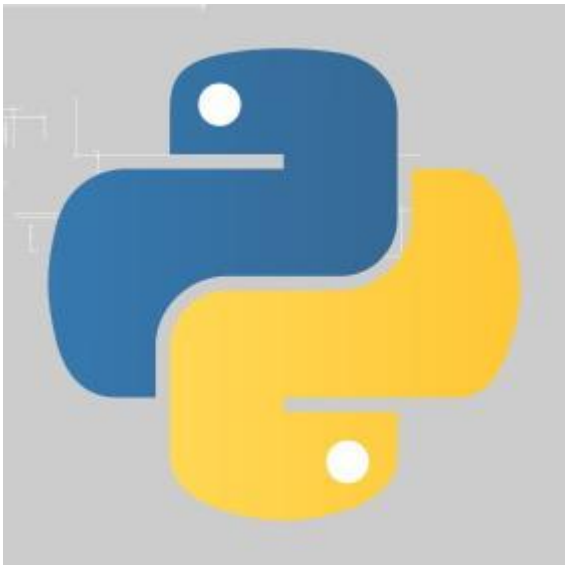
Rapport de langages et compilateur



Auteurs : Lionel Heim et Gaël Jobin
Filière : INF3gen-i
Module : Langage et compilateur
Destinataire : Mr. Ghorbel Hatem

Saint-Imier, le 18 janvier 2011

TABLE DES MATIÈRES

Résumé

Un compilateur, selon Wikipedia, est un « programme informatique qui traduit un langage, le langage source, en un autre, appelé le langage cible, en préservant la signification du texte source ».

D'un point de vue informatique, pouvoir créer notre propre langage de programmation ainsi que le compilateur associé, nous a permis de mettre en pratique ce qui a été vu durant le cours de langage et compilateur. De plus, cela nous a fait comprendre le fonctionnement général d'un compilateur et réalisé ce qui se passait lorsque l'on compile du C, C++, Java,…

Nous avons suivi le processus vu lors des différents laboratoires. Il a fallut commencer par inventer un nouveau langage de programmation. Puis, l'analyse syntaxique a pris passablement de temps ; c'est elle qui contient toutes les règles permettant de découper les différentes parties de notre langage, sans ambigüité. L'analyse sémantique a posé quelques problèmes, notamment concernant le typage des variables. Finalement, l'interpréteur a permis d'exécuter notre programme.

Introduction

Durant notre 3ème année de formation d'ingénieur en informatique, nous avons dû réaliser un projet à la fin du semestre d'automne. Le sujet ainsi que l'environnement de développement étaient fixes : Il fallait inventer un nouveau langage de programmation et écrire, en python, le compilateur associé. Concernant l'exécution du programme, nous avons eu le choix entre l'interpréter avec python ou générer du code pour une machine virtuelle de notre choix, voire même de notre propre création.

Pour notre langage, nous nous sommes beaucoup inspirés du langage C/C++ et avons choisi de créer un interpréteur en python plutôt que de générer du code pour une machine virtuelle.

Au cours de ce rapport, nous vous présenterons les différentes phases du projet, pour débuter nous allons présenter notre langage ainsi que ses particularités. Puis, les différents choix que nous avons pris ainsi que les différents problèmes rencontrés.

Pour terminer, une conclusion dans laquelle nous effectuerons une synthèse du travail accompli.

Notre langage

Comme dit précédemment, notre langage s'inspire énormément des bases C/C++ comme les variables ou les boucles. De par la complexité et le temps mis à disposition, nous n'avons pas traité des aspects tels que les objets ou les fonctions.

Voici un exemple de programme :

```
^Ceci est un commentaire^

principale()
debut

        entier a = 4 * 5$
        a++$
        si a dif 0
        debut
                afficher(a)$
        fin
        sinon
        debut
                afficher("a vaut 0")$
        fin
fin
```

À la façon du `main() {…}`, un programme commencera toujours pas `principale() debut … fin`.

Les commentaires (`/* … */` en C) sont encadrés par des `^`.

Les opérations mathématiques

Elles sont aux nombres de cinq :

- Addition (`+`)
- Soustraction (`-`)
- Multiplication(`*`)
- Division (`/`)
- Modulo (`%`)

La multiplication, division et le modulo sont prioritaires sur l'addition et la soustraction.

Il existe aussi deux autres opérations à part :

- Incrémentation (`++`)
- Décrémentation (`--`)

Compatibilité des opérations et casting

Les entiers (`entier`) supportent toutes les opérations ci-dessus entre eux. Avec un réel comme deuxième argument, l'opération s'effectue sans problème car le réel sera converti en entier et sera toujours arrondi vers le bas.

```
entier a = 2+2.0-2*2/2%2$
a++$
a--$
```

Les réels (`reel`), à la différence des entiers, ne supportent pas le modulo. Un entier peut être utilisé au coté d'un réel mais sera converti dans le même type que ce dernier.

```
reel a = 2.0+2.0-2.0*2.0/2.0$
a++$
a--$
```

Les caractères (`charactere`) ne supportent aucunes de ces opérations. Par contre, lors de la déclaration ou l'assignation d'un caractère avec, comme argument, une chaine de caractères, seul le premier caractère de la chaine sera attribué à la variable.

```
charactere a = "test"$        ^a vaudra 't'^
```

Les chaines de caractères (`text`) sont une particularité de notre langage. Elles peuvent être additionnées à tous les autres types de variables produisant ainsi une concaténation. Elles peuvent être soustraites par une autre chaine ou un caractère ce qui aura pour effet de supprimer toutes les occurrences dans la variable de base. La multiplication, comme en python, permet de concaténer une chaine à lui-même un nombre « n » (uniquement un entier) de fois. La division d'une chaine par un entier « n » divise la chaine en « n » partie et ne garde que la première partie si « n » est positif ou la dernière partie si « n » est négatif. Le reste des opérations n'est pas compatible.

```
text a = "ceci est un tes" + 't' + 2$
a = a + " concluzant"$        ^ceci est un test2 concluzant^
a = a - 'z'$                  ^ceci est un test2 concluant^
a = a - "un test2 "$          ^ceci est concluant^
text b = a/4$                 ^ceci^
a = a/-2$                     ^concluant^
```

Les variables

Il faut savoir qu'il y a quatre types de variables et que chacune se voit attribuer une valeur par défaut lors de la déclaration sans spécifier de valeur :

- Les entiers (par défaut : `0`)
- Les réels (par défaut : `0.0`)
- Les caractères (par défaut : `''`)
- Les chaines de caractères ou le texte (par défaut : `""`)

L'assignation d'une valeur à une variable peut être faite de deux manières, soit avec `=`, soit avec `egal`.

Exemple de définition des variables et d'assignation de valeurs :

```
entier a$
reel b = 23.5$
charactere c egal 'a'$
text coucou egal "coucou"$
b egal 5.0$
c = 'c'$
```

Les conditions

Nous avons une seule condition, le `if` que nous appelons ici `si`. Différentes comparaisons peuvent être faites :

- Egal à a (`==` ou `ea`)
- Différent de (`!=` ou `dif`)
- Plus grand que (`>` ou `pg`)
- Plus petit que (`<` ou `pp`)
- Plus grand ou égal à (`>=` ou `pge`)
- Plus petit ou égal à (`<=` ou `ppe`)

```
si a dif 0
debut
        si b == a
        debut
                afficher("OK")$
        fin
fin
sinon
debut
        afficher("a vaut 0")$
fin
```

Les boucles

Il existe trois types de boucle possibles :

- Le `while` traduit par `tantque … debut … fin`
- Le `do while` qui sera `faire debut … fin tantque …`
- Le `for` qui donne `pour (…) debut … fin`

```
tantque a<0
debut
        a = a + 1 $
fin
```

```
faire
debut
        a = a + 1 $
fin
tantque a<10
```

```
pour
(entier x = 0 $ x pp 10 $ x = x + 1)
debut
        a = a - 1 $
fin
```

Les particularités

Afficher

Comme le `printf()` du langage C, notre langage dispose d'une unique fonction `afficher()` permettant, comme son nom l'indique, d'afficher un contenu dans la console.

```
afficher("Tes"+'t'+2.0)$      ^affiche "Test2"^
```

Echanger

Un opérateur spécial (`<->`), mais pratique, permet l'échange du contenu de deux variables de même type.

```
text a = "test1"$
text b = "test2"$
a <-> b$
afficher(a+b)$          ^affiche "test2test1"^
```

Analyse lexicale

La première étape de notre compilateur fut de créer une analyse lexicale fonctionnelle et complète. Pour cela nous avons dans un premier temps regroupé tous les « tokens » nécessaires pour notre langage.

Type de variable :

entier
reel
charactere
text

Mots réservés :

si
sinon
faire
tantque
pour
debut
fin
egal
afficher
principale
pp
ppe
pg
pge
dif
ea

Données du programme

ENTIER_VAL
REEL_VAL
CHARACTERE_VAL
TEXT_VAL
COMPARE
ECHANGE
INC
DEC
VAR
ADD_OP
MUL_OP
END_LINE

Les données du programme sont des résultats d'expression régulière.

La liste de « tokens » correspond au regroupement des trois rubriques ci-dessus, c'est-à-dire, aux types de variables + mots réservé + données du programme. L'analyse lexicale nous a permis d'avoir un fichier de code source sous forme de lexème.

Nous avons réalisé le code pour générer les lexèmes dans le fichier « lexicale.py ». Nous avons passé passablement de temps à réaliser cette partie pour être avoir une base correct pour la suite.

Exemple d'exécution

Code :

```
principale() debut

pour (entier a = 0 $ a < 2 $ a++)
debut
       afficher a $
fin

fin
```

Résultat :

```
line 1:
PRINCIPALE(principale)
line 1: (()
line 1: )())
line 1: DEBUT(debut)
line 3: POUR(pour)
line 3: (()
line 3: ENTIER(entier)
line 3: VAR(a)
line 3: =(=)
line 3: ENTIER_VAL(0)
line 3: END_LINE($)
line 3: VAR(a)
line 3: COMPARE(<)
line 3: ENTIER_VAL(2)
line 3: END_LINE($)
line 3: VAR(a)
line 3: INC(++)
line 3: )())
line 3: DEBUT(debut)
line 4: AFFICHER(afficher)
line 4: VAR(a)
line 4: END_LINE($)
line 5: FIN(fin)
line 7: FIN(fin)
```

Analyse syntaxique

Après l'analyse lexicale, notre analyse syntaxique est constituée d'une grammaire permettant de mettre sous forme d'arbre syntaxique les lexèmes.

Voici la grammaire que nous avons développée pour notre langage :

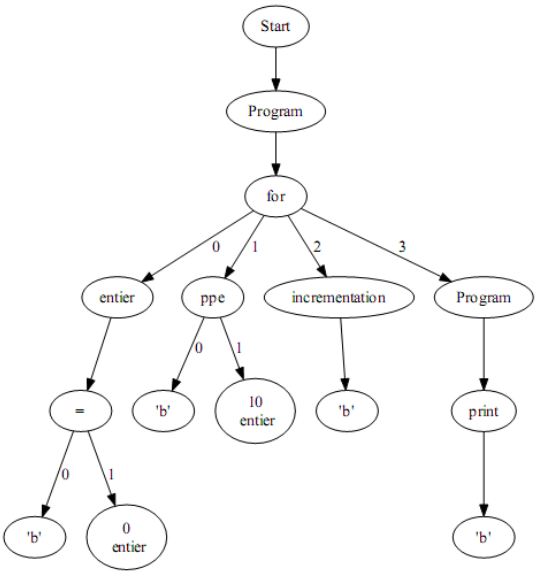
```
start : PRINCIPALE '(' ')' DEBUT program FIN
program : statement
        | statement program
statement : assignation END_LINE
          | affichage END_LINE
          | declaration END_LINE
          | surplace END_LINE
          | test
          | boucle
affichage : AFFICHER expression
boucle : FAIRE DEBUT program FIN TANTQUE condition
       | TANTQUE condition DEBUT program FIN
       | POUR '(' declaration END_LINE condition END_LINE assignation ')'
DEBUT program FIN
       | POUR '(' assignation END_LINE condition END_LINE assignation ')'
DEBUT program FIN
       | POUR '(' declaration END_LINE condition END_LINE surplace ')'
DEBUT program FIN
       | POUR '(' assignation END_LINE condition END_LINE surplace ')'
DEBUT program FIN

assignation : VAR '=' expression
            | VAR EGAL expression

declaration : typevariable VAR '=' expression
            | typevariable VAR EGAL expression
            | declaration :  typevariable VAR


typevariable : REEL
             | ENTIER
             | CHARACTERE
             | TEXT

valeur : CHARACTERE_VAL
       | TEXT_VA

num : REEL_VAL
    | ENTIER_VAL

comparetextuel : PP
               | PPE
               | PG
               | PGE
               | DIF
               | EA


condition : expression COMPARE expression
          | expression comparetextuel expression
          | '(' condition ')'
```

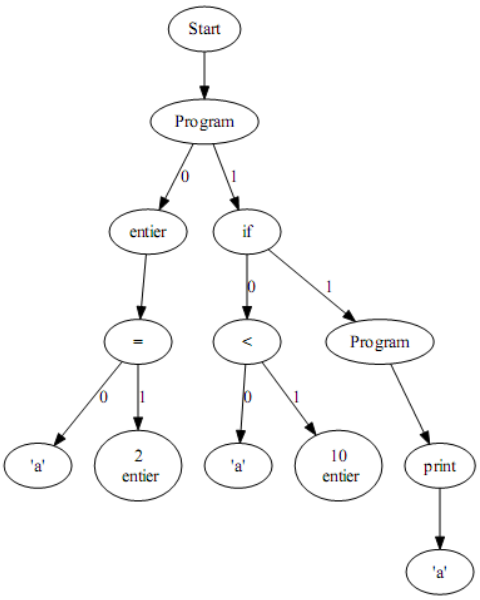
```
test : SI condition DEBUT program FIN
     | SI condition DEBUT program FIN SINON DEBUT program FIN

expression : nom
           | valeur
           | VAR
           | expression MUL_OP expression
           | expression OPP expression
           | '(' expression ')'

surplace : incrementation
         | decrementation
         | echange

incrementation : VAR INC
decrementation : VAR DEC
echange : VAR ECHANGE VAR
```

Exemple d'arbre syntaxique

```
principale() debut

pour (entier b = 0 $ b ppe 10 $ b++)
debut
      afficher(b) $
fin
fin
```

On peut voir que le nœud `for` regroupe tous les éléments utiles à la boucle `for`, c'est-à-dire, la déclaration, la condition, l'incrémentation et le programme qui sera exécuté.






```
principale() debut

entier a = 2$
si a < 10 debut
      afficher(a) $
fin
fin
```

Ici, on représente un nœud `if` (condition `si` de notre langage). Ce nœud contient la condition ainsi que le code à exécuter.

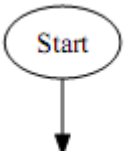
Cette grammaire nous permet de facilement gérer les différentes fonctionnalités telles que les boucles ou les conditions. Nous arrivons facilement à créer des arbres avec des imbrications (`if` dans un `if`).

Génération de l'arbre syntaxique

Pour construire l'arbre syntaxique, nous avons utilisé la librairie AST vu en cours et l'avons passablement complété avec des nœuds supplémentaires pour pouvoir gérer tous nos « tokens ».

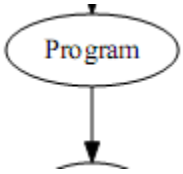
Cette librairie gère tous les types de nœud qui nous sont utiles. Nous avons implémenté le code utile pour réaliser cela dans le fichier « parseur.py ».

StartNode



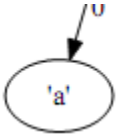
Il représente l'entrée du programme (`main()` en C). Dans notre cas, il est représenté par l'étiquette « Principale ». Le but d'avoir un nœud de ce type est de pouvoir ajouter des fonctions. Malheureusement, le temps ne nous l'a pas permis.

ProgramNode



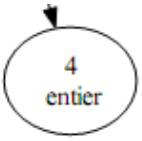
C'est un nœud très important car il contient tous les types d'opérations, boucles, tests, incrémentations, etc…

TokenNode



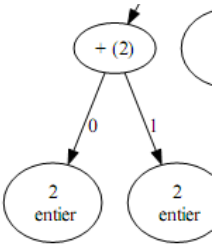
Ce nœud représente le nom d'une variable.

ValeurNode



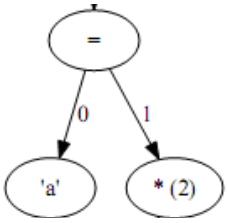
Nous avons différentié les variables et leurs contenus des valeurs mises directement dans le code. Une valeur simple a un type pour pouvoir gérer le type de donnée par la suite.

OpNode



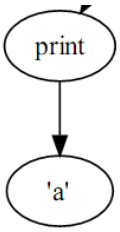
Représente toutes les opérations mathématiques disponibles dans notre langage. Ces deux fils sont les valeurs à traiter et sont type est l'opération à effectuer.

AssignNode



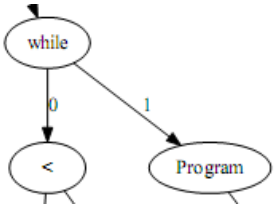
L'assignation est représentée par le signe égal. Le fils gauche est toujours une variable et le fils droit une expression.

PrintNode



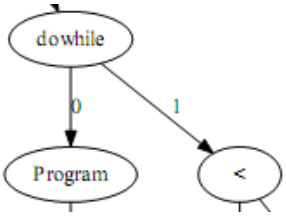
Fonction `afficher()` de notre langage permettant d'afficher une expression dans la console.

WhileNode



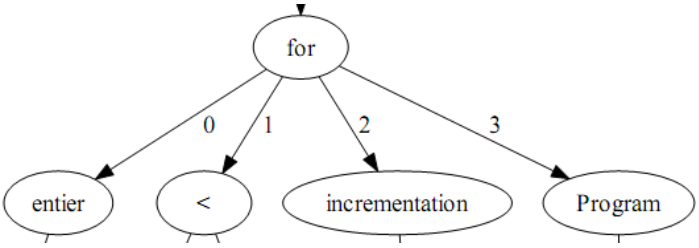
La boucle `while()` commence toujours par une condition. C'est pour cela que nous avons choisi de mettre la condition comme fils gauche et le code a exécuté dans le fils droit.

DoWhileNode



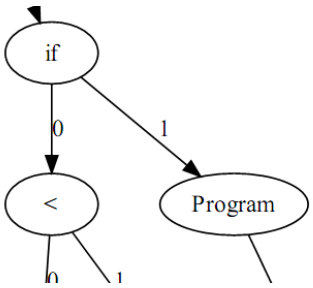
À l'inverse de la boucle `while()`, la boucle `do..while()` exécute une fois le code avant de tester si la condition est valable.

ForNode



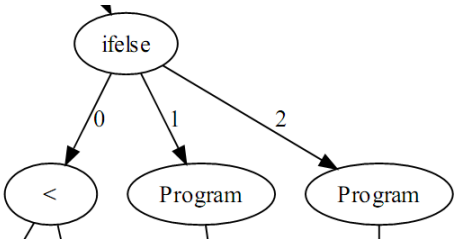
Une boucle `for` est représenté par 4 éléments, une déclaration / initialisation, une condition de sortie, une opération de répétition et un code à exécuter.

IfNode



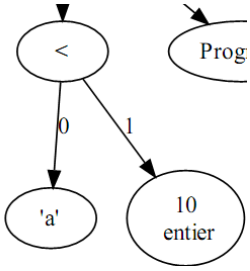
Un test sans close « sinon » est composé d'une condition et d'un programme.

IfElseNode



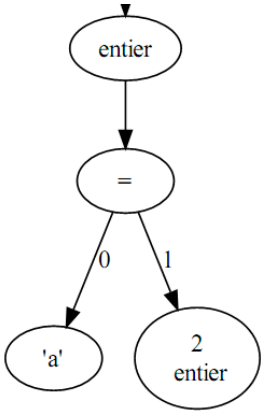
Un test avec une close « sinon » est composé du test et de deux programmes, un pour chaque résultat du test.

ConditionNode



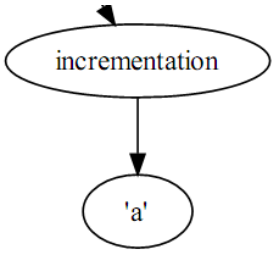
Une condition est composée de deux nœuds qui peuvent être des « tokenNode » ou « valeurNode ». Le type de la condition est sont opérateurs.

DeclarationNode



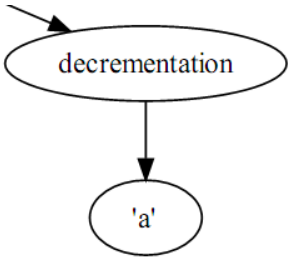
Une déclaration est suivit d'une assignation avec comme type, le type de variable.

IncrementationNode



Une fonction incrémentation sur la variable (token) spécifiée.

DecrementationNode

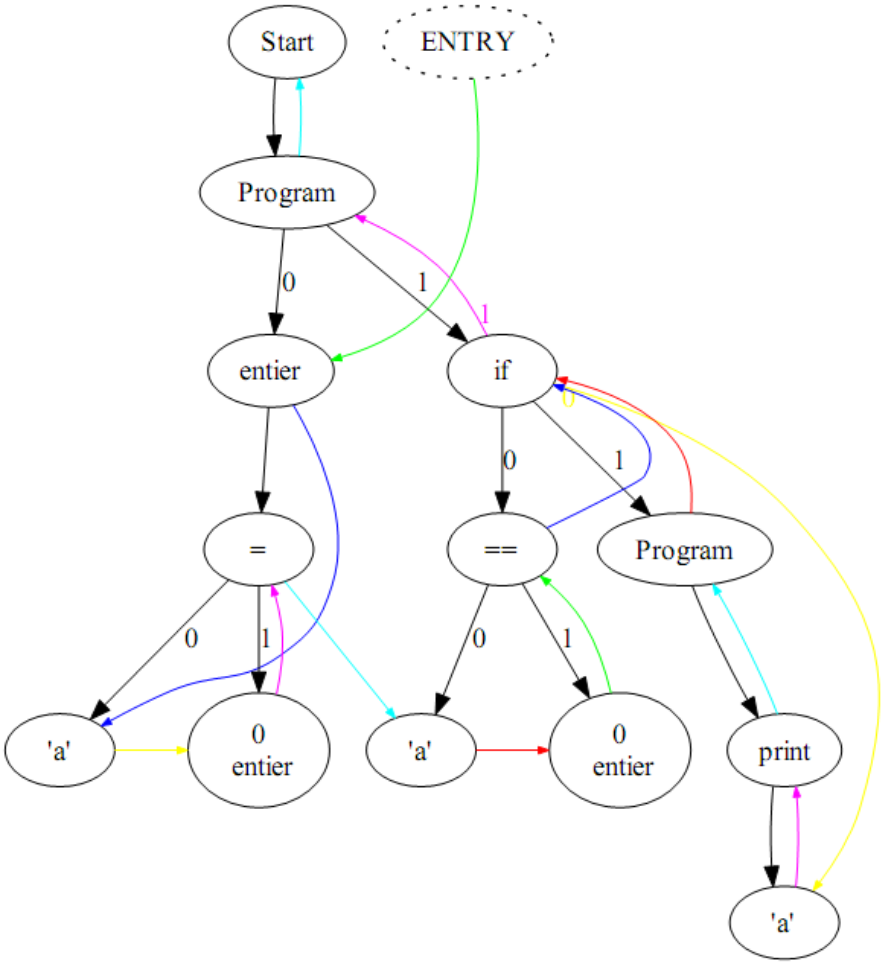


Une fonction décrémentation sur la variable (token) spécifiée

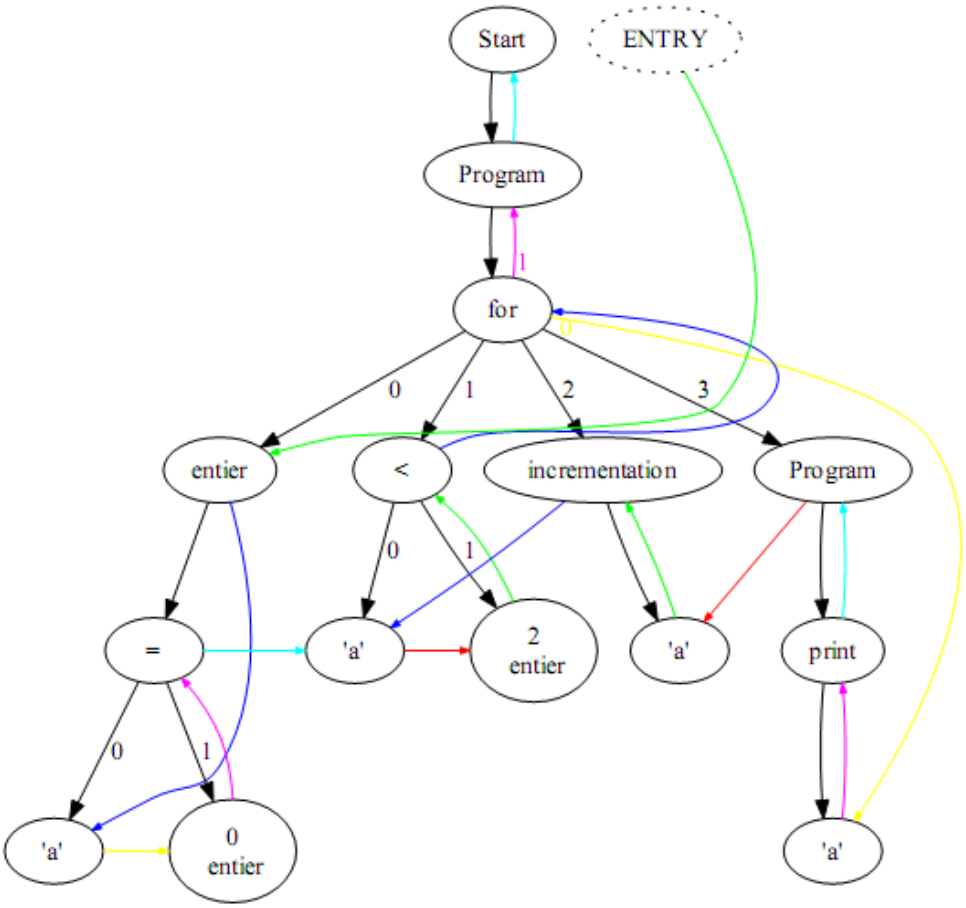
Analyse sémantique

Nous avons décidé de réaliser un arbre cousu pour poursuivre notre projet. Grâce à cet arbre, nous pouvons par la suite traiter les différents types de données lors des opérations mathématiques.

Arbre cousu



Voici un exemple de couture que nous obtenons lors de l'exécution de la couture. On peut voir que lorsqu'on arrive sur le nœud `if` il y a deux départs possible, le premier suis la prochaine instruction du sous programme (fils 1) et un autre la prochaine opération de l'instruction `if`.



Voici un deuxième exemple de couture. Cette fois nous avons la boucle `for`. Si on regarde plus attentivement, on voit que depuis le nœud `entry` on arrive directement sur un nœud de déclaration. Puis une fois la déclaration fini, on arrive sur un nœud de type condition. Une fois la condition passée nous allons sur le nœud `for` qui lui, par la suite, va se charger de l'exécution de la condition. Si le test est vrai, on exécute le programme. Une fois celui-ci terminé, nous allons sur l'incrémentation. De l'incrémentation, nous allons sur la condition.

Pour généré l'arbre cousu, nous avons développé le programme « threader.py ».

Typage des données

Pour le typage de variables, nous avons surchargé chaque opération disponible dans notre langage. Nous testons le type de chaque valeur puis nous agissons en fonction. Par exemple, nous avons rendu possible l'opération modulo entre un nombre entier un nombre réel à l'aide du casting. Pour plus de détail, la section « Variable » de ce rapport décrit le fonctionnement du typage.

Interpréteur

Comme nous avons réalisé un arbre cousu, il a fallut créer un interpréteur itératif. Notre choix était partagé entre l'utilisation d'une machine virtuelle ou un interpréteur. C'est finalement la solution de l'interpréteur que nous avons retenu pour sa simplicité.

Pour faire cela, nous nous sommes basés sur les TP. Nous interceptions tous les nœuds dont nous avons à modifier le cours d'exécution.

Déclaration

Ajoute la valeur dans un tableau pour la sauvegarde.

Print

Afficher sur la console le contenu du dernier élément de la pile.

Opération

Exécute l'opération demandée et la stocke sur la pile. Les deux opérandes sont sur la pile

Assignation

Affecte à la variable le résultat obtenu de la dernière opération. Le résultat ainsi que le nom de la variable sont sur la pile pour l'exécution.

If

Evalue la condition puis agit en fonction. Lors de l'exécution de ce nœud on stocke un nombre de passage dans le nœud car l'arbre cousu arrive deux fois sur lui. Le traitement n'est pas le même à chaque passage le premier test de la condition et exécute en fonction du résultat. Au deuxième passage on remet le compteur à 0 (si l'instruction ce trouve dans une boucle) puis on va sur le nœud suivant dans l'arbre. Les paramètres de la condition sont stockés sur la pile.

IfElse

Evalue la condition puis agit en fonction. La gestion des passages ce fais de la même manière que pour le nœud If.

Boucle (while, do..while,for)

Evalue la condition puis agit en fonction.

Le fichier « interpreteur.py » permet l'exécution du programme dans sont intégralité.

Conclusion

Ce projet à été très instructif sur plusieurs points. Premièrement, nous avons pu mettre en pratique nos connaissances acquises durant ce semestre en « langage et compilateur ». Nous avons eu quelques difficultés supplémentaires par rapport au TP comme la gestion de variables typées pour n'en citer qu'un.

Concernant notre langage de programmation, le compilateur permet d'exécuter notre code sans problème et traite les erreurs correctement. Malgré cela, il est évident que notre langage n'est pas parfait et qu'il comporte bien des lacunes pour pouvoir être utilisé comme un vrai langage de programmation

Nous avons également pris un grand plaisir à la réalisation de ce projet.