BNF

In 1958 John Backus proposed formalism for describing languages.

Peter Naur modified it. It became BNF used first for describing ALGOL.

BNF is metalanguage.

BNF uses abstract syntax structures.

Uses Rules or Productions.

Ex:

<assign> → <var> = <expr>

Abstractions are called non-terminals.

Token → category of lexeme

LHS → RHS → either

↳ non terminal

<E> → <E> + <E>

<E> → <E> − <E>

<E> → <Term>

<Term> → id

brackets = things are non terminal

+ − id are terminal

Lets eliminate < > for brevity

Sequence of Replacements ⇒ Derivation

E → E+E

→ E−E+E

→ Term−E+E

→ id−E+E

→ id−Term+E

→ id−id+id

Sentencial form ⊆ Language

Sentence ∈ Language

Start from the starting non terminal

↳ all intermediate steps are sentencial form

↳ Every sentence is a sentential form (RIP spelling)

By applying grammar rules, we get a derivation

These grammar rules are generators.

<program> → begin <stmt_list> end

<stmt> → <var> <expression>

A grammar for a small lang

① <program> → begin <stmt_list> end

② <stmt_list> → <stmt> | <stmt>; <stmt_list>

③ <stmt> → <var> = <expression>

④ <var> → A | B | C

⑤ <expression> → <var> + <var> |
<var> - <var> | <var>

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt> ; <stmt_list> **end**
=> **begin** <var> = <expression> ; <stmt_list> **end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt> **end**
=> **begin** A = B + C ; <var> = <expression> **end**
=> **begin** A = B + C ; B = <expression> **end**
=> **begin** A = B + C ; B = <var> **end**
=> **begin** A = B + C ; B = C **end**

Last sentential form is a sentence, and has only terminals

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

→ Whitespace is not allowed with a token

→ 001

<int> = <digit_list> | <digit>
<digit-list> = <digit> | <digit> <digitlist>
<digit> = 0|1|2|3|4|5 ... 9
(assume 000 is valid)

Include sign

<int> → <sign><digitlist>
<digitlist> → <digit> | <digit><digit_list>
<digit> ⇒ 0|1|2-- 9
<sign> → +|−

Exclude :01 :0036 :000 etc
[+4036 is valid]

(HW)

<int> → <sign><digit_list>
<digit_list>

Sentence list

E → E+E E → term
E → E-E term → id

NTs: {E, term}
Terminals: {+, −, id}
StartNT: E

→ Your derivation should always start from Starting non terminal
→ Your starting non terminal contains only 1 lexeme

Then choose a non terminal and replace it with a suitable non terminal/terminal
repeat till you reach all ~~no~~ terminals

, if you start from the left-most non terminal and use & repeat

You get a Left-most Derivation (LMD)
they RMD for right-most Derivation (RMD)

LMD

$E \Rightarrow E+E$

$E \rightarrow E+E+E$

$E \rightarrow \text{term}+E+E$

$E \rightarrow id+E+E$

$E \rightarrow id+\text{term}+E$

$\vdots$

$E \Rightarrow id+id+id$

RMD

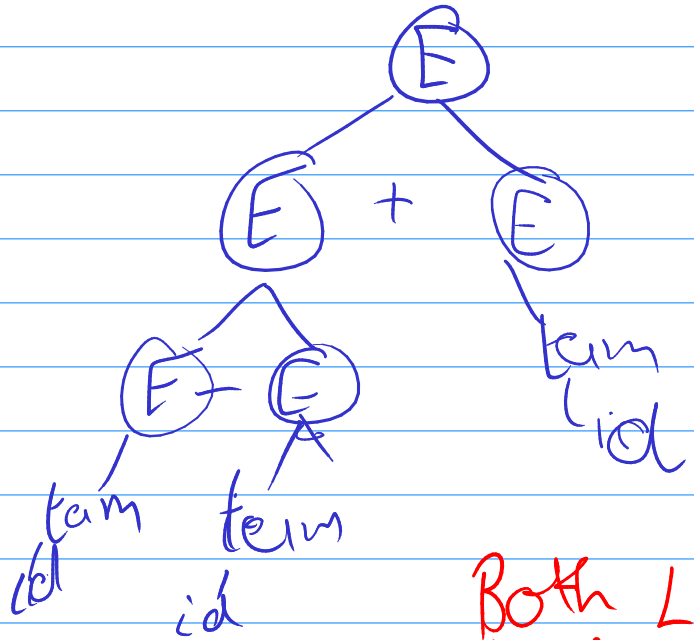
$E \Rightarrow E+E$

$E \Rightarrow E+E-E$

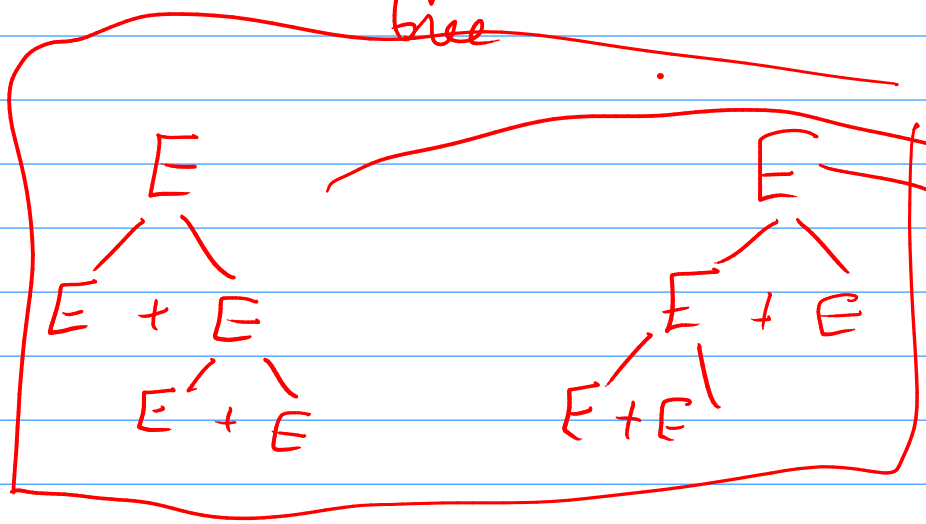
$E \Rightarrow E+E-\text{term}$

$\vdots$

$E \Rightarrow id+id-id$



Both LMD & RMD are represented the same way in a parse tree



However sometimes → different parse tree for some expression → ambiguous grammar