**COMPILATEUR LANGAGE (ONE\_FOR\_ALL)**

**Encadré par :**

* Professeur TABII Youness

**Réalisé par:**

* SABOUR Zakaria
* TAMRY HAMZA
* TAMTAOUI Abdelwadoud
* TAZI Rida
* FALEH Yasser
* SAID EL Aboudi

Table des matières

[**DESCRIPTION** 3](#_Toc65888077)

[**GRAMMAIRE** 4](#_Toc65888078)

[**ANALYSEUR LEXICAL** 11](#_Toc65888079)

[**ANALYSEUR SYNTAXIQUE** 18](#_Toc65888080)

# 

# **DESCRIPTION**

Sans doute que vous aviez eu l’occasion d’apprendre plusieurs langages de programmation, et que vous vous êtes trompez de syntaxe. Trop de langages rend la vie des développeurs moins plaisante c’est pour cela qu’on a décidé de créer un langage universel unique **«ONE\_FOR\_ALL»** compatible avec la plupart des langages de programmation à savoir c, javaScript, pascal , typeScript mais aussi adapté à de nouvelles règles grammaticales .

# **GRAMMAIRE**

S -> S'$

S' -> INSTRUCTIONS

| BLOCK

| epsilon

INSTRUCTIONS -> INSTRUCTION [INSTRUCTIONS | e]

INSTRUCTION -> AFFECTATION ;

| APPEL\_FONCTION ;

| BOUCLE

| RETURN ;

| VAR\_DECLARATION ;

| FONCTION

| CONTROLE

| EXCEPTION

| FILEHANDLING

FILEHANDLING -> print ( PARAMETERS ) ;

| printf ( PARAMETERS ) ;

| scanf ( PARAMETERS ) ;

| print ( PARAMETERS ) ;

| input ( PARAMETERS ) ;

| log ( PARAMETERS ) ;

| fprintf ( PARAMETERS ) ;

| fscanf ( PARAMETERS ) ;

| fread ( PARAMETERS ) ;

| fwrite ( PARAMETERS ) ;

| write ( PARAMETERS ) ;

| read ( PARAMETERS ) ;

| puts ( PARAMETERS ) ;

| gets ( PARAMETERS ) ;

BLOCK -> begin (end | S' end)

AFFECTATION -> const ID AFFECTATION'

| ID AFFECTATION'

AFFECTATION' -> [: [= |e] | = | <-] EXPRESSION

EXPRESSION -> TERM [OPERATEURADD TERM | e]

TERM -> FACTEUR [OPERATEURMULT FACTEUR | OPERATEURSPECIAUX |e]

| + FACTEUR

| - FACTEUR

FACTEUR -> ID

| NUMBER

| ( EXPRESSION )

| BOOLEAN

| APPEL\_FONCTION

| STRING

OPERATEURSPECIAUX -> ++

| --

OPERATEURADD -> +

| add

| -

| minus

OPERATEURMULT -> \*

| mult

| \

| div

| %

| mod

| modulo

APPEL\_FONCTION -> ID ( PARAMETERS ) ;

FONCTION -> TYPE FONCTION'

| function FONCTION'

FONCTION' -> ID(PARAMETERS){ INSTS };

PARAMETERS -> epsilon

| PARAMETER

PARAMETER -> ID TYPE

| PARAMETER , ID TYPE

VAR\_DECLARATION -> VARS'

| epsilon

VARS' -> const TYPE IDS\_CONST

| let JS\_IDS

| VARS''

VARS'' -> ID [: TYPE | is TYPE] OPT [, VARS'' | epsilon]

IDS\_CONST -> ID AFFECTATION' [, IDS\_CONST| e]

JS\_IDS -> ID : TYPE OPT [, JS\_IDS | epsilon]

IDS -> ID OPT [, IDS | epsilon]

OPT -> epsilon

| AFFECTATION'

RETURN -> return EXPRESSION

CONDITION -> BOOLEAN

| EXPRESSION COMPAR EXPRESSION

COMPAR -> >

| <

| >=

| <=

| !=

| ==

###### Controls ######

CONTROL -> IF

| CASE

IF -> if ( CONDITION ) [BLOCK\_IF [elif BLOCK\_IF else BLOCK\_IF | e]

| else BLOCK\_IF]

BLOCK\_IF -> INST

CASE -> switch ( ID ) { BLOCK\_CASE }

BLOCK\_CASE -> case FACTEUR : INSTRUCTIONS [BLOCK\_CASE | e]

###### Boucles ######

BOUCLE -> WHILE

| FOR

| DO\_WHILE

WHILE -> while ( CONDITION ) INST

INST -> INSTRUCTION | { INSTRUCTIONS }

FOR -> for ( INSTRUCTION ; INSTRUCTION ; INSTRUCTION) INST

DO\_WHILE -> do { INSTRUCTIONS } while ( CONDITION );

###### Terminaux ######

TYPE -> string

| number

| int

| boolean

| bool

| char

| integer

| boolean

| void

| float

| double

| signed

| unsigned

| short

ID -> [a-zA-Z]([a-zA-Z0-9\_]\*[a-zA-Z0-9])?

STRING -> " STR "

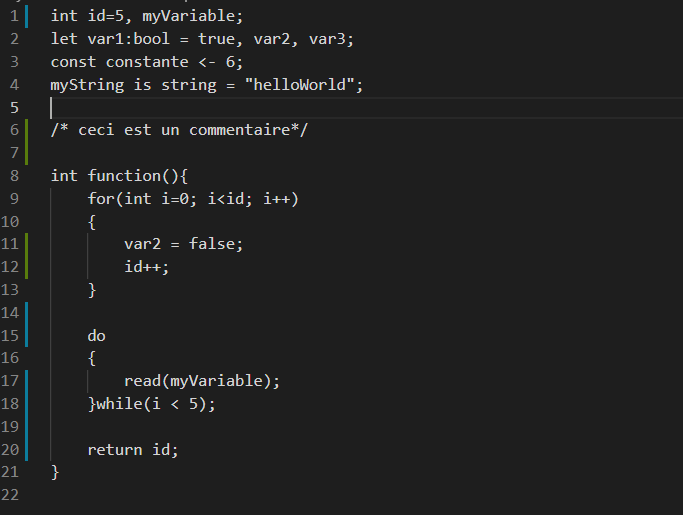
| ' STR '

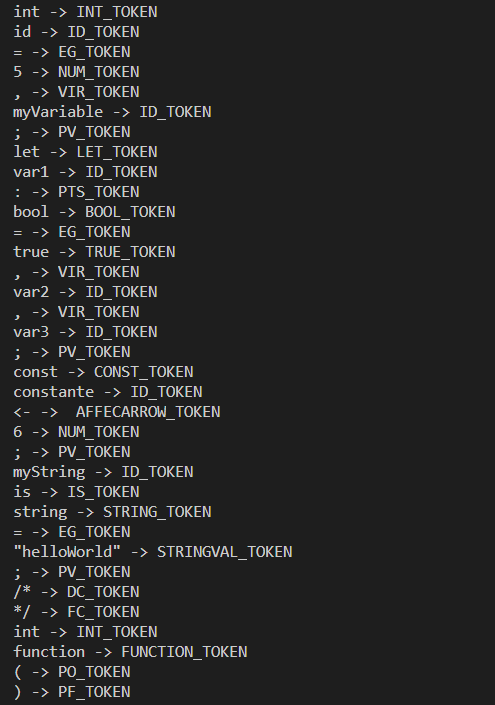
STR -> [ a-zA-Z0-9\_;:!,?\|[](]&~^$µéè@à` ]\*

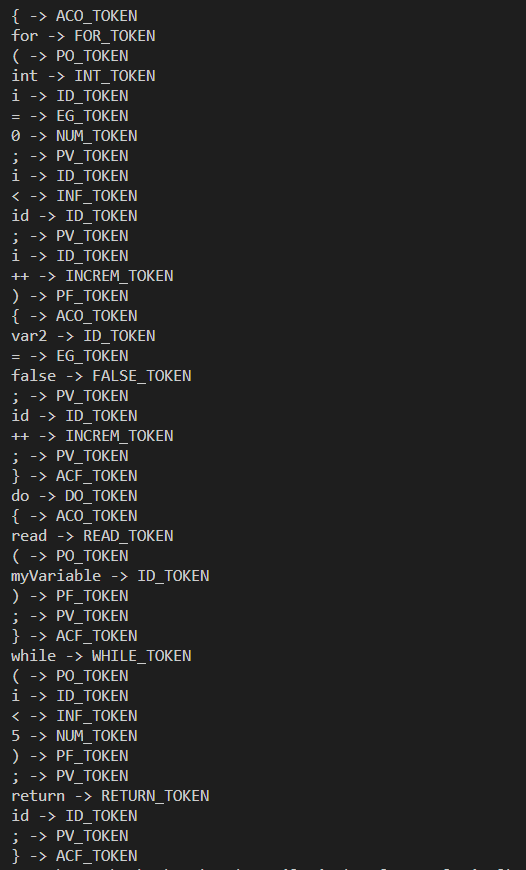
NUMBER -> [0-9]+

BOOLEAN -> true | false

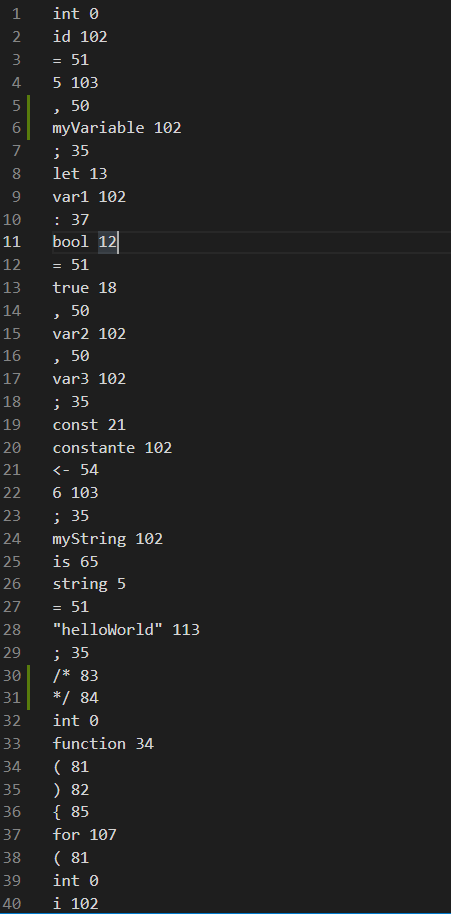
# **ANALYSEUR LEXICAL**

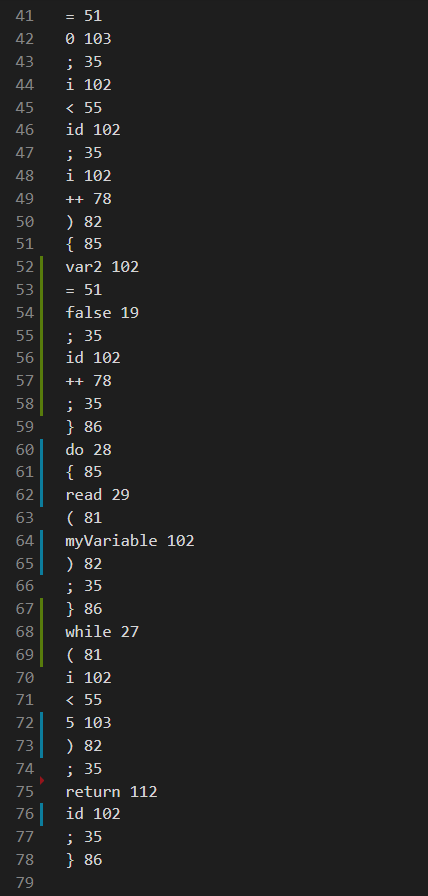
**Fichier source**

**Sortie standard**

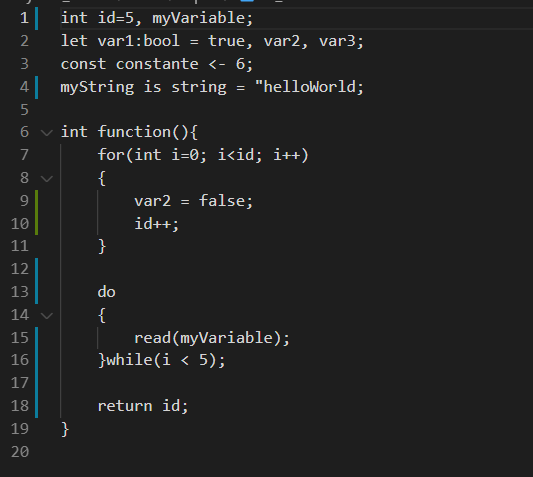


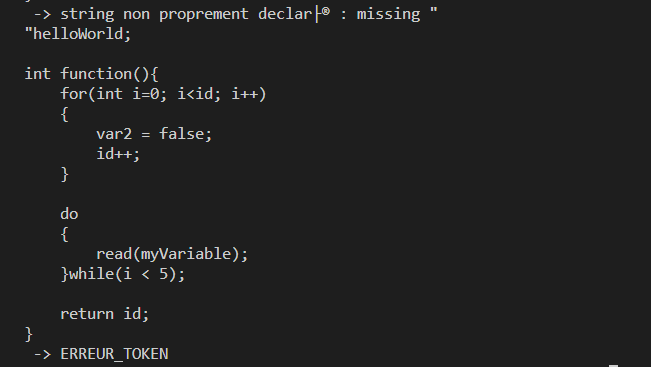
**Fichier sortie de L’analyseur lexical et entrée du syntaxique**

****



**Exemple cas d’erreur « string non proprement déclaré »**

****

****

# **ANALYSEUR SYNTAXIQUE**