Faculté des sciences & technologies

Compilation

Fabien Nicot - Clément Jarrige

18 octobre 2017

Sommaire

Introduction

Dans le cadre de l'UE de Compilation nous devions implémenter un analyseur lexicale et un analyseur syntaxique. Nous avons développés ces analyseurs en C++.

Nous avons dans un premier temps implémenté l'analyseur lexical, puis l'analyseur syntaxique. Toutefois les deux analyseurs ont été développés dans le même programme et s'exécutent à la suite. Enfin nous présenterons très brièvement les différents arguments qu'accepte notre compilateur.

1 Analyseur Lexical

Pour fonctionner l'algorithme de l'analyse lexical à besoin de plusieurs automates, soit un automate par unités lexicale.

1.1 Les automates

Pour que la gestion des automates soit plus facile, nous avons décidé de stocker les automates dans des fichiers CSV. Cela permet également de rendre notre analyseur utilisable pour n'importe quel langage.

Les automates sont stockés sous la forme de tableaux d'états/transitions :

etatTransitions	a	e	f	i	l	n	o	q	r	s	t	u
0	15		3							1	8	
1				2								
2						20						
3	4											
4				5								
5									6			
6		7										
7												
8	9											
9						10						
10											11	
11								12				
12												13
13		14										
14												
15					16							
16							17					
17									18			
18										19		
19												
20							21					
21						22						
22												

FIGURE 1 – Tableau d'états/transitions de l'automate des mots clés

Chaque automate possède une liste des états terminaux. Un mot est donc considéré non reconnue par l'automate, si lors de la lecture du mots aucune transition n'existe pour le symbole en cours de lecture ou si le mot ne se termine pas sur un état terminal.

Nous avons donc implémenté un Parser CSV (*CSVParser*) capable de lire un fichier CSV. Ce parser peut lire des listes, des tableaux à deux dimensions.

Une classe *Automate* a été implémenté, elle possède un tableau d'états/transitions, la liste des états terminaux...

Cette classe possède aussi une méthode *load()* qui, à l'aide du *CSVParser*, lit un fichier .csv pour initialiser l'objet automates.

1.2 L'algorithme

La classe *Automate* implémente une méthode qui lit un mot et retourne vrai si le mot est reconnu par l'automate : *analyseMot(char * mot)*.

```
bool Automate::analyseMot(char *mot){
    int etatActuel = etatInitial; // Initialise a l'etat initial
    int index = 0;

    // On parcourt le mot symbole par symbole
    for(int j = 0; mot[j] != '\0'; j++){
        // On cherche l'indice du symbole dans l'alphabet de l'automate.
        index = indexOf(alphabet,mot[j],nbLettre);
        // Si le symbole n'est pas connue de l'automate -> erreur
        if(index == -1){
            return false;
        }
        // On cherche le nouvel etat avec le tableau d'etats/transitions
        etatActuel = tabTransition[etatActuel][index];
        if(etatActuel == -1){ // S'il n'y a pas de transition -> erreur
            return false;
        }
    }

    // Si l'etat actuel a la fin du mot n'est pas terminal -> erreur
    if(indexOf(listeEtatTerminaux, etatActuel, nbEtatTerminaux) == -1){
        return false;
    }
    return true;
}
```

FIGURE 2 – Code de la methode *analyseMot(char * mot*

Comme nous avons un automate par unité lexicale, l'analyseur lexical exécute cette méthode pour chaque automates jusqu'à ce que le mot soit reconnue par un automate. Si tous les automates ont lu le mot mais qu'aucun ne l'a reconnu alors l'analyseur lexical renvoie une erreur.

Lorsqu'un mot est reconnu par un automate, l'analyseur associe, dans une map, le mot lu au nom de l'automate (= unité lexicale). La table ainsi créer est la table des correspondance.

```
bool analyseLexicale(vector<Automate *> automatesAnalyseLexicale,
↪  vector<string> argv, vector<pair<string, string> >
↪  &tableCorrespondance){
    vector<Automate *>::iterator itAutomates;
    bool inconnu;
    bool ok = true;

    char * chaineActuelle;

    for(vector<string>::iterator it = argv.begin(); it !=
    ↪  argv.end(); it++){
        chaineActuelle = (char*)(*it).c_str();
        inconnu = true;

        for(itAutomates = automatesAnalyseLexicale.begin();
        ↪  itAutomates != automatesAnalyseLexicale.end();
        ↪  itAutomates++){
            if((*itAutomates)->analyseMot(chaineActuelle)){
                tableCorrespondance.push_back( make_pair(
                ↪  chaineActuelle, (*itAutomates)->getName()));
                inconnu = false;
                break;
            }
        }
        if(inconnu){
            tableCorrespondance.push_back(
            ↪  make_pair(chaineActuelle, "INCONNU"));
            ok =! inconnu;
        }
    }
    return ok;
}
```

FIGURE 3 – Code de l'analyseur lexical

Enfin pour s'assurer que l'automate *ID* soit exécuter en dernier, nous avons ajouter une priorité aux automates. Ainsi le tableau *automatesAnalyseLexicale* est trié par ordre de priorité avec l'automate *ID* possédant une priorité plus faible que les autres automates.

2 Analyseur Syntaxique

2.1 Une grammaire LL(1)

Avant de commencer à développer l'analyseur syntaxique, nous avons vérifié que la grammaire décrit un langage LL(1)

<liste_déclarations> : :=<une_déclaration><liste_déclarations> | vide :
(SUIVANT(<liste_déclarations>) = { cin, cout, id, if, '}' } ∩ PREMIER(<une_déclaration><liste_déclarations>) = { int }) = ∅

<liste_instructions> : :=<une_instruction><liste_instructions> | vide :
(SUIVANT(<liste_instructions>) = { '}' } ∩ PREMIER(<une_instruction><liste_instructions>) = { cin, cout, id, if }) = ∅

<une_instruction> : :=<E/S> | <affectation> | <test> :
(PREMIER(<E/S> = { cin, cout } ∩ PREMIER(<affectation>) = { id } ∩ PREMIER(<test>) = { if }) = ∅

<operateur> : := < | > | <= | >= | = | **!=** :
L'intersection entre les PREMIER() de chacun de ces opérateurs donne un ensemble vide.

Donc le langage est LL(1). Nous pouvons commencer à implémenter l'algorithme de l'analyseur syntaxique et créer la table d'analyse correspondant à la grammaire.

2.2 Implémentation

Pour rester dans la même philosophie que l'analyseur lexical, nous avons stockée la table d'analyse dans une fichier CSV.

Nous avons implémenté une classe *TableAnalyse* elle possède un tableau à deux dimensions (la table d'analyse) et elle implémente quelque méthode utiles pour l'analyseur syntaxique.

Nous avons enfin implémenté l'algorithme de l'analyseur syntaxique vu en cours. Mais cette version de l'algorithme ne prenais pas en compte la table de correspondance créer par l'analyseur lexical. Nous ne pouvions pas prendre en compte les mots appartenant à l'unité lexical NOMBRE (0,125...) et ID (tous mot commençant par une lettre n'étant pas reconnu par les autres automates).

Voici notre implémentation de l'analyseur syntaxique :

```
bool analyseSynthaxique(TableAnalyse *tableAnalyse,
↪  vector<pair<string, string> > tableCorrespondance)
{
    string value_vide = "vide";
    unsigned int index = 0;
    stack<string> pile;
    pile.push("\0");
    tableCorrespondance.push_back(make_pair("\0","\0"));
    pile.push(tableAnalyse->getAxiome());
    string X, a = tableCorrespondance[index].first;
    vector<string> regle; string str_regle;

    bool accepte = false, erreur = false;
    while(!accepte && !erreur){
        X = pile.top();
        if(tableAnalyse->isNonTerminal(X) && X != "\0"){
            regle = tableAnalyse->at(X, a);
            if(regle.empty()){ // Si [X, a] ne donne pas de regle
            ↪  on test [X, uniteLexical[a]]
                regle =
                ↪  tableAnalyse->at(X,tableCorrespondance[index].second);
            }
            if(!regle.empty()){
                pile.pop();
                str_regle = "";
                for(vector<string>::reverse_iterator it =
                ↪  regle.rbegin(); it != regle.rend(); it++){
                    pile.push((*it));
                    str_regle = (*it) + str_regle;
                }
                cout << X << " ::= " + str_regle << endl;
            }else{
                cout << "Erreur a la lecture du symbole : " << a
                ↪  << endl;
                erreur = true;
            }
```

FIGURE 4 – Code source de l'analyseur syntaxique - 1

```
        }else{ // cas ou X est terminal ou vide
            if( X == "\0"){
                if(a == "\0"){
                    accepte = true;
                }else{
                    cout << "Erreur fin de chaine prematuree" <<
                    ↪  endl;
                    erreur = true;
                }
            }else{ // cas ou X est terminal
                if(X.compare(a) == 0 ||
                ↪  tableCorrespondance[index].second.compare(X)
                ↪  == 0){
                    pile.pop();
                    index++;
                    a = tableCorrespondance[index].first;
                }else if(X.compare(value_vide) == 0){
                    pile.pop();
                }else{
                    cout << "Erreur symbole inattendu : " << a <<
                    ↪  ", le symbole attendu est " << X << "."
                    ↪  << endl;
                    erreur = true;
                }
            }
        }
    }
    return accepte;
}
```

FIGURE 5 – Code source de l'analyseur syntaxique - 2

3 Utilisation

L'analyseur lexical peut fonctionner pour tous les langages et l'analyseur syntaxique pour toutes les grammaires décrivant un langage LL(1). Il suffit pour cela de créer des tables d'états/transitions pour reconnaître le lexique du langage, ainsi qu'une table d'analyse pour reconnaître la syntaxe du langage.

C'est pour cela que nous pouvons passer des arguments à notre Compilateur pour lui spécifier le chemin des fichier CSV des tableaux d'états/transitions ou de la table d'analyse.

Nous allons maintenant présenter rapidement les arguments qu'accepte notre programme :

- -i "chaîne de caractère" : donner une chaîne à compiler.
- -f "chemin du fichier" : spécifier un fichier source à compiler.
- -l "dossier contenant tous les tableaux d'états/transitions pour l'analyse lexicale" : spécifier le dossier des automates.
- -s "chemin vers la table d'analyse" : spécifier le fichier de la table d'analyse.
- -v : pour vérifier qu'il n'y a pas d'incohérence dans les fichiers.

Si -l est spécifier l'analyse lexicale sera effectué.

Si -s est spécifier l'analyse syntaxique sera effectué. Cependant l'analyse syntaxique ne peut pas être exécutée que si l'analyse lexicale a été exécuté.

Conclusion

Ce projet nous a permis de mieux comprendre les algorithmes des analyses lexicale et syntaxique.

Nous nous sommes efforcé tout au long de ce projet, d'obtenir un programme fonctionnant pour tous les langages LL(1). Pour se faire nous avons utilisé les fichier CSV pour stocker toutes les informations utiles à nos analyseur.

Dans le dossier de notre projet vous trouverez :
- src/ : le dossier contenant toutes nos sources.
- AutomatesLexical_1/ : le dossier contenant les automates décrivant le lexique du langage de la première partie de projet.
- AutomatesLexical_2/ : le dossier contenant les automates décrivant le lexique du langage de la deuxième partie de projet.
- TableAnalyse_2/ : le dossier contenant la table d'analyse pour la deuxième partie du projet.