UNIVERSITÉ DE RENNES 1

# TP2 VSL+ / LLVM

**DELORME Emily**
**LARGEAU Brandon**

## Programmation Dirigée par la Syntaxe

*Année 2019 – 2020*

# Introduction

## Liens utiles

Projet Github : https://github.com/emilydelorme/master1-pds

SonarQube : https://sonar.emilydelorme.xyz/dashboard?id=master1-pds

## Projet

Ce que couvre le projet :

- Les expressions simples
- L'instruction d'affectation
- La gestion des blocs
- La déclaration des variables
- Les expressions avec variables
- Les instructions de contrôle if e twhile
- La définition et l'appel de fonctions (avec les prototypes)
- Les fonctions de la bibliothèque: PRINT et READ
- La gestion des tableaux (déclaration, expression, affectation et lecture)

La gestion des erreurs fonctionne. Tous les fichiers du dossier "testlevelerror" ont étaient testés. Seul le test sur le fichier "test_invalid_call3.vsl" ne fonctionne pas (pas d'erreur). Si dans une appel de fonction j'envoie un tableau alors qu'une variable normale est attendu je devrais avoir une erreur et ce n'est pas le cas.

Tous les fichiers du dossier "testlevel4" sont lus et correctement affichés avec le pretty printing. La grammaire semble donc valide.

Il y a un petit problème d'indentation au niveau des if then else while.

# Build

Pour tester le programme, il faut avoir **Java 11**. Vous devez compiler le projet avec Gradle en faisant la commande (se mettre dans le dossier contenant le fichier) :

Sous Linux (dans un terminal) :

```
./gradlew build
```

Sous Windows (powershell) :

```
.\gradlew build
```

# Tests Gradle

Pour réaliser les test avec gradle, il faut exécuter la commande suivante.

Les résultats des test sont disponibles à cet endroit : *build/reports/tests/test/index.html.*

Sous Linux (dans un terminal) :

```
./gradlew test
```

Sous Windows (powershell) :

```
.\gradlew test
```

## Tests Gradle Run

Pour exécuter la méthode Main comme avec eclipse vous pouvez le faire comme cela :
Sous Linux (dans un terminal) :

```
./gradlew run –args='-f tests\testlevelerror\level4testheap.vsl -o'
```

Sous Windows (powershell) :

```
.\gradlew run –args='-f tests\testlevelerror\level4testheap.vsl -o'
```

Le code Llvm sera sauvegarder dans *build/llvm/output/level4testheap.ll*

## Arguments

Il y a 3 différentes options lorsque qu'on appelle Main (avec gradle run ou directement le jar)

- l'option « -f <fichier.vsl> » permet de spécifier le chemin du fichier à compiler

- l'option « -d <dossier> » permet de compiler les fichier .vsl d'un dossier

- l'option « -o » permet d'activer l'écriture du résultat dans build/llvm/output/*

## Eclipse

Ouvrez Eclipse, « File → Import → Existing Gradle Project » et sélectionnez ensuite la racine du projet.

Il est possible que sur Eclipse vous ayez besoin du plugin ANTLR et Gradle.

Ensuite sur Eclipse, « Run → Run Configurations → (clique droit) Java Application → New Configuration ».

**Project** : TP2
**Main class :** TP2.Main

Dans l'onglet « Arguments » :

Program arguments : -f tests\testlevelerror\level4testheap.vsl -o

Appliquez et Run.

# Description des classes/Packages

**TP2 :** Contient le Main avec 2 classes utilitaire (TypeLabel et Utils)

**TP2.ASD :** Contient Program qui est le point d'entrée de l'ASD, contient aussi toutes les interfaces utilisées dans le projet.

**TP2.ASD.Expression :** Contient la définition de chaque expression (Add, Div, Mul, Sub, Integer = chiffre et Variable = une variable simple). (ExpressionHelper permet de simplifier les opérations avec un code générique).

**TP2.ASD.Item :** « Print » affiche des objets de type item, un item étant soit du texte soit une expression.

**TP2.ASD.Parameter :** Un paramètre étant soit une variable de type Array ou Basic, on utilise des classes séparées car une variable array en paramètre est différent d'une déclaration d'un array « t[] » contre t[10] ».

**TP2.ASD.Ret :** Permet le retour des InstructionHandler ainsi que leurs résultats. Il y a deux variante. Dont une qui permet la gestion des types quand il y a besoin.

**TP2.ASD.Statement :** Contient toutes les classes définissant les instructions. Une instruction étant une affectation, (ControlType qui permet a Statements utils de généré l'IR des blocs de contrôle), appel de fonction, if then else, if then, while do, print, read, return,.
StatementUtils donne des outils pour les noms de variable, la profondeur actuelle, la fonction actuelle et la creation du code IR pour les blocs de contrôles.

**TP2.ASD.Statement.Block :** Un statement peut aussi être un block, un block pouvant contenir une déclaration de variable ou non. Les variable sont des VariableFormDeclaration.

**TP2.ASD.Statement.Block.VariableFormDeclaration :** Contient une classe Array et une class Basic. La classe Array gère la déclaration des tableau « INT t[10] », un tableau ne peut pas être déclaré avec une taille étant une expression, basic gère la déclaration des variables basics « INT a ».

**TP2.ASD.Types :** Contient les classes des types gérés.

**TP2.ASD.Unit :** Contient les classes Prototype et Function.

**TP2.ASD.VariableForm :** Contient les classes pour l'utilisation des variables. Array gère les tableau « t[expression] » et basic gère les variables simple « a ».

**TP2.exceptions :** Contient les exceptions personnalisées.

**TP2.SymbolTable :** Contient toutes les classes liées à la table des symboles. SymbolTable est la classe principale qui va stocker les Symbol. Un symbol peut être une variable, un prototype ou une fonction.

**TP2.Llvm :** Contient toutes les classes qui servent à gérer le code Llvm final

**TP2.Llvm.Instructions :** Contient toutes les instructions permettant de générer le code Llvm

**TP2.Llvm.Instructions.alloca :** Contient les instructions d'allocation

**TP2.Llvm.Instructions.functions :** Contient les instructions en rapport avec les fonctions

**TP2.Llvm.Instructions.io :** Contient les instructions d'entrée et de sortie

**TP2.Llvm.Instructions.load :** Contient les instructions de chargement de variables et tableau

**TP2.Llvm.Instructions.Operations :** Contient les instructions permettant de gérer les operations et une instruction spécifique pour comparer une variable à 0

**TP2.Llvm.Types :** Contient les différents types Llvm

# Détection des erreurs

Le code VSL+ est analysé statiquement pour détecter les erreurs avant de générer le code IR et avant toute compilation du code IR. La fonction qui gère cette analyse est checkError(). Nous utilisons le pattern factory. On a une fonction static **create()** qui analyse le code VSL+. Le constructeur est en privé, si une erreur est détectée alors le programme s'arrête sinon la création d'une instance de Program est faite.

La table des symboles est gérée par la grammaire. Elle est envoyée aux règles de la grammaire qui en ont besoin. Une table des symboles est créée globalement à tout le programme. Elle gérera les fonctions/prototypes et sera utilisée pour les appels de fonction. Pour chaque fonction/prototype, une nouvelle table des symboles est créée pour les arguments et la table des symboles globale est son parent. Ensuite, pour chaque bloc, une nouvelle table des symboles est créée, si c'est le premier bloc alors son parent sera la table des symboles avec les arguments sinon ça sera le bloc au-dessus. En schématisant un peu :

**program:**
*Création d'une table des symboles globale qui va contenir les FonctionSymbol et PrototypeSymbol*
*(on l'appel symbolTable)*
*$out = TP2.ASD.Program.create(units, symbolTable);*

...

**prototype:**
*Création d'une table des symboles qui va contenir les arguments du prototype (comme si c'était un nouveau bloc)*
*(on l'appel symbolTableWithArgs)*
*symbolTableWithArgs.setParent(symbolTable);*
*ajout d'un PrototypeSymbol à symbolTable*
*/!\ symbolTableWithArgs ne sert à rien ici, c'est juste que la règle parameters dans la grammaire attend en paramètre [List<TP2.SymbolTable.VariableSymbol> arguments, TP2.SymbolTable.SymbolTable symbolTableWithArgs] /!*

**function:**
*Création d'une table des symboles qui va contenir les arguments de la fonction (comme si c'était un nouveau bloc)*
*(on l'appel symbolTableWithArgs)*
*symbolTableWithArgs.setParent(symbolTable);*
*ajout d'un FunctionSymbol à symbolTable*
*on envoie à la règle "statement" symbolTableWithArgs (donc à ce point là, on a une table globale avec nom de fonction + proto et qui est le parent de symbolTableWithArgs qui elle contient les arguments de la fonction)*

**parameter:**

o*n récupère symbolTableWithArgs et on y ajoute chaque argument*

**statement:**

o*n a accès dans la grammaire à symbolTableWithArgs qu'on passe à chaque statement (affectation, ifState, ..., block, ...)*

...

**block:**

*Création d'une nouvelle table des symboles qui contiendra les VariableSymbol qui seront fait dans la règle « declaration »*

*(On l'appel symbolTableBlock)*

*symbolTableBlock.setParent(symbolTableWithArgs);*

*On envoie cette nouvelle table à la règle "declaration"*

*Un block étant soit (declarations + statements) soit (statements), on envoie à chaque « statement » la table symbolTableBlock*

**declaration:**

o*n a accès à symbolTableBlock*

*Pour chaque variable détectée en déclaration, on envoie la symbolTableBlock à la règle variableFormDeclaration qui lui va ajouter un VariableSymbol à symbolTableBlock*

symbolTable (contient tous les FunctionSymbol et PrototypeSymbol)
  symbolTableWithArgs (nouvelle fonction : accès aux fonctions/protos + arguments)
    symbolTableBlock (accès aux fonctions/protos + arguments + nouvelles déclarations s'il y en a)
      symbolTableBlock ...
        symbolTableBlock ...

Chaque classe gère sa génération de IR, c'est pour ça que l'on sépare les variables entre paramètre, déclaration et utilisation. Chaque usage a une génération de code IR différente.