CENTRO DE INVESTIGACIÓN Y DE ESTUDIOS AVANZADOS DEL IPN
UNIDAD TAMAULIPAS

Assignment #2

Student: *Luis Alberto Ballado Aradias*
Course: *Tecnologías Computacionales (Sep - Dec 2022)*
Professor: *Dr. Edwin Aldana Bobadilla*
October 7, 2022

BNF Notation

BNF is a formal mathematical way of defining syntax unambiguosly.
It consists of:

- A set of terminal symbols
- A set of non-terminal symbols
- A set of production rules

LHS::= RHS ,LHS - LeftHandSide, ::= - is defined by, RHS - RightHandSide

Note: The LHS is always a non terminal symbol and The RHS is a sequence of symbols (terminals or non terminals)

for exmaple:

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

A letter can then be defined as:

$$< LETTER >::= A|B|C|D|E|.....|Z$$

$$< WORD >::=< LETTER > | < LETTER >< WORD >$$

So this now means a **word** is a single letter or a **letter** and a **word**

<>	Non terminal
::=	Produce
|	Alternative

Expressions

[]	Optional
	Repetition
$A \ldots \beta$	rangos

Why is Backus-Naur form needed?

Can we create and apply BNF to describe the rules of a language?

- **Grammar:** Syntax and structure of a language.
- **Natural Language:** a language that has developed naturally through use. (like Spanish, English .. etc).

BNF is a meta language - a language to describe another languages.
Need for Meta Languages:

- Determine whether a series of characters is valid
- Generate well-formed statements
- Break down a statement into constituent parts so it can be converted into machine code

BNF Deals with Terminals and Non-terminals

- Terminals: Symbols that can appear in the output of a language because of its rules, but cannot be changed by the rules themselves.
- Non-terminals: Syntactic entities that define a part of the grammar (can be change by the rules).

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

In the previuos example Non-terminals is <DIGIT>, and the Terminals al the possible numbers a digit can be

We already know that **regular expressions** can be used to help us describe a simple language by specifying patterns of strings which match conditions.

Many aspects of languages can be defined using **regular expressions**, but it is both lengthy and time consuming.

Some aspects of languages such as the use of nested brackets cannot be difined using **regular expressions** at all, so we turn to meta-languages and Backus-Naur Form is an example of a meta language.

BNF Notation in Python

```
ets may not.
star_targets:
    | star_target !','
    | star_target (',' star_target )* [',']
    star_targets_list_seq: ','.star_target+ [',']
    star_targets_tuple_seq:
    | star_target (',' star_target )+ [',']
    | star_target ','
    star_target:
    | '*' (!'*' star_target)
```

```
    | target_with_star_atom
    target_with_star_atom:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | star_atom
    star_atom:
    | NAME
    | '(' target_with_star_atom ')'
    | '(' [star_targets_tuple_seq] ')'
    | '[' [star_targets_list_seq] ']'

    single_target:
    | single_subscript_attribute_target
    | NAME
    | '(' single_target ')'
    single_subscript_attribute_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead

    del_targets: ','.del_target+ [',']
    del_target:
    | t_primary '.' NAME !t_lookahead
    | t_primary '[' slices ']' !t_lookahead
    | del_t_atom
    del_t_atom:
    | NAME
    | '(' del_target ')'
    | '(' [del_targets] ')'
    | '[' [del_targets] ']'

    t_primary:
    | t_primary '.' NAME &t_lookahead
    | t_primary '[' slices ']' &t_lookahead
    | t_primary genexp &t_lookahead
    | t_primary '(' [arguments] ')' &t_lookahead
    | atom &t_lookahead
    t_lookahead: '(' | '[' | '.'
```

BNF Notation in C

```
<translation-unit> ::= {<external-declaration>}*

<external-declaration> ::= <function-definition>
                         | <declaration>

<function-definition> ::= {<declaration-specifier>}* <declarator> {<declaration>}* <c
```

```
<declaration-specifier> ::= <storage-class-specifier>
                          | <type-specifier>
                          | <type-qualifier>

<storage-class-specifier> ::= auto
                            | register
                            | static
                            | extern
                            | typedef

<type-specifier> ::= void
                   | char
                   | short
                   | int
                   | long
                   | float
                   | double
                   | signed
                   | unsigned
                   | <struct-or-union-specifier>
                   | <enum-specifier>
                   | <typedef-name>

<struct-or-union-specifier> ::= <struct-or-union> <identifier> { {<struct-declaration>
                              | <struct-or-union> { {<struct-declaration>}+ }
                              | <struct-or-union> <identifier>

<struct-or-union> ::= struct
                    | union

<struct-declaration> ::= {<specifier-qualifier>}* <struct-declarator-list>

<specifier-qualifier> ::= <type-specifier>
                        | <type-qualifier>

<struct-declarator-list> ::= <struct-declarator>
                           | <struct-declarator-list> , <struct-declarator>

<struct-declarator> ::= <declarator>
                      | <declarator> : <constant-expression>
                      | : <constant-expression>

<declarator> ::= {<pointer>}? <direct-declarator>

<pointer> ::= * {<type-qualifier>}* {<pointer>}?

<type-qualifier> ::= const
                   | volatile
```

```
<direct-declarator> ::= <identifier>
                      | ( <declarator> )
                      | <direct-declarator> [ {<constant-expression>}? ]
                      | <direct-declarator> ( <parameter-type-list> )
                      | <direct-declarator> ( {<identifier>}* )

<constant-expression> ::= <conditional-expression>

<conditional-expression> ::= <logical-or-expression>
                           | <logical-or-expression> ? <expression> : <conditional-exp

<logical-or-expression> ::= <logical-and-expression>
                          | <logical-or-expression> || <logical-and-expression>

<logical-and-expression> ::= <inclusive-or-expression>
                           | <logical-and-expression> && <inclusive-or-expression>

<inclusive-or-expression> ::= <exclusive-or-expression>
                            | <inclusive-or-expression> | <exclusive-or-expression>

<exclusive-or-expression> ::= <and-expression>
                            | <exclusive-or-expression> ^ <and-expression>

<and-expression> ::= <equality-expression>
                   | <and-expression> & <equality-expression>

<equality-expression> ::= <relational-expression>
                        | <equality-expression> == <relational-expression>
                        | <equality-expression> != <relational-expression>

<relational-expression> ::= <shift-expression>
                          | <relational-expression> < <shift-expression>
                          | <relational-expression> > <shift-expression>
                          | <relational-expression> <= <shift-expression>
                          | <relational-expression> >= <shift-expression>

<shift-expression> ::= <additive-expression>
                     | <shift-expression> << <additive-expression>
                     | <shift-expression> >> <additive-expression>

<additive-expression> ::= <multiplicative-expression>
                        | <additive-expression> + <multiplicative-expression>
                        | <additive-expression> - <multiplicative-expression>

<multiplicative-expression> ::= <cast-expression>
                              | <multiplicative-expression> * <cast-expression>
                              | <multiplicative-expression> / <cast-expression>
```

```
                              | <multiplicative-expression> % <cast-expression>

<cast-expression> ::= <unary-expression>
                    | ( <type-name> ) <cast-expression>

<unary-expression> ::= <postfix-expression>
                     | ++ <unary-expression>
                     | -- <unary-expression>
                     | <unary-operator> <cast-expression>
                     | sizeof <unary-expression>
                     | sizeof <type-name>

<postfix-expression> ::= <primary-expression>
                       | <postfix-expression> [ <expression> ]
                       | <postfix-expression> ( {<assignment-expression>}* )
                       | <postfix-expression> . <identifier>
                       | <postfix-expression> -> <identifier>
                       | <postfix-expression> ++
                       | <postfix-expression> --

<primary-expression> ::= <identifier>
                       | <constant>
                       | <string>
                       | ( <expression> )

<constant> ::= <integer-constant>
             | <character-constant>
             | <floating-constant>
             | <enumeration-constant>

<expression> ::= <assignment-expression>
               | <expression> , <assignment-expression>

<assignment-expression> ::= <conditional-expression>
                          | <unary-expression> <assignment-operator> <assignment-expre

<assignment-operator> ::= =
                        | *=
                        | /=
                        | %=
                        | +=
                        | -=
                        | <<=
                        | >>=
                        | &=
                        | ^=
                        | |=
```

```
<unary-operator> ::= &
                   | *
                   | +
                   | -
                   | ~
                   | !

<type-name> ::= {<specifier-qualifier>}+ {<abstract-declarator>}?

<parameter-type-list> ::= <parameter-list>
                        | <parameter-list> , ...

<parameter-list> ::= <parameter-declaration>
                   | <parameter-list> , <parameter-declaration>

<parameter-declaration> ::= {<declaration-specifier>}+ <declarator>
                          | {<declaration-specifier>}+ <abstract-declarator>
                          | {<declaration-specifier>}+

<abstract-declarator> ::= <pointer>
                        | <pointer> <direct-abstract-declarator>
                        | <direct-abstract-declarator>

<direct-abstract-declarator> ::=  ( <abstract-declarator> )
                                | {<direct-abstract-declarator>}? [ {<constant-expressi
                                | {<direct-abstract-declarator>}? ( {<parameter-type-li

<enum-specifier> ::= enum <identifier> { <enumerator-list> }
                   | enum { <enumerator-list> }
                   | enum <identifier>

<enumerator-list> ::= <enumerator>
                    | <enumerator-list> , <enumerator>

<enumerator> ::= <identifier>
               | <identifier> = <constant-expression>

<typedef-name> ::= <identifier>

<declaration> ::=  {<declaration-specifier>}+ {<init-declarator>}* ;

<init-declarator> ::= <declarator>
                    | <declarator> = <initializer>

<initializer> ::= <assignment-expression>
                | { <initializer-list> }
                | { <initializer-list> , }
```

```
<initializer-list> ::= <initializer>
                     | <initializer-list> , <initializer>

<compound-statement> ::= { {<declaration>}* {<statement>}* }

<statement> ::= <labeled-statement>
              | <expression-statement>
              | <compound-statement>
              | <selection-statement>
              | <iteration-statement>
              | <jump-statement>

<labeled-statement> ::= <identifier> : <statement>
                      | case <constant-expression> : <statement>
                      | default : <statement>

<expression-statement> ::= {<expression>}? ;

<selection-statement> ::= if ( <expression> ) <statement>
                        | if ( <expression> ) <statement> else <statement>
                        | switch ( <expression> ) <statement>

<iteration-statement> ::= while ( <expression> ) <statement>
                        | do <statement> while ( <expression> ) ;
                        | for ( {<expression>}? ; {<expression>}? ; {<expression>}? )

<jump-statement> ::= goto <identifier> ;
                   | continue ;
                   | break ;
                   | return {<expression>}? ;
```

BNF Notation in Java

```
compilation_unit  =
 [ package_statement ]
 < import_statement >
 < type_declaration >  .

package_statement  =
 "package" package_name  ";"  .

import_statement  =
 "import"  (  ( package_name  "."  "*"  ";"  )
 /  ( class_name  / interface_name )  )  ";"  .

type_declaration  =
 [ doc_comment ]  ( class_declaration  / interface_declaration )  ";"  .
```

```
doc_comment  =  "/**"  "... text ..."  "*/"  .

class_declaration  =
 < modifier >  "class" identifier
 [  "extends" class_name ]
 [  "implements" interface_name  <  "," interface_name >  ]
 "{"  < field_declaration >  "}"  .

interface_declaration  =
 < modifier >  "interface" identifier
 [  "extends" interface_name  <  "," interface_name >  ]
 "{"  < field_declaration >  "}"  .

field_declaration  =
 (  [ doc_comment ]  ( method_declaration
 / constructor_declaration
 / variable_declaration )  )
 / static_initializer
 /  ";"  .

method_declaration  =
 < modifier > type identifier
 "("  [ parameter_list ]  ")"  <  "["  "]"  >
 ( statement_block  /  ";"  )  .

constructor_declaration  =
 < modifier > identifier  "("  [ parameter_list ]  ")"
statement_block  .

statement_block  =  "{"  < statement >  "}"  .

variable_declaration  =
 < modifier > type variable_declarator
 <  "," variable_declarator  >  ";"  .

variable_declarator  =
 identifier  <  "["  "]"  >  [  "=" variable_initializer ]  .

variable_initializer  =
expression
 /  (  "{"  [ variable_initializer
 <  "," variable_initializer  >  [  ","  ]  ]  "}"  )  .

static_initializer  =
 "static" statement_block  .

parameter_list  =
parameter  <  "," parameter >  .
```

```
parameter  =
type identifier  <  "["  "]"  >  .

statement  =
variable_declaration
 /  ( expression  ";"  )
 /  ( statement_block )
 /  ( if_statement )
 /  ( do_statement )
 /  ( while_statement )
 /  ( for_statement )
 /  ( try_statement )
 /  ( switch_statement )
 /  (  "synchronized"  "(" expression  ")" statement  )
 /  (  "return"  [ expression ]  ";"  )
 /  (  "throw" expression ";"  )
 /  ( identifier  ":" statement )
 /  (  "break"  [ identifier ]  ";"  )
 /  (  "continue"  [ identifier ]  ";"  )
 /  (  ";"  )  .

if_statement =
 "if"  "(" expression  ")" statement
 [  "else" statement ]  .

do_statement =
 "do" statement  "while"  "(" expression  ")"  ";"  .

while_statement =
 "while"  "(" expression  ")" statement  .

for_statement =
 "for"  "("  ( variable_declaration  /  ( expression  ";"  )  /  ";"  )
 [ expression ]  ";"
 [ expression ]  ";"
 ")" statement  .

try_statement =
 "try" statement
 <  "catch"  "(" parameter  ")" statement >
 [  "finally" statement ]  .

switch_statement =
 "switch"  "(" expression  ")"  "{"
 <  (  "case" expression  ":"  )
 /  (  "default"  ":"  )
 / statement >
```

```
 "}"  .

expression  =
numeric_expression
 / testing_expression
 / logical_expression
 / string_expression
 / bit_expression
 / casting_expression
 / creating_expression
 / literal_expression
 /  "null"
 /  "super"
 /  "this"
 / identifier
 /  (  "(" expression  ")"  )
 /  ( expression
 (  (  "("  [ arglist ]  ")"  )
 /  (  "[" expression  "]"  )
 /  (  "." expression  )
 /  (  "," expression  )
 /  (  "instanceof"  ( class_name / interface_name )  )
 )  )  .

numeric_expression  =
 (  (  "-"
 /  "++"
 /  "--"  )
expression )
 /  ( expression
 (  "++"
 /  "--"  )  )
 /  ( expression
 (  "+"
 /  "+="
 /  "-"
 /  "-="
 /  "*"
 /  "*="
 /  "/"
 /  "/="
 /  "%"
 /  "%="  )
expression  )  .

testing_expression  =
 ( expression
 (  ">"
```

```
 /  "<"
 /  ">="
 /  "<="
 /  "=="
 /  "!="  )
expression  )  .

logical_expression  =
 (  "!" expression )
 /  ( expression
 (  "ampersand"
 /  "ampersand="
 /  "|"
 /  "|="
 /  "^"
 /  "^="
 /  (  "ampersand"  "ampersand"  )
 /  "||="
 /  "%"
 /  "%="  )
expression  )
 /  ( expression  "?" expression  ":" expression )
 /  "true"
 /  "false"  .

string_expression  =  ( expression
 (  "+"
 /  "+="  )
expression  )  .

bit_expression  =
 (  "~" expression )
 /  ( expression
 (  ">>="
 /  "<<"
 /  ">>"
 /  ">>>"  )
expression  )  .

casting_expression  =
 "(" type  ")" expression  .

creating_expression  =
 "new"  (  ( classe_name  "("  [ arglist ]  ")"  )
 /  ( type_specifier  [  "[" expression  "]"  ]  <  "["  "]"  >  )
 /  (  "(" expression  ")"  )  )  .

literal_expression  =
```

```
integer_literal
 / float_literal
 / string
 / character  .

arglist  =
expression  <  "," expression  >  .

type  =
type_specifier  <  "["  "]"  >  .

type_specifier =
 "boolean"
 /  "byte"
 /  "char"
 /  "short"
 /  "int"
 /  "float"
 /  "long"
 /  "double"
 / class_name
 / interface_name  .

modifier =
 "public"
 /  "private"
 /  "protected"
 /  "static"
 /  "final"
 /  "native"
 /  "synchronized"
 /  "abstract"
 /  "threadsafe"
 /  "transient"  .

package_name =
identifier
 /  ( package_name  "." identifier )  .

class_name  =
identifier
 /  ( package_name  "." identifier )  .

interface_name =
identifier
 /  ( package_name  "." identifier )  .

integer_literal =
```

```
 (  (  "1..9"  <  "0..9"  >  )
 /  <  "0..7"  >
 /  (  "0"  "x"  "0..9a..f"  <  "0..9a..f"  >  )  )
 [  "l"  ]  .

float_literal  =
 ( decimal_digits  "."  [ decimal_digits ]  [ exponent_part ]  [ float_type_suffix ]
 /  (  "." decimal_digits  [ exponent_part ]  [ float_type_suffix ]  )
 /  ( decimal_digits  [ exponent_part ]  [ float_type_suffix ]  )  .

decimal_digits  =
 "0..9"  <  "0..9"  >  .

exponent_part  =
 "e"  [  "+"  /  "-"  ] decimal_digits  .

float_type_suffix  =
 "f"  /  "d"  .

character  =
  "based on the unicode character set"  .

string  =
 "’’"  < character  >  "’’"  .

identifier  =
  "a..z,$,_"  <  "a..z,$,_,0..9,unicode character over 00C0"  >  .
```

BNF Notation inSQL

```
                          BNF Grammar for SQL
/*   The grammar rules that follow have been taken from                    *
/*   "System R", Appendix II, M.M. Astrahan, et al., ACM Trans.            *
/*   on Database Systems, Vol. 1, No. 2, June 1976.                        *
/*   The rules given below in BNF have the following assumptions:          *
/*   (1) all non-terminals are in lower-case,                              *
/*   (2) all terminals (recognized by LEX/lex.yy.c) are in upper-case,     *
<statement>            ::=  <dml-statement>
                       |    <ddl-statement>

<dml-statement>        ::=  <selection>
                       |    <insertion>
                       |    <deletion>
                       |    <update>

<selection>            ::=  <select-clause>  FROM  <from-list>
                                <where-clause>  <grp-ord-clause>

<select-clause>        ::=  SELECT  <select-list>

<select-list>          ::=  <sel-expr-list>
                       |    MULT-OP

<sel-expr-list>        ::=  <sel-expr>
                       |    <sel-expr-list>  COMMA  <sel-expr>

<sel-expr>             ::=  <expr>
```

```
<from-list>              ::=  <table-name>
                         |    <from-list>  COMMA  <table-name>

  <where-clause>         ::=  <empty>
                         |    WHERE  <boolean>  <nested-select>

  <nested-select>        ::=  <empty>
                         |  <and-or> <table-name> <comparison> LPAR
                                  <selection> RPAR

  <grp-ord-clause>       ::=  <empty>
                         |    GROUP BY  <field-spec-list>
                         |    ORDER BY  <field-spec-list>

  <insertion>            ::=  INSERT INTO <receiver> COLON <insert-spec-source>

  <receiver>             ::=  <table-name>  <insert-spec-target>

  <insert-spec-target>   ::=  <empty>
                         |    LPAR  <field-name-list>  RPAR

  <field-name-list>      ::=  <field-name >
                         |    <field-name-list>  COMMA  <field-name>

  <field-name-list>      ::=  <field-name >
                         |    <field-name-list>  COMMA  <field-name>

  <insert-spec-source>   ::=  <literal>
```

SQL.2

```
<deletion>              ::=  DELETE  <table-name>  <where-clause>

<update>                ::=  UPDATE <table-name> <set-clause-list> <where-clause>

<set-clause-list>       ::=  <set-clause>

<set-clause>            ::=  SET  <field-name>  EQ  <expr>

<boolean>               ::=  <boolean-term>
                        |    <boolean>  OR  <boolean-term>

<boolean-term>          ::=  <boolean-factor>
                        |    <boolean-term>  AND  <boolean-factor>

<boolean-factor>        ::=  <boolean-primary>

<boolean-primary>       ::=  <predicate>

<predicate>             ::=  <expr>  <comparison>  <table-spec>

<comparison>            ::=  <comp-op>

<comp-op>               ::=  EQ
                        |    <relat-op>  <all-any-opt>
                        |    <in-notin>

<all-any-opt>           ::=  <empty>
                        |    <all-any>
```

SQL.3

```
<all-any>              ::=  ALL  |ANY

<in-notin>             ::=  IN  |NOT IN

<relat-op>             ::=  NE  |RWEDGE  |GE  |LWEDGE  |LE

<table-spec>           ::=  <literal>
                       |    <expr>

<literal>              ::=  <lit-tuple>
                       |    LPAR  <entry-list>  RPAR

<lit-tuple>            ::=  <entry>
                       |    LWEDGE  <entry-list>  RWEDGE

<entry-list>           ::=  <entry>
                       |    <entry-list>  COMMA  <entry>

<entry>                ::=  <constant>
<expr>                 ::=  <arith-term >
                       |    <expr>  ADD-OP  <arith-term>

<arith-term>           ::=  <arith-factor>
                       |    <arith-term>  MULT-OP  <arith-factor>

<arith-factor>         ::=  <opt-add-op>  <primary>
<opt-add-op>           ::=  <empty>  |ADD-OP
<and-or>               ::=  AND  |OR
```

SQL.4

```
<primary>              ::=  <field-spec>
                        |    <set-fn>  LPAR  <expr>  RPAR
                        |    LPAR  <expr>  RPAR
                        |    <constant>

<field-spec-list>      ::=  <field-spec>

<field-spec>           ::=  <field-name>
                        |    <table-name>  DOT  <field-name>

<set-fn>               ::=  AVG  |MAX  |MIN  |SUM  |COUNT

<constant>             ::=  QUOTE  <constant-value>  QUOTE
                        |    INTEGER

<constant-value>       ::=  IDENTIFIER  |VALUE

<field-name>           ::=  IDENTIFIER

<table-name>           ::=  IDENTIFIER

<empty>                ::=  EPSILON
```

```
<ddl-statement>        ::=  <create-table>

<create-table>         ::=  CREATE  TABLE  <table-name>  COLON
                            <field-defn-list>

<field-defn-list>      ::=  <field-defn>
                        |    <field-defn-list>  COMMA  <field-defn>

<field-defn>           ::=  <field-name>  LPAR  <type>  <null-opt>  RPAR

<type>                 ::=  CHAR  LPAR  INTEGER  RPAR
                        |    INT  LPAR  INTEGER  RPAR
                        |    FLOAT  LPAR  INTEGER  RPAR

<null-opt>             ::=  <empty>
                        |    COMMA  NONULL
```