Compilateur LAMAC

Rapport de projet tutoré

Aurélien Peden
Loïc Bernard
Antoine Descamps
Maxime Huyghe

Tuteur : Denis Monnerat

IUT de Sénart-Fontainebleau
Département informatique
26 février 2020

Table des matières

Table des figures

1 Présentation

Nous voulions un sujet ambitieux pour ce projet et avons donc choisi de réaliser un compilateur pour un langage simple. Le but initial était de créer un langage éducatif facile d'accès afin de permettre un apprentissage de la programmation sans encombres.

Dans ce rapport sera mentionné l'organisation mise en place, les difficultés d'organisations rencontrées, le principe de fonctionnement d'un compilateur notamment à l'aide de schémas, ainsi qu'une présentation du langage implémenté. Il s'y trouvera par ailleurs une explication concernant le choix du langage, et les diverses pistes explorées avant d'effectuer ce choix.

Vous pourrez également trouver différentes annexes, comme le cahier des charges initial et un manuel d'utilisation.

2 Organisation et déroulement

Pour ce projet, nous avons constitué une équipe de 4 personnes : Aurélien Peden (Chef de projet), Loïc Bernard, Antoine Descamps ainsi que Maxime Huyghe. Bien qu'un rôle de chef de projet ait été attribué, nous avons décidé du fait que tout le monde ait plus ou moins le rôle de chef de projet afin d'éviter que le projet soit dirigé par une seule personne et que des décisions soit imposées.

Le choix du projet s'est fait assez rapidement, chaque personne ayant une préférence pour la programmation pure plutôt que la réalisation d'une application. Nous avons donc choisit de développer nos connaissances dans ce domaine et de réaliser un compilateur et tenté d'implémenter notre propre langage simple.

Le projet a donc commencé par l'établissement d'un cahier des charges (disponible en annexe) mentionnant les objectifs du projet à l'aide notamment de la méthode MoSCoW, répartissant chaque point du projet en "Must", "Should," "Could" et "Would", allant donc de la fonctionnalités la plus importantes à la plus facultative pour la mise à bien du projet. Nous avons par ailleurs réalisé une syntaxe primitive du langage LAMA (Loïc, Aurélien, Maxime, Antoine) qui nous a permis de réfléchir aux problématiques d'implémentation d'un compilateur pouvant lire un tel langage tout en essayant de garder un langage assez simple d'accès. Nous avons aussi réalisé un diagramme de Gantt en parallèle à ce cahier des charges, disponible lui aussi en annexe.

Cependant, dû à la complexité du projet, il a été difficile pour nous de concevoir un cahier des charges sans avoir commencé à développer le projet. Plusieurs changement ont donc été effectué, notamment le langage dans lequel a été écrit le compilateur dû aux avis divergents au sein du groupe. Nous avons donc choisit d'utiliser le langage Scala après avoir auparavant essayé d'utiliser Rust puis Ocaml.

Nous avons décidé d'abandonner le langage Rust dû à des préférences personnelles concernant la syntaxe du langage que certain ont trouvé trop complexe, une difficulté venant s'ajouter à la difficulté du sujet du projet. Ocaml a ensuite été écarté à cause du manque de ressources récentes concernant ce langage. Nous avons par la suite décidé d'utiliser le langage Scala, plus populaire et utilisé dans des domaines en pleine croissance tels que le

Big Data[1], et disposant par ailleurs de fonctionnalités présentes dans les langages cités précédemment tels que le pattern matching très utile pour la manipulation d'arbres, qui est au centre de notre projet.

Afin de faciliter la collaboration, nous avons utilisé des outils tels que Trello (tableau kanban) et git, cependant nous avons plus ou moins délaissé Trello par la suite, le projet n'évoluant dans une direction précise ce qui a rendu difficile la définition de tâches précises. Nous avons aussi décidé dès le début de ne pas nous imposer de méthode de type agile avec une organisation par sprint, car nous n'aurions pas su diviser notre projet correctement. Pour ce qui est du développement lui-même, le TDD[2] a été utilisé, c'est à dire que les tests étaient écrits simultanément au code auquel ils se rapportent pour éviter les bugs non-détectés et augmenter la vitesse de développement en général.

Nous avons tenté de pousser l'utilisation de git en utilisant les branches et les pull requests afin de faciliter le travail en parallèle sur différentes fonctionnalités.

Une grande partie du temps a été consacré à l'apprentissage du fonctionnement d'un compilateur et de ses différents composants qui seront décrit par la suite, c'est par ailleurs l'une des plus grandes difficultés rencontrée au cours de ce projet ce qui a conduit à plusieurs problèmes d'organisations.

2.1 Problèmes organisationnels

Plusieurs problèmes d'organisation ont été rencontrés, notamment concernant les deadlines ayant été fixée sur le diagramme de Gantt qui n'ont pas pu être respectées à cause du retard accumulé pour la réalisation du projet.

Il a ensuite été très difficile pour la plupart du groupe de réaliser ce projet dû à l'écart majeur de niveau au sein du groupe et le fait qu'une partie du groupe avance beaucoup plus lentement.

1. Volume massif de données + analyse de ces données
2. Test Driven Development ou développement piloté par les tests

3 Le langage LAMA

Le langage LAMA [1] est un langage de programmation simple, impératif, et inspiré par le C dans sa syntaxe et ses sémantiques.

3.1 Fonctionnalités

LAMA dispose des fonctionnalités de base que l'on peut attendre d'un langage de programmation impératif :

- des variables auxquelles on peut assigner de nouvelles valeurs, et qui peuvent être typées (le typage n'est cependant pas encore vérifié) ;
- une instruction `if` qui accepte uniquement des booléens en tant que condition ;
- une instruction `else` ;
- une instruction `while` qui, comme le `if`, accepte uniquement des booléens ;
- une instruction `for` qui, en plus d'un booléen, prend 2 assignations de variables ;
- la définition de procédures ;
- la définition de fonctions ;
- l'appel de procédures et de fonctions avec des arguments ou non ;
- des opérateurs mathématiques et logiques (cf. figure 3.1) ;
- des tableaux avec les fonctions de manipulation de base (push, pop, get, set, remove) ;
- la libération automatique de la mémoire.

Le langage ne dispose cependant pas d'un tas à proprement parler et le programmeur ne gère donc pas les allocations et libérations de mémoire.

3.1.1 Types

LAMA dispose de trois types primitifs : des entiers, des booléens, et des chaînes de caractères. Il est également possible de stocker ces valeurs dans des tableaux. L'inclusion de types utilisateurs comme des structs ou des classes a été considérée mais pas réalisée par soucis de simplicité.

Les types ne sont pas vérifiés statiquement mais le système de type le permet tout à fait

1. Loïc Aurélien Maxime Antoine...

3.1.2 Opérateurs

Les opérateurs disponibles sont les suivants, leur sémantique étant standard :

Opérateur	Sémantique	Priorité
`*`	multiplication	1
`/`	division entière	
`%`	reste de la division entière	
`+`	addition	2
`-`	soustraction	
`==`	égalité	3
`!=`	inégalité	
>=	supériorité	
<=	infériorité	
>	supériorité stricte	
<	infériorité stricte	
`&&`	et booléen	4
`||`	ou booléen	

FIGURE 3.1 – Opérateurs du langage LAMA

3.2 Grammaire

On présente ici la grammaire formelle BNF [2] du langage LAMA, en supposant que toutes les lignes vides ont été supprimées :

⟨*program*⟩ : := ⟨*line*⟩ ⟨*program*⟩ | ⟨*line*⟩

⟨*line*⟩ : := ⟨*statement*⟩ ⟨*newline*⟩ | ⟨*expression*⟩ ; ⟨*newline*⟩

⟨*newline*⟩ : := \n | \n\r

⟨*statement*⟩ : := ⟨*return*⟩ ; | ⟨*if*⟩ | ⟨*while*⟩ | ⟨*for*⟩ | ⟨*funDef*⟩ | ⟨*declaration*⟩ ;
| ⟨*assignment*⟩ ;

⟨*declaration*⟩ : := ⟨*nameSegment*⟩ : ⟨*name*⟩

⟨*assignment*⟩ : := ⟨*name*⟩ = ⟨*expression*⟩

⟨*block*⟩ : := { ⟨*program*⟩ }

2. Backus-Naur Form ou forme de Backus-Naur

⟨*if*⟩ : := if ⟨*expression*⟩ ⟨*block*⟩ else ⟨*block*⟩ | if ⟨*expression*⟩

⟨*while*⟩ : := while ⟨*expression*⟩ ⟨*block*⟩

⟨*for*⟩ : := for ⟨*for-assignment*⟩ ; ⟨*for-expression*⟩ ; ⟨*for-assignment*⟩ ⟨*block*⟩

⟨*for-expression*⟩ : := ⟨*expression*⟩ | rien

⟨*for-assignment*⟩ : := ⟨*assignment*⟩ | rien

⟨*funDef*⟩ : := fn ⟨*name*⟩ (⟨*paramList*⟩) ⟨*block*⟩
| fn ⟨*name*⟩ (⟨*paramList*⟩) : ⟨*name*⟩ ⟨*block*⟩

⟨*paramList*⟩ : := ⟨*declaration*⟩ ⟨*paramList-tail*⟩ | rien

⟨*paramList-tail*⟩ : := , ⟨*declaration*⟩ ⟨*paramList-tail*⟩ | rien

⟨*funCall*⟩ : := ⟨*name*⟩ (⟨*argList*⟩)

⟨*argList*⟩ : := ⟨*name*⟩ ⟨*argList-tail*⟩ | rien

⟨*argList*⟩ : := , ⟨*name*⟩ ⟨*argList-tail*⟩ | rien

⟨*return*⟩ : := return ⟨*expression*⟩

⟨*name*⟩ : := ⟨*nameSegment*⟩ ⟨*name*⟩ | rien

⟨*nameSegment*⟩ : := regex(a-z_A-Z) ⟨*nameSegment-tail*⟩ | rien

⟨*nameSegment-tail*⟩ : := regex(a-z_A-Z0-9) ⟨*nameSegment-tail*⟩ | rien

⟨*literal*⟩ : := intLit | strLit | boolLit

⟨*intLit*⟩ : := negativeInt | regex(0-9) ⟨*intLit-tail*⟩

⟨*intLit-tail*⟩ : := regex(0-9) ⟨*intLit-tail*⟩ | rien

⟨*negativeInt*⟩ : := - regex(0-9) ⟨*intLit-tail*⟩

⟨*strLit*⟩ : := quote | ⟨*strLit-tail*⟩

⟨*strLit-tail*⟩ : := regex([^quote]) | quote

⟨*boolLit*⟩ : := true | false

⟨*expression*⟩ : := ⟨*comparison*⟩ ⟨*expression-tail*⟩ | ⟨*comparison*⟩

⟨*expression-tail*⟩ : := && ⟨*comparison*⟩ ⟨*expression-tail*⟩
| || ⟨*comparison*⟩ ⟨*expression-tail*⟩
| rien

⟨*comparison*⟩ : := ⟨*addition*⟩ ⟨*comparison-tail*⟩ | ⟨*addition*⟩

⟨*comparison-tail*⟩ : := == ⟨*addition*⟩ ⟨*comparison-tail*⟩
| != ⟨*addition*⟩ ⟨*comparison-tail*⟩
| '>=' ⟨*addition*⟩ ⟨*comparison-tail*⟩
| '<=' ⟨*addition*⟩ ⟨*comparison-tail*⟩
| '>' ⟨*addition*⟩ ⟨*comparison-tail*⟩
| '<' ⟨*addition*⟩ ⟨*comparison-tail*⟩

⟨*addition*⟩ : := ⟨*multiplication*⟩ ⟨*addition-tail*⟩ | ⟨*multiplication*⟩

⟨*addition-tail*⟩ : := + ⟨*multiplication*⟩ ⟨*comparison-tail*⟩
| - ⟨*multiplication*⟩ ⟨*addition-tail*⟩

⟨*multiplication*⟩ : := ⟨*simpleExpression*⟩ ⟨*multiplication-tail*⟩
| ⟨*simpleExpression*⟩

⟨*addition-tail*⟩ : := * ⟨*simpleExpression*⟩ ⟨*comparison-tail*⟩
| / ⟨*simpleExpression*⟩ ⟨*addition-tail*⟩
| % ⟨*simpleExpression*⟩ ⟨*addition-tail*⟩

⟨*simpleExpression*⟩ : := ⟨*funCall*⟩ | ⟨*literal*⟩ | ⟨*name*⟩

3.3 Exemples

Étant donné que la conception d'un langage fait partie intégrante de ce projet, l'inclusion d'exemples de code semble pertinente. Voici donc quelques exemples de code en LAMA :

```
i: Int;
for i = 1; i < 100; i = i + 1 {
    if i % 3 == 0 || i % 5 == 0 {
        if i % 3 == 0 {
            print("fizz");
        }
        if i % 5 == 0 {
            print("buzz");
        }
    } else {
        print(i);
    }
    println("");
}

```

FIGURE 3.2 – Fizzbuzz en LAMA

```
t: Array;
t = a_new();
a_push(t, 1);
a_push(t, 2);
a_push(t, 3);

for i = 0; i <= a_len(t); i = i + 1 {
    println(a_get(t, i);
}

```

FIGURE 3.3 – Manipulation de tableaux en LAMA

```
fn add(x: Int, y: Int): Int {
    return x + y;
}
fn abs(x: Int): Int {
    if x > 0 {
        return x;
    } else {
        return 0 - x;
    }
}

println(add(41, abs(0 - 1));

```

FIGURE 3.4 – Fonctions en LAMA

4 Fonctionnement de LAMAC

4.1 Vue d'ensemble

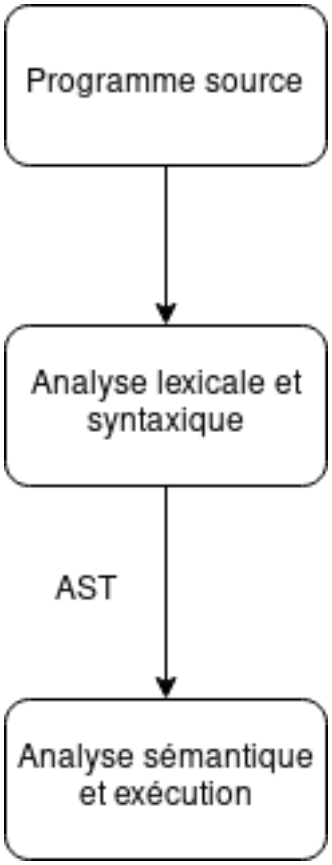

FIGURE 4.1 – Schéma du fonctionnement de LAMAC

L'exécution de LAMAC [1] a lieu en deux grandes phases : on transforme d'abord le programme source en un AST [2] lors de l'analyse lexicale et syntaxique. Ensuite, cet AST est passé à l'interpréteur qui va à la fois exécuter le programme qu'il représente et valider les sémantiques de ce dernier. Ces étapes sont présentées plus en détails dans les sections qui suivent.

1. LAMA Compiler
2. Abstract Syntax Tree ou arbre de syntaxe abstraite

4.2 L'analyse lexicale et syntaxique

Le but de l'analyse lexicale et syntaxique est donc de transformer un programme source en AST. Habituellement, ces deux phases sont séparées, mais il a été décidé de les fusionner par soucis de gain de temps. Si elles avaient été séparées, il aurait d'abord fallu produire des tokens ("mots" du langage) dans la phase lexicale, qui aurait ensuite été transformés en AST.

4.2.1 AST

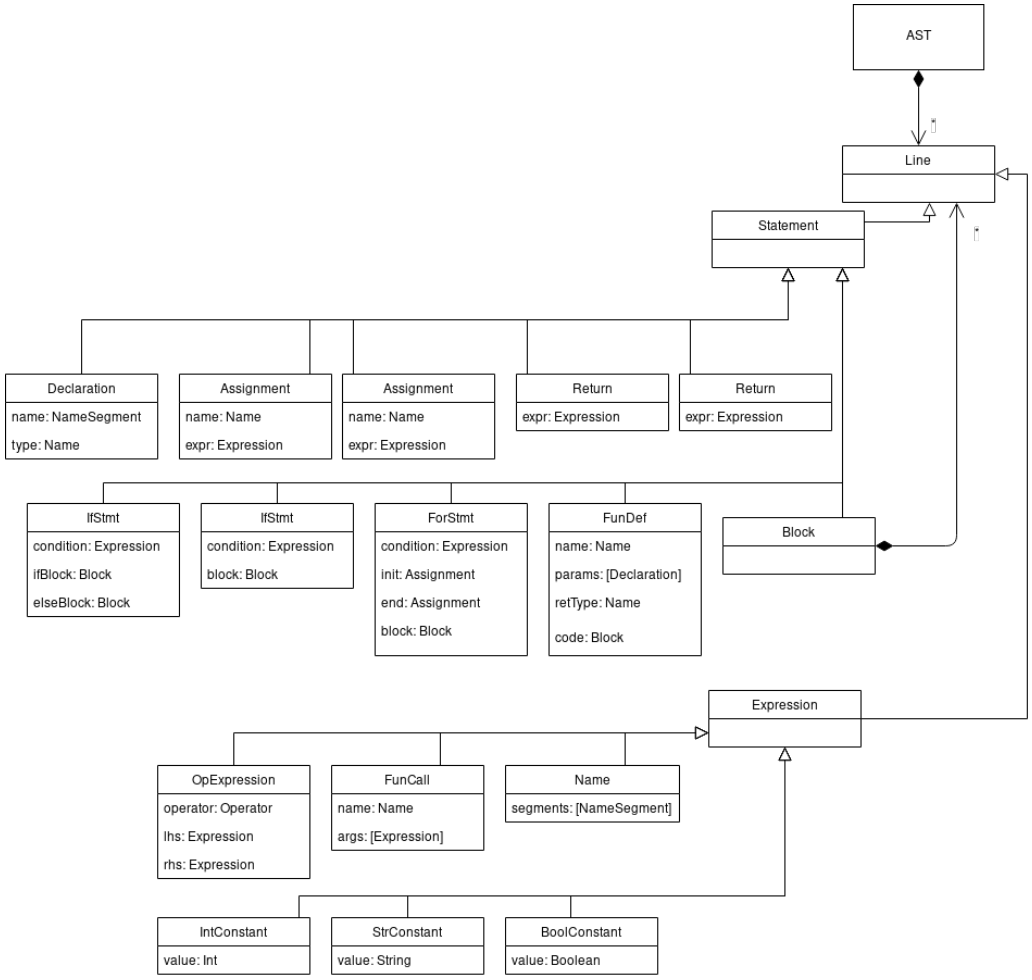

FIGURE 4.2 – Diagramme de classe de l'AST

L'AST est, comme son nom l'indique, un arbre. Il est donc modélisé de sorte qu'une ligne du programme soit soit une instruction (`Statement`) ou

une expression (`Expression`), c'est à dire quelque chose qui produira une valeur une fois évalué.

Sur le même principe, les instructions et expressions peuvent elles-mêmes peuvent prendre beaucoup de formes différentes (cf. figure 4.2).

Il y a aussi un type `Operator`, non représenté sur le diagramme, qui est une simple énumération des différents opérateurs disponibles dans le langage.

Ces différents types n'ont pas de méthode car un style fonctionnel étant utilisé, le filtrage par motif[3] dans une fonction unique est utilisé à la place.

4.2.2 Fonctionnement

Pour réaliser cette analyse, nous utilisons la bibliothèque `fastparse`, qui permet d'écrire un analyseur de façon déclarative, presque comme si l'on écrivait une grammaire formelle. Un analyseur à descente récursive écrit à la main a d'abord été tenté, car cela aurai permis d'avoir un plus grand contrôle sur le processus et de générer de meilleurs messages d'erreur, mais cette idée a été abandonnée faute de temps.

Cet analyseur va donc à la fois vérifier que le programme source se conforme à la grammaire de LAMA et générer l'AST tout en parcourant le code source. Pour cela, `fastparse` permet de déclarer des analyseurs récursifs qui pourront transformer le texte correspondant à la règle de grammaire qu'ils définissent en nœud de l'AST. Le fonctionnement se rapproche donc assez d'un analyseur à descente récursive, mais il y a bien moins de code à écrire. Pour donner un exemple de ce fonctionnement, l'analyseur pour une assignation de variable accepte quelque chose qui match l'analyseur `name` suivi d'un signe = et de quelque chose qui match l'analyseur `expression`. Il récupère ensuite ce qui a été renvoyé par les deux analyseurs qu'il a appelé – un `Name` et un `Expression` respectivement – et construit un `Assignment` qui contient ces deux valeurs.

Pour ce qui est de la gestion des erreurs de syntaxe, c'est celle intégrée à fastparse qui est utilisé, qui produit malheureusement des messages d'erreur peu clairs.

4.3 Exécution d'un programme

Dans cette partie, LAMAC va à la fois exécuter le programme et procéder à une vérification sémantique limitée. Cette vérification sémantique provoque l'arrêt de l'exécution dans les cas suivants :

3. Aussi appelé « pattern matching »

— quand on essaye d'assigner une valeur à une variable non-déclarée ;
— quand on donne un non-booléen a un `if`, `while`, ou `for` ;
— quand on utilise une variable ou fonction qui n'est pas à portée ou qui est non-déclarée ;
— quand on essaye d'appeler un objet qui n'est pas une fonction ;
— quand on appelle une fonction avec le mauvais nombre d'arguments ;
— quand on utilise un opérateur avec les mauvais types ;
— quand on utilise une fonction builtin [4] avec le mauvais nombre d'arguments ou que ces derniers sont du mauvais type
— quand on essaye de récupérer un élément qui n'existe pas dans un tableau

L'exécution du programme en elle-même se déroule comme suit : une fonction `eval` prenant un AST en entrée va le parcourir, ligne par ligne (un AST étant une liste de ligne), tout en appelant la fonction correspondant à la forme de cette ligne grâce au filtrage par motif. Ainsi une fonction `evalExpression` sera appelée s'il s'agit d'une expression, ou `evalStatement` s'il s'agit d'une instruction. Ces fonctions procéderont ensuite de même pour appeler la fonction correspondant au type d'expression ou d'instruction présent.

Pour ce qui est de la gestion de la mémoire, on dispose d'une pile de tables de hachage, chacune correspondant à un scope [5]. À chaque fois que l'interpréteur entre dans un nouveau scope – un bloc ou une fonction – , il empile une nouvelle table, puis la dépile quand il en sort. Pour trouver la valeur associée à un nom, l'interpréteur recherche ce nom dans chaque table en commençant par la plus récemment empilée. Si cette recherche se solde par un échec, il cherche dans une table globale contenant notamment les builtins. Si cette recherche est aussi infructueuse, une erreur est produite, comme mentionné précédement. Les variables et fonctions sont bien sûr insérées dans la table du scope actuel lors de leur déclaration.

4. Intégrée au langage
5. Espace dans lequel un nom sera disponible

5 Résultats et prévisions

5.1 Fonctionnalités

Fonctionnalité prévue	Réalisée
Langage simple	☑
Interprétation	☑
Système de types simple	☑
Système de type statique	☐
Syntaxe simple	☑
Messages d'erreur avancés	☐
Formateur automatique	☐
Ramasse-miette	☑
Bibliothèque standard basique	☑
Compilation rapide	☑
Compilation	☐
Bibliothèque standard avancée	☐

FIGURE 5.1 – Fonctionnalités réalisées

Le système de type statique n'a pas pu être réalisé à cause d'un problème d'organisation dans le groupe.

Les messages d'erreurs avancés et le formateur automatique auraient pris trop de temps.

La compilation et la bibliothèque standard avancée était dés le départ fortement optionnelles et auraient pris trop de temps.

5.2 Temps pris

Globalement, le début du projet a été assez difficile, avec des désaccords quand au langage d'implémentation. Ensuite, non avons eu beaucoup de mal à communiquer et à trouver des horaires concordants, le projet a donc avancé assez lentement, et certaines fonctionnalités on dû être annulées.

L'ordre des événements est détaillé plus en détail dans les diagrammes de Gantt dans l'annexe B (figures B.1 et B.2).

6 Compétences utilisées

Ici sont présentées les compétences qui ont servi pour ce projet, qu'elles soient enseignées en DUT Informatique ou non.

Compétence	Maxime Huyghe	Aurélien Peden	Loïc Bernard	Antoine Descamps
Bases de la programmation	☑	☑	☑	☑
Algorithmie	☑	☑	☑	☑
Types de données	☑	☑	☑	☑
Fonctions	☑	☑	☑	☑
Algèbre booléenne	☑	☑	☑	☑
Lecture de fichiers	☑	☑	☑	☑
Débogage	☑	☑	☑	☑
Récursivité	☑	☑	☑	☑
Structures de données récursives	☑	☑	☑	☑
Théorie des langages	☑	☑	☑	☑
Expressions régulières	☑	☑	☑	☑
Théorie des graphes	☑	☑	☑	☑
Programmation objet	☑	☑	☑	☑
Modélisation objet	☑	☑	☑	☑
Anglais	☑	☑	☑	☑
Gestion de projet	☑	☑	☑	☑
Tableau kanban	☑	☑	☑	☑
Test Driven Development	☑	☑	☑	☑
Tests unitaires	☑	☑	☑	☑
Rédaction d'un cahier des charges	☑	☑	☑	☑
Rédaction d'un rapport	☑	☑	☑	☑
Utilisation de git	☑	☑	☑	☑

FIGURE 6.1 – Compétences apprises en DUT

Compétence	Maxime Huyghe	Aurélien Peden	Loïc Bernard	Antoine Descamps
Programmation fonctionnelle	☑	☑	☑	☑
Pattern matching	☑	☑	☑	☑
Structures d'un compilateur	☑	☑	☑	☑
Utilisation de branches git	☑	☑	☑	☑
Lecture documentation (fastparse)	☑	☑	☑	☐
Fonctionnement parser (fastparse)	☑	☑	☑	☐
Fonctionnement interpreteur	☑	☑	☑	☑

FIGURE 6.2 – Compétences apprises hors-DUT

7 Conclusions personnelles

7.1 Aurélien Peden

Ce projet a été très complexe pour moi, notamment dû à l'écart important en terme de connaissances entre moi et Maxime, écart qui a été de plus en plus difficile à combler et qui s'est accumulé au fil du temps. Ce projet m'a cependant permis d'apprendre un langage fonctionnel, le Scala ainsi que la structure d'un compilateur, chose qui était totalement étrangère pour ma part avant de débuter ce projet. Avec du recul, j'aurais préféré réaliser ce projet en Python comme l'était mon souhait à l'origine, bien qu'une partie du groupe ait été en désaccord avec cette décision, cela m'aurait permis de m'appuyer sur de nombreux tutoriels disponibles sur internet et de ne pas avoir à apprendre la programmation fonctionnelle en plus de la structure d'un compilateur. Il aurait fallu aussi découper le projet en plusieurs deadlines correspondantes chacune à un concept du compilateur (parser, lexer etc.) eux-mêmes décomposés en plusieurs parties (langage simple permettant d'effectuer des opérations basiques : + - *), puis la déclaration de variable, structures de contrôles (if, while, for) etc. Je pense qu'imposer l'utilisation d'un langage plus familier à la plupart des membres du groupe aurait permis au développement de se dérouler avec moins d'encombres, quitte à faire certains compromis par exemple avec l'utilisation du pattern matching propre à un langage fonctionnel.

7.2 Loïc Bernard

Partageant l'avis d'Aurélien, la très grande différence de niveau avec Maxime a grandement augmenté la difficulté du projet. L'écart de niveau m'a également impacté sur le fait que je n'ai presque pas réussi à faire grand chose dans le code. Cependant j'ai quand même appris le fonctionnement d'un compilateur ce qui était très intéressant. Si je pouvais tout refaire, je choisirais plutôt de faire cela en Python dû à l'abondance de tutoriel. Cela m'a permis de mieux définir ce que j'aimerais faire par la suite dans mes études. Comme de la programmation web et moins abstraite qu'un compilateur que je pensais être un projet très intéressant.

7.3 Antoine Descamps

La réalisation de ce projet a été quelque peu houleuse et l'équipe a fait face à de nombreux désaccords quant à la nature du projet lui-même ou quant aux outils à utiliser pour le mener à bien, notamment en ce qui concerne le choix du langage de programmation à utiliser. Ce projet illustre bien la difficulté que peut revêtir la réalisation d'un projet en groupe. Néanmoins Maxime a su motiver et guider le reste du groupe grâce à ses compétences en programmation fonctionnelle, sa connaissance du langage Scala et son expérience dans le développement de compilateurs. Ce projet m'a permis d'en apprendre plus sur l'implantation d'un compilateur et sur les différentes parties qui le composent, et m'a servi d'introduction à la programmation fonctionnelle au travers de la manipulation du langage Scala dont je ne m'étais jamais servi auparavant.

7.4 Maxime Huyghe

La compilation est un sujet qui m'intéresse depuis longtemps, et sur lequel je suis heureux d'avoir pu travailler. Néanmoins, je suis déçu du déroulement de ce projet et ne comprends pas vraiment pourquoi le langage a posé problème, car je pensais que les nouveaux concepts dont j'ai encouragé l'utilisation faciliteraient au contraire la tâche. Une meilleure communication dans le groupe, notamment de ma part, aurait été grandement bénéfique, et mes horaires qui ne correspondaient pas forcément avec ceux des autres membres du groupe rajoutaient de la difficulté sur ce plan.

8 Conclusion générale

Ce projet a été enrichissant pour nous tous et nous a permis de découvrir ou approfondir de nouveaux sujets tels que la compilation ou la programmation fonctionnelle. Nous avons cependant fait face à de nombreuses difficultés à cause notamment de disparités dans les profils, de désaccords sur certains points, et d'horaires différents, et avons pris conscience de la difficulté d'organiser un projet de longue durée en équipe : la communication est primordiale. Cela a donc été une expérience globalement positive pour toute l'équipe, qui nous servira très probablement dans la suite de nos études et dans notre vie professionnelle.

Annexes

A Cahier de charges

Cahier des charges – Langage de programmation éducatif et compilateur

Pour ce projet, nous allons concevoir un langage de programmation à visée éducative et réaliser un compilateur/interpréteur pour ce langage.

Analyse fonctionnelle

Échelle MoSCoW : **M**ust **S**hould **C**ould **W**ould

M Le langage sera raisonablement facile à utiliser.

M Le langage pourra être interprété.

M Le langage aura un système de type statique plutôt simple.

M Le langage aura une syntaxe simple, avec du sucre syntaxique en modération.

Example :

```
fun f(x, y) is
    y := x + 1
    return x + y
end

while a < 10 do
    the_thing()
end

for i in 0..10 do
    print(i)
end

foo(bar) = bar.foo()
```

S Le compilateur produira des messages d'erreur très utiles (type Rust ou Elm)

S Le compilateur incluera un formateur automatique, les débutants ayant tendance à écrire du code difficile à lire.

S La mémoire sera gérée par un ramasse-miette.

S Le langage disposera d'une bibliothèque standard basique. Tableaux (+ strings), Autres types de base, IO console.

C L'utilisateur devra pouvoir exécuter son code rapidement, la compilation /pré-interprétation devra donc être rapide.

C Le langage pourra être compilé (bytecode, C, ou asm).

W Le langage disposera d'une bibliothèque standard assez remplie. IO fichiers, Réseau , Multithreading.

Analyse technique

Nous nous inspirerons de techniques d'écriture de compilateur existantes ainsi que de livres sur le sujet afin de ne pas tout réinventer.

Le compilateur sera écrit en Rust, un langage un peu difficile à apprendre, mais qui dispose des avantages suivants : - Un système de type puissant très propice à l'écriture de compilateurs - Des exécutables très rapides (niveau C[++]) tout en étant plutôt haut niveau

Nous écrirons probablement un parseur descendant récursif à la main.

La structure générale sera : Source =Lexing=> Tokens =Parsing=> Arbre syntaxique (=Opti=> Arbre syntaxique)* Ensuite soit interprétation de l'arbre, soit écriture de C/assembleur/bytecode

Organisation et méthode

Il a été difficile de s'organiser puisqu'aucun de nous ne sait exactement comment développer un langage de programmation. Nous avons donc dû nous renseigner sur différents sites, comme :

— How would I go about creating a programming language? ;

— ou encore I wrote a programming language. Here's how you can, too.. . .

Après que nous nous sommes renseignés, nous avons pu prendre quelques décisions. Le langage que nous allions créer devait être interprété et non pas compilé, car il nous semblait que ce serait plus simple et que le projet était déjà suffisamment complexe en l'état. L'interprétation est un procédé moins efficace mais est plus simple à mettre en place. S'il nous reste du temps une fois l'interpréteur développé, peut-être que nous tenterons de compiler le langage en C ou en bytecode Java.

Nous avons choisi d'utiliser le langage Rust sur les recommandations de Maxime qui avait déjà un peu d'expérience dans l'écriture de compilateurs et d'interpréteurs. Nous avons hésité avec le langage Python qui présente des fonctionnalités de parsing et de traitement de chaînes de caractères avancées.

Nous allons continuer de consulter différentes sources (youtube, livres, tutoriels, ...) afin d'en apprendre plus sur le développement d'un langage de programmation. Il est difficile d'établir un planning précis puisque comme dit précédemment nous en savons peu sur la manière dont il faut s'y prendre,

et que nous ne pouvons pas nous aider de certaines notions apprises en cours comme par exemple UML, les diagrammes de classes, etc... Nous avons malgré tout essayé d'établir un planning et nous avons dressé un diagramme de Gantt pour établir globalement les tâches à réaliser.

Nous avons décidé que d'ici mi-janvier il faudrait qu'on dispose d'un langage, aussi simple soit-il, pouvant être interprété grâce à un programme en Rust. A partir de là il nous sera plus facile d'explorer de nouvelles pistes et de mieux définir le langage en lui ajoutant des fonctionnalités.

B Diagrammes de GANTT

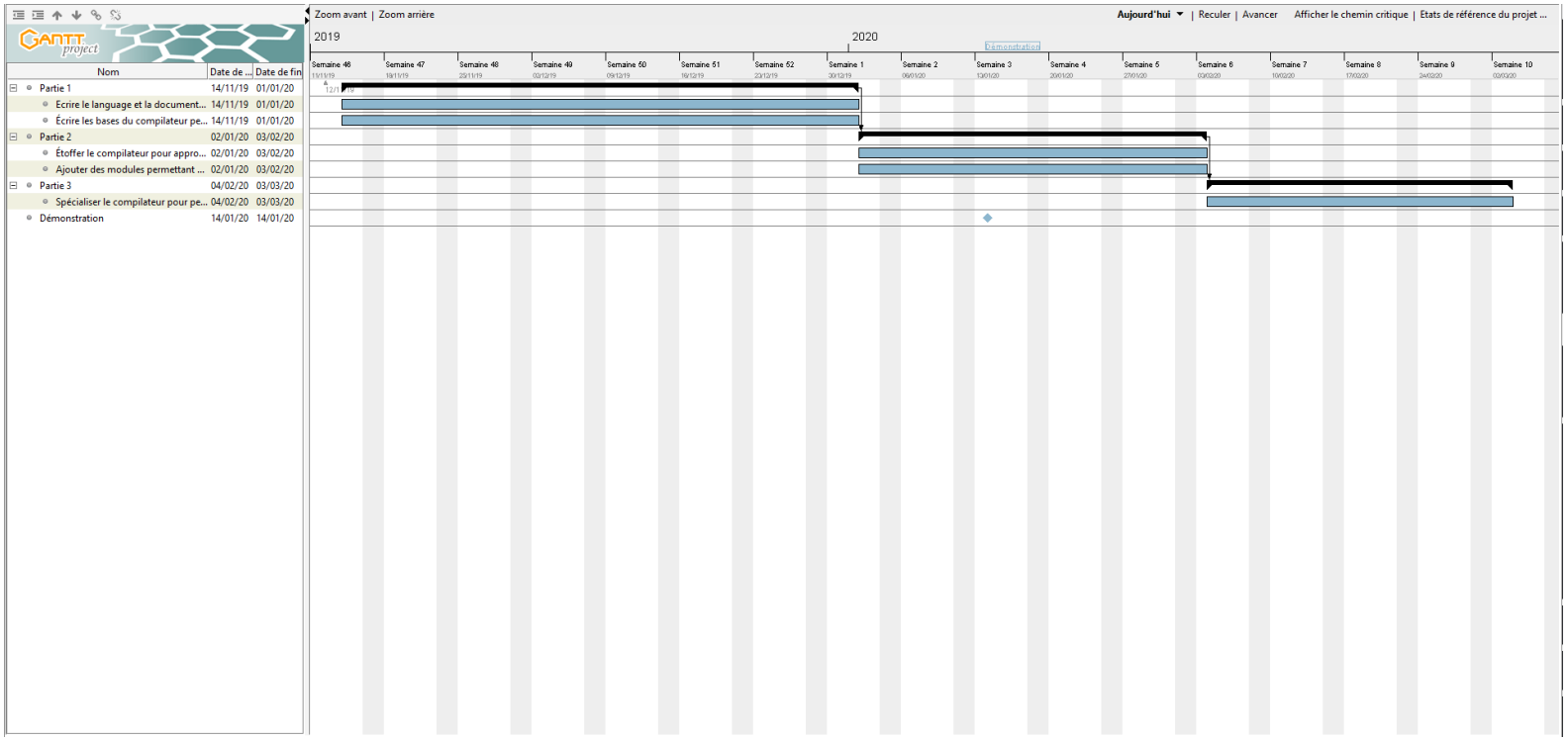

FIGURE B.1 – Diagramme de Gantt prévisionnel

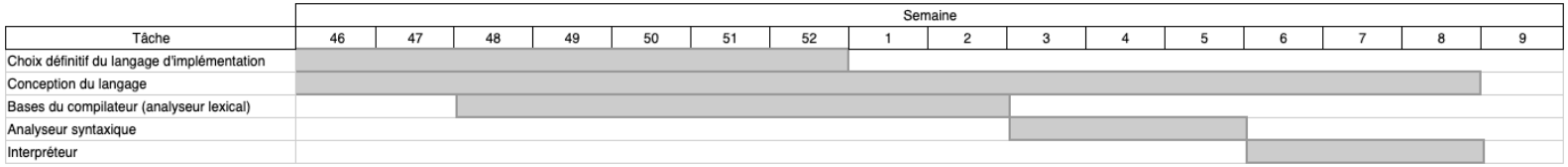

FIGURE B.2 – Diagramme de Gantt effectif

C Lexique

— Big data : gestion de volumes massifs de données combinée à leur exploitation ;
— TDD : Test Driven Dévelopment : l'écriture de tests guide le développement ;
— LAMA : Loïc Aurélien Maxime Antoine ;
— LAMAC : LAMA Compiler ;
— BNF : Backus-Naur Form, une syntaxe pour écrire une grammaire formelle ;
— AST : Abstract Syntax Tree, une représentation interne à un compilateur/interpréteur d'un programme ;
— Filtrage par motif : une technique de programmation permettant de choisir le code à exécuter en fonction du type des données et d'extraire celles-ci de leur conteneur ;
— Scope : en programmation, la portée d'accessibilité d'une variable ;
— Builtin : souvent utilisé pour parler des fonctions et types intégrés à un langage de programmation.

D Ressources utilisées

Logiciels et outils

Les logiciels notables que nous avons utilisés sont :
— scalac, le compilateur scala ;
— sbt, le gestionnaire de projet scala ;
— la JVM, qui nous a permis d'exécuter notre code scala ;
— IntelliJ IDEA, un environnement de développement avec un bon support pour le scala ;
— git, pour versionner notre code et l'écrire en parallèle ;
— Trello, pour avoir une vue d'ensemble des tâches à faire ;
— LaTeX et Overleaf, pour la réalisation de ce rapport ;
— draw.io, pour certains diagrammes ;
— Gantt project, pour le premier diagramme de Gantt.

Ouvrages

Crafting interpreters, Bob Nystrom

E Manuel

Pour utiliser LAMAC, il faut d'abord le compiler. Pour cela, vous aurez besoin de sbt, pour lequel pourrez trouver des instructions d'installation sur le site de scala. Ensuite, une fois dans le répertoire contenant les sources de LAMAC (la racine du dépôt git), lancez la commande `sbt`. Après quelques instants, LAMAC devrait être compilé et une invite de commande devrait s'afficher. Il suffit alors de lancer la commande `run <nom-du-fichier.lama` pour exécuter un programme écrit en LAMA.