

|  |
| --- |
| Compilateur |
| Rapport JSCompilator |
| INF3 DLM-b |

|  |
| --- |
| Paysant Adrien - Monnet Joris  2020/2021 |

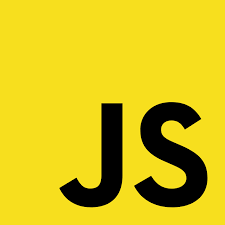
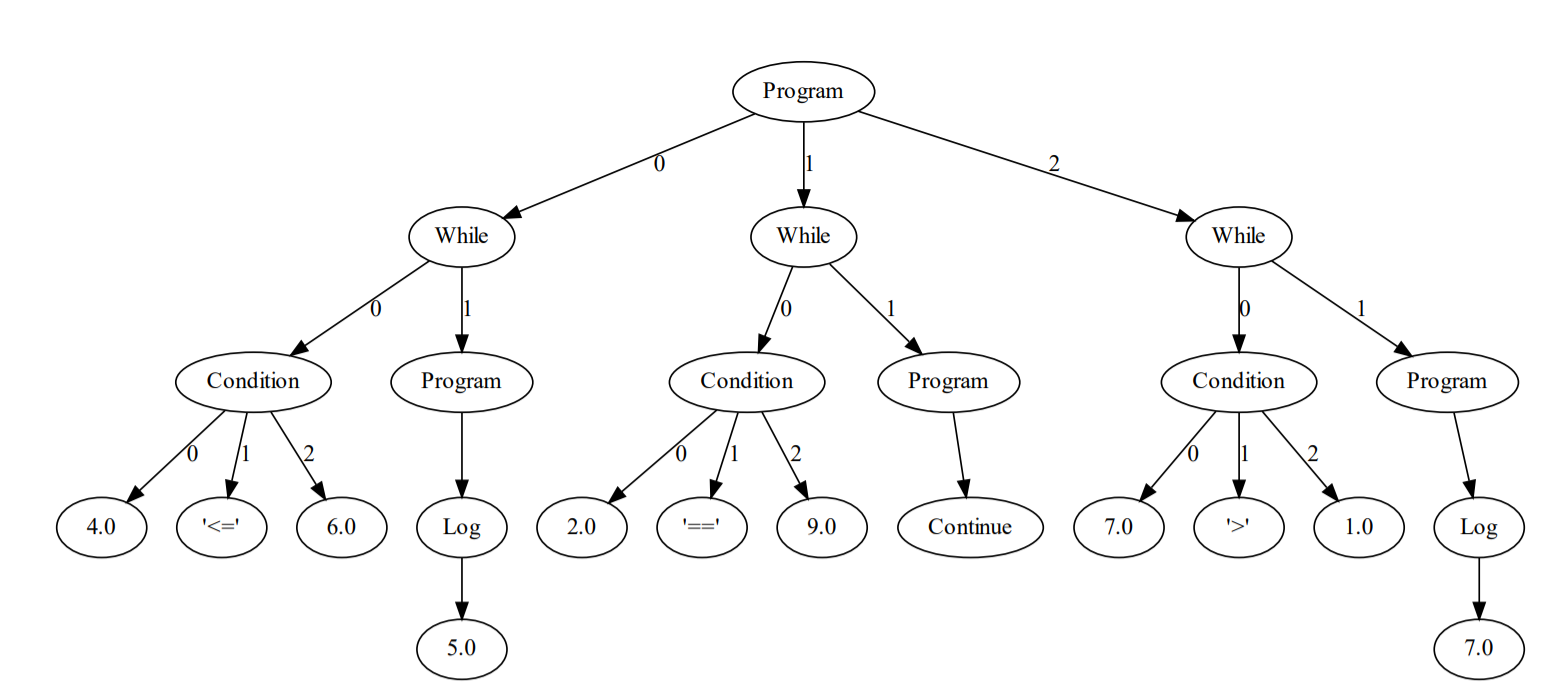


Table des matières

[Introduction 2](#_Toc61221654)

[Objectifs 2](#_Toc61221655)

[Langage 2](#_Toc61221656)

[But 2](#_Toc61221657)

[Objectifs Principaux 2](#_Toc61221658)

[Objectifs Secondaires 2](#_Toc61221659)

[Fonctionnalités Implémentées 2](#_Toc61221660)

[Règles pour les accolades 2](#_Toc61221661)

[Reprise du TP4 3](#_Toc61221662)

[Log 4](#_Toc61221663)

[While 4](#_Toc61221664)

[Raccourci d’opérations pour les expressions 4](#_Toc61221665)

[Conditions 5](#_Toc61221666)

[If/Else 6](#_Toc61221667)

[Opérateur ternaire 7](#_Toc61221668)

[For 8](#_Toc61221669)

[Switch 9](#_Toc61221670)

[Do while 9](#_Toc61221671)

[Var et Let 10](#_Toc61221672)

[Portée des variables 10](#_Toc61221673)

[Règles permissives pour le point-virgule 10](#_Toc61221674)

[Break et Continue 11](#_Toc61221675)

[Tableaux de nombre et de variables 12](#_Toc61221676)

[Fonctions 12](#_Toc61221677)

[Analyse sémantique du nombre d’arguments 12](#_Toc61221678)

[Return 12](#_Toc61221679)

[Gestion des erreurs 12](#_Toc61221680)

[Fonctionnalités non-implémentées (partie arrière) 12](#_Toc61221681)

[Pour aller plus loin 12](#_Toc61221682)

[Bugs corrigés 12](#_Toc61221683)

[Guide d’utilisation 13](#_Toc61221684)

[Analyse lexicale 13](#_Toc61221685)

[Analyse syntaxique 13](#_Toc61221686)

[Analyse sémantique 14](#_Toc61221687)

[Partie arrière 15](#_Toc61221688)

[Tests 15](#_Toc61221689)

[Retour d’expérience 16](#_Toc61221690)

[Conclusion 16](#_Toc61221691)

# Introduction

JSCompilator consiste en un compilateur de javascript réalisé en python.

# 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

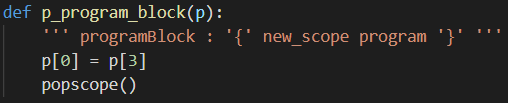
## Objectifs Secondaires

* 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 []

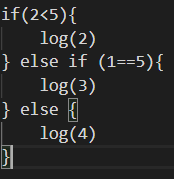
# Fonctionnalités Implémentées

## Règles pour les accolades

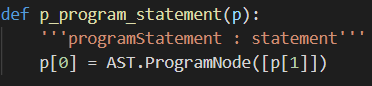
Malgré le fait que ce ne soit pas explicitement écrit dans les objectifs, nous avons fait le choix de donner la possibilité d’utiliser une syntaxe avec accolades (nommé programBlock dans le parser) :



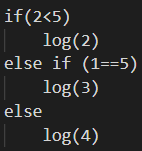
Pour cette syntaxe :



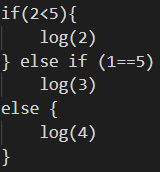
Mais aussi une syntaxe sans accolades pour les structures (pas les fonctions) nommé programStatement dans le parser étant donné que sans accolades, une seule instruction (statement) est autorisée :



Pour cette syntaxe :



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



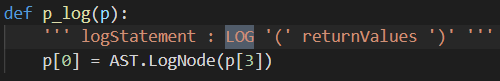
De ce fait, notre parser donne les possibilités complètes de javascript.

## Reprise du TP4

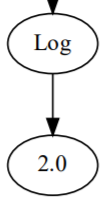
Notre premier objectif était donc de reprendre le TP4, nous avons gardé les calculs sur les expressions : ADD\_OP, MUL\_OP, UMINUS puis en avons ajouté d’autres, récupéré le while et changé le print en log.

### 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 :

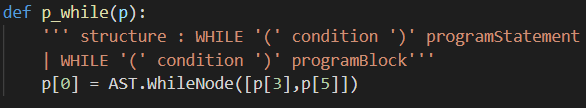


Ici les return values correspondent à ce que peut return une fonction, c’est-à-dire une expression comme une variable ou un calcul, un tableau, ou encore une fonction. Le résultat dans l’arbre : pour un log(2) :



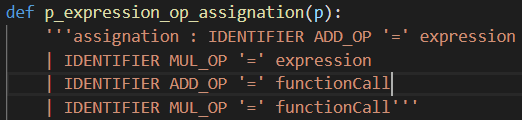
### While

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 :



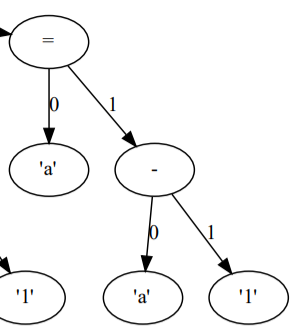
### Raccourci d’opérations pour les expressions

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.



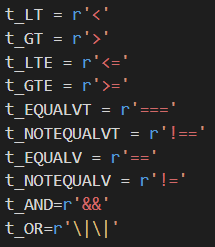


Ces raccourcis sont représentés dans l’arbre comme s’ils avaient été écrits sans raccourcis :

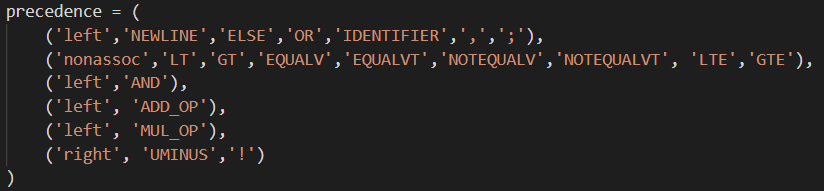
*Correspond à a--*

## Conditions

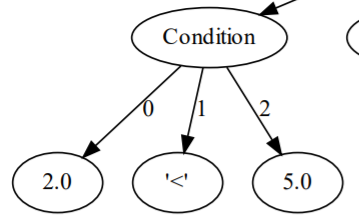
Il semblait nécessaire de mettre en place des conditions, notamment pour le while, le if et le for. Dans le lexer :



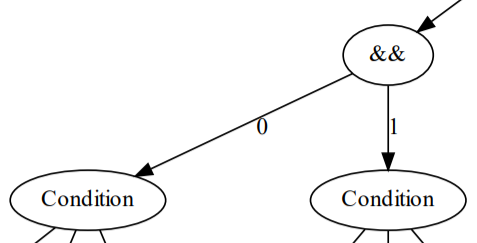
On accepte maintenant tous les opérateurs de comparaison et les opérateurs AND et OR. De plus, nous avons ajouté ! dans les literals afin de permettre l’opérateur NOT. Au niveau du parser, les operateurs de comparaison son non-associatif pour ne pas pouvoir les chainer (a<b<c) et le comparateur ! est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



Une condition se présente comme ceci :

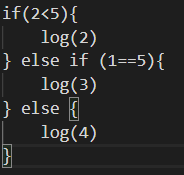


Nous avons aussi ajouté des AND, OR et NOT nodes pour les différents opérateurs entre conditions :

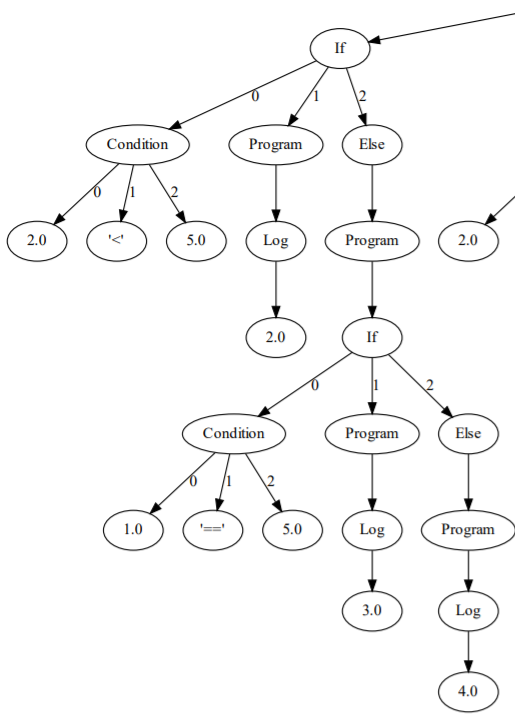


## If/Else

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 :



Qui donne :

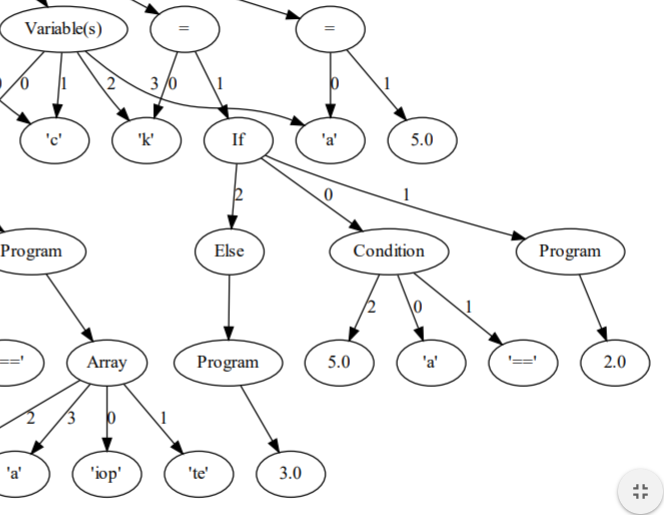


*Ici en rouge le chemin if/else if /else*

Cette récursion fut source de nombre de bugs, notamment dans la version sans accolades, ceux-ci seront détaillés plus loin.

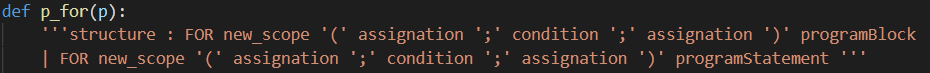
### Opérateur ternaire

L’operateur 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  donne :

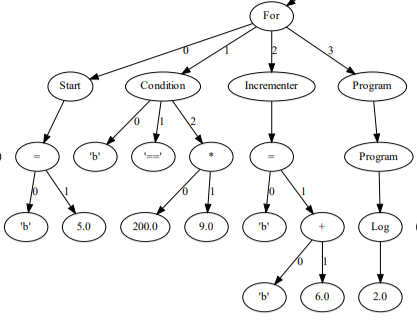


## For

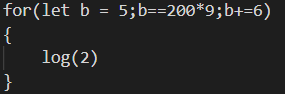
Dans la partie lexer, le mot clé est simplement rajouté. Dans le cadre de ce projet, nous avons implémenter le for classique :



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 :



## Switch

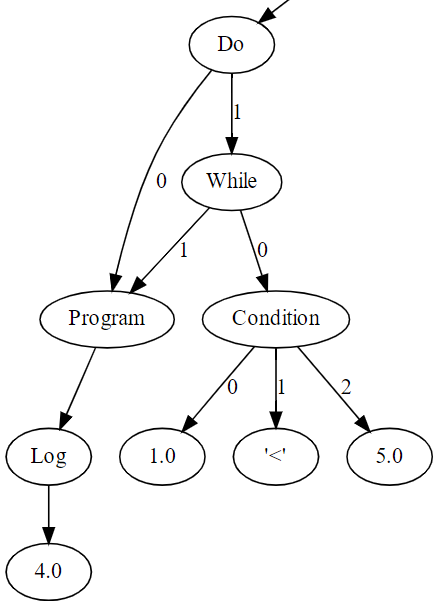
Pour cette fonctionnalité, nous avons ajouté les mots réservés dans le lexer, à savoir SWITCH, CASE et DEFAULT. Ensuite dans le parser, il s’agit d’une des structures les plus complexes. En effet,

## 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 :



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 :



## 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 :



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).

### Portée 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 ;***

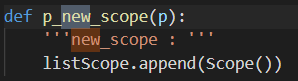
***If(1<5){***

***Let i =1 ;***

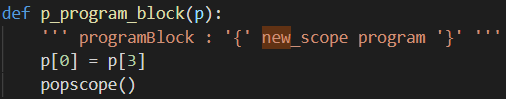
***Console.log(i)***

***}***

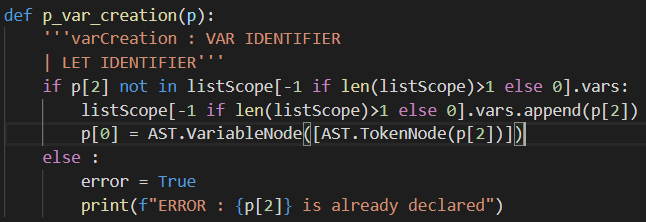
Ici, en javascript, cela afficherait 1, pour nous une erreur est lancée car l’utilisateur redéfinit i. Il est donc nécessaire d’avoir un nom de variable différent dans ce cas-ci. En dehors de cela nous avons mis en place comme en javascript, l’impossibilité d’appeler une variable dans portée plus large 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 :



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 :



Exemple de création de variable :



## Règles 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 :



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 :

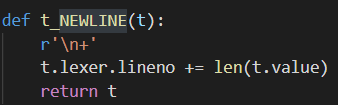


Ici, nous allons expliquer cette ligne pas à pas. En premier lieu, il s’agit de notre dernière ligne de la fonction parser, elle retourne donc notre fichier parser par yacc. En revanche il se passe deux opérations sur le fichier à parser. La plus simple et l’ajout de \n en toute fin de ligne, en effet, 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 :

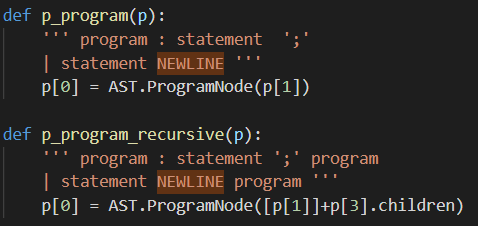


Nous utilisons re.sub() qui permet de remplacer une expression régulière avec une fonction lambda. Nous recherchons avec la regex tous les points virgules séparés que par des retours à la ligne d’une accolade fermante. Nous remplaçons chacun de ces expressions par le même nombre de retour à la ligne 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 :

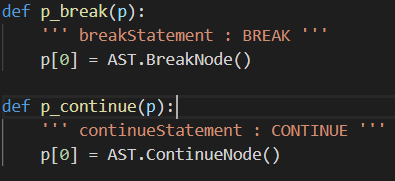


Et l’utilisons dans le parser notamment avec les programmes :



## Break et Continue

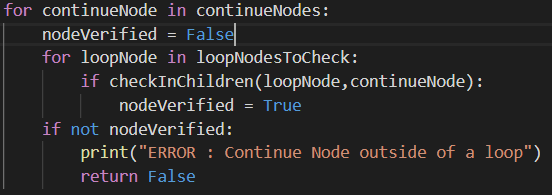
Dans le lexer nous avons rajouté ces deux mots clés. Ensuite nous les avons mis en place dans le parser : continue et break sont des mots à utiliser seuls, on crée des nodes AST pour eux et c’est fini :



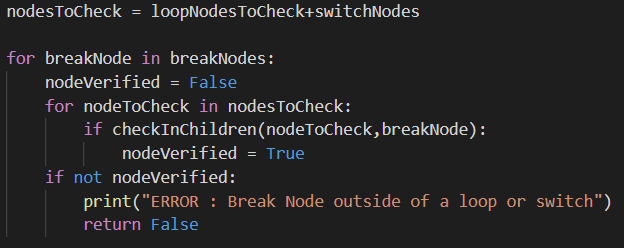
Malheureusement, ce n’est pas si simple. En effet, syntaxiquement ce sera juste, mais sémantiquement, un break et un continue sont forcément dans des boucles (ou switch pour le break).

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 : qui appelle la fonction qui lance .

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 :



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



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

## Tableaux de nombre et de variables

## Fonctions

## Analyse sémantique du nombre d’arguments

## Return

## Gestion des erreurs

# Fonctionnalités non-implémentées (partie arrière)

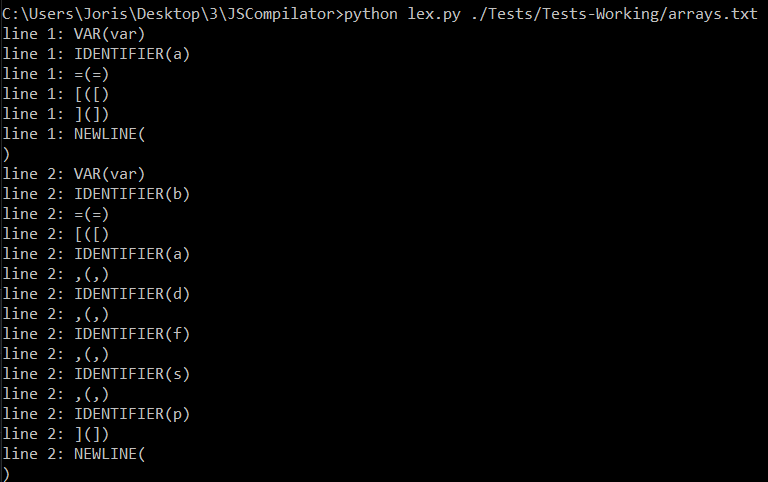
# Pour aller plus loin

# Bugs corrigés

# Guide d’utilisation

## Analyse lexicale

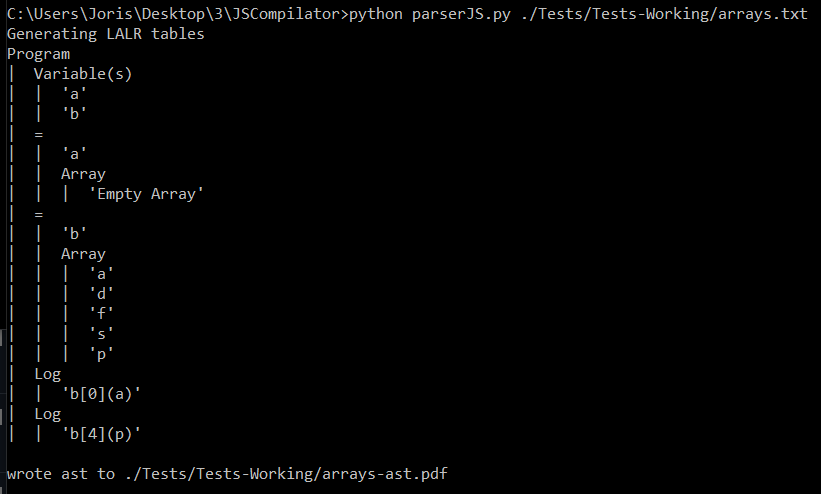
Exactement comme pour les travaux pratiques dont découle notre projet, pour avoir accès à seulement l’analyse lexicale, lancez le fichier lex.py avec le fichier texte de votre choix en argument, ici un de nos tests préparés :



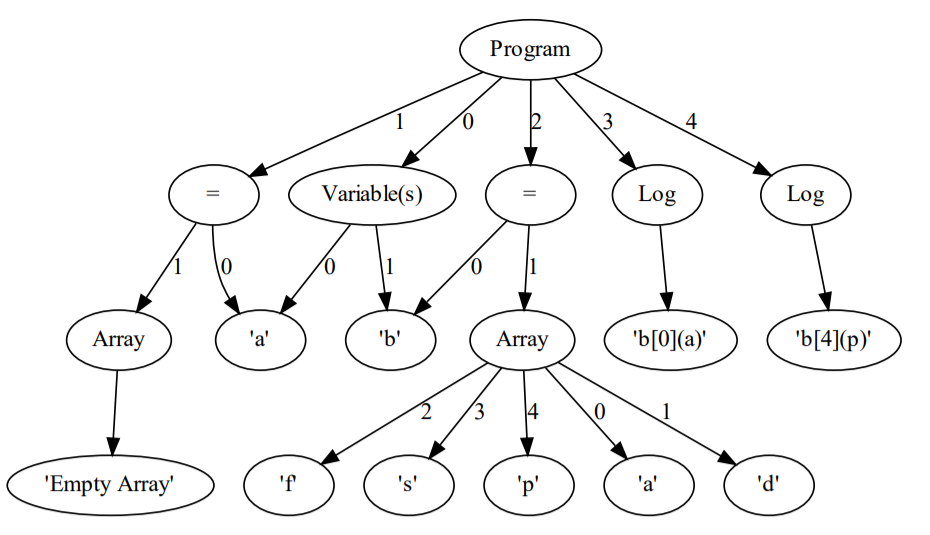
Le résultat est écrit dans la console avec pour chaque groupe de caractère, son groupe lexical si celui-ci est reconnu, sinon une erreur sera affichée à la place.

## Analyse syntaxique

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



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 sémantique

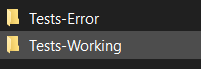
L’analyse sémantique se fait directement dans l’analyse syntaxique, en effet nous n’utilisons pas de couture dans notre projet, pour la tester, lancez donc une analyse syntaxique, notamment sur le fichier de tests var où les portées des variables sont testées.

## Partie arrière

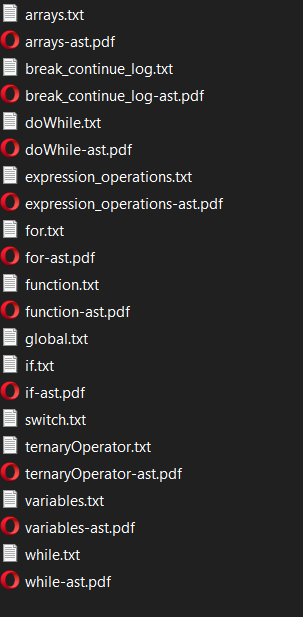
ADRIEN

## 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.

# Retour d’expérience

# Conclusion