PCL
Projet Compilation des Langages
2023-2024

17 Janvier 2024

AULAGNIER Mathis
BOUCHADEL Maxence
CABILLOT Matthias
VATRY Etienne

Table des matières

1 Introduction

Dans le contexte du module PCL, nous avons été chargés de réaliser un compilateur pour le langage canAda. Ce rapport aborde la première partie de ce projet, qui se concentre spécifiquement sur le développement d'un analyseur lexical, d'un analyseur syntaxique et la création d'un arbre AST (Abstract Syntax Tree).

Afin de réaliser ce compilateur, nous avions la liberté de choisir notre langage de programmation. Nous avons opté pour Python comme langage, principalement pour sa facilité d'utilisation et ses bibliothèques (Re pour l'utilisation d'expression régulière par exemple). Cette décision s'est basée sur notre maîtrise collective de Python, nous permettant de démarrer efficacement ce projet complexe.

Notre compilateur doit fonctionner comme ceci : l'analyseur lexical est destiné à identifier les tokens du langage canAda. Ensuite, l'analyseur syntaxique prend le relais pour s'assurer que les programmes respectent la grammaire du langage (grammaire que nous avons optimisé). Parallèlement, la construction de l'arbre AST nous aide à représenter la structure du programme de manière visuelle et hiérarchique, facilitant les étapes de compilation ultérieures.

Ce rapport détaille les méthodes que nous avons employées, les obstacles rencontrés et les stratégies mises en place pour les surmonter. Notre but est de fournir un aperçu clair de notre démarche et de montrer comment nous avons appliqué nos compétences en informatique pour réaliser ce projet.

2 Gestion de projet

2.1 Organisation générale

Afin de réaliser ce projet, nous avons essayé d'appliquer au mieux les différents éléments vu en cours de gestion de projet en première année. Pour commencer notre groupe était composé de quatre membres :

— Mathis AULAGNIER
— Maxence BOUCHADEL
— Matthias CABILLOT
— Etienne VATRY

Nous avons au lancement du projet désigné Mathis AULAGNIER en tant que chef du projet. Il avait pour rôle de fixer les différentes tâches à réaliser. Il avait de plus la mission d'organiser et préparer les réunions en avance pour qu'elles soient les plus productives possible et d'assurer leur bon déroulement en les dirigeants. Le chef de projet s'occupait aussi de réaliser les comptes rendus et la todo list pour la prochaine réunion afin de garder une trace.

Afin d'organiser efficacement notre travail, nous avons établi une liste des tâches à chaque réunion qui était ensuite attribué à chaque personne, basée sur les échéances fixées . Par exemple, pour la semaine 50, il nous était demander de terminer l'analyseur lexical, tandis que l'analyseur syntaxique devait être presque fonctionnel.

Au début, nous avons tous contribué individuellement à l'élaboration de l'analyseur lexical, afin de nous familiariser avec le projet. Une fois cette phase d'initiation terminée, nous avons combiné nos efforts pour finaliser cet analyseur. Pour la seconde partie, concernant l'analyse syntaxique, nous nous sommes divisés en deux groupes : Maxence et Mathis étaient chargés d'optimiser la grammaire pour la rendre la plus compatible possible avec la forme LL(1), en s'appuyant sur les méthodes étudiées en cours de Traduction et sur les enseignements du Dragon Book. Parallèlement, Matthias et Etienne se sont concentrés sur la conception du code et sur la réflexion autour de la gestion et de la création de l'arbre syntaxique.

Ainsi grâce à cette organisation chacun savait exactement ce qu'il avait à faire et ce qu'il devait modifier.

2.2 Les moyens de communication

Afin de pouvoir assurer une bonne communication dans le groupe, différents moyens ont été mis en place. Nous avons commencé par la création d'un dépot gitlab avec une arborescence hiérarchisé permettant la création, l'échange et modifications simples des différents documents nécessaires au projet autre que le code. Ceci a donc permis la création de la todo list que chacun pouvait consulter et modifier simplement, mais également la consultation des comptes rendus ainsi que des préparations des réunions contenant l'ordre du jour, auxquels chacun pouvait rajouter des éléments qu'il souhaitait évoquer en réunion.

Nous avons également créé un groupe Discord, permettant de communiquer à la fois par écrit et par vocal. Ce moyen nous a été très utile pour s'échanger des informations et pour pouvoir travailler ensemble lorsque nous le désirions .

Pour ce projet, le fait d'appartenir tous au même groupe de TD a grandement facilité notre communication. Cette situation a contribué à prévenir les problèmes de coordination (éviter les "effets tunnel"), et a simplifié l'organisation de nos réunions, en nous permettant d'aligner plus aisément nos emplois du temps respectifs.

2.3 Organisation des réunions

Dans le cadre de notre projet, nous avons accordé une importance particulière à l'organisation de nos réunions. Notre objectif était de faire un point de manière hebdomadaire, soit virtuellement sur Discord soit physiquement à l'école, profitant souvent des pauses pour se retrouver.

Ces rencontres régulières nous permettaient de faire le point sur l'avancement du projet, d'aborder les difficultés rencontrées et de discuter des étapes suivantes. De plus, à chaque fois qu'une avancée significative était réalisée, nous planifions une réunion plus longue afin d'en parler entre nous de manière posé et claire.

Ces sessions étaient également l'occasion de mettre à jour notre liste de tâches, assurant ainsi une progression continue et coordonnée du projet. Cette approche structurée nous a permis de maintenir un rythme de travail efficace et de rester constamment alignés sur nos objectifs communs.

3 Le compilateur

3.1 L'analyse Lexicale

L'analyseur lexical est une étape cruciale du processus de création d'un compilateur pour le langage canAda. Son rôle est de lire le fichier source caractère par caractère, identifier les tokens qui composent le langage, et les transmettre à l'analyseur syntaxique. Implémenté sous la forme d'un automate, l'analyseur lexical a été adapté pour répondre aux exigences du projet.

L'automate de l'analyseur lexical s'acquitte de diverses tâches essentielles, débutant par l'élimination des commentaires. Les commentaires détectés sont éliminés, simplifiant ainsi le traitement du code source. Ensuite, l'analyseur lexical procède à la suppression des espaces, tabulations et retours à la ligne, normalisant ainsi la présentation du code et facilitant le processus d'analyse.

Éventuellement, l'analyseur lexical prend en charge l'affichage des erreurs lexicales en signalant des anomalies potentielles telles que la présence de caractères inconnus, d'identificateurs trop longs ou de constantes mal formées. Des limites ont été fixées de manière arbitraire, comme une taille maximale de 40 caractères pour les identificateurs et de 10 caractères pour les constantes.

```
#Definition de la taille max d'un identificateur
MAX_IDENT_SIZE = 40
#Definition de la taille max d'une constante
MAX_CONST_SIZE = 10
```

Par la suite, l'analyseur lexical découpe le code source en "tokens". Ces unités lexicales identifiées sont codées de manière spécifique pour faciliter la manipulation ultérieure. Cette notation attribue des codes numériques aux opérateurs, mots-clés, identificateurs et constantes. L'objectif sous-jacent à cette démarche est l'optimisation de l'efficacité du compilateur. L'utilisation de codes numériques pour représenter les unités lexicales offre une amélioration significative de l'efficacité du traitement. Cette approche permet une comparaison rapide d'entiers plutôt que des chaînes de caractères, contribuant ainsi à l'amélioration globale des performances du compilateur.

3.2 L'analyse Syntaxique

Dans le développement de notre compilateur pour le langage canAda, une étape fondamentale a été la transformation de la grammaire en une forme

LL(1). Cette optimisation était essentielle pour permettre une analyse descendante simple et directe. Pour y parvenir, nous avons utilisé le "Dragon Book" (Compilers : Principles, Techniques, and Tools) comme référence principale. En outre, nous avons employé Gramophone, un outil en ligne, pour vérifier et confirmer la conformité LL(1) de notre grammaire. Cet outil a été crucial pour identifier et résoudre les conflits et ambiguïtés.

Suite à cette optimisation, notre analyseur syntaxique a été conçu pour traiter la grammaire LL(1) du langage canAda. L'analyseur utilise une série de fonctions récursives, chacune correspondant à une règle spécifique de la grammaire. Cette approche permet de décomposer la séquence de tokens obtenue après l'analyse lexicale, facilitant ainsi la construction de l'arbre syntaxique.

Un élément clé de notre analyseur syntaxique est la fonction consume. Cette fonction a pour rôle de vérifier la correspondance entre le token actuel et celui attendu selon la grammaire. Si il y'a correspondance, consume extrait ce token, permettant à l'analyse de progresser. Cette approche garantit que l'analyse syntaxique se déroule de manière correcte en suivant les règles de la grammaire.

De plus, notre analyseur syntaxique est résilient dans une certaine mesure. Grâce à un système multi-pass, l'analyseur est capable de reconnaître et de corriger automatiquement certaines erreurs courantes, comme la confusion entre des mots-clés similaires (par exemple, "frocedure" au lieu de "procedure"). Lorsqu'une erreur est détectée, le système propose un token alternatif qui corrige l'erreur et relance l'analyse. Cette boucle se répète jusqu'à ce que le code soit entièrement analysé sans erreurs ou que le nombre maximum de passes soit atteint. Ce mécanisme augmente la tolérance de l'analyseur aux erreurs syntaxiques et améliore l'expérience utilisateur.

3.3 Difficultés rencontrés

Le développement de notre compilateur en langage canAda n'a pas été sans difficultés. Nous en avons rencontré plusieurs majeures tout au long du projet.

Transformation de la Grammaire en LL(1) : L'une des premières et plus importantes difficultés que nous avons rencontrées fut la transformation de la grammaire du langage canAda en une forme LL(1) afin de faciliter l'analyse syntaxique. Cette tâche s'est révélée complexe en raison de la taille

et des dépendences entre chaque règle de la grammaire originale. Éliminer la récursivité gauche et appliquer une factorisation efficace pour atteindre une grammaire LL(1) a été plutôt difficile mais en travaillant à plusieurs, nous avons réussi à résoudre ces problèmes.

Gestion de la Priorité des Opérateurs : Au début, notre grammaire ne gérait pas correctement la priorité des opérateurs, ce qui a entraîné des ambiguïtés et des erreurs dans l'analyse syntaxique. Modifier la grammaire pour résoudre ce problème était une tâche ardue, nous risquions de perdre le côté LL(1) de celle-ci. Nous avons dû rechercher des solutions alternatives, telles que l'utilisation de techniques de "coupe" dans l'arbre syntaxique, pour contourner ce problèmes sans retravailler la grammaire et sans ajouter de nouveaux états.

Parallélisation des Tâches et Collaboration : Un défi particulier dans le développement de notre compilateur a été la collaboration et la répartition des tâches, surtout en ce qui concerne l'analyseur lexical et la grammaire. L'analyseur lexical, par sa nature, pouvait être développé rapidement et efficacement par un seul membre de l'équipe, tandis que la grammaire exigeait une attention collective soutenue. Cette disparité a créé une asymétrie dans la charge de travail, où certains membres de l'équipe étaient prêts à passer à l'étape suivante, tandis que d'autres étaient encore engagés dans l'optimisation de la grammaire.

Taille et Complexité de l'Arbre AST : Enfin, une difficulté technique que nous avons dû surmonter était la gestion de la taille et de la complexité de l'arbre syntaxique abstrait (AST). À mesure que la complexité du code source augmentait, l'AST devenait proportionnellement plus grand et plus enchevêtré, rendant son analyse et le nommage des différents noeuds plus difficile.

4 Conclusion

En travaillant sur ce projet de compilateur pour le langage canAda, nous avons non seulement relevé des défis techniques, mais nous avons aussi dû nous adapter pour collaborer et mener à bien la gestion de projet. Chaque étape, qu'il s'agisse de transformer la grammaire en LL(1), de développer l'analyseur lexical, ou de construire l'analyse syntaxique et l'AST, nous a apporté une compréhension plus profonde de ce que signifie créer un compilateur.

La transformation de la grammaire en LL(1) a été particulièrement exigeante, mais elle nous a montré à quel point la théorie est cruciale dans la pratique du développement logiciel. Cette étape a nécessité de la créativité et une bonne dose de réflexion pour appliquer des concepts théoriques complexes à notre projet.

Du côté humain, ce projet a renforcé notre capacité à travailler en équipe. Nous avons appris à mieux répartir les tâches, gérer les différences de rythme de travail et à communiquer efficacement.

Pour conclure, ce projet de compilateur a été une expérience riche d'enseignements. Nous sommes fiers de ce que nous avons accompli durant ce module PCL.

5 Annexes

5.1 Diagramme de Gantt

Le diagramme de Gantt est disponible sur le gitlab du projet (aulagnier2u) dans le dossier docs sous le nom Gantt.xlsx. Voici un aperçu :

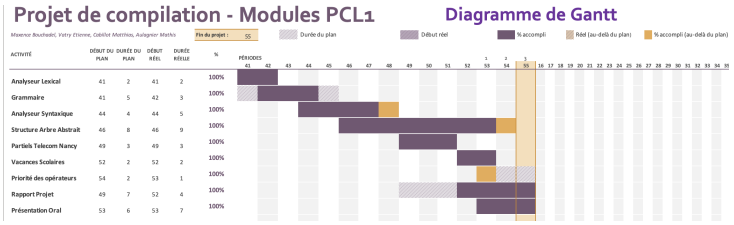

FIGURE 1 – Voici un apperçu de notre diagramme de Gantt

5.2 Grammaire

La grammaire du programme est définie comme suit :

```
FICHIER -> with ada.textio; use ada.textio;
            procedure IDENT is DECLETOILE begin INSTRPLUS end IDENTINTER;
            pointvirgule.

DECL -> type IDENT DINTEROGATION pointvirgule
      | IDENT VIRGULEIDENTETOILE: deuxpoint TYPE DPOINTEGALEXPRINTER pointvirgule
      | PROCEDURE
      | FUNC.

DECLETOILE -> DECL DECLETOILE
           | .

D -> access IDENT
   | record CHAMPS CHAMPSPLUS end record pointvirgule.

DINTEROGATION -> is D
              | .
```

```
VIRGULEIDENTETOILE -> virgule IDENT
                    | .

DPOINTEGALEXPRINTER -> deuxpointsegal EXPR
                    | .

PROCEDURE -> procedure IDENT PARAMSINTER is DECLETOILE begin INSTR INSTRPLUS end I

FUNC -> function IDENT PARAMSINTER return TYPE is DECLETOILE begin INSTR INSTRPLUS

EXPR -> TERM OPTERMETOILE POINTIDENTINTER.

OPTERMETOILE -> OP TERM OPTERMETOILE
              | .

TERM -> int
      | caractere VALEXPR
      | true
      | false
      | null
      | not EXPR
      | moins EXPR
      | IDENT PARENTHESEEXPRVIRGULEEXPRETOILEINTER
      | new IDENT.

POINTIDENTINTER -> point IDENT
                 | .

PARENTHESEEXPRVIRGULEEXPRETOILEINTER -> ( EXPR VIRGULEEXPRETOILE )
                                       | .

VALEXPR -> val EXPR
         | .

VIRGULEEXPRETOILE -> virgule EXPR VIRGULEEXPRETOILE
                   | .

INSTR -> IDENT HELP2
```

```
      | return EXPRINTER pointvirgule
      | BEGIN
      | IF
      | FOR
      | WHILE
      | int FIN
      | caractere VALEXPR FIN
      | true ;
      | false ;
      | null ;
      | not EXPR FIN
      | moins EXPR FIN
      | new IDENT FIN.

FIN -> OPTERMETOILE point IDENT deuxpointsegal EXPR pointvirgule.

HELP2 -> deuxpointsegal EXPR pointvirgule
      | ( EXPR HELP3.

HELP3 -> ) HELP
      | virgule EXPR VIRGULEEXPRETOILE ) OPTERMETOILE IDENT deuxpointsegal EXPR po

HELP -> EXPRPARENTHETOILE pointvirgule
     | IDENT deuxpointsegal EXPR pointvirgule
     | OP TERM OPTERMETOILE IDENT deuxpointsegal EXPR pointvirgule.

EXPRINTER -> EXPR
          | .

EXPRPARENTHETOILE -> ( EXPR ) EXPRPARENTHETOILE
                  | .

INSTRPLUS -> INSTR INSTRPLUS
          | .

BEGIN -> begin INSTR INSTRPLUS end .

IF -> if EXPR then INSTR INSTRPLUS IF_TAIL .
```

```
IF_TAIL -> elsif EXPR then INSTR INSTRPLUS IF_TAIL
         | INSTRPLUSELSEINTER end if .

INSTRPLUSELSEINTER -> else INSTR INSTRPLUS
                   | .

FOR -> for IDENT in REVERSEINTER EXPR troispoints EXPR loop INSTR INSTRPLUS end lo

REVERSE -> reverse .

REVERSEINTER -> REVERSE
              | .

WHILE -> while EXPR loop INSTR INSTRPLUS end loop .

CHAMPS -> IDENT IDENTVIRGULEETOILE : TYPE pointvirgule pointvirgule .

IDENTVIRGULEETOILE -> virgule IDENT IDENTVIRGULEETOILE
                   | .

CHAMPSPLUS -> CHAMPS CHAMPSPLUS
           | .

TYPE -> IDENT
      | access IDENT .

PARAMS -> ( PARAM PARAMVIRGULEETOILE ) .

PARAMVIRGULEETOILE -> pointvirgule PARAM PARAMVIRGULEETOILE
                   | .

PARAMSINTER -> PARAMS
            | .

PARAM -> IDENT IDENTVIRGULEETOILE : MODEINTER TYPE .

MODEINTER -> MODE
```

```
          | .

MODE -> in OUT .

OUT -> out
     | .

OP -> and THEN
     | or ELSE
     | equal
     | different
     | inferior
     | inferioregal
     | superior
     | superioregal
     | mult
     | division
     | rem
     | plus
     | moins .

THEN -> then
      | .

ELSE -> else
      | .

IDENT -> ident .

IDENTINTER -> IDENT
           | .
```

5.3 Exemple d'utilisation du programme

Pour illustrer l'utilisation du programme, voici un exemple simple en pseudo-code Ada :

```
with Ada.Text_IO; use Ada.Text_IO;

procedure unDebut is
   function aireRectangle(larg: integer; long: integer) return integer is
      aire: integer;
   begin
      aire := larg * long;
      return aire;
   end aireRectangle;

   function perimetreRectangle(larg: integer; long: integer) return integer is
      p: integer;
   begin
      p := larg * 2 + long * 2;
      return p;
   end perimetreRectangle;

   -- VARIABLES
   choix: integer; -- test

begin
   choix := 2;
   if choix = 1 then
      valeur := perimetreRectangle(2, 3);
      put(valeur);
   else
      valeur := aireRectangle(2, 3);
      put(valeur);
   end if;
end unDebut;
```

Voici un exrtrait de l'arbre que nous obtenons (disponible entièrement au format svg sur le dépot gitlab à cette adresse) celui-ci ne rentre pas entièrement au format png. :

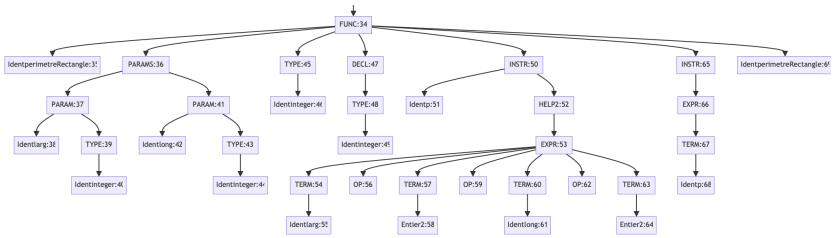

FIGURE 2 – AST pour le code test1_correct.ada