



RAPPORT PROJET COMPILATION GL

COMPILATEUR LANGAGE (ONE_FOR_ALL)

Réalisé par:

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

Encadré par:

Professeur TABII Youness

Année Universitaire: 2020-2021

Table des matières

DESCRIPTION	3
GRAMMAIRE	4
ANALYSEUR LEXICAL	11
ANIALVCEUD CVNITAVIOLIE	4.0
ANALYSEUR SYNTAXIQUE	

DESCRIPTION

Vous avez sans doute eu l'occasion d'apprendre plusieurs langages de programmation, et vous vous êtes trompés de syntaxe. Trop de langages rendent 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
                  -> print ( PARAMETERS );
FILEHANDLING
                  | 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]
                          -> FACTEUR [OPERATEURMULT FACTEUR |
TERM
                          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'
                  -> ID(PARAMETERS){ INSTS };
FONCTION'
```

-> epsilon

PARAMETERS

| 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

-> > | < | >= | <= | != | == ##### Controls ##### CONTROL -> IF | CASE -> if (CONDITION) [BLOCK_IF [elif BLOCK_IF else BLOCK_IF | e] IF | else BLOCK_IF] BLOCK_IF -> INST -> switch (ID) { BLOCK_CASE } CASE BLOCK_CASE -> case FACTEUR : INSTRUCTIONS [BLOCK_CASE | e] ##### Boucles ##### BOUCLE -> WHILE | FOR | DO_WHILE

COMPAR

```
WHILE
                          -> while ( CONDITION ) INST
INST
                          -> INSTRUCTION | { INSTRUCTIONS }
                           -> for (INSTRUCTION; INSTRUCTION; INSTRUCTION) INST
FOR
DO_WHILE
                           -> do { INSTRUCTIONS } while ( CONDITION );
                      ##### Terminaux ######
TYPE
                           -> string
                         | number
                         | int
                         | boolean
                         bool
                         | char
                         | integer
                         | boolean
                         | void
                         | float
                         | double
                         signed
                         | unsigned
                         | short
                          -> [a-zA-Z]([a-zA-Z0-9_]*[a-zA-Z0-9])?
ID
STRING
                           -> " STR "
```

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

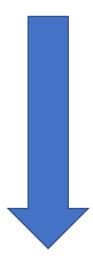
NUMBER -> [0-9]+

BOOLEAN -> true | false

ANALYSEUR LEXICAL

Fichier source

```
int id=5, myVariable;
     let var1:bool = true, var2, var3;
     const constante <- 6;</pre>
     myString is string = "helloWorld";
 6
     /* ceci est un commentaire*/
 7
     int function(){
         for(int i=0; i<id; i++)
10
11
             var2 = false;
12
             id++;
13
         }
14
15
         do
16
17
             read(myVariable);
         }while(i < 5);
18
19
20
         return id;
21
22
```



Sortie standard

```
int -> INT_TOKEN
id -> ID TOKEN
= -> EG TOKEN
5 -> NUM TOKEN
, -> VIR_TOKEN
myVariable -> ID_TOKEN
; -> PV_TOKEN
let -> LET TOKEN
var1 -> ID_TOKEN
: -> PTS_TOKEN
bool -> BOOL TOKEN
= -> EG_TOKEN
true -> TRUE_TOKEN
, -> VIR_TOKEN
var2 -> ID_TOKEN
, -> VIR_TOKEN
var3 -> ID_TOKEN
; -> PV_TOKEN
const -> CONST_TOKEN
constante -> ID_TOKEN
<- -> AFFECARROW_TOKEN
6 -> NUM_TOKEN
; -> PV_TOKEN
myString -> ID_TOKEN
is -> IS_TOKEN
string -> STRING_TOKEN
= -> EG_TOKEN
"helloWorld" -> STRINGVAL_TOKEN
; -> PV_TOKEN
/* -> DC_TOKEN
*/ -> FC TOKEN
int -> INT_TOKEN
function -> FUNCTION_TOKEN
( -> PO TOKEN
) -> PF_TOKEN
```

```
{ -> ACO_TOKEN
for -> FOR_TOKEN
( -> PO_TOKEN
int -> INT_TOKEN
i -> ID_TOKEN
= -> EG_TOKEN
0 -> NUM_TOKEN
; -> PV_TOKEN
i -> ID_TOKEN
< -> INF_TOKEN
id -> ID_TOKEN
; -> PV_TOKEN
i -> ID TOKEN
++ -> INCREM_TOKEN
) -> PF_TOKEN
{ -> ACO TOKEN
var2 -> ID_TOKEN
= -> EG_TOKEN
false -> FALSE TOKEN
; -> PV_TOKEN
id -> ID_TOKEN
++ -> INCREM TOKEN
; -> PV_TOKEN
} -> ACF TOKEN
do -> DO TOKEN
{ -> ACO_TOKEN
read -> READ TOKEN
( -> PO TOKEN
myVariable -> ID_TOKEN
) -> PF TOKEN
; -> PV_TOKEN
} -> ACF_TOKEN
while -> WHILE TOKEN
( -> PO_TOKEN
i -> ID TOKEN
< -> INF TOKEN
5 -> NUM_TOKEN
) -> PF_TOKEN
; -> PV_TOKEN
return -> RETURN_TOKEN
id -> ID_TOKEN
; -> PV_TOKEN
} -> ACF_TOKEN
```

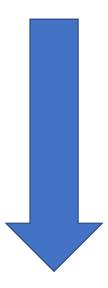
Fichier sortie de L'analyseur lexical et entrée du syntaxique

```
int 0
    id 102
    = 51
    5 103
    , 50
5
    myVariable 102
6
    ; 35
    let 13
    var1 102
    : 37
   bool 12
11
12
   = 51
13
    true 18
    , 50
14
15
    var2 102
    , 50
17
    var3 102
    ; 35
19 const 21
   constante 102
21
   <- 54
22
    6 103
    ; 35
   myString 102
24
   is 65
26
   string 5
27
    = 51
    "helloWorld" 113
    ; 35
30
    /* 83
31
    */ 84
32
    int 0
    function 34
    (81
    ) 82
36
    { 85
    for 107
38
    (81
    int 0
40
    i 102
```

```
= 51
41
42
     0 103
    ; 35
43
    i 102
44
45
    < 55
    id 102
    ; 35
47
    i 102
    ++ 78
     ) 82
51
    { 85
52
    var2 102
53
    = 51
54
    false 19
    ; 35
55
56
    id 102
     ++ 78
57
58
     ; 35
59
     } 86
60
     do 28
61
     { 85
62
     read 29
     (81
64
     myVariable 102
    ) 82
65
    ; 35
    } 86
67
68
     while 27
     (81
69
     i 102
70
71
     < 55
72
     5 103
     ) 82
73
    ; 35
74
75
     return 112
76
    id 102
     ; 35
77
     } 86
79
```

Exemple cas d'erreur « string non proprement déclaré »

```
1
     int id=5, myVariable;
     let var1:bool = true, var2, var3;
     const constante <- 6;</pre>
     myString is string ∈ "helloWorld;
 4
6 ∨ int function(){
         for(int i=0; i<id; i++)
             var2 = false;
9
10
             id++;
11
12
13
         do
14
             read(myVariable);
15
         \}while(i < 5);
17
         return id;
18
19
20
```



```
-> string non proprement declar * missing "
"helloWorld;

int function(){
	for(int i=0; i<id; i++)
	{
		var2 = false;
		id++;
	}

do
	{
		read(myVariable);
	}while(i < 5);
	return id;
}
-> ERREUR_TOKEN
```

ANALYSEUR SYNTAXIQUE