



# Création d'un langage Transpileur

Rapport d'élève ingénieur

Projet de 2<sup>ème</sup> 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<sup>ème</sup> 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

<b>Résumé</b>	<b>2</b>
<b>Abstract</b>	<b>2</b>
<b>Introduction</b>	<b>3</b>
<b>Résumé des références</b>	<b>4</b>
<b>1 Conception d'une grammaire</b>	<b>5</b>
Définition de la grammaire . . . . .	5
Les variables . . . . .	5
Les fonctions . . . . .	5
Les structures de contrôle . . . . .	5
Les opérations . . . . .	5
Inclusion de fichier . . . . .	6
Entrée et sortie . . . . .	6
Exemple de code . . . . .	6
Arbre syntaxique . . . . .	7
Les blocs de code . . . . .	7
Les conteneurs d'opérations . . . . .	7
Les opérations . . . . .	7
<b>2 Construction de l'AST</b>	<b>8</b>
Flex . . . . .	8
Définition . . . . .	8
Implémentation . . . . .	9
Bison . . . . .	11
Définition . . . . .	11
Implémentation . . . . .	11
Fabrique à programme . . . . .	14
<b>3 Analyse des symboles</b>	<b>14</b>
Création d'une table des symboles . . . . .	14
Traitement des des symboles dans le parseur . . . . .	14
<b>4 Le transpileur</b>	<b>14</b>
<b>5 Conception d'un langage informatique</b>	<b>14</b>
<b>6 Le parseur</b>	<b>14</b>
<b>A Diagramme de classes de l'AST</b>	<b>16</b>

# Résumé

L'objectif de ce projet est la création d'un langage **transpilé** en **C++** dans le but de faire découvrir l'informatique et la programmation à des collégiens et lycéens. Ce projet rentre dans le cadre du sujet commun de la filière F2, qui concerne la conception d'outils et plus généralement en développement logiciel.

Ce projet s'inscrit dans le cadre du projet commun de la filière 2. L'objectif du projet commun est la création d'outils de démonstration servant à présenter la filière génie logiciel. Notre travail a consisté en la création d'un langage de programmation transpilé. Pour ce faire, nous avons dû étudier des concepts et des outils utilisés pour la création de compilateurs.

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

# Abstract

The objective of this project is the creation of an **transpiled** language in **C++** in order to introduce computer science and programming to middle and high school students. This project falls within the framework of the common subject of the F2 major, which relates to the design of tools and more generally in software development.

Keywords : **C++**, **interpreter**, **lexer**, **parser**, **programming language**

# Introduction

Dans le cadre du projet de 2<sup>ème</sup> 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.

## Résumé des références

- La partie consultable de [1] présentent les bases de flex.
- [2] présente la construction d'un compilateur avec flex et bison. Le compilateur présenté utilise une **table des symboles** ainsi qu'une sorte de **byte code**. Nous avons choisi l'autre méthode qui consiste à utiliser un object-ABS plutôt que directement du byte code. Article très utilisé au départ pour la mise en place du parseur/lexeur.
- [3] : manuel d'utilisation de Flex.
- [4] : nous a permis d'avoir un exemple de code qui allie flex et bison en C++ et non en C.
- [5] explication du fonctionnement d'un compilateur.
- [6] première version de l'article précédent.
- [7] création d'un analyseur syntaxique pour du C/C++ : ASTROLOG. L'article par d'analyse syntaxique et de la construction d'**ABS**.
- [8] L'objectif de l'article est de présenter l'utilisation des **ABS** pour de la méta programmation. Il comporte pas mal d'exemples sur les **ABS** donc je le trouve pertinent.
- [9] : **ABS** en java.

# 1 Conception d'une grammaire

## Définition de la grammaire

Avant d'implémenter notre langage informatique, 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 choisi 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> ";"
```

### Les fonctions

Pour définir des fonctions, nous utilisons le mot clé **fn**, suivi 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 <identifieur> 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
$\mathbf{a} + \mathbf{b}$	<i>add(a, b)</i>
$\mathbf{a} - \mathbf{b}$	<i>mns(a, b)</i>
$\mathbf{a} \times \mathbf{b}$	<i>tms(a, b)</i>
$\mathbf{a} \div \mathbf{b}$	<i>div(a, b)</i>
$\mathbf{a} == \mathbf{b}$	<i>eql(a, b)</i>
$\mathbf{a} < \mathbf{b}$	<i>inf(a, b)</i>
$\mathbf{a} > \mathbf{b}$	<i>sup(a, b)</i>
$\mathbf{a} \leq \mathbf{b}$	<i>ieq(a, b)</i>
$\mathbf{a} \geq \mathbf{b}$	<i>seq(a, b)</i>
<b>not a</b>	<i>not(a)</i>

## Inclusion de fichier

TODO : Non fonctionnel :/

## 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 (depuis une variable ou une fonction).

## 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));
}
```



## Arbre syntaxique

Pour donner un sens aux éléments textuels du langage, nous utiliseront une représentation sous forme d'arbre. On appelle cela un arbre syntaxique ou AST (Abstract Syntax Tree). Les arbres syntaxiques sont nés avec la théorie des langages et comme on peut le voir dans cet article [5], 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* :

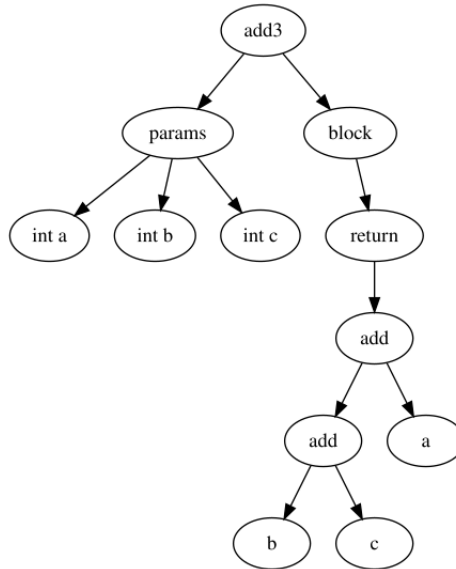


FIGURE 1 – Exemple d'arbre syntaxique

Pour cette partie, nous nous sommes inspiré d'un projet appelé **minijava** [10] ainsi que d'un article [9] 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 A. 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 classe Statement correspond aux éléments qui contiennent des blocs de code. Cela comprend les fonctions et les structures de contrôles. La grammaire ne permet pas l'emploi des blocs ailleurs.

### Les opérations

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

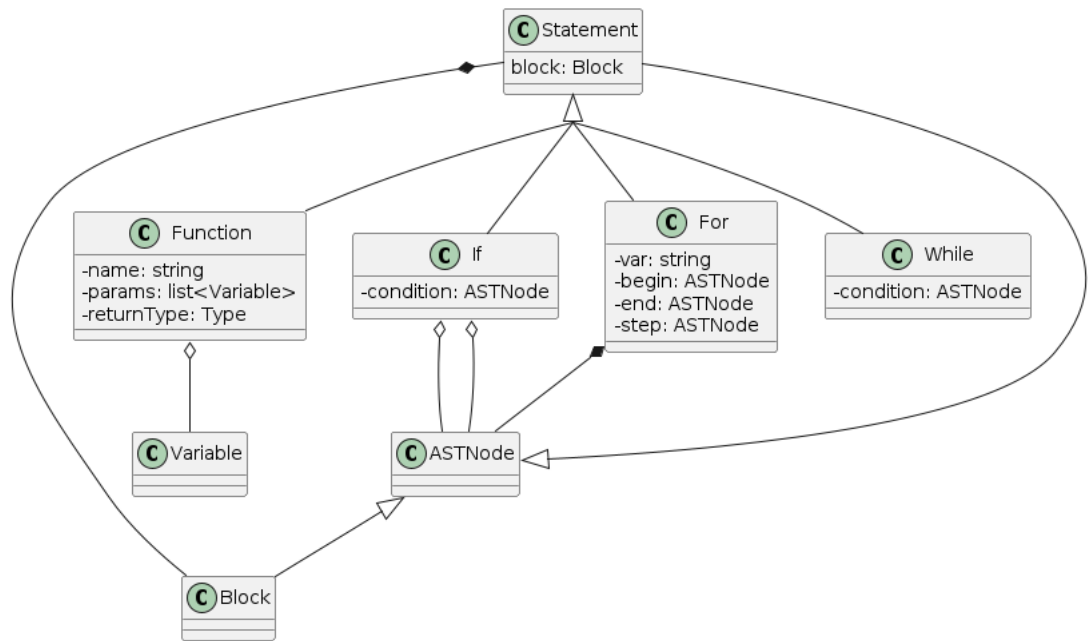


FIGURE 2 – Classe Statement

## 2 Construction de l'AST

### Flex

#### Définition

Le lecteur, ou encore appelé analyseur lexical, a pour but de transformer le texte du code source en des unités lexicales, appelées *tokens*, comme expliqué par la partie consultable de [1].

#### 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 lecteur a également pour rôle de supprimer les informations inutiles, généralement du caractère espace et des commentaires.

## Implémentation

L'outil utilisé pour générer un liseur à partir du code précédent est Flex (Fast LEXical analyser generator). Au lieu d'écrire un liseur à partir de zéro, Flex permet d'avoir un liseur en donnant seulement les modèles des expressions régulières ainsi que le langage de travail (c++ dans notre cas).

[2] a fourni un squelette pour la réalisation du fichier nécessaire à Flex.

Le fichier **main\_cpp.l** contient le code qui permet de générer le liseur avec Flex. Les token doivent être définis dans **main\_cpp.y** au préalable. À noter que l'on peut utiliser la variable `yyval` pour transmettre des éléments au parser.

La structure du fichier **main\_cpp.l** est la suivante :

```
C and parser declaration

%%
Grammar rules and actions

%%
C subroutings
```

On peut définir des règles dans les déclarations du liseur :

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

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

Concernant la ligne d'option :

- **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
- **noyylineno** : désactive l'enregistrement des lignes (**yylineno**)
- **nodefault** : pas de scanner par défaut (=> on doit tout implémenter)
- **outfile : "file.cpp"** : permet de définir le fichier de sortie

Dans les règles, on suit toujours le même principe, on indique les caractères à reconnaître puis on exécute du code :

```

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

```

Ici on a accès à la variable `yylval` qui est un `Parser::semantic_type*` et qui possède une méthode `build` qui nous permet de transmettre des valeurs à bison.

Ces valeurs sont accessible via les variables de bison : `$2`. La variable `yytext` contient le text traité par Flex. De plus, dans le code, on retourne les *tokens*. Ces **tokens** sont définis dans le fichier de 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 [3].

```

#define YY_DECL int interpreter::Scanner::lex(interpreter::Parser::semantic_type *yylval)

```

# Bison

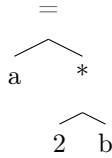

## Définition

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
		a = 2 * b

## Implémentation

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

L'article [2] 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 blocks de code qui seront exécutés au moment où le parseur la reconnaît, ce code permet de créer des objets qui formeront l'ABS (Abstract Syntactic Tree) du programme.

La structure du fichier **main\_cpp.y** est identique au lecteur :

```
C and parser declaration

%%
Grammar rules and actions

%%
C subroutines
```

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 lecteur.

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, suivie 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 parsé 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.

Fabrique à programme

### 3 Analyse des symboles

Création d'une table des symboles

Traitement des des symboles dans le parseur

### 4 Le transpileur

### 5 Conception d'un langage informatique

### 6 Le parseur

## Bibliographie

- [1] 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](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).
- [2] A. A. AABY, "Compiler construction using flex and bison," *Walla Walla College*, 2003. adresse : <http://penteki.web.elte.hu/compiler.pdf>.
- [5] 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/>.
- [6] B. LORHO, "Compilateurs," *Techniques de l'ingénieur Technologies logicielles Architectures des systèmes*, t. base documentaire : TIP402WEB. N° ref. article : h3168, 1996, 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/>.
- [7] R. F. CREW et al., "ASTLOG : A Language for Examining Abstract Syntax Trees.," t. 97, p. 18-18, 1997. adresse : [https://www.usenix.org/legacy/publications/library/proceedings/ds197/full\\_papers/crew/crew.pdf](https://www.usenix.org/legacy/publications/library/proceedings/ds197/full_papers/crew/crew.pdf).
- [8] E. VISSER, "Meta-programming with concrete object syntax," p. 299-315, 2002. adresse : [https://dspace.library.uu.nl/bitstream/handle/1874/23952/visser\\_02\\_metaprogramming.pdf?sequence=2](https://dspace.library.uu.nl/bitstream/handle/1874/23952/visser_02_metaprogramming.pdf?sequence=2).
- [9] 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](https://central.bac-lac.gc.ca/.item?id=MQ44169&op=pdf&app=Library&oclc_number=46811936).



## Webographie

- [3] 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](https://westes.github.io/flex/manual/index.html#SEC_Contents).
- [4] CPPTUTOR. “Generating C++ programs with flex and bison.” (2020), adresse : <https://learnmoderncpp.com/2020/12/18/generating-c-programs-with-flex-and-bison-3/>.
- [10] J. P. JOAO CANGUSSU et V. SAMANTA. “Modern Compiler Implementation in Java : the MiniJava Project.” (2002), adresse : <https://www.cambridge.org/resources/052182060X/#java>.

## A Diagramme de classes de l'AST

