## **ARCHITECTURE DU LOGICIEL**

*Version :* 0.1

*Date :* **20/12/2014**

*Rédigé par :***Majid Tasserie, Nasser Adjibi**

*Relu par :***Majid Tasserie,**

*Approuvé par :* **N.A**

# MISES A JOUR

|  |  |  |
| --- | --- | --- |
| **Version** | **Date** | **Modifications réalisées** |
| 0.1 | 16/12/05 | Création |
|  |  |  |
|  |  |  |
|  |  |  |

**SOMMAIRE  
 Mise à jour………………….2  
1. Objet…………………………4  
2. Document applicables et références……………4  
3. Terminologie utilisée…………….5  
4. Configuration utilisateur requise……………………………………………  
 4.1 .Système d’exploitation de la machine………………………………   
 4.2 .Dépendance du compilateur………………………………………  
5. Architecture statique…………………….  
 5.1. Structure générale d’un compilateur…………………………   
 5.2 Structure détaillé du Front End………………………………  
 Structure de FLEX  
 Structure du fichier générer par FLEX  
 Structure de Bison  
 Structure du fichier générer par BISON  
 Structure du code Intermédiaire………………………………………  
 5.3. Structure de l’arbre syntaxique…………………………………………  
 5.3.a Schéma de l’arbre syntaxique…………………………………...  
 5.3.b Enumération des élément de l’arbre syntaxique…………....**

**6. Fonctionnement dynamique…………………………………………………….  
 6.1 Enchainement dynamique front-end et de l’optimisateur du code intermédiaire…  
 6.2 Enchainement dynamique du Back End…………………………………..**

## 1 Objet

Ce document décrit les différentes solutions d'architecture logicielle répondant aux exigences de notre client, M. Nicart, que nous avons retenues. Le besoin formulé étant la création d'un compilateur capable de créer des exécutables à partir de fichiers sources écrit dans un nouveau langage appelé « Kawa ».

## 2 Documents applicables et de références

1 – Document applicables

STB : Spécification Technique des Besoins

Cahier des Charges.

2 – Références complémentaires au document

Documentation LLMV : <http://llvm.org/docs/LangRef.html>

## 3 Terminologie utilisée

**Compilateur :** Un compilateur est un programme informatique visant à traduire du code écrit dans un langage source en un code écrit dans un langage cible. Dans notre cas nous cherchons à traduire du code en Kawa écrit par l'utilisateur (langage source) en du code compilé exécutable par la machine (langage cible).

**LLVM :** LLVM est un projet issu d'un projet de recherche qui regroupe une collection de compilateurs modulables et réutilisables ainsi qu'une série d'outil. C'est l'outil qui permettra d'assurer la compatibilité de notre compilateur avec toutes les machines.

**Analyse lexicale :** L’analyse lexicale est la phase qui sera responsable du découpage du code source en entités lexicales (lexèmes). Cette phase permet de détecter l’ensemble des mots clés, des identificateurs, des nombres et la fin du fichier.

**Analyse syntaxique :** L'analyse syntaxique est la deuxième étape d'un compilateur. Elle réutilise les lexèmes définis par l'analyse lexicale et défini une hiérarchie qui sera représenté par un arbre syntaxique.

**Analyse sémantique :** L'analyse sémantique est la troisième étape. Elle reprend l'arbre syntaxique et en déduit une signification. Elle sera responsable de la vérification et la validation des exigences fonctionnelles et de réalisations stipulés dans et document des spécifications techniques de besoins .

**Code intermédiaire :** Le code intermédiaire est un code avec un structure linéaire découlant de l'analyse sémantique. Ce code a pour but de préparer la sélection d'instruction sans encore tenir compte de l'architecture de la machine cible.

**Flex :** Flex est la version libre de Lex. Un outil permettant de réaliser un analyseur lexical.

**Bison :** Bison est la version libre de Yacc. Un outil permettant de réaliser un analyseur syntaxique. Souvent couplé avec l'utilisation de Flex.

**Pass :** Les pass sont une composante importante d'un compilateur LLVM. Ils sont chargés d'effectuer les transformations et les optimisations de code du compilateur et de créer les analyses nécessaires en amont et servent également de méthode de structure du code source du compilateur.

**llc :** Commande LLVM permettant de compiler le fichier source LLVM donné en langage assembleur pour la machine cible.

**lld :** Commande LLVM pour la gestion de liens.

**Opt** : Commande d’optimisation de code IR.

**KAWAC :** Kawa Compilateur, le nom de notre compilateur kawa.

**Chaîne de compilation :** une **chaîne de compilation** (en anglais : « toolchain ») désigne l'ensemble des paquets utilisés dans le processus de compilation d'un programme pour un processeur donné. Le compilateur n'est qu'un élément de cette chaîne, laquelle varie selon l'architecture matérielle cible

## 4 Configuration utilisateur requise

### 4.1 Système d’exploitation de la machine.

La compatibilité du compilateur est assuré sur tout type de machine tournant sur le système d'exploitation Ubuntu 14.04 ou Ubuntu 14.10. La configuration matériel de l'utilisateur ne devant qu'impacter les performances et les temps d’exécution du compilateur.

### 4.2 Dépendances du compilateur.

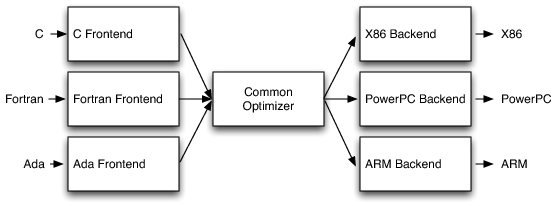
La machine doit être dotée des éléments suivant issus de la chaîne de compilation C de LLVM :

* llc : compilateur statique de LLVM
* lld : éditeur de lien de LLVM
* clang3.4
* Bison : logiciel de génération d’analyseur syntaxique
* Flex : logiciel de génération d’analyse lexicale
* Bibliothèques LLVM

## 5. Architecture statique

Cette partie du document a pour but décrire de manière générale l’architecture du logiciel. Elle servira à orienter l’équipe de développement lors de la réalisation des différentes parties du logiciel.

### 5.1 Architecture générale d’une chaîne de compilation avec l’architecture LLVM



### 5.2 Structure détaillée du front-end

La partie du front-end se compose de l'analyse lexicale du code utilisateur, son analyse syntaxique et sémantique et de la génération de code intermédiaire. Elle s’opéra par l’utilisation couplée de Flex, de Bison et des bibliothèques de générations de code Intermédiaires fournies par LLVM.

L'analyseur lexical sera réalisé grâce à Flex. Flex permet de créer, à partir d'un fichier Flex (d'extension .l par convention), un code contenant la description du texte analysé.

#### Structure d'un fichier Flex :

%{

Pré-code

%}

Définitions et options

%%

Règles de production

%%

Post-code

Les pré-code et post-code contiennent le code (en C ou C++) qui sera recopié tel quel dans le code de sortie. Par exemple on peut y écrire l'inclusion de code externe et les définitions de méthodes qui seront utilisé par le code de sortie.

Les options permettent de paramétrer l'analyseur.

Elles doivent être écrites de telles manières : %option1

%option2

Les définitions sont les définitions des expressions rationnelles. Par exemple si je veux définir l'expression « chiffre » décrivant les chiffres de 0 à 9, je devrais écrire : chiffre [0-9]

Les expressions définies devront être utilisées dans les règles de production. Ces règles associent l’exécution d'un code à la rencontre d'une ou plusieurs expressions rationnelles.

Elles se présentent sous cette forme : Regex Code

Par exemple si je veux afficher un message quand je rencontre un chiffre je devrais écrire : (chiffre) printf(« Un chiffre ! ») ;

En prévision de la prochaine analyse je dois également retourner le type d’élément rencontré. Ici c'est un chiffre donc cela donnera quelques chose du type :

(chiffre) { printf(« Un chiffre ! ») ; return CHIFFRE ; }

#### Structure du fichier généré par Flex:

**Déclarations et macros**

**Table d'analyse**

**Pré-code C**

**int yylex(void){**

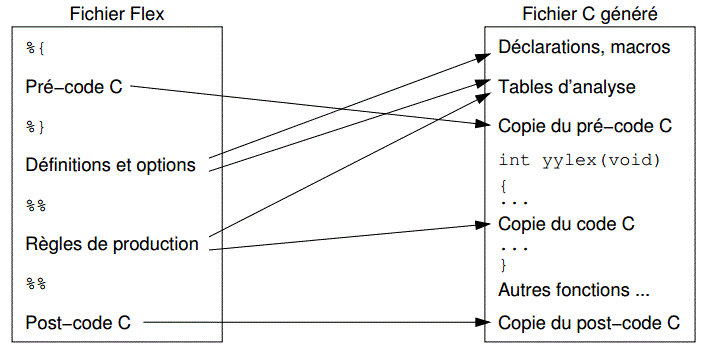
**Code C**

**}**

**Fonctions diverses**

**Post-code C**

Ici ce qui nous intéresse principalement est la fonction yylex(void) et son contenu. C'est la fonction qui réalise l'analyse lexicale et donc celle qui sera appelée par la suite. Cette fonction extrait et lit des caractères sur un flux yyin (à définir plus tard) et confronte les chaînes de caractères obtenues avec les règles définies dans le fichier Flex avant d’exécuter le code associé.

En résumé en utilisant le langage C

L'analyse syntaxique va ensuite être effectuée par Bison (la version libre de Yacc) qui va réutiliser les lexèmes obtenus grâces aux appels à la fonction yylex(void) définie par l'analyse lexicale. Bison va créer, à partir d'un fichier Bison (.y par convention), le code source d'un parseur.

#### Structure d'un fichier Bison :

*%{*

*Pré-code*

*%}*

*Définitions et options*

*%%*

*Règles de production*

*%%*

*Post-code*

Les définitions sont les définitions de lexèmes, de priorités, de types, etc.

Elles sont de la forme : %typeDef nom

Par exemple pour définir le lexème « CHIFFRE » et un nouveau non-terminal « nombre » de type int: %union {int nb}

%token CHIFFRE

%type <nb> nombre

Les règles de production définissent la grammaire de l'analyseur. Nous allons suivre la grammaire définie dans l'Annexe Spécification Techniques des Besoin. Les règles sont tout simplement écrites de la manière suivante :

non-terminal : Nom donné au non-terminal défini par la règle, doit être déclaré dans les définitions.

séquencen: séquence de symboles (terminaux ou non terminaux) susceptible de réduire le non-terminal.

Code: Le code à exécuter lors de la réduction de la règle.

Par exemple avec notre exemple de « CHIFFRE » et « nombre » :

*nbre : nombre CHFFRE {}*

*| nombre {} ;*

Cette règle nous permet de reconnaître un nombre comme une suite de chiffre.

#### Structure du code généré par Bison :

Déclarations et macros

Pré-code

Table d'analyse

int yyparse(void){

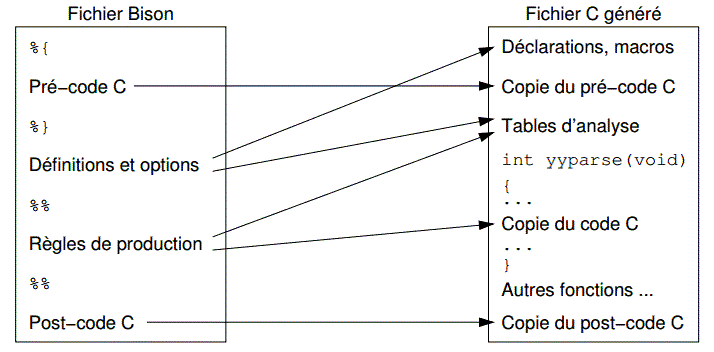
Code

}

Fonctions diverses

Post-code

La fonction **yyparse(void)** définie l'analyse sémantique. Elle consomme les lexèmes obtenus par l'analyse lexicale via des appels à la fonction **yylex(void)** et vérifie que les lexèmes permettent de réduire l'axiome de la grammaire. Elle exécute ensuite les actions sémantiques contenues dans le code associé aux règles réduites et peut signaler des erreurs si une fonction **yyerror()** est définie dans le post-code.



En résumé en utilisant le langage C

### Structure du code intermédiaire

### Les étapes précédentes résulteront à la production d’un arbre syntaxique qui permettra de générer une représentation en LLVM IR du code de la classe KAWAC en cours de compilation. Le code IR sera principalement stockés grâce aux types llvm ::Function et llvm::Value. Il permettront de stocker les opérations, les valeur, les appels et définitions de méthode et définitions de structures. LLVM fournit un large nombre de méthode pour la production de code intermédiare. L’ensembles des defnitions de fonctions et de variables en LLVM IR sera assuré grâce à l’objet à l’objet LLVM ::Module qui permettra de stocker en mémoire ou sur le disc, la representation du code KAWA en cours de compilation. Exemeple de code IR dans un Module

### 

|  |
| --- |
| ; Declaration d’une chaine de caractère globale  @.str = private unnamed\_addr constant [13 x i8] c"hello world\0A\00"  ; Declaration d’une structure LLVM  %mytype = type { %mytype\*, i32 }  ; Declaration d’une function externe  declare i32 @puts(i8\* nocapture) nounwind  ; Declaration d’une function main  define i32 @main() { ; i32()\*  ; Convert [13 x i8]\* to i8 \*...  %cast210 = getelementptr [13 x i8]\* @.str, i64 0, i64 0  ; Call puts function to write out the string to stdout.  call i32 @puts(i8\* %cast210)  ret i32 0  }  ; Metadata  !0 = !{i32 42, null, !"string"}  !foo = !{!0} |

Ce code sera ensuite écrit sur le disc dans un fichier .ll et ce dernier passé à l’optimiseur de code.

**5.3 Structure de l’arbre syntaxique.**

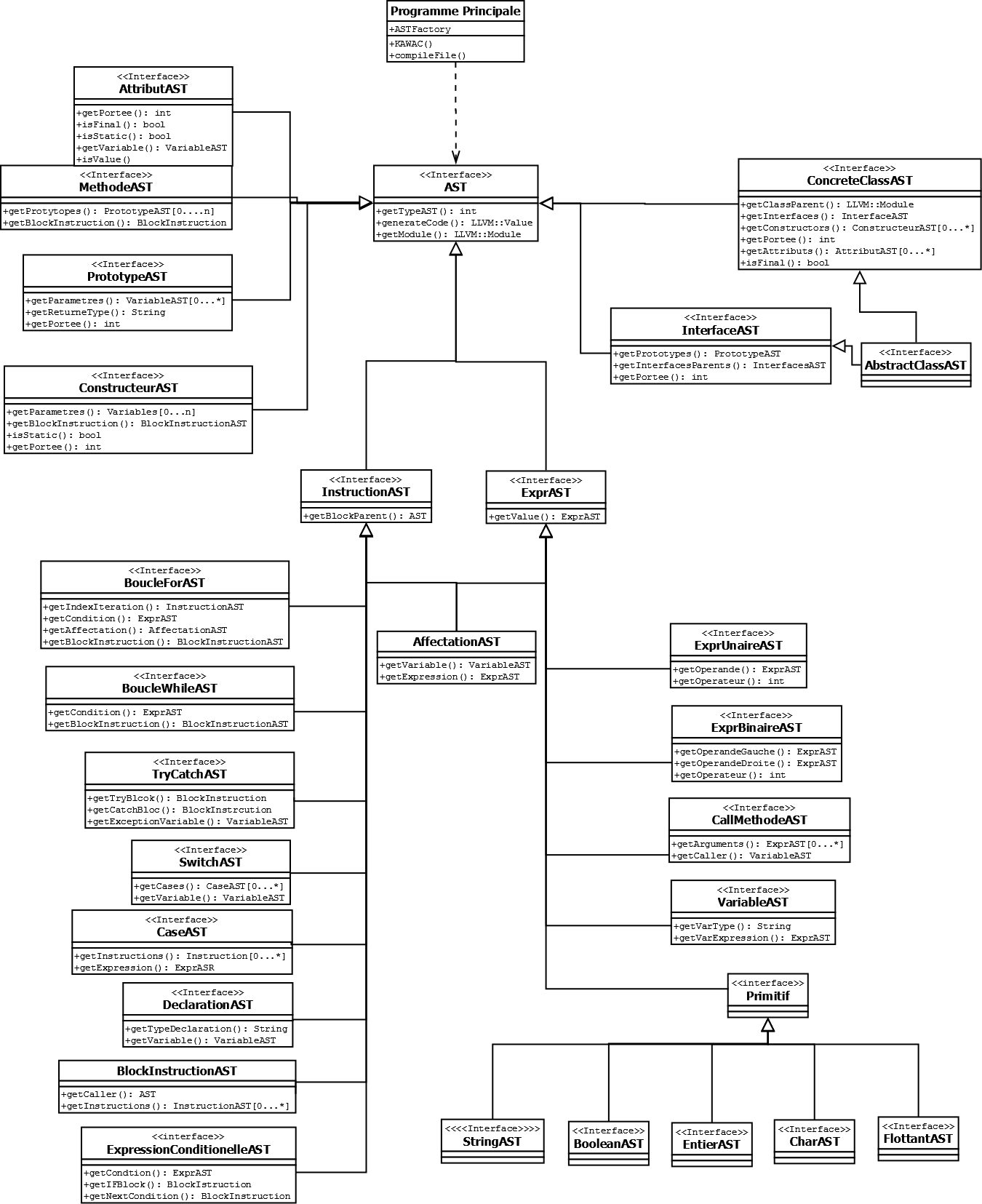
Le schéma suivant décrit la structure de l’arborescence de l’arbre syntaxique généré par l’analyseur syntaxique et lexical.e 

Figure -Arborescence Arbre syntaxique

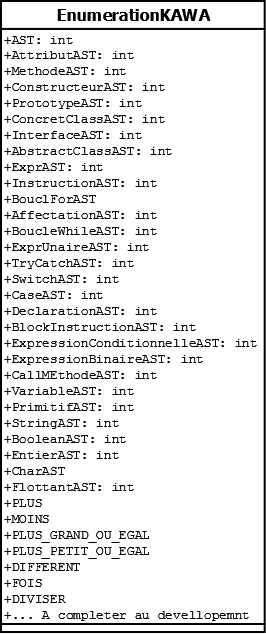


Figure -Enumeration KAWA

**5.3.2 Enumérations des éléments de l’arbre**

<<Interface>> AST

|  |  |
| --- | --- |
| Rôle | L'interface implémentée par tous les nœuds et feuilles de l'arbre syntaxique. Elle regroupe toute les fonctionnalités de base d’un nœud |
| Règle de création |  |
| Méthodes | getModule : retourne le module associé a l’arbre syntaxique |
| getType : retourne le type du nœud |
| generateCode : retourne l’objet associé a la représentation en code IR du nœud |

<<Interface>> MethodeAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant la définition d’une méthode Kawa |
| Règle de création | DECLARATION\_METHODE |
| Methodes | getBlockInstruction : retourne le bloc d’instruction associé à la méthode |
| getPrototype : retourne le prototype de la méthode |

<<Interface>> PrototypeAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les prototypes des méthodes kawa. |
| Regle de création | *PROTOTYPE\_METHODE |PROTOTYPE\_CONTRUCTEUR* |
| Méthodes | getParametres : Retourne les paramètres d’une d’un prototype |
| getReturnType : retourne le type que retourne la fonction |
| getPortee : retourne l’énumération de la portée |
| isFinal : teste si la methode est finale |
| iStatic : teste si la methode est statique |

<<Interface>> ConstructeurAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les constructeurs kawa |
| Règle de création | DECLARATION\_CONSTRUCTEUR |
| Méthodes | getParametres : retourne les paramètres du constructeur |
| getBlockInstruction : retourne le bloc d’instruction associé constructeur |
| isStatic : teste si le constructeur est static |
| getPortee : retourne l’énumération de la portée |

<<Interface>> AttributAST

|  |  |
| --- | --- |
| Role | Interface des éléments représentant les attributs |
| Règle de création | DECLARATION\_ATTRIBUT |
| Méthodes | getPortee : retourne la portée d’une variable |
| isFinal : determine si un attribut est final |
| isStatic : determine si l’attribut est static |
| getVariable : retourne la variable associé |

<<Interface>> InstructionAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les instructions kawa |
| Règle de création |  |
| Méthodes | getBlockParent : retourne le bloc auquel appartient l'instruction |

<<interface>> BoucleForAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les boucles For |
| Règle de création | EXPRESSION\_FOR |
| Méthodes | getCondition : retourne la condition associée a la boucle |
| getIndexIteration  : retourne la déclation /affectation initialisant l’itérateur de boucle |
| getItération : retourne l’itération à effectuer a chaque tour de boucle |

<<interface>> BoucleWhileAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les boucles while kawa |
| Règle de création | EXPRESSION\_WHILE |
| Méthodes | getCondition : retourne la condition du **while** |
| getBlockInstruction : retourne la bloc d’instruction du bloc **while** |

<<interface>> TryCatchAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les try catch kawa |
| Règle de création | EXPRESSION\_TRY |
| Méthodes | getTryBlock : retourne le bloc d’instruction du bloc **try** |
| getCatchBlock : retourne le bloc d’instruction du bloc **catch** |
| getExceptionVariable : retourne la variable de l’exception |

<<Interface>> SwitchAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les constructeurs kawa |
| Règle de création | EXPRESSION\_SWITCH |
| Méthodes | getVariable : retourne la variable associée au bloc **switch** |
| getCases : retourne les cas des du **switch** |

<<Interface>> CaseAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les cases au sein des switch kawa |
| Règle de création | SWICTH\_CASE\_BLOC | SWITCH\_DEFAULT\_BLOC |
| Méthodes | getCondition : retourne la condition du Switch |
| getInstructions : retourne la suite d’instructions à réaliser pour le cas |

<<Interface>> ExpressionConditionnelleAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant une expression conditionnelle en kawa |
| Règle de création | EXPRESSION\_CONDITIONNELLE | EXPRESSION\_CONDITIONNELLE\_SECONDAIRE | EXPRESSION\_TERNAIRE |
| Méthodes | getCondition : retourne la condition associée a l’expression conditionnelle |
| getIfBlockInstruction : retourne le block d’instruction associée à la condition |
| getNextContionnalExpression : retourne l’expression conditionnelle suivante |

<<interface>> ExprAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les expressions kawa |
| Règle de création |  |
| Méthodes | getValue : Retourne lesvaleurs |

<<interface>> ExprUnaireAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les expressions unaires kawa |
| Règle de création | EXPRESSION\_UNAIRE |
| Méthodes | getOperateur : retourne l’énumeration de l’opération |
| getOperande : retourne l’operande |

<<Interface>> ExprBinaireAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les expressions binaires kawa |
| Règle de création | EXPRESSION\_BINAIRE |
| Méthodes | getOperandeGauche : retourne l’opérande gauche |
| getOperandeDroite : retourne l’opérande droit |
| getOperateur : retourne l’énumération de l’opérateur |

<<Interface>> StringAST

|  |  |
| --- | --- |
| Role | Interface des éléments représentant les chaines de caractères |
| Règle de création | STRING |

<<Interface>> CallMethodAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les appels de méthodes kawa |
| Règle de création | APPEL\_METHODE |
| Méthodes | getArguments : retourne les arguments |
| getCaller : retourne l’appelant de la variable methode |

<<Interface>> VariableAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les variables kawa |
| Dépendance | NOM\_CHAMP |
| Méthodes | getVarType : retourne le type de la variable |
| getExpression : retourne l’expression associée a la variable |

<<Interface>> PrimitifAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les types primitifs kawa |

<<Interface>> BooleanAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les valeurs booléennes kawa |
| Dépendance | BOOLEAN |

<<Interface>> EntierAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les valeurs entières kawa |
| Dépendance | ENTIER |

<Interface>> FlotantAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les flottants en kawa |
| Regle de creation | FLOTTANT |

<<Interface>> CaractereAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les types primitifs kawa |
| Regle de creation | CHAR |

<<Interface>> ConcreteClassAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les classes concrètes kawa |
| Regle de creation | AST |
| Méthodes | getConstructeur : retourne les constructeurs |
| getPortee : retourne la portée du |
| getClassesParentes : retourne le nom des classes parentes |

<<Interface>> InterfaceAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les types primitifs kawa |
| Dépendance | AST |
| Méthodes | getPrototypes : retourne les prototypes |
| getInterfacesParentes : retourne les interfaces parents |
| getPortee : retourne la portée |

<<interface>> AbstractClassAST

|  |  |
| --- | --- |
| Rôle | Interface des éléments représentant les classes abstraites kawa |
| Règle de création | DEFINITION\_CLASS\_ABSTRACT |

<<Enumeration>> EnumerationKAWA

|  |  |
| --- | --- |
| Rôle | Enumération contenant les identifiant nécessaire a l’identification ou la construction des nœuds de l’arbre syntaxique |
| Dépendance | Tous les nœuds de l’arbre |

### 5.3 Optimiseur du code LLVM

Le rôle de l’optimiseur est de réduire et d’améliorer le code intermédiaire en réduisant le temps d'exécution d'une fonction, l'espace mémoire occupé par les données, voir la consommation d'énergie du programme.

On emploiera la commande **opt** de la suite de logiciel LLVM. **opt** est le module d’analyse et d’optimisation de LLVM. Il prend en entré un code source en langage intermédiaire LLVM puis renvoie en sortie le fichier optimisé et/ou un affichage du résultat l’analyse. Opt peut également faire appel aux du Framework Pass inclus dans LLVM pour l’optimisation du code intermédiaire.

### 5.4 Structure détaillé du back-end

Le Back-End est un terme désignant un étage de sortie d'un [logiciel](http://fr.wikipedia.org/wiki/Logiciel) devant produire un résultat. On l'oppose au [front-end](http://fr.wikipedia.org/wiki/Frontal) (aussi appelé un **frontal**) qui lui est la partie visible de l'iceberg. C'est dans cet étage que s'effectue le plus gros du traitement. Dans notre cas cette étape correspond à la transformation du code intermédiaire en un code interprétable ou exécutable par notre machine cible.

Notre compilateur fera appel a des outils de la chaine de compilation LLVM : **llc, lld, clang.**

### 5.5 Structure du code généré

Le code généré à la suite du back-end sera au format ELF (Exécutable and Linkable Format). C'est un format de fichier binaire (non-destiné à être lu par autre chose qu'une machine) couramment utilisé pour les codes compilés et exécutable à partir des systèmes UNIX.

Ce type de fichier se décompose en 3 parties :

- Premièrement un en-tête contenant les informations quant aux conditions d’exécution du fichier. L'entête est fixe selon la machine et le type de code.

- Puis les segments contenant les informations nécessaires à l’exécution du programme contenu dans le fichier.

- Et enfin les sections contenant les informations en rapport avec la résolution de liens et les replacements de données

Le code exécuter par les règles de bison sera chargé de créer l'arbre AST. A chaque reconnaissance d'une règle, ce code devra être capable de créer un nouvel élément à ajouter à l'arbre. Ce nouvel élément sera :

- Une feuille de l'arbre représentant une valeur ou un référence vers une valeur.

- Ou un nœud représentant une opération avec un ou plusieurs autre nœud descendant.

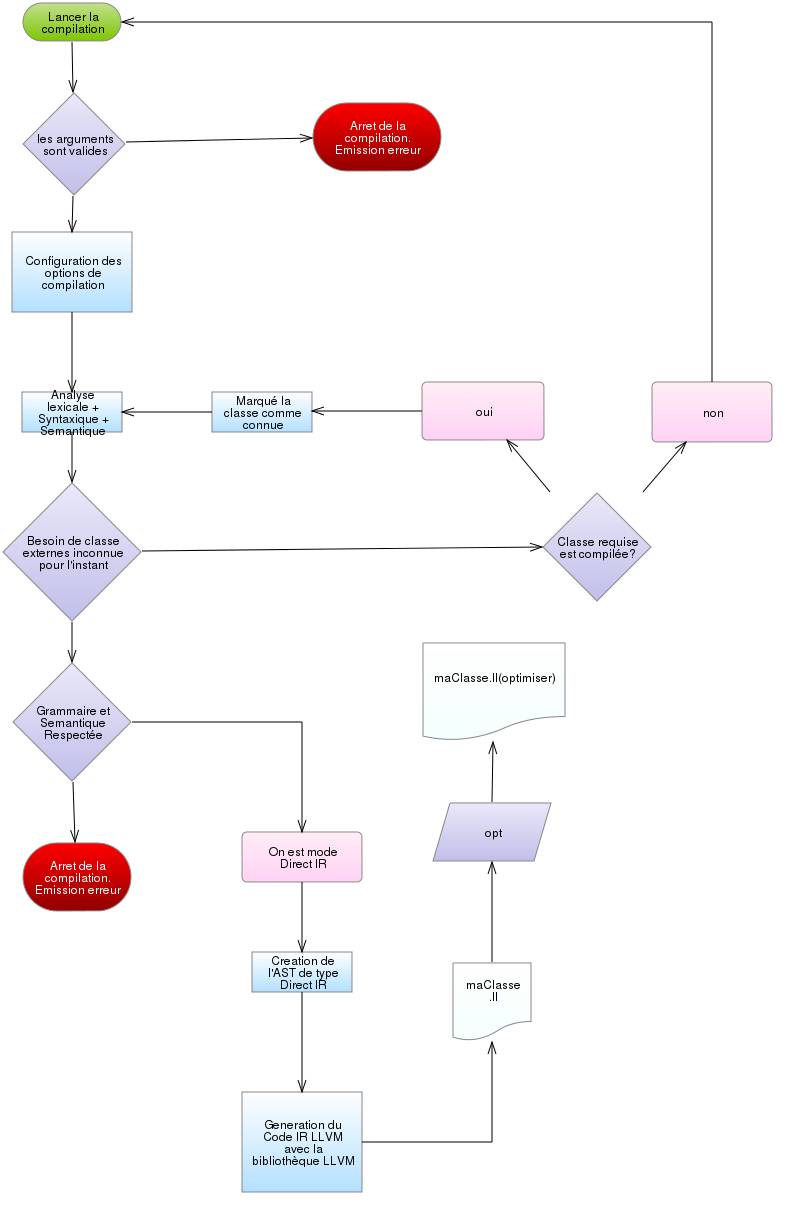
La création de l'arbre se fera grâce à des appels à des fonctions définies dans le code source écrit en C++ utilisant les bibliothèques de LLVM.

L'arbre AST sera à la base de la génération du Code IR. Chaque élément de l'arbre doit pouvoir générer le segment de Code IR qui lui est propre. C'est sur ce principe qu'intervient principalement LLVM : le code source sera chargé de définir non seulement la structure de l'AST et des éléments qu'il contiendra mais également les méthodes de génération de code qui seront appelées sur chaque élément de l'arbre lors de la génération de code IR.

**6. Fonctionnement dynamique**

### 6.1 Enchainement dynamique front-end et de l’optimisateur du code intermédiaire

Le schéma suivant décrit la chaine d’exécution qui produira la code compilé de nombre classe kawa.



### 6.2 Enchainement dynamique du Back End Le schéma suivant décrit la chaine d’exécution qui produira la code compilé de nombre classe kawa. C:\Users\User\Downloads\back_end.jpg