

# 1 Architecture du back end

Le back-end est composé de deux parties. La première partie dont le rôle est de générer le code intermédiaire, et une deuxième partie dont le rôle est de transformer le code en code machine au format ELF.

## 1.1 Le générateur du code intermédiaire LLVM

Il traduit l'arbre syntaxique du programme en code intermédiaire LLVM. Le générateur utilise le KawaTree pour obtenir les informations nécessaires à la génération du programme.

Ce dernier est composé de deux sous-parties :

- Le **IRCompiler** est le composant responsable de la production du code intermédiaire. Il parcourt l'arbre syntaxique grâce au KawaTree et appelle les objets de production du code intermédiaire.
- La famille de générateurs, un ensemble de classes utilisées pour la génération du code intermédiaire. Il s'agit d'une sur-couche à l'API LLVM déjà existante.

## 1.2 Structure du IRCompiler

Les nœuds de l'arbre syntaxique et la génération de code étant découplées, le *patron visiteur* est adapté pour l'implémentation du IRCompiler.

## 1.3 Famille de génération de code

Il s'agit d'une sur-couche aux classes de génération de code déjà existantes. Elle est composée d'un ensemble de classes implémentant l'ensemble des spécifications qui suivent. Les spécifications pourront être implémentées séparément pour le mode monolithique, et pour le mode partagé :

- **AffectationGenerator** : gère les opérations d'affectation
- **BlockGenerator** : gère la construction des blocs d'instruction
- **CallGenerator** : gère les appels de méthodes et de constructeurs
- **FunctionGenerator** : gère la création de fonction et de méthodes
- **GlobalVariableGenerator** : gère la création de variables globales nécessaire à la cohésion du programme
- **NameBuilder** : gère le nommage des éléments du code intermédiaire
- **PrimitiveBinaryOperation** : gère les opérations binaires
- **PrimitiveCreator** : gère la création des primitives
- **PrimitiveValueConverter** : gère la conversion des valeurs primitives
- **TypeGenerator** : gère la création et l'accès au typage LLVM

## 1.4 La chaîne de compilation du code IR

LLVM nous fournit des outils de compilation permettant de traduire le code IR LLVM bien formé en assembleur. Le code assembleur peut ensuite être tra-

duit en code machine. Cette chaîne de compilation est composée des éléments suivant :

- **llc** compile le fichier source en un fichier assembleur que l'on peut passer à un linker pour generer un executable
- **clang** compilateur permettant de compiler le code assembleur.

## 2 Fonctionnement dynamic

## 3 Enchainement des composants du compilateur

Le compilateur suivra la chaîne simple de compilation suivante :

1. **Parsage des fichiers dont les noms ont été donnés en argument**
  - **Nom de la fonction** : parse
  - **Paramètres** :
    - + KawaTree\* : la structure qui contiendra le squelette du programme
    - + vector<string> : liste des noms des fichiers
  - **Indication** : échoue si le programme est mal formé. Le KawaTree est alimenté, et est prêt pour l'analyse syntaxique.
2. **Analyse syntaxique**
  - **Nom de la fonction** : syntaxAnalyse
  - **Paramètres** :
    - + KawaTree\* : la structure qui contenant le squelette du programme
  - **Indication** : si aucune erreur est détectée, le KawaTree a été décorer et contient toutes les informations nécessaire pour la génération du code intermédiaire.
3. **Génération du code intermédiaire**
  - **Nom de la fonction** : generateIRCode
  - **Paramètres** :
    - + KawaTree\* : la structure qui contenant le squelette du programme, décoré.
  - **Indication** : un fichier .ll contenant le code intermédiaire est généré.
4. **Optimisation et compilation du code intermédiaire.**

### 3.1 Fonctionnement dynamic du IRCompiler

Le IRCompiler prendra en entrée un arbre syntaxique représentant la structure du programme à compiler. Les noeuds de l'arbre de cet arbre doivent implémenter les interfaces de la famille d'interface AST. (Voir annexe). Au cours de la production, on decorera l'arbre syntaxique avec des objets LLVM de rendre les noeuds réutilisables pour la production de code. La production du code LLVM IR d'un noeud de l'arbre syntaxique, entrainera la production en cascade des noeuds fils de ce noeud. Le IRCompiler est responsable des algorithmes de génération

de code IR. Selon le mode de compilation, il implémente différent algorithme.

- **MonolithiqueIRCompiler** Compilateur en mode monolithique. Il produit un fichier .ll unique contenant le code intermédiaire représentant toutes les classes compilées.
- **SharedIRCompiler** Compilateur en mode partagé, génère plusieurs fichiers .ll représentant les classes à compiler.

## 3.2 Fonctionnement des générateurs de code IR

Les générateurs de code IR sont des classes outils permettant la génération de code. Ces classes contiennent uniquement des fonctions statiques.

### 3.2.1 Fonctionnement du NameBuilder

L'implémentation des méthodes de cette classe nécessite la définition de contrats de nommage, permettant d'éviter toute ambiguïté entre les variables qui seront générées. Ce composant est utilisé par les autres classes de la famille de génération.

Ces fonction :

1. **buildFunctionName** : Construit un nom pour une fonction normale.
  - **Type de retour** : string
  - **Paramètres** :
    - + string : nom de la classe
    - + string : nom de la fonction
    - + string : type de retour
    - + vector<string> : liste des types des paramètres
  - **Indications** :
2. **buildConstructorName** : Construit le nom des fonctions permettant d'obtenir des instances d'un objet.
  - **Type de retour** : string
  - **Paramètres** :
    - + string : nom de la classe
    - + vector<string> : liste des types des paramètres
  - **Indications** :
3. **buildSubConstructorName** : construit le nom des fonctions permettant d'initialiser les nouvelles instances d'objets de des instances d'un objet.
  - **Type de retour** : string
  - **Paramètres** :
    - + string : nom de la classe
    - + vector<string> : liste des types des paramètres

- **Indications :**
- 4. **buildFunctionIndexName** : construit le nom d'un index de fonction
  - **Type de retour** : string
  - **Paramètres** :
    - + string : nom de la fonction bien construit par le name builder
  - **Indications :**
- 5. **buildAdHocTableName** : construit le nom d'une table adHoc
  - **Type de retour** : string
  - **Paramètres** :
    - + string : nom de la class static
    - + string : nom de la class dynamique
  - **Indications :**
- 6. **buildStaticVariableName** : construit le nom de la variable globale représentant une variable statique.
  - **Type de retour** : string
  - **Paramètres** :
    - + string : classe de la variable statique
    - + string : nom de la variable
  - **Indications :**
- 7. **buildAttributIndexName** : construit le nom de la variable globale représentant l'index d'un attribut.
  - **Type de retour** : string
  - **Paramètres** :
    - + string : nom de la classe
    - + string : nom de la variable
  - **Indications** : ???
- 8. **buildClassTypeName** : construit le nom de la classe pour le module
  - **Type de retour** : string
  - **Paramètres** :
    - + string : nom de la classe
  - **Indications** : Exemple : A devient KAWA\_\_CLASS\_\_A
- 9. **buildClassStructTypeName** : construit le nom de la structure représentant les membres d'une classe
  - **Type de retour** : string
  - **Paramètres** :
    - + string : nom de la classe
- 10. **LLVMTypeToStr** : construit le nom d'un type à partir d'un type LLVM
  - **Type de retour** : string
  - **Paramètres** :
    - + LLVM : :Type\* : type llvm à transformer

- **Indications** : float -> KawaFloat, i32 -> kawaInt, KAWA\_CLASS\_A -> A
- 11. **strToKawaType** : construit le nom pleinement qualifié d'un type pour le compilateur kawa
  - **Type de retour** : string
  - **Paramètres** :
    - + string : nom du type à transformer
  - **Indications** : int -> KawaInt, float -> KawaFloat

### 3.2.2 Fonctionnement du TypeGenerator

Ce composant permet de créer et d'obtenir des objets de typages d'un module llvm. Les types primitifs bool, int, char, float, double et string et tableaux représentant respectivement les types llvm i8, i32, i8, float, double, i8\*, [nb x type]. On les obtient grâce à la classe Type de l'api de LLVM. La structure principale de la classes est composée d'une référence vers un sous type contenant les attributs de la classe et une référence vers un tableau adHoc. Les méthodes et attributs statics sont quant à eux représentés par des variables globales. La génération d'une classe entrainera la génération de variables globales représentant les indexes des attributs dans la structure. Enfin le nommage de ces variables globales sera géré par le NameBuilder. (Voir l'annexe pour un exemple)

Ces Fonctions :

1. **createClassType** : crée une structure représentant une classe
  - **Type de retour** : LLVM : :Structuretype\*
  - **Paramètres** :
    - + string : nom de la classe
    - + vector<string> : liste des noms de attributs
    - + vector<string> : liste des noms des types
    - + vector<bool> : attributs statiques ou non
  - **Indications** : (Voir l'annexe pour un exemple)
2. **strToLLVMType** : retourne un type LLVM associé à un nom. Un type opaque est créé sinon la structure n'existe pas.
  - **Type de retour** : LLVM : :Structuretype\*
  - **Paramètres** :
    - + LLVM : :Module : module dans lequel sera fait l'operation
    - + string : nom du type
  - **Indications** : le nom du devra être un nom produit par le NameBuilder

### 3.2.3 Fonctionnement du PrimitiveCreator

Ce composant permet de créer des objets LLVM représentant des valeurs primitives. Les classes LLVM à utiliser sont : ConstantInt, ConstantFP, ConstantDataArray (Voir l'annexe pour un exemple)

1. **create** : Crée un objet LLVM représentant une valeur primitive.
  - **Type de retour** : LLVM : :Value\*

- **Paramètres :**
  - + string|float|double|int|char : valeur primitive
  - + LLVM : :Context : context de création de l'objet
- **Indications :** la fonction est à surchargée pour chaque cas

### 3.2.4 Fonctionnement du PrimitiveConverteur

Ce composant à pour role de creer les opérations de conversion d'un type vers l'autre. Le LLVM IR code ne supporte nativement les opérations binaires intertypées. Il faudra se servir de l'objet IRBuilder pour faciliter l'implémentation de la conversion. (Voir l'annexe pour un exemple)

1. **convertFormTo** : Convertie une valeur primitive d'un type à l'autre
  - **Type de retour** : LLVM : :Value\*
  - **Paramètres :**
    - + LLVM : :Type\* : type de départ
    - + LLVM : :Type\* : type cible
    - + LLVM : :Value\* : valeur convertir
    - + LLVM : :BasicBlock\* : bloc d'instruction dans lequel s'effectue la conversion.
  - **Indications :**

### 3.2.5 Fonctionnement du PrimitiveBinaryOperation

Ce composant permet la génération des opérations primitives. L'utilisation d'un IRBuilder est conseillée. (Voir l'annexe pour un exemple)

1. **create[Add|Mul|Sub|Div|Mod]** : cree un[e] [addition|produit|soustraction|division|reste]
  - **Type de retour** : LLVM : :Value\*
  - **Paramètres :**
    - + LLVM : :Type\* : type vers lequel sera convertie le résultat
    - + LLVM : :Value\* : opérande gauche
    - + LLVM : :Value\* : opérande droite
    - + LLVM : :BasicBlock\* : bloc dans lequel effectuer l'opération
  - **Indications :** le bloc ne devrait jamais être null
2. **create[And|Xor|Not|Eq|Or|Sup|Inf|SOE|IOE]** : cree une opération de comparaison
  - **Type de retour** : LLVM : :Value\*
  - **Paramètres :**
    - + LLVM : :Value\* : opérande gauche
    - + LLVM : :Value\* : opérande droite (sauf pour Not)
    - + LLVM : :BasicBlock\* : bloc dans lequel effectuer l'opération
  - **Indications :** la valeur llvm retournée est toujours un entier llvm. SOE = Supérieur ou égal. IOE = Inferieur ou égal.
3. **valToBool** : transforme une valeur en un boolean
  - **Type de retour** : LLVM : :Value\*

- **Paramètres :**
  - + LLVM : :Value\* : valeur à transformer
  - + LLVM : :BasicBlock\* : bloc dans lequel effectuer l'opération
- **Indications :** false si égal à 0 ou null, sinon true

### 3.2.6 Fonctionnement du FunctionGenerator

Ce composant permet de déclarer et d'avoir accès au composant d'un module. Il utilise Namebuilder pour générer le nom du code de la fonction dans le code IR. La génération d'une méthode non statique ajoute un paramètre supplémentaire représentant l'objet this. La génération d'un constructeur génère deux fonction llvm. L'une créant et renvoyant une instance de l'objet créer, et une fonction appelée par la précédente permettant l'initialisation de l'instance. Cette fonction prendra un paramètre supplémentaire représentant l'objet à initialiser. De plus il permet aussi de déclarer et d'obtenir des fonctions des bibliothèques présentes sur le système comme, strlen, strcpy, strcat..etc (Voir l'annexe pour un exemple)

1. **createFunction** : Crée une fonction dans un module llvm.
  - **Type de retour** : LLVM : :Function
  - **Paramètres :**
    - + Module module : module dans lequel est construit la fonction
    - + bool isStatic : la fonction est statique
    - + string className : le nom de la classe
    - + string ret\_type : type de retour de la fonction
    - + vector<string> arg\_types : liste des types des paramètres
    - + vector<string> arg\_names : liste des noms des paramètres
  - **Indications :** Le nom de la fonction produite est le nom de la fonction produite est générée avec le **NameBuilder**
2. **createConstructor** : crée une fonction d'instanciation de classe.
  - **Type de retour** : LLVM : :Function
  - **Paramètres :**
    - + Module module : module dans lequel est construit la fonction
    - + string className : nom de la classe
    - + vector<string> args\_types : types des arguments
    - + vector<string> args\_names : nom des arguments
  - **Indications :** Le nom de la fonction produite est le nom de la fonction produite est générée avec le **NameBuilder**. La construction d'un objet est faite par deux fonctions différentes. L'une automatiquement générée, permettant l'allocation d'un nouvel objet, et une fonction d'initialisation de l'objet. Le corps de cette dernière est à définir.
3. **getFunction** : Retourne une fonction du module à partir de son nom dans le module, null sinon
  - **Type de retour** : LLVM : :Function
  - **Paramètres :**
    - + Module module : module dans lequel est construit la fonction
    - + string className : nom pleinement qualifié de la fonction à chercher.

- **Indications** : le nom de la fonction à retourner devra être contruite avec le **NameBuilder**
- 4. **getOrCreateMainFunction** : crée ou retourne la fonction **main**
  - **Type de retour** : LLVM : :Function
  - **Paramètres** :
    - + LLVM : :Module : le module dans lequel creer ou obtenir la fonction
  - **Indications** : Il s’agit de la fonction de point d’entrer du programme.
- 5. **getOrCreatePutsFunction** : crée ou retourne la fonction **puts**
  - **Type de retour** : LLVM : :Function
  - **Paramètres** :
    - + LLVM : :Module : le module dans lequel creer ou obtenir la fonction
  - **Indications** : La fonction permet de l’affichage de chaînes de caractères. La fonction est implémentée par des librairies locales et chargée par clang.
- 6. **getOrCreateStrlenFunction** : crée ou retourne la fonction **strlen**
  - **Type de retour** : LLVM : :Function
  - **Paramètres** :
    - + LLVM : :Module : le module dans lequel creer ou obtenir la fonction
  - **Indications** : La fonction permet de calculer la longueur d’une chaine de caractère. La fonction est implémentée par des librairies locales et chargée par clang.
- 7. **getOrCreateStrcatFunction** : crée ou retourne la fonction **strcat**
  - **Type de retour** : LLVM : :Function
  - **Paramètres** :
    - + LLVM : :Module : le module dans lequel creer ou obtenir la fonction
  - **Indications** : La fonction permet de concatener deux chaines de caractères. La fonction est implémentée par des librairies locales et chargée par clang.
- 8. **setFunctionBody** : permet de définir le corps d’une fonction
  - **Type de retour** : void
  - **Paramètres** :
    - + LLVM : :Module : le module dans lequel creer ou obtenir la fonction
    - + vector<LLVM : :BasicBlock\*> : La liste des blocs llvm comportant les instructions d’une fonction
  - **Indications** : cette fonction s’appliquera aussi aux constructeurs.

### 3.2.7 Fonctionnement du CallGenerator

Ce composant permet de générer les appels aux méthodes et au chargement d’attributs. Le composant fonctionne de la manière suivante.

- Les appels aux membres static sont directes. Une résolution de nom est suffisante pour obtenir l’adresse de l’objet cible. Les méthodes et les attributs static sont des variables globales dans le module, et repérables par leur nom.
- Les attributs et méthodes non statics sont contenus des structures et tableaux représentant les classe et les tables adHoc permettant d’implementer le principe de polymorphisme. Leur position dans la structure doivent pouvoir être determiner afin de les charger. Pour cela il faut employer la méthode getIndexOfMember de **GlobalGenerator**.



- Les appels aux constructeurs d'objets sont traités comme des appels statics.
  - (A compléter)
- (Voir l'annexe pour un exemple)

1. **callStaticMethode** : cree un appel static sur une méthode
  - **Type de retour** :????
  - **Paramètres** :
    - + ???
  - **Indications** :???
2. **callMethode** : cree un appel non static sur une méthode
  - **Type de retour** :????
  - **Paramètres** :
    - + ???
  - **Indications** :???
3. **loadStaticAttribut** : charge un attribut static
  - **Type de retour** :????
  - **Paramètres** :
    - + ???
  - **Indications** :???
4. **loadAttribut** : charge un attribut non static
  - **Type de retour** :????
  - **Paramètres** :
    - + ???
  - **Indications** :???

### 3.2.8 Fonctionnement du GlobalVariableGenerator

Ce composant permet la génération des variables globales nécessaire à la sémantique du programme. Il permet de declarer des variables statics, les tables AdHoc, et les indexes de membres. La compilation en mode Monolithique et partagé divergera à ce niveau au niveau de attributs des variables. (Voir l'annexe pour un exemple)

Ses fonctions :

1. **getOrCreateStaticAttribut** : cree ou retourne un attribut static
  - **Type de retour** :????
  - **Paramètres** :
    - + LLVM : :Module : module llvm dans lequel effectuer l'opération
    - + string : nom de la classe
    - + string : nom de l'attribut
    - + string : nom du type de l'attribut
  - **Indications** :
2. **createAdHocTable** : cree une table adHoc référençant les fonction correspondant au couple static-dynamique de deux classes
  - **Type de retour** : LLVM : :Value\*
  - **Paramètres** :
    - + LLVM : :Module : module dans lequel creer le tableau

- + string : nom de la classe statique
  - + string : nom de la classe dynamique
  - + vector<LLVM : :Function \*> : liste des fonctions de la table
  - **Indications** : le nom des classes doivent avoir été générée avec le Name-Builder. A l'issue de l'opération, chacune des fonction se verra affecté un index par rapport à sa position dans la liste. Il est donc nécessaire que la liste des fonctions soient ordonnées.
3. **getAdHocTable** : permet d'obtenir une table adHoc
    - **Type de retour** : LLVM : :Value\*
    - **Paramètres** :
      - + LLVM : :Module : module dans lequel creer le tableau
      - + string : nom de la classe statique
      - + string : nom de la classe dynamique
    - **Indications** : le nom des classes doivent avoir été généré avec le Name-Builder.
  4. **getAdHocTable** : permet d'obtenir une table adHoc
    - **Type de retour** : LLVM : :Value\*
    - **Paramètres** :
      - + LLVM : :Module : module dans lequel creer le tableau
      - + string : nom de la classe statique
      - + string : nom de la classe dynamique
    - **Indications** : le nom des classes doivent avoir été généré avec le Name-Builder.
  5. **createIndexOfMember** : permet de creer un index pour un élément global dans un module llvm
    - **Type de retour** : LLVM : :Value\*
    - **Paramètres** :
      - + LLVM : :Module : module dans lequel creer le tableau
      - + string : nom de l'élément
      - + int : index
    - **Indications** : le nom de l'élément doit avoir été générée avec le NameBuilder.
  6. **getIndexOfMember** : permet d'obtenir un objet llvm représentant un entier correspondant à l'index
    - **Type de retour** : LLVM : :Value\*
    - **Paramètres** :
      - + LLVM : :Module : module dans lequel creer le tableau
      - + string : nom de l'élément
    - **Indications** : le nom de l'élément doit avoir été générée avec le NameBuilder.

### 3.2.9 Fonctionnement du BlockGenerator

Ce composant permet d'ajancer les bloc d'instruction LLVM lors de la génération du code d' instructions conditionnelles.

Liste des fonctions :

1. `???:???`
  - **Type de retour** :????
  - **Paramètres** :
    - + ???
  - **Indications** :???