2.2.1 Grammaires LL(k)

Le non-déterminisme dans les analyses descendantes vient du fait que, étant donné un non-terminal A à dériver, on a le choix entre toutes les parties droites des règles $A \to \beta$. On a vu que si β commence par un terminal, une décision sur la productivité de la dérivation peut être prise immédiatement (en comparant ce terminal à la chaîne à reconnaître). Mais si β commence par un non-terminal, il n'est pas facile de trouver comment prendre une bonne décision.

L'idée des grammaires LL(k) est de construire, à partir de la grammaire initiale (sans la modifier), une **table de prédiction** : étant donné le non-terminal à dériver, et le symbole courant du mot à reconnaître, cette table donne les parties droites susceptibles de donner finalement le symbole à reconnaître.

S'il n'y a qu'une dérivation possible dans chaque cas (au plus), la grammaire est dite LL(1). Si on peut construire une table où il n'y a qu'une dérivation possible en regardant 2 caractères, la grammaire est dite LL(2).

Pourquoi ce nom LL(k) ? Parce que ces grammaires permettent directement de mettre en œuvre des analyseurs construisant de manière déterministe de gauche à droite (*left-to-right*) des dérivations gauches (*left*) avec un regard avant (*look-ahead*) de k symboles. (SLL(1) désigne les grammaires *simples* de cette famille.)

2.2.1.1 Tables de prédiction

Exemple LL(1) Comme premier exemple de table de prédiction (ou d'analyse prédictive), considérons le cas de la grammaire suivante, qui est (presque) sous forme de Greibach $S \to aSb$; $S \to cC$; $C \to dC$; $C \to c$. La table représente, pour chaque non-terminal à dériver, pour chaque lettre du mot à apparier, la (ou les) règle(s) de dérivation à appliquer.

	a	b	c	d
S	aSb		cC	
C			c	dC

Exemple LL(1) (non Greibach) Cette grammaire n'est pas sous forme de Greibach, ce qui ne l'empêche pas d'être LL(1) : $S \to aSb$; $S \to CC$; $S \to b$; $C \to dC$; $C \to c$.

Langage engendré : $a^n(d^*cd^*c|b)b^n$.

La table, qu'on peut facilement construire à la main en regardant la grammaire (même si pour être sûr de ne pas faire d'erreur, il vaut mieux vérifier avec l'algorithme qu'on va voir ensuite), montre bien que la grammaire est LL(1).

	a	b	c	d
S	aSb	b	CC	CC
C			c	dC

Dans ces deux exemples, il y a au plus une dérivation par case (les cases vides correspondent à des cas d'erreur), il est donc possible de décider quelle dérivation appliquer en considérant 1 caractère de la chaîne à produire, ces grammaires sont LL(1).

Soit le mot *aacddcbb*, on peut vérifier facilement que l'analyse avec la première grammaire sur la base de la table donne une décision unique à chaque étape : l'arbre d'exploration n'a

qu'une branche, et l'analyse est linéaire : voir figure 2.12.

FIGURE 2.12 – Arbre d'exploration pour le mot *aacddcbb* et la grammaire $S \to aSb$; $S \to cC$; $C \to dC$; $C \to c$

$(S\,, aacddcbb)$
$(aSb\,, aacddcbb)$
$(Sb\,, acddcbb)$
$(aSbb\,, acddcbb)$
$(Sbb\,, cddcbb)$
$(cCbb\,, cddcbb)$
$(Cbb\,, ddcbb)$
$(dCbb\,, ddcbb)$
$(Cbb\,, dcbb)$
$(dCbb\,, dcbb)$
$(Cbb\,, cbb)$
$(cbb\,, cbb)$
$(\varepsilon\,, \varepsilon)$

Analogie avec l'automate : si on se souvient que la correspondance entre automates et grammaires passe par la correspondance entre états et non terminaux, on voit que la table construite ici ressemble beaucoup à la table de transition d'un automate... Ce n'est bien sûr pas fortuit...

Exemple LL(2) Avec la grammaire suivante, il y a une case qui contient deux règles. $S \to aSb$; $S \to ab$; $S \to cC$; $C \to dC$; $C \to c$.

	a	b	c	d
S	aSb ab		cC	
C			c	dC

On peut s'intéresser à la table de prédiction considérant **2** caractères en avant. Dans ce cas, il faut aussi prendre en considération le fait que l'on peut se trouver dans une situation où il ne reste plus qu'un seul caractère à produire : ces cas sont représentés par la notation \$ qui correspond à la fin du mot.

	aa	ab	ac	cc	cd	$c\$$	dc	dd	cb
S	aSb	ab	aSb	cC	cC				
C						c	dC	dC	c

Il y a au plus une dérivation par case : on dira que la grammaire est LL(2).

2.2.1.2 Construction d'une table LL(1)

Dans les deux exemples précédents, la table était très facile à construire, grâce au fait que les parties droites de règles commençaient (presque) toujours par un terminal. Cependant, il est possible de construire la table de prédiction pour n'importe quelle grammaire (quitte à ce qu'il y ait plusieurs règles dans certaines cases). Comment procéder ?

Intuitivement, il faut mettre en œuvre la récursivité de la grammaire : il y a bien des règles qui produisent des terminaux à gauche de leur partie droite (la grammaire est non

récursive gauche par hypothèse) — soit par exemple $C \to a\alpha$; alors je peux considérer comme pertinente pour produire un a les règles qui sont de la forme $D \to C\beta$.

Cette observation nous met sur la voie d'une méthode récursive de construction de la table LL, qui passe par la construction de deux ensembles associés à chaque symbole : PREMIER() et SUIVANT() (*first* et *follow*).

PREMIER() est défini pour tout symbole de la grammaire, terminal ou non terminal, et par extension, il peut être défini pour tout mot sur $(X \cup V)^*$, et donc en particulier pour toute partie droite de règle.

Pour $\alpha \in (X \cup V)^*$, PREMIER$(\alpha) = \{a \in X \ / \ \alpha \xrightarrow{*} au\}$

Intuitivement, cette fonction associe à tout proto-mot son « coin gauche » : le premier terminal du proto-mot dérivé. Si le proto-mot commence par un terminal, c'est trivial, sinon, il faut regarder comment le non terminal finit par se réécrire (il peut y avoir plusieurs étapes, mais si la grammaire est non récursive gauche, ça doit se terminer). Une fois identifié, ce coin gauche ne peut pas changer (avec une grammaire algébrique : les terminaux ne sont jamais effacés ou déplacés).

SUIVANT() est défini seulement pour les non terminaux de la grammaire : il s'agit du premier symbole qui peut suivre le mot produit par le non terminal.

Pour $A \in V$, SUIVANT$(A) = \{a \in X \ / \ S \xrightarrow{*} \alpha A a \beta\}$

Par convention, on introduit la notation \$ pour représenter le fait que la fin du mot peut suivre un non terminal donné.

L'algorithme de construction d'une table LL(1) commence par le calcul de ces deux ensembles, qui en pratique doivent être calculés pour chaque non terminal (dans les autres cas, PREMIER() est trivial, et SUIVANT() n'a pas besoin d'être calculé).

Le calcul des PREMIER() se fait avec un algorithme de point fixe. Il faut réitérer la manœuvre suivante (donc pour chaque non terminal) jusqu'à ce qu'aucun changement ne soit enregistré.

```
Pour chaque non terminal A :
    Pour chaque règle A → α :
        Si α = ε, ajouter {ε} à PREM(A)
        Sinon (alors α = A_1 A_2 ... A_k) :
            i = 0
            répeter :
                i = i + 1
                ajouter PREM(A_i) \ {ε} à PREM(A)
            tant que i ≤ k et ε ∈ PREM(A_i)
            si i = k et ε ∈ PREM(A_i) :
                ajouter {ε} à PREM(A)
```

N.B. ce calcul nous donne la valeur de PREM() pour n'importe quel $A \in V$, il peut être facilement généralisé, avec les mêmes précautions concernant la présence d'ε dans les PREM(), et en supposant que si $x \in X$, PREM$(x) = \{x\}$, au calcul de PREM(β) pour tout $\beta \in (X \cup V)^*$.

Le calcul de SUIV() utilise le résultat de PREM(). L'algorithme consiste aussi à réitérér le calcul suivant (pour chaque règle) jusqu'au point fixe :

Mettre \$ dans SUIV(S)
Pour chaque règle $A \to A_1 A_2 ... A_k$:
    Pour chaque A_i ($i \in [1, k[$) :
        ajouter PREM(A_{i+1}) $\setminus \{\varepsilon\}$ à SUIV(A_i)
    Ajouter SUIV(A) à SUIV(A_k)
    j = k
    tant que $\varepsilon \in$ PREM(A_j) (et $j \geq 0$) :
        ajouter SUIV(A) à SUIV(A_{j-1})
        $j = j - 1$

Table LL(1) Une fois les deux ensembles (fonctions) PREM() et SUIV() construites, on peut construire la table LL(1) :

1. Pour la règle n^o i de la forme $A \to \alpha$:
   (a) Pour tout $a \in$ PREMIER(α), ajouter i à la case (A, a).
   (b) Si $\varepsilon \in$ PREMIER(α), ajouter i à la case (A, b) pour chaque $b \in$ SUIVANT(A).
   Si $\varepsilon \in$ PREMIER(α) et \$ $\in$ SUIVANT(A), ajouter i à la case $(A, \$)$.
2. Marquer erreur dans toutes les cases restées vides.

Voir aussi la figure 2.13 pour une autre formulation des mêmes algorithmes.

2.2.1.3 Conclusion

Analyse LL(1) Il devient très facile, ainsi équipé d'une table de prédiction, de formuler un algorithme de parsing descendant : cet algorithme est d'une complexité linéaire, puisque chaque symbole terminal mène à une décision unique et non remise en cause.

L'implémentation d'un tel algorithme est laissée en exercice (il faut évidemment aussi envisager une implémentation de la construction de la table...)

LL1-isation Les grammaires LL(1) sont particulièrement coopératives, mais malheureusement on sait que toutes les grammaires algébriques ne sont pas équivalentes à une grammaire LL(1). Il est cependant intéressant d'évoquer la ou les méthodes que l'on peut utiliser pour se rapprocher d'une grammaire LL(1).

La première méthode consiste à supprimer les récursions gauches, ce qui est fait implicitement dans le processus de mise sous forme normale de Greibach (on a vu l'algo indépendamment) ; une seconde transformation utile consiste à faire une factorisation gauche de la grammaire.

Grammaires LL(k) Mais si ça facilite la construction de la table et peut dans certains cas augmenter le déterminisme, il reste souvent des cases contenant plusieurs règles, et donc des facteurs d'indétermination. Dans ce cas, il reste la possibilité de construire des tables de prédiction en augmentant le regard en avant (cf exemple du début de la section).

Il faut noter que cette "généralisation", d'une part augmente la complexité du pré-traitement de la grammaire, et d'autre part (surtout), ne permet pas de traiter toutes les grammaires. C'est évident pour les grammaires ambigües, mais c'est aussi le cas pour

FIGURE 2.13 – Autres versions des algos prem/suiv/ll

Algorithme de construction de la table LL

Pour $\alpha \in (X \cup V)^*$, PREMIER$(\alpha) = \{a \in X \ / \ \alpha \xrightarrow{*} au\}$

Pour $A \in V$, SUIVANT$(A) = \{a \in X \ / \ S \xrightarrow{*} \alpha A a \beta\}$

Calcul de PREMIER(A) Réitérer jusqu'au point fixe :

1. Si $A \in X$, PREMIER$(A) = \{A\}$.
2. Si $A \to \varepsilon \in P$, ajouter ε à PREMIER(A).
3. Pour les règles $A \to Y_1 \dots Y_k$:
   (a) Ajouter les symboles de PREMIER(Y_1) dans PREMIER(A) ;
   (b) S'il existe un intervalle $[1..l]$ tel que $\forall i \in [1..l]$, $\varepsilon \in$ PREMIER(Y_i), ajouter à PREMIER(A) tous les symboles des PREMIER(Y_i) pour $i \in [1..l+1]$.
   (c) Si $k = l$ (ie ε appartient à tous les PREMIER(Y_i)), alors ajouter ε à PREMIER(A).

Par extention (avec les mêmes précautions concernant ε), on peut calculer PREMIER(β) pour tout $\beta \in (X \cup V)^*$.

Calcul de SUIVANT() Réitérer jusqu'au point fixe :

1. Mettre \$ dans SUIVANT(S).
2. Si $A \to \alpha B \beta$, le contenu de PREMIER(β) (sauf ε) est ajouté à SUIVANT(B).
3. S'il existe une règle $A \to \alpha B$ (ou une règle $A \to \alpha B \beta$, avec $\varepsilon \in$ PREMIER(β)), les éléments de SUIVANT(A) sont ajoutés à SUIVANT(B).

Construction de la table $LL(1)$

1. Pour la règle n^o i de la forme $A \to \alpha$:
   (a) Pour tout $a \in$ PREMIER(α), ajouter i à la case (A, a).
   (b) Si $\varepsilon \in$ PREMIER(α), ajouter i à la case (A, b) pour chaque $b \in$ SUIVANT(A). Si $\varepsilon \in$ PREMIER(α) et \$ $\in$ SUIVANT(A), ajouter i à la case $(A, \$)$.
2. Marquer erreur dans toutes les cases restées vides.

d'autres grammaires, non ambigües, dont on a pu montrer qu'aucun regard en avant de taille bornée ne permettra une analyse descendante déterministe.

Encore un point à noter : les grammaires LL(k) forment des familles imbriquées strictement les unes dans les autres, pour tout k.