



Création d'un langage Transpileur

Rapport d'élève ingénieur

Projet de 2^{ème} 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

Remerciements

Nous tenons à exprimer notre profonde gratitude à :

- Notre encadrant et cher professeur M. Loïc YON pour la qualité de son encadrement, et pour nous avoir guidés durant toute la période du projet.
- Mme Murielle MOUZAT, notre professeur de communication pour son aide précieuse et indispensable pour la réussite de notre projet de 2^{ème} année.

Finalement, nous exprimons nos vifs remerciements à toute personne ayant participé de près ou de loin au bon déroulement de ce projet.

Table des matières

1	Contexte du projet	5
2	Réalisation et conception	5
2.1	Conception des éléments du langage	5
2.1.1	Définition de la grammaire	5
2.1.2	Arbre syntaxique	8
2.2	Construction de l'AST	11
2.2.1	Le lexeur	11
2.2.2	Le parseur	14
2.2.3	Fabrique à programme	17
2.3	Analyse des symboles	17
2.3.1	Gestion des symboles	18
2.3.2	Gestion des erreurs	19
2.3.3	Traitement des des symboles dans le parseur	19
2.4	Le transpileur	19
.1	Diagramme de classes de l'AST	22

Table des figures

1	Exemple d'arbre syntaxique	8
2	Diagramme de classe : Statement	9
3	Diagramme de classe : BinaryOperation	9
4	Énumération Type	10
5	Diagramme de classe : ProgramBuilder	17
6	Diagramme de classe : Symbol	18
7	Structure de la table des symboles	19
8	Diagramme de classe : Syntable	19

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. 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.

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. 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.

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

Introduction

Dans le cadre du projet de 2^{ème} 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, déléguant 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.

1 Contexte du projet

2 Réalisation et conception

2.1 Conception des éléments du langage

2.1.1 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 choisis 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 :

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

A noter qu'ici, on a une légère différence avec le C pour le nom des fonctions et des variable (**identifieur** ici). En effet, la convention du C précise qu'un nom de variable ou de fonction doit commencer avec une lettre minuscule, cependant il est tout à fait possible de commencer un nom de variable par une majuscule. Dans notre cas, les noms commencent forcément par une minuscule ce qui force l'utilisateur à respecter notre guide de style.

Les fonctions

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 retour de la fonction en utilisant **-> type** quand c'est nécessaire.

```
<function> ::= fn <identifieur> "(" <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.

```

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

<range> ::= range "<valueSymbol>, <valueSymbol>, <valueSymbol>)"
<for> ::= for <identifiant> in <range> "{"<instructions>}"

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

```

Les opérations

Comme dit précédemment, tous 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
$a + b$	<i>add(a, b)</i>
$a - b$	<i>mns(a, b)</i>
$a \times b$	<i>tms(a, b)</i>
$a \div b$	<i>div(a, b)</i>
$a == b$	<i>eql(a, b)</i>
$a < b$	<i>inf(a, b)</i>
$a > b$	<i>sup(a, b)</i>
$a \leq b$	<i>ieq(a, b)</i>
$a \geq b$	<i>seq(a, b)</i>
not a	<i>not(a)</i>

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 chaîne de formatage, ce qui aurait grandement augmenté la complexité de notre grammaire si nous avions voulu l'intégrer directement au langage.

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));  
}
```

2.1.2 Arbre syntaxique

Pour donner un sens aux éléments textuels du langage, nous utiliserons une représentation sous forme d'arbre. On appelle 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* :

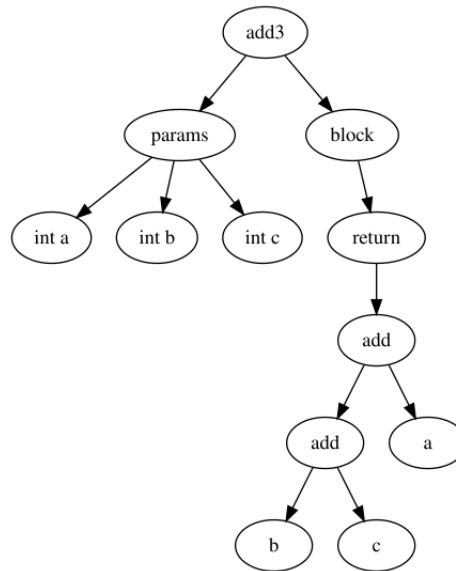


FIGURE 1 – Exemple d'arbre syntaxique

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 choisi 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 très 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 ait un type concret utilisable dans le transpileur. Le diagramme de classes complet est disponible en annexe .1. Détaillons maintenant les parties importantes.

Les blocs de code

La classe Block correspond aux blocs de code entre accolades. Elle possède une liste de nœuds qui sont les opérations contenues dans le bloc.

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 blocs ailleurs.

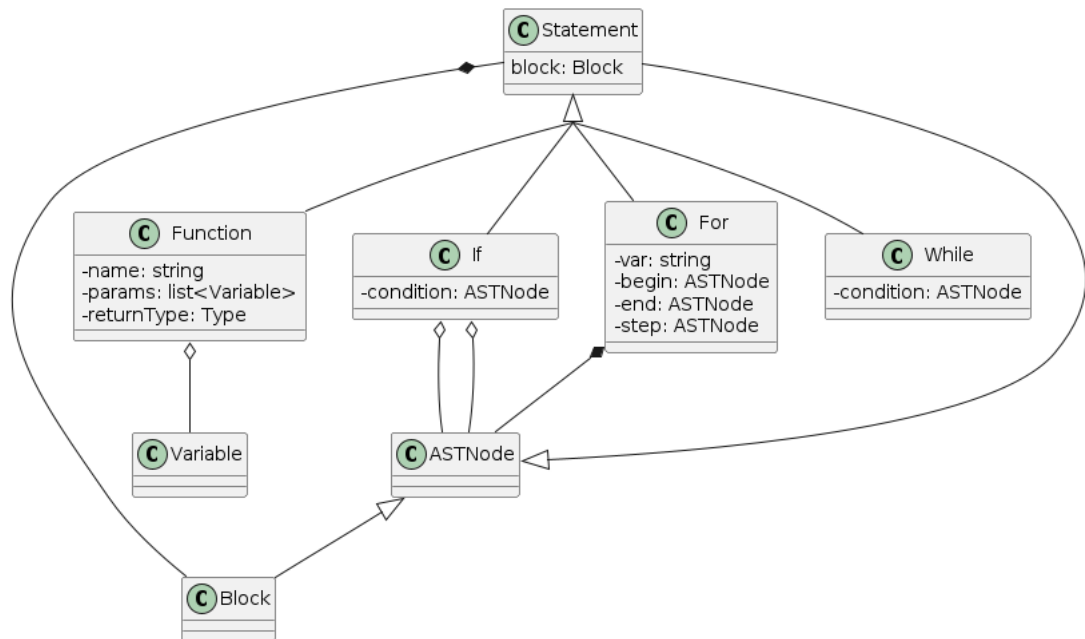


FIGURE 2 – Diagramme de classe : Statement

Les opérations

Les opérations sont traitées avec la classe **OperationBinaire** qui correspond à tous les opérateurs.

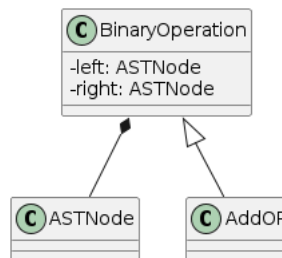


FIGURE 3 – Diagramme de classe : BinaryOperation

Les éléments typé

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

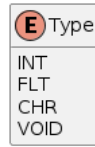


FIGURE 4 – Enumération Type

Les valeurs et les variable possède un simple type, et les fonctions possède une liste de **Type** qui contient le type des paramètres suivit du type de retour de la fonction. À noter que pour stocker une valeur, on utilise une union composé d'un entier, d'un double et d'un char. De ce fait, **Value** peut contenir n'importe quel type de valeur et le champ à utiliser est déterminé par le type de la valeur.

2.2 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].

2.2.1 Le lexeur

Définition

Le lexeur, ou encore appelé analyseur lexical, a pour but de transformer le texte du code source en des unités lexicales, appelées *tokens* [6].

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

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 utilisées :

- **c++** : indique qu'on travaille avec du c++ et non du c
- **interactive** : utile quand on utilise **std::in**. Le scanner interactif regarde plus de caractères avant de générer un token (plus lent mais permet de lutter contre les ambiguïtés)
- **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 définir 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 défini 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 correspondent à une suite de chiffres et peuvent être positifs ou négatifs.
- **float** : similaire aux entiers sauf qu'ici on a obligatoirement un point suivi d'une suite de chiffres à la fin.
- **char** : les caractères sont toujours écrits 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 suivi d'une suite de lettres, de nombres et de _.

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

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 met du code entre accolades, et ce code sera exécuté quand le lexique consommera la chaîne. Par exemple, si on prend la première ligne ci-dessous, quand le lexique 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 fichier 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 [7].

```
#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.

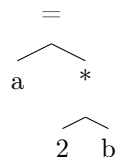

2.2.2 Le parseur

Définition

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.

Exemple

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

Arbre syntaxique	Évaluation de 2 * b	Affectation de a
		<code>a = 2 * b</code>

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.

L'article [4] s'est encore une fois révélé très utile pour la réalisation du fichier nécessaire à Bison

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 blocs de code qui seront exécutés au moment où le parseur la reconnaît, ce code permet de construire l'ABS (Abstract Syntaxic Tree) du programme ainsi que de faire de la vérification sur les symboles (définition et type).

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
```


À noter que l'on peut spécifier le type de l'élément, ce qui sera utile pour récupérer les valeurs retournées par le lexeur.

Bison permet de construire le parser, qui va reconnaître des éléments de syntaxe et non pas juste des mots clés. Par exemple, on peut définir une règle pour reconnaître une suite d'inclusion de fichiers :

```
includes: %empty
        | INCLUDE IDENTIFIER SEMI includes
        ;
```

Ici, on définit une règle **includes** qui décrit la syntaxe des *includes*. Selon cette règle, une suite d'inclusions est soit vide, soit elle comporte une inclusion, suivit d'une suite d'inclusion ('|' signifie "ou"). Il faut noter que la syntaxe est "récursive", ce qui nous permet de définir une suite d'éléments. Enfin, les mots en majuscule sont les tokens retournés par le lexeur.

On peut ajouter des blocks de codes qui seront exécutés au moment où le parseur atteint l'élément qui précède le block. Dans l'exemple ci-dessous, le block sera appelé une fois que Bison aura parser le `;`. À noter que l'on peut accéder aux éléments retournés par Flex ; ici, `$2` fait référence au second élément de la règle qui est **IDENTIFIER**. Le type de **IDENTIFIER** a été défini comme étant une `std::string`. Le block de code nous permet donc de créer une nouvelle inclusion et de récupérer le nom de la bibliothèque.

```
includes: %empty
        |
        INCLUDE IDENTIFIER SEMI
        {
            std::cout << "new include id: " << $2 << std::endl;
            pb.addInclude(std::make_shared<Include>($2));
        }
        includes
        ;
```

Comme dit plus haut, on peut définir des types pour les tokens, ce qui permet de récupérer des valeurs :

```
value: INT {
    std::cout << "new int: " << $1 << std::endl;
    lastValue.i = $1;
    lastValueType = INT;
} | FLOAT {
    std::cout << "new double: " << $1 << std::endl;
    lastValue.f = $1;
    lastValueType = FLT;
} | CHAR {
    std::cout << "new char: " << $1 << std::endl;
    lastValue.c = $1;
    lastValueType = CHR;
}
;
```

Pour la génération du code, on a deux options :

- utiliser des instructions très simples => sorte de 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.

2.2.3 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 commandes 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 :

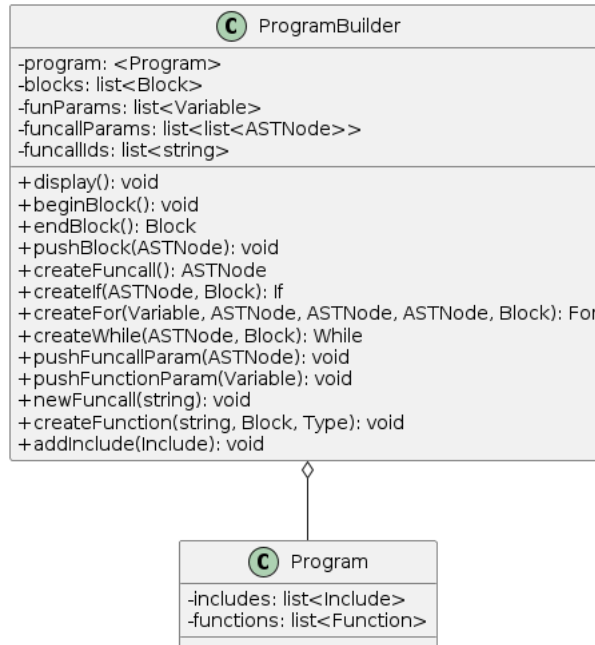


FIGURE 5 – Diagramme de classe : ProgramBuilder

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 composée 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 connaît 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 blocs 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 où **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 récupérés 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.

2.3 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 fonction prend les bons 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 le tout code source traité. À noter que la présence d'erreurs empêche la compilation.

2.3.1 Gestion des symboles

La table des symboles

Pour la création de notre table des symboles nous utiliserons la méthode présentée dans cet article [8] que nous avons simplifiée 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 suit le même principe que dans [8]. 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 typés 2.1.2.

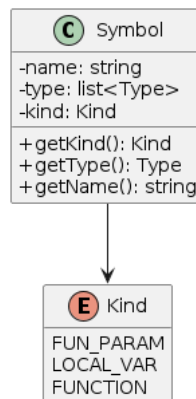


FIGURE 6 – Diagramme de classe : Symbol

Pour implémenter la table des symboles nous nous sommes inspiré du schéma suivant [9]. Ici la table des symboles est une table de tables, et 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. En effet, les éléments stockés dans la table mère sont toujours accessibles.

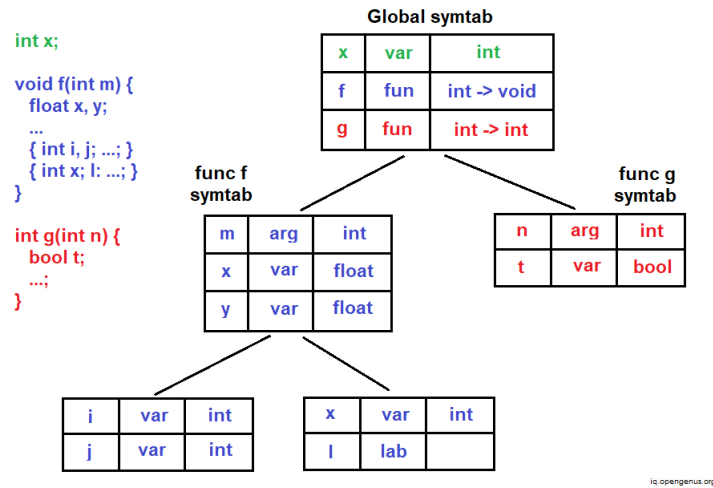


FIGURE 7 – Structure de la table des symboles

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 de récupérer les symboles avec leurs identifiants. De plus, la table possède une liste de sous tables qui est un pointeur sur la table mère qui sera utilisé par la méthode **lookup** pour la recherche de symbole comme expliqué précédemment.

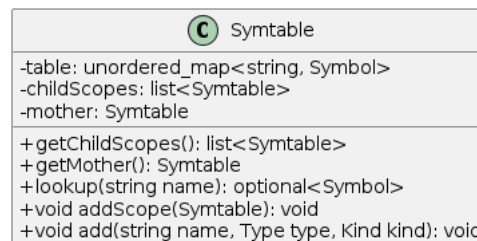


FIGURE 8 – Diagramme de classe : Symtable

Gestion du contexte

2.3.2 Gestion des erreurs

2.3.3 Traitement des des symboles dans le parseur

Vérification des définitions

Vérification des types

2.4 Le transpileur

Bibliographie

- [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>.
- [6] J. LEVINE, *Flex & Bison : Text Processing Tools*. " O'Reilly Media, Inc.", 2009. adresse : https://books.google.fr/books?hl=fr&lr=&id=nYUkAAAAQBAJ&oi=fnd&pg=PR3&dq=flex+bison+interpreter&ots=VX9xrg4D9l&sig=p95S6sNhMxdIIl-7u00nK_1u-fM&redir_esc=y#v=onepage&q=flex%20bison%20interpreter&f=false.
- [8] 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/>.
- [7] 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.
- [9] H. SINGH. “Symbol Table in Compiler.” (2023), adresse : <https://iq.opengenus.org/symbol-table-in-compiler/>.

Glossaire

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

8

fonctions variadiques fonction qui prend un nombre indéfini de paramètres.. 6

forme de Backus-Naur métalangage utilisé pour décrire les langages de programmation.. 5

.1 Diagramme de classes de l'AST

