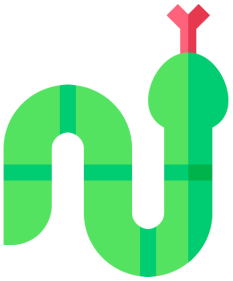

Vyper

Auteurs : David Darmanger

Owen Gombas

1. Table des matières

1.1. Abstract

Dans le cadre de notre formation et plus précisément du cours de Compilateur, nous avons réalisé ce projet de programmation. Il consistait à créer notre propre compilateur en définissant nous-mêmes ses fonctions.

2. Introduction

Notre projet visait à développer un compilateur pour traduire le code Python en code C++, en utilisant les concepts fondamentaux appris lors de notre cours sur les compilateurs. Nous avons examiné les différents éléments qui constituent un compilateur, tels que les analyseurs lexicaux, syntaxiques et sémantiques, ainsi que les techniques de génération et d'optimisation de code.

En mettant en pratique ces connaissances, nous avons pu développer un compilateur capable de convertir le code Python en code C++, offrant ainsi les avantages du code Python tout en bénéficiant des performances du langage C++.

Les livrables pour ce projet comprennent :
un code source du projet, un guide utilisateur, des tests fonctionnels pour vérifier le bon fonctionnement du projet et un rapport détaillé décrivant les étapes de développement et les résultats obtenus. Ce rapport inclut également une description des différentes étapes du processus de compilation.

3. Analyse

Notre projet visant à compiler du code Python vers du langage C++ a mis en place plusieurs fonctionnalités pour atteindre son objectif. Tout d'abord, nous avons implémenté les calculs arithmétiques de base, permettant ainsi d'évaluer des opérations mathématiques simples dans le code Python compilé.

Ensuite, nous avons implémenté les conditions (if, else) qui permettent de prendre des décisions dans le code en fonction de certaines conditions. Cela permet de gérer les flux de contrôle de manière efficace dans le code compilé.

Nous avons également implémenté les boucles while qui permettent de gérer les boucles et les itérations dans le code compilé.

Pour s'assurer que le code compilé respecte les règles sémantiques de Python, nous avons implémenté une vérification sémantique pour détecter les erreurs de syntaxe ou de sémantique dans le code source. Nous avons également implémenté un contrôle de type pour s'assurer que les variables utilisées dans le code sont utilisées de manière appropriée et que les types de données sont respectés.

Pour permettre l'utilisation de variables, nous avons implémenté l'utilisation et la définition de variables dans le code compilé.

Nous avons également implémenté l'indentation pour s'assurer que le code compilé est bien formaté et facile à lire. De plus, l’indentation est essentielle en python.

Enfin, pour permettre l'utilisation de fonctions dans le code compilé, nous avons implémenté l'utilisation et la définition de fonctions, y compris la fonction print pour afficher les résultats.

4. Réalisation

Règle	Définition
0	S' -> program
1	program -> statement_list
2	statement_list -> statement
3	statement_list -> statement_list statement
4	statement_list -> statement_list NEWLINE
5	statement -> assignment
6	statement -> if_statement
7	statement -> while_statement
8	statement -> return_statement
9	statement -> function
10	statement -> function_call
11	statement -> statement NEWLINE
12	block -> INDENT statement_list DEDENT
13	assignment -> ID ASSIGN expression
14	assignment -> ID COLON type ASSIGN expression
15	if_statement -> IF logical COLON block
16	if_statement -> IF logical COLON block ELSE COLON block
17	while_statement -> WHILE logical COLON block
18	while_statement -> WHILE TRUE COLON block
19	return_statement -> RETURN expression
20	id -> ID
21	boolean -> TRUE
22	boolean -> FALSE
23	integer -> INTEGER_VALUE
24	float -> FLOAT_VALUE
25	string -> STRING_VALUE
26	factor -> integer
27	factor -> float
28	factor -> boolean
29	factor -> string
30	factor -> id
31	factor -> function_call
32	unary -> NOT
33	logical -> factor LT factor

Règle	Définition
34	logical -> factor LE factor
35	logical -> factor GT factor
36	logical -> factor GE factor
37	logical -> factor EQUALS factor
38	logical -> factor NEQUALS factor
39	logical -> factor AND factor
40	logical -> factor OR factor
41	math -> factor PLUS factor
42	math -> factor MINUS factor
43	math -> factor TIMES factor
44	math -> factor DIVIDE factor
45	binary -> logical
46	binary -> math
47	expression -> factor
48	expression -> unary
49	expression -> binary
50	type -> INT
51	type -> FLOAT
52	type -> STR
53	type -> BOOL
54	type -> VOID
55	argument_list_definition -> ID COLON type
56	argument_list_definition -> ID COLON type COMMA argument_list_definition
57	function -> DEF ID LPAREN argument_list_definition RPAREN RETURN_TYPE type COLON block
58	function_call -> ID LPAREN RPAREN
59	function_call -> ID LPAREN argument_list_call RPAREN
60	argument_list_call -> expression
61	argument_list_call -> expression COMMA argument_list_call

4.1. Méthode de couture et vérification d'erreur

On a choisi d'utiliser la méthode récursive de génération de code dans la synthèse comme vu en cours, la vérification d'erreur se fait lors de la génération de code, si une erreur est détectée, le code ne sera pas généré et un message d'erreur sera affiché. Cette vérification utilise trois types de tables de symboles pour vérifier que les variables utilisées sont déclarées et que les types sont corrects:

- Une globale

```
1  sum: int = 0
2  sum = sum + 1
3  sum = sum + 2
4  sum = -1 * sum
5
6  b: bool = True
7  f: float = 1.2
8  s: str = "Hello World"
```

figure 1 - vérification erreur global

- Une dédiée aux fonctions et aux types de leurs arguments. Vu de l'extérieur, est-ce que je peux appeler cette fonction avec ces paramètres ?

```
13  def add(a: int, b: int) -> int:
```

```
17  add(1, 2)
```

figure 2 - vérification erreur exemple

- Une table de symboles dépendant d'un contexte (une fonction par exemple) qui contient les variables locales à cette fonction. Vu de l'intérieur, est-ce que je peux effectuer ces opérations sur ces paramètres ?

```
13  def add(a: int, b: int) -> int:
14      return a + b
```

figure 3 - vérification erreur exemple add

4.2. Génération du code

Concrètement le code C++ est écrit par le programme via la classe c_lang_generator.py qui contient les fonctions de génération de code pour chaque type d'expression, de déclaration, de fonction, de boucle, etc. Ces fonctions sont appelées par le programme lors de la génération de l'AST. Le code généré est stocké dans un fichier .cpp qui contient uniquement une fonction main. Les fonctions écrites en python sont converties en expression lambda C++ dans la fonction main. Le programme a donc la même structure et le même ordre que le fichier source Python. Cependant il est nécessaire d'utiliser le standard C++11 pour compiler le code généré du fait de l'utilisation de lambda.

Le code python doit utiliser le typage explicite pour que le programme puisse générer le code C++ correspondant. Le typage explicite est obligatoire pour les fonctions et leurs arguments ainsi que pour les variables cela permet de vérifier que les types sont corrects.

4.3. Conditions (if-else)

Il est important de s'assurer de la bonne gestion des blocs de code. Il est nécessaire de s'assurer que les blocs de code associés aux conditions if-else sont correctement identifiés et traduits en code cible. Si les blocs de code ne sont pas correctement gérés, cela peut entraîner des erreurs de syntaxe ou de sémantique dans le code cible. Il en est de même pour les imbrications. Car les erreurs d'imbrication peuvent entraîner des erreurs de sémantique et rendre le code cible peu lisible et difficile à maintenir.

Pour un if else on a un noeud if dans l'AST contenant 3 enfants

- Condition
- Un noeud Program pour true
- Un noeud Program pour false (optionnel)

```
elif node.type == "if":
    condition = self._determine_operation(node.children[0], nested_types)
    body = CLangGenerator(
        self._convert_ast_to_c(node.children[1], nested_types))

    else_body = None
    if len(node.children) == 3:
        else_body = CLangGenerator(
            self._convert_ast_to_c(node.children[2], nested_types))

    generator.condition(condition, body, else_body)
```

figure 4 - implémentation (transpiler.py) pour conditions

```python
sum: int = 0

if sum != 10:
    print("sum != 10")
else:
    print("sum == 10")
```

figure 5 - code python pour conditions

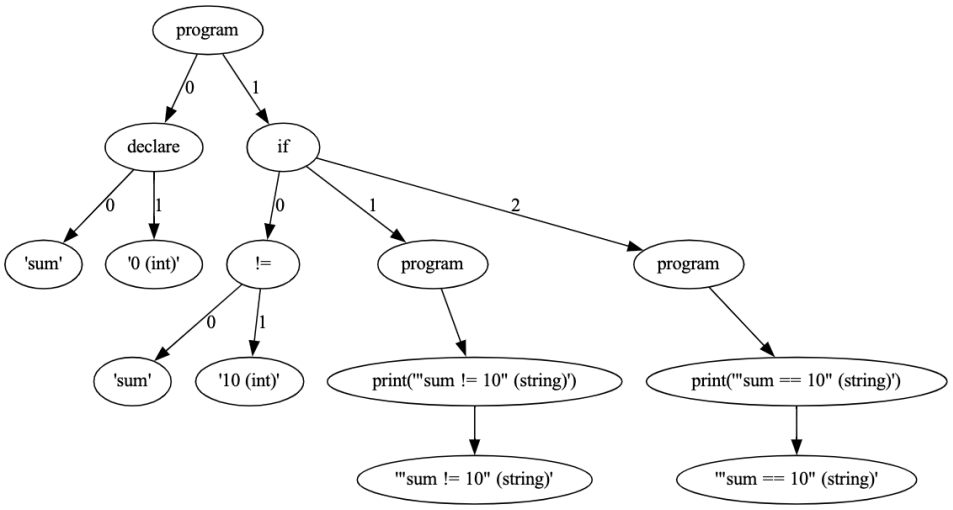

figure 6 - AST pour conditions

```cpp
auto sum = 0;
if (sum != 10) {
    cout << "sum != 10" << endl;
} else {
    cout << "sum == 10" << endl;
}
```

figure 7 - code produit C++ pour conditions

4.4. While

La complication principale est qu'il faut garantir que la condition de boucle soit correctement vérifiée à chaque itération.

Une boucle while a deux enfants dans l'AST, un contenant la condition et un autre contenant le sous-programme.

```
elif node.type == "while":
    condition = self._determine_operation(node.children[0], nested_types)
    body = CLangGenerator(
        self._convert_ast_to_c(node.children[1], nested_types))
    generator.while_loop(condition, body)
```

figure 8 - implémentation (transpiler.py) pour while

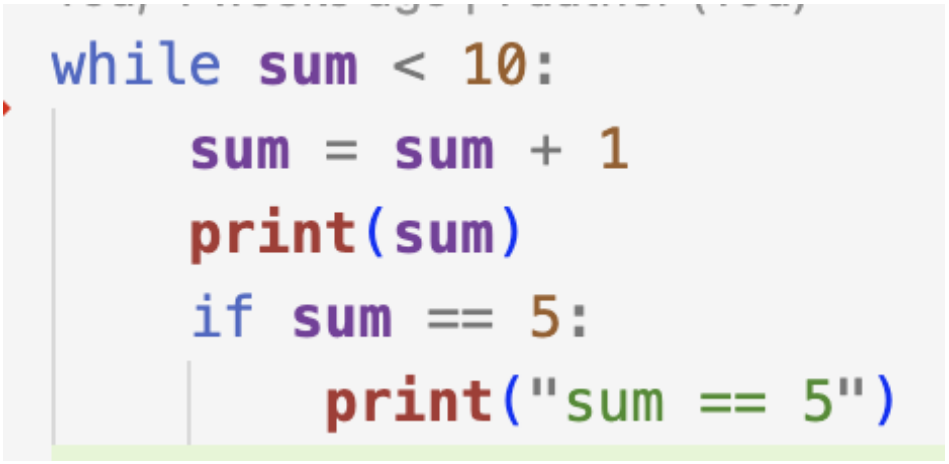

figure 9 - code Python pour while

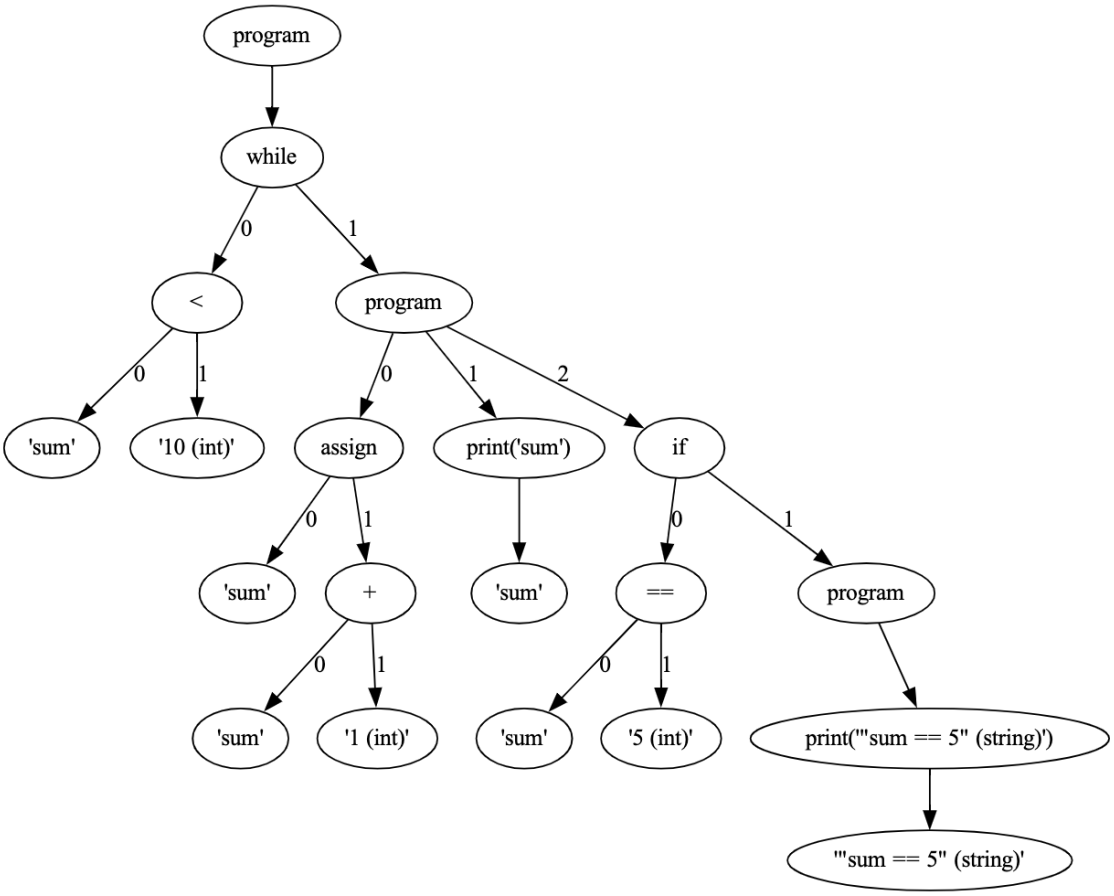

figure 10 - AST pour while

```cpp
while (sum < 10) {
    sum = sum + 1;
    cout << sum << endl;
    if (sum == 5) {
        cout << "sum == 5" << endl;
    }
}
```

figure 11 - code produit C++ pour while

4.5. Utilisation de variables

L'utilisation de variables est un aspect crucial pour garantir la flexibilité et l'extensibilité du code. Le problème principal lors de l'utilisation de variables est de s'assurer que les valeurs sont correctement associées à leurs noms respectifs et que ces associations sont maintenues tout au long de l'exécution du code

Une variable à deux types d'initialisation dans notre implémentation:

- **Déclaration**
  On attribue le type a la variable et il ne peut pas être changé par la suite afin de ne pas générer d'erreur lorsque le C++ est compilé, car C++ n'est pas aussi flexible que Python avec les types.
- **Assignation**
  Lorsqu'on change la valeur stockée de la variable qui a été déclarée auparavant

```
elif node.type == "assign":
    name = self._determine_operation(node.children[0], nested_types)
    value = self._determine_operation(node.children[1], nested_types)

    if node.children[1].value_type:
        if self._get_type(node.children[0], nested_types) != self._get_type(node.children[1], nested_types):
            raise Exception(f"Cannot assign different types for: {node.children[0]} and {node.children[1]}")

    if node.init_type:
        self.types[name] = node.children[0].value_type

    generator.assign(type="auto", declare=bool(
        node.init_type), name=name, value=value)
```

figure 12 - implémentation (transpiler.py) pour variables

```python
sum: int = 0
sum = sum + 1
sum = sum + 2
sum = -1 * sum

b: bool = True
f: float = 1.2
s: str = "Hello World"
```

figure 13 - code Python pour variables

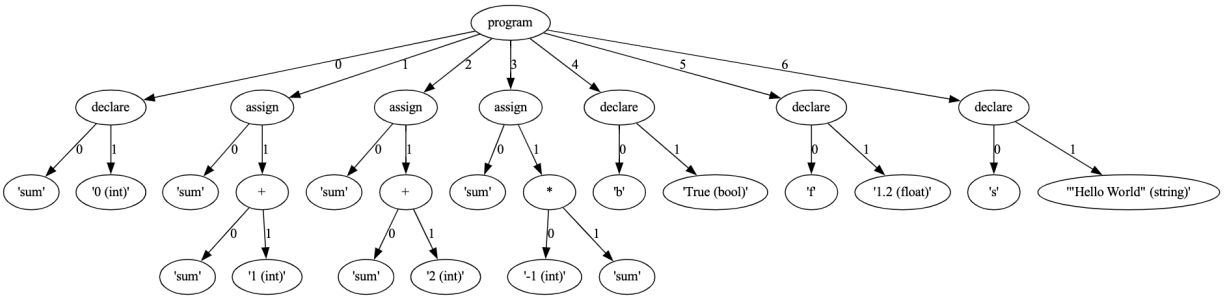

figure 14 - AST pour variables

```
auto sum = 0;
sum = sum + 1;
sum = sum + 2;
sum = -1 * sum;
auto b = true;
auto f = 1.2f;
auto s = "Hello World";
```

figure 15 - code produit (C++) pour variables

4.6. Implémentation de fonctions

Le problème principal dans l'implémentation de fonctions est de s'assurer que les fonctions sont correctement définies et utilisées. Les fonctions peuvent être complexes notamment à cause de la vérification des entrées sorties (et de leur nombre), il est donc important de les structurer correctement pour éviter les erreurs de programmation.

Il y'a deux types de noeud liés aux fonctions:

- **Déclaration**:
  On déclare la fonction et on définit un sous-programme qu'elle exécute
- **Appelle**
  On appelle la fonction avec les valeurs arguments

La fonction print est un noeud "call" spécifique qui est prédéfini par notre programme qui génère le code `cout << TEXT << endl;` dans tous les cas

```
elif node.type == "def":
    args = ", ".join([self._determine_operation(o, nested_types)
                      for o in node.args.children])

    self.functions[node.tok] = {node.args.children[i].tok: node.args.children[i].value_type for i in range(len(node.args.children))}

    nested_types = {}
    for arg in node.args.children:
        nested_types[arg.tok] = arg.value_type

    body = CLangGenerator(
        self._convert_ast_to_c(node.children[0], nested_types))
    generator.lambda_function(node.tok, args, body)
```

```
elif node.type == "return":
    generator.return_statement(
        self._determine_operation(node.children[0], nested_types))
```

```
elif node.type == "call":
    args = ", ".join([self._determine_operation(o, nested_types)
                      for o in node.children])

    if node.tok in self.functions:
        function = self.functions[node.tok]
        for i in range(len(node.children)):
            if self._get_type(node.children[i], nested_types) != self._get_type(function[list(function.keys())[i]]):
                raise Exception(f"Cannot call function with different types for: {node.children[i]} and {function[list(function.keys())[i]]}")

    if node.tok == "print":
        generator.stream("cout", args)
    else:
        generator.call_function(node.tok, args)
```

figure 16 - implémentation (transpiler.py) pour fonctions

```python
print("Hello World")

def add(a: int, b: int) -> int:
    return a + b

add(1, 2)
```

figure 17 - code Python pour fonctions

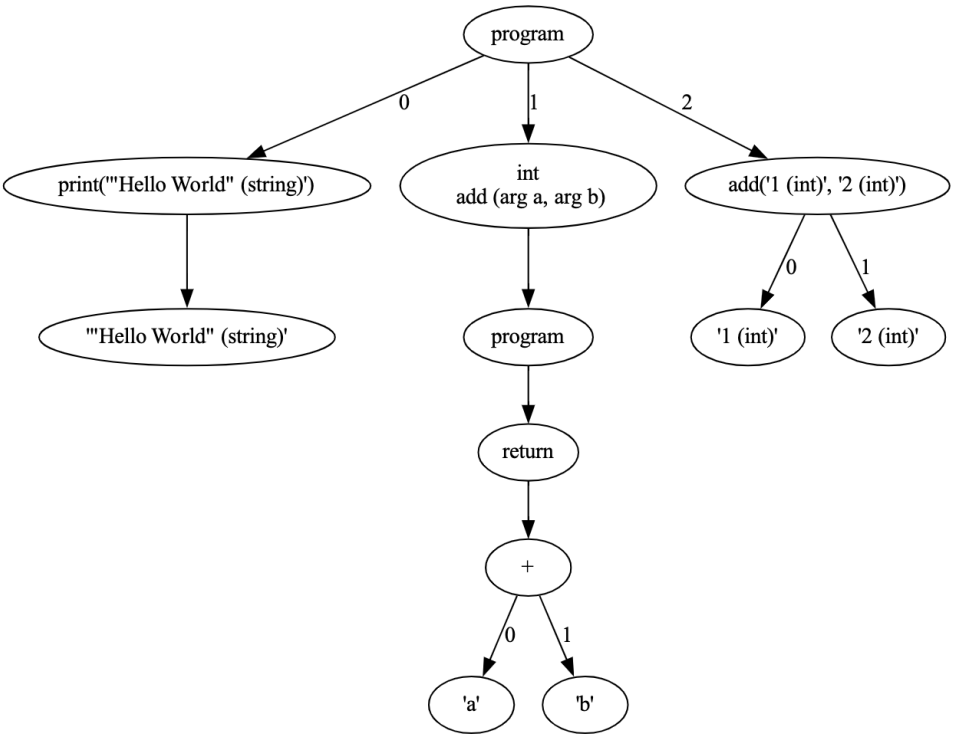

figure 17 - AST pour fonctions

```
cout << "Hello World" << endl;
auto add = [] (int a, int b) {
    return a + b;
};
add(1, 2);
```

figure 18 - code produit C++ pour fonctions

4.7. Problème d'indentation

Le passage de Python à C++ peut poser des problèmes en ce qui concerne l'indentation. En effet, Python utilise l'indentation pour déterminer la structure du code, ce qui n'est pas le cas en C++. Par conséquent, pour assurer la qualité du code et éviter les erreurs de syntaxe, il est important de bien gérer ce problème.

Puisque Python n'indique pas explicitement la fin d'un bloc de code (par endif ou } par exemple, l'indentation s'est faite via deux token (INDENT, DEDENT) détectés automatiquement (par calcul et différence d'espace) dans le lexer. Cela nous a permis de finir les règles facilement dans la partie yacc.

4.8. Vérification sémantique

Le problème principal avec la vérification sémantique est de s'assurer que le code source produit une sortie valide et cohérente en utilisant les bonnes variables, fonctions et autres éléments syntaxiques. Cela nécessite une analyse approfondie du code pour déterminer si les éléments sont utilisés correctement et si les relations entre eux sont logiques. Si la vérification sémantique échoue, il peut y avoir des erreurs de syntaxe, des erreurs de logique ou des problèmes de compatibilité entre les différentes parties du code.

4.9. Contrôle de type

La difficulté est qu'il faut déterminer les types corrects pour les variables, les fonctions et les arguments, et également tenir compte de toutes les interactions possibles entre ces éléments pour éviter des erreurs de type. Comme mentionné avant le typage explicite est obligatoire pour les fonctions et leurs arguments ainsi que pour les variables. Pour les fonctions et leurs arguments, le typage explicite est vérifié lors de la génération de code, si le typage explicite n'est pas respecté, le code ne sera pas généré et un message d'erreur sera affiché.

5. Améliorations possibles

Pour améliorer encore plus notre compilateur, plusieurs fonctionnalités peuvent être ajoutées pour répondre aux besoins de nos utilisateurs.

Premièrement, l'ajout de la boucle "for", en particulier avec la fonction "range", permettra aux utilisateurs de parcourir les ensembles de données de manière plus efficace.

Ensuite, l'ajout d'opérateurs unaires offrira une plus grande flexibilité lors des opérations mathématiques.

En outre, la prise en charge des tableaux et des structures de données améliorera la qualité et les performances de l'application.

De plus, l'implémentation de structures de contrôle plus avancées, telles que les "switch-case" et les opérateurs ternaires, simplifiera la logique de contrôle et la syntaxe pour les conditions simples.

Enfin, la prise en charge des classes et de l'héritage pour les objets orientés objet améliorera considérablement la flexibilité et la puissance de l'application.

En intégrant ces fonctionnalités supplémentaires, notre compilateur sera encore plus complet et répondra aux besoins des utilisateurs dans un large éventail d'applications.

6. Conclusion

En conclusion, notre projet visant à compiler du code Python en langage C++ a rencontré des défis en raison des différences fondamentales entre ces deux langages. Cependant, nous avons surmonté ces difficultés pour obtenir un résultat bénéfique, car la compilation du code Python en langage C++ permet d'améliorer les performances d'exécution du code.

En ce qui concerne l'implémentation de notre projet, nous avons rencontré des difficultés pour gérer correctement l'indentation du code Python en C++, mais nous avons finalement trouvé des solutions pour surmonter ces obstacles.

En fin de compte, nous sommes satisfaits des résultats obtenus et espérons que notre travail pourra être utile pour les développeurs qui cherchent à améliorer les performances de leur code.

7. Références

- Flaticon, https://www.flaticon.com/de/kostenloses-icon/schlange_2186970?term=snake&page=1&position=41&origin=search&related_id=2186970