# Algorithme Earley 2016/2017
# Rapport

*Abderrazak ZIDANE* [1]

1. zidane.rezzak@gmail.com

# Introduction

Durant mon parcours d'étudiant en informatique, j'ai tout d'abord découvert les compilateurs, puis j'ai appris à les aimer, et aujourd'hui je suis amené à les construire et à les développer. Ce projet est une grande opportunité pour moi d'agrandir mes acquis et connaissances, et de comprendre les aspects théorique et logiciels tout en faisant ce à quoi j'aspire. De plus, les problèmes rencontrés durant ce travail, mon appris à faire de la recherche et à lire de la documentation et thèse.

Depuis le séminaire de Donald Knuth[6] sur l'analyse syntaxique LR en 1960, puis les travaux de DeRemer[1, 2] pour l'extension vers LALR, nous somme capable de générer automatiquement des analyseurs syntaxiques pour une grande variété de grammaire non contextuelle. Par contre, plusieurs analyseur syntaxique sont écrit manuellement, car souvent, on a pas le luxe de concevoir une grammaire adapter a un générateur d'analyseur syntaxique. Mais aussi, c'est très claire que les concepteurs de langage informatique, n'écrivent pas naturellement des grammaire LR(1).

Une grammaire, non seulement elle définit la syntaxe du langage, mais aussi, c'est le point d'entrés vers la définition de la sémantique, et souvent la grammaire qui facilite la définition de la sémantique n'ai pas LR(1). Ceci est montré pas le développement de la spécification de JAVA. La premiers édition de cette spécification[4] montre l'effort mis dans la sémantique pour que et la grammaire soit LALR(1), par contre dans la 3ème édition de cette spécification[5], la grammaire est (grandement) ambiguë, et ceci montre la difficulté pour faire les transformations adéquates.

Puisque c'est difficile de construire (ou maintenir) des grammaires LR(1) qui garde la sémantique voulu au départ, les développeurs se sont intéressé a d'autre algorithme comme CYK[9], Earley[3], GLR[8], qui eux ont été développer pour traitement de langage naturelle a la base (gère l'ambiguïté).

Quand on utilise la grammaire comme point d'entré pour la définition de la sémantique, on distingue souvent entre **reconnaisseur syntaxique** qui détermine simplement si un mot appartient ou pas a la grammaire, et **analyseur syntaxique** qui retourne la dérivation détaillé d'un mot si elle existe.

Dans leurs versions de base, l'algorithme CYK et Earley sont des reconnaisseurs syntaxique, alors que GLR est un analyseur syntaxique. Sauf que l'analyseur syntaxique GLR de Tommita a une complexité polynomiale infinie.

Par contre Elizabeth Scott[7], a crée deux algorithme d'analyse syntaxique basé sur Earley, ayant une complexité cubique dans le pire des cas.

Nous allons tout d'abord comprendre les méthodes d'Elizabeth Scott, est proposer une application écrite en C++ qui implémente ces méthode la.

# Introduction

## Qu'est ce que l'alogirthme Earley ?

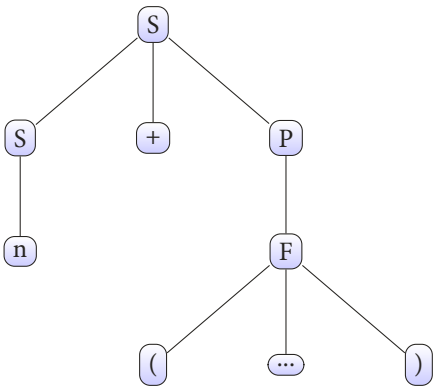
L'algorithme Earley [1], est un parmi les nombreux algorithmes d'analyse syntaxique, est comme tout ces algorithmes, **Earley** a besoin d'une grammaire

```
S = S + P
  | P
P = P * F
  | F
F = ( S )
  | n
```

... pour transformer une chaine de mot ...

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| n | + | ( | n | * | n | + | n | ) |

... en un jolie arbre syntaxique (AST)




Jusqu'à présent rien de spéciale comparé au autre algotithme qu'on conné.

## Pourquoi devrions-nous nous soucier ?

Le plus grand avantage d'Erley, est sans doute son accessibilité. La plus part des autres algorithme offre une restriction sur le type de grammaires, utilisé une grammaire récursif gauche et en rentrera dans une boucle infini, utilisé un autre type et l'algorthme ne marchera plus. Biensur il y a des counternement qu'on peux faire, mais souvant sa complique d'avantage l'algorithme et rendra le travaille plus complexe.

Pour dire simple Earley marche avec tout.

D'une autre part, pour avoir cette généralité, nous devons sacrifié la vitesse. On ne pourra donc pas rivaliser avec des algorithme comme Flex/Bison en terme de rapidité brute. Ce n'ai si grave, puisque

1. text

— Earley a une complixité cubique O(3), dans les pires des cas, ces meme cas qui ne pourons etre tréte pas d'autre algorithme
— La plus part des grammaire simple aurons une complexité linaire
— Meme la pire grammaire nom-ambigue pourra etre analyser en O(3)

# Vocabulaire

Du point de vu de l'algorithme Earley, la grammaire est constitué de règle. Voici un exemple de règle

```
S = S + P
```

S est un symbole non terminale, et tout ce qui ne commence pas avec une lettre majuscule est considéré comme étant un symbole terminale (le symbole + dans notre exemple).

Dans le jargons d'Earley, il y a la notion d'items, Voici un item :

```
A = B • * C  (4)
```

c'est juste une règles de grammaire avec des informations en plus, qui représente une reconnaissance partielle.
— Le point représente la position courante qui indique jusq'ou on a parvenu a parser
— Le chiffre 4 représente la position initial sur l'entré qu'on veux parser
De plus Earley introduit la notion d'ensemble d'items, chaque ensemble est carterisé par le fait que les item qui y sont associé, ont la même position courante

Et tout ces ensemble la, sont souvent nommé table Earley

# Prédiction, Lecture et Complétion

Pour construire la table Earley, on a besoin de définir trois opérations élémentaire qui s'applique sur un item pour produire un autre item :
— Prédiction : le symbole a droite du point est un nom terminal, on ajoute les réglés de ce symbole au même ensemble
— Lecture : le symbole a droite du point est un terminal, on regarde si se symbole coïncide avec la position courante, si oui, on ajoute cette item a l'ensemble suivant.
— Complétion : in n'ya rien a droite du point, et dans se cas il ya reconnaissance partielle, on regarde l'item parent, et on l'ajoute a cette ensemble

# Exemple de construction de table d'Earley

Reprenons cette grammaire :

```
S = S + P
| P
P = P * F
| F
F = ( S )
| n
```

on veux reconnaitre l'entrée

| n | + | ( | n | * | n | ) |
|---|---|---|---|---|---|---|

A l'étape 0, le calcule démarre avec l'ensemble E(0) et les règles de l'axiome 'S'

| **E(0)** |
|---|
| S = •S + P (0) |
| S = •P (0) |

la prédiction du premier item de E(0) nous donnera les mêmes 2 items de E(0), et donc pas besoin de faire quoi que se sois, donc une la grammaire récursive gauche ne posera pas de problème a notre algorithme.

La prédiction du deuxième item de E(0) générera deux nouveaux items :

| E(0) |
|---|
| S = •S + P (0) |
| S = •P (0) |
| P = •P * F (0) |
| P = •F (0) |

Le prédiction du 3ème item de E(0) ne sert a rien. La prédiction du 4ème item de E(0) générera deux nouveaux items supplémentaire :

| E(0) |
|---|
| S = •S + P (0) |
| S = •P (0) |
| P = •P * F (0) |
| P = •F (0) |
| F = •( S ) (0) |
| F = •n (0) |

La Lecture du 5ème item de E(0) échoue puisque le symbole ne correspond pas a l'entrée.

La lecture du 6ème item se fait avec succès, est génère un nouveau item dans l'ensemble suivant E(1)

| E(1) |
|---|
| F = n• (0) |

On a traité tout les items de E(0), attaquons nous a l'ensemble E(1)

La Complétion du premier item de E(1), nous fait ajouter le 4ème item de E(0) a E(1) :

| E(1) |
|---|
| F = n• (0) |
| P = F• (0) |

la Complétion du deuxième item de E(1), nous fait ajouter le deuxième et troisième item de E(0) dans E(1)

| E(1) |
|---|
| F = n• (0) |
| P = F• (0) |
| S = P• (0) |
| P = P• * F (0) |

...

Au finale notre table Earley ressemblera a :

| E(0) |
|---|
| S = •S + P (0) |
| S = •P (0) |
| P = •P * F (0) |
| P = •F (0) |
| F = •( S ) (0) |
| F = •n (0) |

| E(1) |
|---|
| F = n• (0) |
| P = F• (0) |
| S = P• (0) |
| P = P• * F (0) |
| S = S• + P (0) |

| E(2) |
|---|
| S = S + •P (0) |
| P = •P * F (2) |
| P = •F (2) |
| F = •( S ) (2) |
| F = •n (2) |

| E(3) |
|---|
| F = ( •S ) (2) |
| S = •S + P (3) |
| S = •P (3) |
| P = •P * F (3) |
| P = •F (3) |
| F = •( S ) (3) |
| F = •n (3) |

| E(4) |
|---|
| F = n• (3) |
| P = F• (3) |
| S = P• (3) |
| P = P• * F (3) |
| S = S• + P (3) |
| F = ( S• ) (2) |

| E(5) |
|---|
| P = P * •F (3) |
| F = •( S ) (5) |
| F = •n (5) |

| E(6) |
|---|
| F = n• (5) |
| P = P * F• (3) |
| S = P• (3) |
| P = P• * F (3) |
| F = ( S• ) (2) |
| S = S• + P (3) |

| E(7) |
|---|
| F = ( S )• (2) |
| P = F• (2) |
| S = S + P• (0) |

On arrive donc a la fin (TODO : condition de reussite)

# Que va t'on faire ?

Nous allons crée un programme qui va avoir en entré une grammaire, est aura en sorti un analyseur syntaxique suivant l'algorithme Earley.

Par la suite en va modifier cette algorithme pour notre besoin (TO DO :)

# Développement de l'outils

Le programme sera écrit en C++, mais sera facilement traduit dans d'autre langage si nécessaire.

## Petite Pré-analyse avant de commencer

Notre outils aura en entrée une grammaire est en sortie, on aura un analyseur syntaxique. Plusieurs question se sont posés durant le développement de cette outils, Voila un récapitulatif des décision prise :

- Nous appellerons notre programme earley
- La première entrée de notre programme, sera un fichier, qui contiendra la description de la grammaire en format Yacc
- La deuxième entrée de notre programme sera un deuxieme fichier, qui contiendra la chaine a analyser.
- La sorti sera un troisième fichier contenant L'AST
- Les noms de varibale, et de fonction serons en anglais, referai vous donc au dictionnaire (a faire)

Le programme sera exécuter en ligne de commande suivant la syntaxe suivant :

earley <file1> <file2> <file3>

file1 : le fichier de grammaire file2 : le fichier contenant la chaine a analysé file3 : le fichier contenant l'AST

## Le Format Yacc

Pour la grammaire que nous fournissons au programme, nous utiliserons le format Yacc simplifié suivant :

Une règle de grammaire a la forme :

A : BODY ;

A représente un symbole non-terminal, et BODY représente une séquence de zéro ou plus de terminaux et non-terminaux. La côlon et le point-virgule sont la ponctuation de Yacc.

un symbole terminal doit être déclaré comme telle au debut du fichier :

%token n1 n2 p

Si un symbole non terminal correspond à la chaîne vide, cela peut être indiqué de manière évidente :

A : ;

le symbole de départ est considéré comme le côté gauche de la première règle de grammaire dans la section des règles

# Bibliographie


[1] Frank L DeRemer and Thomas J. Pennello. Efficient computation of lalr(1) look-ahead sets. In *ACM Trans. Progam. Lang. Syst., 4(4) :615–649, October 1982*, 1982.

[2] Franklin L DeRemer. *Practical translators for LR(k) languages*. PhD thesis, Massachussetts Institute of Technology, 1969.

[3] J Earley. An efficient context-free parsing algorithm. In *Communications of the ACM, 13(2) :94–102*, February 1970.

[4] Bill Joy James Gosling and Guy Steele. The java language specification. Technical report, Addison-Wesley, 1996.

[5] Guy Steele James Gosling, Bill Joy and Gilad Bracha. The java language specification third edition. Technical report, Addison-Wesley, 2005.

[6] Donald E Knuth. On the translation of languages from left to right. In *Information and Control 8, (6) :607–639*, 1965.

[7] Elizabeth Scott. *SPPF-Style Parsing From Earley Recognisers*. PhD thesis, University of London, 2008.

[8] Masaru Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, Boston, 1986.

[9] D H Younger. Recognition of context-free languages in time n3. In *Inform. Control, 10(2) :189–208*, February 1967.