BNF Grammar (**B**ackus-**N**aur **F**orm)

Metalanguage that describes another language's ***syntax***.

It is equivalent to ***context free grammar*** (CFG) : ({T}, {N}, {P}, S) where
{T}: set of all terminal symbols: 0,1,2,…, a, b, c, …, +, -, …,
Symbols that can not be reduced further more.
{N}: set of all nonterminal symbols: *statement-sequence*, *if-statement*, *expression*,….
Symbols that need to be further reduced (replaced with other expansion symbols) (expanded).
{P}: set of production rules,
the left hand side symbol must be a "nonterminal"

Ex: <unsigned integer> ::= <digit> | <unsigned integer> <digit>

S: starting symbol (S belongs to {N})

Ex: <program> ::= **program** <header> ;
<declaration-section> ; <program-body> **end**

<program> is the starting symbol in this case.

A ***regular grammar*** is either a **left** or **right** grammar.
A **right regular grammar** is same as CFG, but all production rules P are one of the following rules:
1- A → a - **A** is a non-terminal in N and **a** is terminal in T
2- A → a B - A and **B** are non-terminal in N and **a** is terminal in T
3- A → ε (empty string) - **A** is a non-terminal in N
A **left regular grammar** is same as above except for rule 2,
where "A → Ba" replaces of A→ aB
A regular grammar can be both, right and left grammar, otherwise it would be CFG.

The BNF is powerful enough to describe the following ***syntactic*** issues in a programming language definition:

1- Lists of similar constructs: statement-sequence, declaration-sequence,…
2- The order in which different constructs must appear: a label must start with a letter not a digit.
3- Nested structures to any depth: nested statements
4- Matching parentheses: (((((((A+B)))))))
5- Operator precedence: the "/" has higher precedence over the "+"

6- Operator Associativity:

BNF for expressions:

<expr> ::= <expr> <add-op> <term> (* left associative BNF*)
| <term>
<term> ::= <term> <mult-op> <factor> (*precedence: mult-op > add-op*)
| <factor>
<factor> ::= <id> | <number> | **-** <factor> | **(**<expr>**)**

<add-op> ::= **−** | **+**

<mult-op> ::= **/** | *****

In general, there are two way to evaluate this assignment: X := A − B + C

i) If the BNF expresses **left associative** then the above is execute as
X := (A – B) + C
Left associative: <expr> ::= <expr> <add-op> <term>

ii) But in case of **right associative** then the above is execute as
X := A – (B + C) ;
Right associative: <expr> ::= < term> <add-op> <expr>

Yet, BNF can never describe any language semantics issues (context sensitive issues!) (e.g., variable not declared, over/underflow, label length is not acceptable, …).

Extended BNF:

The same BNF power but more descriptive and it increases the readability of the BNF!

Extended BNF uses the following extensions to the regular BNF:

1- "optional" parts in the rhs of a rule:
Ex: <if-stmt> ::= **if (**<logic-expr>**) then** <stmt> [**else** <stmt>];
<integer> ::= [**+**| **–**] <unsigned-integer>

2- zero or more repetitions of some constructs using braces.
Ex: <stmt-list> ::= <stmt> {; <stmt>}

3- At least one occurrence: (*kleene* +)
Ex: <unsigned integer> ::= <digit>$^{+}$

4- multiple-choice options:
Ex: <for-stmt> ::= **for** <id> **:=** <expr> **(to**|**downto)** <expr> **do** <stmt>

EBNF of the a different expression implementation:

<expr> ::= <term> { (+|-) <term> }
<term> ::= <factor> {(* | /) <factor> }
<factor> ::= <expr> {**<expr>} -- The "**" symbol indicates the ***exponent*** operator
<expr> ::= (<expr>) | id

Attribute Grammar to Describe Context Sensitive (static semantics) Languages Aspects:

It associates a set Attr(F) of attributes to each non-terminal symbol in the grammar, Attr(F) has two disjoint sets of attributes: S(F)-- ***synthesized*** & I(F) -- ***inherited*** attributes; where S(F) is synthesized bottom-up and then I(F) is to be inherited top-down over the program parse tree.

<unsigned-int> ::=

<digit>

{Value (**<unsigned-int>**) ← Value (**<digit>**) **}**

$$.value <-- $1.value, synthesizing the inherited value of $1(<**digit**>) into $$ (<**unsigned-int**>).

| **<unsigned-int> <digit>**

{Value (**<unsigned-int>**) ← **10** * Value (**<unsigned-int>**) + Value (**<digit>**)

$$.value <-- 10*$1.value + $2.value, synthesizing the inherited values of $2(<**digit**>) added to the multiplication of $1 (<**unsigned-int**>) by 10).

(***the accumulated/obtained values of <unsigned-int> and <digit> are examples of *inherited* attribute(top-down in the pars tree); yet <unsigned-int> is *synthesized* attribute (bottom-up in the pars tree) *)**

To check the static semantics issue of overflow:

If (Value (<unsigned-int>) ≥ **n**) Then [output error ***overflow***"]

}

<digit> ::= **0**

{Value (**<digit>**) ← **0**} -- synthesizing the value 0 into $1

$$.value <-- $1.value

| **1**

{Value (**<digit>**) ← **1**}-- synthesizing the value 1 into $1

$$.value <-- $1.value

| **2**

{Value (**<digit>**) ← **2**}-- synthesizing the value 2 into $1

$$.value <-- $1.value

| **3**

{Value (<**digit**>) ← **3**}-- synthesizing the value 3 into $1

$$.value <-- $1.value

***

| **9**

{Value (<**digit**>) ← **9**}-- synthesizing the value 9 into $1

$$.value <-- $1.value

(* all values of <digit> 0–to-9 are *synthesized* bottom-up attributes*)

Questions: Give more examples of synthesized and inherited attributes in any Algol program parse tree.

[Hint: think of the type declaration section and its utilization in the program body for type checking]