An EBNF (*Extended Backus–Naur Form*) specifications for the Scol grammar.

A grammar is a set of production rules that describe all possibles strings in a language. It defines its syntax. The rules can be recursive.
This document might well be incomplete. If you see omissions, submit a bug or patch.

First, a short description :
A Scol program is a set of at least one package.
A package is a text file. it contains a set of at least one function and can declare one or more definition(s).
A function is a set of at least one instruction.

Some *Hello World* examples :

```
fun main ()=
     _fooS "Hello world !";;
```

```
fun print (szArg)=
     _fooS szArg;
     0;;

fun main ()=
     _showconsole;
     print  "Hello World !";;
```

```
fun print (szArg)=
     _fooS strcat szArg " World !";
     0;;

fun hello ()=
      print "Hello";;

fun bye ()=
     print "Good bye";;

fun main (iArg)=
     _showconsole;
     if iArg then
          hello
     else
          bye;;
```

Next, here the Scol grammar :

Any Scol application :
<programScol> ::= <package>+

A Scol package :
<package> ::= {<definition>}

```
# Definitions (function, variable, type, ...) :
<definition> ::= "fun" <funName> "(" <args> ")" "=" <program> ";;"
                | "typeof" <varName> = <typeBase> ";;"
                | "var" <varName> = <varValue> ";;"
                | "struct" <typeName> = {<fields>} ";;"
                | "typedef" <typeName> = <typeConstructor> ";;"
                | "defcom" <commName> = <commString> {"I" | "S"} ";;"
                | "defcomvar" <commVarName> = {"I" | "S"} ";;"
                | "proto" <funName> = <Type> ";;"

# In a function :
<args> ::= <none> | <args2>
<args2> ::= <local> | <local> "," <args2>

<program> ::= <expression> | <expression> ";" <program>

<expression> ::= <arithm> | <arithm> "::" <expression>

<arithm> ::= <a1> |  <a1> "&&" <arithm> | <a1> "||" <arithm>

# Priority order in an expression
<a1> ::= <a2> | "!" <a1>

<a2> ::= <a3> | <a3> "==" <a3> | <a3> "!=" <a3>
        | <a3> "<" <a3> | <a3> "<=" <a3>
        | <a3> ">" <a3> | <a3> ">=" <a3>
        | <a3> "<." <a3> | <a3> "<=." <a3>
        | <a3> ">." <a3> | <a3> ">=." <a3>
        | <a3> "!=." <a3> | <a3> "=." <a3>

<a3> ::= <a4> | <a4> "+" <a3> | <a4> "-" <a3>
        | <a4> "+." <a3> | <a4> "-." <a3>

<a4> ::= <a5> | <a6> "*" <a5> | <a6> "/" <a5>
        | <a6> "*." <a5> | <a6> "/." <a5>

<a5> ::= <a6> | <a6> "&" <a5> | <a6> "|" <a5> | <a6> "^" <a5> |
         | <a6> ">>" <a5> | <a6> "<<" <a5>

<a6> ::= <term> | "-" <a6> | "~" <a6>

<term> := <program> | <program> ";"
        | {<program>} | {<program> ";"}
        | <integer> | "'" <char> | <float>
        | <binary> | <octal> | <hexa>
        | <string> | {<arithm>} | "nil"
        | <varName>{.<term>} | "set " <varName>{.<term>} "=" <arithm>
        | <varName>{.<fieldName>} | "set " <varName>{.<fieldName>} "=" <arithm>
        | <funName> <argsFunction> | "@" <funName>
        | "let" <arithm> " -> " <locals> "in"
        | "if" <arithm> "then" <arithm> {"else" <arithm>}            # Scol 6.x and later
        | "while" <arithm> "do" <arithm>
```

```
        | "do" <arithm> "while" <arithm>                          # Scol 6.x and later
        | "mutate" <arithm> "<-" "[" {"_" | <locals>} "]"
        | "exec" <arithm> "with" <arithm>
        | <constr> <arithm> | <constr0> | "match" <arithm> "with" <case>

<argsFunction> ::= {<arithm>}       # as many times as the function has arguments

<locals> ::= <local> | "[" {"_" |  <locals>} "]"

<varValue> ::= <varValue2> | <varValue2> "::" <varValue>
<varValue2> ::= <integer> | <float> | <binary> |
        | """<char> | <string> | <octal>
        | <hexa> | "-" <integer> | "(" "-." <float> ")"
        | "[" {<varValue>} "]" | <varValue>+

# In a structure
<fields> ::= <field> | <fields>
<field> ::= <fieldName> ":" <typeBase>

# New type (from a typedef)
<typeConstructor> ::= <typeConstructor2> | <typeConstructor2> "|" <typeConstructor>
<typeConstructor2> ::= <constr> <typeBase> | <constr0>

<case> ::= <case2> | <case2> "|" <case> | "(" "_" "->" <arithm> ")"
<case2> ::= "(" <constr> <local> "->" <arithm> ")" |  "(" <constr0> "-> <arithm> ")"

# Others :
<local> ::= <name>    # local variable (or linked in a function : param)

<funName> ::= <name>                # function name
<varName> ::= <name>                # variable name
<typeName> ::= <nameU>              # type name
<commName> ::= <nameU>              # name of a communication constructor
<commVarName> ::= <nameU>           # name of a variable communication constructor
<constr> := type constructor        # constructor of a new type
<constr0> ::= empty type constructor
<fieldName> ::= structure field name

# Types :
<typeBase> ::= <t> | <r> | "tab" <typeBase>
        | "[" {<typeBase>} "]" | "fun" "[" {<typeBase>} "]"

<type> ::= <t> | <r> | <u>
        | "tab" <type> | "[" {<type>} "]" | "fun" "[" {<type>} "]"

<t> ::= "I" | "S" | "F" | "Chn" | "Srv" | "Comm" | "Env" | <type>

<u> ::= "u" <n>         # indifferent type  (like "u0", "u1", ...)
<r> ::= "r" <n0>        # recursive type (like "r1", "r2", ...)

<n> ::= [0-9]
<n0> ::= [1-9]
```

Basics (regular expressions) :
<name> ::= {{([a-Z]\|_)}{[0-9]}}
<nameU> ::= {{[A-Z]}{[0-9]}}

<string> ::= \"[0-9a-zA-Z]*\"
<char> ::= [0-9a-zA-Z]
<integer> ::= [0-9]+
<float> ::= {[0-9]+}.{[0-9]+}
<binary> ::= 0b{[0-1]+}
<octal> ::= 0o{[0-8]+}
<hexa> ::= 0x{[0-9a-fA-F]+}

Comments :
<commentLine> ::= "//" {.}

<commentBlock> ::= "/*" {.} "*/"

Launcher scripts (*.scol)
<launcher> ::= <launcherLine>+

<launcherLine> ::= ("_load " <packageName>) | <callMain>

<packageName> ::= package file name # *.pkg

<callMain> ::= <funName> <callArg>

<callArg> ::= <none> | <varValue2> " " <callArg>