Matthieu VINCENT

Loïc REISDOERFER

Mervine LIEFFROY

Martin LEMAITRE

**Projet de compilation**

**Rapport**

L3 MIAGE année 2016 / 2017

**Sommaire**

[Introduction 3](#_Toc480712450)

[Définition de notre langage 4](#_Toc480712451)

[Programme qui calcul la somme des n premiers entiers 4](#_Toc480712452)

[Plus petit programme accepté 4](#_Toc480712453)

[Les variables 4](#_Toc480712454)

[Les commentaires 5](#_Toc480712455)

[Lecture et écriture 5](#_Toc480712456)

[La structure de contrôle 5](#_Toc480712457)

[La conditionnelle 5](#_Toc480712458)

[Déclaration et appel de fonction 6](#_Toc480712459)

[Réalisation du projet 7](#_Toc480712460)

[Organisation 7](#_Toc480712461)

[La grammaire 7](#_Toc480712462)

[La table des symboles (TDS) 7](#_Toc480712463)

[L’arbre syntaxique (AST) 7](#_Toc480712464)

[Le générateur UASM 7](#_Toc480712465)

[Les difficultés rencontrées 7](#_Toc480712466)

[Conclusion 8](#_Toc480712467)

# Introduction

Le but de ce projet était de réaliser un compilateur pour un langage de programmation que nous avons nous même créé. Le compilateur doit dans un premier temps vérifier la syntaxe du programme source (écrit dans notre langage) puis générer un programme cible en langage assembleur beta.

Ce principe est le même pour tous les langages de programmation existant. Le schéma suivant décrit simplement le processus de compilation.

Compilateur

Programme cible

Programme source

Compilateur qui permet la traduction du fichier en assembleur.

Fichier contenant le même code mais écrit en assembleur beta.

Fichier contenant du code (écrit avec notre langage).

On peut donc vulgariser la fonction d’un compilateur en disant que c’est un traducteur. C’est cette partie que nous avons donc programmé.

# Définition de notre langage

## Programme qui calcul la somme des n premiers entiers

Voici le programme dans notre langage qui permet de calculer la somme des n premiers entiers.

Exemple pour la valeur 3 : 0+1+2+3 qui vaut 6

*fonction entier sommePremierEntier(n){*

* *loc entier iterateur = 0;*
* *loc  entier resultat = 0;*

*tantQue (iterateur <= n) {*

*resultat = resultat + iterateur;*

*iterateur = iterateur + 1;*

*}*

* *retourner resultat;*

*}*

## Plus petit programme accepté

Voici le plus petit programme que pourra accepter notre langage. Ce bout de code ne produira rien comme résultat et on peut remarquer que tous les programme commenceront pas la fonction principale.

*fonction principal(){*

*}*

## Les variables

Cette partie permet de voir comment les variables sont déclarées dans notre langage. Les variables seront toujours du type entier (seul type que notre programme prendra en compte au début). Une distinction sera faite pour les variables globales, accessible dans tout le programme, et les variables locales, accessible uniquement dans le bloc de code où elles sont déclarées.

*glob entier maVariable ;*

*loc entier maVariable ;*

Afin de faire une affectation de valeur à une variable, il suffit d’utiliser le signe **=**.

## Les commentaires

Notre langage contiendra un système de commentaire comme dans tout autre langage de programmation. Il sera possible de faire des commentaires sur une ligne ou multi-lignes.

*/\**

*\* Commentaire multi-lignes*

*\*/*

*// Commentaire sur une seule ligne*

## Lecture et écriture

Notre langage permettra la saisie de valeur au clavier et aussi l’écriture à l’écran. Pour cela il faudra appeler ces deux méthodes qui seront déjà implémentés avec le langage.

*glob entier maVariable = lire() ;*

*ecrire(maVariable) ;*

## La structure de contrôle

Notre langage proposera une structure de contrôle qui est le tant que. Voici le code de celle-ci :

*tantQue (condition){*

*}*

Une condition peut être une opération du type =, <, >, <=, >=.

## La conditionnelle

Le langage contiendra une structure conditionnelle qui est le si. Voici son code :

*si (condition){*

*}*

Comme défini au dessus, une condition est une opération du type =, <, >, <=, >=.

## Déclaration et appel de fonction

Afin de réaliser des programmes plus complets, notre langage pourra contenir des fonctions (comme dans la première partie de ce document). Voici la déclaration d’une fonction :

*fonction entier nom(paramètre1, paramètre2, ...){*

*// Code de la fonction*

*retourner variable ;*

*}*

Pour appeler une fonction, il suffit de mettre le résultat de la fonction dans une variable.

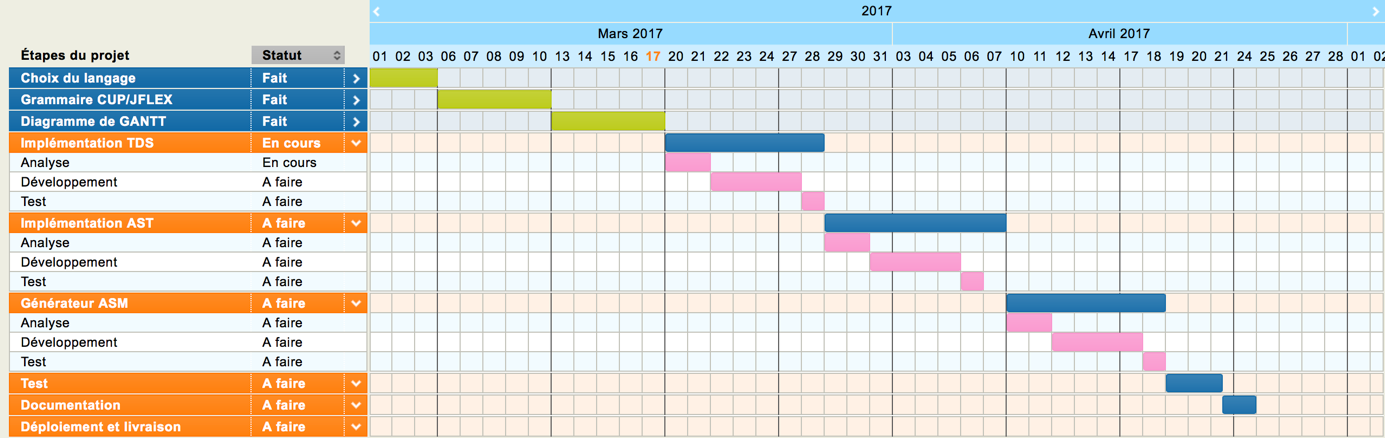
*glob entier maVariable = appel maFonction(paramètre1, paramètre2, ...) ;*

# Réalisation du projet

Dans cette partie nous allons vous expliquer toute la réalisation du projet et les différentes phases de celui-ci. Ensuite nous évoquerons les difficultés que nous avons rencontrées.

## Organisation

Pour bien organiser notre travail, nous avons élaboré un diagramme de GANTT en début de projet afin de spécifier le temps alloué au différentes taches que nous avons identifiées.



Ce diagramme nous a également permis de nous répartir le travail entre nous en fonction des tâches restantes en mettant très régulièrement à jour notre diagramme de GANTT.

## La grammaire

Nous avons créé une grammaire à l’aide de JFlex et de CUP pour pouvoir reconnaître notre langage et ainsi faire les opérations indispensables à la compilation de celui-ci.

Dans un premier temps, nous avons créé un scanner en JFlex pour traiter les caractères présents dans nos fichiers de test. JFlex permet de reconnaître des classes de caractères comme pas exemple les mots clefs de notre langage grâce à l’utilisation d’expressions régulières.

Ensuite, nous avons implémenté un parser à l’aide de CUP. Ce fichier réceptionne chaque classe de caractères reconnues dans le scanner JFlex. Pour simplifier l’explication, on peut dire que CUP s’occupe de vérifier que chaque classe de caractères soit mise dans un ordre spécifique.

Exemple :

JFlex :

...

"fonction" { return new Symbol(sym.FONCTION); }

"principal" { return new Symbol(sym.PRINC); }

...

Lorsque la chaine de caractère « fonction » et/ou « principal » seront reconnues par le scanner, les symboles FONCTION et/ou PRINC sont envoyé au parser CUP.

CUP :

## ...

FONCTION PRINC

...

Le parser va maintenant vérifier que l’ordre des symboles soit bien respecté. Dans cet exemple, le mot fonction doit être devant le mot principal.

## La table des symboles (TDS)

La table des symboles est un tableau qui contient toutes les données utiles relative aux variables et au différentes fonctions qui seront détecté dans notre programme source.

Concernant les variables, la TDS contient leurs noms, leurs scopes (variable globale ou locale) et leurs valeurs associées.

Les informations enregistrées concernant les fonctions sont leurs noms et leurs scopes (fonction normale ou principale).

Ces différentes informations seront utilisées par le générateur de code assembleur pour pouvoir organiser le code, déclarer des variables ou encore faire la jonction entre différentes fonctions.

Concernant l’implémentation de cette structure de données, nous avons utilisé un HashMap<String, Symbole>. La clef correspond à un numéro unique et la valeur associée est un type Symbole que nous avons également créé. Ce type contient tous les attributs nécessaires (nom, scope et valeur).

Pour avoir une utilisation optimale de notre structure de données, nous avons également implémenter différentes méthodes comme :

- l’ajout et la suppression de symbole

- l’accession à un symbole par sa clef unique

## L’arbre syntaxique (AST)

Afin d’instancier un arbre syntaxique, nous avons implémenté une classe Nœud qui possède en attribut une liste de nœud. Cette liste représente les fils de nœud courant. Avec cette conception, nous pouvons gérer par récursivité l’ensembles des nœuds qui composent l’arbre syntaxique.

La classe Nœud possède différentes méthodes qui permettent de bien gérer cette structure. On peut par exemple :

- ajouter des nœud fils

- modifier les caractéristiques d’un nœud

- vérifier que l’arbre possède une structure valide (certains nœuds doivent posséder un nombre spécifique de fils)

Chaque nœud qui forment l’arbre syntaxique sont construit grâce à la récursivité ascendante que fourni le parser CUP.

## Le générateur UASM

Cette partie à été assez simple à mettre en place grâce au différents TD que nous avons fait. Nous avons donc traduit les algorithmes élaborés en cours en Java.

La classe GenerateurUASM prend en paramètre un arbre syntaxique ainsi qu’une table des symboles. Pour chaque nœud de l’arbre une méthode spécifique est appelé et génère du code assembleur qui est retourné à la fin du programme.

## Les difficultés rencontrées

Nous avons éprouvé quelques difficultés notamment lors de la création des nœuds dans le parser CUP. Comme le parser fonctionne de manière ascendante, il était compliqué pour nous d’ajouter le bon fils au bon nœud parents. Notre méthode de vérification de la structure de l’arbre syntaxique nous a permis de résoudre ce problème assez rapidement.

# Conclusion

Ce projet nous a permis d’avoir une vision d’ensemble sur le fonctionnement d’un compilateur.

Nous avons également pu utiliser des outils puissants tels que :

- JFlex et CUP pour l’analyse lexical

- Ant qui permet d’automatiser certaines opérations répétitives