Rapport d'Activité PCL1

Augustin DESBOIS
Théo DEYGAS
Léo GERMAIN
Théo WALTER

15 Janvier 2024

1 Analyse du sujet

Pour commencer, nous avons choisi à l'unanimité de programmer en C++, puis on a découpé le projet en deux grandes parties : l'analyse lexicale et l'analyse syntaxique.

Nous avons découpé chacune de ces grandes parties en plus petites sous-parties, afin de répartir le travail en donnant à chacun des petites tâches vérifiables, et de permettre à chacun de travailler sur les deux parties du projet.

Pour l'analyse lexicale, il fallait :
- Parser le code source, c'est-à-dire enlever les commentaires et supprimer les blancs consécutifs (sauts de ligne, tabulations, espace)
- Préparer les structures de données nécessaires (tokens, arbre, table des symboles, états, ...)
- Programmer le lexeur en lui-même, qui prend en entrée le code source scanné, et qui ressort une liste de tokens.
- Créer des tests en produisant des codes canAda exhaustifs permettant de tester nos programmes tout le long du sujet

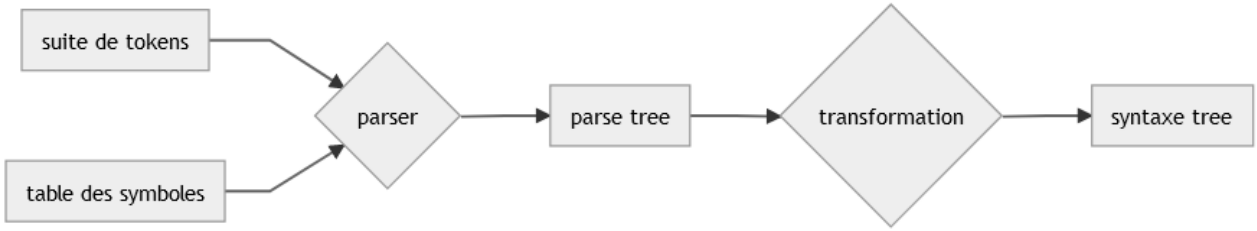

Figure 1 – Schéma d'un analyseur lexical

Pour l'analyse syntaxique (qu'on a choisi ascendante), il fallait :
- Retravailler la grammaire, de manière à supprimer tous les conflits.
- Produire et afficher la table d'analyse (LALR(1) dans notre cas), ainsi que l'automate de cette grammaire, pour nous aider à vérifier que tout fonctionne et est cohérent.
- Développer le parser, qui renvoie l'arbre de dérivation ou parse tree.
- Transformer l'arbre de dérivation en AST, en supprimant les noeuds contenant des non-terminaux.
- Etre capable d'afficher l'AST.

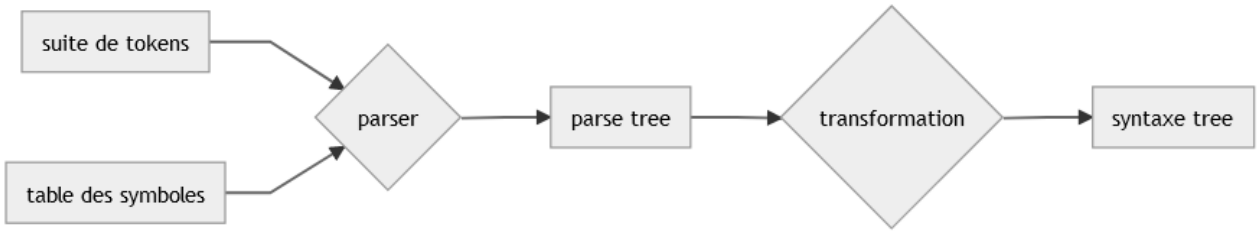

FIGURE 2 – Schéma d'un analyseur lexical

Les prévisions de l'équipe étaient de consacrer le mois de novembre à l'analyse lexicale, le mois de décembre à l'analyse syntaxique, et le début du mois de janvier servait de "période tampon" en cas de retard pris, ainsi que de préparation à la soutenance.

2 Répartition du travail

Dans cette partie, nous allons lister l'ensemble des tâches réalisées par chacun des membres du groupe, ainsi que les éventuelles difficultés rencontrées lors de ce travail.

Augustin DESBOIS :
- Programmation du scanner, qui retire tous les espaces blancs inutiles, ainsi que les commentaires du code source, en s'assurant de pouvoir l'exécuter depuis le terminal. (2h30)
- Unification du code de l'analyseur lexical, notamment après l'ajout de nouvelles classes concernant les Tokens. (4h)
- Visualisation de la structure TreeNode, permettant d'afficher l'AST en utilisant PlantUML. Le choix de PlantUML s'est fait après avoir essayé différentes méthodes de visualisation comme OpenGL ou SDL, qui n'a pas focntionné car l'arbre à afficher était trop lourd. Le parcours de l'arbre se faisait d'abord manuellement, puis automatiquement.(4h)

Théo DEYGAS :
- Ecriture de différents programmes en Ada permettant de tester notre compilateur au cours des différentes étapes du projet, de nouveaux tests ont été ajoutés tout le long du projet lorsque l'on rencontrait de nouveaux cas de figure à vérifier. (8h)
- Automatisation de l'exécution de ces tests. (4h)
- Ajout de la grammaire du sujet, et mise en place de structure permettant d'accéder aux informations de la grammaire une fois parsée. (5h)
- Ecriture d'un programme permettant d'afficher l'automate LR(1) et la table d'analyse LR(1) d'une grammaire, de même ensuite en LALR(1) (+20h)

Léo GERMAIN :
- Création de classes utiles pour le reste du projet et de leurs méthodes associées : TreeNode pour la structure d'arbre, Token et les fonctions qui en héritent pour le lexeur. Ces méthodes seront régulièrement modifiées afin de s'ajuster aux problématiques rencontrées au fur et à mesure. (8h)
- Tentative de l'écriture de l'automate LR(0) à la main. Face à la charge de travail titanesque que cela représente, il a été décidé de le réaliser algorithmiquement. (4h)
- Premiers traitements de la grammaire afin de la rendre LR(1) dans un premier temps. Des difficultés ont été rencontrées car on s'éloigne du cadre des résolutions de conflits vues en cours. (4h)
- Ecriture du rapport d'activité final. (3h)

Théo WALTER :

— Production des schémas présentés plus haut, permettant de découper le projet et faisant office de roadmap, et gestion des makefiles. (1h)
— Programmation du lexeur, qui a connu plusieurs versions qui ont évoluées, avec l'ajout des classes Token et l'apparition de nouvelles contraintes. (12h)
— Correction finale de la grammaire, afin de corriger tous les conflits. (6h)
— Implémentation de l'arbre de dérivation en utilisant la structure TreeNode, et développement du parser, qui renvoie l'arbre de dérivation (6h)
— Transformation de l'arbre de dérivation en AST.(4h)

Chacune de ces fonctionnalités se complétaient par la rédaction d'une documentation dans le Wiki du dépôt git.

On remarque que dans l'ensemble, peu de tâches ont posées de réelles difficultés. Les quelques difficultés rencontrées étaient davantage lors de la partie d'analyse syntaxique, et s'explique logiquement par notre choix de privilégier une analyse ascendante à une analyse descendante. Notre répartition du travail faisait que chacun travaillait souvent seul, mais des réunions régulières et une bonne entraide ont permis d'éviter que les difficultés soient blocantes, et nous ont permis d'aboutir à un résultat à la hauteur de nos attentes en début de projet, ce qui nous permettra de commencer la duexième partie sur de bonnes bases.

3 Annexes

3.1 Grammaire

```
fichier' -> #include_statement+##procedure_statement#
include_statement+ -> #include_statement##include_statement+#
            -> #include_statement#
include_statement -> with #ident#;use #ident#;
procedure_statement -> procedure #ident# is #decl+# begin #instr+# end #ident#;
         ->  procedure #ident# is #decl+# begin #instr+# end;
         ->  procedure #ident# is begin #instr+# end #ident#;
         ->  procedure #ident# is begin #instr+# end;
decl+ -> #decl#
      -> #decl##decl+#
decl   -> type #ident#;
       -> type #ident# is record #champs+# end record;
       -> #ident_virgule+# : #type# := #expr#;
       -> #ident_virgule+# : #type#;
       -> procedure #ident# #params# is #decl+# begin #instr+# end #ident#;
       -> procedure #ident# #params# is #decl+# begin #instr+# end;
       -> procedure #ident# #params# is begin #instr+# end #ident#;
       -> procedure #ident# #params# is begin #instr+# end;
       -> procedure #ident# is #decl+# begin #instr+# end #ident#;
       -> procedure #ident# is #decl+# begin #instr+# end;
       -> procedure #ident# is begin #instr+# end #ident#;
       -> procedure #ident# is begin #instr+# end;
       -> function #ident# #params# return #type# is #decl+# begin #instr+# end #ident#;
       -> function #ident# #params# return #type# is #decl+# begin #instr+# end;
       -> function #ident# #params# return #type# is begin #instr+# end #ident#;
       -> function #ident# #params# return #type# is begin #instr+# end;
       -> function #ident# return #type# is #decl+# begin #instr+# end #ident#;
       -> function #ident# return #type# is #decl+# begin #instr+# end;
       -> function #ident# return #type# is begin #instr+# end #ident#;
       -> function #ident# return #type# is begin #instr+# end;
champs+ -> #champs#
        -> #champs##champs+#
champs -> #ident_virgule+# : #type#;
type   -> #ident#
params -> (#param_points_virgule+#)
param_points_virgule+ -> #param#
                    -> #param#;#param_points_virgule+#
param -> #ident_virgule+# : #mode# #type#
      -> #ident_virgule+# : #type#
mode   -> in
       -> in out
       -> out
expr_virgule+ -> #expr#
            -> #expr#,#expr_virgule+#
expr   -> #expr1#
       -> #expr# or #expr1#
       -> #expr# or else #expr1#
expr1  -> #exprp#
       -> #expr1# and #exprp#
       -> #expr1# and then #exprp#
exprp  -> #expr2#
```

```
      -> not(#expr#)
expr2 -> #expr3#
      -> #expr2# = #expr3#
      -> #expr2# /= #expr3#
expr3 -> #expr4#
      -> #expr3# < #expr4#
      -> #expr3# > #expr4#
      -> #expr3# <= #expr4#
      -> #expr3# >= #expr4#
expr4 -> #expr5#
      -> #expr4# + #expr5#
      -> #expr4# - #expr5#
expr5 -> #expr6#
      -> #expr5# * #expr6#
      -> #expr5# / #expr6#
      -> #expr5# rem #expr6#
expr6 -> 1
      -> 'a'
      -> true
      -> false
      -> null
      -> (#expr1#)
      -> -(#expr1#)
      -> #ident#
      -> #ident# (#expr_virgule+#)
      -> character'val (#expr#)
instr+ -> #instr#
       -> #instr##instr+#
elsif_expr_then_instr+ -> elsif #expr# then #instr+#
                       -> elsif #expr# then #instr+# #elsif_expr_then_instr+#
instr  -> #ident# := #expr#;
       -> #ident#;
       -> #ident# (#expr_virgule+#);
       -> return #expr#;
       -> return;
       -> begin #instr+# end;
       -> if #expr# then #instr+# #elsif_expr_then_instr+# else #instr+# end if;
       -> if #expr# then #instr+# #elsif_expr_then_instr+# end if;
       -> if #expr# then #instr+# else #instr+# end if;
       -> if #expr# then #instr+# end if;
       -> for #ident# in reverse #expr#..#expr# loop #instr+# end loop;
       -> for #ident# in #expr#..#expr# loop #instr+# end loop;
       -> while #expr# loop #instr+# end loop;
ident_virgule+ -> #ident#
               -> #ident#,#ident_virgule+#
ident -> identificateur_123_test
```

Les dièses permettent d'encadrer les mots correspondant à des noms terminaux.
Cette grammaire est celle que l'on a produite, issue de la grammaire du sujet, en corrigeant les conflits pour la rendre LALR(1). Elle est toutefois toujours susceptible de changer dans la deuxième partie du projet.

3.2 Automate

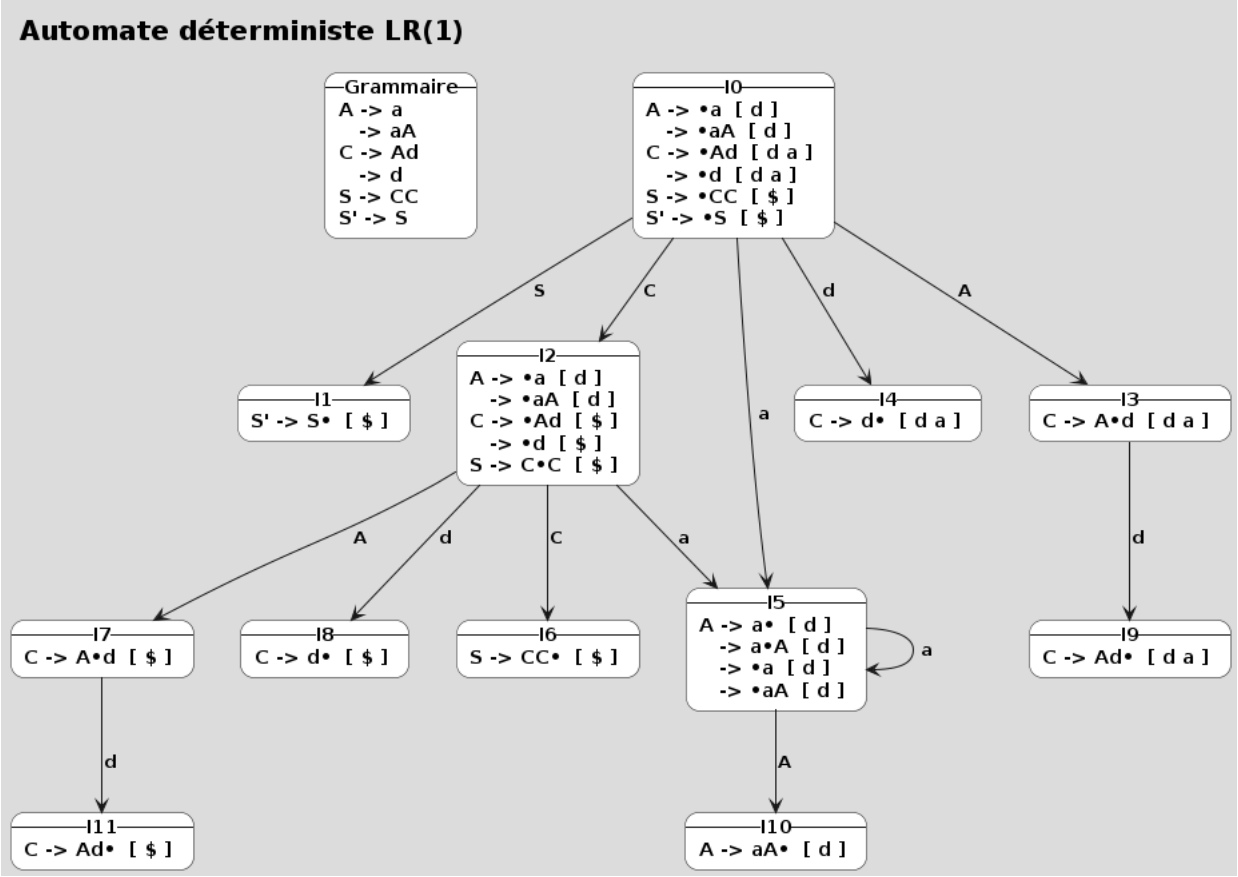

FIGURE 3 – Visualisation d'un automate sur une grammaire exemple, à l'aide de PlantUML

3.3 Structure d'arbre

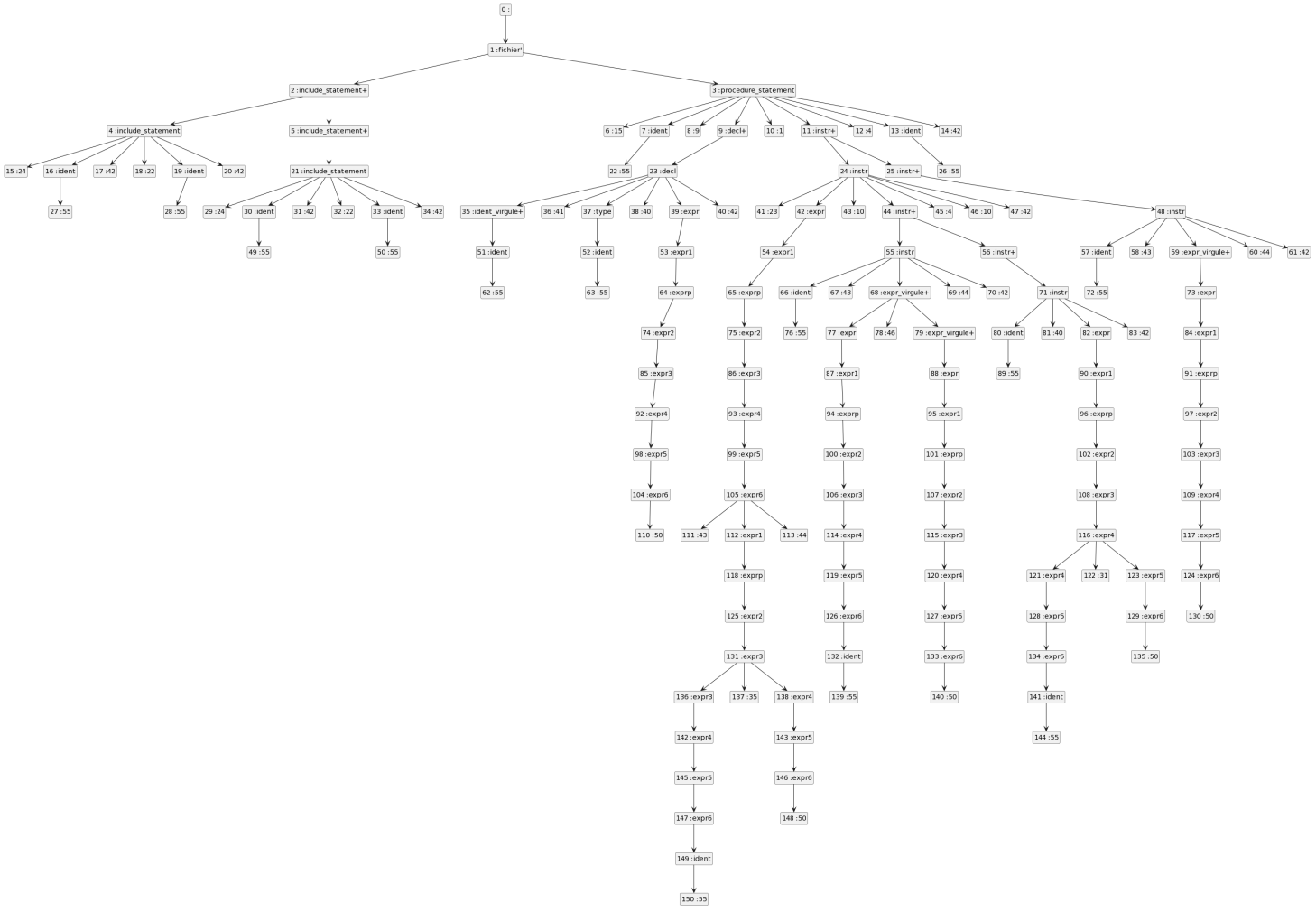

FIGURE 4 – Visualisation d'un arbre de dérivation sur un programme exemple codé en Ada, à l'aide de PlantUML

Le programme Ada en question est le suivant :

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

procedure Test is
    x: integer:=3;
begin
    while (x>0) loop
        Put(x, 0); x:=x-1;
    end loop;
    Put(" Bonne anne !");
end Test;
```