Ymir

Documentation technique

Table des matières

1 Introduction

Ymir est un langage haut niveau inspiré par D, python et OCaml. L'objectif de ce rapport est de décrire le fonctionnement du compilateur. Le compilateur est découpé en deux parties principales, elle même découpé en plusieurs partie.

— **FrontEnd**
  - Analyse Syntaxique, récupère les fichiers passé en entrée et génére l'arbre de syntaxe abstraite.
  - Analyse sémantique, créer les différents symboles du programme et vérifie l'intégrité sémantique du programme.
  - Génération du code intérmédiaire, c'est la dernière partie du **FrontEnd**, elle génére un langage intérmédiaire qui ne dépend pas de l'architecture visé par le compilateur.

— **BackEnd**
  - Génération du code du code cible, le langage intérmédiaire est récupéré et transformé dans le code ciblé par le compilateur.
  - Edition des liens, cette partie est gérer par le compilateur *gcc* pour le moment.

2 FrontEnd

2.1 Analyse Syntaxique

L'analyse syntaxique est la première phase du **FrontEnd**, elle prend en entrée le nom d'un fichier texte et génére un arbre de syntaxe abstraite. Tous les modules composant l'analyse syntaxique se trouve dans le sous dossier source *syntax*. L'analyse syntaxique contient un module qui génére les jetons (*token*), ce module s'appele *Lexer*. Il permet de lire le fichier et de découper les mot en fonctions de paramètre qu'on lui passe. Ces paramètres dépendent de l'énumeration *Token*. On peut lui donner une liste de jetons qui découpe et de paires de jetons qui annoncent le début et la fin d'un commentaire. Le *Lexer* agit sur un fichier, dont on passe le nom en paramètre de son constructeur.

Le module *Visitor* est le module de parcours syntaxique des entrées générés par le *Lexer*. On lui passe un nom de fichier en entrée, il construit un *Lexer* associé, et crée l'arbre de syntaxe abstraire en fonction de la syntaxe d'**Ymir**. Si le fichier comporte une erreur de syntaxe, le visiteur renvoi une exception de type *SyntaxError*, qui contient l'emplacement de l'erreur et le message d'erreur associé.

La syntaxe du langage **Ymir** est la suivante :

```
program := function | import | struct | extern | enum | public | private

public := 'public' (
                     ('{' (function | import | extern | struct | enum)* '}')
                     | (function | import | extern | struct | enum)
                   )

private := 'private' (
                     ('{' (function | import | extern | struct | enum)* '}')
                     | (function | import | extern | struct | enum)
                   )
```

```
import := 'import' Identifiant ('.' Identifiant)*
                               (',' Identifiant ('.' Identifiant))* ';'

struct := 'struct' ('|' Identifiant ':' type)* '->' Identifiant ';'

enum := 'enum' ('|' Identifiant ':' expression)* '->' Identifiant ';'

function := 'def' Identifiant ('(' vardecl (',' vardecl)* ')' )?
                              '(' (vardecl (',' vardecl)*)? ')'
                              (':' type)? block

extern := 'extern' ('(' Identifiant ')')? Identifiant
                                          '(' (vardecl (',' vardecl)*)? ')'
                                           (':' type)? ';'

var := type
vardecl := var (':' type)?

type := ( Identifiant ('!' (('(' expression (',' expression)* ')') | expression ))
        | '[' type ']'

Identifiant := ('\_')* ([a-z]|[A-Z]) ([a-z]|[A-Z]|'\_'|[0-9])*
block := ('{' instruction* '}') | instruction

instruction := if
               | return
               | for
               | while
               | break
               | delete
               | let
               | ';'
               | expressionult


let := 'let' var ('=' right)? (',' var ('=' right)?)* ';'

expressionult := expression (Ultop expression)*

expression := ulow (Expop ulow)*

ulow := low (Ulowop low)*

low := high (Lowop high)*

high := pth (Highop pth)*
```

```
pth := ('(' expression ')') (suite)?
       | (BefUnary expression)
       | (expression AfUnary)
       | constante
       | expand
       | constTuple
       | leftOp (suite)?

constante := Decimal | Flottant | String | Char | Bool | Null

Decimal := [0-9]+ ('UB' | 'US' | 'UL' | 'B' | 'S' | 'L')?
Flottant := ('.' [0-9]+) | ([0-9]+ '.') | ([0-9]+ '.' [0-9]+)
String := ('\"'[.]*'\"') | ('\''[.]+[.]'\'')
Char := '\''.'\''
Bool := 'true' | 'false'
Null := 'null'

leftOp := cast | constArray | funcPtr | var

constArray := '[' (expression (',' expression)*)? ']'

cast := 'cast' ':' type '(' expression ')'

funcPtr := 'function' '('(vardecl (',' vardecl)*)? ')' ':' type (block)?

suite := ('(' par) | ('[' access) | ('.' dot)

par := (expression (',' expression)*)? ')'

access := (expression (',' expression)*)? ']'

dot := Identifiant ('.' Identifiant)*

constTuple := '(' expression (',' expression)+ ')'

expand := 'expand' '(' expression ')'

if := 'if' expression block (else)?

else := 'else' (if | block)

return := 'return' (expression)? ';'

break := 'break' (Identifiant)? ';'
```

```
while := 'while' (':' Identifiant) expression block

for := 'for' (':' Identifiant) ('(' expression 'in' expression ')')
                               | (expression 'in' expression)
```

2.2 Analyse Sémantique

L'analyse sémantique commence par déclarer les différents symboles globaux du module en cours de compilation. Les symboles que l'on déclare au debut sont les fonctions, les externes, les structures, les imports et les enums. C'est certainement la partie la plus compliqué du compilateur. Elle regroupé dans plusieurs module qui vont gérer la disponnibilité des symboles, leurs type et quelque éléments de vérification statique.

L'arbre de syntaxe abstraite, présent dans le module *ast*, est composé de déclarations, elle même composé d'instruction. Les déclaration possèdent une méthode qui les enregistres dans la table des symboles. Il existe deux variantes de cette méthodes, une qui les insérent comme faisant partie du module courant, ou faisant partie d'un module compilé externe. Dans le cas ou elle proviennent d'un module externe leurs corps ne sera compilé que dans certaine condition, mais surtout elle ne sont insérées dans la table que si elle ont été déclaré comme publique.

Cette partie va présenter comment sont traité les différentes déclarations, puis traiter du fonctionnement du typage, pour finir par expliquer le fonctionnement des templates.

2.2.1 Déclaration - Fonctions

Il existe plusieurs types de fonctions :

- pure, les fonctions dont on connait le type de tout les paramètres et qui sera compilé même si elle n'est jamais référencée.
- impure, les fonctions dont il manque au moins un type dans les paramètres. Les types manquant seront inféré lors du référencement de la fonction, il est donc nécéssaire qu'elle le soit au moins une fois pour pouvoir la compiler.
- template, ces fonctions reposent sur le même principes que les fonctions impures, seulement le système d'inférence de type est un peu plus complexe et permet de spécialiser plus facilement.

2.3 Génération langage intérmédiaire

3 BackEnd

3.1 Génération du code cible