COMPILER LECTURES

COMPUTER SCIENCE

3RD CLASS

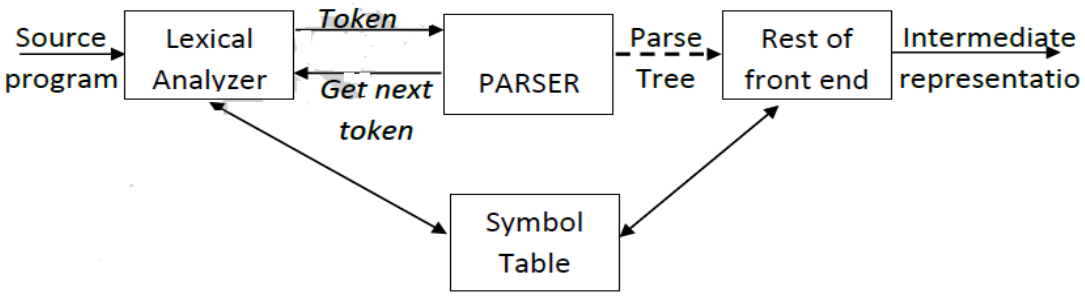
M.SC. SAMER AL-YASSIN

2017-2018

LECTURE 4

Syntax Analysis:

In our compiler model, the parser obtains a string of tokens from the lexical analyzer, and verifies that the string can be generated by the grammar for the source program. We expect the parser to report any syntax errors in an intelligible fashion. It should also recover from commonly occurring errors so that it can continue processing the remainder of its input.



Position of Parser in Compiler Model

Syntactic Errors:

Syntactic errors include ***misplaced semicolons*** or extra or ***missing braces***, that is '((" or ")." As another example, in C++, the appearance of a ***Case*** statement without an enclosing ***switch*** is a syntactic error.

Example:

1. **X:=A+*B;**
2. **For (I=1;I++;I<=10)**
3. **Cin<<X;**
4. **And more…**

Parser:

The parser has two functions:

1) It checks that the tokens appearing in its input, which is the output of the lexical analyzer, occur in patterns that are permitted by the specification for the source language.

2) It also imposes on the tokens a tree-like structure that is used by the subsequent phases of the compiler.

Example: if a source program contains the following expression:

A+/B

Then after lexical analysis this expression might appear to the syntax analyzer as the token sequence:

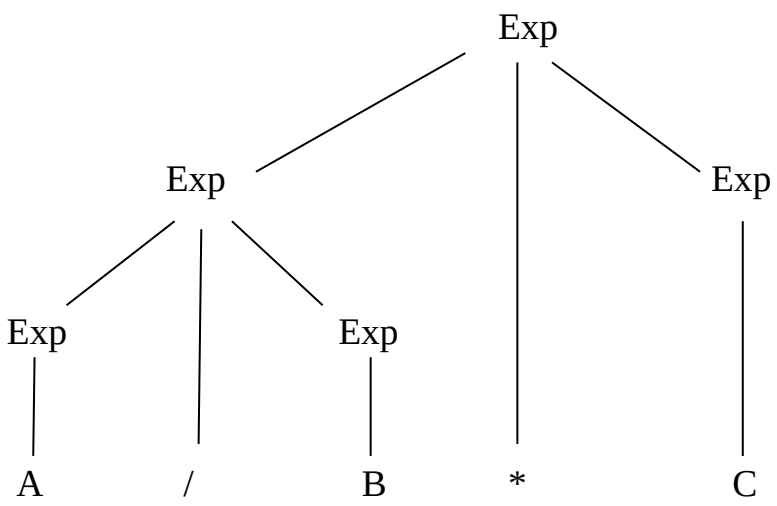
id1 +/id2

On seeing the/,the syntax analyzer should detect an error situation, because the presence of the two adjacent operators violates the formatting rules of a Pascal expression.

Example: identifying which parts of the token stream should be grouped together:

A/B*C

Parser Tree:



Context-free grammars (CFG)

Many programming language construct have an inherently recursive structure that can be defined by context-free grammars.

For ***example***, we might have a conditional statement defined by a rule such as: if $S_1$and S_2 are statements and E is an expression, then

"if E then S_1 else S_2" is a statement.

This form of conditional statement cannot be specified using the notation regular expressions.

Could also express as: ***stmt*** → **if** ***(expr) stmt*** **else** ***stmt***

Such as a role is called syntactic variables, ***stmt*** to denote the class of statements and ***expr*** the class of expression.

Components of context-free Grammars (CFG)

A context free grammar (CFG for short) consists of terminals, nonterminal, a start symbol, and productions.

1- **Terminals**: are the basic symbols from which string are formed. The word "***token***" is a synonym for "***terminal***" when we are talking about grammars for programming languages.
2- **Nonterminal**: are syntactic variables that denoted sets of strings.
3- One nonterminal is distinguished as the **start symbol**.
4- The set of **production** where each production **consist** of a **nonterminal called** the **left side** followed by an **arrow**, followed by a string of **nonterminal and/or terminal** called the **right side**.

Example: the grammar with the following production defines simple arithmetic expression.

Expr→ Expr Op Expr
Expr →(Expr)
Expr→-Expr
Expr→Id
Op→ +
Op→ -
Op→ *
Op→ /
Op→ ↑

In this grammar the terminal symbols are

Id + - * / ↑ ( )

The nonterminal symbols are expr and Op, and Expr is the start symbol.

The above grammar can be written by using short hands as:

E→EAE| (E)|-E/id
A→ + | - | * | / | ↑

Where E and A are non-terminal, with E the start symbol. The remaining symbols are terminal.

Derivations and parse tree

How does a context-free grammar define a language? The central idea is that production is that productions may be applied repeatedly to expand the nonterminal in a string of nonterminal and terminals. For example, consider the following grammar for arithmetic expression:

E→E+E|E*E|(E)|-E|id…….(1-1).

The nonterminal E is an abbreviation for expression. The production E→-E signifies that an expression preceded by minus sign is also an expression.in the simplest case can replace single E by –E. we can describe this action by writing E→-E which is read as "E derives –E"

We can take a single E and repeatedly apply production in any order to obtain a sequence of replacements for example:

E→-E→-(E)→-(id)

We call such as a sequence of replacements a derivation of – (id) from E.

This derivation provides a proof that one particular instance of an expression is the string – (id).

Example: the string – (id + id) is a sentence of grammar (1.1) because there is the derivation

$$E \rightarrow -E \rightarrow -(E) \rightarrow -(E+E)$$

$$\rightarrow -(id+E) \rightarrow -(id+id)$$

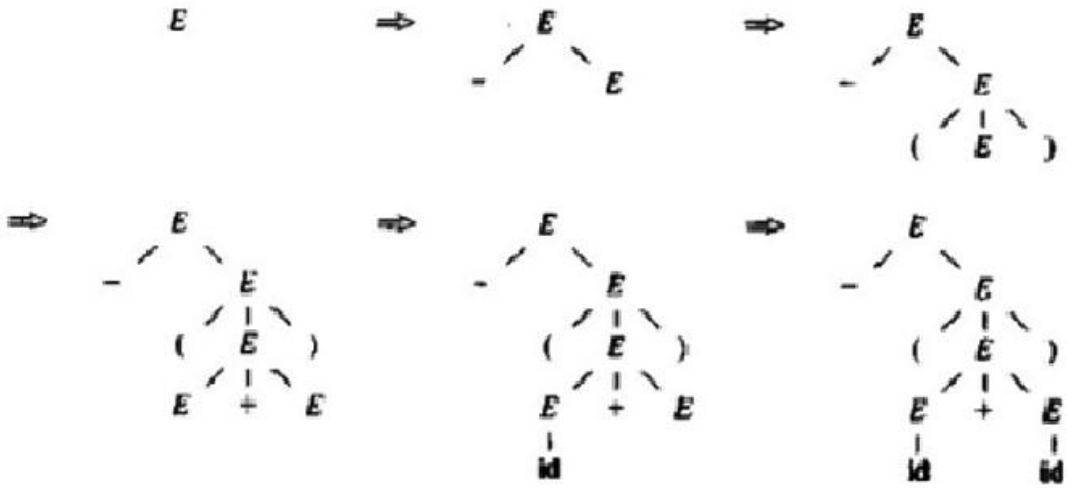
Parse Tree and Derivations

A parse tree may be viewed as a graphical representation for an derivation that fillers out the choice regarding replacement order, that each **interior node** of parse tree is labeled by some **non-terminal A**, and that the children of the node are labeled, from left to right, by the symbols in the right side of the production by which this **A** was replaced in the derivation,

The **leaves** of the parse tree are labeled by **non-terminals or terminals** and, **read** from **left to right**; they constitute a sentential form, **called** the **yield or frontier** of the tree.

For example, the parse tree for -(id+id) implied previously is shown bellow,

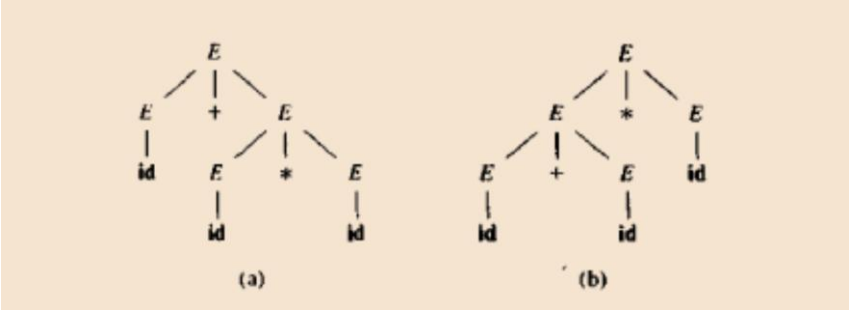
For the grammar: E→ E+E | E-E | E*E | E/E|-E|id



Example: Consider the previous arithmetic expression grammar, the sentence id+id*id has the two distinct left most derivations:

$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow \mathbf{id}+E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
\qquad\qquad
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E+E*E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$

With the two corresponding parse trees shown below:



Two parse trees for id + id * id

In another method to determine whether a statement is accepted or not, this method is called (**Derivation Method**).

There are two type of derivation:

1- Leftmost derivation.

2- Rightmost derivation.

Example:-

Let G be a grammar with this components ({S,E,F,P,R,L},{a, b, (,), +, -, ×,^,/},S,P}

P=

S→E S→+E S→-E E→T
T→F F→P P→b R→ a (L)
E→E+T E→T×F F→F^P L→ (S)
S→E – T E→ T/F P→a P→ (S)

Is statement a×(b + a) accepted or not ?

Leftmost derivation:

S→E→T × F→F×F→ P×F→ a × F → a × P → a × (S) → a×(E)→ a × (E+T) →
a × (T+T) → a× (F+T) → a×(P+T) → a×(b +T) →a×(b +F) →a×(b +P) →a×(b +a)

The statement a × (b + a) is accepted.

Rightmost derivation:

S→E→ T×F→ T×P → T×(S) → T×(E) →T×(E+T) → T× (F+T) →
T×(P+T) → T × (b+T) → T× (b+F) → T × (b+P) → T× (b+a) →F × (b+a)→
p×(b+a) →a×(b+a).

The statement a × (b+a) is accepted

Writing Grammar :

Grammars are capable of describing most, but not all of syntax of programming languages. A limited amount of syntax Analysis is done by produce the sequence of tokens from the input characters, certain constraints on the input, such as the requirement that identifiers be declared before they are used, cannot be described by a context-free-grammar.

Every construct that can be described by a **regular expression** can also be described by a **grammar** For **example**: the regular expression (a|b)*abb the NFA is:

A0 →aA0 | bA0 | aA1
A1 →bA2
A2→ bA3
A3 →ϵ

The grammar above was constructed from **NFA** using the following constructed:

• For each state **i** of **NFA**, create a non-terminal symbol **Ai.**

• If ***state* i** have a transition to ***state* j** on symbol **a**, introduction the production

Ai → aAj

• If state **i** go to state **j** on input **ϵ**, introduce the production Ai → Aj

• If state **i** is on accepting state introduce Ai → ϵ

• If state **i** is the start state, make **Ai** be symbol of the grammar.

RE' S are most useful for describing the structure of lexical constructs such as identifiers, constants, keywords, and so forth.

Grammars, on the other hand, are most useful in describing nested structures such as balanced parenthesis, matching begin- end's. Corresponding if - then else's.

These nested structures cannot be described by **RE.**

Ambiguity

A grammar that produces more than one parse tree for some sentence is said to be ambiguous. Put another way, an ambiguous grammar is one that produces more than one leftmost or more than one rightmost derivation for some sentence. In this type, we cannot uniquely determine which parse tree to select for a sentence.

Example: Consider the following grammar for arithmetic expressions involving

+, -, *, /, and↑ (exponentiation)

E → ***E+E*** | ***E-E*** | ***E*E*** | ***E/E*** | ***E*↑*E*** | **(*E*)** | **-*E*** | **id**

This grammar is ambiguous.

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity.

As an example ambiguous "**else**" grammar:

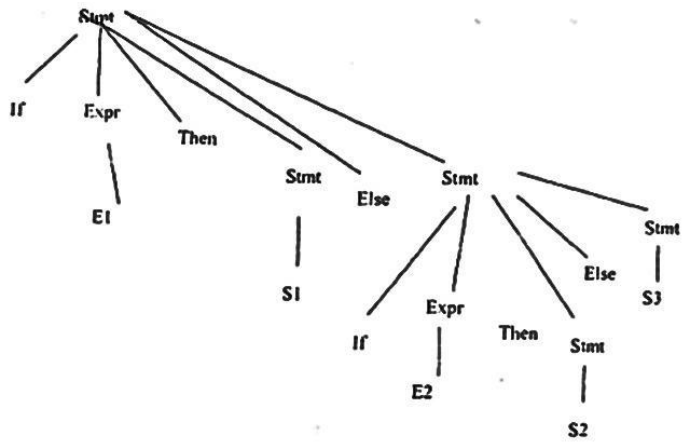
Stmt→ Expr **then** Stmt

| **if** Expr then Stmt **else** Stmt

|other

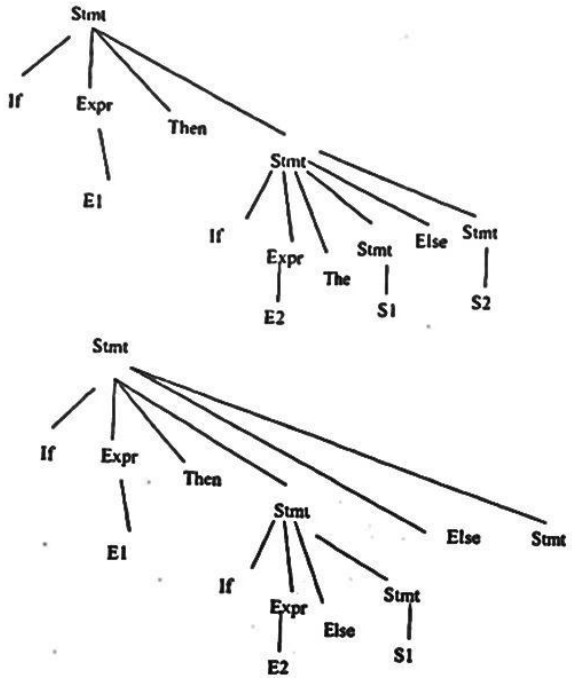
According to this grammar, the compound conditional statement

If E_l then S_1 else if E_2 then S_2 else S_3 has the parse tree link below:



The grammar above is ambiguous since the string:

If **E1** then if **E2** then **S1** else **S2**. Has the two parse trees shown below:



Generating of Context-Free Grammar

Every language that can be described by a regular expression or regular language can also be described by Context-Free Grammar.

Example: Design CFG that accept the RE = ***a*** (***a***|***b***)****b***

S → **a*A*b**

A→ **a*A* | b*A* |** ·

Example: Design CFG that accept the RE = (***a***|***b***)****abb***

S→ bS | aS | aX

X→b Y

Y→b Z

Z→ϵ.

Example: Design CFG that accept the $a_n b_n$ where $n \geq 1$.

S →***aXb***

X → ***aXb*** | ϵ ·

Example: Design CFG ***Singed Integer*** number.
S →***XD***
X →+ | -
D →**0*D*** | **1*D*** | ***2D*** | ***3D*** | ***4D*** | ***5D*** | ***6D*** | ***7D*** | **8*D*** | ***9D*** | ·

Parser techniques:-

Parser (syntax Analyzer)		
Top –down parser (predictive parser)		Bottom – up parser (Operator-precedent parser)
With backtracking	Without backtracking	

Backtracking manipulating:

1- **Factoring**
2- **Substitution**
3- **Left-Recursion Elimination**

Left Recursion Elimination:-

1- **Immediate Left- Recursion Elimination.**
2- **Not – Immediate Left Recursion Elimination.**

Immediate Left- Recursion Elimination

A grammar is left recursion if it has a non- terminal A, such that there is a derivation A →αA For some string **α**. **Top-down parsing methods cannot handle left recursion grammars, so a transformation that eliminates left recursion is needed.**

In the following example, we show how that **left recursion pair** of production

A→ Aα|β
Could be replaced by the non- left -recessional productions:
A →Aα|β
A→ β**A'**
A'→ α**A'**| ϵ

Example: Consider the following grammar for arithmetic expressions.

E →E+T |T
T →T*F|F
F→ (E) | id

Eliminating the immediate left recursion (productions of the form A→ Aα to the production for **E** and then for **T**, we obtain:

E→ TE'
E' →+T E'| ϵ
T→ FT'
T' →*FT' | ϵ
F →(E) | id

Note: No matter how many **A** productions there are, we can **eliminate immediate left recursion** from them by the following technique. First we group the **A** Production as:

A→ $A\alpha_1$| $A\alpha_2$| …| $A\alpha_n$| β_1| β_2| … β_n

Where no **β**i begins with an **A**. then, we replace the **A** -productions by

A →β_1A'| β_2A'| …| β_n A'

A→ α_1A'| α_2A'| …| α_n A' | ϵ

This produce eliminates all **immediate left recursion** from **A** and **A'** production. But it does **not eliminate left recursion** involving derivation of **two or more steps**.

For **Example** consider the grammar.

S→ **A**a | b

A →**A**c | **S**d | ϵ

The non- terminal **S** is **left- recursion** because **S**→ **A**a →**S**da, but is **not immediately left recursion**.

Algorithm: Elimination left recursion.

Input: Grammar G with no cycles or Ɛ-productions.

Output: An equivalent grammar with no left recursion.

Method: Apply the algorithm below to G. not that the resulting non left recursive grammar may have Ɛ-productions.

Arrange the non-terminals in some order $A_1, A_2, \ldots\ldots A_n$.

For i:= 1 to n do

For j:= 1 to i-1 do begin

Replace each production of the form $A_i \rightarrow A_j\gamma$ by the production $\mathbf{A_i \rightarrow \delta_1\gamma | \delta_2\gamma | \ldots | \delta_k\gamma}$

Where $A_j \rightarrow \delta_1 | \delta_2 | \delta_3 | \ldots\ldots | \delta_k$ are all the current A_j-productions

End.

Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing. The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

For example: if we have the two productions if ***A*** **αβ1 | αβ2** are two ***A***-productions, and the input begins with a nonempty string derived from **α**, we do not know whether to expand ***A*** to **αβ1**or to **αβ2**.

However, we may defer the decision by expanding ***A*** to **α*A*'**. Then, after seeing the input derived from **α**, we expand ***A'*** to **β1** or to **β2**. That is left-factored.

A → αβ1 | αβ2

A → α A'

A' → β_1 | β_2

Algorithm : Left factoring a grammar.

Input: Grammar ***G***.

Output*:* An equivalent left-factored grammar.

Method: For each nonterminal ***A*** find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \bullet$, i.e., there is a nontrivial common prefix, replace all the ***A*** productions $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \cdots | \alpha\beta_2 | \gamma$ where γ represents all alternatives that do not begin with α by:

$A \rightarrow \alpha A' | \gamma$

$A' \rightarrow \beta_1 | \beta_2 | \cdots\cdots\cdots | \beta_n$

Here ***A'*** is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

Example:

S → iEtS | iEtSeS | a

E → b

Left-factored, this grammar becomes:

S → iEtSS' | a

S'→eS | •

E →b

Example:

A→aA| bB | ab | a | bA

Solution:

A→aA'| bB'

A' → A | b | •

B' →B | A