Rapport d'activité du Projet de compilation

# Développement d'un compilateur pour le langage CanAda

Alexis MARCEL
Lucas LAURENT
Noé STEINER

*Responsables du module :*
M. Olivier FESTOR
Mme. Suzanne COLLIN

15 Janvier 2024

## Table des matières

# 1 Contexte du projet

Ce rapport présente le projet réalisé dans le cadre du module PCL1 de la deuxième année du cycle ingénieur à TELECOM Nancy. L'objectif principal est de développer, en groupe, un compilateur pour le langage *canAda*, une version simplifiée d'Ada. Ce projet est une opportunité d'approfondir nos compétences en analyse lexicale et syntaxique ainsi que la construction d'un arbre abstrait.

# 2 Introduction

Dans le cadre de nos études, la compréhension et le développement de compilateurs se révèlent cruciaux car ils permettent de mieux comprendre les principes fondamentaux de l'informatique, comme la structure des langages de programmation, l'analyse syntaxique ou encore les arbres abstraits. Cette connaissance est essentielle pour optimiser les performances des programmes, assurer leur sécurité, et développer des logiciels fiables et efficaces. Le projet *canAda* s'inspire d'Ada, un langage connu pour sa fiabilité et sa sécurité. Ce travail nous plonge dans la complexité de la compilation, nous préparant à des applications concrètes dans divers secteurs tels que les systèmes embarqués, la défense, ou l'aéronautique. En développant un compilateur, nous affrontons non seulement les défis techniques relatifs à la conception d'un tel compilateur mais cela constitue aussi une base de connaissances fondamentale pour nous, futurs ingénieurs. Nous avons pris comme décision de faire ce projet en Java car nous avions envie d'approfondir notre connaissance de ce langage et de ses outils.

# 3 Grammaire

## 3.1 Présentation

Le sujet nous a fourni une grammaire associée au langage *canAda*. Cette grammaire est une version simplifiée de la grammaire du langage Ada et était sous une forme abstraire avec notamment des regex.

```
⟨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 1 – Grammaire initiale du Sujet

## 3.2 Étapes de transformation de la Grammaire

### 3.2.1 Grammaire originale en BNF

La grammaire initiale du langage *canAda*, avant sa transformation en grammaire LL(1), se présente comme suit en BNF sans les regex :

```
1   fichier -> withAda.Text_IO ; useAda.Text_IO ; procedure ident is <decls>
        begin <instrs> end <hasident> ; EOF
2
3   decl -> type ident ;
```

```
4           | type ident is access ident ;
5           | type ident is record <champs> end record ;
6           | <identsep> : <type> <typexpr> ;
7           | procedure ident <hasparams> is <decls> begin <instrs> end <
              hasident> ;
8           | function ident <hasparams> return <type> is <decls> begin <
              instrs> end <hasident> ;
9
10      decls -> <decl> <decls>
11             | ϵ
12
13      hasident -> ident
14                | ϵ
15
16      identsep -> ident , <identsep>
17                | ident
18
19      champ -> <identsep> : <type> ;
20
21      champs -> <champ> <champs>
22              | <champ>
23
24      type -> ident
25            | access ident
26
27      params -> ( <paramsep> )
28
29      hasparams -> <params>
30                 | ϵ
31
32      paramsep -> <param> ; <paramsep>
33                | <param>
34
35      typexpr -> := <expr>
36               | ϵ
37
38      param -> <identsep> : <mode> <type>
39
40      mode -> in
41            | in out
42            | ϵ
43
44      expr -> entier
45            | caractre
46            | true
47            | false
48            | null
49            | ( <expr> )
50            | <acces>
51            | <expr> <operateur> <expr>
52            | not <expr>
53            | − <expr>
54            | new ident
55            | ident ( <exprsep> )
56            | character ' val ( <expr> )
57
58      exprsep -> <expr> , <exprsep>
59               | <expr>
```

```
60
61          hasexpr -> <expr>
62                   | ϵ
63
64          instr -> <acces> := <expr> ;
65                 | ident ;
66                 | ident ( <exprsep> ) ;
67                 | return <hasexpr> ;
68                 | begin <instrs> end ;
69                 | if <expr> then <instrs> <elsif> <else> end if ;
70                 | for ident in <hasreverse> <expr> .. <expr> loop <instrs>
                       end loop ;
71                 | while <expr> loop <instrs> end loop ;
72
73          elsif -> elsif <expr> then <instrs> <elsif>
74                 | ϵ
75
76          else -> else <instrs>
77                 | ϵ
78
79          hasreverse -> reverse
80                      | ϵ
81
82          instrs -> <instr> <instrs>
83                  | <instr>
84
85          operateur -> = | /= | < | <= | > | >= | + | - | * | / | rem |
               and | and then | or | or else
86
87          acces -> ident
88                 | <expr> . ident
```

### 3.2.2 Élimination de la Récursivité à Gauche

La grammaire initiale comportait plusieurs instances de récursivité à gauche. Par exemple, les deux règles suivantes :

```
1           expr -> <acces>
2           acces -> ident | <expr> . ident
```

ont été transformées en :

```
1           expr -> [...] -> primary
2           primary -> ident <primary2>
3           primary2 -> ( <exprsep> ) <acces> | <acces>
4           acces -> . ident <acces> | ϵ
```

Comme la priorisation des calculs a beaucoup impacté la structure de la grammaire, on note [...] pour représenter l'enchainement de règles permettant de se retrouver dans le cas de la règle *expr* de départ. Cette modification élimine la (double ici) récursivité à gauche, rendant la grammaire adaptée pour une analyse LL(1). Une autre récursivité à gauche a été supprimé mais elle a été faite aussi à travers la priorisation des règles.

### 3.2.3 Factorisation à Gauche

La factorisation à gauche a été nécessaire pour certaines règles. Par exemple, les règles suivantes :

```
1           exprsep -> <expr> , <exprsep>
2           exprsep -> <expr>
```

ont été réécrites en :

```
1  exprsep -> <expr> <exprsep'>
2  exprsep' -> , <expr> <exprsep'> | ϵ
```

Ceci assure que la règle peut être analysée de manière déterministe en LL(1).

#### 3.2.4 Gestion des Priorités de Calculs

Pour gérer correctement les priorités des opérations, la grammaire a été ajustée notamment au niveau des règles de *expr*. Par exemple, les opérations de *Ou* et *Et* ont été séparées des opérations d'addition et de soustraction pour respecter leur priorité selon le sujet, par exemple :

```
1  expr -> <expr> <operateur> <expr>
```

a été réécrite en partie de cette manière :

```
1  expr -> <or_expr>
2  or_expr -> <and_expr> <or_expr'>
3  or_expr' -> or <or_expr'2> | ϵ
4  or_expr'2 -> <and_expr> <or_expr'>
5  or_expr'2 -> else <and_expr> <or_expr'>
6  and_expr -> <not_expr> <and_expr'>
7  [...]
8  unary_expr -> - <unary_expr>
9  unary_expr -> <primary>
10 primary -> entier
11 primary -> caractre
12 primary -> true
13 primary -> false
14 primary -> null
15 primary -> ( <expr> )
16 primary -> ident <primary2>
17 primary -> new ident
18 primary -> character ' val ( <expr> )
```

Cela permet de respecter la hiérarchie des opérations dans les expressions arithmétiques et logiques.

Les ensembles de sélection distincts ont été calculés pour assurer une sélection univoque lors de l'analyse.

Ces étapes illustrent comment la grammaire initiale a été transformée en une grammaire LL(1), adaptée pour une analyse syntaxique efficace et précise du langage *canAda*

### 3.3 Grammaire Transformée en LL(1)

La grammaire transformée en LL(1) se présente comme suit :

```
1  fichier -> withAda.Text_IO ; useAda.Text_IO ; procedure ident is decls
       begin instrs end hasident ; EOF
2  decl -> type ident hasischoose ;
3  decl -> identsep : type_n typexpr ;
4  decl -> procedure ident hasparams is decls begin instrs end hasident
       ;
5  decl -> function ident hasparams return type_n is decls begin instrs
       end hasident ;
6
7  hasischoose -> is accorrec | ϵ
8
9  accorrec -> access ident
10 accorrec -> record champs end record
11
12 decls -> decl decls
```

```
13          decls -> ϵ
14
15          hasident -> ident
16          hasident -> ϵ
17
18          identsep -> ident identsep2
19
20          identsep2 -> , identsep
21          identsep2 -> ϵ
22
23          champ -> identsep : type_n ;
24
25          champs -> champ champs2
26
27          champs2 -> champs | ϵ
28
29          type_n -> ident
30          type_n -> access ident
31
32          params -> ( paramsep )
33
34          hasparams -> params
35          hasparams -> ϵ
36
37          paramsep -> param paramsep2
38
39          paramsep2 -> ; paramsep
40          paramsep2 -> ϵ
41
42          typexpr -> := expr
43          typexpr -> ϵ
44
45          param -> identsep : mode type_n
46
47          mode -> in modeout
48          mode -> ϵ
49
50          modeout -> out
51          modeout -> ϵ
52
53          expr -> or_expr
54
55          or_expr -> and_expr or_expr'
56
57          or_expr' -> or or_expr'2
58          or_expr' -> ϵ
59
60          or_expr'2 -> and_expr or_expr'
61          or_expr'2 -> else and_expr or_expr'
62
63          and_expr -> not_expr and_expr'
64
65          and_expr' -> and and_expr'2
66          and_expr' -> ϵ
67
68          and_expr'2 -> not_expr and_expr'
69          and_expr'2 -> then not_expr and_expr'
70
```

```
71  not_expr -> equality_expr not_expr'
72
73  not_expr' -> not equality_expr not_expr'
74  not_expr' -> ϵ
75
76  equality_expr -> relational_expr equality_expr'
77
78  equality_expr' -> = relational_expr equality_expr'
79  equality_expr' -> /= relational_expr equality_expr'
80  equality_expr' -> ϵ
81
82  relational_expr -> additive_expr relational_expr'
83
84  relational_expr' -> < additive_expr relational_expr'
85  relational_expr' -> <= additive_expr relational_expr'
86  relational_expr' -> > additive_expr relational_expr'
87  relational_expr' -> >= additive_expr relational_expr'
88  relational_expr' -> ϵ
89
90  additive_expr -> multiplicative_expr additive_expr'
91
92  additive_expr' -> + multiplicative_expr additive_expr'
93  additive_expr' -> - multiplicative_expr additive_expr'
94  additive_expr' -> ϵ
95
96  multiplicative_expr -> unary_expr multiplicative_expr'
97
98  multiplicative_expr' -> * unary_expr multiplicative_expr'
99  multiplicative_expr' -> / unary_expr multiplicative_expr'
100 multiplicative_expr' -> rem unary_expr multiplicative_expr'
101 multiplicative_expr' -> ϵ
102
103 unary_expr -> - unary_expr
104 unary_expr -> primary
105
106 primary -> entier
107 primary -> caractre
108 primary -> true
109 primary -> false
110 primary -> null
111 primary -> ( expr )
112 primary -> ident primary2
113 primary -> new ident
114 primary -> character ' val ( expr )
115
116 primary2 -> ( exprsep ) acces
117 primary2 -> acces
118
119 exprsep -> expr exprsep2
120
121 exprsep2 -> , exprsep
122 exprsep2 -> ϵ
123
124 hasexpr -> expr
125 hasexpr -> ϵ
126
127 instr -> ident instr2
128 instr -> return hasexpr ;
```

```
129         instr -> begin instrs end ;
130         instr -> if expr then instrs elifn elsen end if ;
131         instr -> for ident in hasreverse expr .. expr loop instrs end loop ;
132         instr -> while expr loop instrs end loop ;
133
134         instr2 -> instr3 := expr ;
135         instr2 -> ( exprsep ) instr3 hasassign ;
136         instr2 -> ;
137
138         instr3 -> . ident instr3
139         instr3 -> ϵ
140
141         hasassign -> := expr
142         hasassign -> ϵ
143
144         elifn -> elif expr then instrs elifn
145         elifn -> ϵ
146
147         elsen -> else instrs
148         elsen -> ϵ
149
150         hasreverse -> reverse
151         hasreverse -> ϵ
152
153         instrs -> instr instrs2
154
155         instrs2 -> instr instrs2
156         instrs2 -> ϵ
157
158         acces -> . ident acces
159         acces -> ϵ
```

## 3.4 Table LL(1)

A l'aide de la grammaire transformée en LL(1), nous avons pu construire la table LL(1) suivante pour produire notre *Parser*. Ci-dessous, nous présentons une partie de la table LL(1) pour des raisons de lisibilité.

| | $ | withAda.Text_IO;useAda.Text_IO; | procedure | ident | is | begin | end | ; |
|---|---|---|---|---|---|---|---|---|
| S | | S := fichier $ | | | | | | |
| fichier | | fichier := withAda.Text_IO;useAda.Text_IO; procedure ident is decls begin instrs end hasident ; EOF | | | | | | |
| decl | | | decl := procedure ident hasparams is decls begin instrs end hasident ; | decl := identsep : type_n typexpr ; | | | | |
| hasischoose | | | | | hasischoose := is accorrec | | | hasischoose := ε |
| accorrec | | | | | | | | |
| decls | | | decls := decl decls | decls := decl decls | | decls := ε | | |
| hasident | | | | hasident := ident | | | | hasident := ε |
| identsep | | | | identsep := ident identsep2 | | | | |
| identsep2 | | | | | | | | |
| champ | | | | champ := identsep : type_n ; | | | | |
| champs | | | | champs := champ champs2 | | | | |
| champs2 | | | | champs2 := champs | | | champs2 := ε | |
| type_n | | | | type_n := ident | | | | |
| params | | | | | | | | |
| hasparams | | | | | hasparams := ε | | | |
| paramsep | | | | paramsep := param paramsep2 | | | | |
| paramsep2 | | | | | | | | paramsep2 := ; paramsep |
| typexpr | | | | | | | | typexpr := ε |
| param | | | | param := identsep : mode type_n | | | | |
| mode | | | | mode := ε | | | | |
| modeout | | | | modeout := ε | | | | |
| expr | | | | expr := or_expr | | | | |
| or_expr | | | | or_expr := and_expr or_expr' | | | | |
| or_expr' | | | | | | | | or_expr' := ε |
| or_expr'2 | | | | or_expr'2 := and_expr or_expr' | | | | |
| and_expr | | | | and_expr := not_expr and_expr' | | | | |

FIGURE 2 – Table partielle LL(1)

# 4 Analyse Lexicale pour canAda

L'analyse lexicale, une étape cruciale dans le processus de compilation, est gérée par notre classe *Lexer*. Cette classe est responsable de la conversion du code source en une série de tokens, facilitant ainsi l'analyse syntaxique ultérieure. Elle contient un *PeekingReader* codé par nos soins, qui permet de lire dans le flux de caractères pour identifier correctement les tokens complexes tout en conservant ce qui a été lu et en lisant caractère par caractère. Elle contient également un *ErrorService* qui permet de gérer les erreurs lexicales mais aussi une map de mots clés qui permet de gérer les mots clés du langage, les opérateurs et les symboles.

## 4.1 Structure et Fonctionnement du Lexer

La classe *Lexer* lit le code source et identifie les différents tokens en se basant sur un ensemble de règles prédéfinies, il est codé à l'aide du pattern *Singleton* pour éviter d'avoir plusieurs instances de la classe. Chaque token est une instance de la classe *Token*, qui contient des informations telles que le type de token et la valeur lexicale associée contenu dans l'enum *Tag*.

## 4.2 Gestion des Tokens

Des classes spécifiques, comme *Tag* et *PeekingReader*, sont utilisées pour catégoriser les tokens et gérer efficacement la lecture en avance du code source. La classe *Tag* définit les différents types de tokens, c'est une enum, tandis que *PeekingReader* permet de lire en avance dans le flux de caractères pour identifier correctement les tokens complexes en se passant donc d'un automate à états finis. Cela repose sur deux fonctions complexes qui permettent de lire le flux de caractères et de déterminer si le caractère courant est la fin d'un token.

```
1         /**
2          * Check if the current character is the end of a token
3          *
4          * @return true if the current character is the end of a token,
               false otherwise
5          */
6         private boolean isEndOfToken() {
7             char current = (char) currentChar;
8             int nextInt = this.reader.peek(1);
9             char next = (char) nextInt;
10
11            boolean isCurrentLetterOrDigit = Character.isLetterOrDigit(
                  current) || current == '_';
12            boolean isNextLetterOrDigit = Character.isLetterOrDigit(next
                  ) || next == '_';
13            boolean isNextWhitespace = Character.isWhitespace(next);
14
15            // If the current character is a whitespace or the end of
                  the file, the current character is the end of the token
16            if (nextInt == -1 || isNextWhitespace) {
17                return true;
18            }
19            // If the current character is an identifier or an integer,
                  the next character must not be a letter or a digit
20            if (isCurrentLetterOrDigit) {
21                return !isNextLetterOrDigit;
22            }
23
24            Token token = this.matchToken(lexeme.toString());
25
26            // If the current character is a token and the next
                  character is not a token, the current character is the
                  end of the token
```

```
27              if (token.tag() != Tag.UNKNOWN) {
28                  Token nextToken = this.matchToken(lexeme.toString() +
                        next);
29                  return nextToken.tag() == Tag.UNKNOWN;
30              }
31
32              return false;
33          }
```

```
1           public Token nextToken() {
2
3               while ((this.currentChar = this.reader.read()) != -1) {
4
5                   if (this.isComment()) {
6                       this.skipComment();
7                   } else if (Character.isWhitespace((char) currentChar)) {
8                       this.skipWhitespace();
9                   } else if (isCharacterLiteral()) {
10                      return this.readCharacterLiteral();
11                  } else {
12                      lexeme.append((char) currentChar);
13                      if (this.isEndOfToken()) {
14                          Token token = this.matchToken(lexeme.toString())
                                ;
15                          lexeme.setLength(0); // clear the StringBuilder
16                          if (token.tag() == Tag.UNKNOWN) {
17                              this.errorService.registerLexicalError(new
                                    UnknownTokenException(token));
18                          }
19                          return token;
20                      }
21                  }
22              }
23
24              this.reader.close();
25              return new Token(Tag.EOF, this.reader.getCurrentLine(),
                    lexeme.toString());
26          }
```

### 4.3 Optimisation et Fiabilité

Le *Lexer* est conçu pour être à la fois rapide et fiable, capable d'identifier précisément les tokens même dans des cas de syntaxe complexe grâce aux fonctions présentées ci-dessus. Cette précision est essentielle pour garantir une analyse syntaxique sans erreur dans les étapes suivantes du processus de compilation. Le fait de se passer d'un automate à états finis permet d'optimiser le temps d'exécution du *Lexer*.

## 5 Analyse Syntaxique

### 5.1 Structure et Fonctionnement

La classe *Parser* a été conçue pour analyser les programmes écrits dans le langage *canAda*, il est codé, lui aussi, à l'aide du pattern *Singleton* pour éviter d'avoir plusieurs instances de la classe. Il contient le token courant et l'analyseur lexical qui est utilisé pour analyser le programme source. Chaque méthode de cette classe correspond à un non-terminal de la grammaire LL(1) et est responsable de l'analyse d'une structure syntaxique spécifique du langage. Par exemple, une méthode *expr()* est utilisée pour analyser les expressions, correspondant au non-terminal *expr* de la grammaire. Ces méthodes sont appelées récursivement pour construire l'arbre syntaxique du programme source. En fonction du token courant,

on applique la règle associée à ce token d'après la table LL(1) construite précédemment. Si des terminaux sont dans cette règle, ils ont lu à travers la fonction *analyseTerminal()* trouvable dans la suite du rapport qui permet de vérifier si le token courant correspond bien au terminal attendu. Sinon on appelle la méthode associée au non-terminal de la règle et ainsi de suite ...

```
1        @PrintMethodName
2        private void expr() {
3            switch (this.currentToken.tag()) {
4                case IDENT, OPEN_PAREN, DOT, ENTIER, CARACTERE, TRUE,
                     FALSE, NULL, NEW, CHARACTER -> {
5                    or_expr();
6                }
7            }
8        }
```

## 5.2 Interaction avec l'Analyseur Lexical

Le *Parser* interagit étroitement avec l'analyseur lexical, recevant un flux de tokens qui sont analysés selon les règles de la grammaire. Cette interaction est cruciale pour la décomposition correcte du programme source en ses composants syntaxiques. On lit les tokens un par un et on les compare avec les règles de la grammaire. Si le token correspond à la règle, on passe au token suivant. Si le token ne correspond pas à la règle, on génère une erreur syntaxique.

```
1        @PrintMethodName
2        private void analyseTerminal(Tag tag) {
3            System.out.println("\t\t-> " + this.currentToken);
4            if (!(this.currentToken.tag() == tag)) {
5                Token expectedToken = new Token(tag, this.currentToken.
                     line(), TagHelper.getTagString(tag));
6                if (expectedToken.tag() == Tag.SEMICOLON) {
7                    this.errorService.registerSyntaxWarning(new
                         MissingSemicolonException(this.currentToken));
8                } else {
9                this.errorService.registerSyntaxError(new
                     UnexpectedTokenException(expectedToken, this.
                     currentToken));}
10           }
11           // Contient le prochain token ou <EOF, currentLine,""> si
                 fin de fichier
12           if (this.currentToken.tag() == Tag.EOF) {
13               return;
14           }
15           this.currentToken = lexer.nextToken();
16       }
```

## 5.3 Gestion des Erreurs Syntaxiques

Un aspect essentiel du *Parser* est sa capacité à gérer les erreurs syntaxiques comme nous le voyons dans le code ci-dessus. Lorsque le programme source ne respecte pas les règles de la grammaire, des messages d'erreur descriptifs sont générés, indiquant la ligne et le token attendu par rapport au token reçu, facilitant la localisation et la correction des erreurs par les développeurs. On a notamment appliqué le *Panic Mode* pour gérer les erreurs syntaxiques. Le *Parser* continue à analyser le programme source jusqu'à la fin tout en indiquant les erreurs rencontrées.

# 6 Construction de l'Arbre Abstrait Syntaxique pour canAda

La construction de l'arbre abstrait syntaxique (AST) est une étape essentielle du processus de compilation du langage *canAda*. L'AST représente la structure syntaxique du programme source d'une manière

qui est à la fois concise et facile à manipuler pour les étapes suivantes de la compilation.

## 6.1 Structure et Fonctionnement de l'AST

Notre système d'AST est construit autour de la classe *ASTNode*, qui sert de classe de base pour les différents types de nœuds de l'arbre. Chaque nœud spécifique, comme *OperatorNode*, *ParameterNode*, ou *ProgramNode*, hérite de *ASTNode* et représente une construction syntaxique spécifique du langage.

Par exemple, *OperatorNode* représente une opération arithmétique ou logique, tandis que *ProgramNode* représente la structure globale du programme *canAda*.

```
1         public class ProgramNode extends ASTNode {
2             private ProcedureDeclarationNode rootProcedure;
3
4             public void setRootProcedure(ProcedureDeclarationNode
                  rootProcedure) {
5                 this.rootProcedure = rootProcedure;
6                 rootProcedure.setParent(this);
7             }
8         }
```

```
1         public class OperatorNode extends ASTNode {
2             private String operator;
3
4             public OperatorNode(String operator) {
5                 this.operator = operator;
6             }
7
8         }
```

Et ainsi de suite pour chaque nœud de l'arbre. On utilise alors notre fonction *Override toString()* pour afficher l'arbre abstrait syntaxique de manière recursive qui est dans notre classe abstraite dont extend nos différents nœuds. Ainsi on a juste a appeler la fonction *toString()* sur le nœud racine de l'arbre pour afficher l'arbre abstrait syntaxique qui récupère les différents nœuds de l'arbre et les affiche de manière récursivee en récupérant les différents attributs de chaque nœud.

```
1         public String toString() {
2             Field[] fields = this.getClass().getDeclaredFields();
3             String className = this.getClass().getSimpleName();
4             StringBuilder res = new StringBuilder(colorize(className,
                  Attribute.YELLOW_TEXT()) + " : { \n");
5             if (isJson) {
6                 res = new StringBuilder("{ \n");
7             }
8             int lastIndex = fields.length - 1;
9
10            for (int i = 0; i < fields.length; i++) {
11                Field field = fields[i];
12                field.setAccessible(true);
13                try {
14                    Object attributeValue = field.get(this);
15                    if (attributeValue instanceof String) {
16                        attributeValue = colorize("\"" + attributeValue
                              + "\"", Attribute.GREEN_TEXT());
17                    }
18                    if (attributeValue == null) {
19                        attributeValue = colorize("null", Attribute.
                              BRIGHT_MAGENTA_TEXT());
20                    }
```

```
21                      res.append("\t").append("\"").append(colorize(field.
                            getName(), Attribute.RED_TEXT())).append("\"").
                            append(" : ").append(attributeValue);
22                      if (i < lastIndex || !isJson) {
23                          res.append(",");
24                      }
25                      res.append(" \n");
26
27                  } catch (IllegalAccessException e) {
28                      System.err.println("Erreur lors de l'acces au champ
                            " + field.getName());
29                  }
30              }
31              res.append("}");
32              return format(res.toString());
33          }
```

### 6.2 Représentation des Structures Syntaxiques

Les nœuds de l'AST capturent les éléments essentiels des structures syntaxiques du programme, comme les opérations, les paramètres, et la structure globale du programme. Par exemple, *OperatorNode* représente une opération arithmétique ou logique, tandis que *ProgramNode* représente la structure globale du programme *canAda*.

### 6.3 Rôle dans le Processus de Compilation

L'AST joue un rôle central dans le processus de compilation. Après l'analyse syntaxique, le programme source est transformé en un AST, qui est ensuite utilisé pour les étapes de vérification sémantique, d'optimisation, et de génération de code. Cette représentation permet une manipulation plus aisée et plus efficace du programme source.

## 7 Symbol Tables

Les Symbol Tables sont des structures de données qui stockent des informations sur les identificateurs du programme, comme les variables, les fonctions, et les procédures.

Elles suivent toutes un schéma assez simple, a chaque bloc de code (une fonction, une procédure, une boucle for) on crée une nouvelle table des symboles, et on l'empile sur une pile de tables des symboles. Nous utilisons une pile afin de pouvoir par la suite réaliser les contrôles sémantiques au niveau des déclarations (prise en compte des scopes supérieurs).

Au niveau de la structure de donnée, il s'agit d'une simple Table de Hashage, avec comme clé le nom de l'identificateur et comme valeur un objet de type *Symbol* qui contient des informations sur l'identificateur.

Voici un exemple de Symbol Table pour une fonction *perimetreRectangle* ayant deux paramètres *larg* et *long* et une variable locale *p* :

```
1   Entering new scope -> creating new SymbolTable for
        perimetreRectangle  :
2       larg -> Parameter { identifier = 'larg', type = 'integer', mode
            = IN, shift = 4 }
3       long -> Parameter { identifier = 'long', type = 'integer', mode
            = IN, shift = 8 }
4       p -> Variable { identifier = 'p', type = 'integer', shift = 12 }
```

On peut aisément identifier la clé de la table (le nom de l'identificateur) et la valeur associée de type *Parameter* ou *Variable* qui contient des informations sur l'identificateur tel que le décalage dans la pile,

le type, le mode, etc.

# 8 Contrôles sémantiques

Afin de garantir la cohérence du programme, nous avons implémenté plusieurs contrôles sémantiques. Ces contrôles sont effectués après la construction de l'AST et la création des Symbol Tables.

Nous parcourons l'arbre de haut en bas (Pattern Visitor) en vérifiant les contraintes sémantiques du langage.

Voici une liste des contrôles sémantiques que nous avons implémentés :
- **Déclaration de variables** : Vérification de la déclaration des variables, si elles sont bien déclarées avant d'être utilisées.
- **Déclaration de fonctions et procédures** : Vérification de la déclaration des fonctions et procédures, si elles sont bien déclarées avant d'être utilisées.
- **Déclaration de types** : Vérification de la déclaration des types (Access, Record, Type)
- **Duplication de symbole** : Vérification de la duplication des symboles, si les symboles sont uniques dans le même scope.
- **Type des expressions** : Vérification du type des expressions, si les types des opérandes sont compatibles avec l'opérateur.
- **Type des variables** : Vérification du type des variables, si les types des variables sont compatibles avec les opérations effectuées.
- **Type des paramètres** : Vérification du type des paramètres, si les types des paramètres sont compatibles avec les types des arguments.
- **Type de retour** : Vérification du type de retour des fonctions, si le type de retour est compatible avec le type de la fonction.
- **Nombre de paramètres pour fonctions et procédures** : Vérification du nombre de paramètres pour les fonctions et procédures, si le nombre de paramètres est correct.
- **Vérification des paramètres IN et INOUT** : Vérification des paramètres IN et INOUT.

# 9 Génération de Code

Une fois l'AST, les Symbol Tables et les contrôles sémantiques complétés, la phase de génération de code peut commencer. Nous utilisons l'AST ainsi que les informations contenues dans les noeuds et dans les Symbol Tables pour générer le code final en ASM UAL ARM 32 bits.

## 9.1 Parcours de l'AST

Pour générer le code, nous parcourons l'AST dans la classe *ASMGenerator*. Nous partons du root node et nous descendons au fur et à mesure dans l'arbre grâce à un pattern de Double Dispatch : Visitor. Pour chaque type de noeud, nous générons le code correspondant. Chaque noeud est responsable uniquement de la génération de son propre code. Nous utilisons également une petite classe *Context*, celle-ci nous permet de stocker des informations sur le contexte actuel de génération de code, comme le nom de la fonction actuelle, le nombre de variables locales, etc. Bien qu'assez petite, *Context* est importante au niveau des opérations, en effet c'est elle qui nous permet de savoir dans quel registre enregistrer le résultat d'une opération (par exemple, une addition $3 + 9 + 1$ : 1 va dans R0, 9 dans R1, leur résultat va dans R0, 3 est stocké dans R1 puis R0 et R1 sont ADD dans R0).

## 9.2 Exemple

Afin d'illustrer sur un exemple, voici le code pour le noeud gérant les retours de fonctions :

```
1         @Override
2         public void visit(ReturnStatementNode node) throws Exception {
3             node.getExpression().accept(this);
4             this.output.append("""
```

```
5                       \t STR     R0, [R11, #4 * 2] ; Store return value
                            for in stack-frame
6                       \t MOV     R13, R11 ; Restore frame pointer
7                       \t LDMFD   R13!, {R11, PC} ; Restore caller's frame
                            pointer and return ASM address
8                       """);
9           }
```

On peut voir la façon dont on descend dans l'arbre (`node.getExpression().accept(this)`) et comment on génère le code correspondant à ce noeud. Bien que simple, ce code est représentatif de la manière dont nous générons le code pour chaque noeud de l'AST.

### 9.3 Quelques schémas de traduction

## 10 Tests et Validation

### 10.1 Pour un code source valide

Voici un code source valide en canAda qui permet de tester notre compilateur :

```
1           with Ada.Text_IO; useAda.Text_IO;
2           procedure Main is
3               A : Integer := 5;
4               B : Integer := 10;
5               Sum : Integer;
6           begin
7               Sum := A + B;
8           end Main;
```

On obtient alors après l'analyse lexicale et syntaxique la sortie suivante, on l'a découpée en 3 parties pour plus de lisibilité et laissée sur fond noir pour faciliter la lisibilité des couleurs :

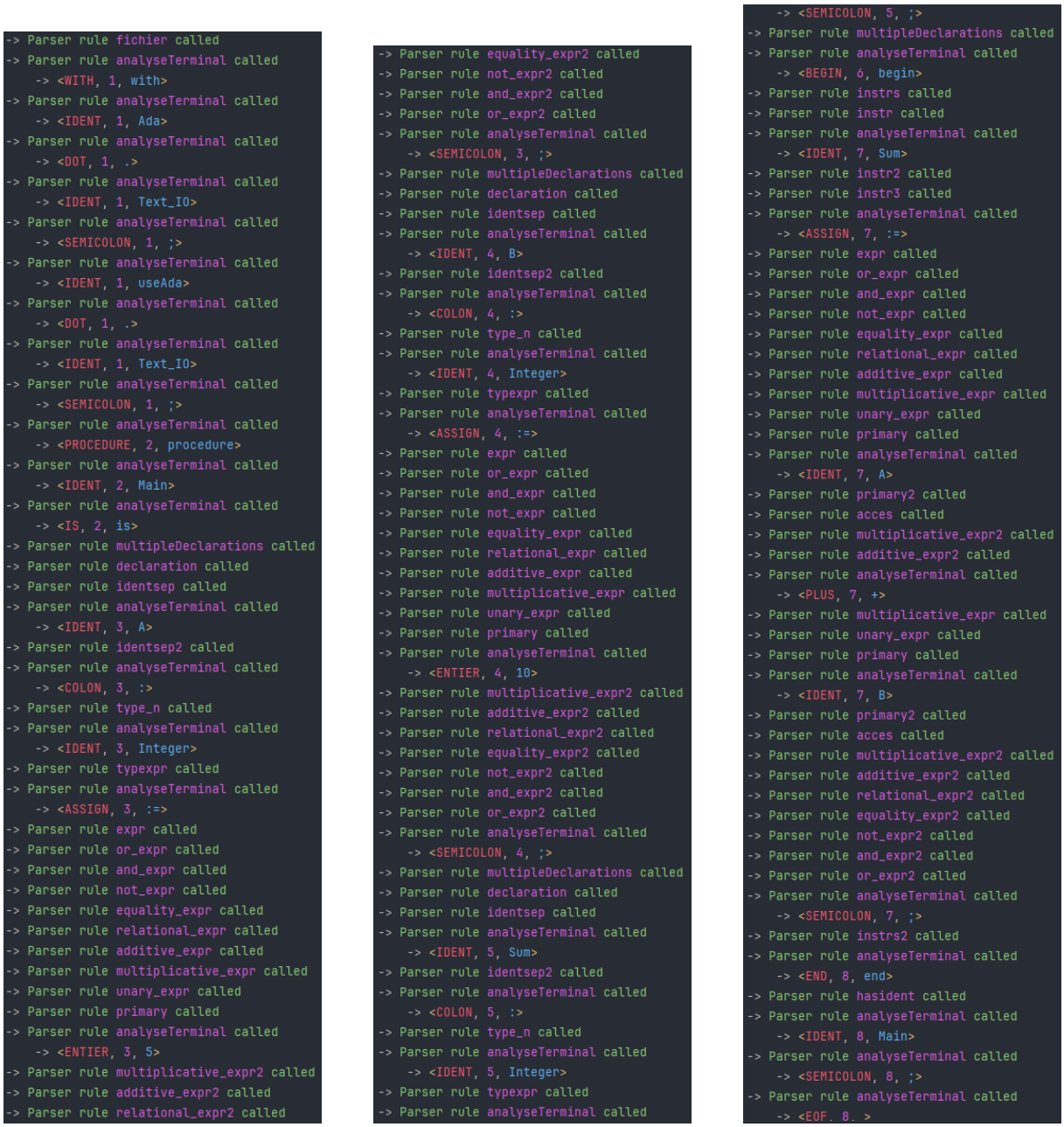


FIGURE 3 – Sortie découpée de gauche à droite

Et on obtient l'arbre abstrait suivant en sortie et donc ici en json :

```
ProgramNode : {
    "rootProcedure" : ProcedureDeclarationNode : {
        "parameters" : null,
        "body" : BlockNode : {
                "statements" : [BlockNode : {
                            "statements" : [],
                            "declarations" : [],
                        }],
                "declarations" : [TypeDeclarationNode : {
                            "type" : SimpleTypeNode : {
                            "typeName" : "Integer",
                        },
                            "name" : "A",
                        }, TypeDeclarationNode : {
                            "type" : SimpleTypeNode : {
                            "typeName" : "Integer",
                        },
                            "name" : "B",
                        }, TypeDeclarationNode : {
                            "type" : SimpleTypeNode : {
                            "typeName" : "Integer",
                        },
                            "name" : "Sum",
                        }],
            },
        "name" : "Main",
    },
}
```

FIGURE 4 – Arbre abstrait

Et à l'aide de notre script python et de graphviz on obtient l'arbre abstrait suivant :

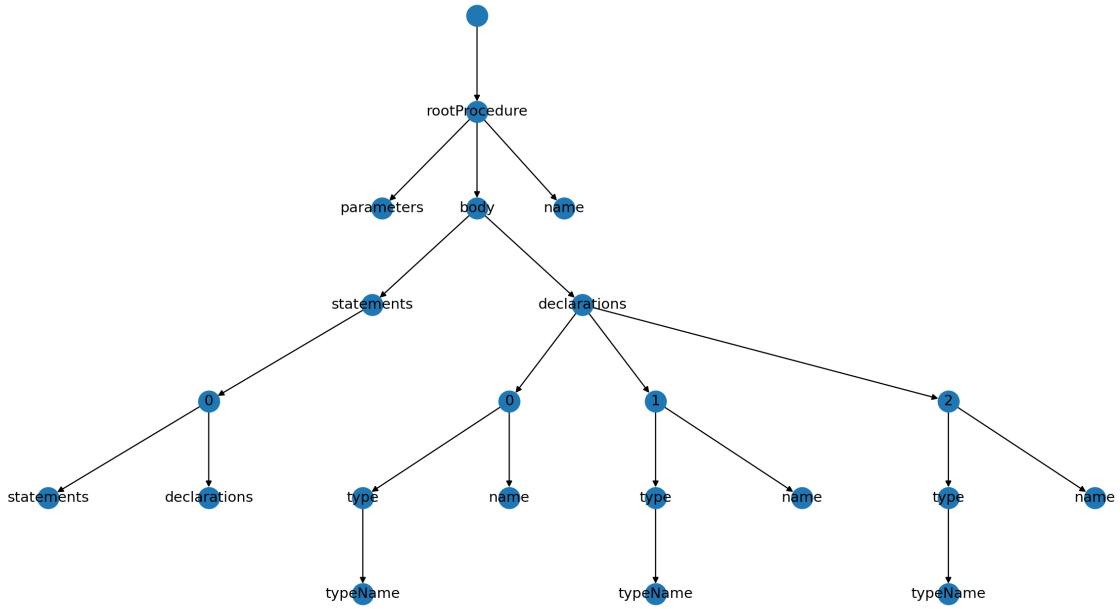


FIGURE 5 – Arbre abstrait

## 10.2 Pour un code source invalide

Voici un code source invalide en canAda qui permet de tester notre compilateur, on a volontairement rajouter un ; en trop à la fin de la ligne 5 :

```
1         with Ada.Text_IO; useAda.Text_IO;
2         procedure Main is
3             A : Integer := 5;
4             B : Integer := 10;
5             Sum : Integer;
6         begin ;
7             Sum := A + B;
8         end Main;
```

On obtient alors après l'analyse lexicale et syntaxique la sortie suivante, on l'a découpée en 3 parties pour plus de lisibilité et laissée sur fond noir pour faciliter la lisibilité des couleurs :

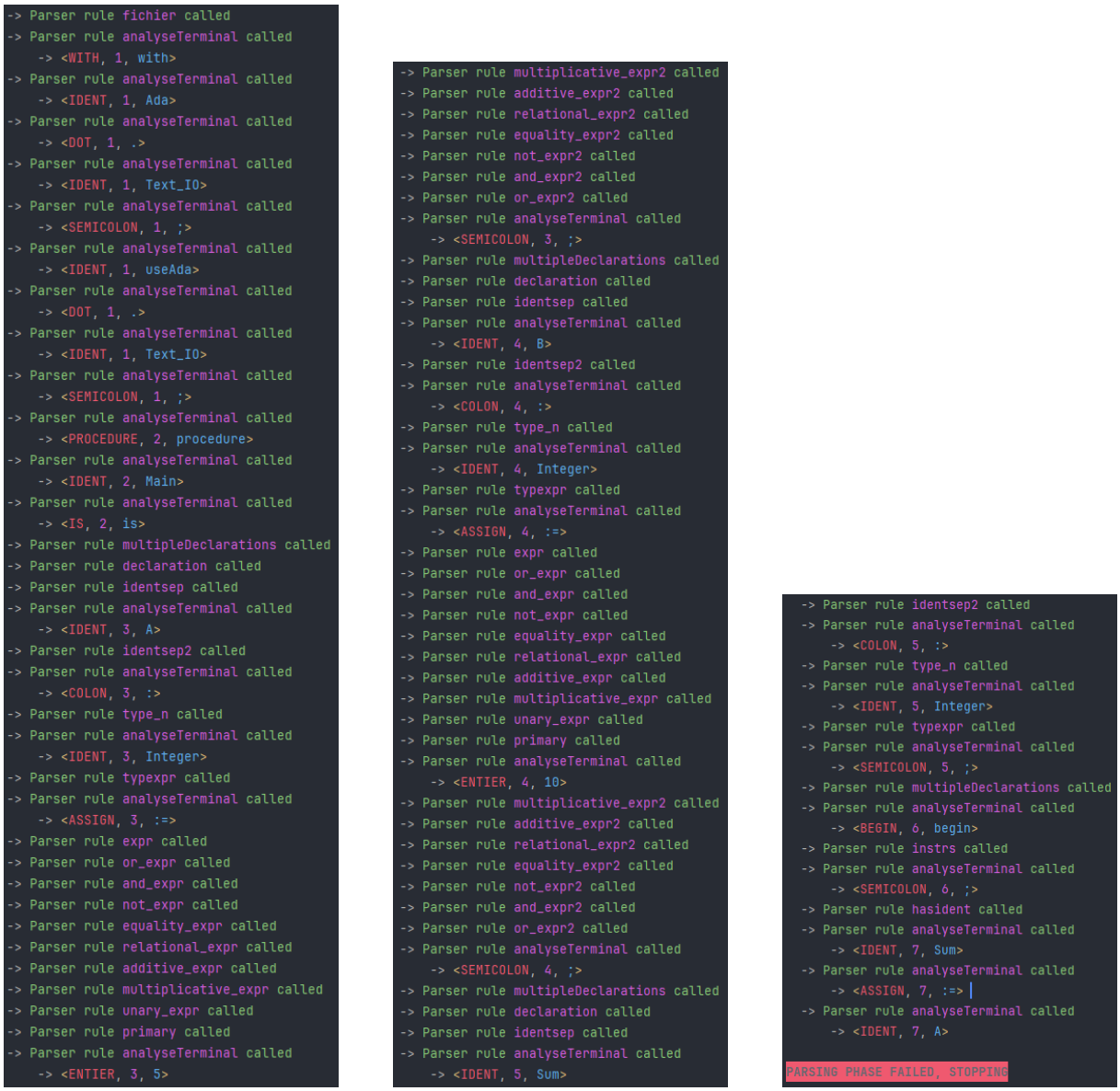


FIGURE 6 – Sortie découpée de gauche à droite

Et on obtient les erreurs et les warnings suivantes :

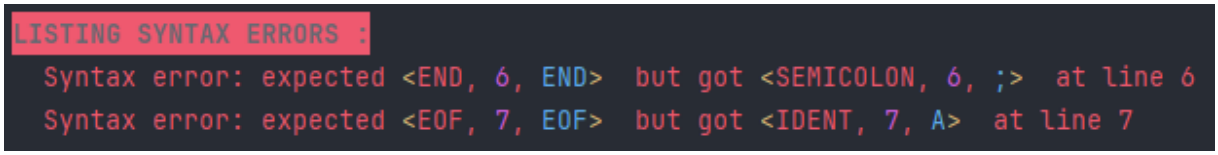


FIGURE 7 – Erreurs

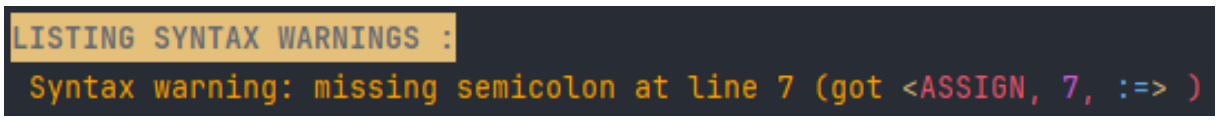


FIGURE 8 – Warnings

On peut ainsi voir que notre compilateur est capable de gérer les erreurs syntaxiques et de les afficher de manière claire et précise. Ainsi que de nous aider à le corriger notamment à travers les warnings et le

*Panic Mode* qui permet de continuer à analyser le programme source jusqu'à la fin tout en indiquant les erreurs rencontrées.

Ce qui conclue la partie sur les tests et la validation de notre compilateur. Bien évidemment, de nombreux autres tests ont été réalisés pour valider notre compilateur notamment des tests unitaires pour tester le *Lexer* et le *Parser* ainsi que la construction l'arbre asbtrait.

# 11 Gestion de projet

## 11.1 Équipe de projet

Ce projet est un projet local réalisé en groupe de 3 personnes :
— Alexis MARCEL
— Lucas LAURENT
— Noé STEINER

## 11.2 Organisation au sein de l'équipe projet

Nous avons réalisé plusieurs réunions, en présentiel dans les locaux de Télécom Nancy mais la plupart de notre collaboration a eu lieu sur Discord. Ces réunions nous ont permis de mettre en commun nos avancées régulièrement, de partager nos connaissances sur des problématiques et de nous organiser de manière optimale. En plus des réunions d'avancement régulières, nous avons également réalisé des réunions techniques afin de résoudre un problème ou bien de réfléchir à la conception.

Ensuite, nous avons utilisé GitLab pour gérer les différentes versions du développement de notre application, ainsi que les différentes branches nous permettant de travailler simultanément sans conflit.

## 11.3 Matrice RACI

Voici la matrice RACI de notre projet, elle nous a permis de nous organiser et de répartir les tâches de manière efficace, on remarque que personne n'approuve les tâches car nous avons travaillé en groupe et que nous avons tous approuvé les tâches entre nous.

| *Matrice RACI (R = Réalise ; A = Autorité ; C = Consulté ; I = Informé)* | **Acteurs** | | |
|---|---|---|---|
| **Tâches** | Lucas | Noé | Alexis |
| **Grammaire** | | | |
| Réalisation d'une grammaire LL(1) | R | I | I |
| **Analyseur Lexical** | | | |
| Réalisation de la structure du Lexer | I | R | R |
| Implémentation des Token et des Tag | R | R | R |
| Gestion de la lecture d'un fichier Token par Token | I | R | R |
| **Analyseur Syntaxique** | | | |
| Réalisation de la structure du Parser | R | C | I |
| Gestion des erreurs | R | R | I |
| Implémentation des règles | R | R | R |
| **Arbre Abstrait** | | | |
| Réalisation de la structure de l'arbre abstrait | I | R | I |
| Implémentation des différentes nodes | I | R | R |
| **Rédaction du rapport** | R | C | C |

FIGURE 9 – Matrice RACI

### 11.4 Répartition du Temps de Travail sur le Projet

On peut voir que la répartition du temps de travail est équilibrée entre les membres de l'équipe, ce qui a permis une contribution égale de tous les membres de l'équipe :

| Tâche | Lucas | Noé | Alexis |
|---|---|---|---|
| Grammaire | 10h | - | - |
| Structure du Lexer | - | 2h | 4h |
| Lecture du fichier (token par token) | - | 5h | 7h |
| Implémentation des Token et des Tag | 2h | 2h | 2h |
| Structure du Parser | 1h | - | - |
| Gestion des erreurs | 4h | 4h | - |
| Implémentation des règles | 3h | 3h | 3h |
| Structure de l'arbre abstrait | - | 2h | - |
| Implémentation des différents nodes | - | 6h | 6h |
| Rédaction du rapport | 4h | - | - |
| Total | 24h | 24h | 22h |

## 12 Conclusion

Ce rapport a présenté en détail le processus de développement d'un compilateur pour le langage *canAda*, réalisé dans le cadre du module PCL1 à TELECOM Nancy. Le projet a débuté par la transfor-

mation de la grammaire originale en une grammaire LL(1), adaptée pour une analyse syntaxique précise. Cette transformation a impliqué tout d'abord d'avoir une grammaire explicite sans regex, puis l'élimination de la récursivité à gauche, la factorisation, et la gestion des priorités de calculs pour assurer une analyse déterministe.

L'analyse lexicale a été effectuée par le *Lexer*, un composant clé qui convertit le code source en tokens, en s'appuyant sur un ensemble de règles prédéfinies et un système efficace de lecture en avance et d'une astuce pour déterminer la fin d'un token ce qui nous a permis de nous passer d'un automate à états finis. La classe *Parser*, quant à elle, a permis l'analyse syntaxique à travers la récupération du flux de *Token* du programme source et l'application des règles établies dans le tableau LL(1), tout en gérant les erreurs/warnings syntaxiques de manière robuste permettant ainsi de continuer à analyser le programme source jusqu'à la fin tout en indiquant les erreurs rencontrées pour aider le développeur à les corriger.

La construction de l'arbre abstrait syntaxique (AST) a été une étape cruciale, permettant une représentation concise et manipulable du programme source pour les étapes ultérieures de la compilation. Chaque nœud de l'AST, tel que *OperatorNode* ou *ProgramNode*, a capturé des éléments essentiels des structures syntaxiques du langage *canAda*.

La répartition du temps de travail sur les différentes tâches a été équilibrée, assurant ainsi une contribution égale de tous les membres de l'équipe.

En conclusion, ce projet a non seulement abouti à la création d'un compilateur fonctionnel pour *canAda* mais a également permis aux membres de l'équipe d'approfondir leurs compétences en informatique, notamment dans les domaines de l'analyse lexicale et syntaxique, ainsi que dans la construction et la manipulation d'arbres abstraits. Ce projet représente une étape significative dans notre parcours d'ingénieurs en informatique, nous préparant efficacement à des applications concrètes dans divers secteurs technologiques.

Il a également renforcé notre compréhension des fondamentaux de la compilation, une compétence essentielle pour tout développeur de logiciels ainsi que pour tout ingénieur en informatique.