# BNF/EBNF:

## *BNF – Recursive:*

**BNF** (which is an acronym for Backus Naur Form) was invented in 1960 and used in the formal description of Algol-60.

The meta-symbols of BNF are:
**::=** meaning "is defined as"
**|** meaning "or"
**< >** angle brackets used to surround category names. e.g., <program>, <expression>, <S>
The angle brackets distinguish syntax rules names (also called non-terminal symbols) from terminal symbols which are written exactly as they are to be represented.
Example of terminals: WHILE,(, 3.
The empty string is written as <empty>

A BNF rule defining a nonterminal has the form:
*nonterminal* ***::=*** *sequence_of_alternatives* consisting of strings of terminals or nonterminals separated by the meta-symbol |
For example, the BNF production for a mini-language is:

*<program> ::= program*
*<declaration_sequence>*
*begin*
*<statements_sequence>*
*end ;*

This shows that a mini-language program consists of the keyword "program" followed by the declaration sequence, then the keyword "begin" and the statements sequence, finally the keyword "end" and a semicolon.

## *EBNF (Extended BNF) – Iterative*

- There's a dot '.' at the end of the line.
- Parentheses, ( and ), represent grouping,
- optional items are enclosed in meta symbols **[** and **]**, example:

  *<if_statement> ::= if <boolean_expression> then*
  *<statement_sequence>*
  ***[*** *else*
  *<statement_sequence>* ***]***
  *end if ;*

- repetitive items (zero or more times) are enclosed in meta symbols **{** and **}**, example:

  *<identifier> ::= <letter>* ***{*** *<letter> | <digit>* ***}***
  this rule is equivalent to the recursive rule:
  *<identifier> ::= <letter> |*
  *<identifier> [ <letter> | <digit> ]*

- terminals are surrounded by quotes (**"**) to distinguish them from meta-symbols, example:

  *<statement_sequence> ::= <statement>* ***{*** *";" <statement>* ***}***

*Here is the definition of EBNF expressed in EBNF:*

```
syntax     = { rule }.
rule       = identifier "::=" expression.
expression = term { "|" term }.
term       = factor { factor }.
factor     = identifier | quoted_symbol | "(" expression ")" | "[" expression "]" | "{" expression "}".
identifier = letter { letter | digit }.
quoted_symbol = """ { any_character } """.
```

**BNF Example:**

<program> ::= BEGIN <statement-seq> END
<statement-seq> ::= <statement>
<statement-seq> ::= <statement> ; <statement-seq>
<statement> ::= <while-statement>
<statement> ::= <for-statement>
<statement> ::= <empty>
<which-statement> ::= WHILE <expression> DO <statement-seq> END
<expression> ::= <factor>
<expression> ::= <factor> AND <factor>
<expression> ::= <factor> OR <factor>
<factor> ::= ( <expression> )
<factor> ::= <variable>
<for-statement> ::= …
<variable> ::= …

**EBNF Examples:**

Program = "BEGIN" Statement-seq "END".
Statement-seq = Statement [ ";" Statement-seq ].
Statement = [While-statement | For-statement ].
While-statement = "WHILE" Expression "DO" Statement-seq "END".
Expression = Factor { ("AND" | "OR") Factor }.
Factor = '(' Expression ')' | Variable.
For-statement = …
Variable = …

ident = letter {letter | digit}.
number = integer | real.
integer = digit {digit} | digit {hexDigit} "H".
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = ("E" | "D") ["+" | "-"] digit {digit}.
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
characterConst = digit {hexDigit} "X".
string = ' " ' {char} ' " ' | " ' " {char} " ' ".

***Parse Trees (Syntax Trees)…***