Chapter IV.
Top-Down Syntactic Analysis

a.hettab@centre-univ-mila.dz

Introduction

- In **top-down parsing**, the syntax tree is constructed starting from the **root** and then proceeding **downward** through the tree.
- **Top-down methods** are **simple to implement** , but the class of **context-free grammars** for which such parsers can be built is **not very large**.
- We will study two versions of the **predictive parsing (LL)** method:
  - **LL(1) parsing.**
  - **recursive descent analysis (parsing).**

LL Syntax Analysis

- **LL analysis** is a **top-down parsing** applicable for certain context-free grammars, known as **LL grammars** .
- **LL analysis** analyzes an input word **from Left to right** (this is **the first L of L L)**, and constructs a **Leftmost derivation** of the sentence (it is **the second L of L L**).

LL Syntactic Analysis (1)

- An **LL analysis** is called an **LL(1) analysis** when it performs **a single pass** over the input string.
- An **LL analysis** is called an **LL(k) analysis** when it performs **k passes** over the input string.
- In most cases, the value of **k** is equal to **1**, because when **k > 1**, the analysis becomes less practical.

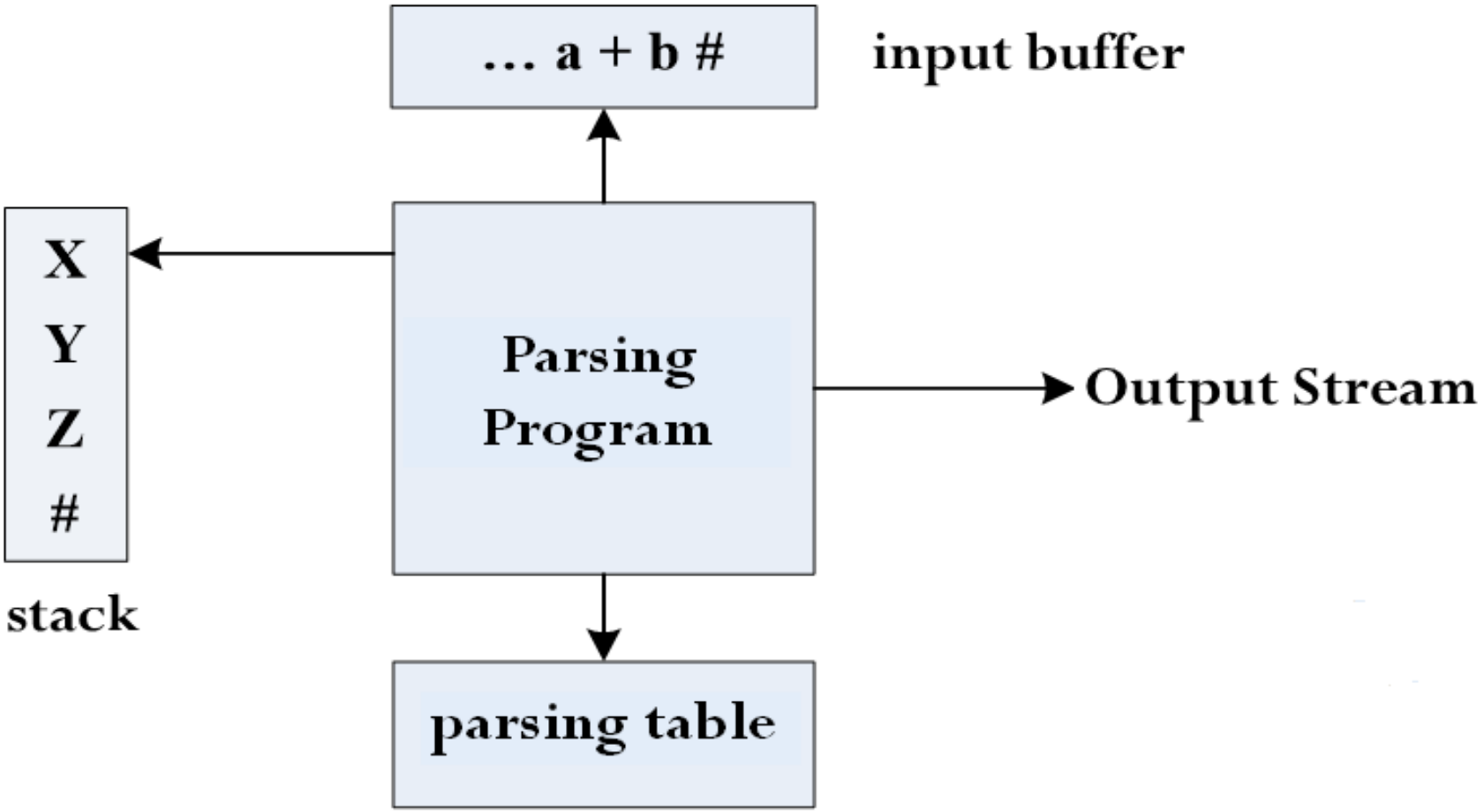
  For this reason, we will use **only LL(1) analysis** in the remainder of this course.

Architecture of an LL(1) analyzer

The **LL(1) parser** is composed of the following components:

- An **input buffer**, which contains the string to be analyzed and provides two operations: reading the **current character** and moving to the **next character**.
- A **stack**, used to hold the **terminals** and **non-terminals** of the grammar that remain to be analyzed.
- A **parsing table**, which indicates **which production rule** to use (if any), based on the **symbol at the top of the stack** and the **next input character**.

Architecture of an LL(1) analyzer



Construction of the LL(1) analysis table

Let **G = <N, T, S, P>** be a **context-free grammar**, where:

- **N** denotes the set of **non-terminal symbols**,
- **T** is the set of **terminal symbols**,
- **P** is the set of **production rules**, and
- **S** is the **axiom** of the grammar

To construct the **parsing table**, we introduce two important functions: **First** and **Follow**.

functions: *First* and *Follow*

First

First Function:

- For any expression $\alpha \in (T \cup N)^+$, the $First(\alpha)$ set is defined as the set of terminal symbols that can appear at the beginning of a string derived from α .
- More formally:
  - $First(\alpha) = \{ a \in T \mid \exists \beta \in (T \cup N)^* \text{ such that } \alpha \Rightarrow^* a\beta \}$ → That is, all terminals **a** that can begin a string derived from α.
  - If $\alpha \Rightarrow^* \varepsilon$ (α can derive the empty string), then $\varepsilon \in First(\alpha)$.
  - If $\alpha = \varepsilon$, then $First(\alpha) = \emptyset$ (the empty set)

Computation of the First Set

First(X)

i) IF **X** is a terminal THEN **First (X)=X** ;

ii) IF **X → ε** THEN add **ε to first (X)** ;

iii) IF **X → $Y_1 Y_2 \dots Y_n$** SO

- IF **ε ∉ First (Y1)**

THEN we only add **first(Y_1) to first(X)** ;

- IF **ε ∈ first (Y_1),…, first (Y_{i-1})** with **$2 \leq i \leq n$**

THEN add **first (Y_1),…, first(Y_{i-1}), first(Y_i) ε excluded to first(X)**

- IF **ε ∈ first (Y_1), first (Y_2),…, first (Y_n)**

THEN add **ε to first (X)**

Noticed :

- These rules are applied until no terminals or **ε** can be added to the **first sets** .

functions: First and Follow Follow

Follow Function:

- For any expression $\alpha \in (T \cup N)+$, the *Follow(α)* set is defined as the set of **terminal symbols** that can **immediately follow** a word derived from α.
- More formally:
  - $Follow(\alpha) = \{ a \in First(\gamma) \mid \exists \beta, \gamma \in (T \cup N)^* \text{ such that } S \Rightarrow^* \beta\alpha\gamma \} \rightarrow$ That is, all terminals **a** that can appear immediately after α in some sentential form derived from the start symbol **S**.
  - $If\ \alpha = \varepsilon\ then\ Follow(\alpha) = \Phi$.
  - The special end-of-input marker '#' is **added** to $Follow(\alpha)$ for every $\alpha \in (T \cup N)+$ such that there exists $\beta \in (T \cup N)^*$ with $S \Rightarrow^* \beta\alpha$, in order to indicate the end of the input string.

Computation of the Follow Set

FOLLOW(X)

i) Put # (which is the end marker of the entry to be analyzed) in **FOLLOW(S)** where **S** is the **axiom** of the grammar;

ii) If there is a production $A \rightarrow \alpha X \beta$

THEN add **FIRST(β) except ε to FOLLOW(X)** ;

iii) If there is a production $A \rightarrow \alpha X$ or a production $A \rightarrow \alpha X \beta$ with $\varepsilon \in$ **FIRST(β)**

THEN add **FOLLOW(A) to FOLLOW(X)** ;

Noticed :

- These rules are applied until no terminals or # can be added to the **FOLLOW sets** .

functions: *First* and *Follow* *Examples*

Example 1:

- Calculate the **First** and **Follow sets** for the non-terminals of the grammar whose productions are given below:

  S → aSBA | ε

  A → aSb | b

  B → bB | ε

	First	Follow
S	a ε	# a b
A	a b	# a b
B	b ε	a b

functions: *First* and *Follow* *Examples*

Example 2:

- Calculate the **First** and **Follow sets** for the non-terminals of the grammar whose productions are given below:

  S → ABSb | ε

  A → aBb | b

  B → bB | cS | ε

	First	Following
S	a b ε	# a b
A	a b	b c a
B	b c ε	a b

Filling the analysis table

- The **analysis table (parsing table)** is a **two-dimensional matrix** , whose **rows** are indexed by ***Non-terminals*** and ***the*** **columns** by **Terminals** .
- **Given** : A context-free grammar **G** .
- **Result:** An analysis table **M** for **G** ;
- The **row indices** of **M** correspond to the **non-terminals** of the grammar;
- The **column indices** correspond to the **terminals** of the grammar, plus the special character # (where # is the end-of-input marker).

Filling the analysis table Algorithm

For each rule of the form $X \rightarrow \alpha$ **Do**

    For each $a \in First(\alpha)$ **Do**

        Add $X \rightarrow \alpha$ to the index box $M[X, a]$

        If $\varepsilon \in First(\alpha)$ **Then**

            For all $b \in Follow(X)$ **Do**

                Add $X \rightarrow \alpha$ to the index box $M[X, b]$

            End for

        End If

    End for

End for

LL(1) Grammars?

- A **grammar** is **LL(1)** when its **parsing table** does not contain **multiple production rules** in the **same cell**. In this case, the parsing table is said to be **single-defined**. Otherwise, it is said to be **multiple-defined.**
- An **LL(1) grammar** allows for **deterministic top-down parsing**. During any derivation, there will be **at most one possible way** to replace a non-terminal by one of its right-hand sides, depending on the **current input symbol** being analyzed.

LL(1) Grammars?

- **Without** constructing the **parsing table**, we can determine whether a grammar is **LL(1)** or not if it satisfies the following properties:
- If there exist two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$ (with $\alpha \neq \beta$) in an **LL(1)** grammar, then :
  - There is **no terminal** *a* such that both α and β generate strings beginning with ***a***;
  - **At most one** of α and β can generate the empty string ε;
  - If $\beta \Rightarrow^* \varepsilon$, then α cannot generate a string whose **first terminal** belongs to **Follow(A)**.

LL(1) Grammars?

Noticed :

- The three previous conditions can be summarized by the following formula.

✓ A grammar **G=<N,T,S,P>** is **LL(1)** if and only if for any pair of rules **$A \rightarrow \alpha \mid \beta$** we have:

$$\text{First}(\alpha \,.\text{Follow}(A)) \cap \text{First}(\beta \,.\text{Follow}(A)) = \varnothing$$

LL(1) analysis algorithm

Initially The stack contains # with the **start symbol (Axiom)** of the grammar on top;

repeat Let **X** be the symbol at **the top of the stack** and let **a** be the current input symbol

If X is a **non-terminal** **then**

If M[X, a]= X → Y1 ….Yn then

Pop X from the stack and **Push Yn then Yn -1**

then….. then Y1 in the stack

Else /*empty box in the table*/

Write (' **ERROR** ')

End if

Else /*X is a terminal*/

(1) (2)

Analysis algorithm

(1) (2)

```
If X = # then
    If a = # then
        Write (' ACCEPT ')
    Else
        Write (' Error ')
    End if
Else /*X is a terminal ≠ #*/
    If X = a then
        Pop X from the stack and advance the input to the next symbol.
    Else /*X is a terminal ≠ a*/
        Write (' Error ')
    End if
End if
End if
End Repeat
```

Example of LL(1) analysis

- Consider the grammar **G = <{E,E',T,T',F},{+,*,(,),i},P,E>** :

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow * FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid i$$

✓ First and follow Sets:

	First	FOLLOWING
E	(i	#)
E'	+ ε	#)
T	(i	+ #)
T'	* ε	+ #)
F	(i	* + #)

Example of LL(1) analysis

- Analysis table for the previous grammar:

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow * FT' \mid \varepsilon$
$F \rightarrow (E) \mid i$

	+	*	(	)	i	#
E			$E \rightarrow TE'$		$E \rightarrow TE'$	
E'	$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$		$E' \rightarrow \varepsilon$
T			$T \rightarrow FT'$		$T \rightarrow FT'$	
T'	$T' \rightarrow \varepsilon$	$T' \rightarrow * FT'$		$T' \rightarrow \varepsilon$		$T' \rightarrow \varepsilon$
F			$F \rightarrow (E)$		$F \rightarrow i$	

The grammar is LL(1)

Example of LL(1) analysis

- Parsing (Analysis) of the string **i + i * i #**

Stack contents	Remaining to be analyzed	Action
#E	i + i * i #	Pop (E) and Push (E'T)
#E'T	i + i * i #	Pop(T) and Push (T'F)
#E'T'F	i + i * i #	Pop(F) and Push (i)
# E'T'i	i + i * i #	Move forward
#E'T'	+ i * i #	Pop(T')
#E'	+ i * i #	Pop(E') and Push (E'T+)
#E'T+	+ i * i #	Move forward
#E'T	i* i #	Pop(T) and Push (T'F)
#E'T'F	i * i #	Pop(F) and Push (i)
# E'T'i	i * i #	Move forward
#E'T'	* i #	Pop(T') and Push (T'F*)
#E'T'F*	* i #	Move forward
#E'T'F	i #	Pop(F) and Push (i)
# E'T'i	i #	Move forward
#E'T'	#	Pop(T')
#E'	#	Pop(E')
#	#	"Accepted string"

Transformation of a grammar

- **Theorem :** A grammar that is **ambiguous**, **left-recursive**, or **not left-factored** is **not an LL(1) grammar**.
- Unfortunately, there is **no general method** to transform a grammar into an LL(1) grammar:
  - An **ambiguous grammar** is a poorly designed grammar. It must be **redesigned**, taking into account **operator precedence** and associating the **first else** with the **last if**, etc.
  - **Eliminate left recursion** if necessary.
  - **Left-factor** the production rules if necessary.

Left factoring of a grammar

- **Non–left-factored productions** of a grammar of the form :

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma \text{ where } \alpha \neq \varepsilon$$

- are replaced by the following **left-factored productions** :

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

Left factoring of a grammar

- **Example** :

Let the grammar **G1** have the following production rules:

S → aC | abD | aBc

After left-factoring, we obtain an equivalent grammar **G'_1** whose production rules are:

S → aS'

S' → C | bD | Bc

Elimination of left recursion in a grammar

- A grammar is said to be **left-recursive** if it contains a non-terminal **A** such that:

$$A \Rightarrow^{*} A\,\alpha$$

  - Where α is any string.
- **Elimination of immediate left recursion:**
- The rules in the following form:

$$A \rightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta n$$

  - where each β does **not** begin with **A**.
  - are replaced by these productions :

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

Elimination of left recursion in a grammar

- **Example:**

Let the grammar G1 have the following production rules:

E → E+T | T

T → T*F | F

F → (E) | id

After eliminating left recursion, we obtain a grammar G'1 equivalent to G1, whose production rules are:

E → TE'

E' → +TE' | ε

T → FT'

T' → *FT' | ε

F → (E) | id

Elimination of indirect left recursion

Grammar without cycles (i.e. no derivation $A \Rightarrow^{+} A$) and without empty productions ($A \rightarrow \varepsilon$)

Order the non-terminals $A_1, A_2, \ldots, A_n$;

For i:=1 **to** n **Do**

For j:=1 **to** i-1 **Do**

Replace each production of the form $A_i \rightarrow A_j \gamma$

By the productions $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \delta_k \gamma$

where the current productions for A_j are $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$

End for ;

Eliminate the immediate left recursion among the productions for A_i ;

End for ;

Elimination of indirect left recursion

- **Example** :
- Eliminate left recursion from the following grammar:

  $S \rightarrow Ab \mid b \mid Sb$

  $A \rightarrow a \mid AS \mid Sb$
- Order **S, A**
- **Step 1:** Eliminating immediate left recursion:

  $S \rightarrow AbS' \mid bS'$

  $S' \rightarrow bS' \mid \varepsilon$
- **Step 2:** Substitution for **A**

  $A \rightarrow a \mid AS \mid AbS'b \mid bS'b$
- Eliminating immediate left recursion:

  $A \rightarrow aA' \mid bS'bA'$

  $A' \rightarrow SA' \mid bS'bA' \mid \varepsilon$

Recursive-descent parsing

Recursive-descent parsing

- **Recursive descent** parsing is a **top-down** parsing method in which a set **of recursive procedures** is performed to process the input string.
- A **procedure** is associated with each **non-terminal** in a grammar.
- An **LL(1) recursive-descent parser** is divided into two parts:
- **(i) Parser construction**: a parsing procedure is written for each non-terminal
- **(ii) Word recognition**: a sequence of **recursive procedure** calls is used to recognize a word of the language.

Construction of the parser

- **Condition:** The grammar **must satisfy** the **LL(1) conditions**
- **Construction stages** :
  - We add the following production rule: **Z → S#** where **S** is the start symbol (axiom) of the grammar and **#** is the end-of-input marker;
  - Each **non-terminal** in the grammar corresponds to **a procedure** ;

Writing the procedures

- We use the variables **tc** and **ts** to denote, respectively, the **current symbol** of the input string being analyzed and its **next symbol**;
- The processing of the right-hand side of a production rule $A \rightarrow \alpha_i$ is done as follows:
- Each **terminal** on the right-hand side is compared with the **current input symbol**:
  - **If** they are **equal**, read the **next symbol**
  - **Otherwise**, report an **error**;
- Each **non-terminal** on the right-hand side corresponds to a **procedure call**.

Example of analysis

- Consider the grammar :

  G = <{S,A}, { a,b,c },S,P> as follows:

  S → a A b | ε

  A → c A | ab

- We add the following rule:

  Z → S #

- **FIRST** and **FOLLOW** sets:

	FIRST	FOLLOWING
S	a ε	#
A	c a	b

- The grammar **G** above satisfies the LL(1) conditions

Writing procedures

```
Procedure Z( )
Begin
S( );
 If tc = '#' then
    "Syntactically correct string"
 Else
    "Error"
 Fndif ;
END .
```

```
Procedure S( )
Begin
 If tc = 'a'   then
        tc = ts ;
        A( );
        If tc = 'b' then
            tc = ts
        Else
            "Error"
        Endif ;
    Endif ;
END .
```

Writing procedures

```
Procedure A( )
Begin
   If tc = 'c' then
      tc = ts ;
      A( );
   Else
      If tc = 'a' then
         tc = ts
         If tc = 'b' then
            tc = ts
         Else
            "Error"
         Endif ;
      Else
         "Error"
      Endif ;
   End if ;
END .
```

Example analysis (1)

- Channel analysis: **accabb #**

Stack Contents	Remaining to be analyzed	Action
Z	accabb #	**Call S**
ZS	accabb #	**Move forward; Call A**
ZSA	ccabb #	**Move forward; Call A**
ZSAA	cab b #	**Move forward; Call A**
ZSAAA	abb #	**Forward; forward; return A**
ZSAA	b#	**return A**
ZSA	b#	**return A**
ZS	b#	**move forward; return S**
Z	#	**"Correct Chain"**

Example analysis (2)

- Analysis of the chain: **acaabb #**

Stack Contents	Remaining to be analyzed	Action
Z	acaabb #	**Call S**
ZS	acaabb #	**Move forward; Call A**
ZSA	caabb #	**Move forward; Call A**
ZSAA	aabb #	**Move forward; "Error"**
ZSAA	abb #	**"Incorrect string"**