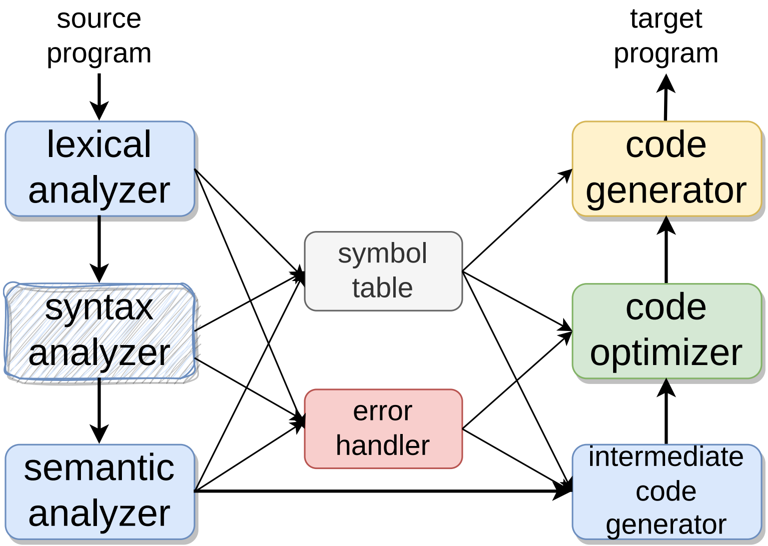
CS 335: Syntax Analysis

Swarnendu Biswas

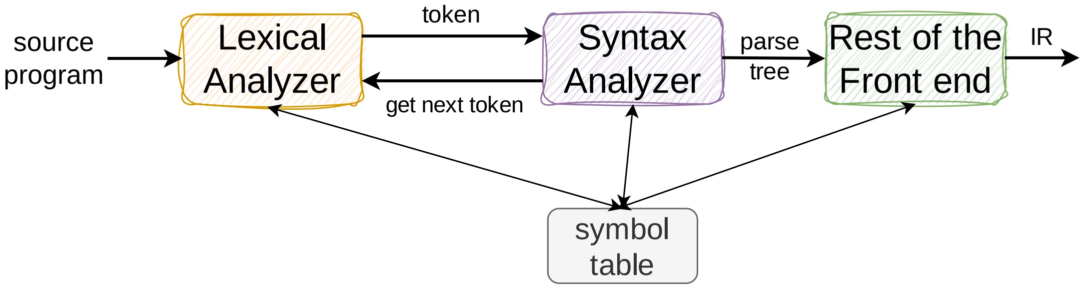
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2023-24-II

An Overview of Compilation



Interfacing with Parser



Syntax Analysis

- Given an input program, a scanner generates a stream of tokens classified according to the syntactic categories of a programming language PL
- Given a grammar G for PL[1,2], a parser determines if the input program, represented by the token stream s, is a **valid sentence** in PL
  - The parser attempts to build a derivation for s using G
  - If the input stream is a valid program, the parser builds a model (e.g., IR) for later phases
  - If the input stream is invalid (i.e., $s \notin L(G)$), the parser reports the problem and diagnostic information to the user

[1] Java 17 Grammar

[2] Python 3.12 Grammar

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 $L(G)$. A terminal symbol is a word that can occur in a sentence and correspond to syntactic categories returned by the scanner.
NT	Set of nonterminal symbols that appear in the productions of G. Nonterminals are syntactic variables that provide abstraction and structure in the productions.
S	Goal or start symbol of the grammar G. S represents the set of sentences in $L(G)$.
P	Set of productions (or rules) in G. Each rule in P is of the form $NT \rightarrow (T \cup NT)^*$.

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

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

Definitions

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

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

- At each point during the 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 S in zero or more steps

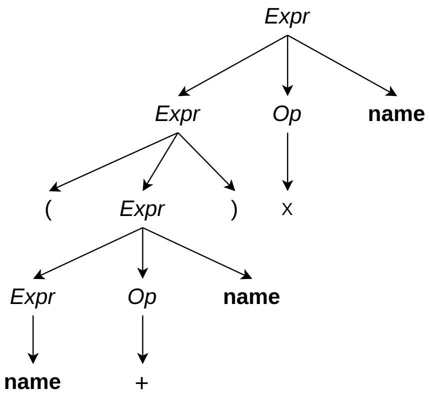
Example of a Context-Free Grammar (CFG)

CFG

$$
\begin{aligned}
Expr &\rightarrow (Expr) \\
&\mid Expr\ Op\ \textbf{name} \\
&\mid \textbf{name} \\
Op &\rightarrow + \mid - \mid \times \mid \div
\end{aligned}
$$

Deriving $(a + b) \times c$

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



Parse Tree

Parse Tree

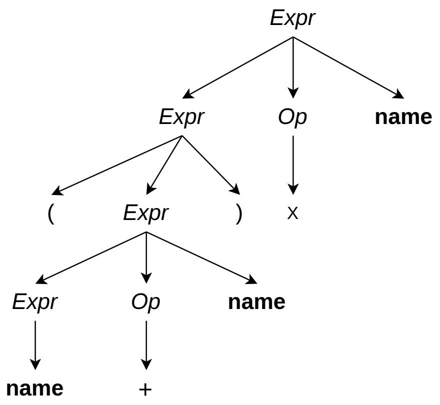
- A parse tree is a graphical representation of a derivation
  - Root is labeled with the start symbol S
  - Each internal node is a nonterminal, and represents the application of a production
  - If A is a nonterminal labeling some internal node and $X_1, X_2, \ldots, X_n$ are the labels of the children of A from left to right, then there must be a production $A \rightarrow X_1X_2 \ldots X_n$ in the grammar
  - 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, and **only 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?
- A leftmost derivation rewrites the leftmost nonterminal at each step, denoted by $\alpha \underset{\text{lm}}{\Longrightarrow} \beta$
  - Every leftmost derivation can be written as $wAy \underset{\text{lm}}{\Longrightarrow} w\delta y$, where $w \in T^*$
- Rightmost (or canonical) derivation rewrites the rightmost nonterminal at each step, denoted by $\alpha \underset{\text{rm}}{\Longrightarrow} \beta$

Leftmost Derivation

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



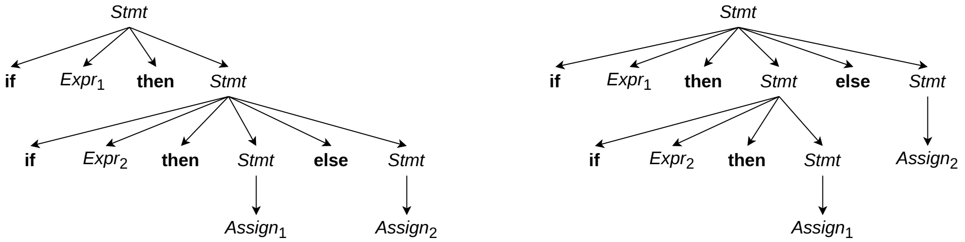
Parse Tree

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

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

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



Dealing with Ambiguous Grammars

- Compilers use parse trees to interpret the meaning of the expressions during later stages
- Ambiguous grammars are problematic for compilers since multiple parse trees can give rise to multiple interpretations
- Ways to fix an ambiguous grammar
  (i) Transform the grammar to remove the ambiguity
  (ii) Include rules to disambiguate during derivations (e.g., associativity and precedence)

Fixing the Ambiguous Dangling-Else Grammar

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

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

$$\textbf{if } \mathit{Expr}_1 \textbf{ then if } \mathit{Expr}_2 \textbf{ then } \mathit{Assign}_1 \textbf{ else } \mathit{Assign}_2$$

$$\Downarrow$$

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

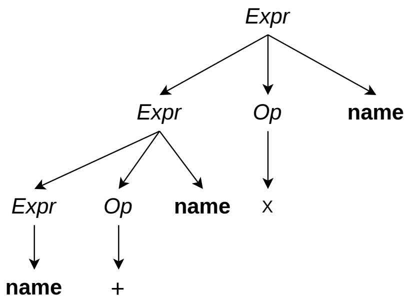
Interpreting the Meaning of Programs

CFG

$$
\begin{aligned}
Expr &\rightarrow (Expr) \\
&\mid Expr\ Op\ \textbf{name} \\
&\mid \textbf{name} \\
Op &\rightarrow + \mid - \mid \times \mid \div
\end{aligned}
$$

$a + b \times c$

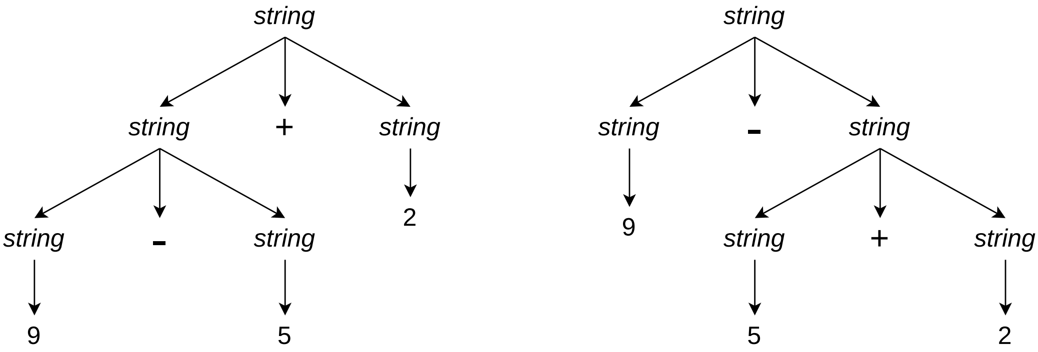
$$
\begin{aligned}
Expr &\rightarrow Expr\ Op\ \textbf{name} \\
&\rightarrow Expr \times \textbf{name} \\
&\rightarrow Expr\ Op\ \textbf{name} \times \textbf{name} \\
&\rightarrow Expr + \textbf{name} \times \textbf{name} \\
&\rightarrow \textbf{name} + \textbf{name} \times \textbf{name}
\end{aligned}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



Associativity

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

9-5+2

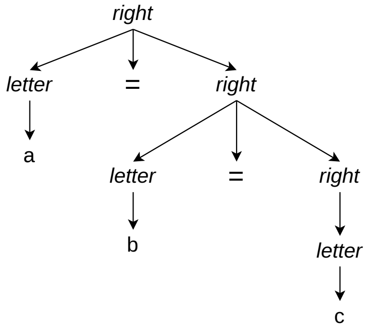


Associativity

- If an operand has operators on both sides, the **side** on which the operator takes this operand is the associativity of that operator
  - For example, +, −, ×, and / are left-associative and ˆ and = are right-associative
- Grammar to generate strings with right-associative operators

$$\begin{aligned} \textit{right} &\rightarrow \textit{letter} = \textit{right} \mid \textit{letter} \\ \textit{letter} &\rightarrow \mathrm{a} \mid \mathrm{b} \mid \ldots \mid \mathrm{z} \end{aligned}$$

a=b=c



Encode Precedence into the Grammar

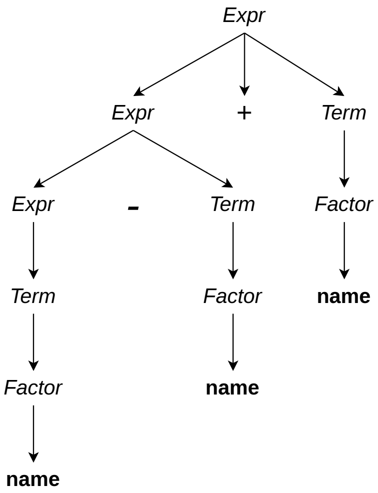
$$
\begin{aligned}
\mathit{Start} &\longrightarrow \mathit{Expr} \\
\mathit{Expr} &\longrightarrow \mathit{Expr} + \mathit{Term} \mid \mathit{Expr} - \mathit{Term} \mid \mathit{Term} \\
\mathit{Term} &\longrightarrow \mathit{Term} \times \mathit{Factor} \mid \mathit{Term} \div \mathit{Factor} \mid \mathit{Factor} \\
\mathit{Factor} &\longrightarrow (\mathit{Expr}) \mid \textbf{num} \mid \textbf{name}
\end{aligned}
$$

↓ priority

Corresponding Parse Tree

$a - b + c$

$$
\begin{aligned}
\mathit{Start} &\rightarrow \mathit{Expr} \\
&\rightarrow \mathit{Expr} + \mathit{Term} \\
&\rightarrow \mathit{Expr} + \mathit{Factor} \\
&\rightarrow \mathit{Expr} + \textbf{name} \\
&\rightarrow \mathit{Expr} - \mathit{Term} + \textbf{name} \\
&\rightarrow \mathit{Expr} - \mathit{Factor} + \textbf{name} \\
&\rightarrow \mathit{Expr} - \textbf{name} + \textbf{name} \\
&\rightarrow \mathit{Term} - \textbf{name} + \textbf{name} \\
&\rightarrow \mathit{Factor} - \textbf{name} + \textbf{name} \\
&\rightarrow \textbf{name} - \textbf{name} + \textbf{name}
\end{aligned}
$$



Types of Parsers

Top-down

Starts with the root and grows the parse tree toward the leaves (e.g., LL parsers)

Bottom-up

Starts with the leaves and grows the parse tree toward the root (e.g., LR parsers)

Universal

More general algorithms, but inefficient to use in production compilers (e.g., Earley's parser)

Programming Errors

Common source of programming errors

- Lexical errors, e.g., illegal characters and missing quotes around strings
  - The scanner cannot deal with most errors, e.g., it will mark misspelled keywords as IDs
- Syntactic errors, e.g., misspelled keywords, misplaced semicolons, or extra or missing braces
- Semantic errors, e.g., type mismatches between operators and operands and undeclared variables
- Logical errors

Goals in Error Handling

(i) Report errors accurately

(ii) Recover from the error and detect subsequent errors

(iii) Add minimal overhead to the compilation of correct programs

Report the source location where the error is detected, chances are the actual error location is close by

Error Recovery Strategies in the Parser

Panic-mode recovery

- Parser discards input symbols until a synchronizing token is found, restarts processing from the synchronizing token
- Synchronizing tokens are usually delimiters (e.g., ; or })

Phrase-level recovery

- Perform local correction on the remaining input (e.g., replace comma by semicolon)
- Can go into an infinite loop because of wrong correction, or the error may have occurred before it is detected

Handling Errors in the Parser

Error productions

- Augment the grammar with productions that generate erroneous constructs
- Works only for common mistakes and 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 (i.e., insertions, deletions, and changes) of tokens required to transform x into y is as small as possible

Limitations of Syntax Analysis

Cannot detect many kinds of programming errors

- 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. Sections 2.2, 4.1–4.3, 2nd edition, Pearson Education.
- K. Cooper and L. Torczon. Engineering a Compiler. Sections 3.1–3.2, 2nd edition, Morgan Kaufmann.