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

Bachelor of Science HE-ARC in Engineeringgg

Orientation : Ingénierie des données

Cours : Compilateur

Py2C

Réalisé par :
El Boudiri Anasse
Steiner Jan
Marques Reis Henrique
Enseignant :
Tièche François
Assistant :
Goffinet Edouard

29 janvier 2023

Table des matières

Chapitre 1

Introduction

Dans le cadre du cours de compilateur, ce projet à pour but de créer un transpilateur de code python en code **C++**. Pour ce faire ce programme utilise une implémentation de lex et yacc en python (package **ply** [1]). Dans la première partie de ce rapport se trouve l'analyse de la problématique ainsi que les différentes fonctionnalités de ce programme. Puis, dans une deuxième partie, la réalisation des différentes fonctionnalités implémentée. Et finalement les résultats obtenus ainsi que la conclusion de ce projet.

Chapitre 2

Analyse

Le but de ce programme est de transformer du code **python** en **C++**. le programme doit être capable de détecter et transformer en **C++** :

1. Les variables
2. La portée des variables
3. Les opérations de base (+, -, *, /, <, >, %, <=, =>, ==, !=)
4. Les structures (while, if, elif, else)
5. Les fonctions

Python étant un langage indenté et non typé, le programme doit également être capable de gérer le typage des variables et remplacer les indentations par des **{}**.

Chapitre 3

Réalisation

Lors de la réalisation du programme, nous avons séparé le programme en 3 parties :
- Analyse lexicale
- Analyse syntaxique (cf. Grammaire)
- transpilateur

3.1 Gestion des types

Les types gérés par notre programme sont :

1. Float
2. Int
3. String
4. Bool

Analyse lexicale

Le type de variable est détecté avec des regex pour chaque type énuméré ci-dessus.

Analyse syntaxique

Lors de l'analyse syntaxique, seul le typage des fonctions est traité. (voir section Gestion des fonctions).

Transpiler

Le transpiler permet de faire des opérations entre les variables de même type. Il ne permet pas de faire des conversions implicites de type. par exemple $Float + Int$

3.2 Gestion des opérateurs de base

Les opérateurs gérés par notre programme sont :

1. +
2. −
3. /
4. <
5. >
6. %
7. <=
8. =>
9. ==
10. ! =

Analyse lexicale

Lors de l'analyse lexicale, les opérateurs font partie des caractères réservés de notre programme et sont directement à détecter.

Analyse syntaxique

Lors de l'analyse syntaxique, les opérateurs détectés dans lexer sont transformés en nœud selon quelles opérations est détecté.

Transpiler

Lors de la transpilation, les variables déclarer avec des opérateurs booléens sont détectés est créé avec le typage *bool*. La gestion des types dans les opérations sont gérées comme indiqué dans la section Gestion des types.

3.3 Gestion portée des variables

La portée des variables est similaire à celle du langage python. Le programme détecte les variables globales et les variables locales, cependant si une variable est définie en dehors de tout bloc (fonction,boucles,conditions), lors de l'appel de la variable dans un bloc elle sera recréée si dans le programme python le mot-clé global n'est pas précisé.

3.4 Gestion des structures (boucles,conditions)

Analyse lexicale

Lors de l'analyse lexicale, les structures font partie des mots réservés de notre programme et sont directement détectées.
Liste des mots réservés :

1. while
2. if
3. elif
4. else

Analyse syntaxique

Sur la figure 3.1 un exemple de l'arbre syntaxique des structures gérées par notre programme

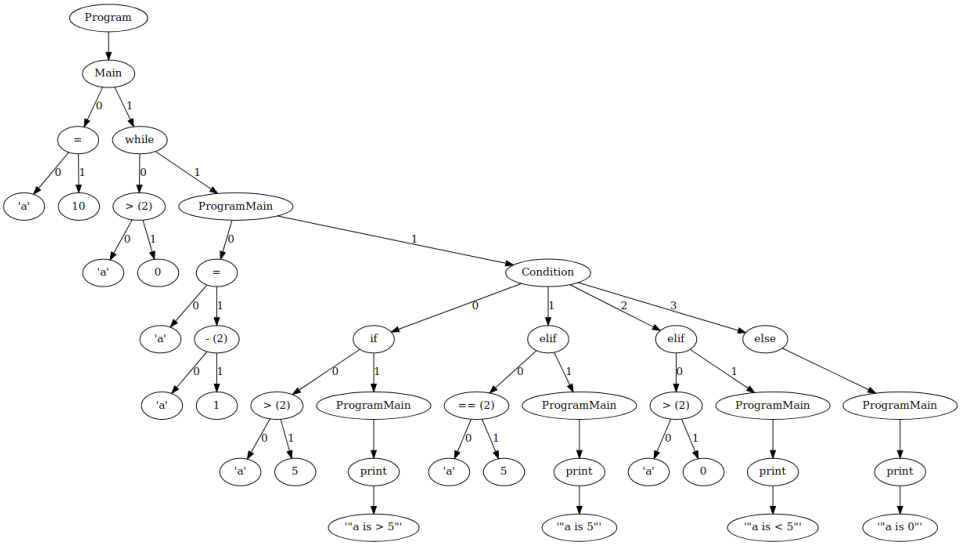

FIGURE 3.1 – Exemple AST structures

3.5 Gestion des fonctions

en **C++** le programme principal ce trouve dans une fonction main et à l'intérieur de celle-ci on ne peut pas créer d'autres fonctions. Pour recréer ce comportement nous avons décidé de brider notre code python en l'empêchant de créer des fonctions à l'intérieur d'un bloc `if __name__ == "__main__"` qui nous sert de fonction main.

Il est également important de noter que les fonctions gérées par notre programme doivent être typées comme sur l'exemple ci-dessous :

```
def fo(x:int) -> int: # fonction prenant un entier et retournant un entier
def fo2() -> None: # fonction de type void
```

Analyse lexicale

Lors de l'analyse lexicale, les fonctions sont détectées avec le mot-clé **def** alors que la fonction **main** est détectée avec un regex.

Analyse syntaxique

Lors de l'analyse syntaxique, le programme distingue 2 types de déclaration de fonctions, la première a des arguments et la deuxième non. Dans les 2 cas, le nœud d'une fonction va contenir le type de retour et le programme dans la fonction. Mais si la fonction possède des arguments, ils doivent alors également être traités et ajouter au nœud.
Pour l'appel de fonction on retrouve nos 2 types comme pour la déclaration, un avec paramètre et un sans. Comme précédemment dans le cas où l'appel contient des paramètres, un nœud de paramètre doit être ajouté à la fonction.

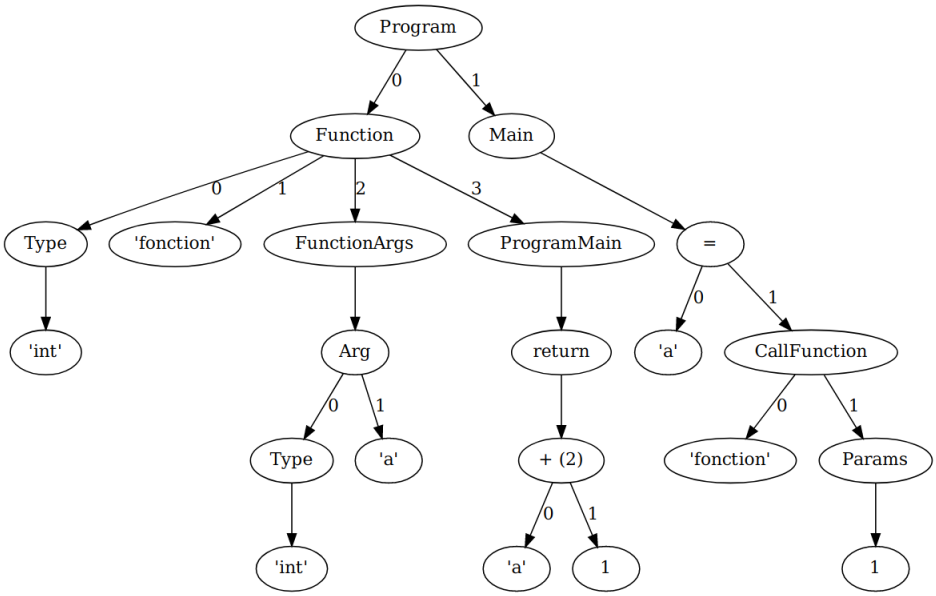

FIGURE 3.2 – Exemple AST fonction

3.6 Gestion des indentations

La gestion des indentations dans notre programme permet de détecter les indentations et dédentations en python et les remplacer par des **{}** en **C++** ainsi que gérer la portée.

Analyse lexicale

Lors de l'analyse lexicale, les indentations sont détectées par un regex puis ajoutées dans un stack avec sa valeur d'indentation. Cette valeur d'indentation visa à identifier le niveau de profondeur de l'indentation.
Après avoir détecté l'indentation et sa valeur, le token doit être modifié pour devenir une nouvelle ligne si le niveau de profondeur est le même. Et dans le cas où le niveau de profondeur est plus bas que l'indentation précédente, le token devient une dédentation.

Sur le code ci-dessous, un exemple de l'évolution de la valeur d'indentation :

```
def fo(x):  # INDENT value = 0
    if x > 5:    # INDENT value = 1
        print("bigger than 5")  # INDENT value = 2
        print("it's great!")    # NEWLINE
fo(6)    # DEDENT value = 2
```

Analyse syntaxique

À la fin de l'analyse lexicale, les tokens d'indentations et dédentations contiennent leurs valeurs respectives. Cependant pour grandement simplifier l'analyse syntaxique avant de passer les tokens dans le parseur, les tokens de dédentations sont multipliés par leur valeur. Par exemple pour un token «DEDENT(2)» on renvoie 2 fois le token «DEDENT». Cela permet d'avoir des blocs bien définis commençant par «INDENT» et finissant par «DEDENT»

3.7 Gestion des erreurs

Lors de la transpilation nous détectons différents types d'erreurs sur le code.

- Contrôle de type sur le passage de paramètre dans une fonction
- Contrôle de type sur le type de retour d'une fonction
- Contrôle d'existence des variables globales
- Division par zéro

Chapitre 4

Tests et résultats

Ce programme a été testé avec plusieurs scripts python se trouvant dans le dossier **src/test**. Sur les figures 4.1 et 4.2 vous trouverez un exemple de code python transformé en code C++. Ainsi que l'arbre syntaxique généré par le parseur à la figure 4.3.

```python
def max(a: int, b: int) -> int:
    if a > b:
        return a
    else:
        return b


if __name__ == '__main__':
    if max(1, 2) == 2:
        print('ok')
    elif 2 > 3:
        print('wtf ?')
    elif 3 > 4:
        print('...')
    else:
        print('nope')
```

FIGURE 4.1 – Exemple code python

```cpp
#include <iostream>
#include <string>

using namespace std;

int max(int a, int b)
{
    if (a > b)
    {
        return a;
    }
    else
    {
        return b;
    }
}
.
.
.
.
.
.
.
int main()
{
    if (max(1, 2) == 2)
    {
        cout << "ok" << endl;
    }
    else if (2 > 3)
    {
        cout << "wtf ?" << endl;
    }
    else if (3 > 4)
    {
        cout << "..." << endl;
    }
    else
    {
        cout << "nope" << endl;
    }
    return 0;
}
```

FIGURE 4.2 – Exemple code C++

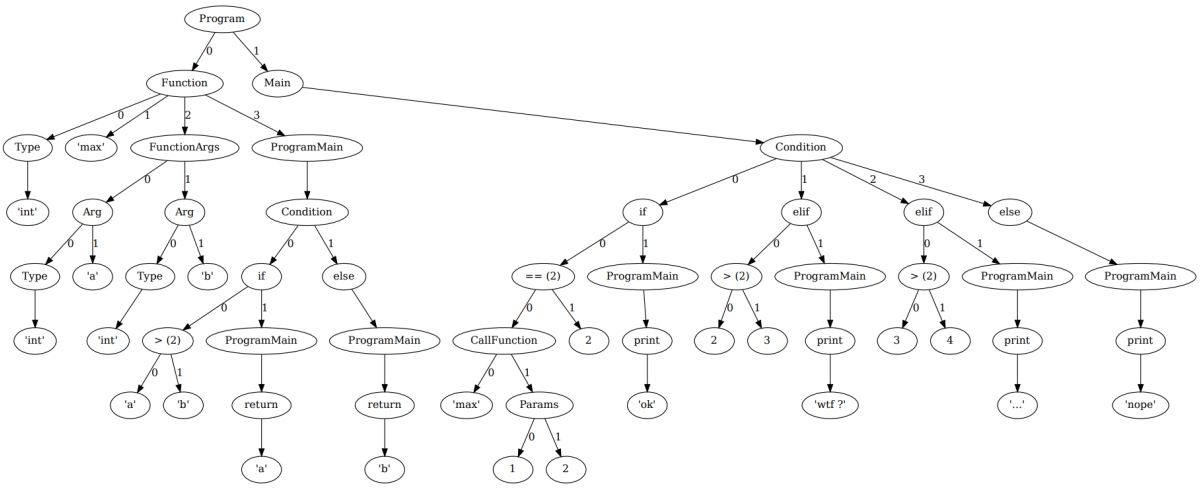

FIGURE 4.3 – Exemple AST

Chapitre 5

Conclusion

Sur la base des résultats obtenus, notre programme est capable de transcrire du python en **C++** avec certaines restrictions. Notamment, le typage des fonctions et le code sont obligés d'avoir une fonction **main** pour que le transpilateur fonctionne. Le programme est également capable de produire des warnings ainsi que des messages d'erreurs.

Annexe

```
Rule 0     S' -> programs
Rule 1     programs -> program main
Rule 2     programs -> main
Rule 3     main -> MAIN INDENT program_main DEDENT
Rule 4     program -> statement
Rule 5     program -> statement NEWLINE
Rule 6     program -> program statement NEWLINE
Rule 7     program -> program statement
Rule 8     statement -> assignation
Rule 9     statement -> function
Rule 10    function -> DEF ID ( args ) ARROW TYPE : INDENT program_main DEDENT
Rule 11    function -> DEF ID ( ) ARROW TYPE : INDENT program_main DEDENT
Rule 12    args -> arg
Rule 13    args -> args , arg
Rule 14    arg -> ID : TYPE
Rule 15    program_main -> statement_main NEWLINE
Rule 16    program_main -> statement_main
Rule 17    program_main -> program_main statement_main NEWLINE
Rule 18    program_main -> program_main statement_main
Rule 19    statement_main -> assignation
Rule 20    statement_main -> global
Rule 21    statement_main -> structure_main
Rule 22    statement_main -> return_func
Rule 23    statement_main -> call_func
Rule 24    statement_main -> condition
Rule 25    global -> GLOBAL ID
Rule 26    statement_main -> PRINT expression
Rule 27    return_func -> RETURN expression
Rule 28    assignation -> ID = expression
Rule 29    structure_main -> WHILE expression : INDENT program_main DEDENT
Rule 30    condition -> if_stmt
Rule 31    condition -> if_stmt else_stmt
Rule 32    condition -> if_stmt elifs else_stmt
Rule 33    if_stmt -> IF expression : INDENT program_main DEDENT
Rule 34    else_stmt -> ELSE : INDENT program_main DEDENT
Rule 35    elifs -> elif_stmt
Rule 36    elifs -> elifs elif_stmt
```

```
Rule 37    elif_stmt -> ELIF expression : INDENT program_main DEDENT
Rule 38    expression -> call_func
Rule 39    call_func -> ID ( )
Rule 40    call_func -> ID ( list_expr )
Rule 41    list_expr -> expression
Rule 42    list_expr -> list_expr , expression
Rule 43    expression -> NUMBER
Rule 44    expression -> ID
Rule 45    expression -> TEXT
Rule 46    expression -> BOOL
Rule 47    expression -> ( expression )
Rule 48    expression -> expression + expression
Rule 49    expression -> expression - expression
Rule 50    expression -> expression * expression
Rule 51    expression -> expression / expression
Rule 52    expression -> expression % expression
Rule 53    expression -> + expression
Rule 54    expression -> - expression
Rule 55    expression -> expression AND expression
Rule 56    expression -> expression OR expression
Rule 57    expression -> expression EQ expression
Rule 58    expression -> expression NE expression
Rule 59    expression -> expression GE expression
Rule 60    expression -> expression LE expression
Rule 61    expression -> expression > expression
Rule 62    expression -> expression < expression
Rule 63    expression -> NOT expression
Rule 64    NUMBER -> INT
Rule 65    NUMBER -> FLOAT
Rule 66    TEXT -> STRING
Rule 67    TEXT -> CHAR
Rule 68    BOOL -> TRUE
Rule 69    BOOL -> FALSE
```

Table des figures

Bibliographie

[1] D. Beazley, “Pypi ply,” 2023-01-28. [Online]. Available : https://pypi.org/project/ply/