Chapter III.
Syntactic Analysis

a.hettab@centre-univ-mila.dz

Introduction

- **Syntactic analysis** has several objectives:
  - determine whether the sequence of **tokens** (lexical units) **conforms** to the **grammar** defining the **source language** ;
  - To detect syntax **errors ;**
  - To define a hierarchical structure (**a syntax tree**).
- **Algebraic or context-free grammars** are powerful enough to describe the main part of the syntax of most programming languages.
- In this chapter, we will review **context-free grammars**.

Context-free grammars

- An **algebraic grammar** , or **context-free grammar**, is a formal grammar in which each production rule is of the form:

$$A \rightarrow \alpha$$

where **A** is a non-terminal symbol and **α** is a string composed of **terminals** and/or **non-terminals** .

- The term " **context-free** " comes from the fact that a non-terminal **A** can be replaced by **α** , regardless of the context in which it appears.

Context-free grammars

- A context-free grammar G is a quadruple **$<N,T,P,S>$** where:
- **N** : a finite non-empty set of symbols called **non-terminal symbols** ,
- **T** : a finite set of symbols called **terminal symbols** ,
- **S** : $S \in N$, the **start symbol** (or **axiom**) of the grammar,
- **P** : the set of **production rules** ,

Each production rule is of the form:

$$A \rightarrow \alpha$$

Where $A \in N$ and $\alpha \in (N \cup T)^*$

Example

- The G grammar describing arithmetic expressions:
- G = <{S}, {+,-,*,/,(,), a,b,c },P,S>
- P:

$$S \rightarrow S + S$$
$$S \rightarrow S - S$$
$$S \rightarrow S * S$$
$$S \rightarrow S / S$$
$$S \rightarrow (S)$$
$$S \rightarrow a$$
$$S \rightarrow b$$
$$S \rightarrow c$$

- Or equivalently (using alternatives):

P: $S \rightarrow S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid a \mid b \mid c$

Derivations and Parse Trees

- There are **two ways** to describe the membership of a sentence in the language generated from a given grammar:
- The first step consists of listing a sequence of rule applications, also known as a **derivation sequence**.
- The second represents this list in the form of a tree, called a **derivation tree**.

Derivation

- **Derivation :** The process by which a grammar defines a language is called ***derivation*** :
  - Let $\boldsymbol{G = (N, T, P, S)}$ be a context-free grammar,
  - $\boldsymbol{A \in N}$ a non-terminal symbol and $\boldsymbol{\gamma \in (N \cup T)^*}$ a sequence of symbols, such as exist in $\boldsymbol{P}$ a production $\boldsymbol{A \rightarrow \gamma}$.
  - For any sequences of symbols $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$, we say that $\boldsymbol{\alpha A \beta}$ derives $\boldsymbol{\alpha \gamma \beta}$ in one step, which is written as:

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

Sequence of derivations

- **Sequence of derivations:**
  - If $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_n$ we say that α_0 derives α_n in ***n* steps** , and we write:

$$\alpha_0 \overset{n}{\Rightarrow} \alpha_n .$$

  - If α derives β in any number, possibly zero, of steps, we simply say that α derives β and we write:

$$\alpha \overset{*}{\Rightarrow} \beta.$$

  - If α derives β in any non-zero number of steps we simply say that α derives β and we write:

$$\alpha \overset{+}{\Rightarrow} \beta.$$

Example

- Let the grammar G_1 be:

  EXPRESSION → EXPRESSION "+" TERM | TERM

  TERM → TERM "*" FACTOR | FACTOR

  FACTOR → number | identifier | "("EXPRESSION")"

- The string **w** = **number " * " identifier " + " number** derives from **EXPRESSION** as follows:

- **EXPRESSION** ⇒ EXPRESSION "+" TERM ⇒ TERM "+" TERM ⇒ TERM "*" FACTOR "+" TERM ⇒ FACTOR "*" FACTOR "+" TERM ⇒ number "*" FACTOR "+" TERM => number "*" identifier "+" TERM ⇒ number "*" identifier "+" FACTOR ⇒ **number "*" identifier "+" number**

Leftmost Derivation and Rightmost Derivation

- **Leftmost derivation** is entirely composed of one-step derivations in which, at each step, the **leftmost non-terminal** is rewritten.
- **Rightmost derivation** is entirely composed of one-step derivations in which, at each step, the **rightmost non-terminal** is rewritten.

Example

- Consider the following grammar:

G = <{S,A}, { a,b },P,S>

P: $S \rightarrow aAS \mid a$

$A \rightarrow SbA \mid SS \mid ba$

- The string **aabbaa** derives from the start symbol **S** using :

  - **The leftmost derivation:**

$S \Rightarrow aAS \Rightarrow aSbAS \Rightarrow aabAS \Rightarrow aabbaS \Rightarrow aabbaa$

  - **The rightmost derivation:**

$S \Rightarrow aAS \Rightarrow aAa \Rightarrow aSbAa \Rightarrow aSbbaa \Rightarrow aabbaa$

language generated by a grammar

- Let $G = (N, T, P, S)$ be a context-free grammar;
- The **language generated by G** is the set of all strings of terminal symbols that can be derived from **S** :

$$L (G) = \{ w \in T^* \mid S \overset{*}{\Rightarrow} w \}$$

  - If a string $v \in L (G)$ we say that v is a ***sentence*** of G .
  - More generally, if $\alpha \in (T \cup N)^*$ is such that $S \overset{*}{\Rightarrow} \alpha$ then we say that α is a ***sentential*** of G .
  - A s***entential*** form whose symbols are all terminals is called a ***sentence***.

Example

Example 1:

- Let the grammar **G1=<N,T,P,S>** be defined as follows :

  N = {S} ,T = { a,b } ,

  P = {S → aSb ; S → ε }

  $$L(G) = \{ a^n b^n \mid n \geq 0 \}$$

Example 2:

Let the grammar **G2=<N,T,P,S>** be defined as follows :

N = { S } ,T = { a, b }

P = { **S → aaS | bbS | ε** }

$$L(G) = \{(aa \mid bb)^*\}$$

Derivation tree

- Let **w** be a string of terminal symbols from the language **L(G)**; there therefore exists a derivation such that $S \overset{*}{\Rightarrow} w$. This derivation can be represented graphically by a tree, called ***a derivation tree*** , defined as follows:
- **the root of the tree** is the start symbol **S** ;
- The **internal** nodes are labeled with non-terminal symbols;
- If an internal node **e** is labeled with the symbol **A**, and $\mathbf{A \rightarrow A_1A_2\ldots A_k}$ is a production of the grammar, then the **children** of **e** are nodes labeled, from left to right, with $\mathbf{A_1, A_2, \ldots, A_k}$;
- The **leaves** are labeled with **terminal symbols**.

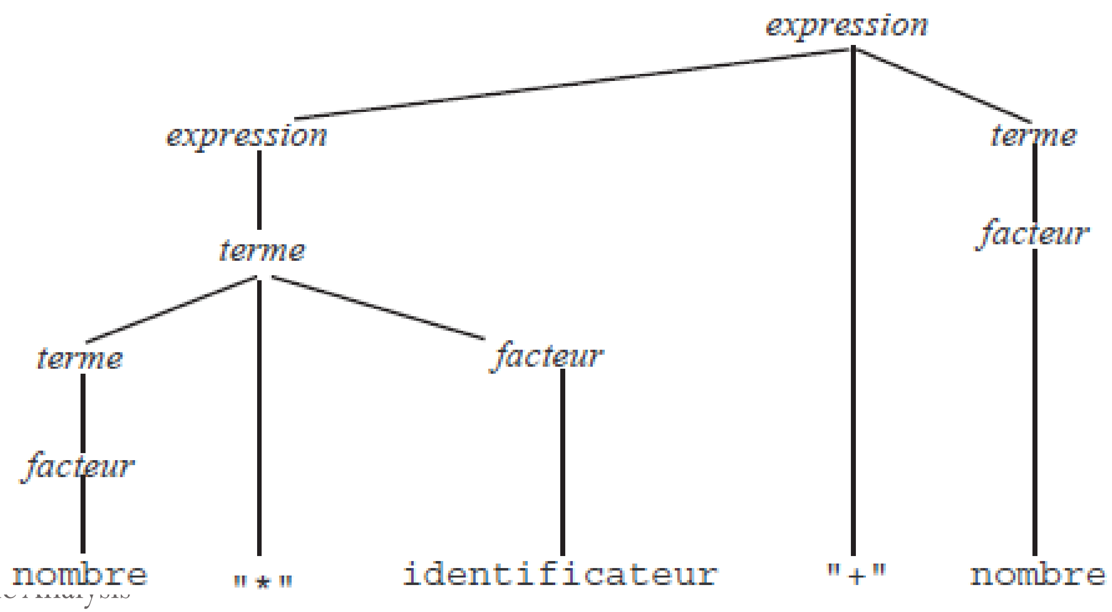
Example

The derivation tree for the string: **number " * " identifier " + " number** using the following grammar rules:

EXPRESSION → EXPRESSION "+" TERM | TERM

TERM → TERM "*" FACTOR | FACTOR

FACTOR → number | identifier | "("EXPRESSION")"



Ambiguous grammar

- A **grammar is ambiguous** if there exist **two** or **more** different leftmost (or rightmost) derivations for the same string of terminal symbols.
- A grammar is also said to be ambiguous if **at least one string** generated by the grammar has **two or more distinct derivation trees**.
- It should be noted that the **order of derivations** (leftmost or rightmost) **cannot be seen directly** from the derivation tree.

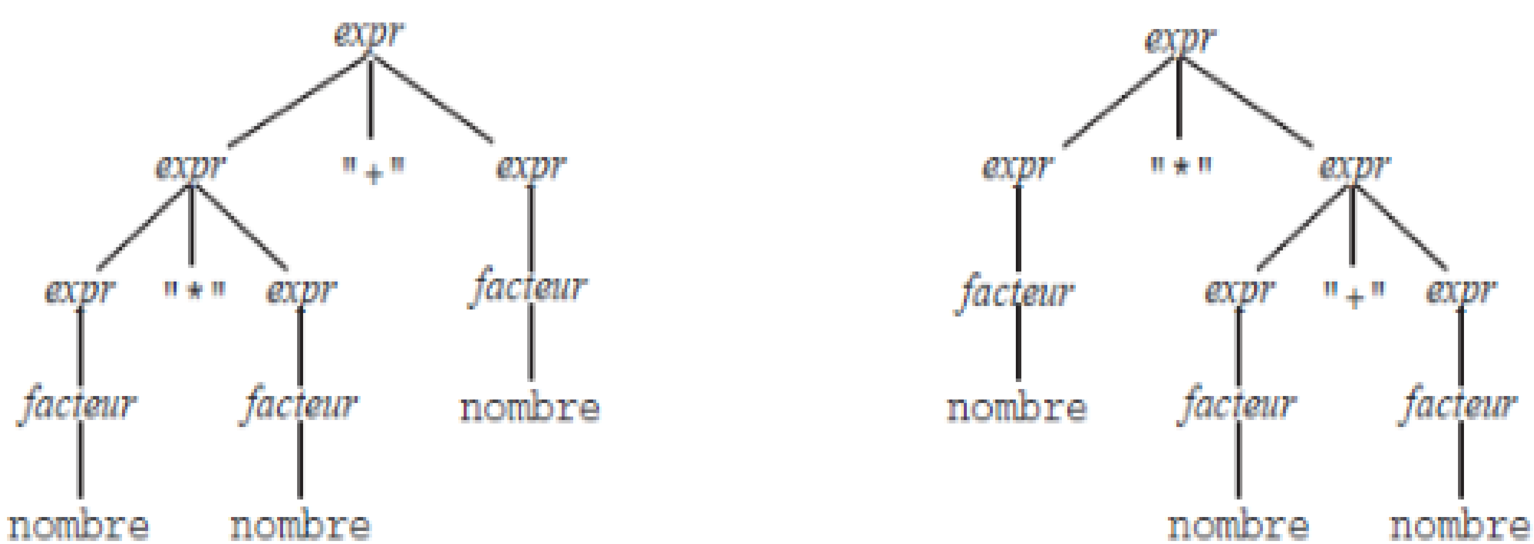
Example

- Let the grammar G_1 be :

  exp → *exp* **"+"** *exp* **|** *exp* **"*"** *exp* **|** *factor*

  factor → **number | ident | "("** *exp* **")"**

- This grammar is **ambiguous** because it has **two distinct derivation trees for the string : number "*" number "+" number**



Top-down and bottom-up parsing

- Most parsing methods fall into **two classes** :
- **Top-down parsing methods**: These are the **most popular** because the corresponding parsers are easy to implement manually **;**
- **Bottom-up parsing methods**: These are the most powerful and are used by tools that automatically generate a parser from a grammar (or example: **yacc** **)**.

Top-down analysis

- In **top-down parsing**:
  - The parse tree is built starting from the **root** and progressing down to the **leaves**;
  - The analysis relies on a **leftmost derivation** of the string, meaning that the **leftmost non-terminal** is always replaced first.

Top-down analysis

Example :

Let the grammar **G=<N,T,P,S>** be defined as follows:

N = { **E,T,F** } ,T = { **+,*,),(,i** } ,

P = {

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T \times F \mid F$$

$$F \rightarrow (E) \mid i$$

}

- The following shows the **derivation of the string (i+i)*i** starting from the **root of the parse tree**.

Top-down analysis

Example (continued) :

E	(Step 0)
$E \Rightarrow T$	(Step 1)
$\Rightarrow T * F$	(Step 2)
$\Rightarrow F * F$	(Step 3)
$\Rightarrow (E) * F$	(Step 4)
$\Rightarrow (E+T) * F$	(Step 5)
$\Rightarrow (T+T) * F$	(Step 6)
$\Rightarrow (F+T) * F$	(Step 7)
$\Rightarrow (i+T) * F$	(Step 8)
$\Rightarrow (i+F) * F$	(Step 9)
$\Rightarrow (i+i) * F$	(Step 10)
$\Rightarrow$ **(i+i) * i**	(Step 11)

Bottom-up analysis

- In **bottom-up parsing**, the process is the opposite :
- The parse tree is built **upward**, starting from the leaves and moving toward the root, using **reverse derivations**;
- The analysis is based on a **rightmost derivation** of the string, meaning that the **rightmost non-terminal** is always replaced first.

Bottom-up analysis

Example :

Let the grammar **G=<N,T,P,S>** be defined as follows:

N = { **E,T,F** } , T = { **+,*,),(,i** } ,

P = {

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T \times F \mid F$$

$$F \rightarrow (E) \mid i$$

}

- Below we show the **reverse (bottom-up) derivation** of the string **(i + i) * i** starting from the **leaves of the parse tree**.

Bottom-up analysis

Example (continued) :

(i+i)*i (Step 0)

⇐(F+i)*i (Step 1)

⇐(T+i)*i (Step 2)

⇐(E+i)*i (Step 3)

⇐(E+F)*i (Step 4)

⇐(E+T)*i (Step 5)

⇐(E)*i (Step 6)

⇐F*i (Step 7)

⇐T*i (Step 8)

⇐T*F (Step 9)

⇐T (Step 10)

⇐ **E** (Step 11)

Handling Syntactic Errors

- In general, most error handling is performed by the **parser**, since a large proportion of errors are **syntactic** in nature**.**
- An **error handler** in a **Parser** must:
  - Clearly indicate the **presence of errors**, specifying both their **location** and **cause:**
    - ***Location***: for example, display the erroneous line with a marker pointing to the error position.
    - ***Cause***: examples include *an extra closing parenthesis* or *a missing semicolon*.
  - deal with every error quickly
  - **Avoid significantly slowing down** the compilation of a correct program.

error recovery strategies

- There are several error recovery techniques. In particular:
  - **Panic mode,**
  - **Local correction**
  - **Error productions**
  - **Global correction**

Panic Mode Error Recovery

- The **simplest technique to implement**, and the one applicable to most parsing methods, is the **panic mode.**
- During the parsing process, a set of **synchronizing tokens (symbols)** is continuously maintained.
  Examples include:
  - The symbol **End if** in a conditional structure,
  - The **semicolon (;)** in an assignment statement.
- When an error is detected, the parser **skips (ignores)** all tokens provided by the scanner **until a synchronizing symbol** is encountered.

Panic Mode Error Recovery

- **Benefits**
  - Simple to implement.
  - Does not result in infinite loops.
- **Disadvantages**
  - A considerable portion of the program may be **skipped** without verifying its validity.
- **Conclusion**
  - **Panic mode recovery** is very suitable when **multiple errors within the same statement** are rare.

Error recovery in local correction mode

- When an error is detected, **local corrections** are performed by **modifying the prefix of the remaining source text** of the current statement.

Examples include:

  - Replacing a comma with a semicolon
  - Removing an extra semicolon
  - Inserting a missing semicolon

- The chosen replacement must **not lead to an infinite loop** (Example: if something is always inserted before the current symbol in the input string).
- **Disadvantage** :

It is **difficult to handle situations** where the actual error occurred **before the point of detection**.

Error recovery in error production mode

- The use of **error production rules** is an **error recovery technique**.
- It consists of **adding productions that include the notion of an error** to the grammar of the language.
- This technique is notably **used by Yacc** .

Error recovery in error production mode

Example : Let the grammar **G=<N,T,P,S>** be defined as follows :

N = { *Expr* , *Op* } ,T = {+,-,*,/,id} ,

P = { :

Expr $\rightarrow$ ***Expr Op Expr*** **| (** ***Expr*** **) | -** ***Expr*** **| id**

Op $\rightarrow$ **+ | - | * | /** }

To detect a **misplaced closing parenthesis**, it is sufficient to add the following rules:

Error $\rightarrow$ ***Op*** **)** **|** ***Expr*** **)**

•**Disadvantage** : A **large number of rules** is required to handle a **wide variety of errors**.

Global correction mode error recovery

- When an **erroneous input string** (instruction X) is encountered, the parser constructs a **parse tree** for the **closest correct instruction** Y**.**

This may allow the parser to make **minimal modifications** to the source code by **replacing X with Y**.

- **Disadvantage:**

This method is **too expensive in terms of memory and time**.
So far, it has **not yet been implemented in practice**.