

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[9] sur l'analyse syntaxique LR en 1960, puis les travaux de DeRemer[1, 2] pour l'extension vers LALR, nous sommes 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'entrées 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 première édition de cette spécification[7] montre l'effort mis dans la sémantique pour que la grammaire soit LALR(1), par contre dans la 3ème édition de cette spécification[8], 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 intéressés à d'autres algorithmes comme CYK[12], Earley[4], GLR[11], qui eux ont été développés pour le traitement de langage naturelle à la base (gère l'ambiguïté).

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

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

Par contre Elizabeth Scott[10], a créé deux algorithmes d'analyse syntaxique basés sur Earley, ayant une complexité cubique dans le pire des cas.

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

Du Reconnaisseur a l'Analyseur syntaxique

Il n'y a pas d'analyseur ou reconnaisseur syntaxique de complexité linéaire qui peut être utilisé à toute les grammaires non contextuelles. Dans sa forme reconnaisseur syntaxique, l'algorithme CYK est de complexité cubique pour des grammaires en forme normale de Chomsky. Le reconnaisseur Earley, lui aussi a une complexité cubique pour toute grammaire non contextuelle, et a même, une complexité n^2 pour une grammaire non-ambigüe. Le reconnaisseur Earley est dit générale, puisque il reconnaît toute la catégorie grammaire non-contextuelle, même ceux qui sont ambigües.

Étendre un reconnaisseur pour qu'il soit un analyseur syntaxique n'est pas chose évidente, et soulève plusieurs problèmes, en plus, on peut avoir beaucoup ou infiniment de dérivation pour un mot donné, un reconnaisseur de complexité cubique peut vite devenir un analyseur de complexité infinie.

Se débarrasser des grammaires Ambigües ? Bonne idée ?

On peut se dire que des grammaires ambigües reflètent des sémantiques ambigües, et donc, ne doivent pas être utilisées en pratique. Se sera une position beaucoup trop extrême à tenir, puisque par exemple c'est très connue que l'expression 'if-else' dans la version ANSI du langage C est ambigüe, mais en attachons le 'else' au plus récent 'if' on arrive à avoir une complexité linéaire et à se débarrasser de l'ambiguïté. De plus le problème de l'ambiguïté est indécidable[6], et donc on ne peut pas reconnaître qu'une grammaire est ambigüe ou pas pour le dire à l'utilisateur.

Retourner un seul arbre de dérivation ?

Une possibilité pour les grammaires ambigües, est de retourner un seul arbre de dérivation, le premier qu'on trouve, par exemple dans les travaux de Graham[5], elle a réussi à créer un analyseur syntaxique basé sur Earley d'une complexité cubique, et qui génère la dérivation la plus à droite d'un mot (génère un seul arbre pour les grammaires ambigües). Par contre si un seul arbre est généré, ceci crée un problème pour l'utilisateur qui veut avoir tous les arbres possibles, ou bien un arbre spécifique qui n'est pas celui donné par l'algorithme. Plus encore, un utilisateur ne se rendra peut-être même pas compte que sa grammaire est ambigüe.

Sous quelle forme ?

Pour qu'un algorithme d'analyse syntaxique (Earley dans notre cas) soit générale, il faudra retourner toutes les dérivations possibles d'un mot. La question qui se pose est : sous quelle forme ? Elizabeth Scott[10] utilise ce qu'on appelle la représentation SPPF (version modifiée) utilisé pour la première fois par Tomita[11]. (Voir la section vocabulaire pour plus de détails sur cette représentation)

Vocabulaire

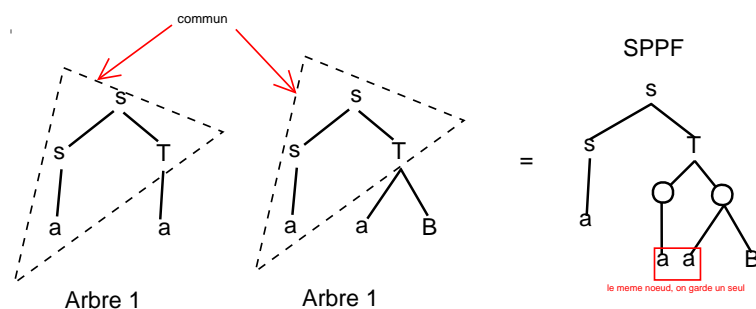
Une grammaire non contextuelle consiste en :

- Un ensemble **N** de symboles non-terminaux.
- Un ensemble **T** de symboles terminaux.
- Un élément **S** qui est le symbole de départ.
- Un ensemble **P** de règles de la forme $A ::= \alpha$, où **A** est un symbole non terminale, et α est une succession de symboles terminaux et non-terminaux possiblement vide).

Une étape de dérivation est de la forme : $\gamma A \beta \rightarrow \gamma \alpha \beta$ ou $A ::= \alpha$ est une règle de la grammaire. Une dérivation de τ à partir de σ est une succession d'étape de dérivation de la forme $\sigma \rightarrow \beta_1 \rightarrow \beta_2 \rightarrow \dots \rightarrow \tau$. On peut aussi écrire : $\sigma \xrightarrow{*} \tau$

Un arbre de dérivation est un arbre ordonné, où la racine correspond au symbole de départ S , et les feuilles sont des symboles terminaux ou bien le symbole vide ϵ . Les nœuds intermédiaires sont des symboles non-terminaux, qui ont des enfants adéquatement à la règle de la grammaire.

Une forêt partagée d'arbre de dérivation (SPPF) est une représentation permettant de réduire l'espace pour représenter tout les dérivations possible d'un mot d'une grammaire ambiguë. On trouve plusieurs variante de cette représentation, mais l'idée générale est la même, Pour un nœud du SPPF, tout ce qui se trouve en haut de ce nœud est commun à tout les arbre de dérivation, et pour les nœuds qui représente une dérivation différente du même symbole non terminale au même endroit dans le mot, seront regroupé dans le même nœud. Voici un schéma qui illustre l'idée générale autour des SPPF, il reste à définir les noms des nœuds, mais sa, on le verra plus-tard, pour l'instant gardons des noms simple :



Earley Alogirithme

Prenons cette grammaire :

```

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

```

on veux reconnaître l'entrée

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

À l'étape 0, le calcul démarre avec l'ensemble $E(0)$ et les règles de l'axiome 'S'

$E(0)$
$S = \bullet S + P (0)$
$S = \bullet P (0)$

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 soit, donc une la grammaire réursive gauche ne posera pas de problème à notre algorithme.

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

$E(0)$
$S = \bullet S + P (0)$
$S = \bullet P (0)$
$P = \bullet P * F (0)$
$P = \bullet F (0)$

Le prédiction du 3ème item de $E(0)$ ne sert à 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 à 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 à l'ensemble E(1)

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

E(0)	E(1)	E(2)	E(3)	E(4)
S = •S + P (0) S = •P (0) P = •P * F (0) P = •F (0) F = •(S) (0) F = •n (0)	F = n• (0) P = F• (0) S = P• (0) P = P• * F (0) S = S• + P (0)	S = S + •P (0) P = •P * F (2) P = •F (2) F = •(S) (2) F = •n (2)	F = (•S) (2) S = •S + P (3) S = •P (3) P = •P * F (3) P = •F (3) F = •(S) (3) F = •n (3)	F = n• (3) P = F• (3) S = P• (3) P = P• * F (3) S = S• + P (3) F = (S•) (2)
E(5)	E(6)	E(7)		
P = P * •F (3) F = •(S) (5) F = •n (5)	F = n• (5) P = P * F• (3) S = P• (3) P = P• * F (3) F = (S•) (2) S = S• + P (3)	F = (S)• (2) P = F• (2) S = S + P• (0)		

Le mot est reconnu uniquement si on a un item de la forme $(S = \alpha \bullet (0))$ dans E(7), ce qui est le cas dans notre exemple.

Essayant de construire un l'Analyseur Earley

Earley lui même a donné un brève description sur comment construire une représentation de tout les dérivations possible à partir de l'algorithme de base qui ne fais que de la reconnaissance. Et il dit aussi que sa ne requière qu'une complexité cubique en temps et en mémoire au pire des cas.

L'idée de Earley est très simple, à chaque fois qu'on fait une complétion, on ajoute un pointeur depuis chaque symbole

non terminale a gauche du point du nouveau item, vers les items qui ont engendrés cette item.

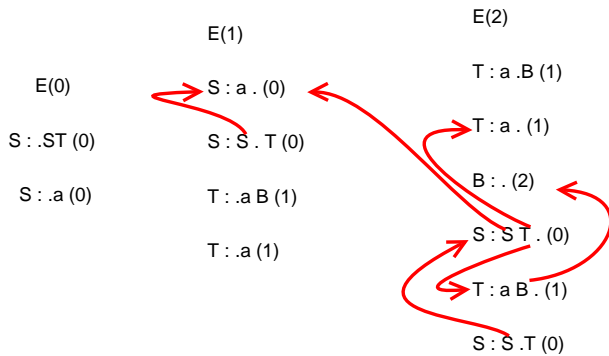
Prenons cette grammaire :

```

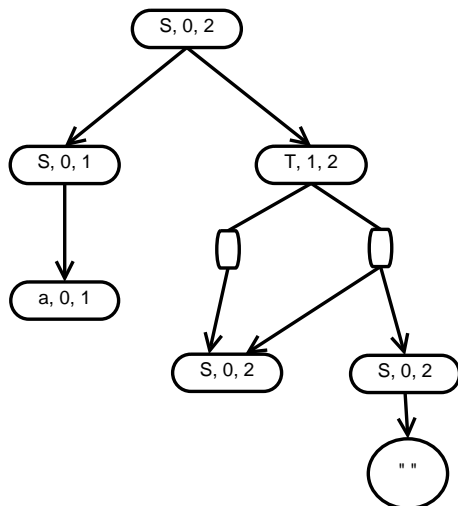
S = S T
  | a ;
B = ;
T = a B
  | a ;

```

On applique l'idée d'Earley précédente pour reconnaître le mot **aa**, on aura :



Ce qui se traduira par la représentation SPPF suivante :



On voit bien que cette représentation inclue bien toutes les dérivations possibles du mot **aa**, on va pouvoir se dire qu'on a trouvé notre algorithme d'analyse syntaxique basé sur Earley, malheureusement, dans certains cas, cette modification de l'algorithme d'Earley pour le transformer en analyseur syntaxique, n'est pas suffisante.

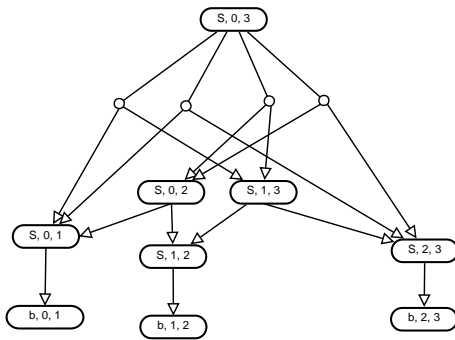
Prenons un autre exemple pour démontrer sa :

```

S = SSS
  | SS
  | b ;

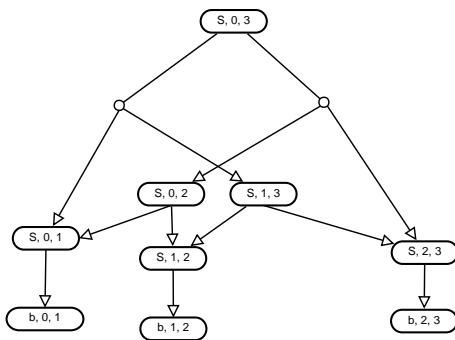
```

On appliquant la modification d'Earley sur le mot **bbb** on aura le SPPF suivant :



Cette représentation SPPF contient des dérivations superflus qui sert a reconnaître les mot **bb** et **bbbb**, alors que nous voulions juste connaître les dérivations du mot **bbb**. Dans la page 74[11], Tomita nous explique qu'ajouter plusieurs pointeurs a la même instance du symbole non terminale est une mauvaise idée et résulte sur des dérivation superflus d'autre mot.

Il existe une solution très simple a ce problème. il suffit de dupliquer la règle qui cause se problème. Dans l'exemple précédent, on aura deux règles "S = SS • (0)" identique dans E(3) mais avec des pointeurs différents, ce qui engendrera cette représentation SPPF :



cette fois si c'est la bonne, la duplication de la règle problématique a permis d'exclure les dérivations superflu.

On viens de trouver un algorithme d'analyse syntaxique basé sur Earley, mais quand est-il de la complexité? Ben, comme démontrer par Johnson[3] pour un problème similaire et comme confirmé par Scott[10], la complexité polynomiale est infinie. On peut se dire qu'on a gardé l'algorithme de base d'Earley et donc la complexité aurais du être $O(n^3)$, mais la duplication des règles engendrent un problème, en effet, la taille de la table d'Earley ne peut être borné par $O(n^p)$ pour n'importe quelle entier p. Et puisque la complexité en taille et en temps sont liées, on aura une complexité polynomiale infinie.

Introduction

Qu'est ce que l'algorithme Earley?

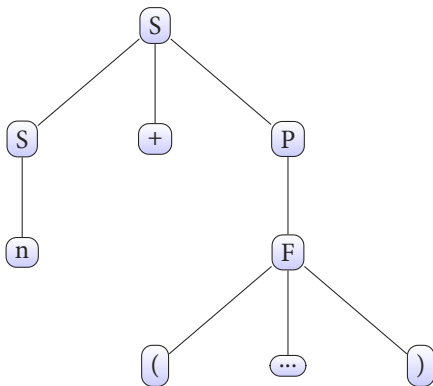
L'algorithme Earley¹, est un, parmi les nombreux algorithmes d'analyse syntaxique, est comme tout ces algorithmes, **Earley** a besoin d'une grammaire

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

... pour transformer une chaîne de mot ...

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

... en un joli arbre syntaxique (AST)



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

Pourquoi devrions-nous nous soucier?

Le plus grand avantage d'Earley, est sans doute son accessibilité. La plupart des autres algorithmes offre une restriction sur le type de grammaires, utilisent une grammaire récursive gauche et en rentrent dans une boucle infinie, utilisent un autre type et l'algorithme ne marchera plus. Bien sûr il y a des contournements qu'on peut faire, mais souvent ça complique davantage l'algorithme et rendra le travail plus complexe.

Pour dire simple Earley marche avec tout.

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

1. text

- Earley a une complexité cubique $O(3)$, dans les pires des cas, ces même cas qui ne pourront être traités par d'autre algorithme
- La plupart des grammaires simples auront une complexité linéaire
- Même la pire grammaire non-ambigue pourra être analysée en $O(3)$

Vocabulaire

Du point de vue de l'algorithme Earley, la grammaire est constituée de règles. 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 jargon d'Earley, il y a la notion d'items, Voici un item :

$A = B \cdot * C \quad (4)$

C'est juste une règle de grammaire avec des informations en plus, qui représente une reconnaissance partielle.

- Le point représente la position courante qui indique jusqu'où on a parvenu à parser
- Le chiffre 4 représente la position initiale sur l'entrée qu'on veut parser

De plus Earley introduit la notion d'ensemble d'items, chaque ensemble est caractérisé par le fait que les items qui y sont associés, ont la même position courante

Et tous ces ensembles là, sont souvent nommés table Earley

Prédiction, Lecture et Complétion

Pour construire la table Earley, on a besoin de définir trois opérations élémentaires qui s'appliquent sur un item pour produire un autre item :

- Prédiction : le symbole à droite du point est un nom terminal, on ajoute les règles de ce symbole au même ensemble
- Lecture : le symbole à droite du point est un terminal, on regarde si ce symbole coïncide avec la position courante, si oui, on ajoute cet item à l'ensemble suivant.
- Complétion : il n'y a rien à droite du point, et dans ce cas il y a reconnaissance partielle, on regarde l'item parent, et on l'ajoute à cet ensemble

Exemple de construction de table d'Earley

Reprenons cette grammaire :

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

on veut reconnaître l'entrée

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

À l'étape 0, le calcul démarre avec l'ensemble $E(0)$ et les règles de l'axiome ' S '

$E(0)$
$S = \bullet S + P \quad (0)$
$S = \bullet P \quad (0)$

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 soit, donc une grammaire récursive gauche ne posera pas de problème à 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)	E(1)	E(2)	E(3)	E(4)
S = •S + P (0) S = •P (0) P = •P * F (0) P = •F (0) F = •(S) (0) F = •n (0)	F = n• (0) P = F• (0) S = P• (0) P = P• * F (0) S = S• + P (0)	S = S + •P (0) P = •P * F (2) P = •F (2) F = •(S) (2) F = •n (2)	F = (•S) (2) S = •S + P (3) S = •P (3) P = •P * F (3) P = •F (3) F = •(S) (3) F = •n (3)	F = n• (3) P = F• (3) S = P• (3) P = P• * F (3) S = S• + P (3) F = (S•) (2)
E(5)	E(6)	E(7)		
P = P * •F (3) F = •(S) (5) F = •n (5)	F = n• (5) P = P * F• (3) S = P• (3) P = P• * F (3) F = (S•) (2) S = S• + P (3)	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 créer un programme qui va avoir en entrée une grammaire, et aura en sortie un analyseur syntaxique suivant l'algorithme Earley.

Par la suite en va modifier cet 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 et en sortie, on aura un analyseur syntaxique. Plusieurs questions se sont posées durant le développement de cet outil. Voici un récapitulatif des décisions prises :

- 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 deuxième fichier, qui contiendra la chaîne à analyser.
- La sortie sera un troisième fichier contenant l'AST
- Les noms de variables, et de fonction seront en anglais, je referai donc au dictionnaire (à faire)

Le programme sera exécuté en ligne de commande suivant la syntaxe suivante :

```
earley <file1> <file2> <file3>
```

`file1` : le fichier de grammaire `file2` : le fichier contenant la chaîne à analyser `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. Le colon et le point-virgule sont la ponctuation de Yacc.

un symbole terminal doit être déclaré comme tel au début 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] A.I.C. Johnstone E.A. Scott and G.R. Economopoulos. Brn-table based glr parsers. Technical report, Computer Science Department, Royal Holloway, University of London, 2003.
- [4] J Earley. An efficient context-free parsing algorithm. In *Communications of the ACM*, 13(2) :94–102, February 1970.
- [5] Susan L. Graham and Michael A. Harrison. *Parsing of general context-free languages*. *Advances in Computing*. 14 :77–185, 1976.
- [6] John E Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Series in Computer Science. Addison-Wesley, 1979.
- [7] Bill Joy James Gosling and Guy Steele. The java language specification. Technical report, Addison-Wesley, 1996.
- [8] Guy Steele James Gosling, Bill Joy and Gilad Bracha. The java language specification third edition. Technical report, Addison-Wesley, 2005.
- [9] Donald E Knuth. On the translation of languages from left to right. In *Information and Control* 8, (6) :607–639, 1965.
- [10] Elizabeth Scott. *SPPF-Style Parsing From Earley Recognisers*. PhD thesis, University of London, 2008.
- [11] Masaru Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, Boston, 1986.
- [12] D H Younger. Recognition of context-free languages in time n^3 . In *Inform. Control*, 10(2) :189–208, February 1967.