TELECOM Nancy

PROJET DE COMPILATION DES LANGAGES

PCL - SEMESTRE 1 - 2023/2024

CHRISTIAEN Adrien
DAMIENS Lana
HNID Meyssam
HOYAU Christophe

Responsables du module :
Suzanne Collin
Baptiste Buchi

Table des matières

1 Objectifs du projet

1.1 Définition du projet

Le projet en question vise la conception d'un compilateur pour le langage Ada, couvrant toutes les phases nécessaires, de l'analyse lexicale à la production du code assembleur ARM. La compilation, processus essentiel dans le domaine de l'informatique, se divise en trois étapes cruciales :

- l'analyse lexicale,
- l'analyse syntaxique,
- et l'analyse sémantique.

Dans le cadre de la première partie du projet, nous nous concentrons sur l'analyse lexicale et l'analyse syntaxique. L'analyse lexicale consiste à étudier minutieusement les caractères du langage source afin de les regrouper en unités lexicales significatives, également appelées "tokens". Ces tokens servent de base à l'analyse syntaxique, qui a pour objectif de déterminer la structure grammaticale du programme source. Au cours de cette étape, le compilateur vérifie la conformité du code avec les règles syntaxiques du langage, créant ainsi une représentation arborescente abstraite du programme.

Cette approche méthodique garantira la cohérence et la fiabilité du compilateur dans son ensemble, posant ainsi les fondations nécessaires pour réussir le projet Ada. En utilisant le langage C pour le développement, nous exploitons ses avantages pour assurer la création d'un compilateur robuste et performant, ouvrant la voie à la production d'un code assembleur ARM correct et optimisé dans les phases à venir du projet.

1.2 Language de programmation choisi

Il est à noter que, pour ce projet, nous avons eu la flexibilité d'utiliser n'importe quel langage de programmation pour implémenter le compilateur. Dans notre cas, nous avons choisi le langage C pour le développement du compilateur Ada. Cette décision a été guidée par divers facteurs, tels que la familiarité de l'équipe avec le langage C, sa performance reconnue, et son utilisation répandue dans le domaine de la compilation.

2 Analyseur lexical

2.1 Introduction

L'étape fondamentale de l'analyse lexicale consiste à convertir le flux de caractères du programme source en une séquence de tokens significatifs. Ces tokens serviront de base pour l'analyse syntaxique, où la structure du programme sera validée par rapport à la grammaire définie pour le langage canAda.

2.2 Suppression des commentaires

Avant de commencer l'implémention de notre parseur, il a fallu enlever les commentaires du fichier ada.txt. Pour cela, nous avons créé la fonction void supprimer_commentaires(FILE* fichier_entree, FILE* fichier_sortie), qui crée un nouveau fichier ada.txt en enlevant les commentaires. Nous allons d'ailleurs travailler sur ce fichier créé pour pour la suite du projet.

```
void supprimer_commentaires(FILE* fichier_entree, FILE* fichier_sortie) {
    int caractereActuel;
    int caracterePrecedent = EOF; // (End Of File)
    bool dans_commentaire = false; // si on est dans un commentaire ou pas

    while ((caractereActuel = fgetc(fichier_entree)) != EOF) {
        if (!dans_commentaire && caractereActuel == '-' && caracterePrecedent == '-') {
            dans_commentaire = true;
        } else if (dans_commentaire && caractereActuel == '\n') {
            dans_commentaire = false;
        }

        if (!dans_commentaire) {
            fputc(caractereActuel, fichier_sortie);
        }
        caracterePrecedent = caractereActuel;
    }
}
```

2.3 Structure de Stockage des Tokens

Avant de commencer l'implémentation de l'analyse lexicale, une réflexion approfondie a été menée sur la manière de stocker les tokens extraits du fichier Ada. Cette structure de stockage est cruciale pour la phase d'analyse syntaxique, où une grammaire LL(1) sera utilisée. Cette grammaire garantit qu'à chaque token, il existe un seul chemin vers le prochain token.

Nous avons choisi d'utiliser une liste chaînée pour représenter cette séquence de tokens. Cette décision découle de la nécessité d'une structure de données flexible et dynamique pour prendre en charge l'évolution de la séquence de tokens pendant l'analyse syntaxique.

```
struct element_token_valeur {
    int tokenCodageId;
    char *valeur[MAX_LENGTH];
    int line;
    int column;
    struct element_token_valeur *next;
};

struct linked_list_token_valeur {
    struct element_token_valeur *head;
};
```

`tokenCodageId` représente l'identifiant du token correspondant à un mot-clé spécifique, `valeur[MAX_LENGTH]` stocke le mot (utile pour les noms de procédures, variables, fonctions), `line` et `column` conservent la ligne et la colonne du token, et `next` pointe vers le prochain token dans la séquence.

Lors de l'initialisation, `tokenCodageId` est mis à 0, les valeurs de 1 à `MAX_LENGTH` sont initialisées à 0, `line` et `column` à 0, et `next` à `null`.

2.4 Liste des Tokens

Avant de plonger dans l'implémentation des fonctions liées à l'analyse lexicale, nous avons établi des listes exhaustives regroupant tous les types de tokens disponibles en Ada. Ces listes ont été catégorisées pour représenter les différentes classes de tokens nécessaires à la construction de l'analyseur lexical.

Chaque type de token est associé à deux listes parallèles : une contenant les chaînes de caractères représentant les tokens et l'autre stockant les indices correspondants. Cette approche facilite l'implémentation des prochaines fonctions, en permettant un accès rapide aux informations relatives à chaque token.

```
const char* file_token[] = {"with", "use", "procedure", "is", ";", "put",","};
const int file_token_index[] = {1, 2, 3, 4, 5, 52, 59};

const char* declaration_token[] = {"type", "access", "record", "end", ":", ":=",
"begin", "function", "return","Ada.Text_IO"};
const int declaration_token_index[] = {6, 7, 8, 9, 10, 11, 12, 13, 14,60};

const char* mode_token[] = {"in", "out"};
const int mode_token_index[] = {15, 16};

const char* expression_token[] = { "true", "false", "null", "(", ")", "not",
"new","\'","val"};
const int expression_token_index[] = {17, 18, 19, 20, 21, 22, 23,58,62};

const char* instruction_token[] = { "if", "then", "elsif", "else", "for",
"reverse", "loop", "while"};
const int instruction_token_index[] = {24, 25, 26, 27, 28, 29, 30, 31};

const char* operator_token[] = {"=", "/=", "<", "<=", ">", ">=", "+", "-",
"*", "/", "rem", "and", "then", "or", "."};
const int operator_token_index[] = {32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46};

const char* type_token[] = {"character", "boolean", "integer", "float", "access"};
const int type_token_index[] = {47, 48, 49, 50, 51};

const char* identifier_token[] = { "identifier"} ;
const int identifier_token_index[] = {53} ;

const char* literal_token[] = { "integer_literal", "float_literal", "char_literal",
"boolean_literal_true", "boolean_literal_false"} ;
const int literal_token_index[] = {54, 55, 56, 57, 61} ;
```

2.5 Implémentation des fonctions

Voici les différentes fonctions que nous avons implémentées :

```
int comparer_mot(const char* mot, const char** liste_mots, const int* liste_indices,
int taille_liste);

int index_token_word( char* mot);

int estUnFloat(char*mot);

int estUnEntier(char*mot);

int estUnChar(char*mot);

int estUnBoolean(char*mot);

void litMotFichier(FILE* fichier, struct linked_list_token_valeur * list_token);

void afficher_liste_tokens(struct linked_list_token_valeur * list_token);
```

2.6 Fonctions et Analyse Lexicale

Les fonctions `estUnFloat`, `estUnEntier`, `estUnChar`, `estUnBoolean` renvoient 1 si c'est le cas et -1 sinon.

La fonction `afficher_liste_tokens` permet d'afficher les tokens dans le terminal selon la structure suivante :

```
(tokenCodageId,valeur[0]) line : line
column : column
```

La fonction `comparer_mot` permet de comparer un mot avec les listes définies dans la sous-partie 2.4 et renvoie le token correspondant. Si aucun token n'est trouvé, le résultat sera le token numéro 53, qui correspond à un identifiant, c'est-à-dire une variable, un nom de fonction ou un nom de procédure.

La fonction `litMotFichier` lit le fichier Ada et à chaque espace, elle s'arrête. On utilise la fonction `comparer_mot`. Si le mot est différent de 53, il est ajouté à la liste de tokens, et on passe au mot suivant. Si le mot vaut 53, on examine en détail pour voir s'il s'agit vraiment d'un identifiant.

On parcourt le mot de taille n avec deux indices : j et k. j parcourt le mot de 0 à $n-1$ et k parcourt le mot de $j+1$ à n. On examine le mot$[j:k]$.

Si ce mot est un token différent de 53, alors :

- Si $j = 0$ et k différent de n, on ajoute le mot$[0:k]$ dans la structure, on décale j à $k+1$ et on recommence le processus de mot$[j:k]$ jusqu'à ce que $j = n-1$ et $k = n$.
- Si j différent de 0 et k différent de n, on ajoute mot$[0:j-1]$ s'il n'a pas été ajouté et mot$[j:k]$ dans la structure de token. On décale j à $k+1$ et on recommence le processus de mot$[j+1:k]$ jusqu'à ce que $j = n-1$ et $k = n$.
- Si $k = n$, on ajoute le mot$[0:j-1]$ s'il n'a pas été ajouté et le mot$[j+n]$, et on s'arrête.

Ce processus garantit la gestion correcte des identificateurs dans la liste de tokens.

2.7 Conclusion

L'analyseur lexical joue un rôle crucial dans la préparation du terrain pour l'analyse syntaxique en convertissant le flux de caractères du programme source en une séquence significative de tokens.

Dans un premier temps, la suppression des commentaires et la structure de stockage des tokens ont été des étapes préliminaires essentielles.

De plus, la création de listes détaillées de tokens pour chaque classe Ada a facilité l'implémentation des fonctions d'analyse lexicale.

Enfin, les fonctions dédiées à la reconnaissance de types spécifiques et à la gestion des identificateurs garantissent une représentation précise des éléments lexicaux.

3 Grammaire

3.1 Introduction

Cette partie décrit le processus de construction d'une grammaire en Ada à l'aide de Grammophone. Les étapes clés comprennent la prise de connaissance du sujet, l'implémentation initiale de la grammaire sur le site Grammophone, l'identification des problèmes, les corrections apportées pour rendre la grammaire LL1 et enfin l'implémentation de la grammaire en C.

3.2 Première Étape : Prise de Connaissance du Sujet

Au début du processus, une analyse approfondie du sujet a été réalisée, se concentrant sur la grammaire à implémenter en Ada. Cette étape a fourni une compréhension essentielle des exigences.

3.3 Deuxième Étape : Implémentation Initiale avec Grammophone

La première version de la grammaire a été élaborée en utilisant Grammophone. Cependant, des problèmes majeurs ont été identifiés, tels que la confusion entre les terminaux et les non-terminaux, ainsi que des erreurs dans certaines règles.

3.3.1 Version Initiale de la Grammaire

```
Fichier -> with Ada '.' Text_IO ';' use Ada '.' Text_IO ';'
procedure IDENT is DeclStar
begin InstrPlus end IdentInterro ';' EOF .

Decl -> IdentPlusVirgule ':' TYPE '(' ':=' ExprInterro ')' ';' .
Decl -> procedure IDENT ParamsInterro is DeclStar
begin InstrPlus end IdentInterro ';' .
Decl -> function IDENT ParamsInterro return TYPE is DeclStar
begin InstrPlus end IdentInterro ';' .
Decl -> type IDENT Declbis .
Declbis -> ';' .
Declbis -> is Declter .
Declter-> access IDENT ';' .
Declter -> record ChampsPlus end record ';' .

Champs -> IdentPlusVirgule ':' TYPE ';' .

Type -> IDENT .
Type -> access IDENT .

Params -> '(' ParamsPlusPointVirgule ')' .

Param -> IdentPlusVirgule ':' ModeInterro TYPE .

Mode -> IN Modbis .
Modebis -> OUT .
Modebis -> .

Expr -> ENTIER .
Expr -> CARACTERE .
Expr -> true .
Expr -> false .
Expr -> null .
Expr -> '(' EXPR ')' .
Expr -> ACCES .
Expr -> EXPR OPERATEUR EXPR .
Expr -> not EXPR .
Expr -> '-' EXPR .
Expr -> new IDENT .
Expr -> IDENT '(' ExprPlusVirgule ')' .
Expr -> character '‘' val '(' EXPR ')' .


Instr -> ACCES ':=' Expr ';' .
```

```
Instr -> return ExprInterro ';' .
Instr -> begin InstrPlus end ';' .
Instr -> if EXPR then InstrPlus '(' Boucle1 ')' '(' Boucle2 ')' end if ';' .
Instr -> for Ident in Reverse Expr '..' Expr
loop InstrPlus end loop ';' .
Instr -> while Expr loop InstrPlus end loop ';' .
Instr -> IDENT Instrbis .
Instrbis -> ';' .
Instrbis -> '(' ExprPlusVirgule ')' ';' .

Operateur -> '=' .
Operateur -> '/=' .
Operateur -> '<' .
Operateur -> '<=' .
Operateur -> '>' .
Operateur -> '>=' .
Operateur -> '+' .
Operateur -> '-' .
Operateur -> '*' .
Operateur -> '/' .
Operateur -> rem .
Operateur -> and Operateurbis .
Operateurbis -> then .
Operateurbis -> .
Operateurbis -> or Operateurter .
Operateurter -> else .
Operateurter -> .

Acces -> IDENT .
Acces -> EXPR '.' IDENT .

DeclStar -> DECL DeclStar .
DeclStar -> .

InstrPlus -> INSTR InstrPlus2 .
InstrPlus2 -> InstrPlus .
InstrPlus2 -> .

IdentInterro -> IDENT .
IdentInterro -> .

ChampsPlus -> CHAMPS ChampsPlus2 .
ChampsPlus2 -> ChampsPlus .
ChampsPlus2 -> .

IdentPlusVirgule -> IdentPlusVirgule2 .
IdentPlusVirgule2 -> ',' IdentPlusVirgule .
IdentPlusVirgule2 -> .

ExprInterro -> EXPR .
ExprInterro -> .

ParamsInterro -> PARAMS .
ParamsInterro -> .

ParamsPlusPointVirgule -> ParamsPlusPointVirgule2 .
ParamsPlusPointVirgule2 -> ';' ParamsPlusPointVirgule .
ParamsPlusPointVirgule2 -> .

ModeInterro -> MODE .
ModeInterro -> .

ExprPlusVirgule -> ExprPlusVirgule2 .
ExprPlusVirgule2 -> ',' ExprPlusVirgule .
ExprPlusVirgule2 -> .
```

```
Boucle1 -> '(' elsif EXPR then InstrPlus ')' Boucle1 .
Boucle1 -> .

Boucle2 -> else '(' InstrPlus ')' ';' .
Boucle2 -> .

Reverse -> REVERSE .
Reverse -> .
```

3.3.2 Explication des Sous-Règles Créées

DeclStar → DECL DeclStar
DeclStar → ϵ
Cette règle permet la répétition du motif DECL un nombre quelconque de fois ou aucune fois.

InstrPlus → INSTR InstrPlus2
InstrPlus → ϵ
Cette règle permet la répétition du motif INSTR au moins une fois.

IdentInterro → IDENT
IdentInterro → ϵ
Cette règle permet l'utilisation optionnelle de IDENT, c'est-à-dire 0 ou 1 fois.

IdentPlusVirgule → IdentPlusVirgule2
IdentPlusVirgule → ϵ
Cette règle permet la répétition du motif IDENT au moins une fois, avec les occurrences séparées par le terminal ",".

3.4 Troisième Étape : Correction des Bugs

La grammaire initiale était potentiellement LL1 mais comportait des erreurs significatives. Des ajustements ont été apportés, en se concentrant sur la correction des terminaux, des non-terminaux et des règles inappropriées.

3.4.1 Version Finale de la Grammaire

```
Fichier -> with Ada'.'Text_IO';'use Ada'.' Text_IO ';'
procedure IDENT is DECLSTAR
begin INSTRPLUS end IDENTINTERRO ';' EOF .

DECL -> IDENTPLUSVIRGULE ':' TYPE SUITE .
SUITE -> .
SUITE ->  ':=' EXPR  ';' .
DECL -> procedure IDENT PARAMSINTERRO is DECLSTAR
begin INSTRPLUS end IDENTINTERRO ';' .
DECL -> function IDENT PARAMSINTERRO return TYPE is DECLSTAR
begin INSTRPLUS end IDENTINTERRO ';' .
DECL -> type IDENT DECLBIS .
DECLBIS -> ';' .
DECLBIS -> is DECLTER .

DECLTER -> record CHAMPSPLUS end record ';' .

CHAMPS -> IDENTPLUSVIRGULE ':' TYPE ';' .

TYPE -> IDENT .
TYPE -> integer .
TYPE -> character .
TYPE -> boolean .
TYPE -> float .

PARAMS -> '(' PARAMSPLUSPOINTVIRGULE ')' .

PARAM -> IDENTPLUSVIRGULE ':' MODEINTERRO TYPE .
```

```
MODE -> IN MODEBIS .
MODEBIS -> OUT .
MODEBIS -> .

EXPR -> EXPRSIMPLE EXPRRESTANT .

EXPRSIMPLE -> ENTIER .
EXPRSIMPLE -> CARACTERE .
EXPRSIMPLE -> true .
EXPRSIMPLE -> false .
EXPRSIMPLE -> null .
EXPRSIMPLE -> '(' EXPR ')' .
EXPRSIMPLE -> not EXPRSIMPLE .
EXPRSIMPLE -> '-' EXPRSIMPLE .
EXPRSIMPLE -> IDENT EXPRACCES .
EXPRSIMPLE -> character ''' val '(' EXPR ')' .
EXPRACCES -> '(' EXPRPLUSVIRGULE ')' .
EXPRACCES -> '.' IDENT IDENTPOINT .
EXPRACCES -> .

EXPRRESTANT -> OPERATEUR EXPR .
EXPRRESTANT -> .

INSTR -> return EXPRINTERRO ';' .
INSTR -> begin INSTRPLUS end ';' .
INSTR -> if EXPR then INSTRPLUS  BOUCLE1  BOUCLE2  end if ';' .
INSTR -> for IDENT in REVERSE EXPR '..' EXPR
loop INSTRPLUS end loop ';' .
INSTR -> while EXPR loop INSTRPLUS end loop ';' .
INSTR -> IDENT INSTRBIS .
INSTRBIS -> ';' .
INSTRBIS -> '(' EXPRPLUSVIRGULE ')' ';' .

INSTRBIS -> IDENTPOINT ':=' EXPR ';' .
IDENTPOINT -> '.' IDENT IDENTPOINT .
IDENTPOINT -> .

OPERATEUR -> '=' .
OPERATEUR -> '/=' .
OPERATEUR -> '<' .
OPERATEUR -> '<=' .
OPERATEUR -> '>' .
OPERATEUR -> '>=' .
OPERATEUR -> '+' .
OPERATEUR -> '-' .
OPERATEUR -> '*' .
OPERATEUR -> '/' .
OPERATEUR -> rem .
OPERATEUR -> and OPERATEURBIS .
OPERATEURBIS -> then .
OPERATEURBIS -> .
OPERATEURBIS -> or OPERATEURTER .
OPERATEURTER -> else .
OPERATEURTER -> .

DECLSTAR -> DECL DECLSTAR .
DECLSTAR -> .

INSTRPLUS -> INSTR INSTRPLUS2 .
INSTRPLUS2 -> INSTRPLUS .
INSTRPLUS2 -> .

IDENTINTERRO -> IDENT .
IDENTINTERRO -> .

CHAMPSPLUS -> CHAMPS CHAMPSPLUS2 .
```

```
CHAMPSPLUS2 -> CHAMPSPLUS .
CHAMPSPLUS2 -> .

EXPRINTERRO -> EXPR .
EXPRINTERRO -> .

PARAMSINTERRO -> PARAMS .
PARAMSINTERRO -> .

PARAMSPLUSPOINTVIRGULE -> PARAM PARAMVIRGULE .

PARAMVIRGULE -> .
PARAMVIRGULE -> PARAM ',' .

MODEINTERRO -> MODE .
MODEINTERRO -> .

EXPRPLUSVIRGULE -> EXPR EXPRVIRGULE .

EXPRVIRGULE -> ',' EXPRPLUSVIRGULE .
EXPRVIRGULE -> .
BOUCLE1 ->  elsif EXPR then INSTRPLUS BOUCLE1 .
BOUCLE1 -> .

BOUCLE2 -> else  INSTRPLUS  ';' .
BOUCLE2 -> .
```

3.4.2 Modifications Apportées

Les modifications apportées à la grammaire incluent la suppression des productions liées aux mots-clés "access" et "new". D'autres ajustements ont été effectués pour améliorer la clarté et la performance de la grammaire.

3.5 Implémentation des Classes et Fonctions

Pour la mise en œuvre de la grammaire LL(1), chaque non-terminal a été traduit en une classe ou une fonction en langage C. Ces entités prennent en argument l'adresse de la liste de tokens afin de pouvoir naviguer et valider la structure syntaxique du programme source.

Chaque règle définie pour un non-terminal est représentée par une série de cas dans la classe ou la fonction correspondante. Lorsque la règle implique un déplacement à travers la grammaire, l'adresse de la liste de tokens est mise à jour pour pointer vers l'élément suivant, rendant ainsi le processus récursif.

Pour illustrer ce processus, prenons l'exemple du non-terminal `DECLBIS` dans la grammaire :

```
DECLBIS -> ';' .
DECLBIS -> is DECLTER .
```

Cette règle a été traduite dans le langage C comme suit :

```
int Declbis(struct element_token_valeur **element_token){

    int valider = 1;

    // DECLBIS ->  ;
    if ((*element_token)->tokenCodageId == 5){
        (*element_token) = (*element_token)->next;
        return valider;
    }

    // DECLBIS -> is DECLTER
    if ((*element_token)->tokenCodageId == 4){
        (*element_token) = (*element_token)->next;

        valider = Declter(element_token);
        if (valider == -1){
            return -1;
        }
```

```
    }

    return valider;
}
```

3.6 Conclusion

En conclusion, la construction de la grammaire pour le langage Ada a été une entreprise complexe mais cruciale pour le développement de l'analyseur syntaxique. La prise de connaissance approfondie du langage Ada, la tentative initiale d'implémentation avec Grammophone, l'identification des erreurs et la correction pour rendre la grammaire LL(1) ont été des étapes fondamentales.

La version finale de la grammaire est désormais mieux structurée, prenant en compte les règles spécifiques du langage Ada. Les ajustements apportés ont permis de clarifier les terminaux, les non-terminaux et les règles, créant ainsi une base solide pour l'implémentation en C.

L'implémentation des classes et fonctions en C pour chaque non-terminal de la grammaire a été entreprise de manière systématique. Chaque règle a été traduite en code, avec des mécanismes de récursivité pour suivre le flux du programme et valider la structure syntaxique.

4 Étude de l'AST

4.1 Introduction

L'Arbre Syntaxique Abstrait (AST) joue un rôle crucial dans la compréhension structurelle des programmes informatiques. Pour visualiser et analyser l'AST généré par notre analyseur syntaxique, nous avons opté pour l'utilisation de la bibliothèque `cairo.h`. Cette bibliothèque, téléchargée grâce à Homebrew, offre des fonctionnalités graphiques puissantes, nous permettant de représenter l'arbre syntaxique abstrait sous forme d'une image au format PNG. Cette approche visuelle facilite grandement la détection d'erreurs, l'analyse et la compréhension de la structure des programmes Ada que nous traitons.

4.2 Structure de l'arbre

Notre AST est constitué de nœuds, chacun représentant un élément syntaxique du programme Ada. La structure d'un nœud est définie comme suit :

```
#define MAX_WORD_SIZE 20

struct Node {
    char word[MAX_WORD_SIZE];
    struct Node** children;
    size_t numChildren;
};
```

- `word[MAX_WORD_SIZE]` : Stocke le mot qui correspond au token associé à ce nœud.
- `numChildren` : Stocke le nombre d'enfants du nœud donné.
- `children` : Stocke toutes les structures des enfants du nœud donné.

Cette structure hiérarchique permet de représenter de manière précise et organisée la séquence syntaxique du programme, où chaque nœud est associé à un élément du langage Ada.

4.3 Implémentation des fonctions

Nous avons mis en place deux fonctions principales pour manipuler et représenter notre AST :

```
struct Node* createNode(const char* word, struct Node* parent);
void drawTree(struct Node* root, cairo_t* cr, double x, double y, double level);
```

- `createNode` : Permet de créer un nœud fils à partir d'un nœud parent. Elle prend en paramètre le mot associé au token du nœud fils et le nœud parent auquel il est rattaché.
- `drawTree` : A pour objectif de représenter graphiquement notre arbre. Ses paramètres incluent le nœud racine, une structure Cairo (`cairo_t`) pour la manipulation graphique, les espacements entre les nœuds en x et y, et le niveau de déplacement selon l'axe des x. Cette fonction traverse l'arbre en profondeur, dessinant chaque nœud et établissant les connexions entre eux.

En combinant ces fonctions, nous sommes en mesure de créer une représentation visuelle intuitive de la structure syntaxique des programmes Ada analysés. Cette approche facilite l'interprétation et le débogage tout en offrant une vision claire de la hiérarchie du code source.

4.4 Utilisations dans la grammaire

Pour intégrer la représentation de l'arbre syntaxique abstrait (AST) dans notre implémentation de la grammaire Ada, nous avons enrichi chaque classe avec un second argument, permettant de lier chaque nœud de l'AST à la classe correspondante. Par exemple :

```
Expr(struct element_token_valeur ** element_token, struct Node * root)
Decl(struct element_token_valeur ** element_token, struct Node * root)
```

À l'entrée de chaque classe, nous avons introduit la création de nœuds fils à l'aide d'un appel à `malloc`, définissant le nombre maximal d'enfants. Par exemple, pour une classe avec un maximum de deux enfants :

```
root->children = (struct Node**)malloc(2 * sizeof(struct Node*));
```

La gestion du nombre d'enfants varie en fonction des spécifications de chaque classe dans la grammaire Ada. Certaines classes, telles que la classe `FICHIER`, définissent un nombre maximal d'enfants. En revanche, des classes comme `PARAMPLUSINTERRO` peuvent avoir un nombre infini de nœuds fils en raison des déclarations de variables du même type. Pour résoudre cela, nous avons alloué 100 emplacements mémoire lors de la création du tableau de nœuds fils.

À chaque vérification de token par la grammaire, nous mettons à jour les nœuds fils en conséquence. Avant de terminer l'exécution de la classe, nous spécifions le nombre d'enfants (`numChildren`), qui est crucial lors de la traversée de l'AST.

Pour lancer le processus, nous créons le nœud principal appelé `root` :

```
struct Node* root = createNode("FICHIER", NULL);
```

Nous initialisons ensuite notre arbre en appelant la classe `FICHIER` :

```
int valider Fichier (list_token, root)
```

Si la validation réussit (retourne 1), cela signifie que le fichier `ada.txt` respecte notre grammaire. En conséquence, une image de l'arbre est générée avec la fonction `cairo.h` suivante :

```
cairo_surface_write_to_png(surface, "tree.png");
```

4.5 Conclusion

L'intégration de l'arbre syntaxique abstrait dans l'analyse syntaxique de la grammaire Ada représente une avancée significative dans notre compréhension et notre débogage des programmes Ada. La visualisation graphique de la structure syntaxique offre une perspective claire et facilite la détection des erreurs. Cette approche, combinée à une gestion dynamique du nombre d'enfants, améliore considérablement la qualité de l'analyse syntaxique réalisée par notre programme. En résumé, l'utilisation de l'AST apporte une dimension visuelle précieuse à notre processus d'analyse, renforçant ainsi la robustesse de notre analyseur syntaxique pour le langage Ada.

5 Tests et Performances

5.1 Tests sans erreur

Nous avons procédé à des tests exhaustifs de la fonction `litMotFichier` en utilisant différents codes Ada. Chaque test a été conçu pour évaluer la capacité de l'analyseur lexical à extraire les tokens pertinents du code source. Ci-dessous, nous présentons un exemple de code Ada utilisé pour les tests et les résultats obtenus.

```
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 ;
 -- fin aireRectangle
choix : integer ;

begin
  choix := 2;
  if choix = 1
     then valeur := perimetreRectangle(2, 3) ;

     else valeur := aireRectangale(2, 3) ;

  end if;
end unDebut ;
```

```
(1,with)
(60,Ada.Text_IO)
(5,;)
(2,use)
(60,Ada.Text_IO)
(5,;)
(3,procedure)
(53,unDebut)
(4,is)
(13,function)
(53,aireRectangle)
(20,()
(53,larg)
(10,:)
(49,integer)
(5,;)
(53,long)
(10,:)
(49,integer)
(21,))
(14,return)
(49,integer)
(4,is)
(53,aire)
(10,:)
(49,integer)
(5,;)
(12,begin)
(53,aire)
(11,:=)
(53,larg)
(40,*)
(53,long)
(5,;)
(14,return)
(53,aire)
(5,;)
(9,end)
(53,aireRectangle)
(5,;)
(53,choix)
(10,:)
(49,integer)
(5,;)
(12,begin)
(53,choix)
(11,:=)
(54,2)
(5,;)
(24,if)
(53,choix)
(32,=)
(54,1)
(25,then)
(53,valeur)
(11,:=)
(53,perimetreRectangle)
(20,()
(54,2)
(59,,)
(54,3)
(21,))
(5,;)
(27,else)
(53,valeur)
(11,:=)
(53,aireRectangale)
(20,()
(54,2)
(59,,)
(54,3)
(21,))
(5,;)
(9,end)
(24,if)
(5,;)
(9,end)
(53,unDebut)
(5,;)
Le fichier est valide !!!!!
```

FIGURE 1 – Test de la fonction LitMotFichier

5.2 Tests avec une erreur

Voici des tests montrants une erreur dans le 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)  integer is
  p : integer;
  begin
     p := larg*2+ long*2;
  return p;
  end perimetreRectangle;

choix : integer ;

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

FIGURE 2 – Test de la fonction LitMotFichier avec un return manquant

```
Erreur : il faut le mot return
 Ligne : 3
 Colonne : 9
le fichier n'est pas valide
```

FIGURE 3 – Erreur de la fonction LitMotFichier

6 Gestion de projet

6.1 Equipe de projet

L'équipe se compose de quatre étudiants en deuxième année :
- CHRISTIAEN Adrien
- DAMIENS Lana
- HNID Meyssam
- HOYAU Christophe

Notre groupe tentait de se regrouper de manière hebdomadaire afin d'éviter un effet tunnel, tout en consultant régulièrement l'état d'avancement du projet chez chacun. Néanmoins, nous avons constaté qu'il était complexe de travailler seul sur certains aspects de ce projet. Par exemple, travailler sur la grammaire de manière séparée n'était pas optimal puisque si deux membres modifiaient une même règle, alors cela entraînait des difficultés pour fusionner les deux versions de la grammaire. Nous avons donc sur certains aspects, privilégié le travail en groupe.

Toute la partie Gestion de Projet a été réalisée sur des documents WORD, POWERPOINT et LATEX crées par le groupe. Toute la programmation a été réalisée sur le Git fourni par l'école. La rédaction du rapport a, quant à elle, été faite sur OverLeaf en LATEX.

6.2 Fiche Projet

	Dates	Octobre 2023 - Janvier 2024
	Intitulé	Projet de Compilation des Langages 1 (PCL1)
	Membres	CHRISTIAEN Adrien, DAMIENS Lana, HNID Meyssam, HOYAU Christophe
	Objectifs	Ecrire un compilateur en Ada
	Outils utilisés	Gitlab, Vscode, Overleaf...
	Travail	-Définition de la grammaire LL(1) du langage -Construction de l'AST -Tests de bon fonctionnement

6.3 Analyse du projet : Définition des objectifs

Nous avons défini nos objectifs grâce à la méthode SMART. Chacun de nos objectifs devait respecter cette méthode :

	Critère	Indicateur
S	Spécifique	L'objectif est clairement défini.
M	Mesurable	L'objectif est mesurable, par des indicateurs chiffrés.
A	Atteignable	L'objectif doit être motivant sans être décourageant et doit apporter un plus par rapport au lancement du projet.
R	Réaliste	L'objectif doit être réaliste au regard des compétences et de l'investissement de l'équipe du projet.
T	Temporellement défini	L'objectif doit être inscrit dans le temps, avec une date de fin et des jalons.

6.4 Matrice SWOT

Avant de nous lancer, nous avons réalisé une analyse des facteurs internes et externes afin de mieux nous permettre d'appréhender ce projet. Pour ce faire, nous avons réalisé une matrice SWOT.

	POSITIF	NEGATIF
INTERNE	-Bonne entente au sein du groupe -Cours de Traduction -Compétences des membres complémentaires -Expérience de deux projets durant la première année	-Potentiel manque de connaissances sur certains points : nécessité de formation entraînant une perte de temps
EXTERNE	-Professeurs et chargés de TD disponibles en cas de problèmes -Nombreuses documentation accessible	-Charge de travail importante en dehors du projet

Figure 4 – Matrice SWOT

6.5 WBS et diagramme de Gantt

WBS (Work Breakdown Structure) ou décomposition hiérarchique du travail, que l'équipe projet doit exécuter pour atteindre les objectifs et produire les livrables. Chaque lot est un objectif qui doit normalement respecter la méthode SMART.

— Un seul A par ligne
— Minimiser les C
— Minimiser les R

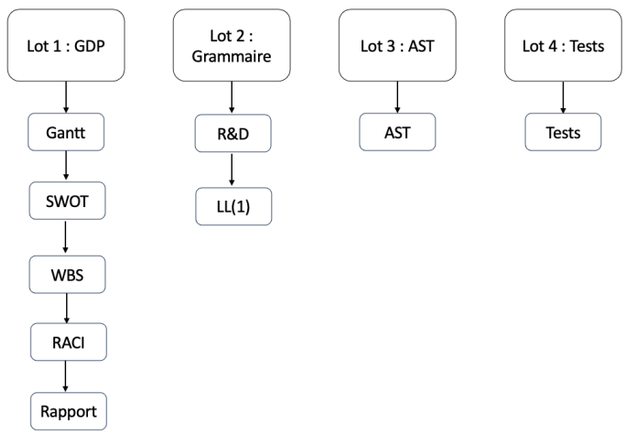

Figure 5 – Work Breakdown Structure (WBS)

Nous avons également réalisé un diagramme de Gantt afin de suivre et de contrôler au mieux l'avancée de notre projet ainsi que les différentes tâches à réaliser.

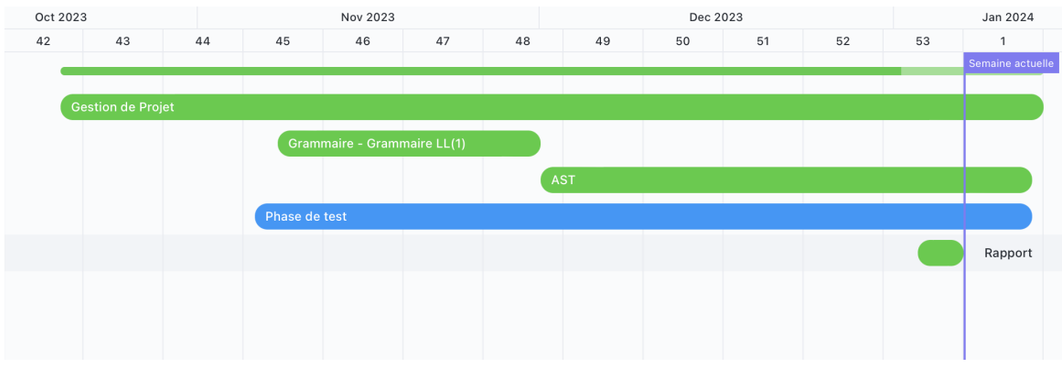

FIGURE 6 – Diagramme de Gantt (dernière prise le 01/01/24)

6.6 Comptes-rendus de réunion

Voici un exemple de compte-rendu de nos réunions. Vous pouvez retrouver l'intégralité des comptes-rendus sur le Gitlab, dans le dossier "GDP" situé à la racine du projet.

Compte Rendu Réunion
PCL-1 n°1

Date de la réunion : 08/11/2023

[Adrien Christiaën, Christophe Hoyau, Lana Damiens, Meyssam Hnid]

Participants

Membres du groupe présents :

- CHRISTIAEN Adrien
- DAMMIENS Lana
- HNID Meyssam
- HOYAU Christophe

Membres du groupe absents :

Table des matières

1 Points Abordés

1.1 Définition d'un Analyseur Lexical pour le Langage PCL

- **Présentation de l'Analyseur Lexical:** L'analyseur lexical est un programme chargé de prendre un code source en entrée et de produire une séquence de jetons (tokens) en sortie.
- **Catégories de Tokens:** Les tokens incluent les mots-clés, les identificateurs, les nombres, les opérateurs, les symboles et les textes.
- **Réalisation de l'Analyseur Lexical:** Pour implémenter l'analyseur lexical, différentes tâches doivent être réalisées, y compris la suppression des commentaires, l'élimination des espaces inutiles, la reconnaissance des tokens et la création d'une liste de paires (valeur, token).

1.2 Fonctions à réaliser

- **Suppression des commentaires:** Mise en place d'une fonction pour supprimer les commentaires dans le code source.
- **Élimination des Espaces Inutiles:** Création d'une fonction pour éliminer les espaces indésirables.
- **Reconnaissance des Tokens:** Développement de fonctions pour reconnaître et associer les tokens aux valeurs correspondantes.
- **Création de la Liste de Paires (Valeur, Token)** Fonction pour lier les valeurs reconnues aux tokens correspondants et créer une liste de paires.
- **Tâches à Réaliser:** Finalisation des fonctions de reconnaissance des tokens, implémentation de la fonction de suppression des commentaires, création de la liste de paires valeur-token, intégration de l'analyseur lexical dans le fichier analyseur.cpp, écriture du fichier Makefile pour compiler le programme.

2 TO-DO LIST Prochaines Étapes

Point	Responsable	Tâches à Réaliser
Code Ada	[Tous]	- Apprendre le code Ada
Header	[Christophe et Adrien]	- Finir le Header.
Suppression des Commentaires	[Lana]	- Implémenter la fonction de suppression des commentaires dans le code source.
Élimination des Espaces Inutiles	[Lana]	- Développer la fonction d'élimination des espaces inutiles dans le code source.
Reconnaissance des Tokens	[A voir]	- Élaborer les fonctions de reconnaissance des tokens et d'association avec les valeurs correspondantes.
Création de la Liste de Paires (Valeur, Token)	[A voir]	- Créer une fonction pour lier les valeurs reconnues aux tokens correspondants et créer une liste de paires.

2.1 Prochaine réunion

- **Date:** [09/11/2023]

7 Conclusion Générale

À travers ce projet de compilation des langages, nous avons exploré différentes facettes du processus de compilation, en commençant par l'analyse lexicale, en passant par l'analyse syntaxique, la définition de la grammaire, et en terminant par la création de l'arbre syntaxique abstrait (AST).

L'analyse lexicale a permis de créer un analyseur capable de convertir le flux de caractères du programme source en une séquence de tokens significatifs. En parallèle, la définition de la grammaire en utilisant Grammophone a été une étape cruciale pour définir la structure syntaxique du langage canAda.

La création de l'arbre syntaxique abstrait (AST) a été une étape majeure dans le projet. L'AST offre une représentation structurée du code source, ce qui facilite la compréhension et la manipulation de la structure du programme.

Les tests ont joué un rôle essentiel dans la validation du compilateur, permettant de détecter et de corriger d'éventuelles erreurs. La gestion de projet a assuré une progression efficace et organisée, permettant de respecter les délais et de répondre aux exigences du projet.

En conclusion, ce projet a été une opportunité d'approfondir nos connaissances en compilation des langages et de mettre en pratique les concepts théoriques appris en cours. Il a également renforcé nos compétences en programmation et en gestion de projet.