What is BNF ?

Backus-Naur notation (shortly BNF) is a formal mathematical way to describe a language, (to describe the syntax of the programming languages).

The Backus-Naur Form is a way of defining **syntax**. It consists of

- a set of terminal symbols
- a set of non-terminal symbols
- a set of production rules of the form
  `Left-Hand-Side ::= Right-Hand-Side`

where the LHS is a non-terminal symbol and the RHS is a sequence of symbols (terminals or non-terminals).

- The meaning of the production rule is that the non-terminal on the left hand side may be replaced by the expression on the right hand side.

- Any sentence which is derived using the production rules is said to be **syntactically** correct.
  It is possible to check the syntax of a sentence by building a **parse tree** to show how the sentence is derived from the production rules. If it is not possible to build such a tree then the sentence has syntax errors.
- Syntax rules define how to produce *well-formed sentences*. But this does not imply that a well formed sentence has any sensible meaning. Semantics define what a sentence means.
- It is used to formally define the grammar of a language

How it works ?

BNF is sort of like a mathematical game: you start with a symbol (called the start symbol and by convention usually named S in examples) and are then given rules for what you can replace this symbol with. The language defined by the BNF grammar is just the set of all strings you can produce by following these rules.

The rules are called production rules, and look like this:

```
symbol := alternative1 | alternative2 ...
```

- A production rule simply states that the symbol on the left-hand side of the := must be replaced by one of the alternatives on the right hand side.
- The alternatives are separated by |s. (One variation on this is to use ::= instead of :=, but the meaning is the same.) Alternatives usually consist of both symbols and something called terminals.
- Terminals are simply pieces of the final string that are not symbols.
- There is one special symbol in BNF: @, which simply means that the symbol can be removed. If you replace a symbol by @ you do it by just removing the symbol. This is useful because in some cases it is difficult to end the replacement process without using this trick.
- So, the language described by a grammar is the set of all strings you can produce with the production rules. If a string cannot in any way be produced by using the rules the string is not allowed in the language.

A real example

Below is a sample BNF grammar:

```
  S  := '-' FN |
        FN
  FN := DL |
        DL '.' DL
  DL := D |
        D DL
  D  := '0' | '1' | '2' | '3' | '4' | '5' | '6'
| '7' | '8' | '9'
```

- The different symbols here are all abbreviations: S is the start symbol, FN produces a fractional number, DL is a digit list, while D is a digit.
- Valid sentences in the language described by this grammar are all numbers, possibly fractional, and possibly negative. To produce a number, start with the start symbol S
- Then replace the S symbol with one of its productions. In this case we choose not to put a '-' in front of the number, so we use the plain FN production and replace S by FN:
- The next step is then to replace the FN symbol with one of its productions. We want a fractional number, so we choose the production that creates two decimal lists with a '.' between them, and after that we keep choosing replacing a symbol with one of its productions once per line in the example below:

```
Step1:   DL . DL
Step2:   D . DL
Step3:   3 . DL
Step4:   3 . D DL
Step5:   3 . D D
Step6:   3 . 1 D
Step7:   3 . 1 4
```

Here we've produced the fractional number 3.14.

EBNF: What is it, and why do we need it?

In DL We had to use recursion (ie: DL can produce new DLs) to express the fact that there can be any number of Ds. This is a bit complicated and makes the BNF harder to read. Extended BNF (EBNF, of course) solves this problem by adding new three operators:

- ? : which means that the symbol (or group of symbols in parenthesis) to the left of the operator is optional (it can appear zero or one times)
- * : which means that something can be repeated any number of times (and possibly be skipped altogether)
- + : which means that something can appear one or more times

An EBNF sample grammar

So in extended BNF the above grammar can be written as:

```
  S := '-'? D+ ('.' D+)?

  D := '0' | '1' | '2' | '3' | '4' | '5' | '6'
| '7' | '8' | '9'
```

which is rather nicer. :)

Just for the record: EBNF is not more powerful than BNF in terms of what languages it can define, just more convenient. Any EBNF production can be translated into an equivalent set of BNF productions.

Abbreviation for Alternative Rules

The alternative production rules may be listed more concisely using a choice bar, | , as a separator. For example the rules above could be given as:

```
<NaturalNumber> ::=  <digit>                    Rule1a
                   | <digit> <NaturalNumber>  Rule1b

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

                                               Rule2
```

Abbreviation for Optional Symbols

An optional element in a production rule is denoted by using square brackets [ ].
Symbols enclosed in curly brackets may be repeated zero, one or more times. A subscript may give the a minimum and maximum number of repetitions.

$\{\ \}$ - zero, one or more
$\{\ \}_1$ - at least one
$\{\ \}_{1..5}$ - at least one and no more than five occurrences