Rapport du projet

Par Gabriel Arsenault

Table des matières

[Préparation de l’expérimentation 1](#__RefHeading___Toc170_325362768)

[1. Sujet 1](#__RefHeading___Toc172_325362768)

[2. Objectif 2](#__RefHeading___Toc174_325362768)

[3. Critères d’évaluation 2](#__RefHeading___Toc176_325362768)

[Réalisation de l’expérimentation 3](#__RefHeading___Toc178_325362768)

[4. Expérimentation 3](#__RefHeading___Toc180_325362768)

[1.1. Exigences techniques 3](#__RefHeading___Toc182_325362768)

[1.2. Déroulement 3](#__RefHeading___Toc184_325362768)

[1.3. Résultats 3](#__RefHeading___Toc186_325362768)

[1.1.1 Tutoriel ANTLR 3](#__RefHeading___Toc1900_1301357424)

[1.1.2 Le fichier de grammaire (/src/antlr) 4](#__RefHeading___Toc1902_1301357424)

[1.1.3 Fonctionnalités du langage 6](#__RefHeading___Toc226_1432643333)

[1.1.3.1 Intro 6](#__RefHeading___Toc228_1432643333)

[1.1.3.2 Arithmétiques (CRIT*ÈRE #2)* 7](#__RefHeading___Toc230_1432643333)

[1.1.3.3 Variables (CRIT*ÈRE #3)* 9](#__RefHeading___Toc232_1432643333)

[1.1.3.4 Blocs « if » et blocs « for » (CRIT*ÈRE #4) (CRITÈRE #5, SEMI COMPLÉTÉ)* 11](#__RefHeading___Toc234_1432643333)

[1.1.3.5 Objectifs/Crit*ères* non-complétés (CRIT*ÈRES 6,7,8,9)* 13](#__RefHeading___Toc522_1432643333)

[Conclusion 14](#__RefHeading___Toc1904_1301357424)

# Préparation de l’expérimentation

## Sujet

J’ai fait mes recherches sur des projets qui requièrent de la connaissance avancé sur la programmation. Un de ces projets est la conception de ton propre langage de programmation. Créer ton langage de programmation est utile pour répondre à tes besoins, surtout quand tu connais les forces et les faiblesses de ton langage mieux que tout le monde. C’est pourquoi je voudrais essayer de créer mon propre langage de programmation.

## Objectif

Mon projet consiste à créer mon propre langage de programmation qui fonctionnera comme les scripts de Python ou les exécutables de C++, dépendamment des efforts nécessaire pour créer un interpréteur versus créer un compiler.

Pour commencer, je vais créer une librairie standard qui contiendra les fonctions de base pour mon langage, puis je vais coder une application qui présentera ces différentes fonctions.

Le contenu de cette ou ces applications dépend entièrement sur ce que je réussi à implémenter dans ma librairie standard. Une librairie standard de n’importe quel langage de programmation contient de nombreuses composantes sur différents sujets. Voici quelques composantes que je planifie intégrer, en ordre de priorité:

1. Fonctions mathématiques de base (abs, sin, cos, tan, racine, min, max, exposent, tronquer/arrondir, etc.)
2. Fonctions sur les manipulations de strings (longueur du string, remplacer A par B, substring, pad left/right, trim, etc.)
3. Fonctions sur les arrays (array, liste, dictionnaire, etc.)
4. Fonctions sur les inputs (lire la ligne de texte, lire une touche du clavier)
5. Fonctions sur l’interaction du terminal (print, changer la couleur du texte et de l’arrière-plan, cls/clear, etc.)
6. Fonctions audio
7. Fonctions de base de données embarquée dans l’application (SQLite, de la même façon que la librairie SQLite de Python)

Ce n’est pas garanti que je vais toutes les intégrer, mais une bonne portion devrait suffire.

## Critères d’évaluation

1. Le langage doit contenir un shell pour exécuter des commandes en temps réel.
2. Le langage doit supporter les arithmétiques, incluant du bit shifting.
3. On doit pouvoir déclarer des nouvelles variables et assigner des variables existantes.
4. Le langage doit supporter les if statements avec des comparaisons (==, ≥, <, !=, etc.) et des opérateurs logiques (&&, || et !).
5. Le langage doit supporter les loops for et/ou les loops while, incluant les mots-clé break et continue.
6. Le langage doit supporter la déclaration et l’appel des fonctions, incluant des paramètres.
7. Le langage doit supporter la création de classes qui incluent des variables et des fonctions.
8. Le langage doit pouvoir importer du code de d’autres fichiers du même langage, donc des modules.
9. Je dois créer des applications avec mon langage, comme par exemple une librairie standard et des petits jeux (tic-tac-toe, jeu textuel et/ou jeu de labyrinthe « rogue-like »).

# Réalisation de l’expérimentation

## Expérimentation

## Exigences techniques

1. ANTLR (ANother Tool for Language Recognition)
   1. Générateur de parser qui permet de traduire le code en binaires.
2. Python 3.10
   1. Langage de programmation qui sera ma base pour créer la logique de mon langage

## Déroulement

Pour commencer, je dois comprendre le processus de la création d’un langage de programmation. Je vais suivre des tutoriels et lire la documentation d’ANTLR afin de pouvoir définir les syntaxes de mon langage, puis de les rendre fonctionnelle.

## Résultats

### Tutoriel ANTLR

<https://tomassetti.me/learning-build-languages/>

<https://tomassetti.me/building-a-simple-expression-language/>

Le tutoriel de tomassetti sur la création d’un langage m’a donné la direction que j’avais besoin pour trouver les bons outils. Cependant, afin de ne pas copier le travail de tomassetti, j’ai décidé de ne pas utiliser Gradle, Kotlin ou son fichier de grammaire pour mon projet.

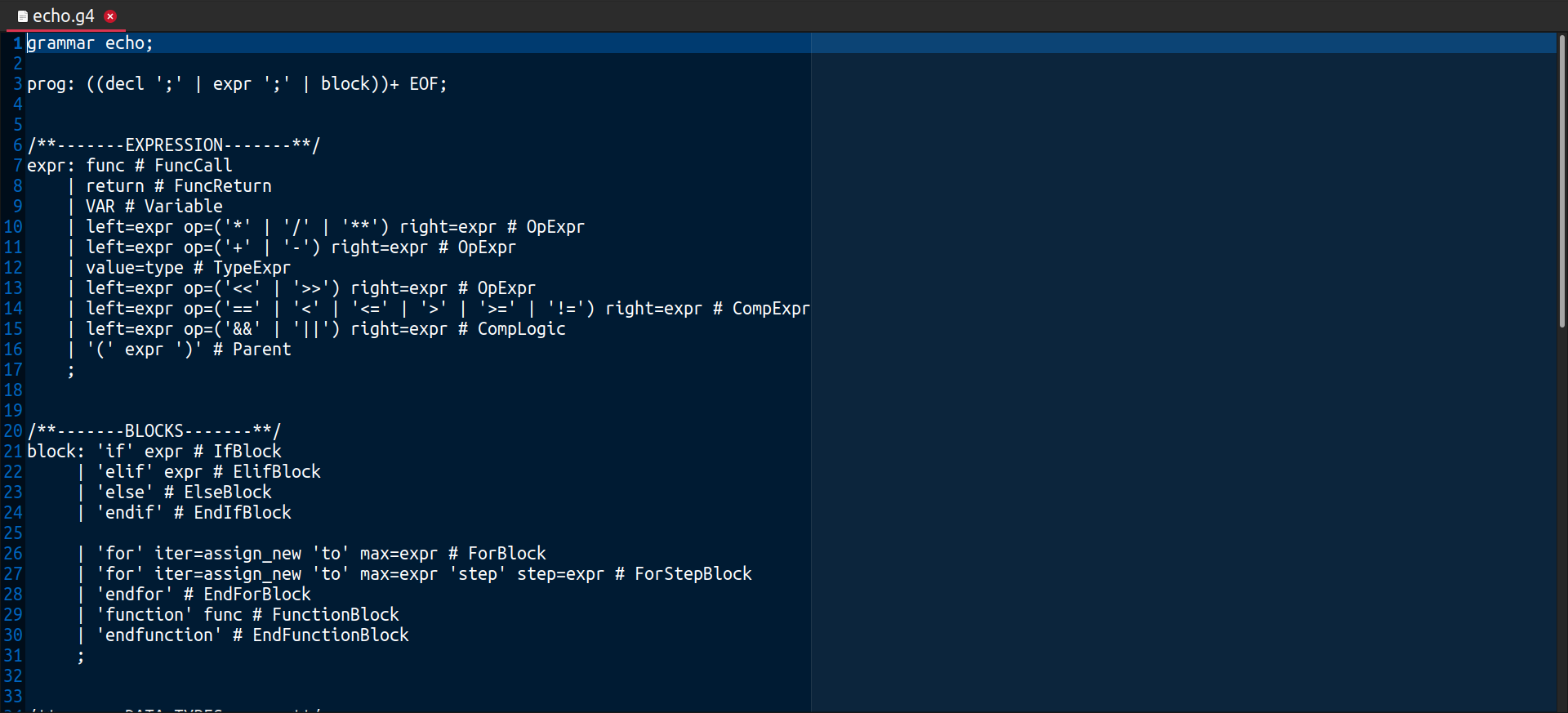
<https://www.youtube.com/playlist?list=PL5dxAmCmjv_4FGYtGzcvBeoS-BobRTJLq>

J’ai suivi un tutoriel sur l’utilisation d’ANTLR. J’ai appris à définir les syntaxes dans le fichier de grammaire et à générer le parser grâce à cette collection de vidéos.

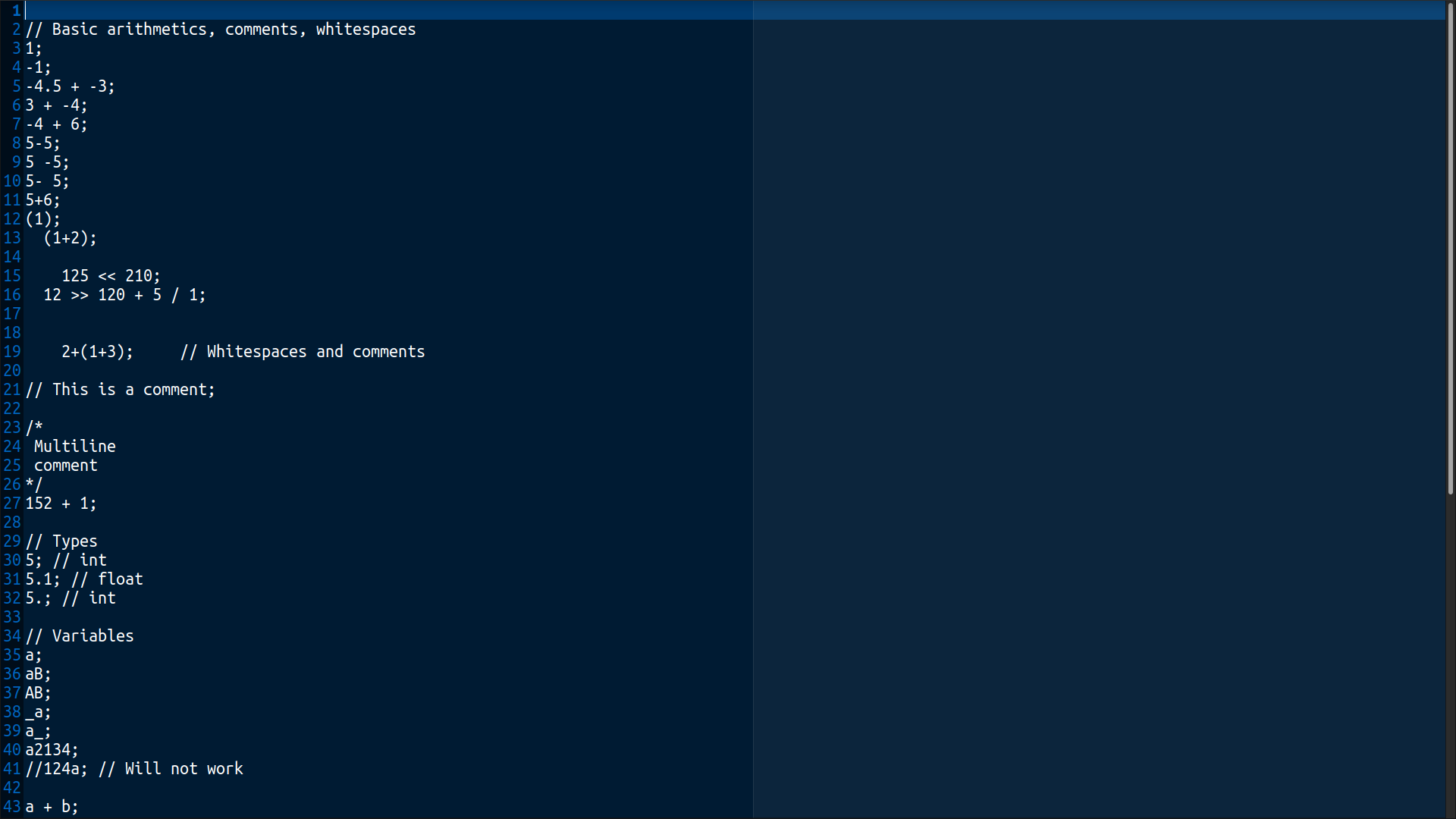
<https://github.com/antlr/antlr4/blob/master/doc/index.md>

Je me suis aussi référé à la documentation officielle d’ANTLR. Par contre, la documentation pour la version Python du parser était limité, donc j’ai dû régler les bogues de mon fichier de grammaire par moi-même la majorité du temps.

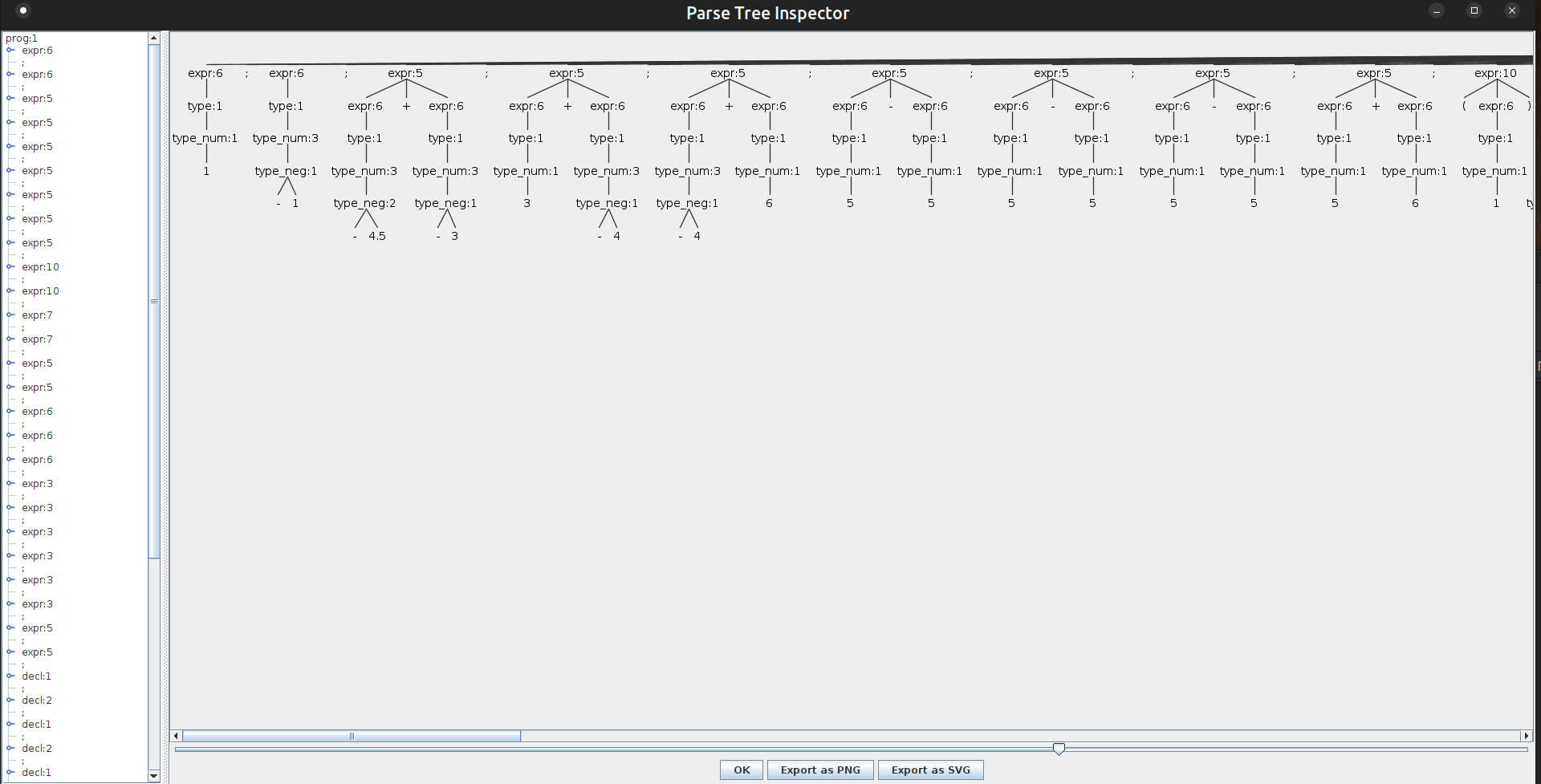
### Le fichier de grammaire (/src/antlr)



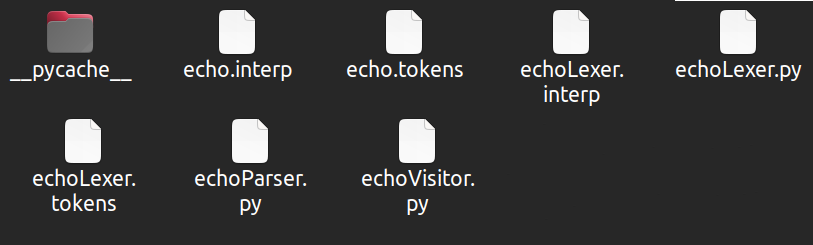
Afin de générer le parser pour mon langage, je dois créer un fichier de grammaire (.g4) qui définit ses syntaxes. Je définis les syntaxes pour les types de données (int, float, bool, string), une expression (*1 + 5, a \* 2*, *(124 – 12) \* (a – abs(-5)*, etc.), un bloc if (if 1 == True, elif 1 <= 2, else, endif), un bloc for (for i = 0 to 10), une déclaration/assignation de variable, les commentaires, etc.



J’ai créé des fichiers tests pour tester les syntaxes définies par Echo.g4. Ces fichiers tests sont seulement conçus pour tester la syntaxe du langage et non sa logique. On peut tester chaque fichier en exécutant la commande suivante (Nécessite ANTLR) : **antlr4-parse echo.g4 prog -gui tests/test1.ec**



Nous avons ce résultat. les syntaxes de test 1 et test 2 fonctionnent correctement depuis la dernière itération du fichier de grammaire (Il se peut qu’un des tests retourne « token recognition error at ‘.’ ». C’est normal puisque je voulais supporter les nombres à virgule n’ayant pas de chiffre après la virgule, donc « 5. ». Malgré qu’il dit que c’est une erreur, le parser continue de fonctionner comme prévu).



Maintenant qu’on a complété les syntaxes (on peut toujours en ajouter plus tard), il faut générer le parser avec la commande suivante : **antlr4 echo.g4 -Dlanguage=Python3 -visitor -no-listener -o ../bin/**

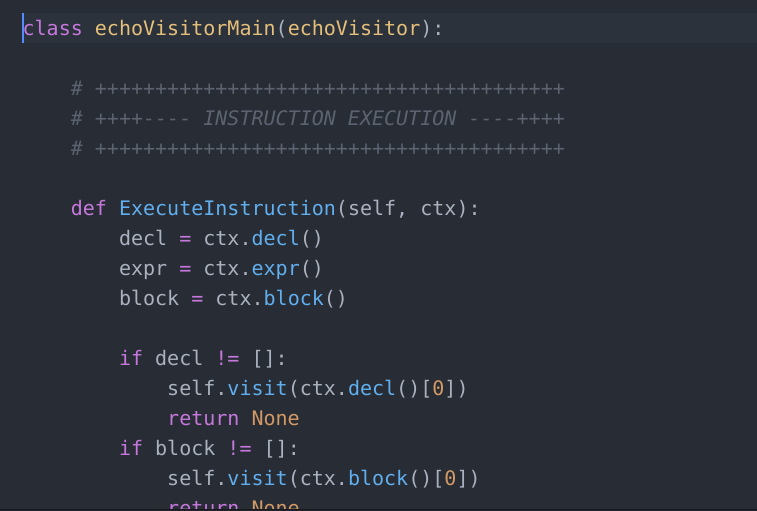
Ceci va générer les fichiers Python ci-dessus dans le dossier « bin ».

### Fonctionnalités du langage

#### *Intro*

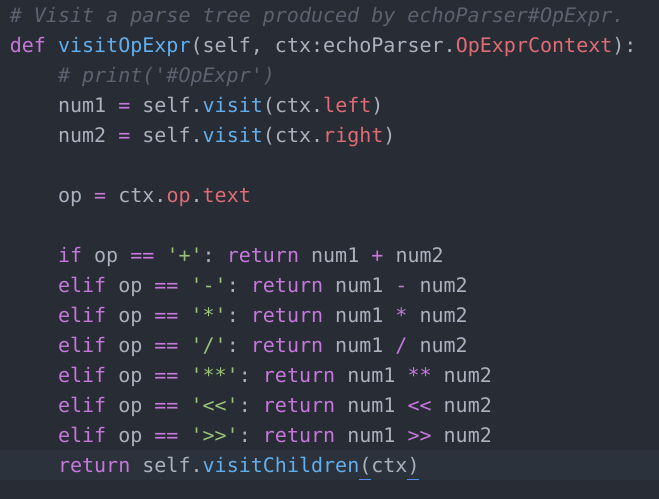


Maintenant, il reste à rendre le langage fonctionnel. Le parser t’offre un script Visitor (Nommé echoVisitor.py puisque le langage se nomme Echo) qui contient une fonction pour chaque token du langage. Ceci te permet de coder la logique de ton langage lorsque le parser interprète le code Echo. Il y a donc une fonction pour un « if statement », une fonction pour une expression (1 + 1, a – 5, (1 \* 10) / (b \* c), etc.), une fonction pour un nombre entier, etc. Ce que ces fonctions retournent dépendent entièrement du résultat que tu t’attends pour chaque token.



Puisque echoVisitor.py est automatiquement recréé à chaque fois que tu génères une nouvelle version du parser, on ne peut pas directement coder sur ce script. Il faut donc créer un script Python séparé (nommé echoVisitorMain.py dans ce cas-ci. Son nom n’est pas important) qui contient une classe qui hérite de la classe echoVisitor. C’est là qu’on va coder la logique du langage.

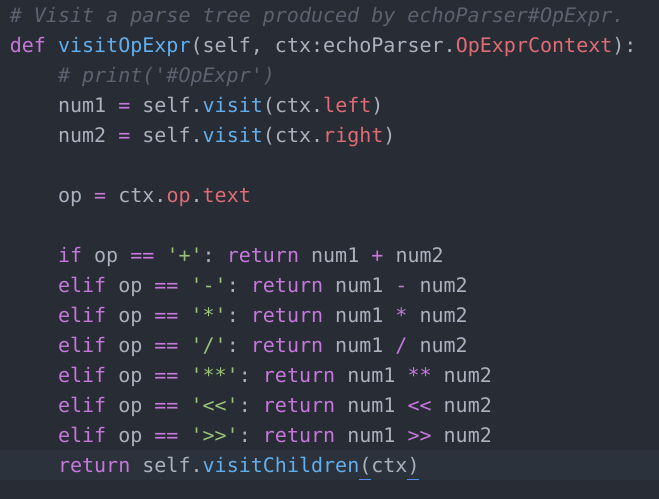
#### Arithmétiques (CRIT***ÈRE #2)***



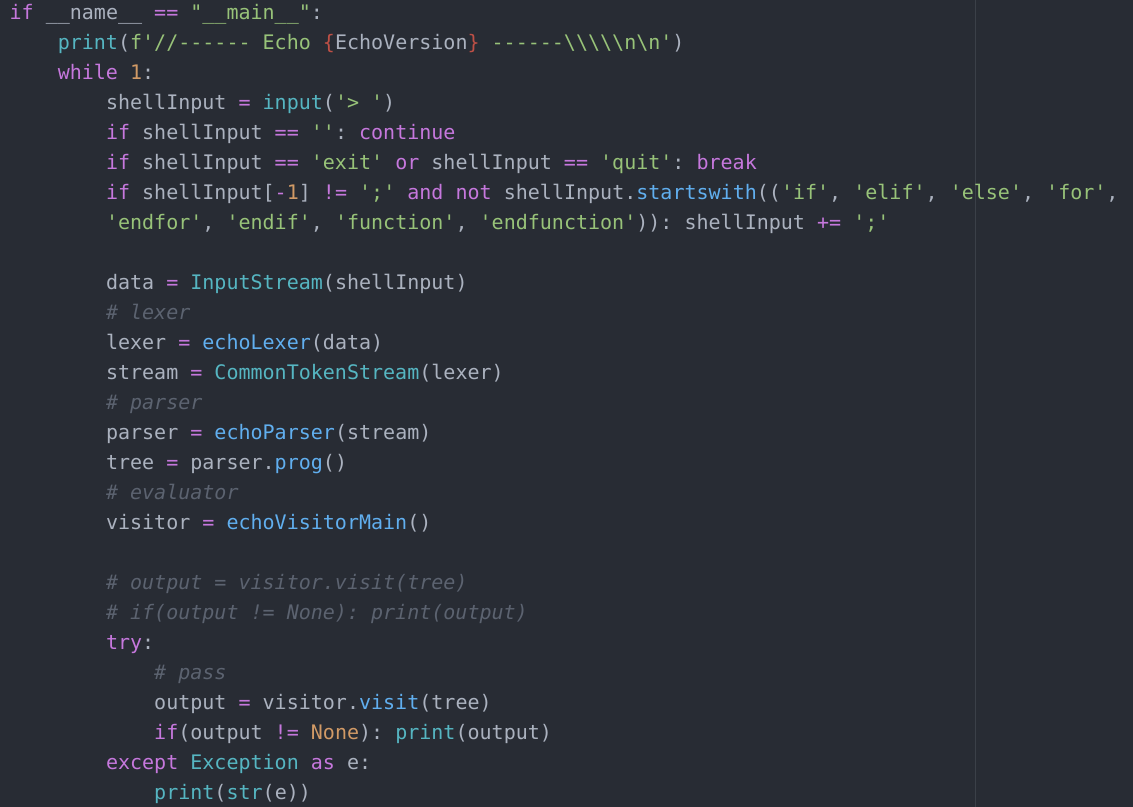
Commençons par la base. En ce moment, notre langage contient seulement des syntaxes pour les arithmétiques. On va donc coder le token qui est supposé retourner le résultat entre deux nombres à partir d’un opérateur spécifique.



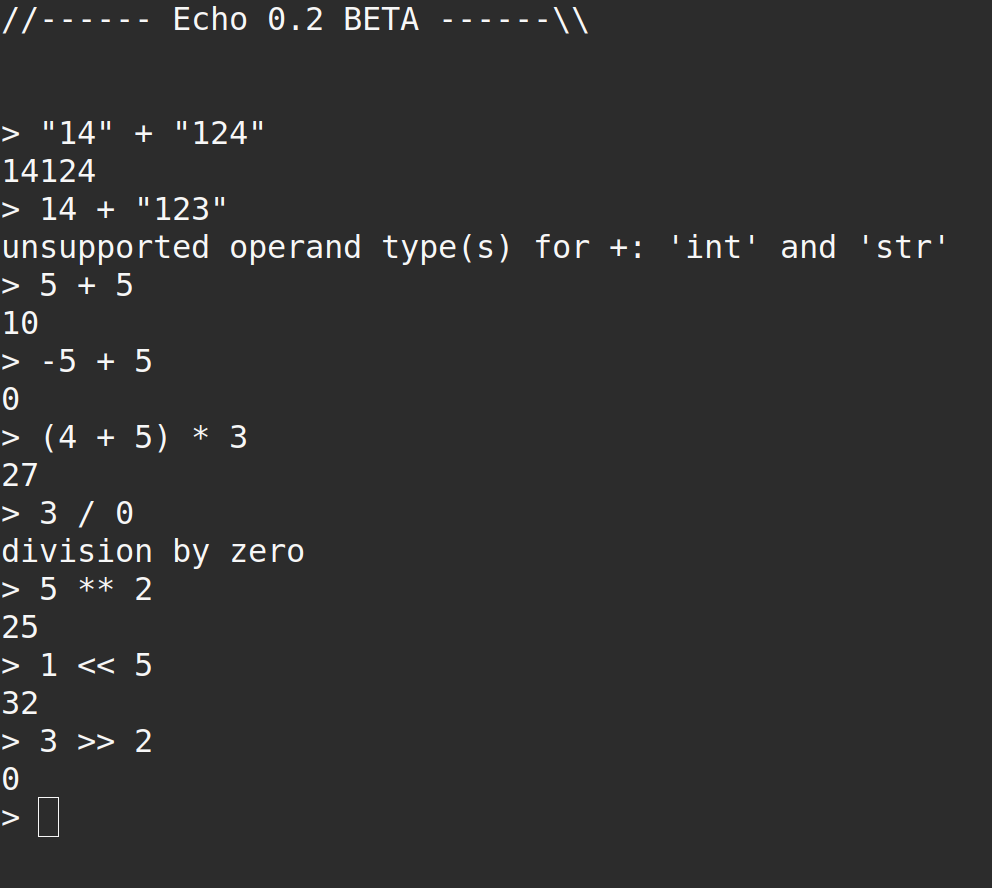
D’abord, nous devons convertir les nombres puisqu’ils sont seulement des strings en ce moment. Le fichier de grammaire est capable de distinguer un nombre entier d’un nombre décimal, d’une chaîne de caractères, d’un booléen, etc. Donc, les self.visit() de l’image précédent va « visiter » les autres fonctions de tokens qui, dans ce cas-ci, va appeler les fonctions illustrées dans l’image ci-dessus, puis retourner le nombre converti en int, float, string, etc.



Maintenant qu’on a converti les deux nombres à calculer, il nous reste l’opérateur. Pas besoin de « visiter » celui-ci puisqu’on va l’interpréter nous-même. Grâce à notre switch-case de manière Python, nous retournons un résultat si l’opérateur correspond à un opérateur valide. Nous avons droit à une somme, une soustraction, un produit, une division, un exposant, un bitshift gauche et un bitshift droit.

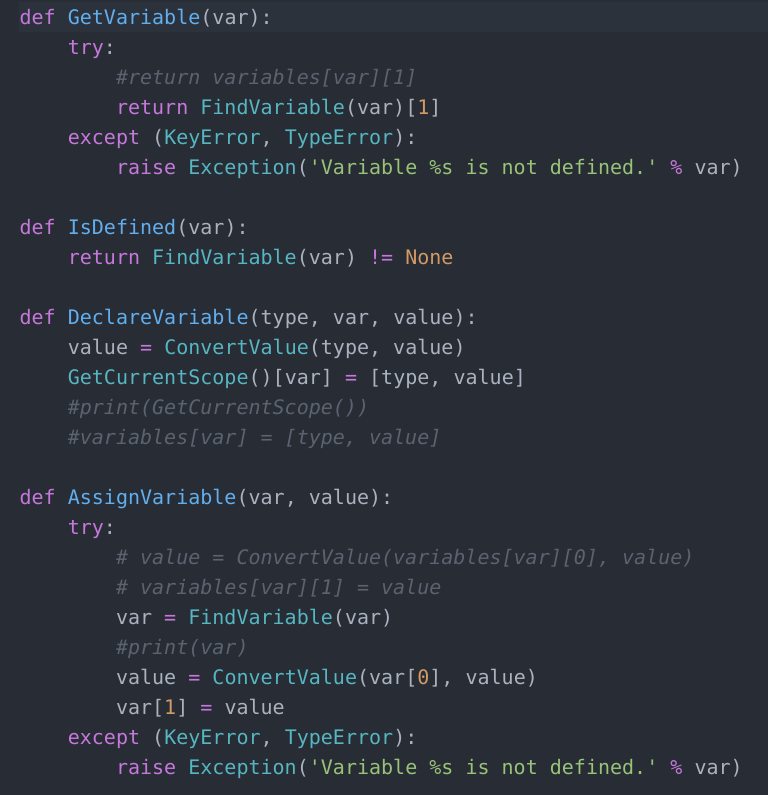


Coder les tokens de notre langage est une chose, mais comment allons-nous tester que tout fonctionne correctement? Le meilleur moyen est de créer un shell. Grâce au tutoriel sur Youtube que j’ai noté au début du chapitre, j’ai créé un script Shell.py qui va lire le code que je rentre comme input et l’envoyer au parser afin de l’interpréter et de retourner un résultat. Ceci règle enfin le dilemme entre créer un interpréteur ou un compilateur puisque le shell s’occupe d’interpréter mon code **(CRITÈRE #2)**.



Grâce au shell, nous pouvons tester les arithmétiques. Après quelques ajustements, on peut voir qu’il retourne le bon résultat pour chaque opérateur, y compris une concaténation entre deux strings. Il y a par contre quelques limitations, comme par exemple tenter de calculer un nombre entier avec une string, comme un peut voir dans la deuxième commande **(CRITÈRE #2)**.

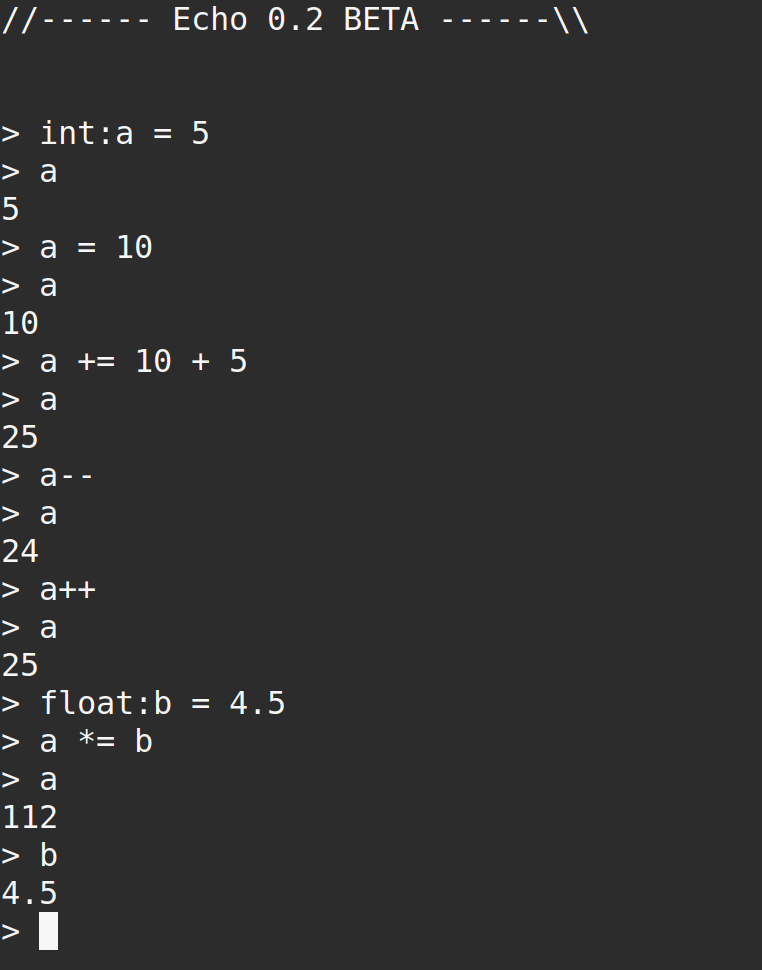
#### Variables (CRIT***ÈRE #3)***



Faire des calculs mathématiques sert à rien si on ne peut pas stocker les résultats quelque part, alors il nous faut des variables. Déclarer, assigner et retourner le contenu d’une variable est un processus plus complexe que des simples arithmétiques. Afin de faciliter la tâche, j’ai créé le script echoLib.py qui va contenir des fonctions et des variables pour les fonctionnalités avancées du langage, incluant la gestion de variables.

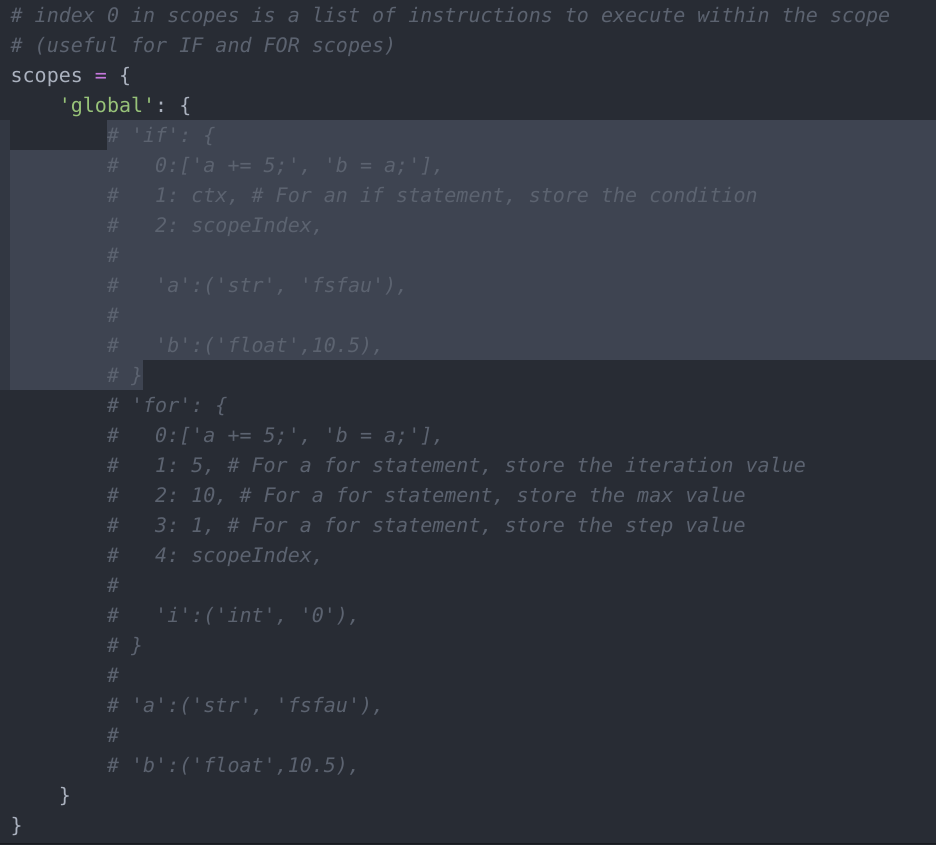


De retour à echoVisitorMain, j’appelle les fonctions que j’ai créé dans echoLib afin de déclarer, assigner et lire une variable. J’ai aussi intégré une incrémentation et une décrémentation comme a++ et a--, ainsi que des assignations avec des opérateurs comme +=, -=, \*=, etc.



Comme on peut voir dans le shell, les variables fonctionnent et peuvent être utilisés comme valeur arithmétique comme on peut voir dans la commande « a \*= b ».

#### Blocs « if » et blocs « for » (CRIT***ÈRE #4) (CRITÈRE #5, SEMI COMPLÉTÉ)***

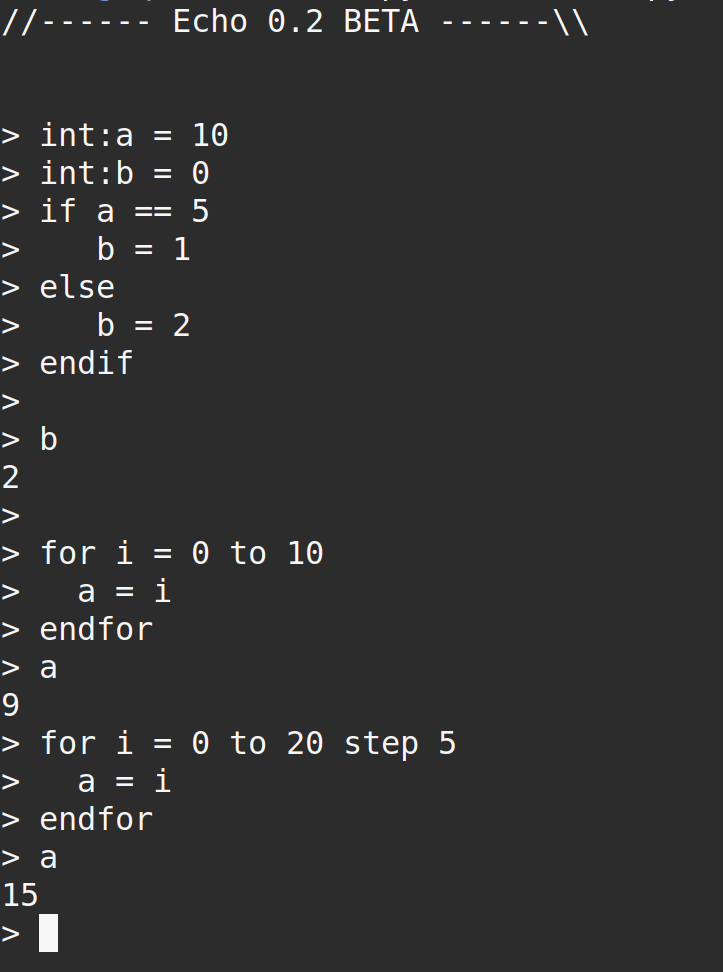


Les blocs « if » et les blocs de boucle étaient difficile à intégrer. Dans echoLib, j’ai dû coder un système de « scopes » afin de stocker les variables temporaires, les instructions à exécuter pour le bloc, ainsi que des valeurs diverses qui sont utiles pour le bloc en question (ex : l’instruction qui retourne la condition requise pour exécuter un bloc « if », la valeur initiale d’un bloc « for », etc.).

‘global’ est le « scope » primaire. Il peut contenir des variables et d’autres « scopes ». Il est le seul « scope » où que les instructions sont exécutées instantanément au lieu d’être stockée dans une liste jusqu’à la fin du « scope » (voir le paragraphe suivant)

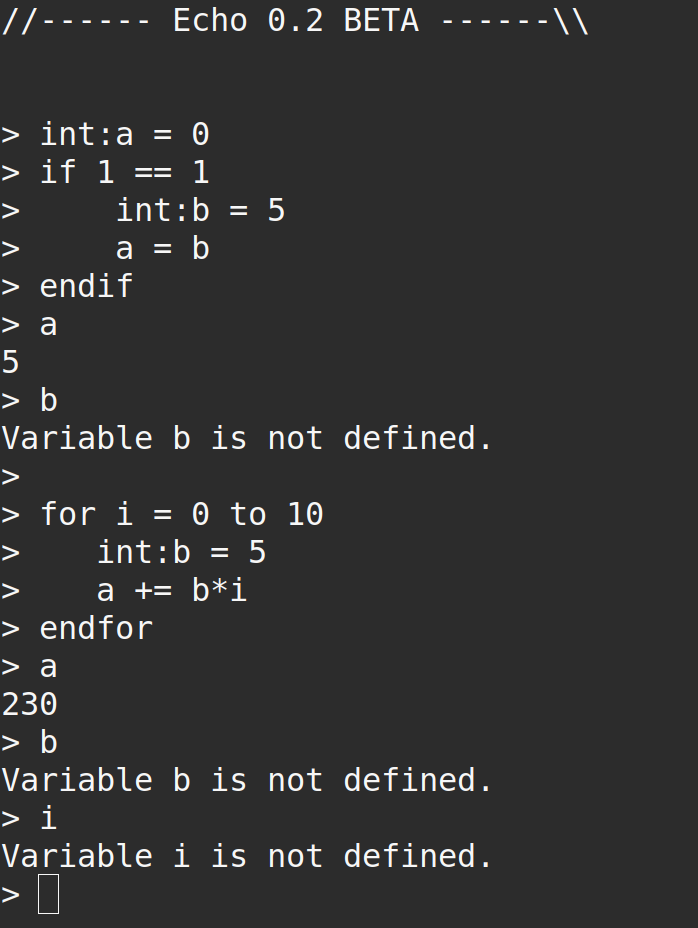


Comme mentionné, les instructions sont immédiatement exécutées quand nous sommes dans le « scope » global **[ExecuteInstruction()]**. Cependant, si on commence un bloc « if » ou un bloc « for », nous sommes maintenant en mode bloc, où que toutes les instructions que nous rentrons sont stockées dans une liste jusqu’à ce qu’elles sont toutes exécutées lorsqu’on ferme le bloc avec un « endif » ou un « endfor » **[ExecuteBlockInstructions()]**. L’exécution des instructions d’un bloc n’est pas parfait, mais il fait le boulot.



Le bloc « if » dans l’image nous montre que seulement l’instruction « else » a été exécutée puisque « a » n’équivaut pas à 5, donc « b » se fait assigner la valeur 2. Les blocs « for » sont aussi visiblement fonctionnels et on peut voir que la valeur maximum est exclusif, donc on finit avec 9 au lieu de 10 et 15 au lieu de 20.

En ce qui concerne les boucles « for », je n’avais pas encore codé les commandes « break » et « continue » comme j’avais mentionné dans le critère #5.



Les variables sont automatiquement supprimées lorsqu’on sort du « scope », donc on ne revoit plus les variables « b » et « i » lorsque les blocs « if » et « for » ont terminés, tandis que la variable globale « a » est toujours là.

Le système de « scopes » a cependant des limitations majeures. Par exemple, je n’ai pas réussi à intégrer des blocs « nested » à l’intérieur de d’autres blocs sans qu’il soit rempli de bogues comme être « softlocked » en mode bloc parce qu’il ne parvient pas à sortir du « scope » correctement sous le capot.

*En d’autres mots, ajouter des blocs « if » et des blocs « for » dans le scope « global » fonctionne correctement, mais il n’est pas recommandé d’ajouter des blocs à l’intérieur de d’autres blocs!*

#### Objectifs/Crit***ères*** non-complétés (CRIT***ÈRES 6,7,8,9)***

J’étais ambitieux. Je croyais pouvoir au moins commencer ma librairie standard, mais je n’avais même pas encore fini ou commencé les fonctionnalités nécessaire pour y arriver.



Le code pour la déclaration et l’appel de fonctions était partiellement complété, mais il n’était pas encore testé. C’était le dernier bout de code que j’ai fait. **(CRITÈRE #6)**.

Je voulais aussi intégrer des classes qui contiennent des fonctions et/ou des variables, mais ce critère était de basse priorité et je n’ai pas trop pensé à celui-là **(CRITÈRE #7)**.

Importer le code d’un ou plusieurs fichiers externes afin de pouvoir coder des modules était aussi une étape nécessaire pour commencer la librairie standard. Je n’avais pas encore commencé à intégrer la fonctionnalité cependant **(CRITÈRE #8)**.

Finalement, les applications codées par Echo étaient la dernière étape, suivie par la création de ma librairie standard. Encore une fois, je n’avais rien commencé à ce sujet **(CRITÈRE #9)**.

# Conclusion

|  |  |
| --- | --- |
| **Critère** | **Statut** |
| 1 : Shell | Complet |
| 2 : Arithmétiques | Complet |
| 3 : Variables | Complet |
| 4 : « If statements » | Complet |
| 5 : « For loops » | Complet, excluant les mots-clé « break » et « continue » |
| 6 : Fonctions | Incomplet |
| 7 : Classes | Incomplet |
| 8 : Modules | Incomplet |
| 9 : Applications (librairie standards et jeux) | Incomplet |

Pour conclure, mon projet de création d’un langage de programmation est partiellement un succès. J’ai fait beaucoup de progrès pour rendre le langage fonctionnel, mais j’avais des critères qui ne pouvaient pas être accompli de manière réaliste en seulement trois mois. Quand j’avais mis la librairie standard comme objectif principal, je n’avais pas prit compte de l’effort requis pour créer le langage comme tel.

Cependant, je suis fier de ce que j’ai accompli et je pourrais utiliser les connaissances que j’ai acquis au cours de cette veille pour créer un meilleur langage de programmation à l’avenir.