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 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[7] montre l'effort mis dans la sémantique pour que et 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 intéressé a d'autre algorithme comme CYK[12], Earley[4], GLR[11], 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[10], 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.

Du Reconnaisseur a l'Analyseur syntaxique

In y a pas d'analyseur ou reconnaisseur syntaxique de complexité linéaire qui peut être utilisé a toute les grammaire non contextuelle. Dans sa forme reconnaisseur syntaxique, l'algorithme CYK est de complexité cubique pour des grammaire en forme normale de Chomsky. Le reconnaisseur Earley, lui aussi a une complexité cubique pour toute grammaire non contextuelle, et a même, une complexité n2 pour une grammaire non-ambigüe. Le reconnaisseur Earley est dit générale, puisque il reconnais toute la catégorie grammaire non-contextuelle, même ceux qui sont ambiguë.

Étendre un reconnaisseur pour qu'il soit un analyseur syntaxique n'ai pas chose évidente, et soulève plusieurs problème, 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 Ambigüe ? Bonne idée ?

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

Retourner un seul arbre de dérivation ?

Une possibilité pour les grammaire ambigüe, est de retourner un seul arbre de dérivation, le premiers qu'on trouve, par exemple dans les travaux de Graham[5], elle a réussi à crée un analyseur syntaxique basé sur Earley d'une complexité cubique, et qui génère la dérivation la plus a droite d'un mot (génère un seul arbre pour les grammaire ambigüe). Par contre si un seul arbre est généré, ceci crée un problème pour l'utilisateur qui veux avoir tout les arbres possibles, ou bien un arbre spécifique qui n'ai 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 q'un algorithme d'analyse syntaxique (Earley dans notre cas) soit générale, il faudra retourner toute les dérivations possible d'un mot. La question qui se pose est : sous quelle forme ? Elizabeth Scott[10] utilise ce qu'on appelle la représentation SPPF (version modifié) utilisé pour la premier fois par Tomita[11]. (Voir la section vocabulaire pour plus de détaille sur cette représentation)

Vocabulaire

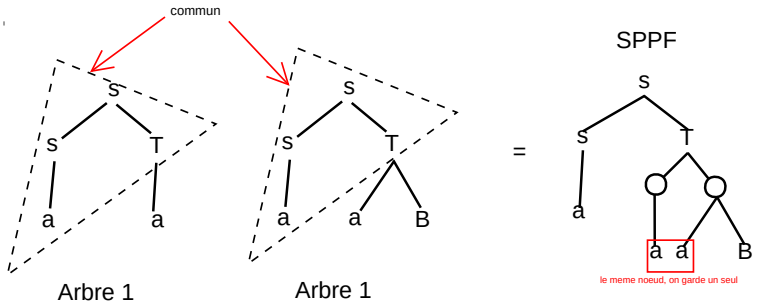
Une grammaire non contextuelle consiste en :

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

Une étape de dérivation est de la form : $\gamma A \beta \rightarrow \gamma \alpha \beta$ ou $A ::= \alpha$ est une règle de la grammaire. Une dérivation de τ a partir de σ est une succession d'étape de dérivation de la forme $\sigma \rightarrow \beta 1 \rightarrow \beta 2 \rightarrow ... \rightarrow \tau$. On peux aussi écrire : $\sigma \overset{*}{\rightarrow} \tau$

Un arbre de dérivation est un arbre ordonné, ou 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 a la règle de la grammaire.

Une foret partagé 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 ambigüe. On trouve plusieurs variante de cette représentation, mais l'idée générale est la même, Pour un noued du SPPF, tout se qui trouve en haut de ce nœud est commun a 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 a 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 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)

Le mot est reconnu uniquement si on a un item de la forme (S = α • (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 a 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.

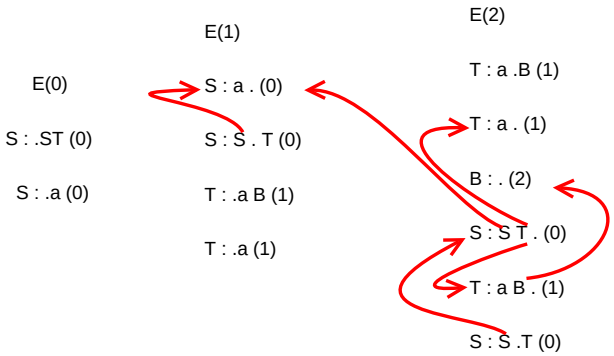
L'idée de Earley est très simple, a chaque fois qu'on fait une complétion, on ajoute un pointeur depuis chaque symbole

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

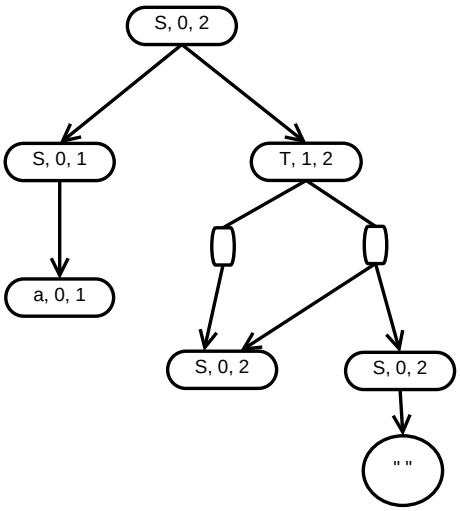
Prenons cette grammaire :

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

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



Ce qui se traduira par la représentation SPPF suivante :

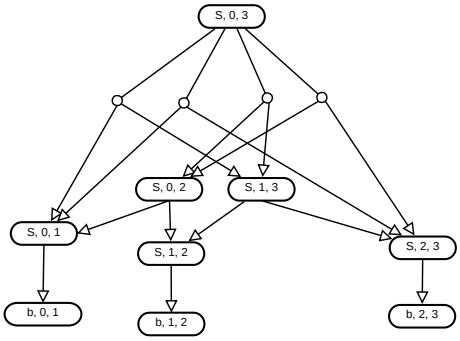


On vois bien que cette représentation inclue bien tout les dérivations possible du mot `aa`, on va pouvoir se dire qu'on a trouver notre algorithme d'analyse syntaxique basé sur Earley, malheureusement, dans certain cas, cette modification de l'algorithme d'Earley pour le transformer en analyseur syntaxique, n'ai pas suffisant.

Prenons un autre exemple pour démontrer sa :

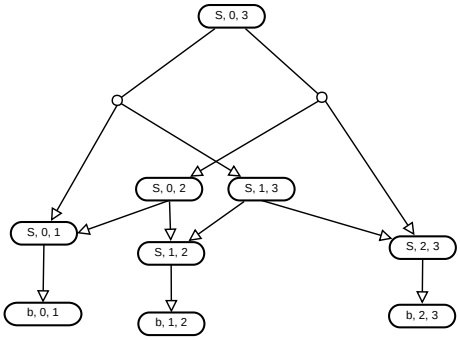
```
S = SSS
  | SS
  | b ;
```

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 reconnaitre les mot **bb** et **bbbb**, alors que nous voulions juste connaitre 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.

L'algorithme

Dans cette section, nous allons nous baser sur l'algorithme d'Earley de base et les travaux de Scott[10] ainsi que la conclusion de la précédente section, pour donner un algorithme qui sera de complexité $O(n^3)$ en temps et en mémoire. Il sera ensuite utilisé par notre programme que nous décrivons dans les sections suivantes.

Scott c'est rendu compte que il suffit d'ajouter des pointeurs entre items pour pouvoir crée un analyseur syntaxique d'Earley. les pointeur seront nommé pour pouvoir construire la structure SPPF en une seule itération.

Scott définit deux types de pointeur, un pointeur de réduction, et un pointeur de décalage qui font allusion au opération shift/reduce dans les autres algorithmes d'analyse syntaxique.

L'idée est de d'abord crée la table Earley décoré avec ces fameux pointeurs , ensuite on fait une seule itération sur cette table pour déduire la représentation SPPF (la foret) de toutes les dérivations possibles.

Création de la table d'Earley

Les opérations de base du reconnaisseur Earley, qui sont : prédiction, lecture et complétion vont être remplacer avec d'autre opérations.

- Mettre dans E_0 tout item de la forme (S ::= $\bullet\alpha$, 0) qui appartient au règles de la grammaire
- Pour chaque $i > 0$, faire l'étape d'initialisation
- Avant d'initialiser E_{i+1}, il faut faire l'étape de prédiction et l'étape de complétion a E_i.

Donc on aura les trois étapes suivantes : initialisation, prédiction et complétion. La prédiction sera exactement la même que dans l'algorithme de base d'Earley, par contre la complétion sera modifier pour permettre l'ajout de pointeur.

L'Initialisation

L'initialisation sera faite a tout les ensembles sauf E_0, voici l'algorithme qui décrit cette opération :

Soit le mot a reconnaitre $a_0 a_1 ... a_n$.

On ajoute $p = (A ::= \alpha a_i \bullet \beta, j)$ pour chaque $q = (A ::= \alpha \bullet a_i \beta, j)$ appartenant a E_{i-1}

Si $\alpha \neq \epsilon$ alors crée un pointeur de décalage nommé $i - 1$ de q vers p.

Donc l'initialisation ressemble a l'algorithme de lecture avec on plus, des pointeur pour trouver l'origine de cette lecture.

La Prédiction

La prédiction reste exactement la même que dans l'algorithme de base :

Pour chaque item $(B :== \gamma \bullet D\delta, k)$ appartenant a E_i et pour chaque règle de grammaire de la forme $D :== \rho$, on ajoute a E_i l'item $(D :== \bullet\rho, i)$

La Complétion

La complétion est la même que celle de l'algorithme de base, mais il faut ajouter deux types d'informations pour chaque complétion, a savoir, d'ou on a fait la complétion et grâce a qui on la fait. L'algorithme deviendra comme ceci :

Pour chaque item $t = (B :== \tau, k)$ appartenant a E_i et pour chaque item correspondant $q = (D ::= \alpha \bullet B \mu, h)$ appartenant a E_k, si $p = (D ::= \alpha B \bullet \mu, h)$ n'ai pas présent dans E_i alors ajouter p a E_i.

On ajoute un pointeur de réduction nommé k de p a t

Si $\tau \neq \epsilon$, On ajoute un pointeur de décalage nommé aussi k de p a q.

L'Algorithme d'analyse syntaxique

Voici l'algorithme qui permet de construire la représentation SPPF a partir de la table Earley modifié :

```
Buildtree(u, p)
{
  suppose that p ∈ E_i and that p is of the form (A ::= α•β, j)

  mark p as processed

  if p = (A ::= •, j) {
    if there is no SPPF node v labelled (A, i, i)
    create one with child node ϵ
    if u does not have a family (v) then add the family (v) to u }

  if p = (A ::= α•β, j) (where a is a terminal) {
    if there is no SPPF node v labelled (a, i−1, i) create one
    if u does not have a family (v) then add the family (v) to u }

  if p = (A ::= C•β, j) (where C is a non−terminal) {
    if there is no SPPF node v labelled (C, j, i) create one
    if u does not have a family (v) then add the family (v) to u
    for each reduction pointer from p labelled j {
      suppose that the pointer points to q
      if q is not marked as processed Buildtree(v, q) } }


  if p = (A ::= α'a•β, j) (where a is a terminal, α' ≠ ϵ) {
    if there is no SPPF node v labelled (a, i−1, i) create one
    if there is no SPPF node w labelled (A ::= α'•aβ, j, i−1) create one
    for each target p' of a predecessor pointer labelled i−1 from p {
      if p' is not marked as processed Buildtree(w, p') }
    if u does not have a family (w, v) add the family (w, v) to u }

  if p = (A ::= α'C•β, j) (where C is a non−terminal, α' ≠ ϵ) {
    for each reduction pointer from p {
      suppose that the pointer is labelled l and points to q
      if there is no SPPF node v labelled (C, l, i) create one
      if q is not marked as processed Buildtree(v, q)
      if there is no SPPF node w labelled (A ::= α'x•Cβ, j, l) create one
      for each target p' of a predecessor pointer labelled l from p {
        if p' is not marked as processed Buildtree(w, p') }
      if u does not have a family (w, v) add the family (w, v) to u }}
}
```

Et Pour construire la représentation entière depuis la racine, on exécute la fonction suivante :

```
PARSER {
  create an SPPF node u0 labelled (S, 0, n)
  for each decorated item p = (S ::= α•, 0) ∈ E_n  Buildtree(u0, p)
}
```

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 n3. In *Inform. Control, 10(2) :189–208*, February 1967.