

Algorithme Earley 2016/2017

Rapport

*Abderrazak ZIDANE*¹

Encadrant : Régis-Gianas

1. zidane.rezzak@gmail.com

Table des matières

1	Introduction	2
2	Du Reconnaisseur a l'Analyseur syntaxique	3
2.1	Se débarrasser des grammaires Ambigüe? Bonne idée?	3
2.2	Retourner un seul arbre de dérivation?	3
2.3	Sous quelle forme?	3
2.4	Vocabulaire	3
2.5	Earley Alogirithme	4
2.6	Essayant de construire un l'Analyseur Earley	5
3	L'algorithme	8
3.1	Création de la table d'Earley	8
3.2	L'Initialisation	8
3.3	La Prédiction	8
3.4	La Complétion	9
3.5	L'Algorithme de construction de la représentation SPPF	9
3.6	Problème de la règle nulle	10
3.7	Détecter qu'un symbole est annulable	10
4	Le programme	11
4.1	Utiliser le programme	11
4.2	Diagramme De Classe	12
4.3	La Classe Grammar	13
4.4	La Classe Rule	13
4.5	La Classe EarleyTable	14
4.6	La Classe EarleySet	14
4.7	La Classe EarleyItem	15
5	ANNEXE A : Quelque images produite par le programme	16

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[11] 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[8] montre l'effort mis dans la sémantique pour que la grammaire soit LALR(1), par contre dans la 3ème édition de cette spécification[9], 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[14], Earley[4], GLR[13], 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[12], 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 le raisonnement utilisé par Elizabeth Scott pour construire ces algorithmes, trouver les erreurs de ces algorithmes, et les corriger. On proposera ensuite une application écrite en C++ qui implémente ces algorithmes.

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 reconnait 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.

Dans cette section, nous allons tout d'abord reprendre la question courante qu'un lecteur peut se poser, ces repenses sont tirées des articles de Scott[12].

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 utilisés 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'ambigüité. De plus le problème de l'ambigüité est indécidable[7], 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[6], 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[12] utilise ce qu'on appelle la représentation SPPF (version modifiée) utilisée pour la première fois par Tomita[13]. (Voir la section vocabulaire pour plus de détails sur cette représentation)

Vocabulaire

Une grammaire non contextuelle consiste en :

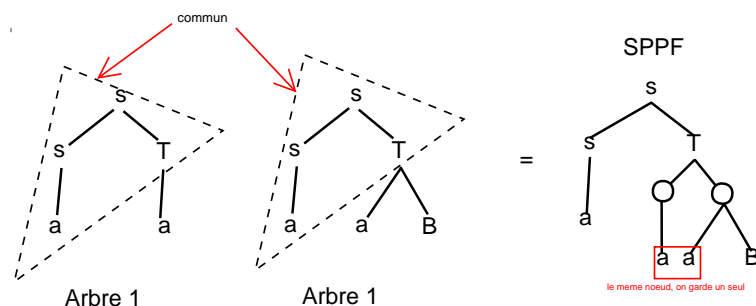
- Un ensemble **N** de symboles non-terminaux.
- Un ensemble **T** de symboles terminaux.

- Un élément S qui est le symbole de départ.
- Un ensemble P de règle de la forme $A ::= \alpha$, 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 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 dérivation 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 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)
$S = \bullet S + P (0)$
$S = \bullet P (0)$
$P = \bullet P * F (0)$
$P = \bullet F (0)$
$F = \bullet (S) (0)$
$F = \bullet n (0)$

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)
$F = n \bullet (0)$

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)
$F = n \bullet (0)$
$P = F \bullet (0)$

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

E(1)
$F = n \bullet (0)$
$P = F \bullet (0)$
$S = P \bullet (0)$
$P = P \bullet * F (0)$

...

Au finale notre table Earley ressemblera a :

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

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 dérivation 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.

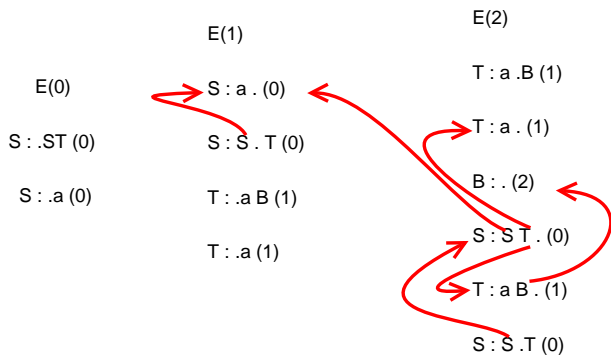
Prenons cette grammaire :

```

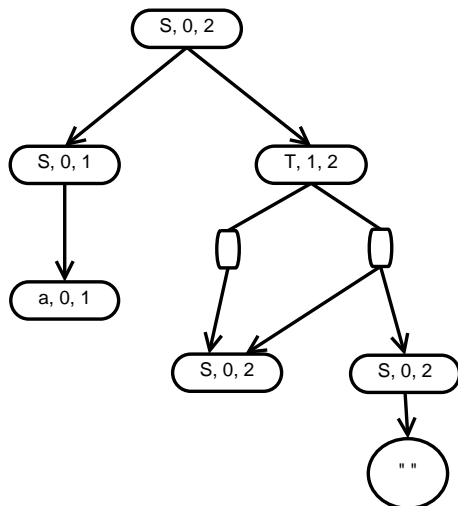
S = S T
  | a ;
B = ;
T = a B
  | a ;

```

On applique l'idée d'Earley précédente pour 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.

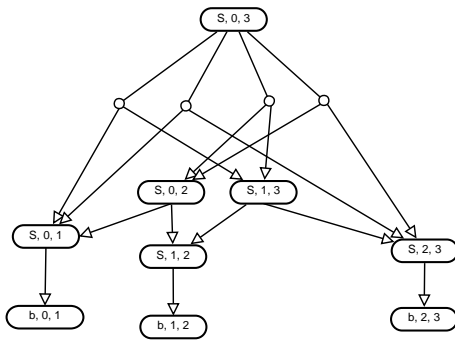
Prenons un autre exemple pour démontrer ça :

```

S = SSS
  | SS
  | b ;

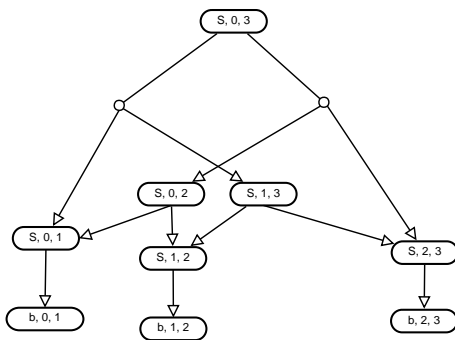
```

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



Cette représentation SPPF contient des dérivation superflus qui sert a reconnaître les mot **bb** et **bbbb**, alors que nous voulions juste connaître les dérivation du mot **bbb**. Dans la page 74[13], 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 dérivation 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[12], 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[12] 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 créer 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 à l'opération shift/reduce dans les autres algorithmes d'analyse syntaxique.

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

Création de la table d'Earley

Les opérations de base du reconnaiseur Earley, qui sont : prédiction, lecture et complétion vont être remplacées avec d'autres opérations.

- Mettre dans E_0 tout item de la forme $(S ::= \bullet\alpha, 0)$ qui appartient aux 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 à 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 modifiée pour permettre l'ajout de pointeur.

L'Initialisation

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

Soit le mot à reconnaître $a_0a_1\dots a_n$.

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

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

Donc l'initialisation ressemble à l'algorithme de lecture avec en plus, des pointeurs 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 à E_i et pour chaque règle de grammaire de la forme $D ::= \rho$, on ajoute à 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'où 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 de construction de la représentation SPPF

Scott[12] avait proposer un algorithme récursive pour passé de la table d'Earley décoré avec des pointeurs a la représentation SPPF, mais après un test, il s'avère qu'il comportait une erreur, qu'on a corrigé Voici la version corrigé :

```
Buildtree(u, p)
{
  suppose that  $p \in E_i$  and that p is of the form  $(A ::= \alpha \bullet \beta, j)$ 

  mark p as processed

  if  $p = (A ::= \bullet, j)$  {
    if u does not have a child node  $\epsilon$  then add a child node  $\epsilon$  to u }

  if  $p = (A ::= \alpha \bullet \beta, j)$  (where  $\alpha$  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  $p = (A ::= C \bullet \beta, j)$  (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  $p = (A ::= \alpha' a \bullet \beta, j)$  (where a is a terminal,  $\alpha' \neq \epsilon$ ) {
    if there is no SPPF node v labelled  $(a, i-1, i)$  create one
    if there is no SPPF node w labelled  $(A ::= \alpha' \bullet a \beta, j, i-1)$  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  $p = (A ::= \alpha' C \bullet \beta, j)$  (where C is a non-terminal,  $\alpha' \neq \epsilon$ ) {
    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  $(A ::= \alpha' \bullet C \beta, j, l)$  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  $p = (S ::= \alpha \bullet, 0) \in E_n$  Buildtree(u0, p)
}
```

Problème de la règle nulle

l'algorithme précédent est parfait, néanmoins, on remarque qu'il y a un problème pour construire la table Earley (la version décoré avec des pointeurs) quand il y a des symboles annulable dans la grammaire.

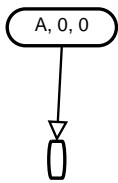
Prenons un exemple de grammaire pour illustrer le problème :

```
A ::= ε
A ::= B
B ::= A
```

Si nous construisons notre table Earley pour reconnaître le mot vide, on aura le résultat suivant :

```
E(0)
A ::= • (0)
A ::= •B (0)
B ::= •A (0)
```

et nous aurons le graphe SPPF suivant :



il manque deux item dans notre ensemble E_0 , a savoir $A ::= B \bullet (0)$ et $B ::= A \bullet (0)$.

Le problème vient d'un mauvais timing pour faire l'opération de complétion. En effet, au moment de compléter $A ::= \bullet (0)$, l'item $B ::= \bullet A (0)$ n'ai pas encore présent dans E_0 .

Pour résoudre ce problème, il ya la méthode simple qui consiste a faire plusieurs itérations de complétion et de prédiction sur chaque ensemble E_i jusqu'à aucun item n'ai ajouté. Plusieurs itération signifie complexité dans le pire cas va s'envoler, et donc on utilisera pas cette méthode.

La solution a ce problème est donné par Ayrcock et Horspool[5]. Ils ont remarqué qu'il pouvait faire avancer certain items sans attendre de faire une complétion. Plus précisément, au moment de la prédiction de $q = (B ::= \alpha \bullet C \tau, k) \in E_i$, si C est annulable, alors on ajoute aussi $q = (B ::= \alpha C \bullet \tau, k)$ a E_i . On appellera cette opération, la prédiction magique.

Cette solution n'ai pas complète encore, puisque il faut aussi mettre les différents pointeurs de réduction a ces nouveaux items issu de la prédiction magique. L'idée est de marqué l'item suite a une redirection magique. Ensuite a la fin du des opérations de prédictions et de complétions sur E_i , on parcours tout les items marqué (suite une prédiction magique) et on complète les pointeur de réductions qui manque.

Détecter qu'un symbole est annulable

Le réel problème maintenant est de trouver (ou crée) un algorithme pour détecter les symboles annulables et que sa complexité, doit être inférieure a $O(n_3)$, qui est la complexité de l'analyseur syntaxique Earley.

Il s'avère que la solution est simple, On va lire chaque règle de la grammaire, et maintenir une liste de symbole annulable qu'on a trouvé. Si une règle est annulable ou bien contient uniquement des symbole annulable (dans sa parti droite), alors on ajoute le symbole du gauche de cette règle a cette liste, et on recommence le processus, jusqu'à aucun symbole n'ai rajouté.

Cette algorithme est largement suffisant pour notre analyseur Earley et a une complexité quadratique $O(n_2)$ par rapport a la taille de la grammaire.

Il s'avère qu'il existe un meilleur algorithme pour ce problème, avec une complexité linéaire $O(n)$. Il a été documenté dans les pages 293-295 du livre sur Marpa[10]. Mais on va pas l'utiliser dans notre programme puisque cette algorithme est assez complexe a implémenter et va grandement diminuer la lisibilité du code.

Le programme

Dans cette section, je vais mettre tout ce qui a un rapport avec le programme que j'ai développé. A savoir, comment l'utiliser, le diagramme de classe et les différentes dépendances de notre programme.

Utiliser le programme

Le projet se trouve sur Github¹. C'est le dossier "code" qui contient le code source du programme.

Pour compiler le programme, il faut d'abord installer les dépendances :

```
$ sudo apt-get update
$ sudo apt-get install graphviz
```

Ensuite :

```
$ git clone https://github.com/rezid/earley.git
$ cd earley
$ cd code
$ make
```

Cela compilera le programme et créera l'exécutable `bin/app.out`.

Pour exécuter le programme, on a besoin de deux fichiers. Le premier qui contient la définition de la grammaire, il faut faire attention à bien séparer les symboles par des espaces :

```
%token the a this he she book boys girl with in takes take ;

S : NP VP | Aux NP VP | VP ;
NP : PRON | Det Nom ;
Nom : N | Nom N | Nom PP ;
PP : PRP NP ;
VP : V | V NP | VP PP ;
Det : the | a | this ;
PRON : he | she ;
N : book | boys | girl ;
PRP : with | in ;
V : takes | take ;
```

et le deuxième fichier contient le mot à analyser séparé par des espaces :

```
take this book
```

il suffit d'exécuter la commande suivante :

```
$ ./bin/app.out grammar.txt string.txt
```

Deux cas peuvent se produire. Soit le mot n'a pas reconnu est dans ce cas le message suivant s'affiche sur la console :

```
PARSE FAIL : word does not belong to the grammar
```

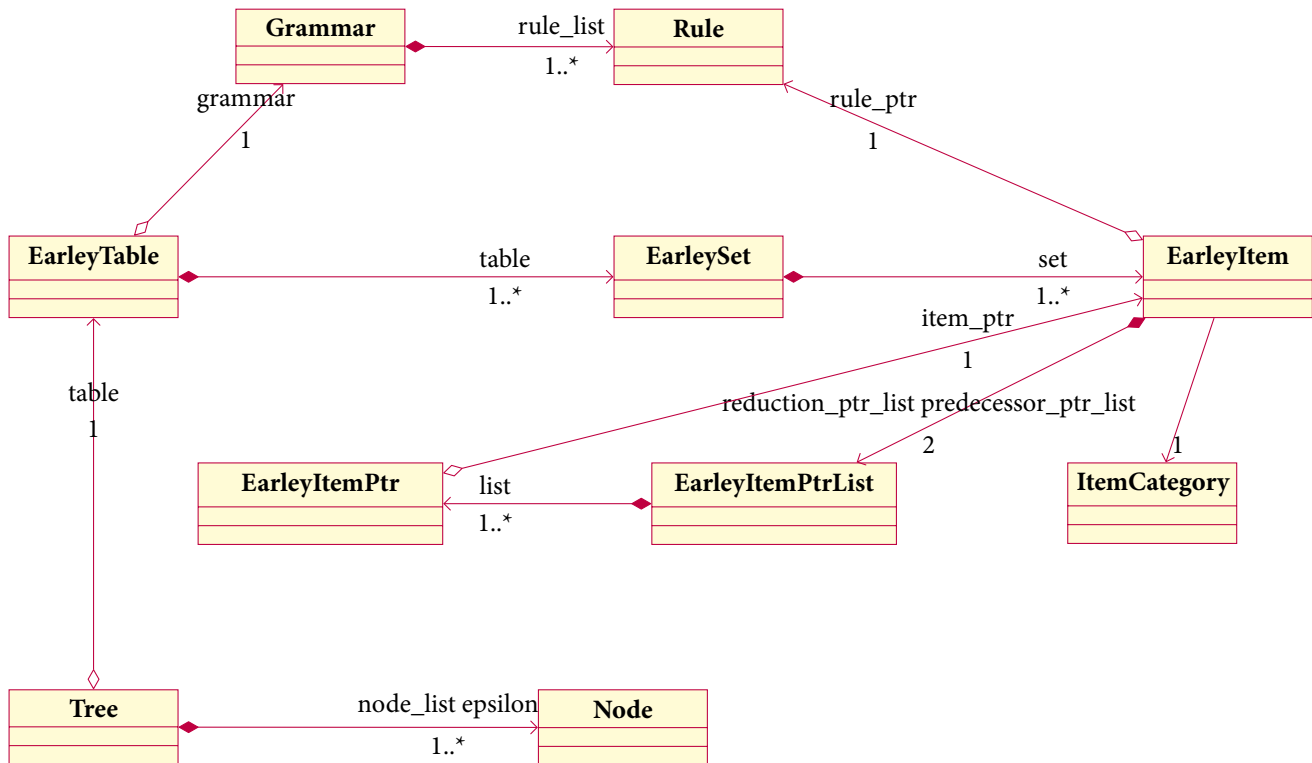
Soit le mot est reconnu par la grammaire et dans ce cas là, le PDF contenant le graphe SPPF du mot s'affiche.

Voici un diagramme de classe comportant uniquement les noms des classes utilisés et leurs relations.

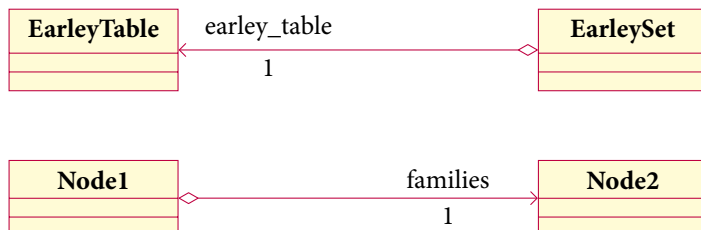
1. <https://github.com/rezid/earley>

Diagramme De Classe

Voici un diagramme de classe comportant uniquement les noms des classes utilisés et leurs relations, le lecteur doit connaître la différence entre une agrégation et une composition pour lire correctement le diagramme.



Pour pas encombrer le diagramme voici la suite :



remarque que Node1 et Node2 réfèrent a la même classe : Node

La Classe Grammar

Grammar
start_symbole : string terminal_symbole_list : list_of_string rule_list : list_of_rule nullable_symbole_list : list_of_string
Grammar () set_start_symbole (string) set_terminal_symbole_list (list_of_string) add_rule_if_not_present (rule) add_terminal_symbole_if_not_present (string) add_nullable_symbole_if_not_present (string) get_rule_list () get_start_symbole () print_terminal_symbole_list () print_rule_list () print_nullable_symbole_list () is_terminal_symbole (string) is_not_terminal_symbole (string) is_nullable_symbole (string) compute_nullable_symbole_list() create_earley_table_from_input(reference of string_list)

Cette classe représente une grammaire, elle comporte toute les constituantes de cette derniers, a savoir :

- Les symbole terminaux
- Les règles
- Le symbole de départ

On a ajouté dans cette classe la liste des symboles annulables.

La Classe Rule

Rule
main_symbole : string body : list_of_string main_symbole_is_set : boolean
Rule () Rule (string, list_of_string) print_rule (string) print_Earley_rule (integer, integer) get_main_symbole () get_symbole (integer) set_main_symbole (string) set_body (list_of_string) get_body () push_back_symbole_to_body (string) clear_rule () clear_body () operator== (reference to rule) size ()

Cette classe représente une règle de production, elle comporte toute les constituantes de cette derniers, a savoir :

- Le symbole de départ (la partie gauche de la règle).
- Le body de la règle (la partie droite de la règle).

La Classe EarleyTable

EarleyTable
table : list_of_EarleySet input_symbole_list : list_of_string grammar : reference to Grammar
EarleyTable (Grammar, reference to list_of_string) add_item_to_set_if_not_present (int, EarleyItem) size () get_input_symbol (integer) get_grammar () get_set (integer) compute_earley_table () front () back () print_table () generate_sppf_structure () get_category (reference to EarleyItem)

Cette classe représente la table d'Earley, elle est constituée de plusieurs ensemble d'Earley (EarleySet), on trouve aussi dans cette classe le mot à reconnaître, puisque chaque table d'Earley correspond à un mot donné. On garde aussi une référence vers la grammaire utilisée pour calculer la table.

La Classe EarleySet

EarleySet
set : list_of_EarleyItem index : integer earley_table : reference to EarleyTable precedent_set : reference to EarleySet
EarleySet (reference to EarleySet, reference to EarleyTable) magical_prediction (reference to EarleyItem) prediction (EarleyItem) completion (reference to EarleyItem) set_precedent_set (reference to EarleySet) get_set () add_item_if_not_present (reference to EarleyItem) front () size () print_set () initialize () complete () resolve_magical_prediction_reduction_ptr () get_index ()

Cette classe représente un ensemble d'Earley, elle est constituée de plusieurs item (EarleyItem), chaque ensemble garde en mémoire une référence vers le précédent ensemble formant ainsi une liste chaînée. Chaque ensemble possède aussi un index (Pour E_0 l'index c'est 0). On garde aussi une référence vers la table d'Earley où appartient l'ensemble, histoire de pouvoir remonter vers la table si besoin.

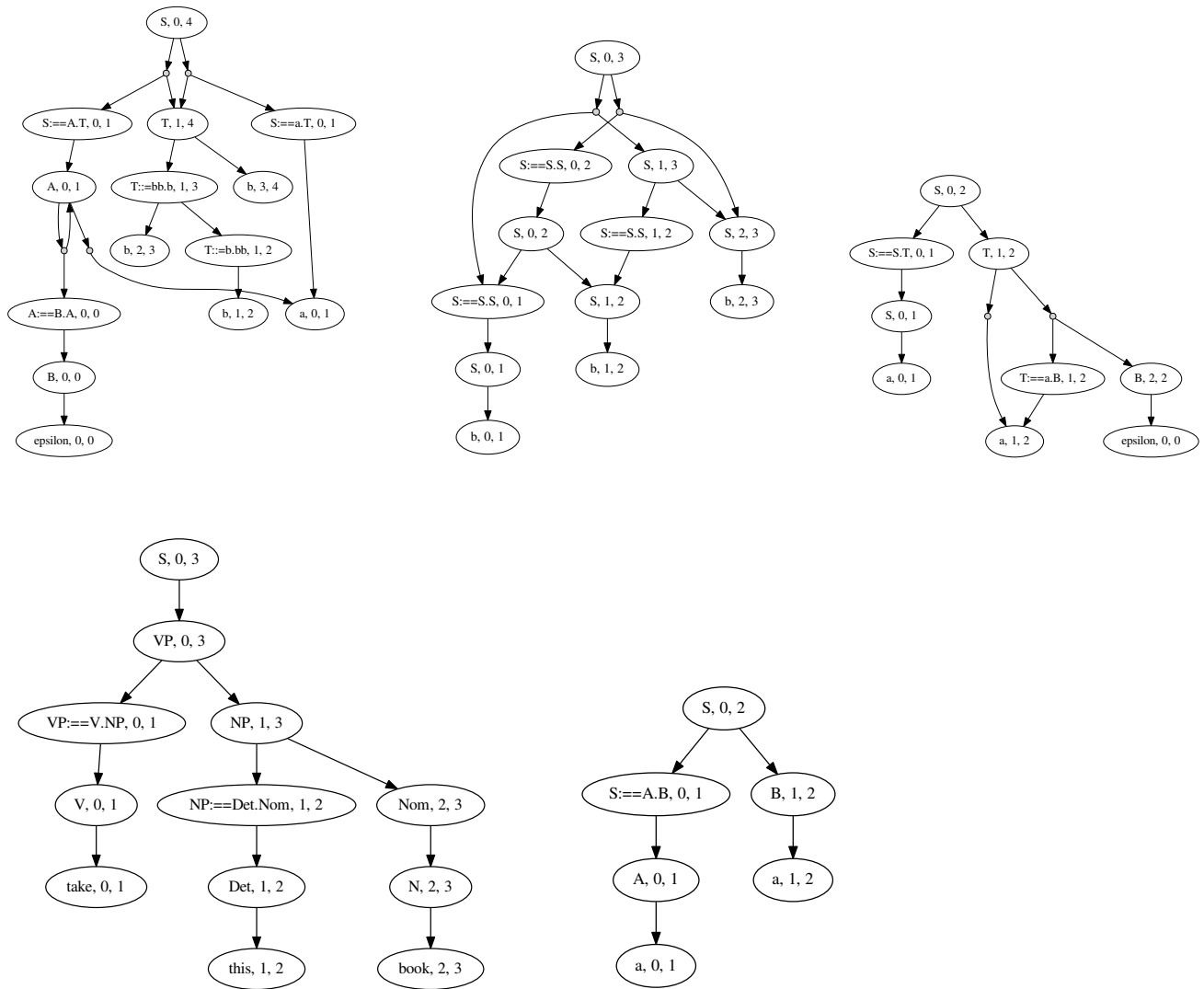
La Classe EarleyItem

EarleyItem
<p>rule_ptr : reference to Rule position : integer item_start : integer reduction_ptr_list : EarleyItemPtrList predecessor_ptr_list : EarleyItemPtrList symbole_before_position_is_null : boolean from_magical_reduction : boolean processed : boolean</p>
<p>EarleyItem (reference to Rule, integer, integer, boolean) next_symbole () precedent_symbole () print_item () print_item_without_pointer () operator== (reference to EarleyItem) next_item (boolean) get_item_start () get_position () get_rule () add_reduction_ptr (EarleyItemPtr) add_predecessor_ptr (EarleyItemPtr) get_reduction_ptr_list () get_predecessor_ptr_list () is_symbole_before_position_is_null () is_symbole_before_position_is_not_null () is_complete () is_from_magical_reduction () set_from_magical_reduction (boolean) set_processed (boolean) is_processed () is_not_processed () mark_as_processed () is_symbole_before_two_position_is_null () get_alpha_prim () get_beta ()</p>

Cette classe représente un item, elle comporte toute les constituantes de cette derniers, a savoir :

- La règle de production utilisé
- La position du point
- le numéro entre parenthèse
- les pointeurs de réduction
- les pointeur de décalage

ANNEXE A : Quelques images produite par le programme



Bibliographie

- [1] Frank L DeRemer and Thomas J. Pennello. Efficient computation of lalr(1) look-ahead sets. In *ACM Trans. Program. Lang. Syst.*, 4(4) :615–649, October 1982, 1982.
- [2] Franklin L DeRemer. *Practical translators for LR(k) languages*. PhD thesis, Massachusetts Institute of Technology, 1969.
- [3] et G.R. Economopoulos. E.A. Scott, A.I.C. Johnstone. 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] JOHN AYCOCK et R. NIGEL HORSPPOOL. Practical earley parsing. In *Department of Computer Science, University of Calgary*, 2002.
- [6] Susan L. Graham and Michael A. Harrison. *Parsing of general context-free languages*. *Advances in Computing*. 14 :77–185, 1976.
- [7] John E Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Series in Computer Science. Addison-Wesley, 1979.
- [8] Bill Joy James Gosling and Guy Steele. The java language specification. Technical report, Addison-Wesley, 1996.
- [9] Guy Steele James Gosling, Bill Joy and Gilad Bracha. The java language specification third edition. Technical report, Addison-Wesley, 2005.
- [10] Jefrey Kegler. Marpa : the program. In *part of Libmarpa. Libmarpa is free software*, 2014.
- [11] Donald E Knuth. On the translation of languages from left to right. In *Information and Control* 8, (6) :607–639, 1965.
- [12] Elizabeth Scott. *SPPF-Style Parsing From Earley Recognisers*. PhD thesis, University of London, 2008.
- [13] Masaru Tomita. *Efficient parsing for natural language*. Kluwer Academic Publishers, Boston, 1986.
- [14] D H Younger. Recognition of context-free languages in time n^3 . In *Inform. Control*, 10(2) :189–208, February 1967.