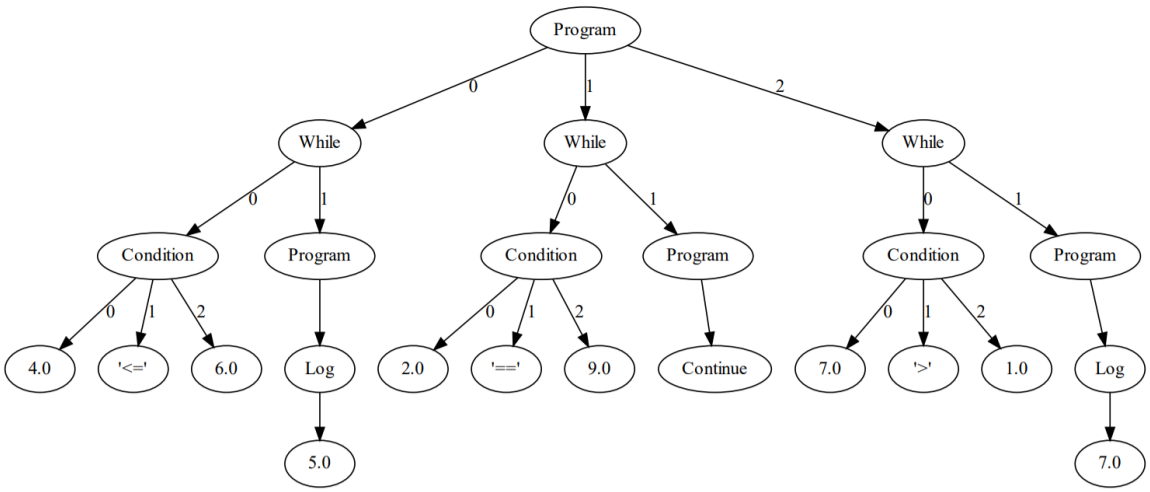
haute école
neuchâtel berne jura

ingénierie
www.he-arc.ch

Compilateur JS

Rapport JSCompilator

INF3 DLM-b



Paysant Adrien - Monnet Joris

2020/2021

TABLE DES MATIERES

INTRODUCTION

JSCompilator est le Terminator de tous les compilers Javascript ! Notre projet qui vous est présenté ci-dessous à pour but de construire de a à z un compilateur de javascript en python, à l'aide de yacc/PLY pour le lexer/parser et d'une machine à pile pour la partie arrière. Ce projet est réalisé dans le cadre du cours de compilateur de la HE-ARC.

OBJECTIFS

LANGAGE

Nous avons décidé de créer notre compilateur en javascript pour diverses raisons. Nous voulions repartir du TP4 et pour ce faire, il nous fallait un langage relativement similaire au C. De plus le javascript est un langage que nous utilisons depuis l'an dernier sur différents cours ce qui nous permet de bien connaitre sa syntaxe et ses fonctionnalités.

BUT

Le but de notre projet est d'avoir un compilateur qui reconnait la majorité des mots réservées et des fonctionnalités hors javascript orientée objet. Pour ce faire, nous avons mis en place ce cahier des charges d'objectifs :

OBJECTIFS PRINCIPAUX

- Reprise du TP4
- Ajout du if/else
- Ajout du for
- Ajout du switch/case/default
- Ajout du do (pour le do while)
- Ajout du var/let
- Ajout des règles permissives pour le point-virgule

OBJECTIFS SECONDAIRES

- Ajout des fonctions et du mot clé function
- Vérifier le nombre d'arguments (analyse sémantique)
- Ajout du mot clé return
- Ajout du break/continue
- Ajout des tableaux avec les crochets []

FONCTIONNALITES IMPLEMENTEES

REGLES POUR LES ACCOLADES

Malgré le fait que ce ne soit pas explicitement écrit dans les objectifs, nous avons fait le choix de donner la possibilité d'utiliser une syntaxe avec accolades (nommé programBlock dans le parser) :

```
def p_program_block(p):
    ''' programBlock : '{' new_scope program '}' '''
    p[0] = p[3]
    popscope()
```

Pour cette syntaxe :

```
if(2<5){
    log(2)
} else if (1==5){
    log(3)
} else {
    log(4)
}
```

Mais aussi une syntaxe sans accolades pour les structures (pas les fonctions) nommé programStatement dans le parser étant donné que sans accolades, une seule instruction (statement) est autorisée :

```
def p_program_statement(p):
    '''programStatement : statement'''
    p[0] = AST.ProgramNode([p[1]])
```

Pour cette syntaxe :

```
if(2<5)
    log(2)
else if (1==5)
    log(3)
else
    log(4)
```

Ainsi que des mélanges des deux syntaxes (pas conseillé car illisible mais possible en javascript) :

```
if(2<5){
    log(2)
} else if (1==5)
    log(3)
else {
    log(4)
}
```

De ce fait, notre parser donne les possibilités complètes de javascript.

REPRISE DU TP4

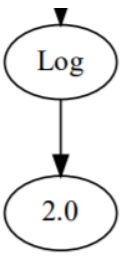
Notre premier objectif était donc de reprendre le TP4, nous avons gardé les calculs sur les expressions : ADD_OP, MUL_OP, UMINUS puis en avons ajouté d'autres, récupéré le while et changé le print en log.

LOG

En effet, en javascript pour afficher quelque chose, il faut mettre console.log(), étant donné que nous n'implémentions pas la partie objet de javascript, le print a simplement été remplacé par log. C'est la seule différence entre notre langage et du javascript. Dans le lexer, on a rajouté le mot clé, et ci-dessous l'implémentation dans le parser :

```
def p_log(p):
    ''' logStatement : LOG '(' returnValues ')' '''
    p[0] = AST.LogNode(p[3])
```

Ici les return values correspondent à ce que peut return une fonction, c'est-à-dire une expression comme une variable ou un calcul, un tableau, ou encore une fonction. Le résultat dans l'arbre : pour un log(2) :



WHILE

Nous avons repris le while du TP4 en modifiant deux choses. Tout d'abord le while comme le if, contient une condition, présentées ci-dessous. De surcroit, nous rajoutons la possibilité du while sans accolades. Le parser :

```
def p_while(p):
    ''' structure : WHILE '(' condition ')' programStatement
    | WHILE '(' condition ')' programBlock'''
    p[0] = AST.WhileNode([p[3],p[5]])
```

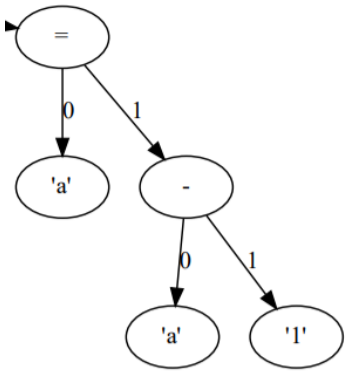
RACCOURCI D'OPERATIONS POUR LES EXPRESSIONS

En plus des opérations déjà en place, nous avons permis les raccourcis quand on assigne une variable, par exemple a*=2 ou a++ qui correspondent à a = a*2 et a=a+1.

```
def p_expression_op_assignation(p):
    '''assignation : IDENTIFIER ADD_OP '=' expression
    | IDENTIFIER MUL_OP '=' expression
    | IDENTIFIER ADD_OP '=' functionCall
    | IDENTIFIER MUL_OP '=' functionCall'''
```

```
def p_expression_op_assign_double(p):
    '''assignation : IDENTIFIER ADD_OP ADD_OP'''
```

Ces raccourcis sont représentés dans l'arbre comme s'ils avaient été écrits sans raccourcis :



Correspond à a--

CONDITIONS

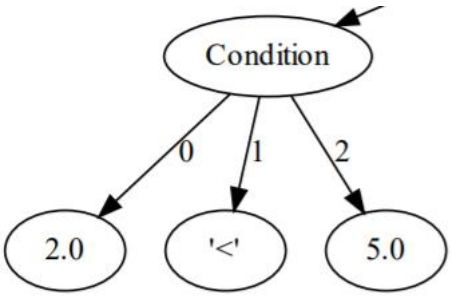
Il semblait nécessaire de mettre en place des conditions, notamment pour le while, le if et le for. Dans le lexer :

```
t_LT = r'<'
t_GT = r'>'
t_LTE = r'<='
t_GTE = r'>='
t_EQUALVT = r'==='
t_NOTEQUALVT = r'!=='
t_EQUALV = r'=='
t_NOTEQUALV = r'!='
t_AND=r'&&'
t_OR=r'\|\|'
```

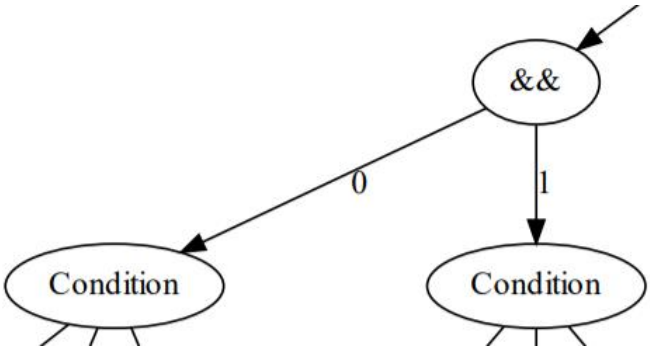
On accepte maintenant tous les opérateurs de comparaison et les opérateurs AND et OR. De plus, nous avons ajouté ! dans les literals afin de permettre l'opérateur NOT. Au niveau du parser, les operateurs de comparaison son non-associatif pour ne pas pouvoir les chainer (a<b<c) et le comparateur ! est prioritaire au && qui est prioritaire au | | comme en javascript :

```
precedence = (
    ('left','NEWLINE','ELSE','OR','IDENTIFIER',',',';'),
    ('nonassoc', 'LT','GT','EQUALV','EQUALVT','NOTEQUALV','NOTEQUALVT', 'LTE','GTE'),
    ('left','AND'),
    ('left', 'ADD_OP'),
    ('left', 'MUL_OP'),
    ('right', 'UMINUS','!')
)
```

Une condition se présente comme ceci :



Nous avons aussi ajouté des AND, OR et NOT nodes pour les différents opérateurs entre conditions :



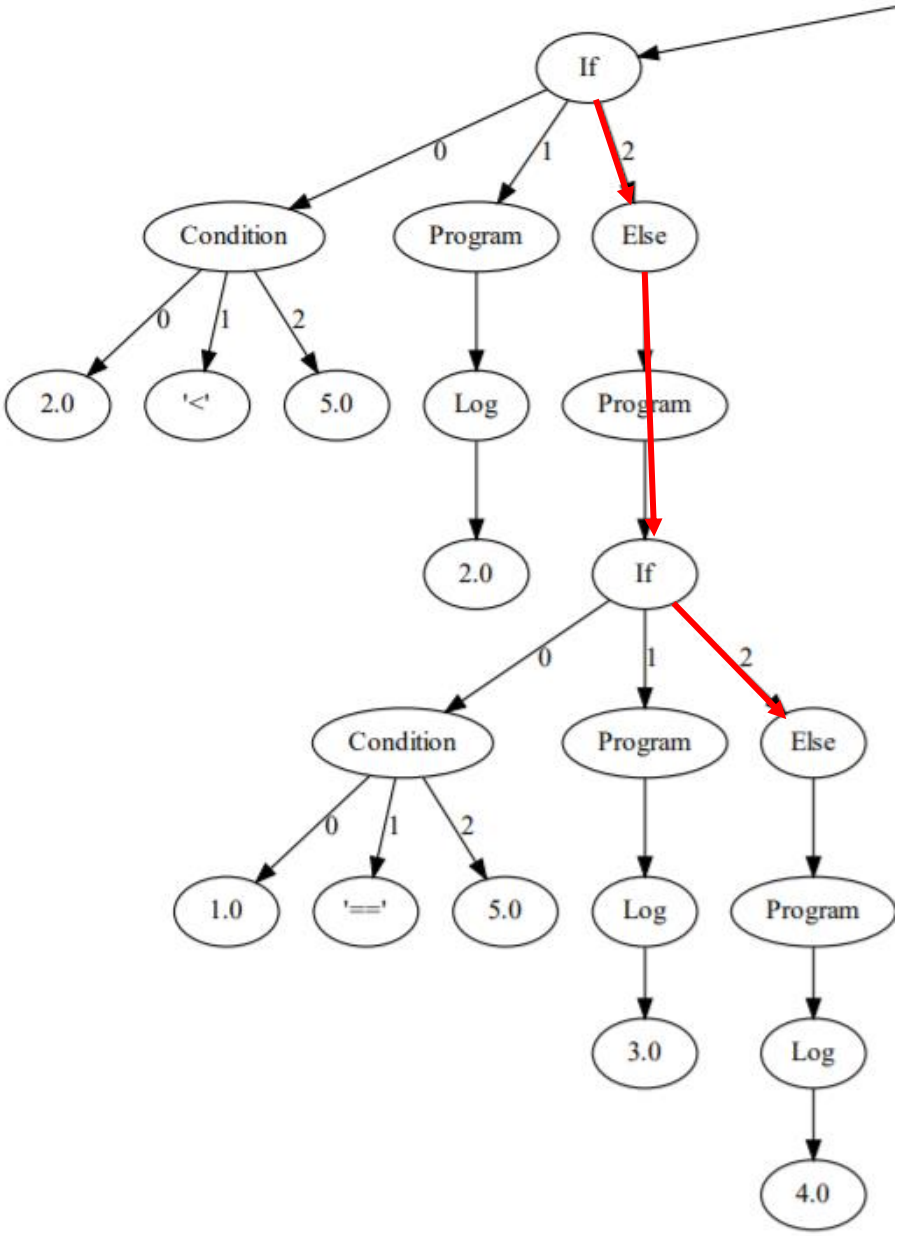
Les conditions font partis des éléments qui ont été implémentées dans la partie arrière. Leur calcul se réalise dans la svm.

IF/ELSE

Dans le lexer, nous avons ajouté ces deux mot clés ainsi que les caractères ? et : pour l'opérateur ternaire. Dans le parser, le if fonctionne avec ou sans accolades. La subtilité et la difficulté dans cette fonctionnalité réside dans le fait que les if/else sont récursifs dans le cas de if /else if /else par exemple. Nous avons fait le choix dans l'affichage qu'un if ait deux ou trois nœuds enfants. Le premier est sa condition. Le deuxième est son programme (le then). Le troisième qui est facultatif dans le cas d'un if seul est le else. Ce dernier a pour enfant un programme. Dans le cas d'un else if, le if est dans le programme du else. Un exemple étant plus parlant, en voici un avec :

```
if(2<5){
    log(2)
} else if (1==5){
    log(3)
} else {
    log(4)
}
```

Qui donne :

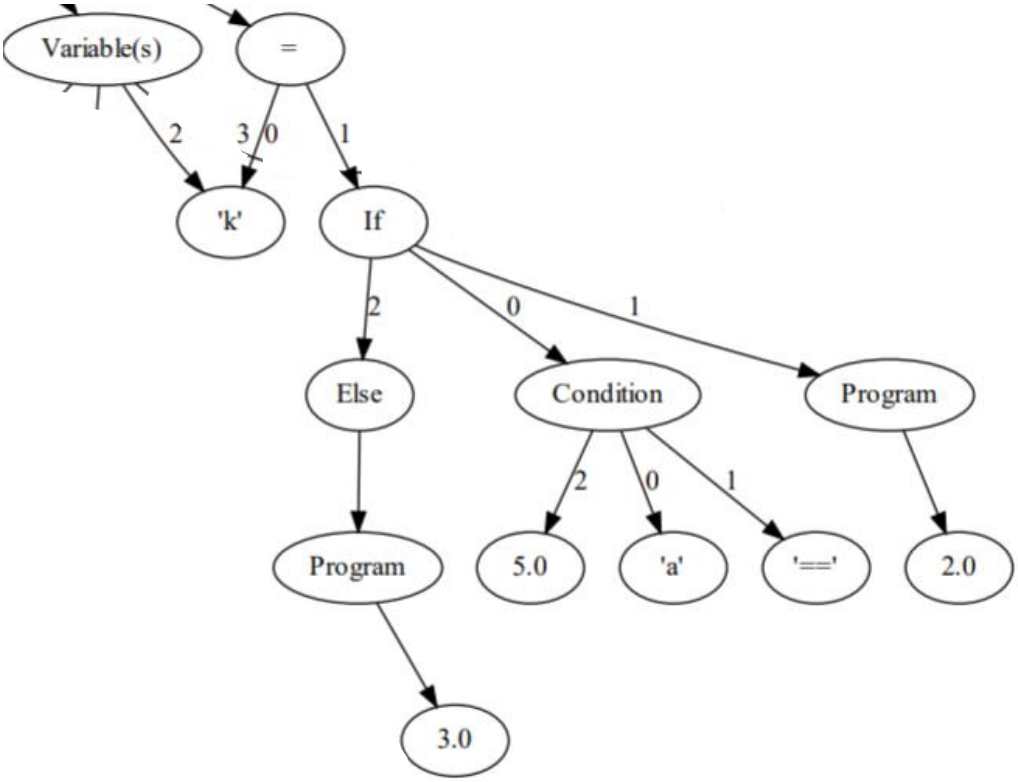


Ici en rouge le chemin if/else if /else

Cette récursion fut source de nombre de bugs, notamment dans la version sans accolades, ceux-ci seront détaillés plus loin. En partie arrière, le if fonctionne mais pas le else, c'est donc une fonctionnalité à améliorer.

OPERATEUR TERNAIRE

L'opérateur ternaire comme pour les raccourcis avec les opérations et les expressions n'est qu'un raccourci. En effet, il permet d'utiliser cette notation mais ne diffère dans sa construction en arbre en rien d'un if/else. Par exemple `var k = (a==5)?2:3;` donne :

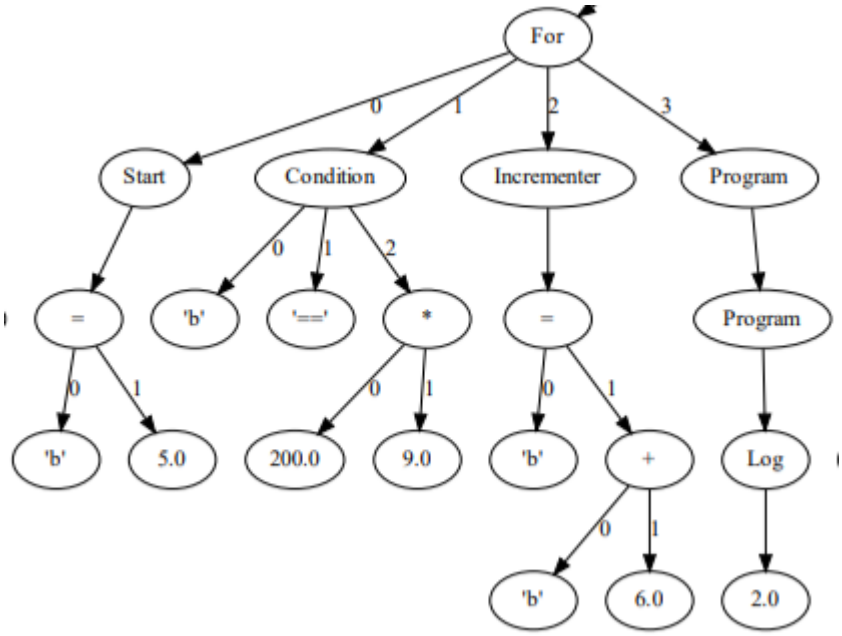


FOR

Dans la partie lexer, le mot clé est simplement rajouté. Dans le cadre de ce projet, nous avons implémenter le for classique :

```
def p_for(p):
    '''structure : FOR new_scope '(' assignation ';' condition ';' assignation ')' programBlock
    | FOR new_scope '(' assignation ';' condition ';' assignation ')' programStatement '''
```

Il est composé d'une assignation de départ (startForNode dans AST), d'une condition de fin, et d'un incrementer, puis de son programme. Les enfants de la ForNode sont donc ces quatre nœuds. Cela se traduit par ceci en arbre :



Pour le fichier en entrée :

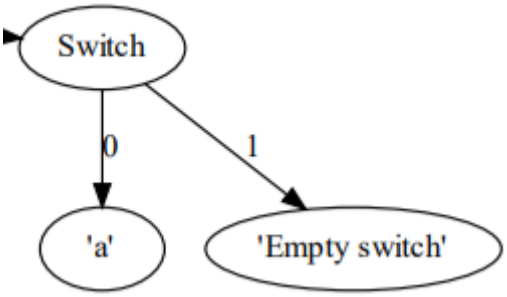
```
for(let b = 5;b==200*9;b+=6)
{
    log(2)
}
```

SWITCH

Pour cette fonctionnalité, nous avons ajouté les mots réservés dans le lexer, à savoir SWITCH, CASE et DEFAULT. Ensuite dans le parser, il s'agit d'une des structures les plus complexes. En effet, on l'oublie, mais un switch peut être vide :

```
def p_switch_void(p):
    ''' structure : SWITCH '(' IDENTIFIER ')' '{' '}' '''
```

Ce qui donne :

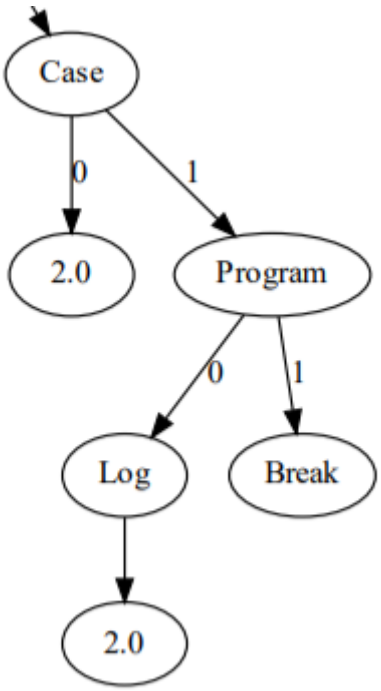


Sinon, il est composé de CASE et/ou d'un DEFAULT :

```
def p_switch(p):
    '''structure : SWITCH '(' IDENTIFIER ')' '{' new_scope caseList '}' '''
```

Dans les deux cas on vérifie si le IDENTIFIER est bien une variable déclarée.

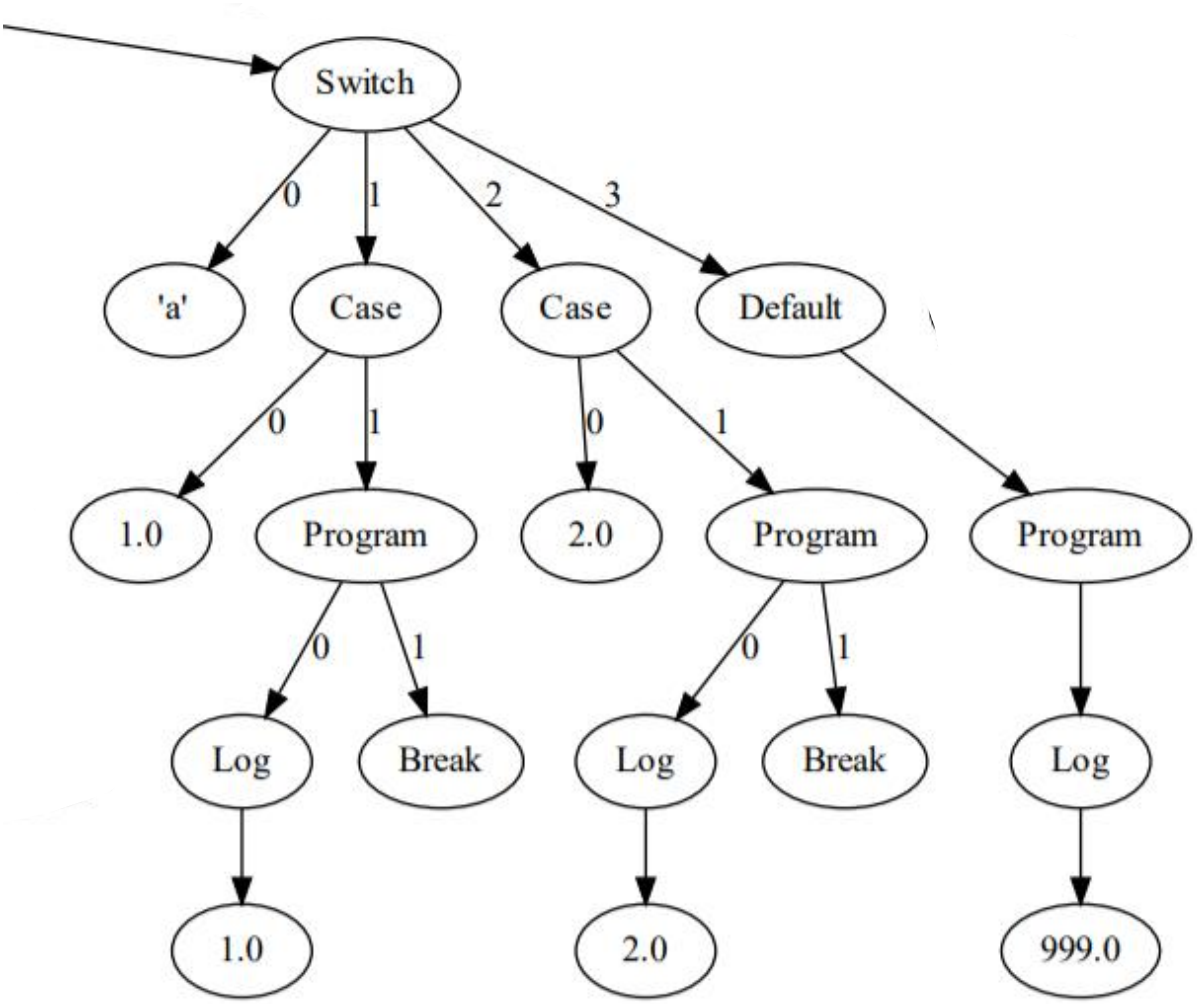
La caseList est une suite de CASE et de DEFAULT (sans limite de nombre). Chaque case possède un nombre et un programme associé :



Un default ne possède qu'un programme. Comme indiqué plus haut, il n'y a pas de limite de default dans notre parser. On passe donc par une analyse sémantique dans le AST pour vérifier qu'il y ait un seul ou aucun default dans chaque SwitchNode :

```
def verifyDefault(self):
    return len([child for child in self.children if child.type == 'Default']) < 2
```

Résultat total d'un switch :



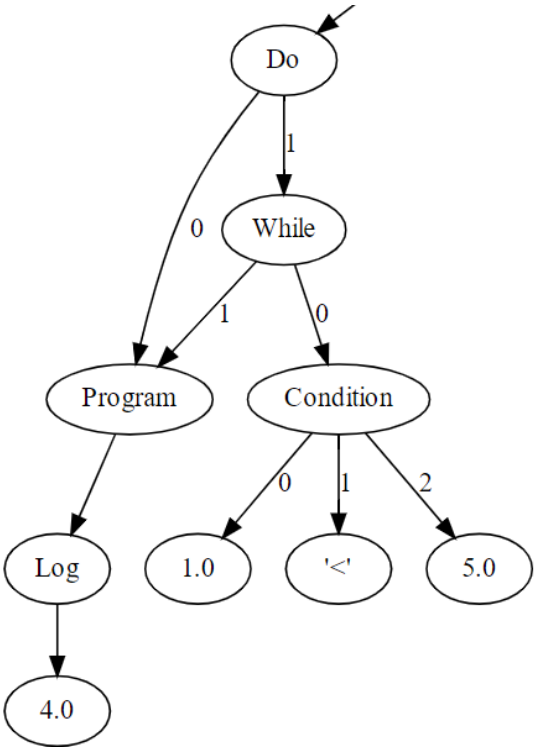
Le switch est mis en place dans la partie arrière.

DO WHILE

Le « do » est ajouté au lexer et ensuite on autorise deux syntaxes dans le parser : avec ou sans accolades. Ici la syntaxe avec ces dernières :

```
def p_do_while(p):
    '''structure : DO programBlock WHILE '(' condition ')' '''
```

L'arbre en sorti est un while normal avec simplement un nœud Do qui va directement au programme du nœud pour la première itération, avant que le while ne soit lancé :



Le do while, comme le while ont été implémenté dans la partie arrière.

VAR ET LET

En javascript, on peut déclarer les variables via deux mots clés, var et let, ils sont utilisés indifféremment dans notre programme et ajouté aux mot clés réservés dans notre lexer. Dans le parser, ils sont utilisés dans différentes syntaxes. Ils servent à créer une variable ou plusieurs, par exemple var a ou let a,b. On utilise donc une liste de variable récursive :

```
def p_var_creation_list_recursive(p):
    '''varList : varList ',' IDENTIFIER'''
```

Il est possible d'assigner aux variables des valeurs numériques, comme dans le TP4, mais aussi des tableaux de valeurs et des valeurs données par des fonctions (on verra cela plus en détail dans les chapitres consacrés à cela).

PORTEE DES VARIABLES

Les variables demandent une analyse sémantique. En effet, les variables sont soumises à une portée. A la différence du réel javascript, notre langage ne prend pas en compte ceci :

```
Let i=2 ;
If(1<5){
    Let i =1 ;
    Console.log(i)
}
```

Ici, en javascript, cela afficherait 1, pour nous une erreur est lancée car l'utilisateur redéfinit i. Il est donc nécessaire d'avoir un nom de variable différent dans ce cas-ci. En dehors de cela nous avons mis en place comme en javascript, l'impossibilité d'appeler une variable dans portée plus large que celle où elle fut créée. Pour ce faire nous utilisons une fonctionnalité intéressante que vous nous a proposé venant de https://www.dabeaz.com/ply/ply.html (chapitre 6.11) où nous utilisons ceci :

```
def p_new_scope(p):
    '''new_scope : '''
    listScope.append(Scope())
```

Dans les structures ou nous changeons de portée, nous mettons cet identifiant « new_scope ». A chaque fois qu'il est rencontré, il mène à cette fonction qui ajoute un nouvel objet Scope à une liste de portées. Cet objet scope contient la liste de tous les noms de variables (et de nom de fonctions) des scopes l'englobant. Une fois terminé on appelle la fonction popScope pour enlever le scope terminé, c'est-à-dire a chaque fin de blocs d'instructions entre accolades. Ensuite, à chaque fois qu'on crée une nouvelle variable, on vérifie qu'elle n'existe pas déjà, et on l'ajoute à la liste du scope en cours. Exemple avec un programBlock :

```
def p_program_block(p):
    ''' programBlock : '{' new_scope program '}' '''
    p[0] = p[3]
    popscope()
```

Exemple de création de variable :

```
def p_var_creation(p):
    '''varCreation : VAR IDENTIFIER
    | LET IDENTIFIER'''
    if p[2] not in listScope[-1 if len(listScope)>1 else 0].vars:
        listScope[-1 if len(listScope)>1 else 0].vars.append(p[2])
        p[0] = AST.VariableNode([AST.TokenNode(p[2])])
    else :
        error = True
        print(f"ERROR : {p[2]} is already declared")
```

REGLES PERMISSIVES POUR LE POINT-VIRGULE

Une autre difficulté de javascript est que le ; est facultatif mais permet de mettre deux instructions sur la même ligne. Dans le TP4, nous avions émis une solution avec l'utilisation de points-virgules elle souffrait de deux problèmes. D'abord, la dernière instruction du

programme ne pouvait terminer par un point-virgule la solution la plus simple est donc de le supprimer avant de parser le programme :

```
if program[-1]==';': #allow to finish with a ;
    program=program[:-1]
```

Ensuite, dans le même style, on ne pouvait pas finir par un point-virgule dans un block d'instructions comme dans le if ou dans le while. Ici la solution est la même : supprimer ce dernier point-virgule à chaque fois avant de parser. Problème, entre ce point-virgule et l'accolades, il peut y a voir des espaces, et pour notre gestion d'erreur qui indique le numéro de ligne de l'erreur on ne peut pas modifier le nombre de retour à la ligne par rapport au fichier original. Nous utilisons donc cette solution :

```
return yacc.parse(re.sub(r";(\n)+}",lambda x : "\n"*(len(x.group())-2)+"}",program)+"\n")
```

Ici, nous allons expliquer cette ligne pas à pas. En premier lieu, il s'agit de notre dernière ligne de la fonction parser, elle retourne donc notre fichier parser par yacc. En revanche il se passe deux opérations sur le fichier à parser. La plus simple et l'ajout de \n en toute fin de ligne, en effet, si notre programme finit sans nouvelle ligne, le parser ne fonctionne pas (comme dans le TP4 si on finissait avec un point-virgule). Ensuite, la suppression des dernier points virgules dans chaque structure :

```
re.sub(r";(\n)+}",lambda x : "\n"*(len(x.group())-2)+"}",program)
```

Nous utilisons re.sub() qui permet de remplacer une expression régulière avec une fonction lambda. Nous recherchons avec la regex tous les points virgules séparés que par des retours à la ligne d'une accolade fermante. Nous remplaçons chacun de ces expressions par le même nombre de retour à la ligne et l'accolade fermante, c'est-à-dire que nous n'enlevons que le point-virgule. Ainsi, nous réussissons à autoriser à finir avec un point-virgule n'importe où, sans influer sur notre gestion des erreurs.

Maintenant que le point-virgule fonctionne pour tous les cas de figures, il reste les expressions sans point-virgule. Pour ceci nous avons redéfini NEWLINE dans le lexer :

```
def t_NEWLINE(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
    return t
```

Et l'utilisons dans le parser notamment avec les programmes :

```
def p_program(p):
    ''' program : statement  ';'
    | statement NEWLINE '''
    p[0] = AST.ProgramNode(p[1])

def p_program_recursive(p):
    ''' program : statement ';' program
    | statement NEWLINE program '''
    p[0] = AST.ProgramNode([p[1]]+p[3].children)
```

BREAK ET CONTINUE

Dans le lexer nous avons rajouté ces deux mots clés. Ensuite nous les avons mis en place dans le parser : continue et break sont des mots à utiliser seuls, on crée des nodes AST pour eux et c'est fini :

```
def p_break(p):
    ''' breakStatement : BREAK '''
    p[0] = AST.BreakNode()

def p_continue(p):
    ''' continueStatement : CONTINUE '''
    p[0] = AST.ContinueNode()
```

Malheureusement, ce n'est pas si simple. En effet, syntaxiquement ce sera juste, mais sémantiquement, un break et un continue sont forcément dans des boucles (ou switch pour le break).

Il est de ce fait nécessaire d'avoir une analyse sémantique. Or, pour vérifier si dans les nœuds parents il y a une boucle, il faut parcourir un arbre finit. De ce fait, nous lançons cette vérification dans le main : `if AST.verifyNode():` qui appelle la fonction qui lance `verifyBreakContinueNode()`.

Cette dernière, débute par créer une liste de tous les nœuds Break, puis une de tous les nœuds continue, puis une de tous les nœuds de boucle eu une de tous les switches. Ensuite on vérifie si tous les nœuds continue sont bien dans les enfants des boucles :

```
for continueNode in continueNodes:
    nodeVerified = False
    for loopNode in loopNodesToCheck:
        if checkInChildren(loopNode,continueNode):
            nodeVerified = True
    if not nodeVerified:
        print("ERROR : Continue Node outside of a loop")
        return False
```

Puis si tous les nœuds break sont bien dans des boucles ou des switchs :

```
nodesToCheck = loopNodesToCheck+switchNodes

for breakNode in breakNodes:
    nodeVerified = False
    for nodeToCheck in nodesToCheck:
        if checkInChildren(nodeToCheck,breakNode):
            nodeVerified = True
    if not nodeVerified:
        print("ERROR : Break Node outside of a loop or switch")
        return False
```

Dans le cas contraire des messages d'erreur s'affichent et le PDF n'est pas généré.

Le continue fonctionne dans la partie arrière, il correspond à un JMP vers la dernière condition mais le break n'as pas pu être mis en place.

TABLEAUX

Nous avons mis en place deux types de tableaux, des tableaux de nombre : [2,3,5,6,4] et des tableaux d'identifiants, pour mimer un tableau de strings partiellement : [a,r,t,e]. Un tableau peut être vide, ou remplit :

```
def p_array_empty(p):
    '''arrayDeclaration : '[' ']' '''
```

```
def p_array_declaration(p):
    '''arrayDeclaration : '[' tokenList ']' '''
```

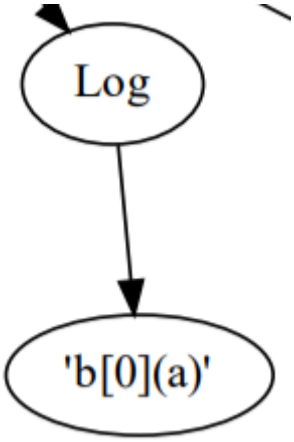
Ensuite on peut appeler un élément du tableau :

```
def p_array_access(p):
    ''' expression : IDENTIFIER '[' NUMBER ']' '''
```

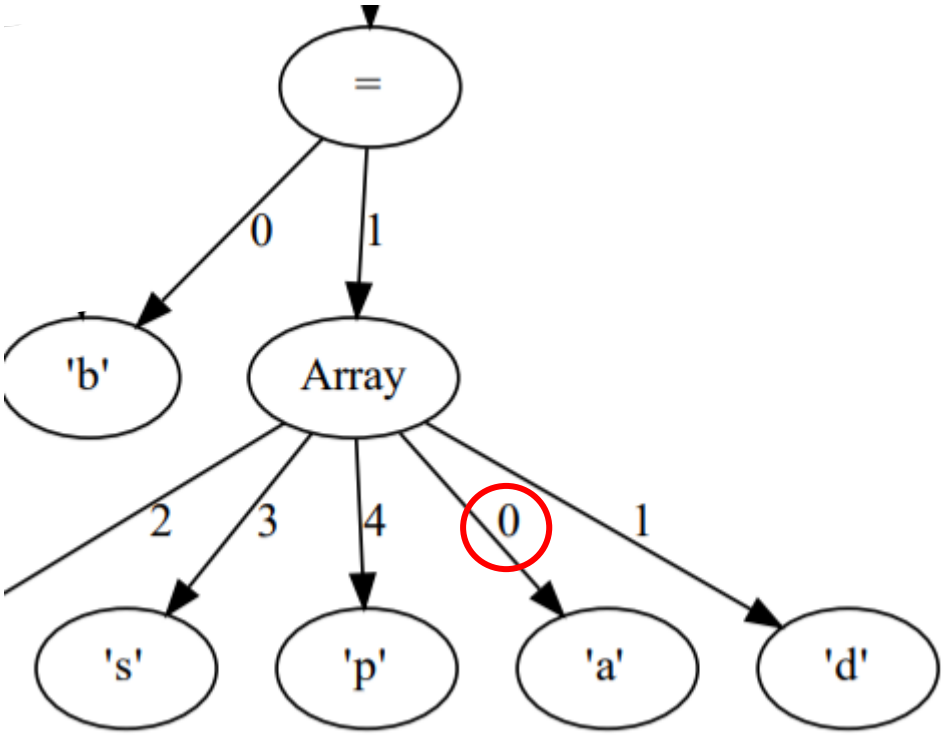
Il y a ici une analyse sémantique, on vérifie que le tableau est déclaré avant de l'appelé, que le nombre entre croché est un entier, et que cet index ne soit pas plus grand que la taille du tableau. Pour ce faire, nous récupérons le nœud correspondant au tableau dans AST :

```
def getArrayNodeById(id):
```

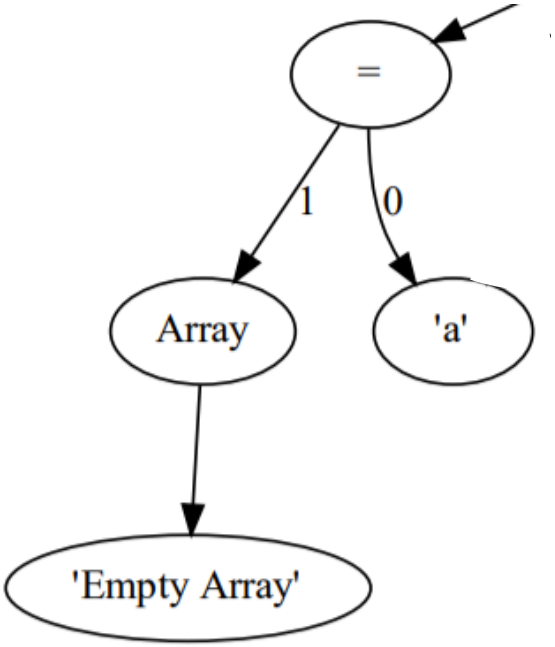
Cela nous permet en plus de récupérer la valeur de cet appel, ce qui donne :



Ici, b[0] vaut a, ce qui est juste au vu de la déclaration de b :



Dans le cas d'un tableau vide :



FONCTIONS

Les fonctions étaient l'objectif majeur dans les objectifs secondaires. Dans le lexer, nous avons ajouté le mot clé function. Dans le parser, d'abord, ce fut la déclaration qui fut mise en place, avec arguments :

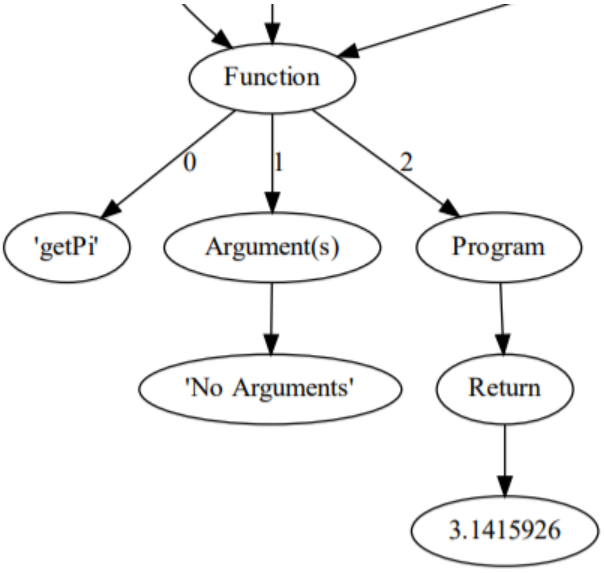
```
def p_function_creation(p):
    '''functionDeclaration : FUNCTION IDENTIFIER '(' new_scope argList ')' programBlock'''
```

Ou sans arguments :

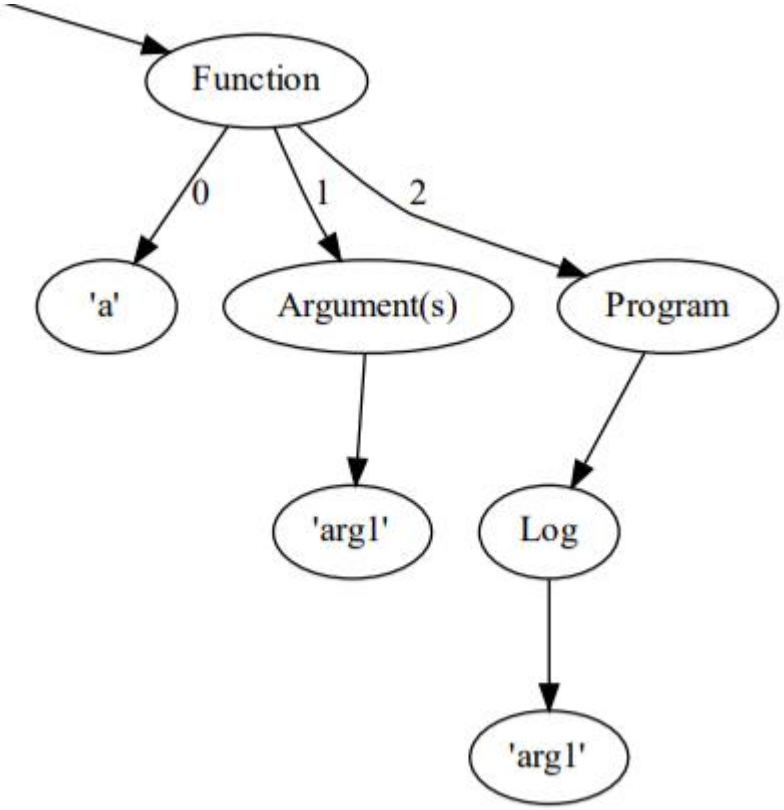
```
def p_function_creation_without_arg(p):
    '''functionDeclaration : FUNCTION IDENTIFIER '(' ')' programBlock'''
```

En javascript une fonction n'a pas de type de retour et ses arguments n'ont pas de types. Deux fonctions ne peuvent pas avoir le même nom exactement comme les variables mais ne sont accessibles que dans leur scope, comme les variables de nouveau. A chaque déclaration on vérifie donc que la fonction n'existe pas déjà. argList est une suite récursive d'arguments(IDENTIFIER) séparé pas des virgules.

La déclaration d'une fonction sans arguments donne :



Et avec arguments :



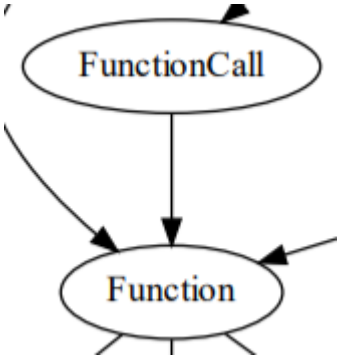
Ensuite, une fonction peut être appelée sans arguments :

```
def p_function_call_withous_args(p):
    '''functionCall : IDENTIFIER '(' ')' '''
```

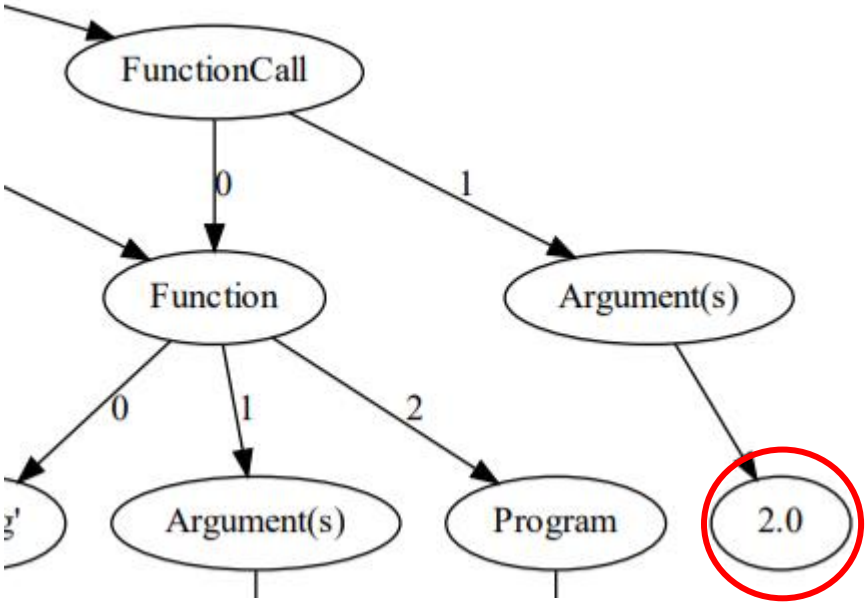
Ou avec arguments :

```
def p_function_call(p):
    '''functionCall : IDENTIFIER '(' expressionList ')' '''
```

Dans l'arbre cela donne une functionCallNode qui est parente du nœud fonction créé à la déclaration : sans arguments :



Ou avec arguments :



Concernant la partie arrière, la gestion des fonctions n'est pas totalement implémentée, et donc non-fonctionnelle. Cependant certains avancements peuvent être constater, à l'instar du traitement de l'appel des fonctions, de la gestion du return dans une fonction. Le principal frein à la réalisation de cette partie est le mécanisme de fin de la function. En effet, un nouveau mot clé (« EF ») a été rajouté dans le svm dans le but de baliser la fin d'un bloc d'exécution d'une fonction ; mais l'implémentation du EF dans le svm n'est pas fonctionnelle.

ANALYSE SEMANTIQUE DU NOMBRE D'ARGUMENTS

Comme vu plus haut, on peut appeler n'importe quelle fonction avec ou sans arguments, or en javascript ce n'est pas le cas. C'est pourquoi nous avons mis en place une analyse sémantique ou le nombre d'arguments nécessaire à l'appel de la fonction est vérifié :

```
functionCallNode = AST.getFunction(p[1])
if functionCallNode.children[0].verifyArgumentsNumber(len(p[3].children)):
```

Ici p[1] es le nom de la fonction et p[3] est la liste d'arguments. Elle appelle cette fonction sur la FunctionNode :

```
class FunctionNode(Node):
    type = 'Function'
    def verifyArgumentsNumber(self,nb):
        if nb == 0:
            return self.children[1].children[0].tok == 'No Arguments'
        return len(self.children[1].children) == nb and self.children[1].children[0].tok != 'No Arguments'
```

Si p[3] est vide, on vérifie que la fonction appelée avait bien aucun arguments. Si p[3] est un nombre on vérifie que la fonction appelée à le même nombre d'arguments et que cette fonction avait bien des arguments. En effet, si p[3] = 1, et que la fonction n'a pas d'arguments, le nœud 'No arguments' compterait pour un et de ce fait le résultat serait faussé.

RETURN

Return est un mot réservé (rajouté dans le lexer) qui peut être suivi d'une expression ou non. En effet, on peut l'utiliser simplement pour arrêter une fonction sans rien renvoyer :

```
def p_return(p):
    '''returnStatement : RETURN '''
    p[0] = AST.ReturnNode()
```

Sinon, on peut renvoyer dans notre cas, un tableau, une expression, une autre fonction, ou une condition (donne un booléen) :

```
def p_return_expression(p):
    '''returnStatement : RETURN returnValues
    | RETURN condition'''
```

Avec :

```
def p_return_values(p):
    '''returnValues : expression
    | arrayDeclaration
    | functionCall'''
```

Le return demande une analyse sémantique, en effet il ne devrait jamais être placée en dehors d'une fonction. De la même manière que le break et le continue, on vérifie une fois que l'arbre est fait si tous les nœuds return sont bien en dessous d'un nœud fonction avec

```
def verifyReturnNode():
```

Qui est placée dans le fichier AST et lancée en même temps que la vérification break et continue. Si un des nœuds return est mal placée, une erreur est lancée et un message s'affiche.

Comme vu plus haut, notre langage permet de faire un ***var a = getPi()*** par exemple. Dans le cas ou getPi() n'aurait pas de valeur de retour, javascript renvoit simplement « Undefined » mais ne lève pas d'erreur. C'est pourquoi nous ne vérifions pas, comme javascript, si la function qu'on appelle retourne quelque chose.

GESTION DES ERREURS

Dans notre projet les erreurs, sont affichés dans la console et une variable error est mise à True pour empêcher d'afficher l'arbre dans la console afin d'empêcher de surcharger d'erreurs en cascades l'utilisateur. Nous utilisons le système d'erreur mise en place pour le TP4 pour les erreurs de syntaxes avec le numéro de ligne correspondant :

```
def p_error(p):
    error = True
    if p:
        print (f"Syntax error in line {p.lineno} with {p}")
        parser.errok()
    else:
        print ("Sytax error: unexpected end of file!")
```

Mais en plus de cela, étant donné que notre analyse sémantique se réalise dans le même fichier, les erreurs sémantiques sont elles aussi affichés dans la console, par exemple, une erreur de déclaration de variable :

```
else :
    error = True
    print(f"ERROR : {p[3]} is already declared")
```

FONCTIONNALITES NON-IMPLEMENTEES

Voici une liste exhaustive des fonctionnalités qui n'ont pues être développées dans leur entièreté.

- L'utilisation de tableaux dans la partie arrière ;
- L'emploi du mot break dans la partie arrière ;
- L'utilisation des fonctions dans la partie arrière
- Le if/else dans la partie arrière
- Le switch dans la partie arrière considère qu'il y a un break dans chaque case puisque le mot clé break ne fut pas implémenté.
- On ne peut pas écrire de structure vides comme un while(cond){}

POUR ALLER PLUS LOIN

Notre projet est bien avancé, mais bien sûr il est impossible d'implémenter la compilation d'un langage complet en si peu de temps. La partie objet serait la première chose à ajouter, les classes, les strings. Mais nous n'avons pas implémenter quelques autres fonctionnalités intéressantes comme le try/catch/finally par exemple. Pour pousser un peu plus loin les conditions, il serait nécessaire de mettre en place les mots true et false, et bien sûr, dans le cas ou tout ceci serait fait, il reste toujours à mettre à jour en même temps que les dernières mises à jour du langage.

BUGS CORRIGES

Nous avons fait face à un de nombreux bugs. En effet, sur 47 issues, 20 sont des résolutions de bugs, découvert à l'aide d'une phase de tests. La plupart du temps, il s'agissait de problème dans la grammaire ou dans la vérification des places des nœuds dans le fichier AST. Il est vite apparu qu'un warning reduce/reduce implique de gros problèmes de grammaire et donc de nombreux bugs et qu'un warning shift/reduce indique un plus léger problème mais qui a dans notre cas forcement entrainé un bug par la suite. De ce fait, conseil aux futurs étudiants (puissent-ils m'entendre) : il faut absolument résoudre ces warnings, qui ne sont pas de simple attention mais bel et bien des nids à erreurs en tout genre. De plus, il faut les résoudre dès leur apparition afin d'éviter de devoir revenir sur un morceau de code réalisé il y a plusieurs semaines qui bug et dont le changement va lui aussi entrainer des bugs dans d'autres fonctions. En effet, la plupart de ces bugs ont un effet boule de neige, les solutions entrainant d'autres bugs ailleurs, et rapidement plus rien n'est correct dans la grammaire.

BUGS NON CORRIGES

Nous n'avons pas réussi à corriger un bug sur le if dû à l'utilisation ou non de accolades. Il est donc nécessaire de mettre des accolades lorsqu'on utilise un if seul et de coller le else à l'accolade dans le cas d'un if else avec accolade comme ceci:

If(){

} else {}

GUIDE D'UTILISATION

REQUIREMENTS

Afin de lancer notre projet, vous devez au préalable installer via la commande pip les modules PLY et pydot ainsi qu'installer Graphviz à cette adresse : https://graphviz.org/download/

INSTALLATION

Après l'installation des modules et de Graphviz, il suffit d'extraire l'archive fournie, et de lancer depuis un terminal. Nous vous conseillons d'utiliser powershell, l'accès des tests est ainsi facilité grâce aux tabulations qui permettent de voir ce qui est dans un dossier.

ANALYSE LEXICALE

Exactement comme pour les travaux pratiques dont découle notre projet, pour avoir accès à seulement l'analyse lexicale, lancez le fichier lex.py avec le fichier texte de votre choix en argument, ici un de nos tests préparés :

```
C:\Users\Joris\Desktop\3\JSCompilator>python lex.py ./Tests/Tests-Working/arrays.txt
line 1: VAR(var)
line 1: IDENTIFIER(a)
line 1: =(=)
line 1: [([)
line 1: ](])
line 1: NEWLINE(
)
line 2: VAR(var)
line 2: IDENTIFIER(b)
line 2: =(=)
line 2: [([)
```

Le résultat est écrit dans la console avec pour chaque groupe de caractère, son groupe lexical si celui-ci est reconnu, sinon une erreur sera affichée à la place.

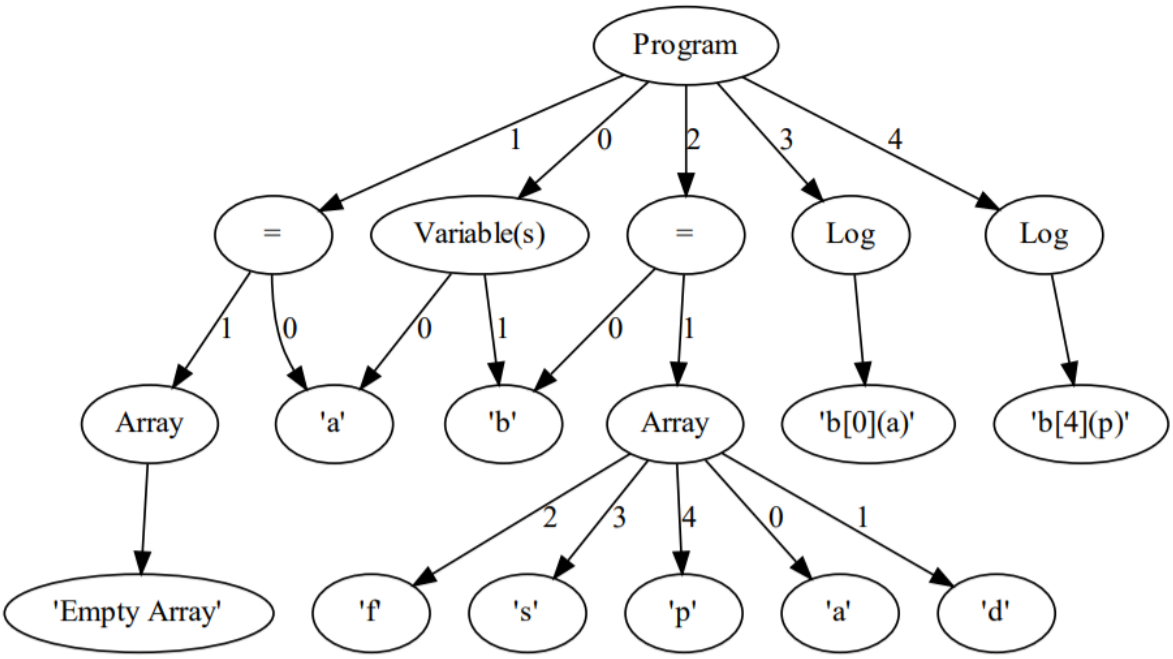
ANALYSE SYNTAXIQUE

De la même manière, pour lancer l'analyse syntaxique, il faut exécuter le fichier parserJS.py avec en argument un fichier texte :

```
C:\Users\Joris\Desktop\3\JSCompilator>python parserJS.py ./Tests/Tests-Working/arrays.txt
Generating LALR tables
Program
|  Variable(s)
|  |  'a'
|  |  'b'
|  =
|  |  'a'
|  |  Array
|  |  |  'Empty Array'
|  =
|  |  'b'
|  |  Array
|  |  |  'a'
|  |  |  'd'
|  |  |  'f'
|  |  |  's'
|  |  |  'p'
|  Log
|  |  'b[0](a)'
|  Log
|  |  'b[4](p)'

wrote ast to ./Tests/Tests-Working/arrays-ast.pdf
```

Cette fois-ci, il y a deux résultats, d'une part l'arbre syntaxique affiché directement dans la console et d'autre part l'arbre syntaxique dessiné par pydot, dans le fichier PDF généré :



Dans le cas d'une erreur, celle-ci sera affichée au-dessus de l'arbre syntaxique dans la console, si ce dernier est grand et que des erreurs sont affichées, pensez à scroller au-dessus de l'arbre, là où les erreurs sont affichées.

ANALYSE SEMANTIQUE

L'analyse sémantique se fait directement dans l'analyse syntaxique, en effet nous n'utilisons pas de couture dans notre projet, pour la tester, lancez donc une analyse syntaxique, notamment sur le fichier de tests var où les portées des variables sont testées.

PARTIE ARRIERE

La partie arrière se décompose en deux commandes séparées. En effet, nous transformons tout d'abord notre code en bytecode avant de l'exécuter. Pour transformer le code en bytecode, nous utilisons cette commande :

```
\JSCompilator> python .\svmInterpreter.py .\Tests\Tests-Working\for.txt
```

On exécute svmInterpreter.py en lui passant en argument le fichier texte à transformer. Une fois le bytecode généré dans un fichier .vm : if.vm comme ici avec le if, on exécute svm en lui donnant comme argument ce fichier .vm avec la commande : `python .\svm.py .\Tests\Tests-Working\if.vm`

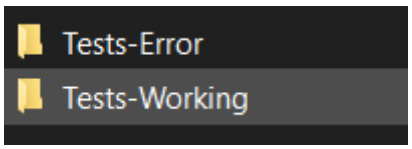
Un exemple de bytecode généré et son résultat dans la console :

```
PUSHC 5.0
SET a
PUSHC 2.0
PRINT
PUSHC 4.0
PRINT
PUSHC 2.0
PRINT
PUSHC 5.0
PRINT
PUSHC 6.0
PRINT
PUSHC 7.0
PRINT
PUSHC 8.0
PRINT
```

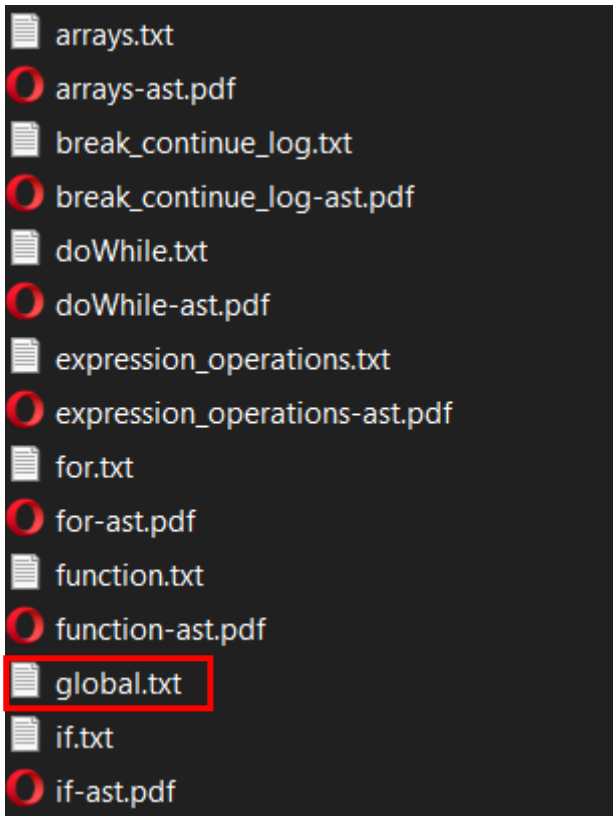
```
2.0
4.0
2.0
5.0
6.0
7.0
8.0
```

TESTS

En même temps que ce rapport nous vous proposons une multitude de fichiers de tests pour chaque fonctionnalité. Ils sont rangés dans ces deux dossiers contenus dans Tests/:



Tests-Error ne comprend que des tests erronés permettant notamment de montrer notre gestion des erreurs. Au contraire Tests-Working, regroupe des tests fonctionnels montrant toutes les possibilités de syntaxe et toutes les fonctionnalités de notre compilateur. Leurs noms correspondent à une fonctionnalité testée. Dans le dossier tests-Working, nous vous proposons de plus un fichier global qui permet de tester toutes les fonctionnalités en même temps (attention, arbre conséquent qui commence à devenir difficile à lire) :



De plus dans le cas ou vous voudriez tester la syntaxe javascript normale, nous avons un fichier HTML dans le dossier DebugTests/ permettant de rapidement tester une syntaxe (par exemple ;;;;; dans une ligne). Dans le même dossier se trouve un autre fichier texte nommé test.txt qui vous permettra de tester vos propres syntaxes dans notre compilateur.

CONCLUSION

D'abord un point sur l'avancement global du projet :

- Le lexer est fini à 100 % dans le cadre de nos objectifs
- Le parser est fini à 99% (bug sur le if) dans le cadre de notre projet
- L'analyse sémantique est réalisée à 100% dans le cadre de nos objectifs
- La partie arrière est réalisée à 10% dans le cadre de notre projet

Ce projet nous a permis de mettre en pratique plus profondément les notions apprises en cours. Pour le parser, qui est la partie nous ayant pris le plus de temps, nous avons eu besoin en plus du cours, des précédences de poids égales, du principe de « new_scope » pour la portée des variables et d'autres fonctionnalités de PLY mineures. Pour l'analyse sémantique, reparcourir l'arbre pour vérifier la place des nœuds a permis de mieux comprendre l'arbre dans sa globalité et à retravailler les list comprehensions de python. Enfin, la partie arrière fut pour nous la partie la plus difficile du projet. Au cours de nos recherches, nous avons vu qu'aucun compilateur en python pour javascript n'existe. Nous sommes donc repartis du fichier svm.py du TP4 mais ce fut difficile de le mettre en pratique pour nos structures. Pour

finir, en termes de retour d’expérience, cette dernière fut enrichissante, mais dans le même temps quelque peu frustrant puisque nous n’avons pas fini tout ce qui fut commencé.