

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

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

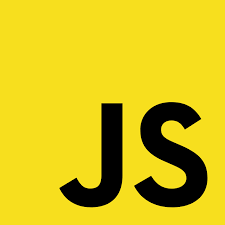
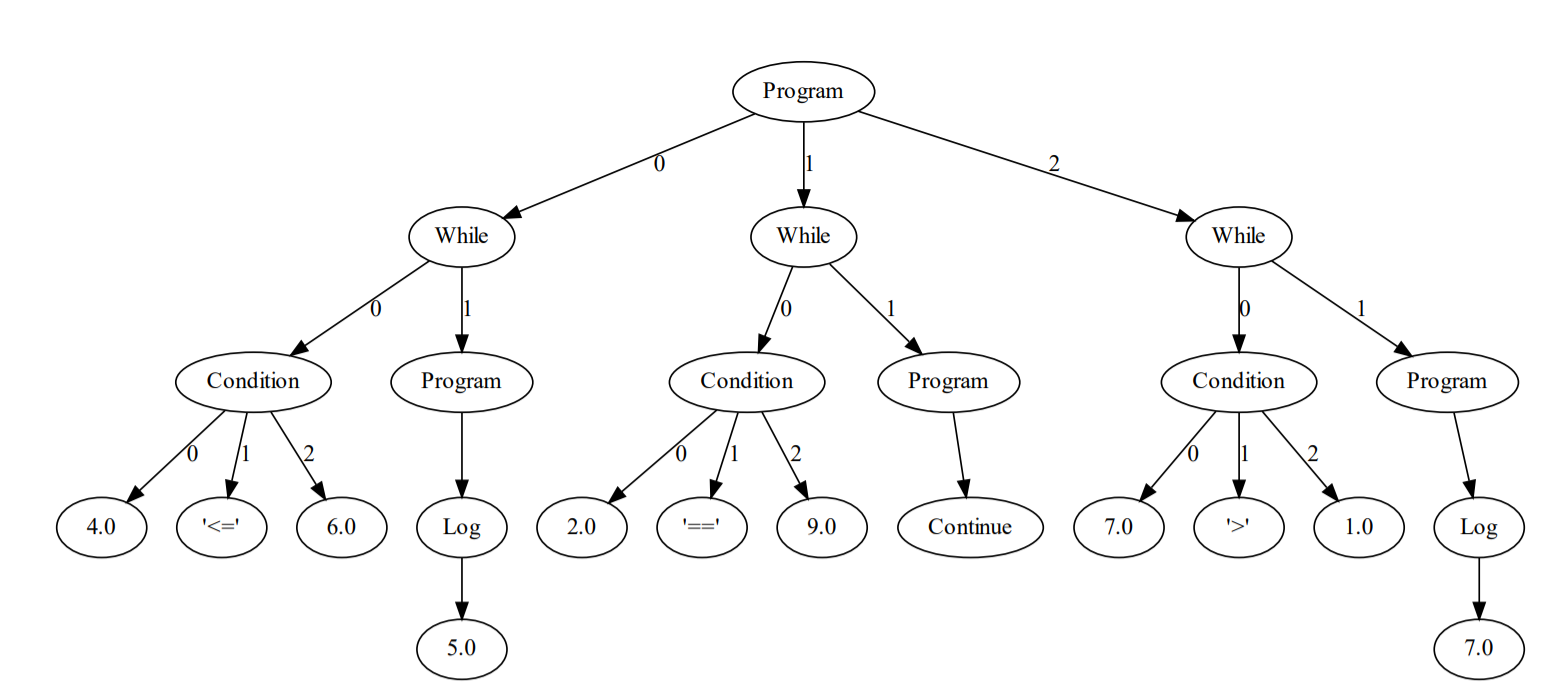


Table des matières

[Introduction 2](#_Toc61143089)

[Objectifs 2](#_Toc61143090)

[Langage 2](#_Toc61143091)

[But 2](#_Toc61143092)

[Objectifs Principaux 2](#_Toc61143093)

[Objectifs Secondaires 2](#_Toc61143094)

[Fonctionnalités Implémentées 2](#_Toc61143095)

[Règles pour les accolades 2](#_Toc61143096)

[Reprise du TP4 3](#_Toc61143097)

[Log 4](#_Toc61143098)

[While 4](#_Toc61143099)

[Raccourci d’opérations pour les expressions 4](#_Toc61143100)

[Conditions 5](#_Toc61143101)

[If/Else 6](#_Toc61143102)

[Opérateur ternaire 7](#_Toc61143103)

[For 8](#_Toc61143104)

[Switch 9](#_Toc61143105)

[Do while 9](#_Toc61143106)

[Var et Let 9](#_Toc61143107)

[Portée des variables 9](#_Toc61143108)

[Règles permissives pour le ; 9](#_Toc61143109)

[Break et Continue 9](#_Toc61143110)

[Tableaux de nombre et de variables 9](#_Toc61143111)

[Fonctions 9](#_Toc61143112)

[Analyse sémantique du nombre d’arguments 9](#_Toc61143113)

[Return 9](#_Toc61143114)

[Gestion des erreurs 9](#_Toc61143115)

[Fonctionnalités non-implémentées (partie arrière) 9](#_Toc61143116)

[Pour aller plus loin 9](#_Toc61143117)

[Bugs corrigés 9](#_Toc61143118)

[Guide d’utilisation 10](#_Toc61143119)

[Analyse lexicale 10](#_Toc61143120)

[Analyse syntaxique 10](#_Toc61143121)

[Analyse sémantique 11](#_Toc61143122)

[Partie arrière 11](#_Toc61143123)

[Tests 11](#_Toc61143124)

[Retour d’expérience 12](#_Toc61143125)

[Conclusion 12](#_Toc61143126)

# 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.

## 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 ‘;’

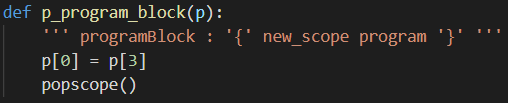
## Objectifs Secondaires

* 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 []

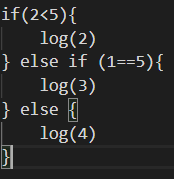
# Fonctionnalités Implémentées

## Règles pour les accolades

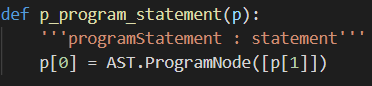
Malgré le fait que ce ne soit pas explicitement écrit dans les objectifs, nous avons fait le choix de donner la possibilité d’utiliser une syntaxe avec accolades (nommé programBlock dans le parser) :



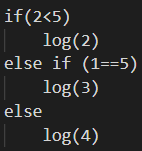
Pour cette syntaxe :



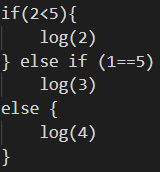
Mais aussi une syntaxe sans accolades pour les structures (pas les fonctions) nommé programStatement dans le parser étant donné que sans accolades, une seule instruction (statement) est autorisée :



Pour cette syntaxe :



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



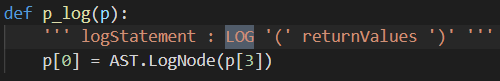
De ce fait, notre parser donne les possibilités complètes de javascript.

## Reprise du TP4

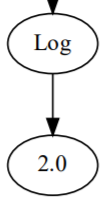
Notre premier objectif était donc de reprendre le TP4, nous avons gardé les calculs sur les expressions : ADD\_OP,MUL\_OP,UMINUS puis en avons ajouté d’autres, récupéré le while et changé le print en log.

### 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 :

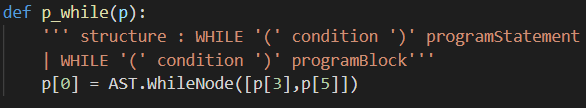


Ici les return values correspondent à ce que peut return une fonction, c’est-à-dire une expression comme une variable ou un calcul, un tableau, ou encore une fonction. Le resultat dans l’arbre : pour un log(2) :



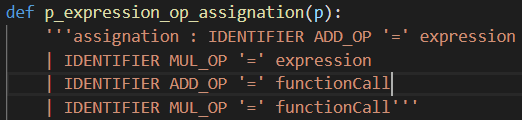
### While

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 :



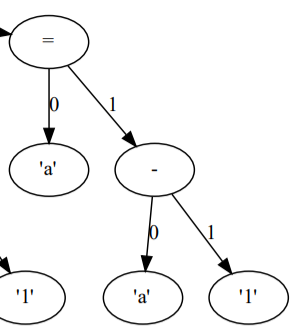
### Raccourci d’opérations pour les expressions

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.



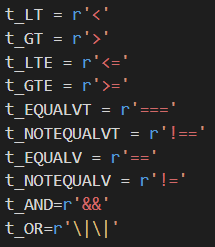


Ces raccourcis sont représentés dans l’arbre comme s’ils avaient été écrits sans raccourcis :

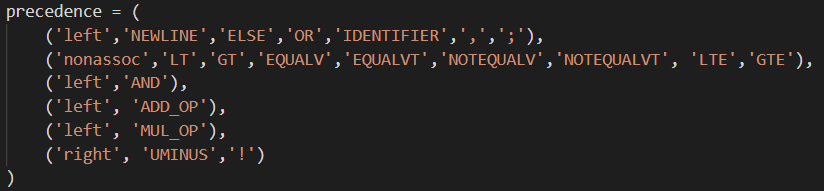
*Correspond à a--*

## Conditions

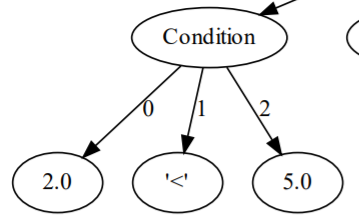
Il semblait nécessaire de mettre en place des conditions, notamment pour le while, le if et le for. Dans le lexer :



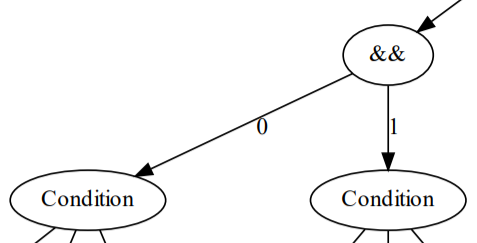
On accepte maintenant tous les opérateurs de comparaison et les opérateurs AND et OR. De plus, nous avons ajouté ! dans les literals afin de permettre l’opérateur NOT. Au niveau du parser, les operateurs de comparaison son non-associatif pour ne pas pouvoir les chainer (a<b<c) et le comparateur ! est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



Une condition se présente comme ceci :

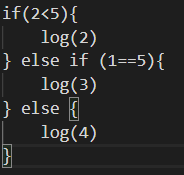


Nous avons aussi ajouté des AND, OR et NOT nodes pour les différents opérateurs entre conditions :

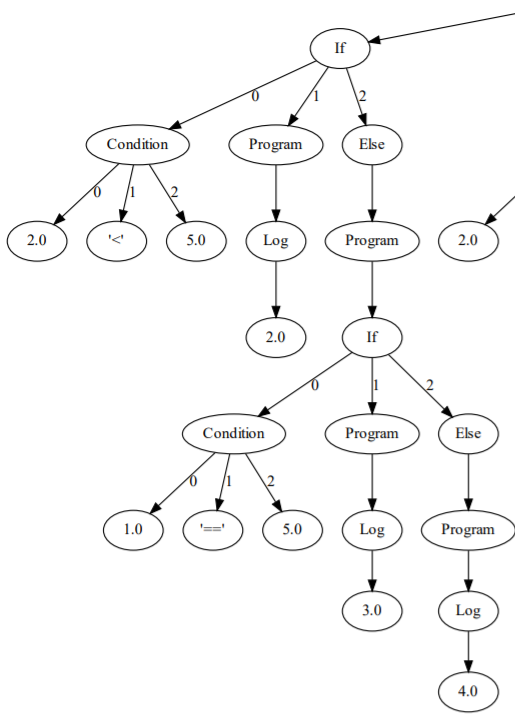


## If/Else

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 :

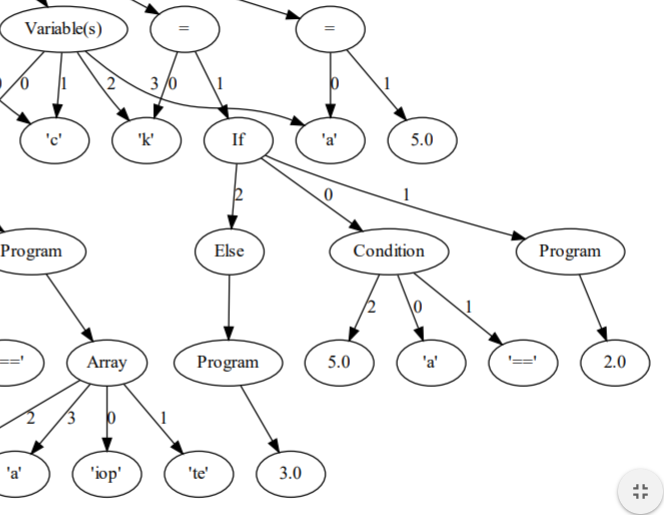


*Ici en rouge le chemin if/else if /else*

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

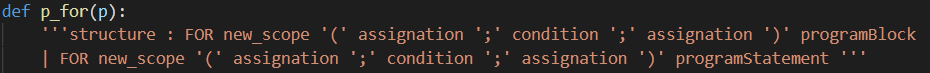
### Opérateur ternaire

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 :

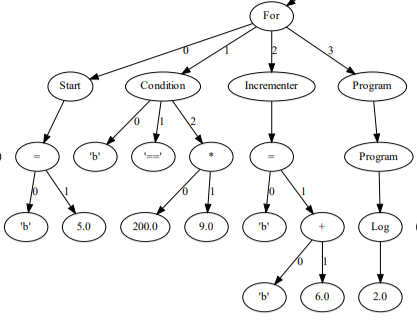


## For

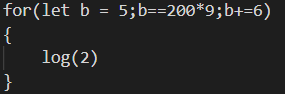
Dans la partie lexer, le mot clé est simplement rajouté. Dans le cadre de ce projet, nous avons implémenter le for classique :



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 en entrée :



## Switch

## Do while

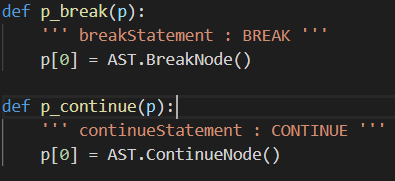
## Var et Let

### Portée des variables

## Règles permissives pour le ;

## Break et Continue

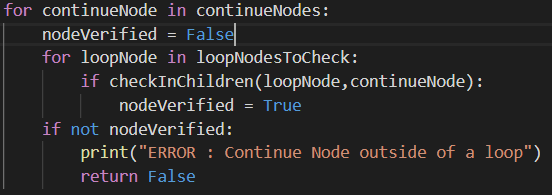
Dans le lexer nous avons rajouté ces deux mots clés. Ensuite nous les avons mis en place dans le parser : continue et break sont des mots à utiliser seuls, on crée des nodes AST pour eux et c’est fini :



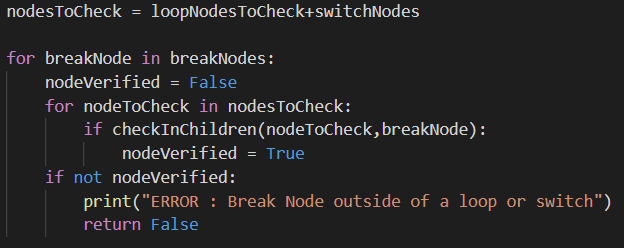
Malheureusement, ce n’est pas si simple. En effet, syntaxiquement ce sera juste, mais sémantiquement, un break et un continue sont forcément dans des boucles (ou switch pour le break).

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 tout les switch. 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)

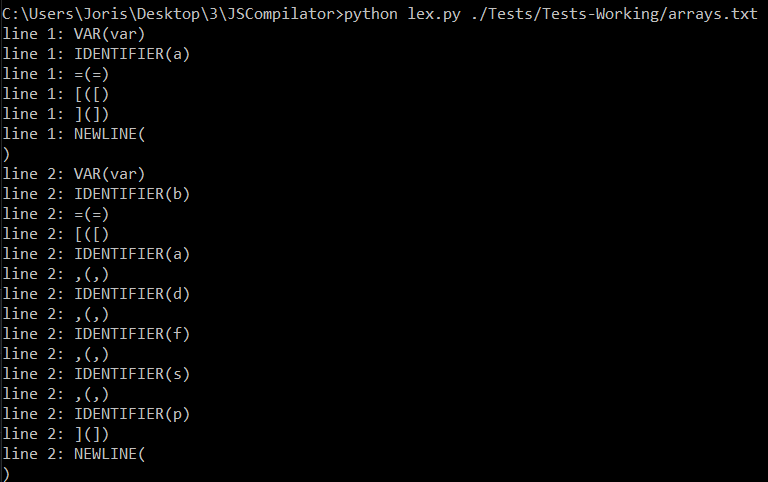
# Pour aller plus loin

# Bugs corrigés

# Guide d’utilisation

## Analyse lexicale

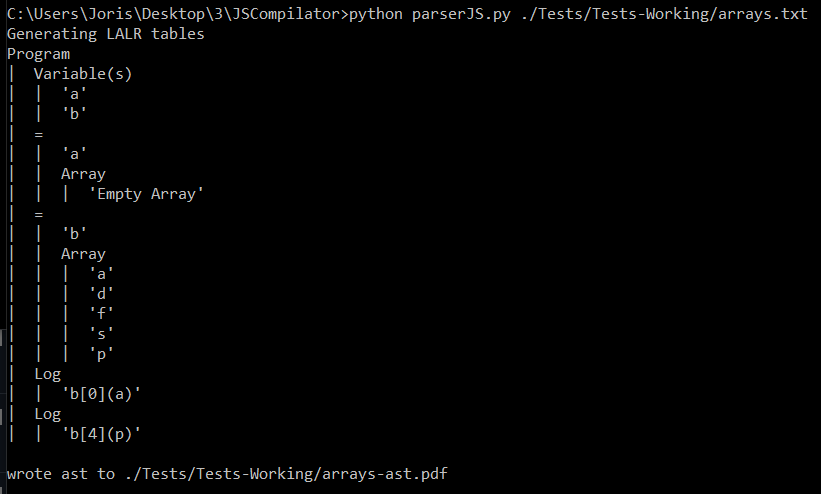
Exactement comme pour les travaux pratiques dont découle notre projet, pour avoir accès à seulement l’analyse lexicale, lancez le fichier lex.py avec le fichier texte de votre choix en argument, ici un de nos tests préparés :



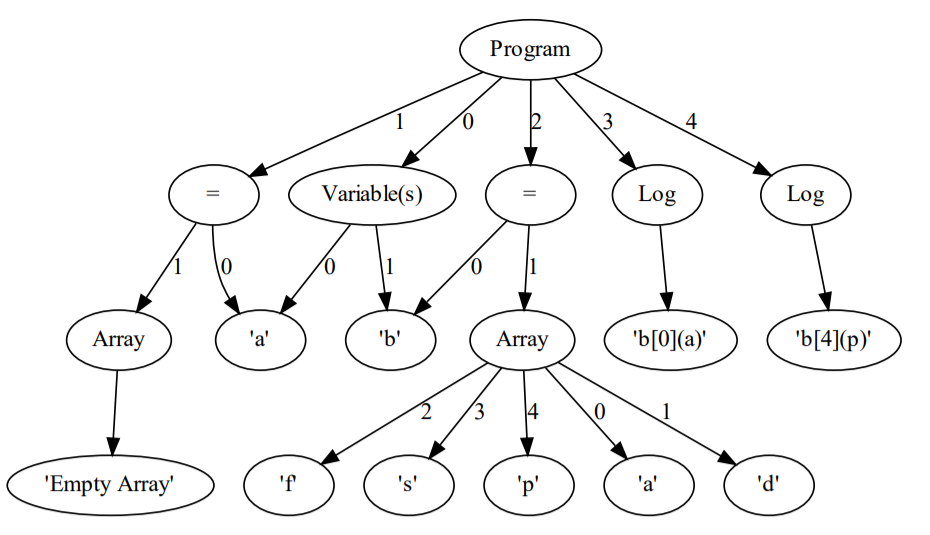
Le résultat est écrit dans la console avec pour chaque groupe de caractère, son groupe lexical si celui-ci est reconnu, sinon une erreur sera affichée à la place.

## 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

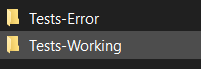
L’analyse sémantique se fait directement dans l’analyse syntaxique, en effet nous n’utilisons pas de couture dans notre projet, pour la tester, lancez donc une analyse syntaxique, notamment sur le fichier de tests var où les portées des variables sont testées.

## Partie arrière

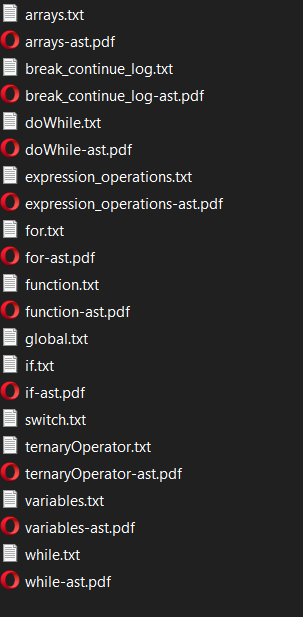
ADRIEN

## 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