Rapport final de Projet de compilation

Développement d'un compilateur pour le language CanAda

Alexis MARCEL
Lucas LAURENT
Noé STEINER

Responsable 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 première 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 pour "canAda", nous affrontons non seulement les défis techniques relatifs à la conception d'un tel compilateur mais aussi nous nous créons 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

3.3 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 :

```
fichier -> with Ada.Text_IO ; use Ada.Text_IO ;
           procedure ident is <decls> begin <instrs> end <hasident> ; EOF
```

```
decl -> type ident ;
       | type ident is access ident ;
       | type ident is record <champs> end record ;
       | <identsep> : <type> <typexpr> ;
       | procedure ident <hasparams> is <decls> begin <instrs> end <hasident> ;
       | function ident <hasparams> return <type> is <decls> begin <instrs> end <hasident> ;

decls -> <decl> <decls>
        |

hasident -> ident
           |

identsep -> ident , <identsep>
           | ident

champ -> <identsep> : <type> ;

champs -> <champ> <champs>
         | <champ>

type -> ident
       | access ident

params -> ( <paramsep> )

hasparams -> <params>
            |

paramsep -> <param> ; <paramsep>
           | <param>

typexpr -> := <expr>
          |

param -> <identsep> : <mode> <type>

mode -> in
       | in out
       |

expr -> entier
       | caractère
       | true
       | false
       | null
       | ( <expr> )
       | <accès>
       | <expr> <opérateur> <expr>
       | not <expr>
       | - <expr>
       | new ident
       | ident ( <exprsep> )
       | character ' val ( <expr> )

exprsep -> <expr> , <exprsep>
          | <expr>
```

```
hasexpr -> <expr>
          |

instr -> <accès> := <expr> ;
         | ident ;
         | ident ( <exprsep> ) ;
         | return <hasexpr> ;
         | begin <instrs> end ;
         | if <expr> then <instrs> <elsif> <else> end if ;
         | for ident in <hasreverse> <expr> .. <expr> loop <instrs> end loop ;
         | while <expr> loop <instrs> end loop ;

elsif -> elsif <expr> then <instrs> <elsif>
         |

else -> else <instrs>
         |

hasreverse -> reverse
              |

instrs -> <instr> <instrs>
          | <instr>

opérateur -> = | /= | < | <= | > | >= | + | - | * | / | rem | and | and then | or | or else

accès -> ident
        | <expr> . ident
```

3.3.1 Élimination de la Récursivité à Gauche

La grammaire initiale comportait plusieurs instances de récursivité à gauche. Par exemple, la règle :

```
expr -> <accès>
accès -> ident | <expr> . ident
```

a été transformée en :

```
expr -> ident primary2
primary2 -> ( exprsep ) acces | acces
acces -> . ident acces |
```

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.

3.3.2 Factorisation à Gauche

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

```
exprsep -> <expr> , <exprsep>
exprsep -> <expr>
```

a été réécrite en :

```
exprsep -> <expr> exprsep'
exprsep' -> , <expr> exprsep' |
```

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

3.3.3 Gestion des Priorités de Calculs

Pour gérer correctement les priorités des opérations, la grammaire a été ajustée. Par exemple, les opérations de multiplication et division ont été séparées des opérations d'addition et de soustraction pour respecter leur priorité :

```
expr -> <expr> <opérateur> <expr>
```

a été réécrite en :

```
expr -> or_expr
or_expr -> and_expr or_expr'
or_expr' -> or or_expr'2 |
or_expr'2 -> and_expr or_expr'
or_expr'2 -> else and_expr or_expr'
and_expr -> not_expr and_expr'
```

[...]

```
unary_expr -> - unary_expr
unary_expr -> primary
primary -> entier
primary -> caractère
primary -> true
primary -> false
primary -> null
primary -> ( expr )
primary -> ident primary2
primary -> new ident
primary -> character ' val ( expr )
```

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

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".

3.4 Grammaire Transformée en LL(1)

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

```
fichier -> withAda.Text_IO;useAda.Text_IO; procedure ident is decls begin instrs end hasiden
decl -> type ident hasischoose ;
decl -> identsep : type_n typexpr ;
decl -> procedure ident hasparams is decls begin instrs end hasident ;
decl -> function ident hasparams return type_n is decls begin instrs end hasident ;

hasischoose -> is accorrec |

accorrec -> access ident
accorrec -> record champs end record

decls -> decl decls
decls ->

hasident -> ident
```

```
hasident ->

identsep -> ident identsep2

identsep2 -> , identsep
identsep2 ->

champ -> identsep : type_n ;

champs -> champ champs2

champs2 -> champs |

type_n -> ident
type_n -> access ident

params -> ( paramsep )

hasparams -> params
hasparams ->

paramsep -> param paramsep2

paramsep2 -> ; paramsep
paramsep2 ->

typexpr -> := expr
typexpr ->

param -> identsep : mode type_n

mode -> in modeout
mode ->

modeout -> out
modeout ->

expr -> or_expr

or_expr -> and_expr or_expr'

or_expr' -> or or_expr'2
or_expr' ->

or_expr'2 -> and_expr or_expr'
or_expr'2 -> else and_expr or_expr'

and_expr -> not_expr and_expr'

and_expr' -> and and_expr'2
and_expr' ->

and_expr'2 -> not_expr and_expr'
and_expr'2 -> then not_expr and_expr'

not_expr -> equality_expr not_expr'

not_expr' -> not equality_expr not_expr'
```

```
not_expr' ->

equality_expr -> relational_expr equality_expr'

equality_expr' -> = relational_expr equality_expr'
equality_expr' -> /= relational_expr equality_expr'
equality_expr' ->

relational_expr -> additive_expr relational_expr'

relational_expr' -> < additive_expr relational_expr'
relational_expr' -> <= additive_expr relational_expr'
relational_expr' -> > additive_expr relational_expr'
relational_expr' -> >= additive_expr relational_expr'
relational_expr' ->

additive_expr -> multiplicative_expr additive_expr'

additive_expr' -> + multiplicative_expr additive_expr'
additive_expr' -> - multiplicative_expr additive_expr'
additive_expr' ->

multiplicative_expr -> unary_expr multiplicative_expr'

multiplicative_expr' -> * unary_expr multiplicative_expr'
multiplicative_expr' -> / unary_expr multiplicative_expr'
multiplicative_expr' -> rem unary_expr multiplicative_expr'
multiplicative_expr' ->

unary_expr -> - unary_expr
unary_expr -> primary

primary -> entier
primary -> caractère
primary -> true
primary -> false
primary -> null
primary -> ( expr )
primary -> ident primary2
primary -> new ident
primary -> character ' val ( expr )

primary2 -> ( exprsep ) acces
primary2 -> acces

exprsep -> expr exprsep2

exprsep2 -> , exprsep
exprsep2 ->

hasexpr -> expr
hasexpr ->

instr -> ident instr2
instr -> return hasexpr ;
instr -> begin instrs end ;
instr -> if expr then instrs elifn elsen end if ;
instr -> for ident in hasreverse expr .. expr loop instrs end loop ;
```

```
instr -> while expr loop instrs end loop ;

instr2 -> instr3 := expr ;
instr2 -> ( exprsep ) instr3 hasassign ;
instr2 -> ;

instr3 -> . ident instr3
instr3 ->

hasassign -> := expr
hasassign ->

elifn -> elif expr then instrs elifn
elifn ->

elsen -> else instrs
elsen ->

hasreverse -> reverse
hasreverse ->

instrs -> instr instrs2

instrs2 -> instr instrs2
instrs2 ->

acces -> . ident acces
acces ->
```

3.5 Table LL(1)

	$	with Ada.Text_IO; use Ada.Text_IO;	procedure	ident	is	begin	end	;
S		S -> fichier $						
fichier		fichier -> with Ada.Text_IO; use Ada.Text_IO; procedure ident is decls begin instrs end hasident ; EOF						
decl			decl -> procedure ident hasparams is decls begin instrs end hasident ;	decl -> ident sep : type_n typeexpr ;				
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
typeexpr								typeexpr -> 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 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.

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. Chaque token est une instance de la classe 'Token', qui contient des informations telles

que le type de token et la valeur lexicale associée.

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, tandis que 'PeekingReader' permet de lire en avance dans le flux de caractères pour identifier correctement les tokens complexes.

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. Cette précision est essentielle pour garantir une analyse syntaxique sans erreur dans les étapes suivantes du processus de compilation.

5 Analyse Syntaxique

La classe 'Parser' a été conçue pour analyser les programmes écrits dans le langage "canAda". 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.

5.1 Structure et Fonctionnement

Dans notre 'Parser', chaque fonction représente un non-terminal de la grammaire. 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.

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.

5.3 Gestion des Erreurs Syntaxiques

Un aspect essentiel du 'Parser' est sa capacité à gérer les erreurs syntaxiques. Lorsque le programme source ne respecte pas les règles de la grammaire, des messages d'erreur descriptifs sont générés, 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.

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 Tests et Validation

8 Gestion de projet

8.1 Équipe de projet

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

8.2 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.

8.3 Matrice RACI

Nous avons choisi de ne pas faire de matrice RACI à cause des choix d'organisation que nous avons faits. En effet, nous avons décidé de travailler en groupe sur toutes les tâches, ce qui fait que nous sommes tous responsables de toutes les tâches et qu'il n'y a, de ce fait, pas de répartition précise des tâches.

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 3 – Matrice RACI

8.4 Répartition du Temps de Travail sur le Projet

Tâche	Lucas	Noé	Alexis
Grammaire	10h	-	-
Structure du Lexer	-	2h	2h
Lecture du fichier (token par token)	-	7h	7h
Implémentation des Token et des Tag	2h	2h	2h
Structure du Parser	2h	-	-
Gestion des erreurs	4h	4h	-
Implémentation des règles	3h	3h	3h
Structure de l'arbre abstrait	-	2h	-
Implémentation des différents nodes	-	5h	5h
Rédaction du rapport	4h	-	-
Total	25h	25h	19h

9 Conclusion

Le développement du compilateur pour le langage "canAda" a été une expérience enrichissante et formatrice. À travers les diverses phases du projet, de la conception de la grammaire LL(1) à l'implémentation de l'analyseur lexical et syntaxique, en passant par la construction de l'arbre abstrait syntaxique, chaque membre de l'équipe a contribué de manière significative. Ce projet nous a permis de consolider nos connaissances théoriques en informatique et de développer des compétences pratiques essentielles en

matière de compilation. Les défis rencontrés, notamment dans la gestion des erreurs et l'optimisation des performances, ont renforcé notre compréhension des aspects complexes de la conception de compilateurs. Ce projet a non seulement abouti à la création d'un compilateur fonctionnel pour "canAda", mais il a également été une opportunité précieuse d'apprentissage et de collaboration pour notre équipe. Nous avons pu mettre en pratique les connaissances acquises en cours de PCL1 et nous avons pu approfondir nos connaissances en programmation et en compilation et avons hâte de poursuivre ce projet en PCL2.