Natural to Python Compiler

Nino Jeannet et Sébastien Peiris

Présenté à Fabrizio Albertetti

12 Janvier 2020

haute école
neuchâtel berne jura
arc ingénierie
www.he-arc.ch

Contents

1 Introduction

1.1 Présentation

Ce projet a été réalisé dans le cadre du cours de Compilateur. Il a pour but la création d'un langage de programmation ainsi que sa compilation vers un langage de haut niveau. Ce compilateur est intégralement réalisé en *Python* et les analyseurs lexical et syntaxique sont basés sur *PLY*.

1.2 Langage Natural

Natural est donc le langage que nous avons créé. Sa principale caractéristique est son très haut niveau. En effet, celui-ci est proche du langage français parlé. Son but serait d'être compréhensible pour des personnes n'ayant pas ou peu de bases en programmation. Étant donné la complexité de la langue française, nous avons été forcés de limiter l'étendue du langage. En effet, implémenter un compilateur qui comprend le langage "parlé" et qui puisse le traduire instantanément en *bytecode* Python relève de l'impossible. Il nous a donc fallu déterminer des règles et des limites à *Natural*.

1.3 Natural to Python Compiler

Notre compilateur a donc pour but de compiler du code *Natural* en python. *Natural* est capable de:

- Déclarer et instancier des variables
- Effectuer des opérations entre variables (*/-+)
- Effectuer des boucles *for*
- Effectuer des test *if / else*
- Afficher du contenu

2 Analyse lexicale

L'analyseur lexical a pour but de définir les différents lexèmes du langage *Natural* ainsi que d'autres modalités telles que les mots réservés. Ceux-ci ne pourront pas servir de nom de variable. L'analyse lexicale s'effectue dans le fichier *lex.py*.

2.1 Mots réservés

Les mots réservés du langage sont les suivants:

```
egal, a, vaut, plus, moins, fois, divise, par, pour, allant, de, pas, afficher,
si, inferieur, superieur, sinon, debut, fin
```

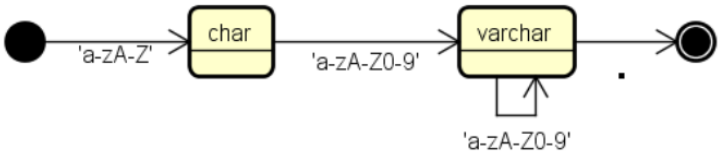
2.2 Lexèmes

Nous avons définit différents lexèmes (tokens) qui composeront notre langage. Ces lexèmes sont définis à l'aide d'expressions régulières. Nous avons défini les lexèmes suivants :

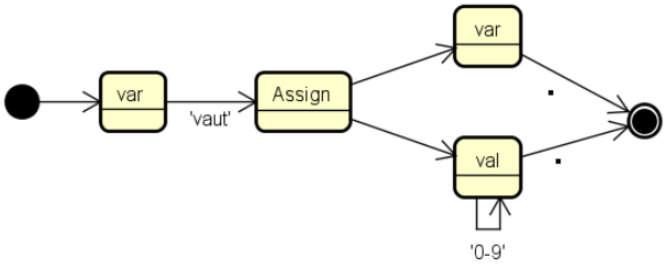
- NUMBER
- ADD_OP
- MUL_OP
- COMPARABLE
- TO
- STEP
- IDENTIFIER
- STRING
- + la liste des mots réservés

2.3 Automates d'états finis

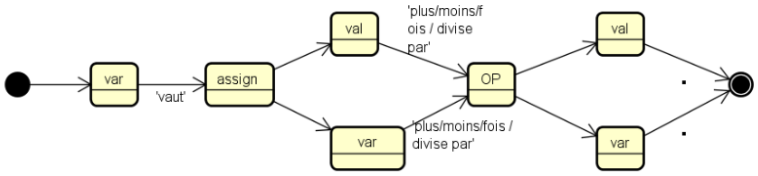
2.3.1 Variable



2.3.2 Assignation



2.3.3 Assignation et calcul



3 Analyse syntaxique

Lors de la création de notre langage, nous avons dû définir une syntaxe à respecter pour que l'on arrive à le compiler. Pour ceci, nous avons défini une syntaxe à respecter pour chaque type d'éléments.

3.1 Définition de variable

Un nom de variable ne peut contenir que des chiffres et des lettres et au minimum un caractère. Il doit également débuter par une lettre (majuscule ou minuscule). L'expression régulière correspondant est la suivante: $[A-Za-z][A-Za-z0-9]+$

3.2 Instanciation de variable

Pour déclarer et instancier une variable il faut utiliser la syntaxe :

```
<nom_de_variable> vaut <valeur_de_variable>.
```

On peut instancier une variable, soit avec une valeur numérique, soit avec une expression qui vaut une valeur numérique.

3.3 Boucle FOR

La boucle *FOR* s'écrit de la manière suivante :

```
1 pour <var> allant de <depart> a <arrivee> par pas de <pas>.
2 debut
3     # do something
4 fin.
```

Cette syntaxe va faire une boucle en initialisant la variable d'incrémentation avec la valeur de départ, en l'incrémentant du pas à chaque itération tant qu'il est plus petit que la valeur d'arrivée. **Important**: il n'est pas nécessaire d'utiliser une tabulation avant le contenu de la boucle mais il est fortement recommandé de l'utiliser pour des raisons de lisibilité.

3.4 Bloc IF...ELSE

Pour utiliser un bloc de condition(if..else), il faut utiliser la syntaxe suivante :

```
si <condition>.
debut
    # do something
sinon
    #do something else
fin.
```

3.5 Affichage

Pour afficher une valeur, il faut utiliser la syntaxe ci-dessous:
Si on affiche une chaîne de caractère :

```
afficher '<chaine>'
```

Si on affiche une valeur :

```
afficher <valeur>
```

3.6 Problèmes de syntaxe

Durant la définition des ces différentes syntaxes, nous avons détecté plusieurs cas qui pourraient poser problème dans la suite du projet. Voici quelques exemples de cas où nous avons adaptés notre syntaxe pour éviter toute ambiguïté.

3.6.1 Boucle FOR

Nous avions, au départ, décidé d'utiliser le symbole "-" pour définir les lignes contenues dans la boucle sans avoir besoin de délimiteur. Mais après réflexion, cette méthode ne fonctionne pas dans le cas de boucles imbriquées car nous aurions des enchaînements de "-" et toute lisibilité serait perdue. Nous avons donc choisi d'utiliser des délimiteurs (*debut* et fin).

3.6.2 Condition

Idem que pour la boucle for, nous utilisons des délimiteurs plutôt que des "-".

3.6.3 "=" et "=="

Afin de différencier le = du == nous avons simplement décidé d'utiliser des lexème unique pour les différencier. Donc nous avons *egal a* qui correspond à == et *vaut* qui correspond à =.

3.7 Nombres

3.7.1 Nombres décimaux

Afin de d'éviter la confusion avec les fins de lignes, nous avons décidé d'utiliser la virgule dans les nombres décimaux. Exemple :

```
x vaut 13,5
```

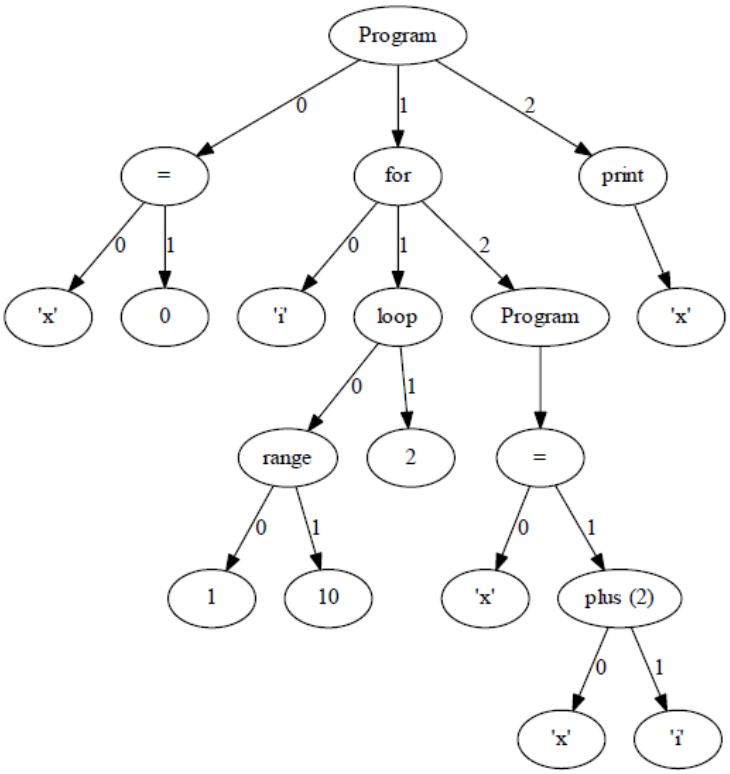
3.8 Arbres syntaxiques

3.8.1 Boucle FOR

Code *Natural*:

```
x vaut 0.
pour i allant de 1 a 10 par pas de 2.
debut
    x vaut x plus i.
fin.
afficher x.
```

Arbre syntaxique:

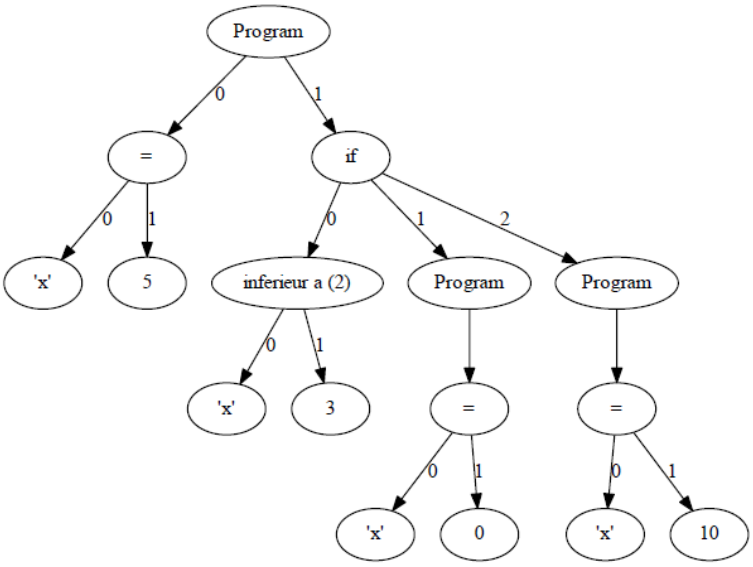


3.8.2 Condition

Code *Natural*:

```
x vaut 5.
si x inferieur a 3.
debut
    x vaut 0.
sinon
    x vaut 10.
fin.
```

Arbre syntaxique:



4 Grammaire

Après avoir définit les différentes syntaxes, on peut maintenant tout regrouper pour définir notre grammaire. Cette grammaire sera notamment utile lors de l'analyse syntaxique afin de construire un arbre syntaxique à partir d'un code source.

```
PROGRAMME -> STATEMENT '.'
PROGRAMME -> STATEMENT '.' PROGRAMME

STATEMENT -> ASSIGNATION | STRUCTURE
STATEMENT -> afficher EXPRESSION | afficher EXPRESSIONSTRING

STRUCTURE -> pour ITERATEUR to LOOP '.' debut PROGRAMME fin
ITERATEUR -> IDENTIIFIER
LOOP -> RANGE STEP EXPRESSION
RANGE -> EXPRESSION a EXPRESSION

STRUCTURE -> si BOOLEAN '.' debut PROGRAMME sinon PROGRAMME fin
BOOLEAN -> EXPRESSION comparable EXPRESSION

CONDITION -> EXPRESSION inferieur a EXPRESSION
CONDITION -> EXPRESSION superieur a EXPRESSION
CONDITION -> EXPRESSION egal a EXPRESSION

EXPRESSIONSTRING -> STRING
EXPRESSION -> NUMBER | IDENTIFIER
EXPRESSION -> EXPRESSION ADD_OP EXPRESSION
EXPRESSION -> EXPRESSION MUL_OP EXPRESSION
EXPRESSION -> EXPRESSION SUB_OP EXPRESSION
EXPRESSION -> EXPRESSION DIV_OP EXPRESSION

ASSIGNATION -> IDENTIFIER vaut EXPRESSION
```

4.1 Exemples de code

4.1.1 Instanciation de variables

Python :

```python
x = 0
y = x + 12

for i in range(0, 10, 1):
    x = 2 * y
    y = y - 1
print(x)
print(y)
```

Natural :

```
x vaut 0.
y vaut x plus 12.
pour i allant de 0 a 9 par pas de 1.
debut
    x vaut 2 fois y.
    y vaut y moins 1.
fin.
afficher x.
afficher y.
```

4.1.2 Bloc de condition

Python :

```
if x < y:
    print('success')
else:
    print('error')
```

Natural:

```
si x inferieur a y.
debut
    afficher 'success'.
sinon
    afficher 'error'.
fin.
```

4.1.3 Exemple complexe

Python :

```
x = 1
y = 2 + 12

for i in range(0, 10, 1):
    if x < y:
        print(x-y)
    else:
        for i in range(1,3,2):
            x = x + 1
    print(x)
```

Natural:

```
x vaut 1.
y vaut 2 plus 12.
pour i allant de 0 a 10 par pas de 1.
debut
    si x inferieur a y.
    debut
        afficher x moins y.
    sinon
        pour i allant de 1 a 3 par pas de 2.
        debut
            x vaut x plus 1.
        fin.
    fin.
    afficher x.
fin.
```

5 Analyse sémantique

Une partie d'analyse sémantique est nécessaire afin d'effectuer certaines vérifications et empêcher un certain nombre d'erreurs en afficher des messages d'avertissement. Étant donné que le projet se limite à la réalisation d'un compilateur, il n'est pas nécessaire d'arrêter l'exécution si une erreur apparaît. Nous avons donc choisi d'afficher des *warnings* mais de tout de même effectuer l'opération de compilation jusqu'au bout.

5.1 Couture

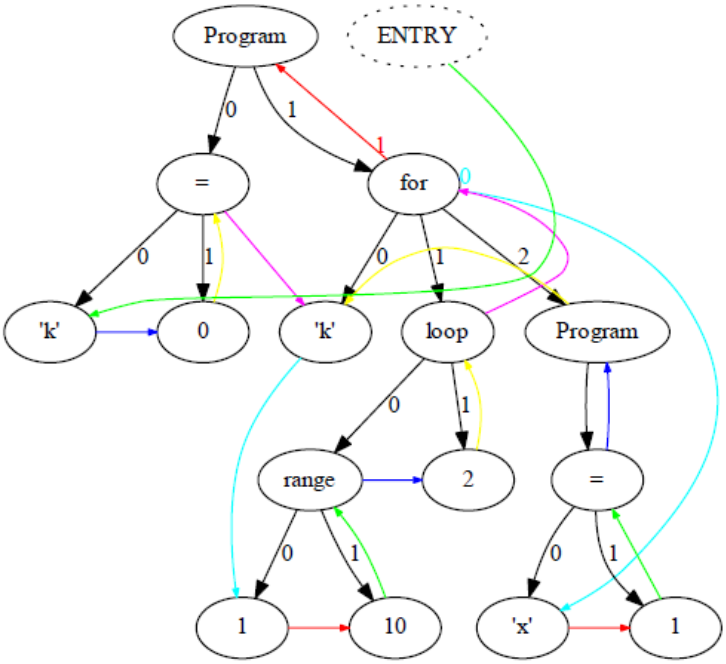
Afin de détecter les potentielles erreurs, il est nécessaire d'implémenter la couture de l'arbre syntaxique. Cette couture parcours l'arbre syntaxique créé par le parseur et traduit le code noeud par noeud.

5.1.1 Exemples de coutures

Code *Natural* :

```
k vaut 0.
pour k allant de 1 a 10 par pas de 2.
debut
    x vaut 1.
fin.
```

Couture de l'arbre syntaxique :



5.2 Gestion d'erreurs

Nous avons décidé de gérer 3 types d'erreurs pour lesquels nous avons généré des *warnings* afin d'en informer l'utilisateur :

- L'utilisation de variable non instanciée : Si une variable est utilisée sans être instanciée, on soulève un w*arning*. Pour vérifier ceci, nous sauvegardons chaque variable dans un dictionnaire au moment de son instanciation. A chaque token, nous vérifions que la variable existe déjà.
- L'utilisation d'une variable en dehors de son *scope* (pour les boucles for) : Cette fonctionnalité fonctionne avec plusieurs boucles for dans le programme et une imbrication maximum de 2. A chaque fois que nous instancions une variable, la "position" à laquelle elle à été instanciée est sauvegardée. Nous sauvegardons entre autre, l'*id* de la boucle et la profondeur dans laquelle à eu lieu l'instanciation.
- La division par zéro : Pour vérifier la division par zéro, nous vérifions simplement dans un OpNode, que l'opération soit une division et que le second enfant ne soit pas égal à zéro.

6 Compilateur

La partie du compilateur a pour but d'effectuer la traduction finale du code source en *Natural* vers le code de sortie en *Python*. Pour ce faire, le module *compiler.py* parcourt chaque noeud et compile ses enfants afin de traduire le code en langage *Python*.

7 Problèmes rencontrés

Vérification du *scope* pour la condition if :

Dans l'analyse sémantique, nous avons voulu vérifier qu'une variable instanciée dans un block if soulève un warning si elle est utilisé en dehors de son scope (dans le else ou block parent). L'implémentation fonctionnait bien avec une simple condition IF mais demandait trop de tests avec des conditions imbriquées et plusieurs conditions au même niveaux dans le programme. Il fallait gérer la profondeur, l'*id* et la position(if ou $else$) dans le bloc, de la condition. Nous avons donc préféré ne pas laisser cette fonctionnalité incomplètement implémentée dans le programme.

8 Conclusion

Le but final de ce projet était, dans un premier temps, de parvenir à créer un langage de programmation en définissant sa syntaxe et ses règles de grammaire afin de, dans un deuxième temps, le compiler vers un langage de haut niveau existant (ici en *Python*). La démarche suivie a été celle vue en cours, notamment à l'aide des TPs. Nous avons donc suivi les étapes d'analyse lexicale, syntaxique et sémantique avant d'implémenter le compilateur.

Dans la partie dédiée à l'**analyse lexicale**, nous avons défini les différents lexèmes du langage ainsi que les mots réservés. Nous avons également représenté les automates d'états finis pour quelques cas tels que la l'assignation de variables, par exemple.

Lors de l'**analyse syntaxique**, nous avons du définir la syntaxe que chaque élément du code (nombres, blocs FOR/ IF, délimiteurs, etc...). Il a également fallu implémenter la construction des arbres syntaxiques, éléments indispensable aux étapes suivantes.

L'**analyse sémantique** consiste en l'implémentation de la couture de l'arbre syntaxique réalisé par l'analyseur syntaxique. Cette couture est essentielle afin de pouvoir parcourir le code et définir s'il contient des incohérences, telles que des variables utilisées mais non déclarées ou encore des divisions pas *zero*.

Au vu des résultats obtenus, nous pouvons constater que l'objectif de la réalisation du compilateur *Natural* vers *Python* a été atteint. Nous obtenons en sortie un fichier script phyton parfaitement exécutable en ligne de commande et correct syntaxiquement parlant. Ce projet nous a permis de nous rendre compte de la manière dont est implémenté un compilateur, outil que nous utilisions depuis le début de notre formation sans pour autant en comprendre le fonctionnement. Nous avons également pu nous familiariser avec les notions d'analyse lexicale, syntaxique et sémantique, et plus généralement, avec l'analyse de textes en informatique.