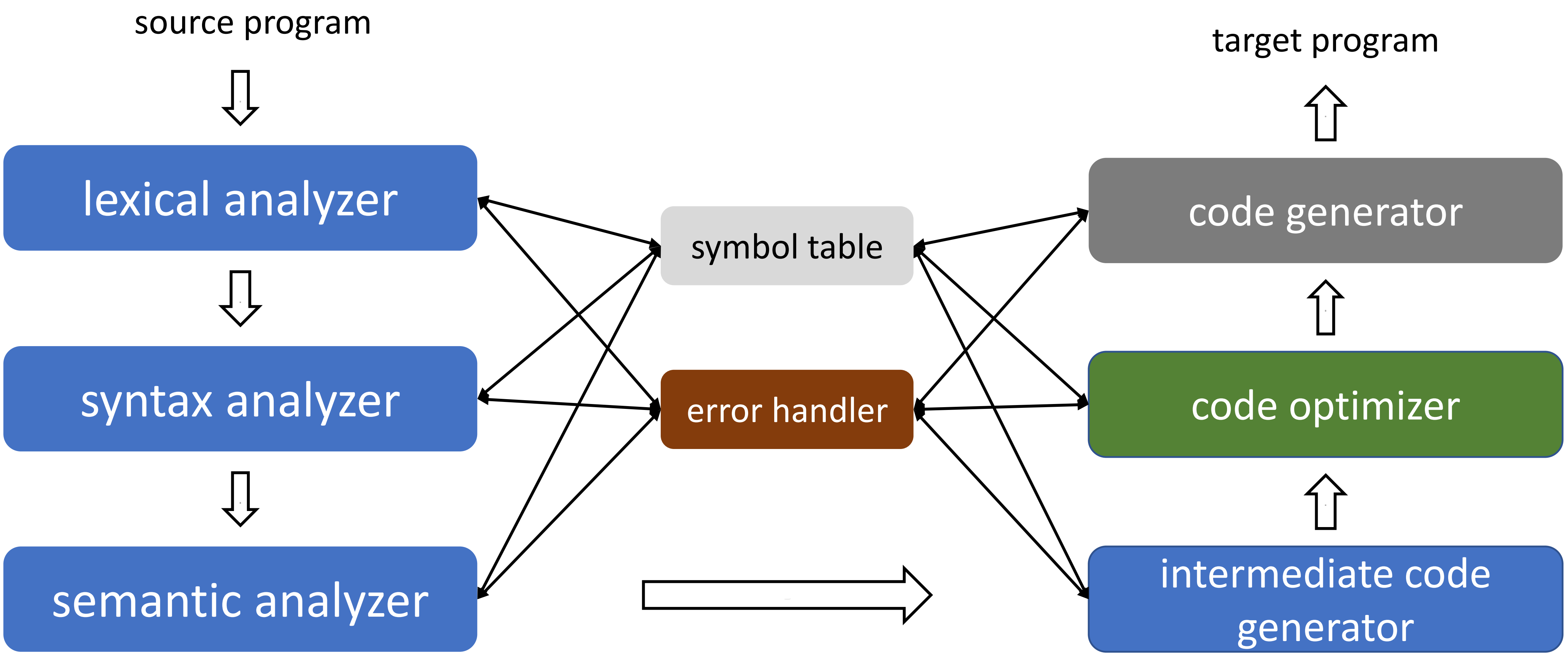
Swarnendu Biswas

Semester 2019-2020-II

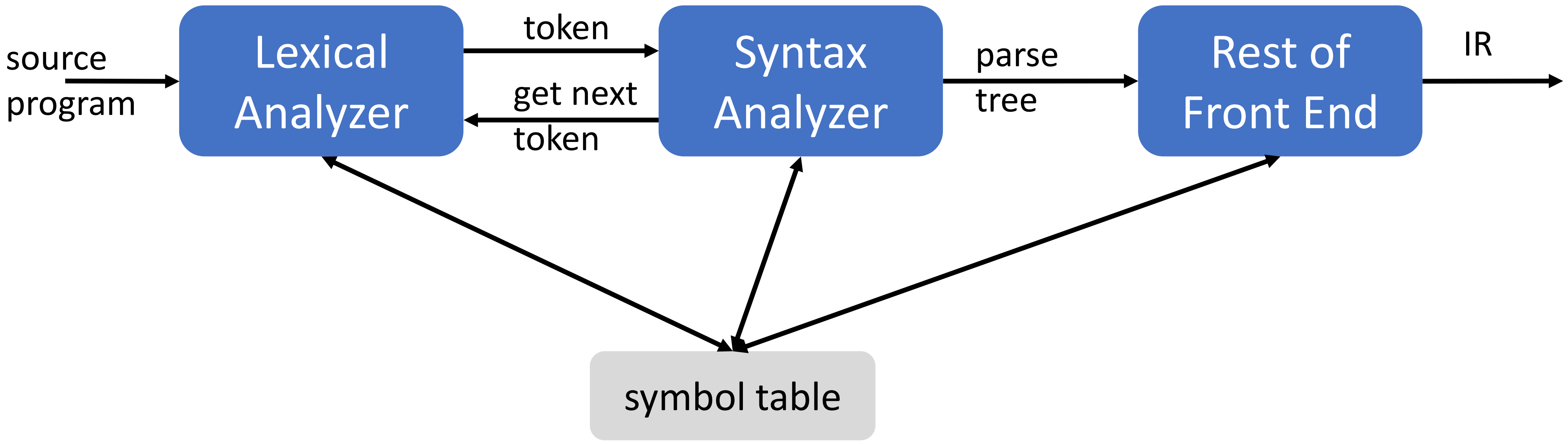
CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

An Overview of Compilation



Parser Interface



Need for Checking Syntax

- Given an input program, scanner generates a stream of tokens classified according to the syntactic category
- The parser determines if the input program, represented by the token stream, is a valid sentence in the programming language
- The parser attempts to build a derivation for the input program, using a grammar for the programming language
  - If the input stream is a valid program, parser builds a valid model for later phases
  - If the input stream is invalid, parser reports the problem and diagnostic information to the user

Syntax Analysis

- Given a programming language grammar G and a stream of tokens s, parsing tries to find a derivation in G that produces s
- In addition, a syntax analyser
  - Forward the information as IR to the next compilation phases
  - Handle errors if the input string is not in $L(G)$

Context-Free Grammars

- A context-free grammar (CFG) G is a quadruple (T, NT, S, P)

T	Set of terminal symbols (also called words) in the language $L(G)$
NT	Set of nonterminal symbols that appear in the productions of G
S	Goal or start symbol of the grammar G
P	Set of productions (or rules) in G

Context-Free Grammars

- Terminal symbols correspond to syntactic categories returned by the scanner
  - Terminal symbol is a word that can occur in a sentence
- Nonterminals are syntactic variables introduced to provide abstraction and structure in the productions
- S represents the set of sentences in $L(G)$
- Each rule in P is of the form $\boldsymbol{NT} \rightarrow (\boldsymbol{T} \cup \boldsymbol{NT})^*$

Definitions

- Derivation is a a sequence of rewriting steps that begins with the grammar G's start symbol and ends with a sentence in the language

$$S \overset{+}{\Rightarrow} w \text{ where } w \in L(G)$$

- At each point during derivation process, the string is a collection of terminal or nonterminal symbols

$$\alpha A \beta \rightarrow \alpha \gamma \beta \text{ if } A \rightarrow \gamma$$

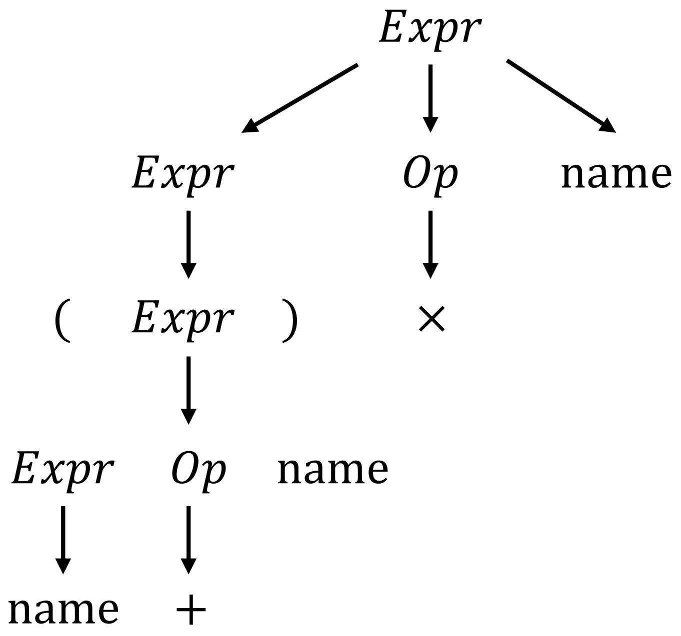
- Such a string is called a sentential form if it occurs in some step of a valid derivation
- A sentential form can be derived from the start symbol in zero or more steps

Example of a CFG

CFG	$(\boldsymbol{a}+\boldsymbol{b})\times\boldsymbol{c}$
$Expr \rightarrow (\ Expr\ )$	$Expr \rightarrow Expr\ Op\ \text{name}$
$\quad\quad \mid Expr\ Op\ \text{name}$	$\quad\quad \rightarrow Expr \times \text{name}$
$\quad\quad \mid \text{name}$	$\quad\quad \rightarrow (Expr) \times \text{name}$
$Op \rightarrow + \mid - \mid \times \mid \div$	$\quad\quad \rightarrow (Expr\ Op\ \text{name}) \times \text{name}$
	$\quad\quad \rightarrow (Expr + \text{name}) \times \text{name}$
	$\quad\quad \rightarrow (\text{name} + \text{name}) \times \text{name}$

Sentential Form and Parse Tree

$$Expr \rightarrow Expr\ Op\ \text{name}$$
$$\rightarrow Expr \times \text{name}$$
$$\rightarrow (Expr) \times \text{name}$$
$$\rightarrow (Expr\ Op\ \text{name}) \times \text{name}$$
$$\rightarrow (Expr + \text{name}) \times \text{name}$$
$$\rightarrow (\text{name} + \text{name}) \times \text{name}$$



Parse Tree

Parse Tree

- A parse tree is a graphical representation of a derivation
  - Root is labeled with by the start symbol S
  - Each internal node is a nonterminal, and represents the application of a production
  - Leaves are labeled by terminals and constitute a sentential form, read from left to right, called the **yield** or **frontier** of the tree
- Parse tree filters out the order in which productions are applied to replace nonterminals
  - It just represents the rules applied

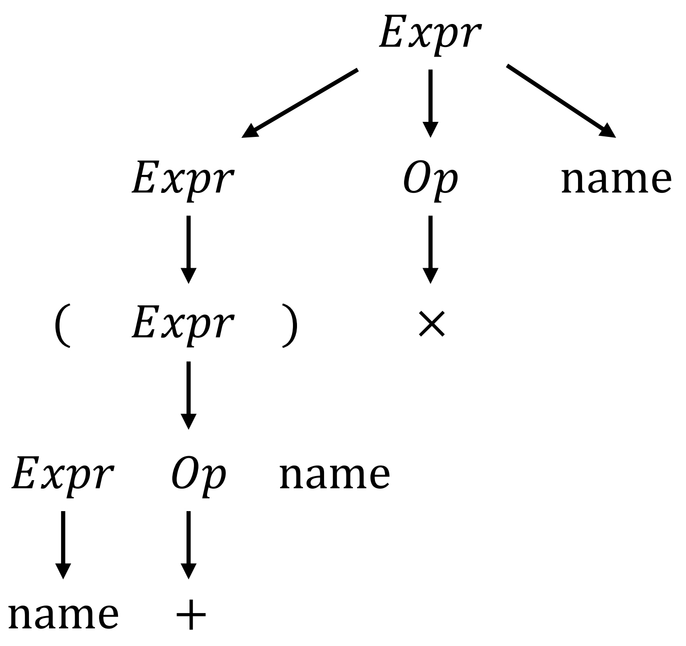
Derivations

- At each step during derivation, we have **two choices** to make
    1. Which nonterminal to rewrite?
    2. Which production rule to pick?

- Rightmost (or canonical) derivation rewrites the rightmost nonterminal at each step, denoted by $\alpha \underset{rm}{\Rightarrow} \beta$
    - Similarly, leftmost derivation rewrites the leftmost nonterminal at each step, denoted by $\alpha \underset{lm}{\Rightarrow} \beta$
- Every leftmost derivation can be written as $wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$

Leftmost Derivation

$$
\begin{aligned}
Expr &\rightarrow Expr\ Op\ \text{name} \\
&\rightarrow (Expr)\ Op\ \text{name} \\
&\rightarrow (Expr\ Op\ \text{name})\ Op\ \text{name} \\
&\rightarrow (\text{name}\ Op\ \text{name})\ Op\ \text{name} \\
&\rightarrow (\text{name} +\ \text{name})\ Op\ \text{name} \\
&\rightarrow (\text{name} +\ \text{name})\ \times \text{name}
\end{aligned}
$$



Parse Tree

Ambiguous Grammars

- A grammar G is ambiguous if some sentence in $L(G)$ has more than one rightmost (or leftmost) derivation

- An ambiguous grammar can produce multiple derivations and parse trees

Example of Ambiguous Grammar

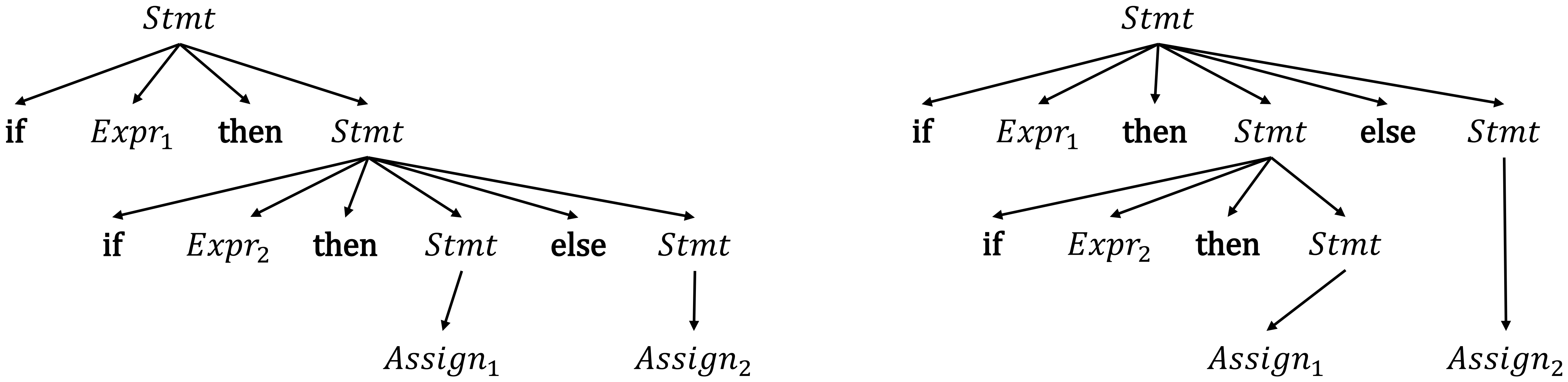
- A grammar G is ambiguous if some sentence in $L(G)$ has more than one rightmost (or leftmost) derivation

- An ambiguous grammar can produce multiple derivations and parse trees

$$
\begin{aligned}
Stmt \rightarrow\ & \textbf{if}\ Expr\ \textbf{then}\ Stmt \\
\mid\ & \textbf{if}\ Expr\ \textbf{then}\ Stmt\ \textbf{else}\ Stmt \\
\mid\ & Assign
\end{aligned}
$$

Ambiguous Dangling-Else Grammar

if $Expr_1$ **then if** $Expr_2$ **then** $Assign_1$ **else** $Assign_2$



Dealing with Ambiguous Grammars

- Ambiguous grammars are problematic for compilers
  - Compilers use parse trees to interpret the meaning of the expressions during later stages
  - Multiple parse trees can give rise to multiple interpretations

- Fixing ambiguous grammars
  - Transform the grammar to remove the ambiguity
  - Include rules to disambiguate during derivations
    - For e.g., associativity and precedence

Fixing the Ambiguous Dangling-Else Grammar

- In all programming languages, an `else` is matched with the closest `then`

$$Stmt \rightarrow \textbf{if } Expr \textbf{ then } Stmt$$
$$\mid \textbf{if } Expr \textbf{ then } ThenStmt \textbf{ else } Stmt$$
$$\mid Assign$$
$$ThenStmt \rightarrow \textbf{if } Expr \textbf{ then } ThenStmt \textbf{ else } ThenStmt$$
$$\mid Assign$$

Fixed Dangling-Else Grammar

if $Expr_1$ **then if** $Expr_2$ **then** $Assign_1$ **else** $Assign_2$

$\Downarrow$

$Stmt \rightarrow$ **if** $Expr$ **then** $Stmt$

$\rightarrow$ **if** $Expr$ **then if** $Expr$ **then** $ThenStmt$ **else** $Stmt$

$\rightarrow$ **if** $Expr$ **then if** $Expr$ **then** $ThenStmt$ **else** $Assign$

$\rightarrow$ **if** $Expr$ **then if** $Expr$ **then** $Assign$ **else** $Assign$

Interpreting the Meaning

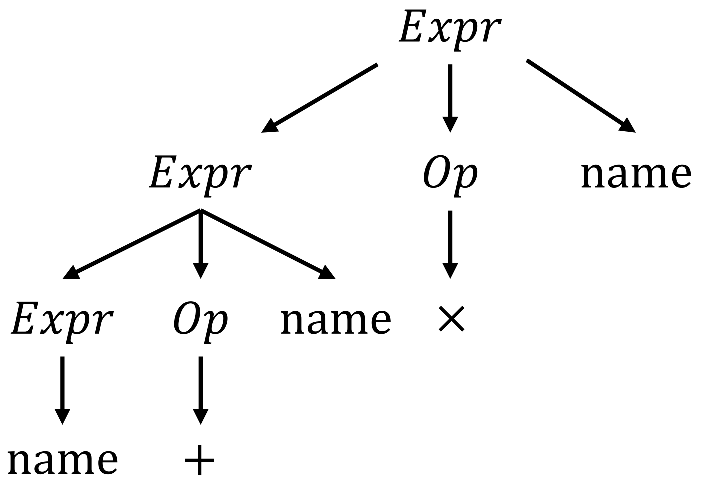
CFG	$\boldsymbol{a + b \times c}$
$Expr \rightarrow (Expr)$	$Expr \rightarrow Expr\ Op\ \text{name}$
$\mid Expr\ Op\ \text{name}$	$\rightarrow Expr \times \text{name}$
$\mid \text{name}$	$\rightarrow Expr\ Op\ \text{name} \times \text{name}$
$Op \rightarrow + \mid - \mid \times \mid \div$	$\rightarrow Expr + \text{name} \times \text{name}$
	$\rightarrow \text{name} + \text{name} \times \text{name}$

rightmost derivation

Corresponding Parse Tree

$$\boldsymbol{a} + \boldsymbol{b} \times \boldsymbol{c}$$

$$\begin{aligned} Expr &\rightarrow Expr\ Op\ \text{name} \\ &\rightarrow Expr \times \text{name} \\ &\rightarrow Expr\ Op\ \text{name} \times \text{name} \\ &\rightarrow Expr + \ \text{name} \times \text{name} \\ &\rightarrow \text{name} + \ \text{name} \times \text{name} \end{aligned}$$

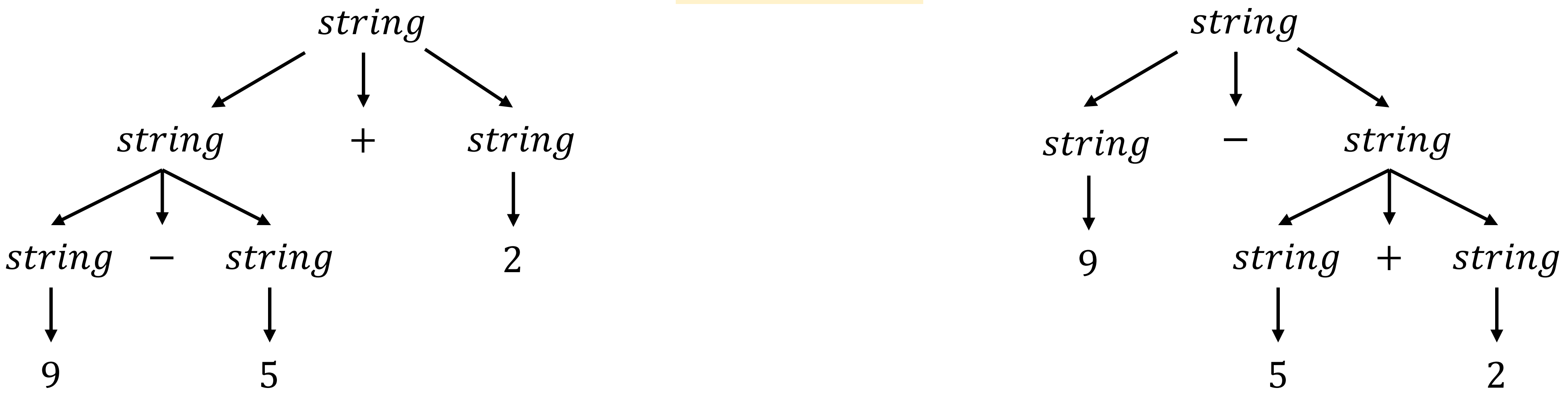


How do we evaluate the expression?

Associativity

$$string \rightarrow string + string \mid string - string \mid 0 \mid 1 \mid 2 \mid \ldots \mid 9$$

$$9 - 5 + 2$$



Associativity

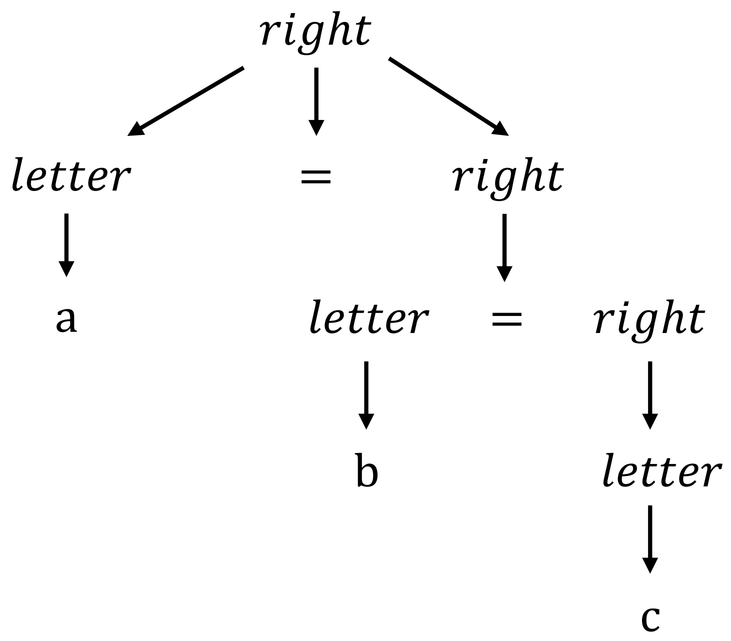
- If an operand has operator on both the sides, the side on which operator takes this operand is the associativity of that operator
  - +, -, *, / are left associative
  - ^, = are right associative

- Grammar to generate strings with right associative operators

$$right \rightarrow letter = right | letter$$
$$letter \rightarrow a | b | \ldots | z$$

Parse Tree for Right Associative Grammars

```
a = b = c
```



Encode Precedence into the Grammar

$Start \rightarrow Expr$

$Expr \rightarrow Expr + Term | Expr - Term | Term$

$Term \rightarrow Term \times Factor | Term \div Factor | Factor$

$Factor \rightarrow (Expr) | \text{num} | \text{name}$

priority ↓

Corresponding Parse Tree

$$\boldsymbol{a - b + c}$$

$Start \rightarrow Expr$

$\rightarrow Expr + Term$

$\rightarrow Expr + Factor$

$\rightarrow Expr + \text{ name}$

$\rightarrow Expr\ - Term + \text{name}$

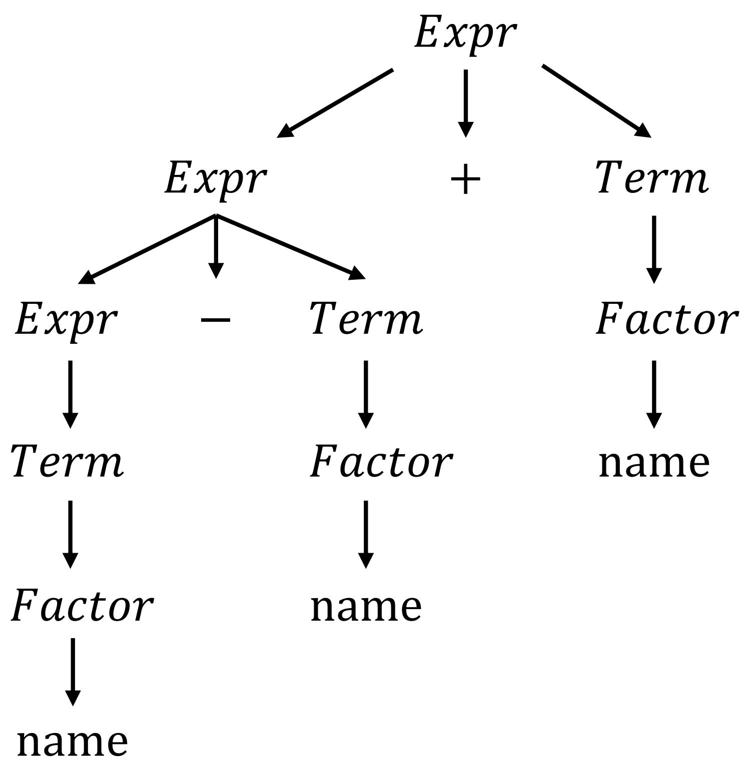
$\rightarrow Expr - Factor + \text{name}$

$\rightarrow Expr - \text{name} + \text{name}$

$\rightarrow Term - \text{name} + \text{name}$

$\rightarrow Factor - \text{name} + \text{name}$

$\rightarrow \text{name} - \text{name} + \text{name}$



Types of Parsers

Top-down

- Starts with the root and grows the tree toward the leaves

Bottom-up

- Starts with the leaves and grow the tree toward the root

Universal

- More general algorithms, but inefficient to use in production compilers

Error Handling

- The scanner cannot deal with all errors
- Common source of programming errors
    - Lexical errors
        - For e.g., illegal characters, missing quotes around strings
    - Syntactic errors
        - For e.g., misspelled keywords, misplaced semicolons or extra or missing braces
    - Semantic errors
        - For e.g., type mismatches between operators and operands, undeclared variables
    - Logical errors

Handling Errors

Panic-mode recovery

- Parser discards input symbols one at a time until a **synchronizing** token is found
- Synchronizing tokens are usually delimiters (for e.g., ; or })

Phrase-level recovery

- Perform local correction on the remaining input
- Can go into an infinite loop because of wrong correction, or the error may have occurred before it is detected

Handling Errors

Error productions

- Augment the grammar with productions that generate erroneous constructs
- Works only for common mistakes, complicates the grammar

Global correction

- Given an incorrect input string x and grammar G, find a parse tree for a related string y such that the number of modifications (insertions, deletions, and changes) of tokens required to transform x into y is as small as possible

Context-Free vs Regular Grammar

- CFGs are more powerful than REs
  - Every regular language is context-free, but not vice versa
  - We can create a CFG for every NFA that simulates some RE

- Language that can be described by a CFG but not by a RE

$$L = \{a^n b^n \mid n \geq 1\}$$

Limitations of Syntax Analysis

- Cannot determine whether
  - A variable has been declared before use
  - A variable has been initialized
  - Variables are of types on which operations are allowed
  - Number of formal and actual arguments of a function match

- These limitations are handled during semantic analysis

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, Chapters 2 and 4.
- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, Chapter 3.