BNF Grammar (Backus-Naur Form)

*Meta*language 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>*

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

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

*<program>* is t he 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:

\[
<\text{expr}> ::= <\text{expr}> \langle \text{add-op} \rangle <\text{term}> \quad (* \text{left associative BNF*})
\]

\[
| <\text{term}>
\]

\[
<\text{term}> ::= <\text{term}> \langle \text{mult-op} \rangle <\text{factor}> \quad (*\text{precedence: mult-op > add-op*})
\]

\[
| <\text{factor}>
\]

\[
<\text{factor}> ::= <\text{id}> | <\text{number}> | - <\text{factor}> | (<\text{expr}>)
\]

\[
<\text{add-op}> ::= - | +
\]

\[
<\text{mult-op}> ::= / | *
\]

In general, there are two way to evaluate this assignment: \( X := A - B + C \)

i) If the BNF expresses \textbf{left associative} then the above is execute as

\[
X := (A - B) + C
\]

Left associative: \(<\text{expr}> ::= <\text{expr}> \langle \text{add-op} \rangle <\text{term}>\)

ii) But in case of \textbf{right associative} then the above is execute as

\[
X := A - (B + C) ;
\]

Right associative: \(<\text{expr}> ::= <\text{term}> \langle \text{add-op} \rangle <\text{expr}>\)

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

\textbf{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: \( <\text{if-stmt}> ::= \text{if} (<\text{logic-expr}> \text{then} <\text{stmt}> [\text{else} <\text{stmt}>]; <\text{integer}> ::= [ + - ] <\text{unsigned-integer}>\)

2- zero or more repetitions of some constructs using braces.

Ex: \( <\text{stmt-list}> ::= <\text{stmt}> \{ ; <\text{stmt}>>\}

3- At least one occurrence: (\textit{kleene} +)

Ex: \( <\text{unsigned integer}> ::= <\text{digit}>^+\)

4- multiple-choice options:

Ex: \( <\text{for-stmt}> ::= \text{for} <\text{id}> := <\text{expr}> \langle \text{to|downto} \rangle <\text{expr}> \text{do} <\text{stmt}>\)
Attribute Grammar to Describe Context Sensitive Languages Aspects:

\[ \text{<unsigned-int> ::= } \begin{cases} \text{<digit>} \\ \{ \text{value (<unsigned-int>) } \rightarrow \text{value (digit)} \} \\ \text{<unsigned-int> <digit>} \\ \{ \text{Value (<unsigned-int>) } \rightarrow 10 \times \text{Value (<unsigned-int>)} \\ + \text{value (digit)} \} \\ \text{(* the value of <unsigned-int> is } \text{inherited} \text{ attribute, top-down*)} \end{cases} \]

\[ \text{If (Value (<unsigned-int>) } \geq n \text{) Then [output error message “overflow”]} \} \]

\[ \text{<digit> ::= } \begin{cases} 0 \\ \{ \text{Value (<digit>) } \rightarrow 0 \} \\ 1 \\ \{ \text{Value (<digit>) } \rightarrow 1 \} \\ 2 \\ \{ \text{Value (<digit>) } \rightarrow 2 \} \quad \text{(* all values of digits 0–to-9 are } \text{synthesized} \text{ (bottom-up) attributes*)} \\ 3 \\ \{ \text{Value (<digit>) } \rightarrow 3 \} \quad \text{***} \\ 9 \\ \{ \text{Value (<digit>) } \rightarrow 1 \} \end{cases} \]

**Questions:** Give more examples of synthesized and inherited attributes in the program parse tree.

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

------------------------------------------------------------------------------------------------------------

**PASCAL**

- ALGOL’s like language, but more **reliable**, **efficient**, and **simple** for pedagogic purposes.

- PASCAL introduced a **much richer type system** than ALGOL:

  A) **“type”** and **“Const”** declarations.
  B) **“Enumeration Types”**: To handle non-numeric data, eliminating the insecurity of overworking integers as such needed types.

    (exists in C, Pascal, and Ada)

    ```
    type
      month = (Jan, Feb, March, Apr, …, Dec);
    Day   = (Sun, Mon, Tue, …, Sat);
    
    In C : enum StudentClass {Fresh, Soph, Junior, Senior}
           enum EmployeeGender {Male, Female}
    In Ada : type Days is ((Sun, Mon, Tue, …, Sat);
    
    They are **true ADTs**, with **user** defined data type, and **system** built-in operators:
    ```
Advantages:

1- High level application oriented, allowing the language to cover wider area of applications.
2- Efficient use of memory to represent the type values.
3- Secure use of type elements, the compiler protects against any meaningless operations by the users on elements of the defined types. Pred(Jan) and succ (Sat) will produce compile time errors.

Question: Can we still use the standard input/output commands for enumerated types?

Problems? Yes:
1- No Input/output built in operations! (why?)
2- Overloaded enumerated literal constants when appearing in different definitions at the same environment.
   Ex: type favoriteColors = (red, yellow, magenta, brown, aqua, blue, green);
   TrafficLightColors = (red, yellow, green);

   for color in ‘red’ .. ’green’ do

Notice that “color” is implicitly typed with the specified discrete range by the compiler. The discrete range is ambiguous(!) since the compiler will not know to which type it belongs: favoriteColors or TrafficLightColors ?

Hence, C and Pascal do not allow the same element name to be used in more than one enumeration type in the same scope.

Solution by Ada-- Ada allows it but to resolve the name overloading:
   for color in favoriteColors ‘(‘red’) .. favoriteColors’(‘green’)’ do

C) “Subrange Types”: Pascal (and other languages like Ada) allows the programmer to define subranges of only discrete types (enumerated, int, characters), where it inherits all of its parent defined set of operations.

   type uppercase = ‘A’..’Z’; index = 1..100; WeekDays = Mon..Fri;

Advantages:
1- Enhances readability.
2- Security, errors due the assignment of out of range value will be detected at compile time, in case of literal constant value; or at run time, in case of expression/variable assignment.
3- Efficient memory representation of the type values.

*Notice that when the “subrange” inherits its parent set of operations, it introduces “security loophole”. For example, “dayOfMonth = 1..31”, it is maybe ok to add/subtract the days of month, but what about dividing/multiplying them??!!
Subrange are built in Algol 68, Pascal, and Ada; but not C++/Java.

**In C++, it is defined via the following:**
(http://www.russel.org.uk/subrange.html)

```c
typedef subrange::<
  subrange::<
    ordinal_range<short, 1, 12> > Month;
Month m ;
m = 3 ; // OK
m += 20 ; // ERROR outside range 1..12
```

what about: `m *= 2;`

or

`m /= 3;` // what does it mean?

**D) “Set Type”:**

Advantages:

1- High level and application oriented
2- Efficient representation.
3- ADT and readability.

```c
type favoriteColors = (red, yellow, magenta, brown, aqua, blue, green);
colorset          = set of favoriteColors;
```

```c
var set1, set2 : colorset;
digits : set of char;
```

```c
begin
  ***
  set1 := [red, blue, yellow, aqua];
  set2 := [brown, green];
  T := [ 1..10]; /* set of integers 1 to 10*/
  S := [1, 2, 5, 7, 12, 13];
  T := T * S ; /* T will have 1,2,5,7 */
(* the operator * is used instead of the intersection \intersection *)
  if T = [1, 2, 5, 7] then …
  digits := ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'];
  read (ch);
  if ch in digits the …. /* the in operation is used for membership *)
```

More operations on sets: `< , > strict subset (S1<S2  S2>S1) : Boolean
<= not equal sets (S1<>S2) : Boolean
<=, >= subset or equal (S1\leq S2  S2 \geq S1) : Boolean
− difference between two sets (S1– S2) : set
+ Union of two sets (S1 + S2): set
= equal sets (S1 =S2): Boolean
E) “Record Type”: One of the most important contributions by PASCAL as a “heterogeneous” data structure that aggregates a group of related, but different data types fields that pertain to an object (e.g., employee, student, etc). It is called “structure” in C.

Variant Records: (pages 187-189): Similar to the C language “union”, variant records aim at sharing memory again, with “aliasing” which will introduce a problem!

F) Typed “Pointer”: Pascal is among the pioneer languages to introduce typed pointers (memory locations), for linked structures. C and C++ also have typed pointers. Java has no pointers, but it has references (pointers to structures, class instances, no need to dereferences, and it is nonsense to apply arithmetic ops on them!).

In PASCAL: var p: ^ real; x: real; c : char;
begin new (p); p^ := 3.14159; c := p^; {compiler error!}

In C&C++:
int *ptr; int count, init; *** ****

ptr = &init; count = *ptr;

They also have a non-types pointers “void *ptr” as generic pointers of type any; in case we need to have a function to deal with memory blocks of byte/words (any type). Just define the formal parameters to be “void *ptr”, to deal with any actual parameter pointer type you send to the function!

Questions: Is the use of non-types pointers secure? What type of tradeoff is that of the “void *ptr” facility?

We can have pointers to any type, not only basic types (int, real, ..).
Ex: var p: ^ plane; (* plane is a record type*)

p^.parked := ....

- A pointer with the value “nil” points to nowhere! (what is the type of a nil pointer?)

( C&C++ use value 0 instead of nil)