Projet de compilation - Partie 1

Définition complète de la grammaire du langage, conception des analyseurs lexical et syntaxique, construction de l'arbre abstrait, visualisation.

Rapport d'activité

Ugo Birkel
Léo FORNOFF
Matthias Germain
Elise Neyens

Responsable de module : Suzanne Collin

18 Octobre 2023 - 13 décembre 2024

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

Ce document présente la première partie du projet de compilation effectué dans le cadre des modules PCL1 de la 2ème année à TELECOM Nancy, d'octobre 2023 à janvier 2024. L'objectif principal est de développer un compilateur pour le langage canAda, une version simplifiée du langage Ada.. Au cours de cette période, notre équipe composée de Ugo Birkel, Léo FORNOFF, Matthias Germain, et Elise Neyens, a réalisé à les étapes de création du processus de compilation qui sont les suivantes : définition de la grammaire, la conception et l'implémentation des analyseurs lexical et syntaxique, ainsi que la construction de l'arbre abstrait.

Le rapport qui suit documente ces étapes. Celle-ci représentent les jalons essentiels sur lesquels reposera l'intégralité de notre projet de compilation. L'adoption stratégique de Java comme langage de programmation, justifiée par ses avantages polyvalents et sa connaissance par l'équipe, qui s'est donc tourné vers intelliJ comme IDE pour réaliserz ce projet.

En poursuivant votre lecture, vous découvrirez les choix, les développements réalisés, ainsi que les résultats obtenus au cours de cette phase initiale. Ce rapport consiste à fournir une compréhension de notre progression dans la réalisation du compilateur canAda.

Chapitre 2

Gestion du projet

Notre démarche dans la réalisation du projet de compilation a été encadrée par une gestion de projet efficace, assurant un suivi de nos avancées. Dans cette section, nous mettons en lumière les principaux éléments de notre gestion de projet, dont les réunions régulières, les comptes rendus (CR) associés, les stand-up meetings, ainsi que le diagramme de Gantt.

2.1 Réunions et Comptes Rendus

Pour maintenir une communication au sein de l'équipe, des réunions ont été tenues. Un modèle type de réunion a été établi, observable dans les Annexes. . Ces réunions ont offert l'opportunité de discuter des avancements individuels, de résoudre les éventuels blocages, et de planifier les prochaines étapes. Les comptes rendus détaillés de chaque réunion sont annexés à ce rapport, fournissant une traçabilité de nos échanges et décisions.

2.2 Stand-Up Meetings

Des stand-up meetings courts et fréquents ont également été intégrés à notre routine de gestion de projet. Ces réunions rapides, ont permis à chaque membre de l'équipe de partager brièvement son état d'avancement et d'exprimer d'éventuelles difficultés. Cette approche agile a favorisé une collaboration fluide et a contribué à identifier rapidement les points de blocage.

2.3 Diagramme de Gantt

Afin de visualiser l'évolution temporelle de notre projet, un diagramme de Gantt a été élaboré. Celui-ci ne dépeint pas le projet entièrement, mais nous a permis de faire les premiers pas dans notre projet sans perdre de vue les objectifs à moyen terme.

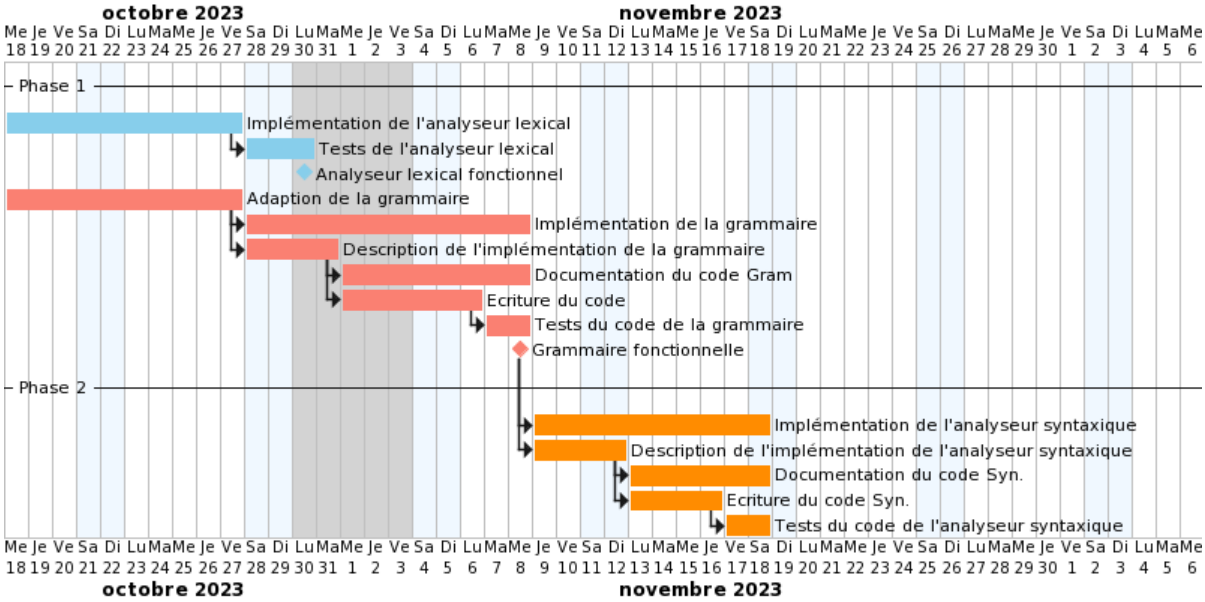


FIGURE 2.1 – Diagramme de Gantt

Chapitre 3

Grammaire

3.1 Présentation de la grammaire canAda

La grammaire présentée ci-dessous est celle d'origine. Après observation, nous avons constaté qu'elle ne satisfaisait pas les conditions d'une grammaire LL(1). En réponse à cette constatation, nous avons entrepris diverses méthodes afin d'éliminer les caractéristiques qui rendaient la grammaire non LL(1).

```
⟨fichier⟩    ::= with Ada.Text_IO; use Ada.Text_IO;
                 procedure ⟨ident⟩ is ⟨decl⟩*
                 begin ⟨instr⟩+ end ⟨ident⟩? ; EOF
⟨decl⟩       ::= type ⟨ident⟩ ;
               | type ⟨ident⟩ is access ⟨ident⟩ ;
               | type ⟨ident⟩ is record ⟨champs⟩+ end record ;
               | ⟨ident⟩+, : ⟨type⟩ (:= ⟨expr⟩)? ;
               | procedure ⟨ident⟩ ⟨params⟩? is ⟨decl⟩*
                 begin ⟨instr⟩+ end ⟨ident⟩? ;
               | function ⟨ident⟩ ⟨params⟩? return ⟨type⟩ is ⟨decl⟩*
                 begin ⟨instr⟩+ end ⟨ident⟩? ;
⟨champs⟩     ::= ⟨ident⟩+, : ⟨type⟩ ;
⟨type⟩       ::= ⟨ident⟩
               | access ⟨ident⟩
⟨params⟩     ::= ( ⟨param⟩+; )
⟨param⟩      ::= ⟨ident⟩+, : ⟨mode⟩? ⟨type⟩
⟨mode⟩       ::= in | in out

⟨expr⟩       ::= ⟨entier⟩ | ⟨caractère⟩ | true | false | null
               | ( ⟨expr⟩ )
               | ⟨accès⟩
               | ⟨expr⟩ ⟨opérateur⟩ ⟨expr⟩
               | not ⟨expr⟩ | - ⟨expr⟩
               | new ⟨ident⟩
               | ⟨ident⟩ ( ⟨expr⟩+, )
               | character ' val ( ⟨expr⟩ )
⟨instr⟩      ::= ⟨accès⟩ := ⟨expr⟩ ;
               | ⟨ident⟩ ;
               | ⟨ident⟩ ( ⟨expr⟩+, ) ;
               | return ⟨expr⟩? ;
               | begin ⟨instr⟩+ end ;
               | if ⟨expr⟩ then ⟨instr⟩+ (elsif ⟨expr⟩ then ⟨instr⟩+)*
                 (else ⟨instr⟩+)? end if ;
               | for ⟨ident⟩ in reverse? ⟨expr⟩ .. ⟨expr⟩
                 loop ⟨instr⟩+ end loop ;
               | while ⟨expr⟩ loop ⟨instr⟩+ end loop ;
⟨opérateur⟩  ::= = | /= | < | <= | > | >=
               | + | - | * | / | rem
               | and | and then | or | or else
⟨accès⟩      ::= ⟨ident⟩ | ⟨expr⟩ . ⟨ident⟩
```

FIGURE 3.1 – Grammaire d'origine

3.2 Étapes pour rendre la grammaire LL(2)

Dans le cadre de l'optimisation de notre grammaire pour une analyse LL(2), nous avons suivi plusieurs étapes, en suivant le cours de théorie dans langaes vu en 1ère année à Telecom Nancy.

1. **Dérécursivisation Immédiate puis Indirecte :** La première étape a impliqué la suppression de la récursivité immédiate, suivie de l'élimination de la récursivité indirecte.
2. **Élimination des Factorisations :** Nous avons identifié les factorisations à gauche dans la grammaire et appliqué des techniques pour les éliminer, simplifiant ainsi les règles et réduisant la complexité.
3. **Élimination des Factorisations Indirectes :** L'élimination des factorisations indirectes a été réalisée de la même manière.
4. **Modification et Simplification Progressives :** Au fur et à mesure de notre travail, nous avons continué à modifier et simplifier la grammaire pour essayer de résoudre les conflits qui restait présent dans la grammaire.

Problème Type Rencontré dans la Grammaire : Nous avons identifié un problème spécifique dans la grammaire. L'élément OPERATOR se trouve à la fois dans Premiers(E1) et dans Suivants(A1), alors que Suivants(A) est inclus dans Suivants(A1) et contient également OPERATOR. Cette situation a généré des conflits entre Premiers(E1) et Suivants(A), avec OPERATOR présent dans les deux ensembles.

5.**Tentatives d'Implémentation de Prédicats :** Nous avons initialement envisagé l'ajout de prédicats pour résoudre des conflits. Cependant, cette approche a été écartée en raison de sa complexité par rapport aux autres solutions envisagées.

En suivant ces étapes, nous avons réussi à obtenir une grammaire LL(2), prête à être intégrée dans notre compilateur.

3.3 Présentation de la grammaire obtenue

Après avoir identifié et résolu les défauts, nous avons abouti à une version conforme à une analyse LL(2). Cette évolution permet une interprétation plus efficace des expressions canAda, simplifiant ainsi le processus d'analyse syntaxique pour notre projet de compilation.

⟨Axiome⟩ → with Ada.Text_IO ; use Ada.Text_IO ; procedure ident is ⟨DeclarationStar⟩
  begin ⟨Instruction⟩ ⟨InstructionPlus⟩ end ⟨Identificator0or1⟩ ;
⟨Identificator0or1⟩ → ident
  | ε
⟨DeclarationStar⟩ → ⟨Declaration⟩⟨DeclarationStar⟩
  | ε
⟨Declaration⟩ → ⟨Type⟩ ident ⟨DeclarationFact1⟩
  | ident ⟨IdentificatorCommaPlus⟩ : ⟨Type⟩ ⟨ExpressionAssignement⟩ ;
  | procedure ident ⟨Parameters0or1⟩is⟨DeclarationStar⟩"begin" ⟨Instruction⟩⟨InstructionPlus⟩
  end⟨Identificator0or1⟩ ;
  | function ident ⟨Parameters0or1⟩ return ⟨Type⟩ is ⟨DeclarationStar⟩ begin ⟨Instruction⟩
  ⟨InstructionPlus⟩ end ⟨Identificator0or1⟩ ;
⟨DeclarationFact1⟩ → ;
  | is ⟨DeclarationFact2⟩
⟨DeclarationFact2⟩ → access ident ;
  | record ⟨Field⟩ ⟨FieldPlus⟩ end record ;
⟨Field⟩ → ident ⟨IdentificatorCommaPlus⟩ :⟨Type⟩ ;
⟨IdentificatorCommaPlus⟩ → , ident ⟨IdentificatorCommaPlus⟩
  | ε
⟨FieldPlus⟩ → ⟨Field⟩ ⟨FieldPlus⟩
  | ε
⟨Method⟩ → in ⟨MethodFact⟩
⟨MethodFact⟩ → out
  | ε
⟨Parameters⟩ → (⟨Parameter⟩ ⟨ParametersSemicolonPlus⟩)
⟨ParametersSemicolonPlus⟩ → ; ⟨Parameter⟩ ⟨ParametersSemicolonPlus⟩
  | ε
⟨Parameters0or1⟩ → ⟨Parameters⟩
  | ε
⟨Parameter⟩ → ident ⟨IdentificatorCommaPlus⟩ : ⟨MethodOor1⟩ ⟨Type⟩

⟨Instruction⟩ → entier ⟨ExpressionFollow⟩ := ⟨Expression⟩ ;
| char ⟨ExpressionFollow⟩ := ⟨Expression⟩ ;
| true ⟨ExpressionFollow⟩ := ⟨Expression⟩ ;
| false ⟨ExpressionFollow⟩ := ⟨Expression⟩ ;
| null ⟨ExpressionFollow⟩ := ⟨Expression⟩ ;
| float ⟨ExpressionFollow⟩ := ⟨Expression⟩ ;
| (⟨Expression⟩) ⟨ExpressionFollow⟩ := ⟨Expression⟩ ;
| not ⟨Expression⟩⟨ExpressionFollow⟩ := ⟨Expression⟩ ;
| new ⟨Expression⟩⟨ExpressionFollow⟩ := ⟨Expression⟩ ;
| - ⟨Expression⟩⟨ExpressionFollow⟩ := ⟨Expression⟩ ;
| character'val (⟨Expression⟩) ⟨ExpressionFollow⟩ := ⟨Expression⟩
| return ⟨Expression0or1⟩ ;
| begin ⟨InstructionPlus⟩ end ;
| if ⟨Expression⟩ then ⟨Instruction⟩⟨InstructionPlus⟩
⟨Elseif⟩ else ⟨Else1⟩ end if ;
| for ⟨Ident⟩ in reverse ⟨Expression⟩ to ⟨Expression⟩ loop
⟨Instruction⟩⟨InstructionPlus⟩ end loop ;
| while ⟨Expression⟩ loop ⟨Instruction⟩ ⟨InstructionPlus⟩ end loop ;

⟨InstructionExpressionAssignment⟩ → ⟨ExpressionFollow⟩ := ⟨Expression⟩ ;
|(⟨Expression⟩⟨ExpressionCommaPlus⟩)⟨InstructionAssignement⟩

⟨InstructionAssignment⟩ → ;
| ⟨ExpressionFollow⟩. Ident := Expression ;

⟨InstructionPlus⟩ → ⟨Instruction⟩ ⟨InstructionPlus⟩
| ε

⟨Expression⟩ → entier ⟨ExpressionFollow⟩
| char ⟨ExpressionFollow⟩
| true ⟨ExpressionFollow⟩
| false ⟨ExpressionFollow⟩
| null ⟨ExpressionFollow⟩
| float ⟨ExpressionFollow⟩
| (⟨Expression⟩) ⟨ExpressionFollow⟩.
| ident ⟨ExpressionFact⟩
| not ⟨Expression⟩ ⟨ExpressionFollow⟩
| - ⟨Expression⟩ ⟨ExpressionFollow⟩
| character'val (⟨Expression⟩) ⟨ExpressionFollow⟩

⟨ExpressionFollow⟩ → ⟨Operator⟩ ⟨Expression⟩ ⟨ExpressionFollow⟩
| . ident ⟨ExpressionFollow⟩
| ε

⟨ExpressionCommaPlus⟩ → , ⟨Expression⟩ ⟨ExpressionCommaPlus⟩
| ε

⟨ExpressionAssignment⟩ → := ⟨Expression⟩
| ε

⟨ExpressionFact⟩ → ⟨ExpressionFollow⟩
|(⟨Expression⟩⟨ExpressionCommaPlus⟩)⟨ExpressionFollow⟩
⟨Expression0or1⟩ → ⟨Expression⟩
| ε
⟨Elseif⟩ → elseif ⟨Expression⟩ then ⟨Instruction⟩ ⟨InstructionPlus⟩ ⟨Elseif⟩
| ε
⟨Else1⟩ → else ⟨Instruction⟩ ⟨InstructionPlus⟩
| ε
⟨Reverse⟩ → reverse
| ε
⟨Type⟩ → ident
|access ident

3.4 Implémentation de la grammaire

Après avoir défini la grammaire optimisée pour le langage canAda, nous avons entrepris son implémentation dans le cadre de notre projet de compilation.

Nous avons organisé notre implémentation de la grammaire canAda en plusieurs packages distincts, chacun avec des responsabilités spécifiques. La structure globale de notre architecture est résumée dans le diagramme UML suivant :

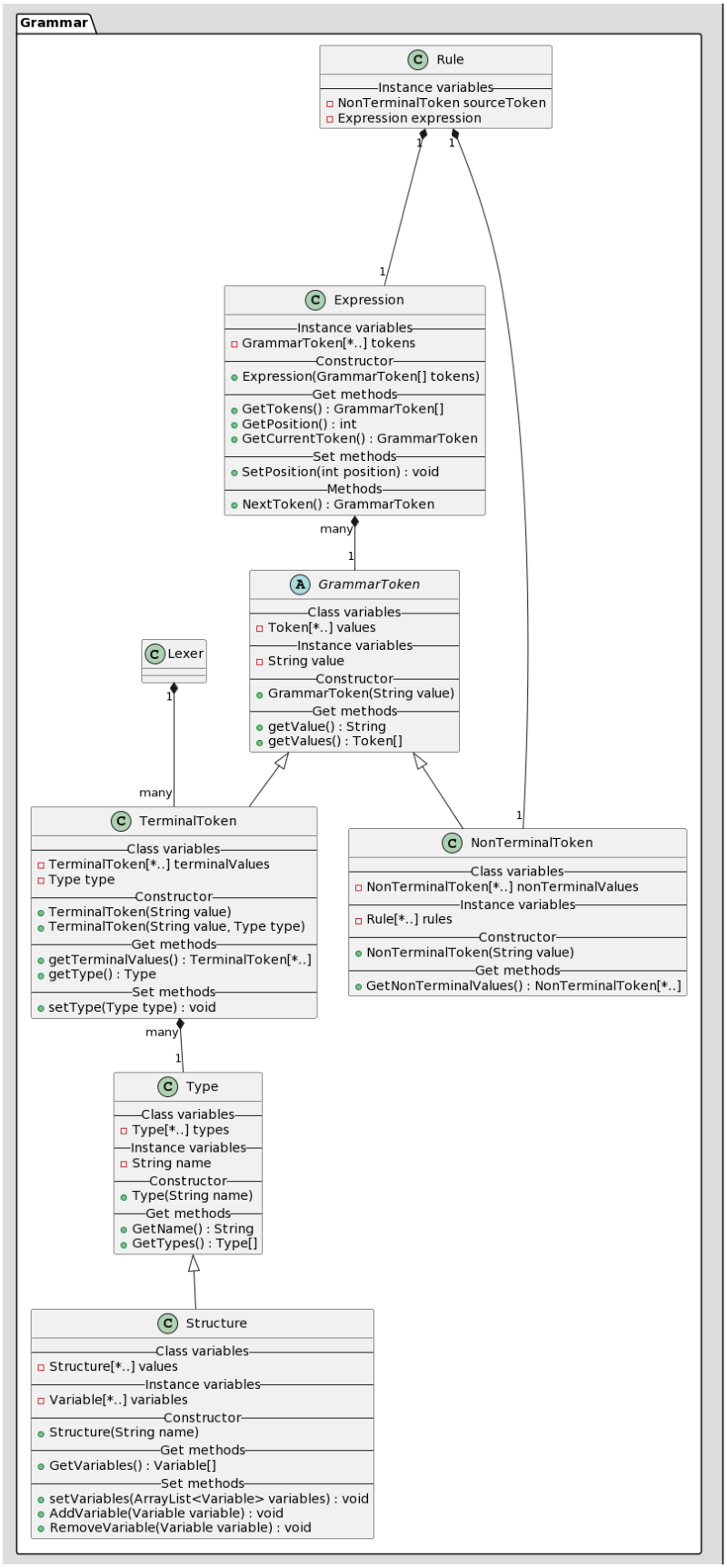


FIGURE 3.2 – Conception de la grammaire

Chapitre 4

Analyseur lexical

4.1 Conception - Architecture et Tokens

Lors de la conception de l'analyseur lexical, nous avons décidé de mettre en place deux composants principaux jouant chacun un rôle essentiel dans le traitement de fichier texte correspondant à du code écrit en Ada. Le premier composant, contenu dans le fichier FileReaderUtil.java sert de lecteur de fichier ligne par ligne, il utilise pour cela un BufferedReader pour la lecture et un StringBuilder pour le stockage. Dans un second temps, le fichier Lexer.java représentant le Lexer utilise le texte fourni par "FileReaderUtil" pour effectuer un parcours caractère par caractère. Différentes méthodes ont étés implémentées pour identifier les différents types de tokens présents dans un code Ada. On retrouve notamment les identifiants, les opérateurs, les mots-clés ou encore les ponctuation.

Ce lexeur reconnaît et peut catégoriser différents types de tokens : Les mots-clés (mots spécifiques au langage comme 'if' 'else' ...) Les identifiants qui sont spécifiques au code donné au parseur Les entiers, les flottants, les opérateurs ou encore les symboles de ponctuation comme les '(', ';'.

4.2 Implémentation

Nous allons aborder dans cette partie plus en détails l'implémentation du lexeur et les méthodes mises en place afin de pouvoir analyser un code Ada soumis au format texte. La méthode tokenize() permet de renvoyer un ArrayList<TerminalToken> à partir du code Ada en texte en effectuant des méthodes caractère par caractère. La première effectuée lors de la boucle est skipWhiteSpace qui permet de d'ignorer les espaces. skipOneLineComment() permet ensuite d'ignorer les commentaires qui n'ont aucun intérêt pour l'analyse syntaxique. C'est ensuite au tour des symboles de ponctuation, des opérateurs, des chaînes de caractères, des nombres et des mots-clés d'être reconnus. Lorsqu'un token est reconnu par le lexeur, il y a création d'un TerminalToken à l'aide de la méthode newTerminalToken().

4.3 Exemple de codes de test et optimisations

Si on teste le lexeur sur le code test suivant :

```
1  with Ada.Text_IO; use Ada.Text_IO;
2  procedure Fibonacci is
3      -- on va implementer fibonacci
4      N : Integer;
5      A : Integer;
6      B : Integer;
7      Next : Integer;
8  begin
9      A := 0;
10     B := 1;
11     Get(N);
12     for I in 1..N loop
13         if I <= 1 then
14             Next := I;
15         else
16             Next := A + B;
17             A := B;
18             B := Next;
19         end if;
20     end loop;
21     New_Line;
22 end Fibonacci;
```

Listing 4.1 – Fibonacci procedure in Ada

On obtient alors la tokenisation suivante :

```
1
2  with                 KEYWORD
3  Ada                  IDENTIFIER
4  .                    DOT
5  Text_IO              IDENTIFIER
6  ;                    SEMICOLON
7  use                  KEYWORD
8  Ada                  IDENTIFIER
9  .                    DOT
10 Text_IO              IDENTIFIER
11 ;                    SEMICOLON
12 procedure            KEYWORD
13 Fibonacci            IDENTIFIER
14 is                   KEYWORD
15 N                    IDENTIFIER
16 :                    COLON
17 Integer              IDENTIFIER
18 ;                    SEMICOLON
19 A                    IDENTIFIER
20 :                    COLON
21 Integer              IDENTIFIER
22 ;                    SEMICOLON
23 B                    IDENTIFIER
24 :                    COLON
25 Integer              IDENTIFIER
26 ;                    SEMICOLON
27 Next                 IDENTIFIER
28 :                    COLON
29 Integer              IDENTIFIER
30 ;                    SEMICOLON
31 begin                KEYWORD
32 A                    IDENTIFIER
33 :=                   ASSIGNMENT
34 0                    INTEGER
35 ;                    SEMICOLON
36 B                    IDENTIFIER
37 :=                   ASSIGNMENT
38 1                    INTEGER
39 ;                    SEMICOLON
40 Get                  IDENTIFIER
41 (                    L_PARENTHESIS
42 N                    IDENTIFIER
43 )                    R_PARENTHESIS
44 ;                    SEMICOLON
45 for                  KEYWORD
46 I                    IDENTIFIER
47 in                   KEYWORD
48 1.                   FLOAT
49 .                    DOT
50 N                    IDENTIFIER
51 loop                 KEYWORD
52 if                   KEYWORD
53 I                    IDENTIFIER
54 <=                   INFERIOR_EQUALS
55 1                    INTEGER
56 then                 KEYWORD
57 Next                 IDENTIFIER
58 :=                   ASSIGNMENT
59 I                    IDENTIFIER
60 ;                    SEMICOLON
61 else                 KEYWORD
62 Next                 IDENTIFIER
63 :=                   ASSIGNMENT
64 A                    IDENTIFIER
65 +                    PLUS
66 B                    IDENTIFIER
67 ;                    SEMICOLON
68 A                    IDENTIFIER
69 :=                   ASSIGNMENT
70 B                    IDENTIFIER
71 ;                    SEMICOLON
72 B                    IDENTIFIER
73 :=                   ASSIGNMENT
74 Next                 IDENTIFIER
75 ;                    SEMICOLON
76 end                  KEYWORD
77 if                   KEYWORD
78 ;                    SEMICOLON
79 end                  KEYWORD
80 loop                 KEYWORD
81 ;                    SEMICOLON
82 New_Line             IDENTIFIER
83 ;                    SEMICOLON
84 end                  KEYWORD
85 Fibonacci            IDENTIFIER
86 ;                    SEMICOLON
```

Listing 4.2 – Fibonacci Ada Code Tokenisé

4.4 Gestion des erreurs

Dans le projet, les erreurs sont gérées par la classe LexerError qui est spécifiquement conçue pour les erreurs qui ont lieu durant l'analyse lexicale. Lors d'une erreur, un message est imprimé en sortie avec le lieu de son apparition. Cela permet de signaler précisément les problèmes rencontrés lors de la reconnaissance des tokens et de pouvoir ainsi corriger son erreur. Dans le Lexeur, une instance de LexeurError est générée lorsqu'un caractère inattendu est rencontré

4.5 Résumé et perspective d'amélioration

Bien que l'analyseur soit efficace pour traiter une gamme de tokens standard comme les mots-clés, identifiants, et opérateurs, certaines limitations inhérentes à la grammaire et à la conception actuelles ont été identifiées. Notamment, l'analyseur ne gère pas la concaténation de chaînes de caractères (string), les exceptions, ni certaines structures de code plus complexes telles que les instructions "case". De même, certaines conventions syntaxiques spécifiques, comme la déclaration de variables multiples sur une même ligne ou l'utilisation de bibliothèques Ada supplémentaires (par exemple, Ada.Integer_Text_IO), ne sont pas prises en charge en raison des contraintes de la grammaire.

Plusieurs axes d'amélioration peuvent être envisagés pour accroître les fonctionnalités de l'analyseur : gestion des exceptions et des erreurs du code Ada, support de bibliothèques supplémentaires, gestion de déclarations de la manière suivante : "a, b, c : Integer ;". Mais pour effectuer toutes ses modifications, il faudrait étendre la grammaire.

Chapitre 5

Analyseur syntaxique

La réalisation de l'analyseur syntaxique s'est faite par une approche de descente récursive. La descente récursive est une méthode d'analyse syntaxique où chaque règle de grammaire est associée à une fonction spécifique. Ces fonctions sont récursives, permettent ainsi de suivre la structure hiérarchique de la grammaire.

Une particularité de cette implémentation est l'analyse sur deux caractères, garantissant une analyse LL(2). Cela signifie que l'analyseur prend en compte jusqu'à deux symboles d'entrée à la fois pour prendre des décisions, améliorant ainsi la précision de la reconnaissance syntaxique.

5.1 Réalisation de l'analyseur syntaxique

Le code source de l'analyseur syntaxique pour le langage canAda est structuré de manière à refléter directement les règles de grammaire. Chaque règle est implémentée sous la forme d'une fonction récursive, permettant une correspondance claire entre le code et la spécification de la grammaire. Par exemple, la fonction associée à la règle declarationStar est responsable de reconnaître la structure particulière de cette règle dans le code source.

$$\langle \text{DeclarationStar} \rangle \rightarrow \langle \text{Declaration} \rangle \langle \text{DeclarationStar} \rangle \mid \varepsilon$$

```
1         declarationStar.setAction(() -> {
2             System.out.println("declarationStar");
3             // Try to do rules D and F1 else do nothing
4             try {
5                 Node node = new Node("declarationStar -> declaration declarationStar",
      declarationStar);
6                 node.setStatus(2);
7                 node.addChild(declaration.execute());
8                 node.addChild(declarationStar.execute());
9                 return node;
10            } catch (Exception e) {
11                System.out.println("catch F1");
12                return null;
13            }
14        });
```

Listing 5.1 – extrait de l'analyseur syntaxique

Dans cet extrait, la fonction associée à declarationStar tente de suivre la règle D, qui consiste à avoir une Declaration suivie d'une autre DeclarationStar. Si cette tentative échoue (attrapée par l'exception), cela signifie qu'il s'agit d'une déclaration vide, et la règle F1 est appliquée (une déclaration sans DeclarationStar).

5.2 Gestion des Erreurs dans l'Analyseur Syntaxique

La gestion des erreurs dans l'analyseur syntaxique est important pour débuguer le code. Nous avons mis en place un mécanisme de gestion des erreurs en utilisant les classes dédiées 'Error' et 'LexerError'.

5.2.1 Classes d'Erreurs

1. `Error` (Classe de Base)

La classe `Error` est conçue comme une classe de base étendue pour créer des classes d'erreurs spécifiques, contenant des informations sur la localisation de l'erreur.

```java
package Error;

public class Error extends Exception {
    private final int line;
    private final int column;

    public Error(int line, int column) {
        this.line = line;
        this.column = column;
    }

    public int getLine() {
        return this.line;
    }

    public int getColumn() {
        return this.column;
    }

    public String toString() {
        return "Error at line " + this.line + ", column " + this.column + ": " + this.getMessage()
    ;
    }
}
```

2. `LexerError`

La classe `LexerError` étend la classe `Error` et inclut une propriété `message` pour des informations spécifiques au lexer.

```java
package Error;

public class LexerError extends Error {

    private final String message;

    public LexerError(int line, int column, String message) {
        super(line, column);
        this.message = message;
    }

    public String getMessage() {
        return this.message;
    }

    public String toString() {
        return message + "\n" + "Lexer error at line " + this.getLine() + ", column " + this.
    getColumn() + ": " + this.getMessage();
    }
}
```

5.2.2 Utilisation dans l'Analyseur Syntaxique

Les classes d'erreurs, notamment `LexerError`, sont utilisées pour signaler des erreurs pendant l'analyse syntaxique, cela permet ainsi une localisation de l'incident.

```java
try {
    // ...
} catch (LexerError lexError) {
    System.err.println(lexError.toString());
    // Autres actions de gestion d'erreur si n cessaire
}
```

Listing 5.2 – extrait de l'analyseur syntaxique

Avantages :

- Fiabilité accrue grâce à des informations détaillées sur les erreurs.
- Prise en charge d'erreurs multiples dans une séquence de code.
- Facilite le processus de débogage et d'amélioration du code.

5.3 Test

Pour évaluer l'efficacité de l'analyseur syntaxique, nous avons effectué des tests avec un exemple de code représentatif du langage canAda. L'exemple de code utilisé est le suivant :

```
1 with Ada.Text_IO; use Ada.Text_IO;
2
3 procedure unDebut is
4 begin
5     x := a > b.bb + a ;
6 end unDebut;
```

Listing 5.3 – Exemple de code en canAda

L'analyseur syntaxique a généré l'arbre syntaxique suivant :

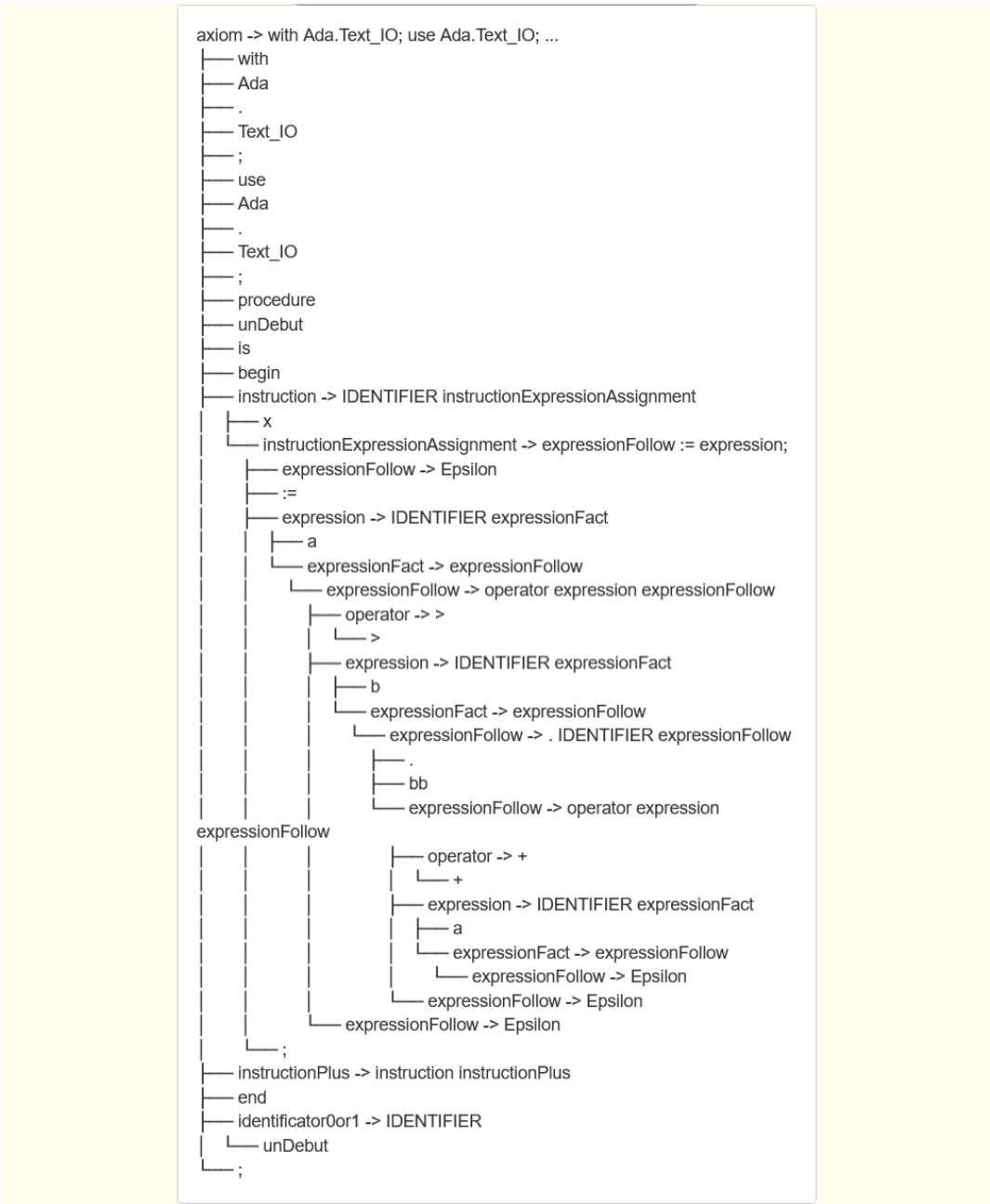


FIGURE 5.1 – Arbre syntaxique

Cet arbre syntaxique détaille la structure interne du code et met en évidence la hiérarchie des différentes règles de grammaire appliquées lors de l'analyse syntaxique.

Pour évaluer notre analyseur syntaxique, nous avons réalisé une batterie de tests sur des exemples de code représentatifs du langage canAda. Ces tests ont permis de montrer la capacité de l'analyseur à traiter différents codes et à générer des arbres syntaxiques cohérents. De plus, certains tests ont révélé des erreurs dans notre implémentation. Grâce à ces retours d'erreurs, nous avons pu apporter des améliorations à notre analyseur syntaxique, renforçant ainsi sa fiabilité et sa précision. Cette démarche itérative de tests et de corrections a été essentielle pour garantir la qualité de notre outil d'analyse syntaxique pour le langage canAda.

Chapitre 6

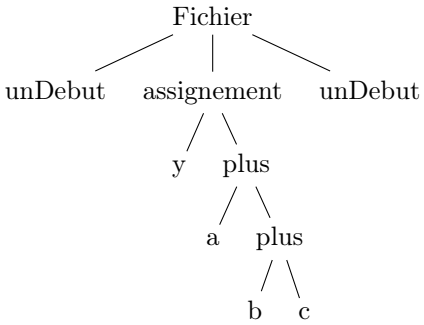
Arbre Abstrait

6.1 Exemple de code

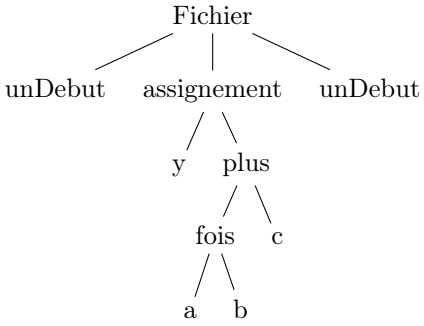
6.1.1 Exemple 1

```
1 with Ada.Text_IO ; use Ada.Text_IO ;
2
3 procedure unDebut is
4 begin
5     y := a + b + c;
6 end unDebut ;
```

Listing 6.1 – Exemple 1



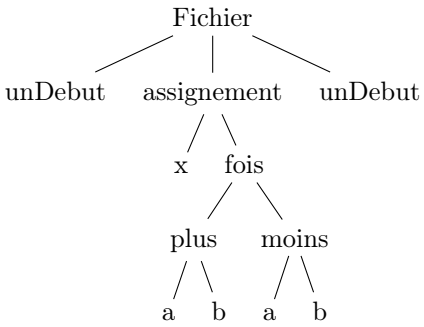
6.1.2 Exemple 2



6.2 Exemple 3

```
1 with Ada.Text_IO ; use Ada.Text_IO ;
2
3 procedure unDebut is
4 begin
5     x := ( a + b ) * ( a - b );
6 end unDebut ;
```

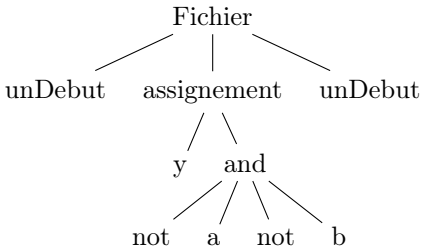
Listing 6.2 – Exemple 3



6.2.1 Exemple 4

```
1 with Ada.Text_IO ; use Ada.Text_IO ;
2
3 procedure unDebut is
4 begin
5     y := not a and not b;
6 end unDebut ;
```

Listing 6.3 – Code Ada mis à jour



6.2.2 Exemple 5

```
1  with Ada.Text_IO ; use Ada.Text_IO ;
2
3  procedure unDebut is
4      function aireRectangle (larg : integer; long : integer; aire : integer; surface : integer)
       return integer is
5      aire: integer;
6      begin
7          aire := larg * long ;
8          return aire;
9      end aireRectangle ;
10
11
12     function perimetreRectangle(larg : integer) return integer is
13     p : integer;
14     begin
15         p := larg2 + long2 ;
16         return p;
17     end perimetreRectangle;
18
19         -- VARIABLES
20 choix : integer ;
21         -- PROCEDURE PRINCIPALE
22     begin
23         choix := 2.1;
24
25     end unDebut ;
```

Listing 6.4 – Code Ada mis à jour

```
Fichier
├── unDebut
├── declarationStar -> declaration declarationStar
|   ├── declaration -> function IDENTIFIER [...]
|   |   ├── aireRectangle
|   |   ├── parameters -> ( parameter parametersSemicolonPlus )
|   |   |   ├── param
|   |   |   |   ├── larg
|   |   |   |   └── integer
|   |   |   ├── param
|   |   |   |   ├── long
|   |   |   |   └── integer
|   |   |   ├── param
|   |   |   |   ├── aire
|   |   |   |   └── integer
|   |   |   └── param
|   |   |       ├── surface
|   |   |       └── integer
|   |   ├── integer
|   |   ├── declaration -> IDENTIFIER indentificatorCommaPlus : type expressionAssignment;
|   |   |   ├── aire
|   |   |   └── integer
|   |   ├── :=
|   |   |   ├── aire
|   |   |   └── fois
|   |   |       ├── larg
|   |   |       └── long
|   |   └── aireRectangle
|   └── declarationStar -> declaration declarationStar
|       ├── declaration -> function IDENTIFIER [...]
|       |   ├── perimetreRectangle
|       |   ├── parameters -> ( parameter parametersSemicolonPlus )
|       |   |   ├── larg
|       |   |   └── integer
|       |   ├── integer
|       |   ├── declaration -> IDENTIFIER indentificatorCommaPlus : type expressionAssignment;
|       |   |   ├── p
|       |   |   └── integer
|       |   ├── :=
|       |   |   ├── p
|       |   |   └── plus
|       |   |       ├── larg2
|       |   |       └── long2
|       |   ├── end
|       |   └── perimetreRectangle
|       └── declaration -> IDENTIFIER indentificatorCommaPlus : type expressionAssignment;
|           ├── choix
|           └── integer
├── :=
|   ├── choix
|   └── 2.1
└── unDebut
```

Chapitre 7

Conclusion

Synthèse du Projet

En conclusion, ce projet de compilation pour le langage canAda a été une expérience enrichissante. Nous avons réussi à surmonter divers défis, de la définition de la grammaire à la mise en œuvre des analyseurs lexical et syntaxique, en passant par la construction de l'arbre abstrait.

L'analyseur lexical a été conçu pour reconnaître différents types de tokens, et la gestion des erreurs a été intégré. De même, l'analyseur syntaxique, basé sur une approche de descente récursive, a démontré sa capacité à générer des arbres syntaxiques cohérents.

L'implémentation de l'arbre abstrait a fourni une représentation du code source. Les exemples de code fournis ont illustré la capacité de l'arbre abstrait à refléter la logique du code fournit.

Bien que le projet ait atteint ses objectifs principaux, des perspectives d'amélioration demeurent. En fin de compte, ce projet a été une occasion d'apprentissage précieuse et a renforcé notre compréhension des processus de compilation et de conception de langages de programmation.

7.1 Annexes

COMPTE RENDU DE LA REUNION

Le 18 Octobre 2023, Télécom Nancy

Participants

Elise Neyens
Léo Fornoff
Ugo Birkel
Matthias Germain

Ordre du jour

- Présentation des membres de l'équipe projet.
- Lecture du projet, attendus généraux et deadline
- Fonctionnement de l'équipe projet
- Choix du langage
- Outils du projet
- Travail pour la prochaine fois

Autre points abordés

Résumé de la réunion

Première réunion du groupe qui durée 35 min. Elle a permis de rassembler les membres de l'équipe afin de discuter du déroulement du projet, des outils que l'on utilisera et de repartir les premières tâches.
Durant la réunion, les grands axes de projets ont été définis et la construction d'un diagramme de Gantt a été décidée.
Léo Fornoff a été désigné comme chef de projet. Les discussions seront faites via discord et à l'oral. Pour la gestion des tâches, l'application Trello sera utilisée et pour l'écriture du code Intellij a été proposé mais n'est pas imposé. Le langage principal du projet est JAVA.

Taches pour la prochaine fois

Tache	Responsable	Date rendu
Travail sur la grammaire pour la rendre LL(1) : éliminer la récusivité	Ugo et Matthias	01/11
Réalisation du lexeur	Léo et Elise	01/11
Creation du trello	Léo	26/10
Gantt du projet	Léo et Elise	01/11

Lecture du projet

> Discussion sur la création du dépôt GIT qui devra être fait par les élèves.
> Discussion autour de la mise en conformité de la grammaire proposée.
> Explication de l'utilité et du fonctionnement global d'un lexer (analyseur lexical).
> Rappel des contours du langage "canAda" à traiter dans le cadre du projet.

Fonctionnement de l'équipe projet

> La régularité des réunions d'avancement estimée à une semaine environ.
> Des séances de travail "en groupe" pour l'avancement du projet seront programmées certains mercredis après midi afin de permettre des discussions lors du travail de chacun des membres pour faciliter l'avancement.

Choix du langage

> Le Java a été choisi par l'ensemble des membres du projet. Il permet l'utilisation d'objets, du typage des données et n'a pas encore été utilisé lors d'un projet à l'école mais est connu par l'ensemble des membres du groupe.

Outils du projet

> Les outils du projets sont : - Discord pour la communication écrite et les appels a distance
> - Google drive pour les documents partagés
> - Overleaf pour les rapports des réunions et le rapport du projet
> - Intellij pour le code (non imposé mais suggéré)
> - plantUML pour les diagrammes
> - Trello pour la répartition des tâches, son utilisation n'est pas encore fixée

---

COMPTE RENDU DE LA REUNION

Le 27 Octobre 2023, Télécom Nancy

Participants

Elise Neyens
Léo Fornoff
Ugo Birkel
Matthias Germain

Backlog

Tâche	Responsable	Date rendu	Avancement
Travail sur la grammaire pour la rendre LL(1) : éliminer la récusivité	Ugo et Matthias	01/11	Achevé
Réalisation du lexeur	Léo et Elise	01/11	Achevé
Creation du trello	Léo	26/10	Non Achevé/ Abandon

Ordre du jour

- Présentation du diagramme de Gantt
- Présentation de la grammaire LL(1) du langage
- Présentation de la structure de l'analyseur lexical et de la grammaire
- Travail pour la prochaine fois

Autre points abordés

- Discussions autour des conventions de notations des règles de la grammaire

Résumé de la réunion

Première réunion d'avancement qui a duré environ 50 min. Elle a permis de à chacun des membres du groupe de présenter leurs avancées sur le projet. Durant la réunion, le diagramme de Gantt a été abordé et validé par tous les membres du projet. Ugo et Mathias ont présentés leur travail concernant la grammaire pendant la réunion. Léo et Elise ont présentés leur travail sur l'analyseur lexical et la structure de la grammaire. Pendant les explications de chaque parties, des discussions sur la structure de l'analyseur syntaxique ont été abordés ainsi que les conventions d'écriture pour les règles.

Taches pour la prochaine fois

Tache	Responsable	Date rendu
Dernière vérifications pour la grammaire et implémentation des règles	Ugo et Matthias	03/11
Présentation du fonctionnement de l'analyseur syntaxique sur une grammaire plus petite et LL(1)	Léo et Elise	03/11
Réalisation d'une fonctionnalité qui vérifie si une grammaire est LL(1)	Léo et Elise	03/11

Présentation du diagramme de Gantt

> Le chef de projet a présenté le diagramme de Gantt du projet qui a été validé par l'ensemble des membres du projet. Ce dernier a mis en évidence les différentes dead-lines.Il a insisté sur le fait de ne pas prendre de retard pour ne pas avoir à tout finaliser pendant les vacances de noël.

Présentation de la grammaire LL(1) du langage

> Ugo et Matthias, les responsables de la grammaire ont présenté leur travail sur les différentes règles de la grammaire. Pour rendre la grammaire LL(1), il a été nécessaire d'ajouter de nombreux non-terminaux. IL ya en tout maintenant 42 non-terminaux.De plus, en abordant les charactères de notre grammaire , il a été proposé d'ajouter le caractère espace qui devra être implémenté par un autre caractère.

Présentation de la structure de l'analyseur lexical et de la gram-maire

> Léo et Elise ont présenté la structure générale de l'implémentation de la grammaire et de l'analyseur lexicale à l'aide d'un diagramme de classe.Pendant cette présentation, Léo a aussi abordé l'implémentation de la grammaire et des règles de cette dernière.

Discussions autour des conventions de notations des règles de la grammaire

> Il a été convenu par l'ensemble des membres du projet qu'il était nécéssaire de définir des conventions pour ne pas se perdre dans l'implémentation de la grammaire et ses règles.Il a ainsi été convenu que chaque non-terminal est noté en majuscule avec un numéro correspondant au nombre de réecriture pour ce non-terminal(E devient E0, E' E1 ...).Il est aussi convenu que les expressions s'écriront nom du terminal en majuscule underscore le numéro de l'expression.

COMPTE RENDU DE LA REUNION

le 8 Novembre 2023, Telecom Nancy

Participants

Elise Neyens
Léo Fornoff
Ugo Birkel
Matthias Germain

Backlog

Tâche	Responsable	Date rendu	Avancement
vérification des premiers, suivants et symboles directeurs de la grammaire en vue de la présentation de la grammaire aux autres membres	Ugo	08/11	achevé
Présentation du fonctionnement de l'analyseur syntaxique sur une grammaire plus petite et LL(1)	Léo	08/11	achevé
Recherche sur les différentes options pour l'implémentation de l'analyseur syntaxique	Léo et Elise	15/11	en cours
Dernière vérifications pour la grammaire et implémentation des règles et prédicats	Ugo et Matthias	08/11	Non achevé / Abandon
Réalisation d'une fonctionnalité qui vérifie si une grammaire est LL(1)	Ugo et Matthias	08/11	en cours

Ordre du jour

- Présentation de la grammaire
- Travail en groupe de réflexion et de résolution de problème
- Travail en groupe de réflexion et de résolution de problème concernant : la grammaire, l'analyse syntaxique

Autre points abordés

- La gestion des conflits et le fait que la grammaire puisse ne pas être LL(1)
- Principes des prédicats et fonctionnement

Résumé de la réunion

La réunion s'est organisée en deux temps : une réunion d'avancement et ensuite une après-midi de travaille en groupe. La séance a commencé par la présentation de la grammaire aux autres membres du groupe, lors de la séance et pendant une discussion autour d'un conflit créé par la grammaire, nous avons pu le résoudre mais il fallait donc revoir la grammaire et refaire une étude de celle-ci.

Tâches pour la prochaine fois

Tache	Responsable	Date rendu
faire des tests de méthodes à utiliser pour l'analyseur syntaxique	Léo et Elise	29/11
s'informer sur la visualisation de l'abre syntaxique	Matthias	29/11
réécriture de la grammaire pour la rectifier	Ugo	29/11

Grammaire

> Lors de la réunion, nous avons discuté de différentes étapes qu'il faudra mettre en place une fois que la version de la grammaire sera stable. Le travail en groupe aura été la clef à la résolution d'un problème auquel aura fait face l'un des membres qui ne pouvait trouver de solution. La grammaire devra alors être réécrite afin de régler ce problème.

Analyse syntaxique

> Le chef de projet a expliqué comment procéder à l'analyse selon une méthode abordée en cours de première année : la descente récurssive. Cela a permis à l'ensemble du groupe de comprendre comment le projet allait avancer mais aussi de pouvoir donner son avis et de discuter des choix.

COMPTE RENDU DE LA REUNION

le 29 Novembre 2023, Telecom Nancy

Participants

Elise Neyens
Léo Fornoff
Ugo Birkel
Matthias Germain

Backlog

Tâche	Responsable	Date rendu	Avancement
faire des tests de méthodes à utiliser pour l'analyseur syntaxique	Léo et Elise	29/11	achevé
s'informer sur la visualisation de l'abre syntaxique	Matthias	29/11	achevé
réécriture de la grammaire	Ugo	29/11	achevé

Ordre du jour

- Mise au point des conflits restants autour de la grammaire
- Travail en groupe pour les résoudre
- Installation de JavaFX pour commencer une visualisation d'un abre abstrait

Autre points abordés

- Exclusion de certains problèmes de la grammaire car gérées en dehors de celle-ci

Résumé de la réunion

La réunion s'est organisée en deux temps : une réunion d'avancement et ensuite une après-midi de travaille en groupe. Pendant cette séance d'une après-midi, Elise et Ugo se sont lancés dans la résolution d'un conflit dans la grammaire et l'ont modifié. Léo pendant ce temps était en train d'implémenter des démonstrations de code pour un analyseur syntaxique, il a en parallèle essayé d'implémenter un code python pour permettre l'automatisation des calculs de premiers, suivants et par conséquent des symboles directeurs pour faire gagner du temps au personne en charge de l'amélioration de la grammaire. Matthias, quant à lui, a essayé d'implémenter un arbre abstrait en utilisant JavaFX mais a rencontré des difficultés pour débuter cette tâche par rapport à ce qui était escompté. Nous avons enfin tous travaillé en groupe lors de séance de réflexion au tableau pour pouvoir discerner les décisions qu'il faudrait envisager pour la suite du projet.

Tâches pour la prochaine fois

Tache	Responsable	Date rendu
Réalisation de l'analyseur syntaxique	Léo et Elise	21/12
Réalisation d'une visualition d'arbre abstrait avec JavaFX	Matthias	21/12
Amélioration de la grammaire	Elise et Ugo	21/12
Implémentation de la grammaire	Ugo	21/12
Gestion des priorités opératoires	Elise et Ugo	21/12

Grammaire

> La dernière séance de travaille ne groupe aura été l'occasion de mettre en avant une modification majeure à effectuer dans la grammaire, cette séance aura donc permis de repartir sur une nouvelle grammaire en l'analysant et en réalisant une tablle LL(1). La mise en avant des conflits aura permis de résoudre une partie d'entre eux. Un conflit sera laissé en suspens après cette séance. Nous envisageons alors que l'on ne puisse pas avoir une grammaire LL(1) et cherchons à résoudre le problème autrement.

Analyse syntaxique

> Le chef de projet a travaillé sur le sujet et expliqué au reste du groupe les techniques qui peuvent être implémentées. Il expliqua notamment la structure théorique que devrait avoir notre code pour réaliser une descente récursive.

Gestion du Temps

> Etant donné que le rythme de travaille a commencé à s'intensifier avec la venue des partiels, le temps laissé pour réaliser ses tâches ont été augmentés pour permettre à tous les membres de gérer au mieux leur temps de révision.

COMPTE RENDU DE LA REUNION

le 21 Décembre 2023, Telecom Nancy

Participants

Elise Neyens
Léo Fornoff
Ugo Birkel
Matthias Germain

Backlog

Tâche	Responsable	Date rendu	Avancement
Réalisation de l'analyseur syntaxique	Léo et Elise	21/12	en cours
réalisation d'une visualition d'arbre abstrait avec JavaFX	Matthias	21/12	en cours
amélioration de la grammaire	Elise et Ugo	21/12	en cours
implémentation de la grammaire	Ugo	21/12	achevé
gestion des priorités opératoires	Elise et Ugo	21/12	Non achevé

Ordre du jour

- Point sur l'évolution de chacun
- Résumé de ce qu'il reste à faire

Autre points abordés

- difficulté pour régler les derniers problèmes liés aux conflits

Résumé de la réunion

Le stand-up meeting a permis à chacun de faire le point avant les vacances d'hiver. Pour savoir ce qu'il restait à faire.

Tâches pour la prochaine fois

Tache	Responsable	Date rendu
Réalisation de l'analyseur syntaxique	Léo et Elise	09/01
Réalisation d'une visualition d'arbre abstrait avec JavaFX	Matthias	09/01
Amélioration de la grammaire	Elise et Ugo	09/01
Gestion des priorités opératoires	Elise et Ugo	09/01

TELECOM Nancy
Ingénieurs du numérique • Inspiring your digital future