Création d'un langage Transpileur

Rapport d'élève ingénieur
Projet de 2ème année
Filière F2 : Génie Logiciel et Systèmes Informatiques

Présenté par : **Franck ALONSO** et **Rémi CHASSAGNOL**

Responsable ISIMA : **Mardi 31/01/2023**

Projet de 60h

Campus des Cézeaux. 1 rue de la Chébarde. TSA 60125. 63178 Aubière CEDEX

Table des matières

Remerciements

Nous tenons à remercier Loïc YON pour nous avoir guidé et conseillé lors de la réalisation du projet. Nous remercions aussi Murielle MOUZAT pour l'aide qu'elle nous a fourni à travers son cours pour la rédaction de ce rapport. Parmi nos professeurs, nous souhaitons aussi remercier Alexis PEREDA pour ses conseils sur l'utilisation de C++ et pour avoir partagé ses connaissances sur les compilateurs ainsi que Yves-Jean DANIEL pour m'avoir conseillé l'utilisation de Flex et Bison pour l'écriture d'un parseur.

Table des figures

Résumé

Ce projet s'inscrit dans le cadre du projet commun de la filière génie logiciel dont l'objectif est la création d'outils de démonstration servant à présenter la filière. Notre travail a consisté en la création d'un langage de programmation transpilé simple avec une syntaxe similaire à celle du C. Le programme doit lire un fichier de code source et de le traduire en code python. Il doit être capable de détecter les erreurs de syntaxes et de les spécifier à l'utilisateur. De plus, le langage est à typage statique, de ce fait, le programme doit aussi être capable de détecter les erreurs de types.

Le développement a été réalisé avec le langage C++ et emploi le paradigme objet. De plus, nous avons utilisé les outils GNU Flex et GNU Bison pour la création du parseur. À noter que le programme ne nécessite pas de système d'exploitation particulier, cependant, il requière l'installation des outils cités précédemment ainsi qu'une version de python3 pour fonctionner.

Au final nous avons réussi à implémenter un parseur fonctionnel ainsi que tous les éléments du langage à part pour l'inclusion de bibliothèques. Pour le transpileur, nous avons pu fournir une solution très simpliste qui ne correspond cependant pas à ce que l'on pourrait trouver sur un vrai transpileur. Le projet n'est donc pas entièrement terminé mais il présente tout de même une partie de solution et permet bien de traduire notre code en python.

Mots-clés : **C++**, **transpileur**, **lexeur**, **parseur**, **flex**, **bison**, **langage de programmation**

Abstract

This project is a part of the common project of the software development pathway. The objective is to create a tool for presenting software development. Our work consisted in the creation of a simple transpiled programming language with a syntax similar to C language. The program has to be able to read a source code file and generate a python program. It must detect syntax errors in the and report theme to the user. Furthermore, the language is statically typed so the program also has to detect and report type errors.

The development has been done using C++ and object oriented programming. Moreover, we have used the tools GNU Flex and GNU Bison to generate our parser. We can notice that the program doesn't require any particular operating system, however the tools previously quoted and a version of python3 have to be installed.

In the end we've been able to implement a parser and all the elements of our language without libraries inclusions. For the transpiler, we've made a very simple solution that doesn't correspond to what we could expect of a real transpiler. The project is not completely finished, however, it presents a piece of solution and the program eventualy is able to translate our code in python.

Keywords : **C++**, **transpiler**, **lexer**, **parser**, **flex**, **bison**, **programming language**

Introduction

Dans le cadre du projet de 2ème année à l'ISIMA, nous avons choisi de réaliser un travail concernant le sujet commun de la filière 2, génie logiciel et systèmes d'informations. Le but du projet commun est la création d'outils de démonstration servant à présenter la filière.

Nous souhaitions au départ, créer un langage de programmation interprété ainsi qu'un IDE, cependant, ce projet s'est avéré trop ambitieux pour être réalisé en 60 heures. Pour simplifier, nous avons décidé de concevoir un langage transpilé, délégant ainsi une partie des tâches complexes comme la gestion de la mémoire à un compilateur existant.

Ce projet permettra de présenter un langage de programmation très simple pour introduire des lycéen à la programmation. De plus, il pourra constituer une maquette pour les élèves de l'ISIMA souhaitant étudier le fonctionnement des compilateurs.

Pour présenter ce projet, nous commencerons par la forme du langage à créer souhaité, puis nous détaillerons sa structure. Une fois familiarisé avec les différentes étapes de son implémentation, nous introduirons les concepts et outils informatiques qui permettent la réalisation de notre langage.

Première partie

Contexte du projet

1 Concept de langage transpilé

L'objectif du projet commun est de créer un outils qui pourra servir à faire des démonstrations pour présenter la filière génie logiciel. Étant donné le fait que le code fait partie intégrante du développement logiciel nous avons décidé de créer un langage de programmation simple pouvant servir à l'initiation au développement ainsi qu'à certain concepts relatifs aux langages de programmation.

Il existe quatre familles de langages de programmations. La première est celle des langages compilés comme le C ou le C++. Le principe est de transformer le code source en assembleur puis de générer un exécutable. Cette méthode est la plus complexe, cependant les langages compilés sont généralement les plus rapides et les plus légers.

La seconde méthode pour exécuter du code est l'utilisation d'une machine virtuelle. Le code source du langage est traduit en un code de bas niveau, le bytecode, qui est ensuite exécuté par une machine virtuelle. C'est par exemple le cas du langage Java.

La troisième famille est celle des langages interprétés. Ici c'est l'interpréteur qui se charge directement de l'exécution du code source, sans passer par une représentation intermédiaire comme le bytecode.

La dernière famille est celle que nous allons étudier lors de la réalisation de ce projet, ce sont les langages transpilés. L'objectif d'un transpileur et de traduire le code source dans un autre langage. Le principal avantage du transpileur est qu'il permet de déléguer les taches complexes au compilateur du langage cible. Par exemple, cela évite d'avoir à écrire un garbage collector pour gérer la mémoire.

2 Les fonctionnalités du langage

Notre objectif global est de pouvoir écrire du code dans un fichier puis utiliser notre transpileur pour générer un fichier python (qui sera exécuté avec sous python3). Le langage doit être simple mais doit tout de même posséder les fonctionnalités suivantes :

Le langage doit permettre le stockage d'information dans des variables. Nous souhaitons que notre langage possède trois types de données, les caractères, les entiers et les flottants. Étant donné que l'objectif du projet est de pouvoir présenter la programmation, il est intéressant d'utiliser un langage qui possède des types car cela permet de mieux comprendre comment l'ordinateur gère les données.

On doit aussi avoir la possibilité d'écrire des fonctions et des procédures paramétrées qui pourront être utilisées dans différents endroits du programmes. Ces procédures et fonctions pourront contenir une suite d'instructions dans un bloque de code.

De plus, le langage devra posséder des structures de contrôles permettant la création de conditions et de boucles.

Enfin, il doit aussi intégrer des fonctions arithmétiques permettant les opérations de bases, des fonctions booléennes permettant de faire des tests ainsi que des fonctions permettant des actions d'entrée et de sortie (affichage et récupération de données via la console).

Étant donné le fait que notre langage pourra être utilisé pour faire des présentations lors de portes ouvertes, nous souhaitons aussi que la syntaxe soit assez éloignée de celle du langage python. Le but étant de pouvoir présenter un langage différent à des lycéens qui on pour la plupart déjà utilisé python.

3 Analyse du problème

Nous souhaitons créer un langage simple qui peut être écrit dans un fichier, la première chose à faire va donc être de définir sa syntaxe. La syntaxe du langage va dépendre des fonctionnalités décrites plus haut. Il faudra donc un moyen de déclarer des variables, des fonctions ou encore un moyen de délimiter les instructions.

Étant donné le fait que notre langage doit être écrit dans un fichier texte, il faudra un moyen d'extraire les informations contenues dans le texte. L'outil généralement utilisé pour ce type de tache s'appelle un parseur, qui va permettre de reconnaitre des règle de grammaires dans le code source. Nous avons décidé de ne pas implémenter de parseur par nous même mais plutôt d'utiliser deux outils qui vont permettent la génération du code du parseur, Flex et Bison. C'est deux outils vont nous permettre de gagner du temps et d'avoir du code beaucoup plus efficace car les codes qu'ils génère sont très performant.

Ensuite, il nous faudra un moyen de représenter les différentes instructions récupérées par le parseur. Pour ce faire, il existe plusieurs méthode comme par exemple, récupérer des instructions bas niveau (proche de l'assembleur) pour construire un programme. Une autre solution serait d'utiliser la programmation orienté objet pour gérer toutes les instructions du programme. Nous choisirons l'approche objet car il sera beaucoup plus simple de générer le code python au moment de la transpilation. Le code bas niveau aurait peut-être été un meilleur choix si nous avions implémenté un interpréteur ou un compilateur.

Le code python généré par le transpileur ne doit pas contenir d'erreurs de syntaxes ou de problème de définitions de variables ou de fonctions. Il faudra donc faire toutes les vérifications nécessaires au préalable. Pour ce faire, nous devrons trouver les erreurs de syntaxes dans le programme de base ainsi que de faire des vérifications sur la définitions et le type des différents éléments. La traduction du code ne devra se faire que si le programme de base ne comporte aucune erreur. Il faudra donc un moyen de récupérer les erreurs de compilations ainsi qu'une table des symboles pour vérifier les définitions et les types.

Pour la traduction du code il y a aussi plusieurs possibilités. Avec le modèle objet, on peut avoir un transpileur qui se charge de traduire les différents éléments ou avoir des méthodes de traductions pour chacun des éléments du programme. Il sera peut-être aussi nécessaire de modifier le programme récupéré avant la traduction. À noter que l'objectif du projet est de créer un langage assez simple, il n'y aura donc pas d'optimisation avant la traduction. Par manque de temps, nous choisirons l'option d'avoir des méthode de traduction des les objets des élément du langage.

Pour mener à bien le projet, il nous faudra donc un parseur pour reconnaitre la grammaire du langage et un moyen de représenter et de stocker les éléments du langage. Il nous faudra aussi un moyen de collecter les erreurs et une table des symboles pour vérifier les définitions et les types. Enfin il faudra faire la traduction des éléments du langage en code python.

Deuxième partie

Réalisation et conception

4 Conception des éléments du langage

4.1 Définition de la grammaire

Avant d'implémenter notre langage, il faut avoir une idée de sa grammaire. Puisque nous souhaitons utiliser ce langage à des fins pédagogiques, nous avons décidé de simplifier au maximum la syntaxe. Par exemple, nous avons choisit de supprimer tous les opérateurs, ainsi, toutes les opérations arithmétiques et booléennes auront la même syntaxe que les fonctions. Par exemple, $a + b$ s'écrira add(a, b). De plus, pour différer au maximum de python, notre syntaxe s'inspirera de celle des langages C et Rust. Pour définir la grammaire de manière formelle, nous utiliserons la forme de Backus-Naur.

Les variables

Pour stocker des données, notre langage utilise des variables dont la convention de nommage est la même qu'en **C**. Le langage est à typage statique, ce qui signifie que toutes les variables doivent être déclarées avec leur type avant d'être utilisées. Pour l'instant, nous possédons trois types : les entiers (*int*), les nombre à virgule flottante (*flt*) et les caractères (*chr*). Voici la syntaxe pour déclarer une variable :

```
<identifier> ::= 'a'-'z'( <alpha> | '0'-'9' )*
<int> ::= "int"
<flt> ::= "flt"
<chr> ::= "chr"
<type> ::= <int> | <flt> | <chr>
<declaration> ::= <type> <identifier> ";"
```

A noter qu'ici, on a une légère différence avec le C pour le nom des fonctions et des variable (**identifier** ici). En effet, le C permet aux noms de variables et de fonctions de pouvoir commencer par une majuscule. Ici, nous avons décidé de forcer les noms à commencer par une lettre minuscule, le non respect de cette règle entrainera une erreur de syntaxe. Le but étant d'une part de différer encore plus de python et d'autre part de forcer l'application de notre guide de style.

Les fonctions

La syntaxe pour les fonctions est la même qu'en Rust. Pour définir des fonctions, nous utilisons le mot clé **fn**, suivit du nom de la fonction avec ses paramètres entre parenthèses et enfin le bloque de code qui contient les instructions entre accolade. De plus, il faut spécifier le type de la valeur de retour de la fonction en utilisant **->** **type** quand c'est nécessaire.

```
<function> ::= fn <identifier>"("<parameters>")" [ "->" <type> ] "{"<instructions>"}"
```

Pour pouvoir retourner une valeur, il faudra utiliser le mot clé *return* dans la fonction.

Les structures de contrôle

Nous avons aussi ajouté les structures de contrôle présentes dans la plupart des langages de programmation. Il y a une structure *if* qui permet de tester une condition. Un *if* peut-être suivit d'un bloque *else* si besoin. À noter cependant que cette grammaire n'autorise pas les *else if*, dans ce cas il faudra mettre la

nouvelle condition à l'intérieur du bloque de code du *else*. La grammaire autorise aussi les boucles *while*, et la syntaxe est la même qu'en C. Enfin, il y a la boucle *for* qui utilise la règle *range* qui contient trois valeurs séparé par des virgules. C'est trois valeurs correspondent au début et à la fin de boucle et au pas. Par exemple, range(0, 10, 1) vas générer une boucle ou la variable de boucle prendra les valeurs de 0 à 9 avec un pas de 1. Il est obligatoire de toujours spécifier ces trois valeurs au *range*.

```
<if> ::= if "("<condition>")" "{"<instructions>"}" [ else "{" <instructions>"}" ]

<while> ::= while "("<condition>")" "{" <instructions> "}"

<range> ::= range "("<valueSymbol>, <valueSymbol>, <valueSymbol>")"
<for> ::= for <identifier> in <range> "{"<instructions>"}"
```

On rappel que même si la grammaire ne le précise pas, toute variable doit être déclarée avant d'être utilisée, il faudra donc déclarer la variable de boucle utilisée dans un *for*.

Les opérations

Comme dit précédemment, le langage ne comporte pas d'opérateur donc toutes les opérations se font à l'aide de fonctions directement intégrées dans le transpileur. Voici une liste des opérations possibles :

opération	syntaxe
$\mathbf{a} + \mathbf{b}$	$add(\mathbf{a}, \mathbf{b})$
a - b	$mns(\mathbf{a}, \mathbf{b})$
$\mathbf{a} \times \mathbf{b}$	$tms(\mathbf{a}, \mathbf{b})$
$\mathbf{a} \div \mathbf{b}$	$div(\mathbf{a}, \mathbf{b})$
$\mathbf{a} == \mathbf{b}$	$eql(\mathbf{a}, \mathbf{b})$
$\mathbf{a} < \mathbf{b}$	$inf(\mathbf{a}, \mathbf{b})$
$\mathbf{a} > \mathbf{b}$	$sup(\mathbf{a}, \mathbf{b})$
$\mathbf{a} <= \mathbf{b}$	$ieq(\mathbf{a}, \mathbf{b})$
$\mathbf{a} >= \mathbf{b}$	$seq(\mathbf{a}, \mathbf{b})$
not a	$not(\mathbf{a})$

Entrée et sortie

Le langage possède aussi la possibilité d'afficher et de lire du texte depuis la console. La lecture se fait avec la fonction *read* qui prend en paramètre une variable qui stockera le résultat. Pour la l'affichage, il faudra utiliser la fonction *print* qui peut être utilisée de deux manières :

— print("Hello, World!") : affiche du texte

— print(variable) : affichage d'une valeur.

Nous n'avons pas implémenter de fonction d'affichage similaire à **printf** en C. La première raison est le fait que notre langage ne supporte pas les fonctions variadiques. De plus, l'ajout de ce type de fonctions d'affichage nécessite la gestion d'une chaine de formatage, ce qui aurai grandement augmenté la complexité de notre grammaire si nous avions voulu l'intégrer directement au langage. Voici la syntaxe pour ces fonction :

```
<print> ::= print"("<variable>")" | print"("<string>")"
<read> ::= read"("<variable>")"
```

La règle *string* utilisé ci-dessus correspond à une chaîne de caractères entourée de guillemets comme montré plus haut.

Le programme

La syntaxe du programme est décrite ci-dessous. Un programme est composé de zéro ou plusieurs inclusion de fichiers ainsi qu'une ou plusieurs fonctions. La grammaire n'autorise donc pas l'ajout de code en dehors des fonctions, il ne sera donc pas possible de faire une programme avec juste une instruction d'affichage comme en python. De plus, il ne sera pas non plus possible de déclarer des variables globales.

```
<program> ::= <include>* | <function>+
```

Exemple de code

Dans cet exemple, nous disposons de 2 variables **a** et **b**. La variable **a** est lue par commande de l'utilisateur tandis que la valeur 4 est assignée à **b**.
Nous cherchons ensuite à additionner les 2 variables avec la valeur 5. Nous avons alors la syntaxe suivante :

— une fonction *add3* qui additionne les valeurs de ses 3 paramètres

— une fonction *affiche* qui affiche sur l'écran le nombre passé en paramètre

— une fonction *main*, où se trouve toutes les commandes voulues de notre programme

```
fn add3(int a, int b, int c) -> int {
    return add(a, add(b, c));
}

fn affiche(int n) {
    print("nombre : ");
    print(n);
}

fn main() {
    int a;
    int b;
    read(a);
    set(b, 4);
    affiche(add3(a, b, 5));
}
```

Maintenant que nous avons défini la syntaxe du langage nous allons voir l'implémentation d'un outils qui nous sera utile pour la construction d'un programme.

4.2 Arbre syntaxique

Pour donner un sens aux éléments textuels du langage, nous utiliserons une représentation sous forme d'arbre. On appel cela un arbre syntaxique ou AST. Les arbres syntaxiques sont nés avec la théorie des langages et comme on peut le voir dans cet article [1], les AST sont très utilisés par les compilateurs car ce sont des structures plus simples à manipuler que du texte. Voici l'arbre syntaxique de la fonction *add3* :

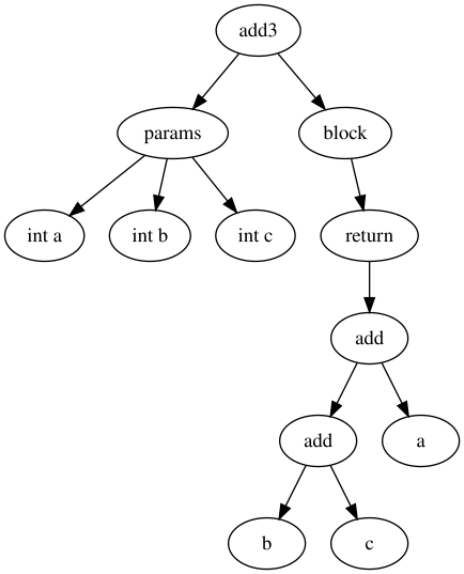


FIGURE 1 – Exemple d'arbre syntaxique

Pour cette partie, nous nous sommes inspiré d'un projet appelé **minijava** [2] ainsi que d'un article [3] qui couvre aussi l'utilisation des AST en java. Nous avons choisit l'approche objet, avec un AST représenté par des classes, où chaque nœud de l'arbre a sa propre classe.

Toutes les classes de l'arbre sont stockées dans le fichier AST/AST.hpp et héritent toutes d'une classe abstraite ASTNode. L'emploi de l'héritage ici est important car nous profiterons par la suite des avantages du polymorphisme pour stocker des opérations de différents types. La classe ASTNode est abstraite car elle ne doit pas être instanciée, il faut que chaque nœud ai un type concret utilisable dans le transpileur. Le diagramme de classes complet est disponible en annexe A. Détaillons maintenant les parties importantes.

Les bloques de code

La classe Block correspond aux bloques de code en entre accolades. Elle possède une liste de nœuds qui sont les opérations contenues dans le bloque. Nous verrons dans la partie suivante comment les opérations sont récupérées dans le parseur pour pouvoir construire les bloques.

Les conteneurs d'opérations

La classes Statement correspond aux éléments qui contiennent des blocks de code. Cela comprend les fonctions et les structures de contrôles. La grammaire ne permet pas l'emploi des bloques ailleurs. Comme on peut le voir sur le diagramme suivant, la classe Statement permet de transmettre un Block par héritage aux structures de contrôles et aux fonctions.

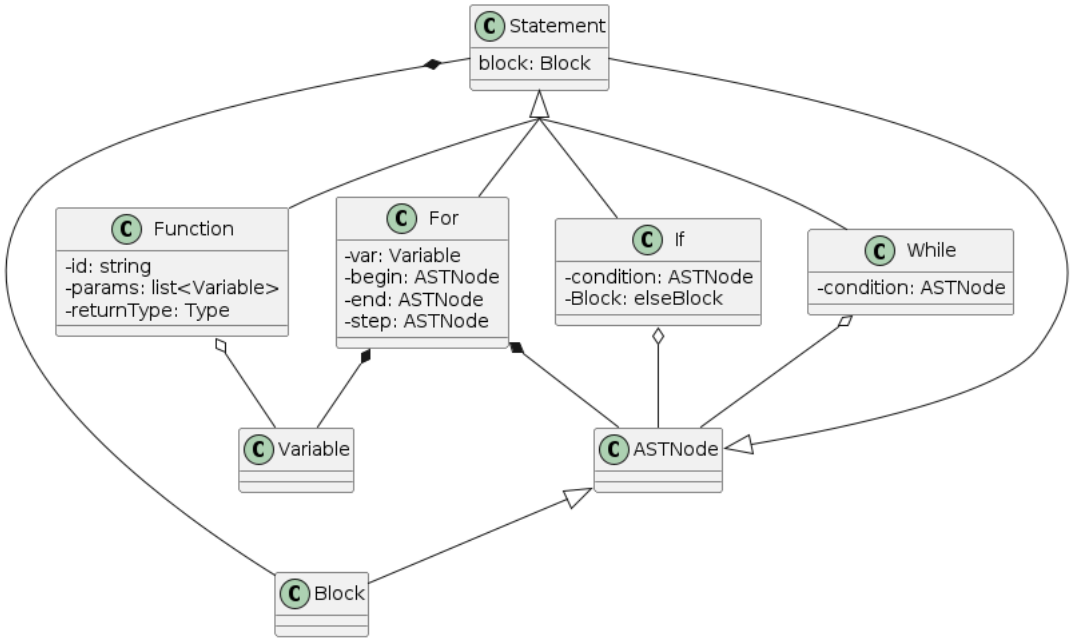


FIGURE 2 – Diagramme de classe : Statement

De plus, on peut aussi voir sur ce diagramme la composition de la classe fonctions ainsi que celles des classes des structures de contrôle.

La classe *Function* possède un attribut *id* qui correspond au nom de la fonction. Elle possède aussi un liste de variables qui sont ses paramètres ainsi qu'un type de retour (le type *Type* est détaillé plus loin).

La classe *For* possède une variable qui est la variable de boucle et les trois valeurs fournis par le *range* décrites précédemment.

La classe *If* possède un nœud qui contiendra l'opération booléenne de la condition. Elle possède aussi un second *Block* qui contient les instructions du *else* s'il y en a un.

Enfin la classe *While* possède comme *If*, un nœud pour la condition.

Les opérations

Les opérations sont traitées avec la classe OperationBinaire qui correspond à tous les opérateurs. Cette classe possède deux attributs de type *ASTNode* qui représentent les parties gauche et droite d'une opération binaire. Pour ne pas trop alourdir le schéma ci-dessous, seul la classe *AddOP* a été représentée. Cependant, toutes les fonctions décrites dans partie 4.1 ont une classe qui hérite de OperationBinaire.

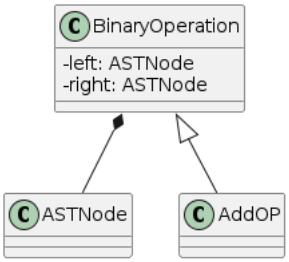


FIGURE 3 – Diagamme de classe : BinaryOperation

Les éléments typé

Les fonctions, les valeurs et les variables possède un champ type. Le type **Type** est une énumération qui possède les éléments suivant :

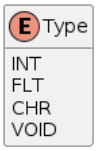


FIGURE 4 – Enumération Type

Le champ INT correspond au type des entiers, le champ FLT au type des flottants et le champ CHR au type des caractères. Le type VOID est utilisé pour le type de retour des procédures et il est aussi utilisé par le parseur pour créer des variables qui n'ont pas été déclarées.

Les valeurs

Le dernier élément de l'AST que nous traiterons est la classe Value qui permet de représenter les valeurs utilisées dans code. Comme on peut le voir sur le diagramme ci-dessous, cette classe possède deux attributs, *type* qui représente le type de la valeur et *value* qui permet de stocker la valeur. Le type *type_ t* est une union qui contient un caractère, un entier et un flottant et permet donc de stocker des valeurs pour ces trois types.

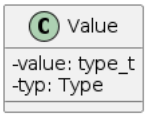


FIGURE 5 – Diagramme de classe : Value

Nous avons définit la grammaire du langage et déterminer les différents éléments qui vont composer notre arbre syntaxique. Dans la partie suivante nous écrirons la grammaire décrite précédemment dans le langage de bison et nous montrerons la construction d'un programme en utilisant les classes de l'AST que nous venons de décrire.

5 Construction de l'AST

Dans cette partie nous expliquerons la création d'un parseur en utilisant les outils **GNU Flex** et **GNU Bison**. De plus, nous détaillerons la génération de l'AST en utilisant les classes décrites précédemment. Pour cette partie, nous nous sommes aidé d'un article [4] qui présente la construction d'un compilateur avec Flex et Bison en C. Pour porter le code en C++, nous avons utilisé la documentation officielle des deux outils ainsi qu'un article qui nous a permis d'avoir un squelette de base pour le lexeur et le parseur [5].

5.1 Le lexeur

Pour faciliter le travail du parseur, nous allons utiliser un lexeur. Dans cette partie, nous détaillerons la conposition du fichier de configuration qui nous permettra de générer le code du lexeur avec l'outil Flex. Nous verrons dans la partie suivante comment utiliser les tokens dans les règles de grammaire.

Exemple

Pour l'expression simple **a = 2 * b**
Les tokens apparaissant sont :

Token	Sa nature
a	Identificateur de variable
=	Symbole d'affectation
2	Valeur entière
*	Opérateur de multiplication
b	Identificateur de variable

Le lexeur a également pour rôle de supprimer les informations inutiles, généralement du caractère blancs (espaces et tabulations) et des commentaires.

Implémentation

Comme évoqué précédemment, l'outil utilisé pour générer le lexeur est **GNU Flex** (Fast LEXical analyser generator). Il permet de générer le code C++ du lexeur à partir d'un fichier. Dans notre cas, le fichier utilisé sera **main_cpp.l** et possède la structure suivante [4] :

```
// code C++, options et declarations de raccourcis

%%
// Definition des tokens et actions

%%
// Fonctions C++
```

Dans la première partie en haut du fichier, on place les inclusions de bibliothèques C/C++ ainsi que des options pour flex.

```
%{
#include "parser.hpp"
#include "lexer.hpp"
%}

%option c++ interactive noyywrap yylineno nodefault outfile="lexer.cpp"
```

Détail des fonctions utilisée :

— **c++** : indique qu'on travail avec du cpp et non du c
— **interactive** : utile quand on utlise **std : :in**. Le scanner interactif regarde plus de caractères avant de générer un token (plus lent mais permet de lutter contre les ambiguitées)
— **noyywrap** : ne pas appeler **yywrap()** qui permet de parser plusieurs fichiers
— **nodefault** : pas de scanner par défaut (=> on doit tout implémenter)
— **outfile :"file.cpp"** : permet de definir le fichier de sortie

Après les options on peut définir des raccourcis en utilisant des expressions régulières. Par exemple, dans le code ci-dessous, nous avons définit les règles suivantes :

— **alpha :** un caractère alphabétique est composé d'une lettre minuscule ou d'une lettre majuscule.
— **digit :** les chiffres sont les caractères entre 0 et 9.
— **int :** les entiers correspondes à une suite de chiffres et peuvent être positifs ou négatifs.
— **float :** similaire aux entiers sauf qu'ici on a obligatoirement un point suivit d'une suite de chiffre à la fin.
— **char :** les caractère sont toujours écrit entre **'** (par exemple **'a'**).
— **identifier :** correspond aux noms de fonctions et de variables et suit le standard du C. Un identifiant commence par une lettre minuscule et peut être suivit d'une suite de lettres, de nombre et de **_**.

```
alpha [a-zA-Z]
digit [0-9]
int [+-]?{digit}+
float [+-]?{digit}+\.{digit}+
char '{alpha}'
identifier [a-z]({alpha}|{digit}|_)*
```

On peut déjà noter que nous venons de définir l'identifiant qui correspond aux noms des variables et fonctions et que cela correspond bien à ce qui été énoncé dans la grammaire.

Dans la seconde partie du fichier, on définit des règles et des actions. À noter que l'on peut utiliser les raccourcis définis précédemment en mettant leurs noms entre accolades comme fait ci-dessous pour **identifier**. La définition d'une règle suit le principe suivant, on commence par donner une suite de caractères qui sera consommée. Ensuite ajoute du code entre accolades, qui sera exécuté quand le lexeur consommera la chaine. Par exemple, si on prend la première ligne ci-dessous, quand le lexeur trouvera le mot **for**, il affichera *L_for* dans le terminal puis retournera le token **FOR**.

À noter que les tokens disponibles dans l'espace de nom **Parser : :token** doivent être définis dans le ficher bison que l'on détaillera dans la partie suivante.

```
for          { AFFICHE("L_for"); return Parser::token::FOR; }
{identifier} {
  AFFICHE("L_id");
  yylval->build<std::string>(yytext);
  return Parser::token::IDENTIFIER;
}
```

Ici on a accès à la variable yylval de type Parser : :semantic_type* qui possède une méthode build permettant de transmettre des valeurs à bison.

La fonction appelée par défaut est yylex, cependant, pour pouvoir travailler avec bison, nous devons fournir nos propres fonctions, pour ce faire on utilise la macro YY_DECL, comme expliqué dans la partie *9 The Generated Scanner* du manuel pour Flex [6].

```
#define YY_DECL int interpreter::Scanner::lex(Parser::semantic_type *yylval, Parser::
    location_type *yylloc)
```

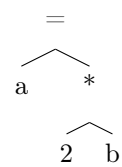
Cette macro permet de définir le type de la fonction **lex**, ici, on a pour paramètre yylval (qui comme dit précédemment permet de transmettre des valeurs à bison) et yylloc qui doit être fourni quand on utilise les positions dans le parseur (pour avoir le numéro de ligne en cas d'erreur par exemple), mais cela nécessite une option particulière pour bison.

5.2 Le parseur

Dans cette partie nous allons utiliser le travail précédemment effectué sur le lexeur pour écrire des règles de grammaires. Nous détaillerons la composition de ces règles et en quoi elle permettent la construction de notre arbre syntaxique.

Exemple

Pour l'expression simple **a = 2 * b**
On peut voir dans le tableau suivant l'analyse d'une affectation par le parseur :

Arbre syntaxique	Évaluation de 2 * b	Affectation de a
		a = 2 * b

Implémentation

À l'instar de Flex pour le lexeur, Bison est un générateur de grammaire qui convertit une description de grammaire en un programme C++ qui analyse cette même grammaire.

Le fichier **main_cpp.y** contient le code qui permet de générer le parseur avec Bison. Toutes les règles syntaxiques qui définissent la grammaire du langage y sont comprises. Chaque règle va contenir des bloques de code qui seront exécutés au moment où le parseur la reconnait, ce code permet de construire l'ABS du programme ainsi que de faire de la vérification sur les symboles (définition et type), comme nous le verrons plus tard.

La structure du fichier **main_cpp.y** est similaire à celle du lexeur :

```
// C++, options, et declaration des tokens

%%
// Regles de grammaire

%%
// fonctions C++
```

Les tokens sont définis en début de fichier avec la syntaxe suivante :

```
%token IF ELSE FOR WHILE FN INCLUDE IN
%token <long long>  INT
%token <double>     FLOAT
%token <char>       CHAR
%token <std::string> IDENTIFIER
```

On peut aussi voir dans le code ci-dessus que certains token possède un type. Cela permet de pouvoir récupérer les valeurs retournées par le lexeur. Il est aussi possible de déclarer le type d'une règle de grammaire pour que celle-ci puisse retourner une valeur comme nous le verrons plus loin.

L'objectif de bison est de pouvoir générer le code d'un parseur à partir des règles définis dans le fichier. Voici un exemple de règle :

```
function:
        FN IDENTIFIER[name]'('paramDeclarations')' block[ops]
        |
        FN IDENTIFIER[name]'('paramDeclarations')' ARROW type[rt] block[ops]
        ;
```

Dans le code ci-dessus, on définis la règle pour les fonctions. La définition d'une règle se fait de la façon suivant :

- On donne le nom de la règle suivit de ' :'.
- On décrit ensuite la règle en utilisant les tokens ou directement des caractères comme c'est le cas ci-dessus pour les parenthèses. On peut aussi utiliser les autre règles définis dans le fichier, par exemple, ici on utilise les paramètres, les bloque ainsi que les types.
- Le caratère **'|'** est utilisé lorsque la règle est composée de plusieurs sous règles. Dans le code ci-dessus, il y a deux sous règles pour les fonction, une ou on spécifie le type de retour et l'autre ou on ne le spécifie pas. À noter que l'on ne peut pas ajouter d'éléments optionnels comme on pouvait le faire avec la forme de Backus-Naur entre **'[]'**. Dans ce cas il faut définir plusieurs sous règles.

On peut ajouter des bloques de codes qui seront exécutés au moment où le parseur atteint l'élément qui précède le bloque. Si on reprend l'exemple des fonction on obtient le code suivant :

```
function:
        // premiere regle sans le type de retour
        |
        FN IDENTIFIER[name]'('paramDeclarations')' ARROW type[rt]
        {
          contextManager.newSymbol($name, $rt, FUNCTION);
          contextManager.enterScope();
        }
        block[ops]
        {
          pb.createFunction($name, $ops, $rt);
          contextManager.leaveScope();
        }
        ;
```

Dans cet exemple on a ajouté deux bloques de code. Le premier sera exécuté juste avant de renter dans le bloquer d'instructions *block* de la fonction. Le second sera exécuté une fois que le bloque d'instructions aura été traité. On peut voir que dans les bloques que nous venons d'ajouter, nous pouvont accéder au valeurs des différents éléments qui compose la règle. Par exemple, $name permet d'accéder à la valeur de l'identifiant et de la même façon, $ops permet de récupérer la valeur renvoyée par la règle **block**. À noter qu'ici tous les éléments que l'on doit accéder on été nommé, cependant, il n'est pas obligatoire d'ajouter un nom entre **'[]'**. Si il n'y a pas de nom, on utiliser des nombre, par exemple, s'il n'y avait pas **[name]** derrière **IDENTIFIER**, on aurait utilisé $1 car c'est le premier élément qui possède une valeur (le token **FN** n'a pas de type et Flex ne retourne rien).

Comme dit plus haut, on peut définir une valeur de retour pour les règles, pour ce faire, il faut définir un type à la règle, dans la première partie en haut du fichier puis utiliser $$. L'exemple ci-dessous correspond à la règle pour les valeurs, dans ce cas on on récupère la valeur retournée par Flex, puis on

instancie une nouvelle *Value* qui est ensuite retournée.

```
// en haut du fichier
%nterm <Value> value

%%
// regles

value:
     INT {
       type_t v = { .i = $1 };
       $$ = Value(v, INT);
     }
     |
     FLOAT {
       type_t v = { .f = $1 };
       $$ = Value(v, FLT);
     }
     |
     CHAR {
       type_t v = { .c = $1 };
       $$ = Value(v, CHR);
     }
     ;
```

Pour la génération du code, on a deux options :

— utiliser des instruction très simples (comme du bytecode)

— créer un code objet où tous les éléments sont des objets.

Choix de la représentation objet :

— plus simple à comprendre et à visualiser

— plus compliqué à générer : on peut générer du bytecode au fil de l'exécution du parseur, en utilisant des goto pour sauter de block d'instruction en block d'instruction. Pour le code objet, les éléments à l'intérieurs des blocks doivent être créés avant le block, et le block est détecté avant les instructions, il faut donc stocker les instructions.

5.3 Fabrique à programme

Pour construire l'arbre syntaxique dans le programme, nous utiliserons la classe **ProgramBuilder**. Cette classe permet de stocker des éléments comme le nom de la fonction courante ou les commendes récupérée par le parseur. De plus cette classe permet d'instancier les éléments de l'AST pour construire le programme. Le diagramme de cette classe est le suivant :

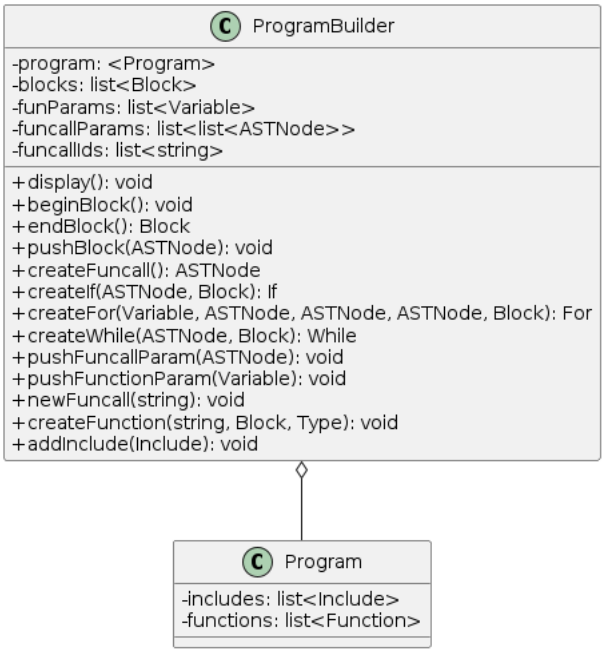


FIGURE 6 – Diagramme de classe : ProgramBuilder

On peut noter sur ce diagramme que l'on a des fonctions de créations qui permettent de créer les éléments de l'AST définis dans la première partie. De plus la classe est composé du programme, auquel elle ajoute les fonctions et les fichiers inclus.

L'emploi de cette classe permet de limiter le code dans le parseur mais aussi de pouvoir récupérer des éléments dans des piles quand on ne connait pas leurs nombre. Par exemple, pour les **Block**, on ne sait pas combien de commandes ils contiennent, c'est pourquoi le **ProgramBuilder** possède une pile qui permet d'ajouter les commandes au bloques ainsi que d'avoir accès au dernier bloque pour construire les **Statement**. Il en est de même pour les paramètres de fonctions, dans ce cas nous avons une liste de liste. Si une fonction **A** prend le résultat d'une autre fonction **B** en paramètre (*A(..., B(...), ...)*), la dernière liste contient les paramètres de **B** au moment ou **B** est traitée. À l'origine, ce choix avait été fait car il permet de toujours avoir accès aux éléments qui sont en train d'être récupérer depuis n'importe quel endroit du parseur. Cependant, cette fonctionnalité n'est pas utilisée, il aurait donc été plus simple de construire ces listes en utilisant les valeurs retour de bison.

Le **ProgramBuilder** permet donc de stocker les éléments important ainsi que de construire le **Program** au fur et à mesure de l'exécution du parseur.

6 Analyse des symboles

Dans cette partie, nous allons créer une table des symboles qui, de façon similaire à ce que l'on voit dans [4], permet de vérifier si les symboles (variables et fonctions) utilisés ont bien été déclarés. Cependant, ici notre table des symboles sera plus complexe du fait que les symboles peuvent avoir des portées différentes. De plus la table des symboles nous permettra aussi de faire de la vérification de types pour par exemple, vérifier qu'une fonctions prend les bon paramètres.

Du fait que nous allons devoir faire de la détection d'erreurs, nous créerons une nouvelle classe qui permettra de stocker tous les messages d'erreurs (et avertissements) au fil de l'exécution du parseur. Ces messages seront affichés à la fin une fois tout le code source traité. À noter que la présence d'erreurs empêche la compilation.

6.1 Gestion des symboles

La table des symboles

Pour la création de notre table des symboles nous utiliserons la méthode présenté dans cet article [7] que nous avons simplifié pour la récupération et la recherche de symboles.

Commençons par étudier la composition d'un symbole. Chaque symbole est composé d'un nom qui lui sert d'identifiant, d'un type et d'un genre. Le genre est une énumération qui suis le même principe que dans [7]. Le type du symbole est une liste de **Type** qui est une énumération décrite dans l'AST, il a la même utilité que pour les éléments typées (voir 4.2). Ici nous stockons une liste de *Type* et non juste un *Type*. En effet le type d'une variable est bien composé d'un seul *Type* (une liste de un élément) mais le type des fonctions ne correspond pas uniquement au type de la valeur de retour. Le type d'une fonction comprend aussi le type de chacun des paramètres.

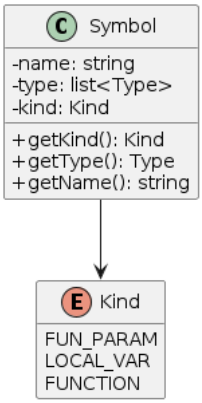


FIGURE 7 – Diagramme de classe : Symbol

Pour implémenter la table des symbole nous nous sommes inspiré du schéma suivant [8]. Ici la table des symboles est une table de tables. Chaque sous table correspond à une zone de code dans laquelle peuvent être définis des symboles (**scope**). Cette représentation est intéressante car elle permet de pouvoir aisément vérifier si un symbole existe en remontant l'arborescence de la table jusqu'aux **scope** global. Les éléments stockés dans la table mère sont toujours accessibles.

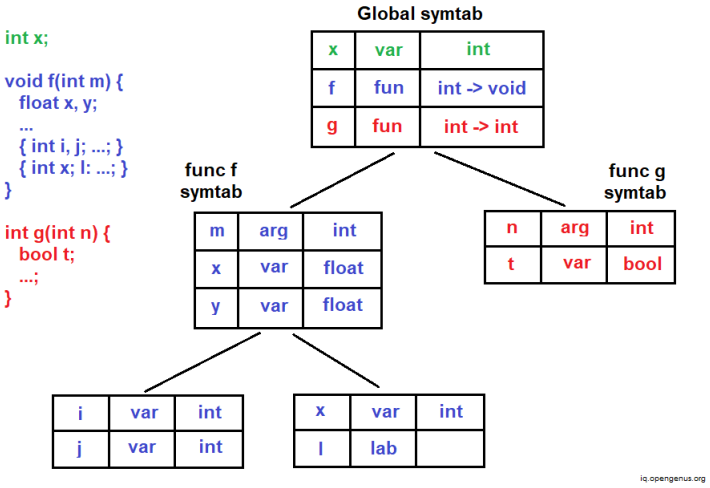


FIGURE 8 – Structure de la table des symboles

Pour l'implémentation, nous suivrons le diagramme ci-dessous. Les symboles sont stockés dans une **unordered_map** qui est une table de hachage qui permettra l'accès aux valeurs avec l'identifiant. De plus, la table possède un liste de sous tables qui et un pointeur sur la table mère qui sera utilisé par la méthode **lookup** pour la recherche de symbole comme expliqué précédemment. À noter que la méthode **lookup** ne retourne pas directement un symbole. Elle utilise la classe std :: optional et retourne une valeur vide dans le cas ou le symbole recherché n'a pas été trouvé, ce qui est pratique étant donné le fait que C++ ne possède pas de valeur nulle. Une autre solution aurait été de retourner un pointeur.

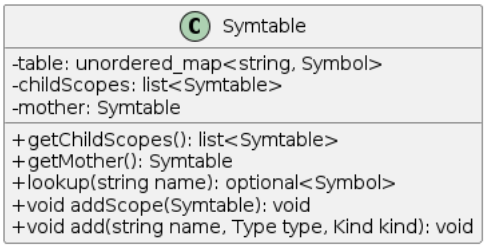


FIGURE 9 – Diagramme de classe : Symtable

Nous avons donc une table des symboles qui nous permet de connaître tous les éléments nommés du programme, leur porté ainsi que leur type. Dans la partie suivante, nous détaillerons le fonctionnement de l'objet responsable de la création de la table dans le parseur.

Gestion du contexte

Nous avons une table des symboles, cependant il nous faut un moyen de gérer les *scopes*, et donc de gérer l'arborescence de la table. Pour ce faire nous aurons une classe **ContextManager** qui permettra de créer des nouveaux nœuds dans l'arborescence de la table des symboles à chaque fois que l'on entre dans une nouveau *scope*. La structure de cette classe est la suivante :

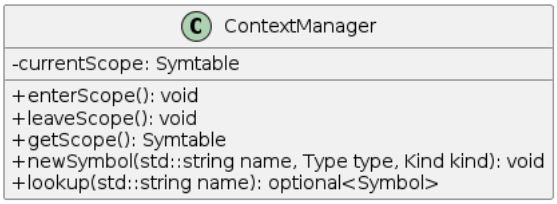


FIGURE 10 – Diagramme de classe : ContextManager

Le **ContextManager** possède un attribut **currentScope** qui est pointeur sur la sous table qui contient le scope courant. La classe possède aussi deux méthodes permettant de renter ou sortir d'un scope. La méthode **enterScope** créer une nouvelle sous table dans la table des symboles et pointe dessus avec **currentScope**. Lorsqu'on sort du scope courant, on retourne dans le *scope père*, la méthode va donc faire pointer **currentScope** sur la table mère. Enfin cette classe possède des méthodes pour ajouter ou rechercher des symboles qui font appel au méthode de la sous table courante.

Le **ContextManager** permet donc de créer une table des symboles dans le parseur, ainsi que de pouvoir faire toutes les vérifications décrites dans la partie précédente. La partie suivante concerne la création d'une classe pour la gestion des erreurs de compilation. Nous verrons ensuite comment combiner toutes ces classes dans le parseur pour pouvoir collecter et afficher toutes les erreurs de types et de définitions.

6.2 Gestion des erreurs

L'objectif de cette partie est de pouvoir vérifier si les éléments du programmes ont bien été définis et possèdent les bon type. Dans le cas contraire, cela va impliquer de devoir retourner les erreurs qui conviennent. Pour ce faire, nous avons une classe dédiée à la collections des erreurs et des avertissements dans le parseur, ErrorManager.

Comme on peut le voir ci-dessous sur le diagramme, c'est une classe très simple qui ne possède que deux attributs. Le premier et un flux de caractères qui va permettre de stocker tous les messages d'erreurs. Le second est un booleen, il est initialisé à *false* au départ et passe à *true* quand une erreurs est enregistrée. Cet attributs sert à savoir si on doit compiler ou pas quand le parseur a consommé tous le texte. En effet, seules les erreurs de syntaxes sont bloquantes, et cause un arrêt du parseur avec un retour non nul. Cependant, il n'est pas nécessaire de planter au niveau du parseur pour les erreurs de types. De ce fait il nous est possible de récupérer toutes les erreurs de type dans un programme qui ne comporte pas d'erreur de syntaxe et donc de rendre la correction d'erreurs plus rapide.

Concernant les méthodes, la classe possède une méthode *report* qui doit être appelée après le parseur et affiche tous les messages enregistrés. La classes possède aussi deux méthodes permettant l'ajout de nouvelles erreurs et de nouveaux avertissements. La différence entre les deux est que les avertissements ne vont pas empêcher la transpilation. Par exemple, un avertissement peut être levé lors de l'affectation d'un caractère à une variable de type entier. Ici, l'avertissement va permettre de signifier à l'utilisateur que les

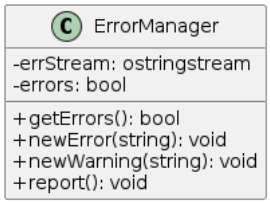


FIGURE 11 – Diagramme de classe : ErrorsManager

type diffères, cependant, ce dernier souhaite peut-être extraire le numéro ASCII du caractère assigné, il ne faut donc pas planter dans ce genre de cas.

6.3 Traitement des des symboles dans le parseur

Dans cette partie, nous allons utiliser les composants implémentés précédemment pour les intégrer dans le parseur. Nous allons donc modifier les bloques de codes dans le fichier bison pour faire des vérifications lorsque c'est nécessaire.

Vérification des définitions

Ici, notre objectif va être de vérifier que toutes les variables utilisées dans le code ont été déclarées au préalable. Nous vérifierons aussi que toutes les fonctions appelées dans le programme ont été définis. Enfin, il faudra aussi s'assurer de la définition d'un fonction *main*. En effet, de façon similaire au C, la grammaire de notre langage n'autorise pas l'ajout d'instructions à l'extérieur des fonctions. De ce fait, tout programme doit avoir un point d'entrée.

Pour faire les vérifications, nous avons implémenté diverse fonctions. La première fonction *isDefined* prend en entré un nom, les numéros de ligne et de colonne ainsi qu'une référence sur une liste de *Type*. Cette fonction renvoie vrai lorsque le symbole demandé (le nom) est défini et elle modifie le type par effet de bord. De ce fait, on peut accéder facilement au type dans les bloques de code du parseur. La fonction se charge aussi de générer une erreurs en utilisant les numéros de ligne et de colonnes pour indiquer la position de l'erreur.

Nous avons aussi une fonction qui permet de vérifier le type des paramètres lors d'un appel de fonction. Celle-ci n'utilise pas directement la table des symboles, elle prend en paramètre deux listes qu'elle compare et renvoi un booléen en fonction du résultat.

Une seconde fonction de vérification de type à été écrite, mais celle-ci génère un avertissement et non une erreur. Elle est utilisée au niveau de l'affectation pour ne pas générer d'erreur lorsque l'on fait de la conversion de type comme expliqué précédemment.

Enfin une fonction *getType* qui permet de récupérer le type de la valeur d'un symbole (uniquement le type de retour s'il s'agit d'une fonction).

À noter que cette partie n'est pas idéale mais n'a pas pu être modifiée à cause des contraintes de temps. Nous détaillerons le problème que nous avons ici ainsi qu'une solution dans la partie discussion.

La vérification des définitions se fait se fait à différents endroits. Tout d'abord elle se fait au niveau de la règle de l'affectation pour vérifier si la variable a qui on affecte une valeur à bien été déclarée. Ici

on utilise bien la fonction *isDefined* qui génère un message d'erreur expliquant à l'utilisateur qu'il tente d'affecter une valeur à une variable non définie. On fait aussi une vérification au niveau de la règle qui permet de récupérer tous les élément qui peuvent avoir une valeur. Ici, on peut avoir des fonctions comme des variables. La vérification se fait au niveau de la sous règle composée uniquement d'un *IDENTIFIER*, car on ne souhaite pas faire la vérification sur des *Value* ou des opérations arithmétiques.

Pour les fonctions, la vérification est aussi faite au niveau de la règle qui décrit les appels de fonctions. Comme nous l'avons expliqué, *isDefined* se charge aussi de récupérer le type de la fonction appelée. Nous détaillerons dans la partie suivante la vérification du type. On peut voir ici que la fonction *isDefined* est appelée deux fois. La vérification est faite une première fois au niveau de la règle qui décrit les symboles avec une valeur, et une seconde ici. Ici, *isDefined* permet uniquement de savoir si on doit tester le type de la fonction appelée.

À noter que comme les symboles sont récupérés et testés en même temps que le parseur construit l'arbre, la collection des symboles suis le sens du texte. De ce fait, toute fonction utilisée doit être définie au dessus de la fonction dans laquelle elle est appelée. Cela peut être un problème car notre grammaire ne permet pas la déclaration de fonctions, de ce fait il est pour l'instant impossible d'avoir deux fonctions qui s'appellent entre elles sans avoir d'erreur.

La vérification de la définition de la fonction *main* se fait en dernier, juste avant de transpiler. Nous savons que quand le parseur à fini de tourner le *ContextManager* est retourné au *scope* global. Il suffit donc de vérifier que la méthode *lookup* retourne bien une valeur lorsque l'on fait une recherche du symbole *"main"*. À noter que du fait que la grammaire n'autorise pas de variable globale, il ne peut y avoir qu'un seul symbole nommé *main* dans le *scope* global, et celui ci correspond forcément à une fonction. Il n'est donc pas nécessaire d'utiliser l'attribut *Kind*, cependant, si l'on souhaitait ajouter des variables globales, il faudrait faire plus de vérifications. De plus, nous n'avons pas implémenté la possibilité de donner des arguments au programme, ce n'est donc pas géré dans cette partie.

Vérification des types

La seconde vérification à faire concerne les types des éléments. Pour la récupération des types nous utiliserons certaines des fonctions décrites dans la partie précédente.

Tout d'abord on fait une vérification de type de au niveau de l'affectation. Ici, on compare le type de la variable avec le type de la valeur affectée. On utilise bien la fonction qui génère un avertissement et non une erreur pour la conversion de type. La récupération des type se fait avec *isDefined* pour la variable et pour la valeur on utilise la fonction *getType* décrite plus haut.

C'est pour les fonctions que la vérification du type est complexe. Comme dit précédemment, le type des fonctions comprend aussi leurs paramètres. La vérification se fait en deux phases. Au niveau des appels de fonctions, on va simplement vérifier le type de chacun des paramètres sans prendre en compte le type de retour. Par contre, lorsque la fonction est traitée comme une valeur, on ne regarde que le type de retour. Par exemple, si on regarde le code ci-dessous.

```
fonction1(a, 1, fonction2(3, 3));
```

Dans cette exemple, on souhaite vérifier le type de *fonction1*. Ici, c'est un appel de fonction, on va donc vérifier les paramètres. Lors de cette vérification, *fonction2* sera traitée comme une valeur, on va donc comparer le type de la valeur de retour de *fonction2* avec celui du dernier paramètre de *fonction1*. À noter que la vérification des paramètres de *fonction2* a été faite au moment où l'appel de fonction a

été créer, donc avant la vérification de *fonction1*.

Par manque de temps, la conversion des types au niveau des appels de fonctions ne sera pas prise en charge. De ce fait, si une fonction prend en entrée un entier, il faudra obligatoirement lui donner une variable ou une valeur de type entier en paramètre pour ne pas avoir d'erreur. La conversion de type au niveau des variables peut cependant ce faire avec la fonction *set*, le manque de cette fonctionnalité n'est donc pas un problème.

Une dernière vérification de type est fait au niveau de la règle du *return*. Ici, on récupère le type de la fonction courante et on le compare avec le type de la valeur renvoyée. Si les types ne correspondent pas, une erreur est renvoyée.

Nous avons décrit la construction du l'arbre syntaxique du langage et expliqué la vérification des définitions et des types. De ce fait, nous somme capable de donner un sens au code source ainsi que de vérifier qu'il est correcte. La dernière étape concerne donc la traduction de l'arbre en python, ce qui fait l'objet de la partie suivante.

7 Le transpileur

Dans cette partie, nous allons utiliser l'arbre syntaxique construit dans les parties précédentes pour générer du code python. Pour la réalisation de cette partie, nous n'avons pas eu le temps d'implémenter la solution que nous souhaitions au départ. Cependant nous avons utiliser une solution temporaire plus simple pour vérifier que nous étions bien capable de générer du code correcte en python. De plus, les éléments de notre transpileur actuel peuvent être réutilisés pour créer le transpileur de façon plus propre.

7.1 La solution idéale

Comme on peut le voir dans cet article [9], une bonne solution aurait nécessiter la génération d'un second arbre syntaxique correspondant à la syntaxe de python. En effet, le principe du transpileur et de faire une traduction d'un langage vers un autre, cependant, la syntaxe et les fonctionnalités du langage de départ peuvent être très différentes de celui d'arrivé. Cela implique que certains éléments du langage source deviennent impossible à traduire dans le langage cible. Par exemple, si le langage source est un langage fonctionnel, ou toutes les boucles doivent être remplacées par des fonctions récursives. Si nous souhaitons transpiler ce type de langage en python il faudra obligatoirement supprimer la récursivité car python limite le nombre de fois qu'une fonction peu s'appeler elle même (en plus du fait que cela soit très lent). Dans ce cas extrême il y aurait beaucoup de modification à faire et il faudrait dérécursifier toutes les fonctions de l'AST d'origine pour créer l'AST de python.

7.2 Notre solution temporaire

Par manque de temps, nous avons du improviser une solution qui n'utilise pas de deuxième arbre syntaxique. Nous avons donc décidé de directement faire la traduction de notre arbre en python. Cela est possible car en l'état, notre langage est très simple et ne possède pas de fonctionnalité que l'on ne peut pas directement réécrire en python. Cela pourra cependant poser beaucoup de problèmes si nous décidons d'ajouter de nouveaux éléments à notre langage ou encore si nous souhaitons changer le langage cible pour du C par exemple.

Pour faire notre traduction, nous avons ajouté une méthode abstraite *compile* à la classe *ASTNode*. Cette méthode est implémentée par tous les éléments du langage. La méthode prend deux paramètre, le premier est un flux std :: ofstream qui permet d'écrire dans le fichier de sortie. Le second paramètre est un entier qui permet de connaitre le nombre de tabulations à écrire pour indenter le code car python fonctionne avec l'indentation.

8 Déroulement du projet

8.1 Les outils utilisés

8.1.1 Gestion de version

Pour la gestion de version nous avons utilisé **git**, avec un dépôt distant stocké sur github pour faciliter le partage du code. L'utilisation de git permet d'avoir un suivit des modifications ce qui est utile lorsqu'une mauvaise modification entraine de grosses erreurs. Le système de branche de git permet aussi d'avoir un suivit des fonctionnalités et de pouvoir facilement travailler à plusieurs. Cependant, du fait de

la petite taille du projet et du faible nombre de personnes dans le groupe, nous n'avons pas eu besoin de mettre en place une méthode tel que gitFlow. Nous n'avons pas non plus eu le temps de mettre en place des tests automatisés bien que cela aurait pu être nécessaire sur la fin du projet pour tester le code python généré avec le transpileur. Une suite au projet nécessitera certainement la mise en place d'une tel structure de tests automatisés.

8.1.2 Compilation

Pour compiler le programme, nous avons au départ utilisé un *Makefile*. Cependant, nous avons basculé sur *cmake* au moment de la création de la table des symboles. Cmake permet de générer un *Makefile* très complet à partir d'un fichier de configuration. Un des avantages est que le fichier de configuration de *cmake* est moins verbeux que *Makefile* car il automatise la gestion des bibliothèques, des dépendances ainsi que des fichiers objets. De plus le langage de *cmake* permet beaucoup de choses comme par exemple la possibilité de faire des conditions et des boucles. Cependant nous n'avons pas utilisé ses fonctionnalité. Nous avons tout de même utilisé l'ajout de cibles personnalisées pour la génération du lexeur et du parseur avec les commandes **flex** et **bison**.

8.2 Les recherches

8.3 Planification des tâches

Troisième partie

Résultats et discussions

9 Fonctionnement du langage

9.1 La grammaire

9.2 Récupération des éléments du code

9.3 Gestion des erreurs

9.4 Le transpileur

10 Discussion et perspectives

Biliographie

[1] H.-P. Charles et C. Fabre, "Compilateur," *Techniques de l'ingénieur Technologies logicielles Architectures des systèmes*, t. base documentaire : TIP402WEB. N° ref. article : h3168, 2017, fre. doi : 10.51257/a-v2-h3168. eprint : basedocumentaire:TIP402WEB.. adresse : https://www.techniques-ingenieur.fr/base-documentaire/technologies-de-l-information-th9/systemes-d-exploitation-42305210/compilateur-h3168/.

[3] E. M. Gagnon et L. J. Hendren, *SableCC, an object-oriented compiler framework*. IEEE, 1998. adresse : https://central.bac-lac.gc.ca/.item?id=MQ44169&op=pdf&app=Library&oclc_number=46811936.

[4] A. A. Aaby, "Compiler construction using flex and bison," *Walla Walla College*, 2003. adresse : http://penteki.web.elte.hu/compiler.pdf.

[7] J. F. Power et B. A. Malloy, "Symbol table construction and name lookup in ISO C++," in *Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems. TOOLS-Pacific 2000*, IEEE, 2000, p. 57-68. adresse : http://mural.maynoothuniversity.ie/6456/1/JP-Symbol-table.pdf.

Webographie

[2] J. P. Joao Cangussu et V. Samanta. "Modern Compiler Implementation in Java : the MiniJava Project." (2002), adresse : https://www.cambridge.org/resources/052182060X/#java.

[5] Cpptutor. "Generating C++ programs with flex and bison." (2020), adresse : https://learnmoderncpp.com/2020/12/18/generating-c-programs-with-flex-and-bison-3/.

[6] E. M. Gagnon et L. J. Hendren. "Lexical Analysis With Flex, for Flex 2.6.2." (2016), adresse : https://westes.github.io/flex/manual/index.html#SEC_Contents.

[8] H. Singh. "Symbol Table in Compiler." (2023), adresse : https://iq.opengenus.org/symbol-table-in-compiler/.

[9] F. Tomassetti. "How to write a transpiler." (2023), adresse : https://tomassetti.me/how-to-write-a-transpiler/.

Glossaire

AST (Abstract Syntax Tree) structure sous forme d'arbre utilisée pour représenter une grammaire.. 11

bytecode code intermédiaire entre les instructions machine et le code source. Il n'est pas directment exécutable par l'ordinateur.. 6, 19

fonctions variadiques fonction qui prend un nombre indéfini de paramètres.. 9

forme de Backus-Naur métalangage utilisé pour décire les langages de programmation.. 8, 18

garbage collector (ramasse-miettes) outils inventé par John McCarthy qui permet de collecter les zones mémoires non utilisées lors de l'exécution d'un programme.. 6

lexeur Le lexeur, ou encore appelé analyseur lexical, a pour but de transformer le texte du code source en des unités lexicales, appelées *tokens*.. 14, 15, 17, 28

parseur Également appelé analyseur syntaxique, son rôle principal est la vérification de la syntaxe du code en regroupant les tokens selon une structure suivant des règles syntaxiques.. 2, 4, 7, 11, 13, 14, 16–22

A Diagramme de classes de l'AST