SYNTAX ANALYSIS

Backus-Naur Form.

- BNF stands for Backus-Naur Form or Backus Normal Form.
- A meta language is a language that is used to describe another language.
- BNF is a meta language (formal notations) for programming languages syntax.
- A production is a rule relating to a pair of strings, say ά and β, specifying how one may be transformed into the other. This may be denoted

  $$ά \rightarrow β.$$

- For simple theoretical grammars, upper case letters are used for non-terminals and lower case letters are used for terminals.
- For more realistic grammars, such as those used to specify programming languages, the most common way of specifying productions is to use the notations invented by Backus commonly called BNF.
- These notations were first introduced by Backus for describing ALGOL 58.
- These notations were later modified slightly by Peter Naur for the description of ALGOL 60.

Backus-Naur Form.

- In BNF:
  - A non-terminal and terminal are usually given some descriptive names.
  - Non-terminals symbols are written in angle brackets to distinguish it from a terminal symbol.
  - If there are multiple definitions for the same non-terminal symbol, then they can be written as single rule, separated from each by using (|) vertical bar which means logical OR.

Example.

G = {N , T , S , P}

N = {<sentence> , <qualified noun> , <noun> , <pronoun> , <verb> , <adjective> }

T = { the , man , girl , boy , lecturer , he , she , drinks , sleeps , mystifies , tall , thin , thirsty }

S = <sentence>

P = {

<sentence>	$\rightarrow$	the <qualified noun> <verb>	(1)
	|	<pronoun> <verb>	(2)
<qualified noun>	$\rightarrow$	<adjective> <noun>	(3)
<noun>	$\rightarrow$	man | girl | boy | lecturer	(4, 5, 6, 7)
<pronoun>	$\rightarrow$	he | she	(8, 9)
<verb>	$\rightarrow$	talks | listens | mystifies	(10, 11, 12)
<adjective>	$\rightarrow$	tall | thin | sleepy	(13, 14, 15)

}

- Derive the “The sleepy boy listens” by using the above grammar.

Problems of a CFG.

- Three types of problems are mainly faced in a CFG.
  - Ambiguity.
  - Left Recursion.
  - Common Prefixes.
- Three problems must be removed from a CFG, otherwise the grammar will not work accurately.

Ambiguity.

- An ambiguous grammar is one that:
  - Produces more than one parse trees for the same sentence.
  - Produces more than one leftmost derivations or rightmost derivations for the same sentence.
- A grammar becomes ambiguous when a single non-terminal appears twice or more times on the L.H.S of the production rules in the grammar.
- If more than one parse trees can be produced for a sentence; then the compiler would not be able to generate the code uniquely.

Example.

- Consider the following grammar.

```
<assign>  →  <id>  =  <expr>
<id>      →  A | B | C
<expr>    →  <expr> + <expr>
           |  <expr> * <expr>
           |  (<expr>)
           | <id>
```

- Now show that this grammar is ambigous for the sentence A = B + C * A.

Parse Trees

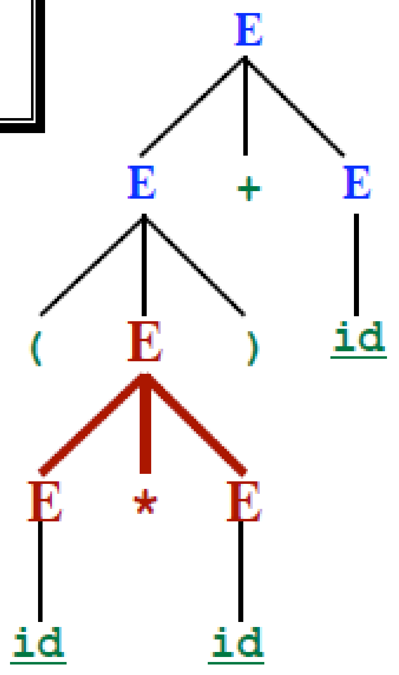
Two choices at each step in a derivation...

- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Leftmost Derivation:

$$\begin{aligned} & E \\ \Rightarrow\ & E+E \\ \Rightarrow\ & (E)+E \\ \Rightarrow\ & (E*E)+E \\ \Rightarrow\ & (\underline{id}*E)+E \\ \Rightarrow\ & (\underline{id}*\underline{id})+E \\ \Rightarrow\ & (\underline{id}*\underline{id})+\underline{id} \end{aligned}$$



1. $E \rightarrow E + E$
2. $ \rightarrow E * E$
3. $ \rightarrow (E)$
4. $ \rightarrow - E$
5. $ \rightarrow \text{ID}$

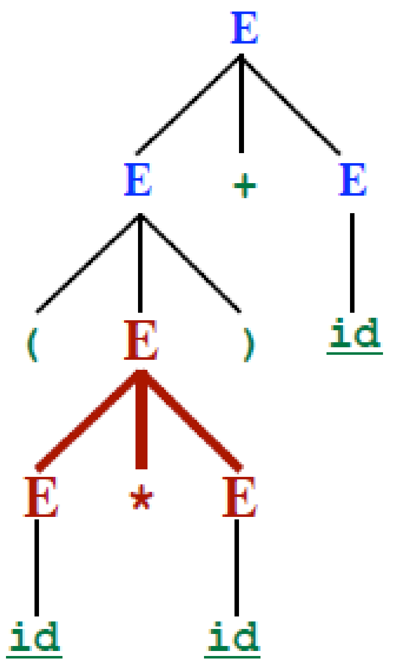
Parse Trees

Two choices at each step in a derivation...
- Which non-terminal to expand
- Which rule to use in replacing it

The parse tree remembers only this

Rightmost Derivation:

E
⇒ E+E
⇒ E+id
⇒ (E)+id
⇒ (E*E)+id
⇒ (E*id)+id
⇒ (id*id)+id



1.	E	→ E + E
2.		→ E * E
3.		→ (E)
4.		→ - E
5.		→ ID

Parse Trees

Two choices at each step in a derivation...

- Which non-terminal to expand
- Which rule to use in replacing it

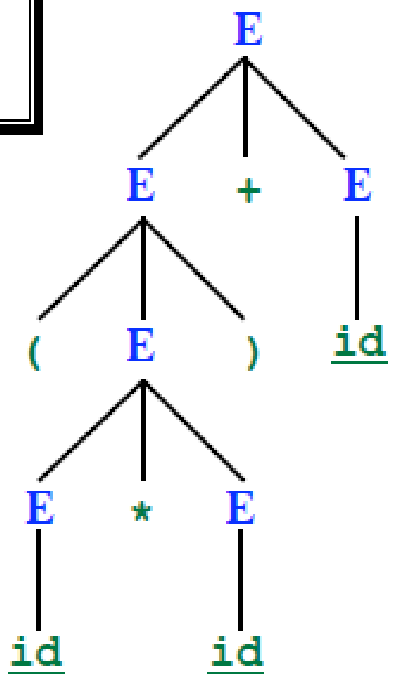
The parse tree remembers only this

Leftmost Derivation:

```
     E
⇒    E+E
⇒    (E)+E
⇒    (E*E)+E
⇒    (id*E)+E
⇒    (id*id)+E
⇒    (id*id)+id
```

Rightmost Derivation:

```
     E
⇒    E+E
⇒    E+id
⇒    (E)+id
⇒    (E*E)+id
⇒    (E*id)+id
⇒    (id*id)+id
```



1.	E	→ E + E
2.		→ E * E
3.		→ (E)
4.		→ - E
5.		→ ID

Given a leftmost derivation, we can build a parse tree.
Given a rightmost derivation, we can build a parse tree.

Lefttmost Derivation of `(id*id)+id` **Rightmost Derivation of** `(id*id)+id`

Same Parse Tree

Every parse tree corresponds to...

- A single, unique leftmost derivation
- A single, unique rightmost derivation

Ambiguity:

However, one input string may have several parse trees!!!

Therefore:

- Several leftmost derivations
- Several rightmost derivations

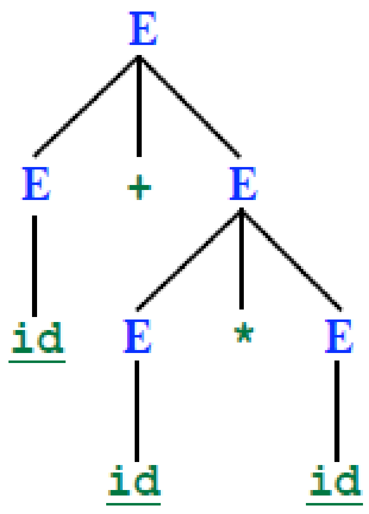
Ambuiguous Grammars

1. E → E + E
2. → E * E
3. → (E)
4. → - E
5. → ID

Input: `id+id*id`

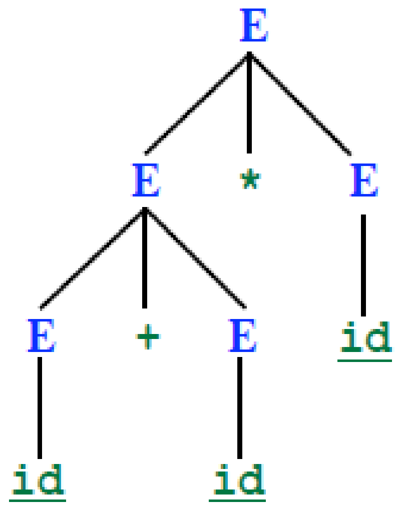
Leftmost Derivation #1

```
     E
⇒    E+E
⇒    id+E
⇒    id+E*E
⇒    id+id*E
⇒    id+id*id
```



Leftmost Derivation #2

```
     E
⇒    E*E
⇒    E+E*E
⇒    id+E*E
⇒    id+id*E
⇒    id+id*id
```



Ambiguous Grammar

More than one Parse Tree for some sentence.

- The grammar for a programming language may be ambiguous
  - Need to modify it for parsing.

Also: Grammar may be left recursive.

- Need to modify it for parsing.

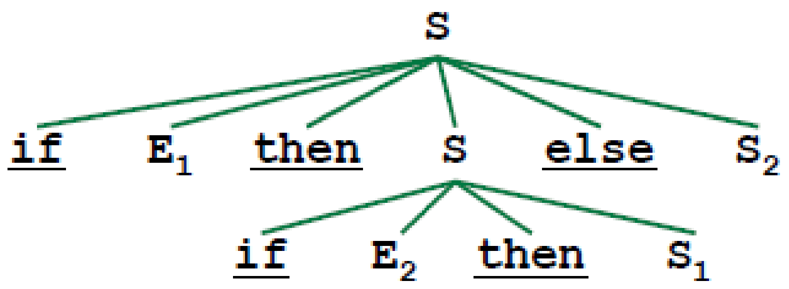
The Dangling "Else" Problem

This grammar is ambiguous!

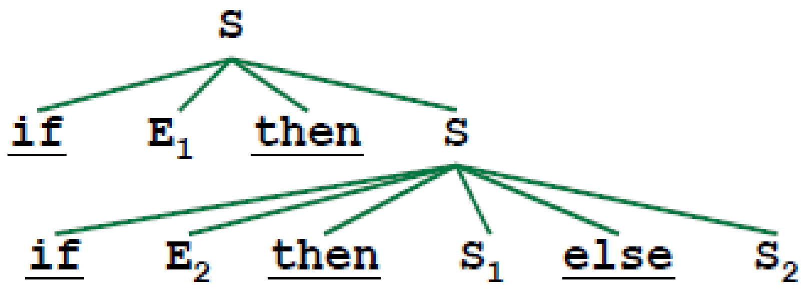
Stmt → if Expr then Stmt
→ if Expr then Stmt else Stmt
→ ...Other Stmt Forms...

Example String: `if` E_1 `then if` E_2 `then` S_1 `else` S_2

Interpretation #1: `if` E_1 `then (if` E_2 `then` S_1`) else` S_2



Interpretation #2: `if` E_1 `then (if` E_2 `then` S_1 `else` S_2`)`



The Dangling "Else" Problem

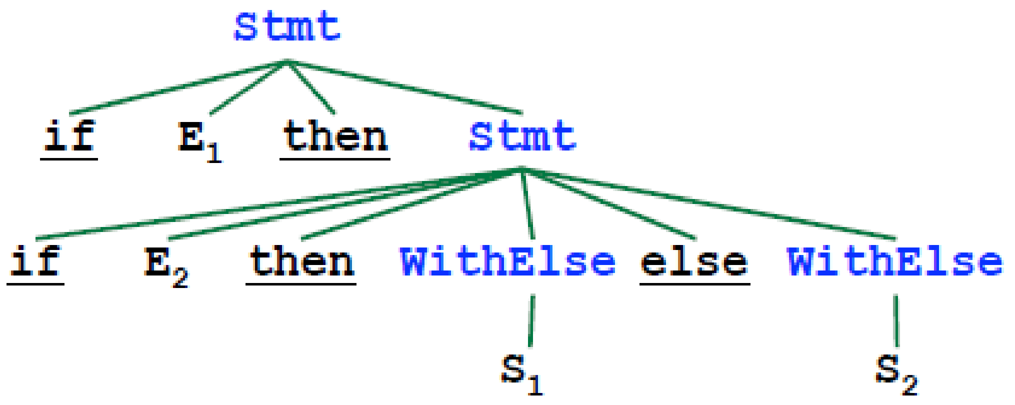
Goal: "Match `else`-clause to the closest `if` without an `else`-clause already."

Solution:

Stmt	→	`if` Expr `then` Stmt
	→	`if` Expr `then` WithElse `else` Stmt
	→	...Other Stmt Forms...
WithElse	→	`if` Expr `then` WithElse `else` WithElse
	→	...Other Stmt Forms...

Any Stmt occurring between `then` and `else` must have an `else`.
i.e., the Stmt must not end with "`then` Stmt".

Interpretation #2: `if` E_1 `then (if` E_2 `then` S_1 `else` S_2`)`



The Dangling "Else" Problem

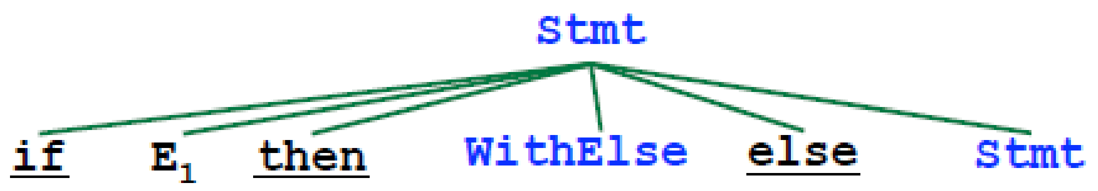
Goal: "Match else-clause to the closest if without an else-clause already."

Solution:

```
Stmt      →  if Expr then Stmt
          →  if Expr then WithElse else Stmt
          →  ...Other Stmt Forms...
WithElse  →  if Expr then WithElse else WithElse
          →  ...Other Stmt Forms...
```

Any Stmt occurring between then and else must have an else.
i.e., the Stmt must not end with "then Stmt".

Interpretation #1: `if` E_1 `then (if` E_2 `then` S_1`) else` S_2



The Dangling "Else" Problem

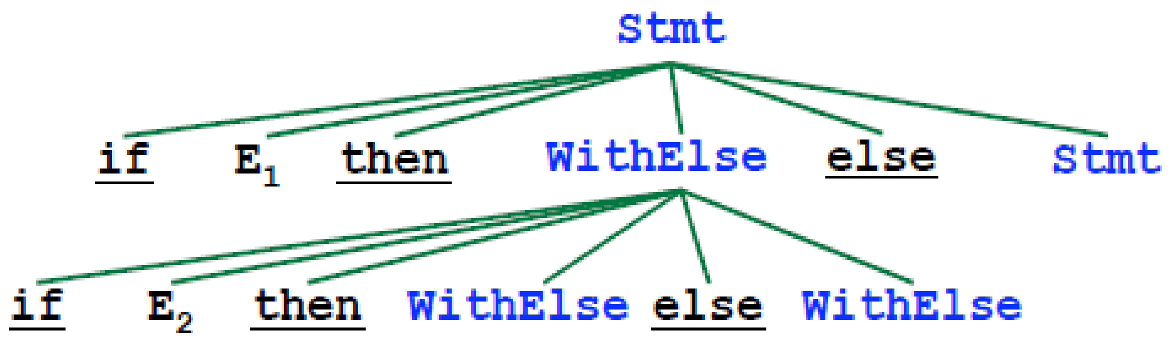
Goal: "Match `else`-clause to the closest `if` without an `else`-clause already."

Solution:

Stmt $\rightarrow$ `if` Expr `then` Stmt
 $\rightarrow$ `if` Expr `then` WithElse `else` Stmt
 $\rightarrow$ ...Other Stmt Forms...
WithElse $\rightarrow$ `if` Expr `then` WithElse `else` WithElse
 $\rightarrow$ ...Other Stmt Forms...

Any Stmt occurring between `then` and `else` must have an `else`.
i.e., the Stmt must not end with "`then` Stmt".

Interpretation #1: `if` E_1 `then (if` E_2 `then` S_1`) else` S_2



The Dangling "Else" Problem

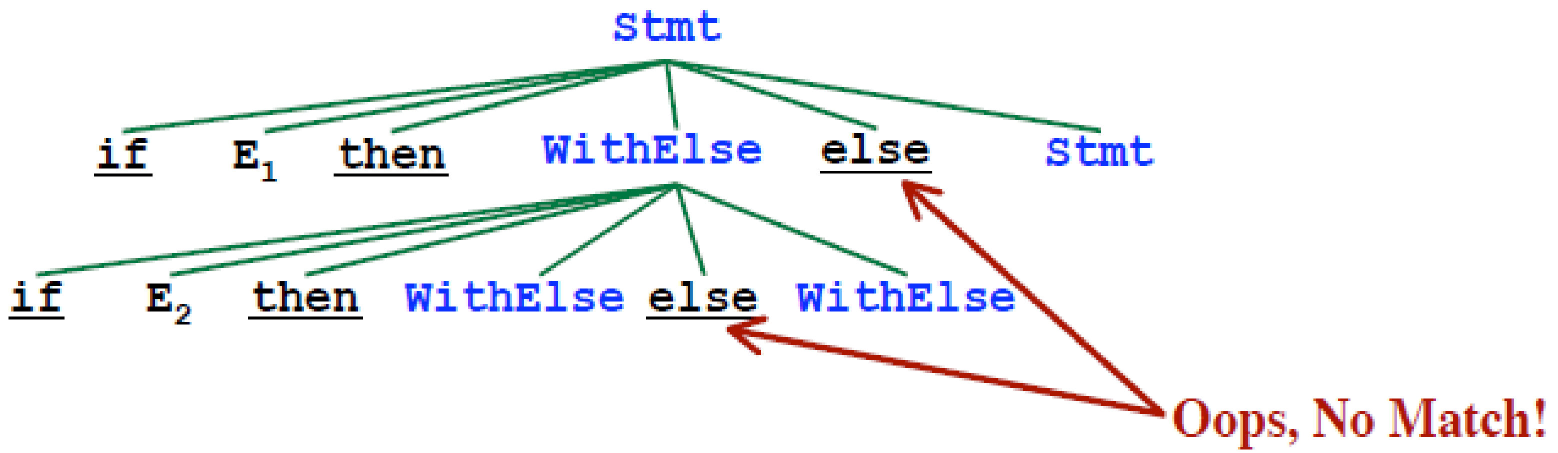
Goal: "Match `else`-clause to the closest `if` without an `else`-clause already."

Solution:

Stmt	→	`if` Expr `then` Stmt
	→	`if` Expr `then` WithElse `else` Stmt
	→	...Other Stmt Forms...
WithElse	→	`if` Expr `then` WithElse `else` WithElse
	→	...Other Stmt Forms...

Any Stmt occurring between `then` and `else` must have an `else`.
i.e., the Stmt must not end with "`then` Stmt".

Interpretation #1: `if` E_1 `then (if` E_2 `then` S_1`) else` S_2



Null Production

<u>Definition</u>: The production of the form

nonterminal → ☹

is said to be ***null production***.

<u>Example</u>: Consider the following CFG

S → aA|bB|☹, A → aa|☹, B → aS

Here S → ☹ and A → ☹ are null productions.

Following is a note regarding the null productions

Note

If a CFG has a null production, then it is possible to construct another CFG without null production accepting the same language with the exception of the word ☹ *i.e.* if the language contains the word ☹ then the new language cannot have the word ☹.

Following is a method to construct a CFG without null production for a given CFG

Null Production continued …

Method: Delete all the Null productions and add new productions *e.g.*

consider the following productions of a certain CFG X → aNbNa, N → ☹, delete the production N → ☹ and using the production

X → aNbNa, add the following new productions

X → aNba, X → abNa and X → aba

Thus the new CFG will contain the following productions X → aNba|abNa|aba|aNbNa

Note: It is to be noted that X → aNbNa will still be included in the new CFG.

Nullable Production

Definition: A production is called ***nullable production*** if it is of the form

N → ☹

or

there is a derivation that starts at N and leads to ☹ *i.e.*

N_1 → N_2, N_2 → N_3, N_3 → N_4, …, N_n → ☹, where N, N_1, N_2, …, N_n are non terminals.

Following is an example

Example

Consider the following CFG

S → AA|bB, A → aa|B, B → aS | ☹

Here S → AA and A ✹ B are nullable productions, while B → ☹ is null a production.

Following is an example describing the method to convert the given CFG containing null productions and nullable productions into the one without null productions

Example

Consider the following CFG

S → XaY|YY|aX|ZYX

X → Za|bZ|ZZ|Yb

Y → Ya|XY|☹

Z → aX|YYY

It is to be noted that in the given CFG, the productions S → YY, X → ZZ, Z → YYY are Nullable productions, while Y → ☹ is Null production.

Example continued …

Here the method of removing null productions, as discussed earlier, will be used along with replacing nonterminals corresponding to nullable productions like nonterminals for null productions are replaced.

Thus the required CFG will be

S → XaY|Xa|aY|a|YY|Y|aX|ZYX|YX|ZX|ZY

X → Za|a|bZ|b|ZZ|Z|Yb

Y → Ya|a|XY|X|Y

Z → aX|a|YYY|YY|Y,

Following is another example

Example

Consider the following CFG

S → XY, X → Zb, Y → bW

Z → AB, W → Z, A → aA|bA|☹

B → Ba|Bb|☹.

Here A → ☹ and B → ☹ are null productions, while Z → AB, W → Z are nullable productions. The new CFG after, applying the method, will be

Example continued …

$S \rightarrow XY$

$X \rightarrow Zb|b$

$Y \rightarrow bW|b$

$Z \rightarrow AB|A|B$

$W \rightarrow Z$

$A \rightarrow aA|a|bA|b$

$B \rightarrow Ba|a|Bb|b$

Note

While adding new productions all Nullable productions should be handled with care. All Nullable productions will be used to add new productions, but only the Null production will be deleted.

Unit production

Unit production: The productions of the form

nonterminal ✹ one nonterminal,

is called the ***unit production***.

Following is an example showing how ***to eliminate the unit productions from a given CFG.***

Unit production continued …

<u>Example</u>: Consider the following CFG

$S \rightarrow A|bb,$

$A \rightarrow B|b,$

$B \rightarrow S|a$

Separate the unit productions from the nonunit productions as shown below

unit prods.	nonunit prods.
$S \rightarrow A$	$S \rightarrow bb$
$A \rightarrow B$	$A \rightarrow b$
$B \rightarrow S$	$B \rightarrow a$

Example continued …

S → A gives S → b (using A → b)

S → A → B gives S → a (using B → a)

A → B gives A → a (using B → a)

A → B → S gives A → bb (using S → bb)

B → S gives B → bb (using S → bb)

B → S → A gives B → b (using A → b)

Thus the new CFG will be

S → a|b|bb, A → a|b|bb, B → a|b|bb.

Which generates the finite language {a,b,bb}.