Université Paris-Saclay - Faculté des Sciences d'Orsay - L2 Informatique *Principes d'Interprétation des Langages* TD

Cours de Langage Formel (anciennement PIL) 2025-2026 Frédéric Gruau

s1 Introduction

Source: Le cours s'est beaucoup inspiré de `https://www.lix.polytechnique.fr/~jouannaud/articles/cours-info-theo.pdf`. Ce manuscrit en libre accés est très clair, et très complet, mais ne contient pas les figures. Pour l'analyse ascendante, j'ai utilisé aussi `https://www.dicosmo.org/CourseNotes/Compilation/0304/Cours05/notes.pdf`. Les cours et TD de LF fait pendant le confinememt, en 2021, sont accessible sur la chaine utube "cours, TD, TP à distance de Frédéric Gruau" sur la play liste "cours et TD de language formel",

`https://www.youtube.com/playlist?list=PLyZNfaifI2hWDdvrtM55ttGEb_7Cb7pd5`

0.1 Déroulement, synchro sur TD

Le cours est fait au tableau, cela est plus vivant, et plus interactif. Ce support ne comprend pas les exemples nombreux développés en cours. Son utilité principale et de faciliter le rattrapage en cas d'absence, et de permettre aux étudiants de pas être obligé de tout le temps prendre des notes. Il permet également d'avoir une vue globale sur tout le contenu. Le cours est étroitement synchronisé avec les TDs, on peut le voir comme une préparation aux TDs. Il est par ailleurs structuré par rapport aux examens. Chaque cours fait l'objet d'un TD, et d'un exercice au partiel ou à l'examen. Certaines notions sont abordées en cours, mais pas en TD, et font néanmoins l'objet de questions aux examens.

Il y a 12 cours(1:30) suivi de 12 TD(2h) avec le découpage suivant :

- 5 semaines sur Langage formel, expression rationnelle et leur automate d'état fini
- 4 semaines sur Grammaire hors contexte et leur automate à pile
- 3 semaines sur l'analyse syntaxique.

— semaine 1 —

0.2 Panoramique

Un langage formel est un ensemble de mots le plus souvent infini. Les langages formels furent initialement utilisés pour formaliser les langues naturelles. Ils sont au cœur du traitement automatique des langages de programmations. C'est également la base de toute l'informatique théorique. On distingue trois niveaux de langage décrit dans la table 1. Chaque niveau est associé à un type de machine abstraite capable de reconnaître les mots du langage. Pour les deux premiers niveaux, la machine abstraite est générée automatiquement, nous verrons comment le faire. Le troisième niveau est le cas général de ce qu'on peut calculer avec un ordinateur (théorie de la calculabilité). Il ne sera pas abordé dans ce cours, mais dans le cours d'informatique théorique en L3.

0.3 Lien avec la compilation

L'exemple de référence de langage formel est tous simplement l'ensemble des programmes écrits dans un langage de programmation donné. On n'a pas l'habitude de dire qu'un programme est un mot ; cependant, du point de vue de LF, un mot n'est rien d'autre qu'une suite de lettres, et ces lettres comprennent aussi l'espace ou les caractères de ponctuation. La compilation est le procédé par lequel on transforme ce mot en un code binaire exécutable par une machine. On effectue d'abord une analyse lexicale qui transforme le texte initial en une suite de tokens. Pour un programme de 1000 caractères, il y aura par exemple seulement 200 tokens. En effet, chaque token s'écrit sur plusieurs lettres. Un exemple de token est un identificateur « toto » qui prend 4 lettres, ou un

Nom	Description du langage	phase de compilation	machine abstraite
Langage régulier	Expression rationnelles	Analyse lexicale	Automate d'état fini
Langage algébrique	Grammaires hors contexte	Analyse syntaxique	Automates à piles
Langage décidable	Grammaire avec contexte.	Typage	Machine de turing

TABLE 1 – Les trois niveaux de langages principaux.

nombre « 2026 » qui prend aussi 4 lettres, en considérant qu'un chiffre est aussi pour nous une lettre. L'ensemble des tokens possibles est aussi un langage formel. Il est beaucoup plus simple que l'ensemble des programmes Les mots qui représentent des tokens sont des langages très simples, formés en langage qu'on appelle « réguliers ». Les machines capables de reconnaître si un mot donné appartient à un langage régulier donné sont les automates d'états finis. Ce sont ces machines qui sont à l'œuvre durant l'étape d'analyse lexicale, qui saucissonnent notre programme en une suite de tokens. Après l'analyse lexicale vient l'analyse syntaxique qui permet en même temps de vérifier la syntaxe du programme, et de construire sa structure, à partir de laquelle on pourra finalement faire l'analyse semantique qui regroupe tout un tas de calculs pour au final pondre du code exécutable. Vous imaginez bien que cette étape syntaxique est nettement plus complexe que l'étape lexicale. Pour la mener à bien, nous aurons besoin d'ajouter une pile à notre automate d'état fini, qui ne sera donc plus d'état fini, car on peut empiler sur une pile autant qu'on veut, et c'est donc une mémoire non bornée, non finie. La syntaxe est décrite par des grammaires dites « hors contexte » ; nous verrons comment on peut automatiquement générer un automate à pile permettant de mener l'analyse, à partir d'une grammaire donnée. Il faudra que la grammaire vérifie certaines bonnes propriétés. Nous utiliserons en priorité des exemples simples de grammaire, mais nous ferons quand même le lien avec de vraies grammaires de langages de programmation.

1 Langages formels

Definition 1 *Un alphabet est un ensemble fini noté* Σ *dont les éléments sont appelés lettres, Un mot est une suite finie de lettres, notée* $u = u_1 \dots u_n$.

La longueur du mot u notée $|u|$ est le nombre de lettres. On note Σ^* l'ensemble des mots sur Σ, ϵ est le mot de longueur nulle, appelé mot vide.

Definition 2 *L'ensemble des mots est muni d'une opération interne, le "produit de concaténation", telle que si* $u = u_1 \dots u_n$ *et* $v = v_1 \dots v_p$ *, alors le produit* $u.v$ *est le mot* w *tel que* $w_i = u_i$ *pour* $i \in [1..n]$ *et* $w_{n+j} = v_j$ *pour* $j \in [1..p]$.

La concaténation des mots est associative et possède ϵ pour élément neutre. On note le produit sous la forme $u.v$

La puissance n^{ime} d'un mot est définie par récurrence sur n comme suit : $u_0 = \epsilon$ et $u_{n+1} = u.u_n$. Muni du produit de concaténation et de son élément neutre, $(\Sigma^*, ., \epsilon)$ est un monoïde libre sur Σ : ici, l'adjectif libre signifie que tout mot u est soit le mot vide ϵ, soit commence par une certaine lettre a auquel cas il s'écrit de manière unique sous la forme $a.v$ pour un certain mot v De taille diminuée d'une unité. Il sera donc possible de prouver une propriété P d'un ensemble de Mots en montrant $P(\epsilon)$, puis en montrant $P(a.v)$ en supposant $P(v)$. Exemple: prouvons que $|v \cdot w| = |v| + |w|$. Il faut d'abord définir par induction la concaténation comme suit: $v \cdot \varepsilon = v$, pour tout $v \in A^*$. Si $w \in A^*$, $a \in A$ et $v \in A^*$, alors $v \cdot (wa) = (v \cdot w)a$. Et la longeur comme suit: $|\varepsilon| = 0$. Si $w \in A^*$ et $a \in A$, alors $|wa| = |w| + 1$.

Démontrons à présent que pour tous $u, v \in A^*$, on a $|v \cdot u| = |v| + |u|$.

Préuve. Fixons $v \in A^*$ et faisons une induction sur u.

Cas de Base. Pour $u = \varepsilon$, $|v \cdot \varepsilon| = |v| = |v| + 0 = |v| + |\varepsilon|$.

Induction. Supposons que pour un $w \in A^*$ on ait $|v \cdot w| = |v| + |w|$. Prenons $u = wa$ avec $a \in A$. Alors, par la définition de la concaténation, $v \cdot (wa) = (v \cdot w)a$. Par la définition de la longueur, $|v \cdot (wa)| = |(v \cdot w)a.| = |w \cdot v| + 1$. Par l'hypothèse de récurrence, $|w \cdot v| + 1 = (|w| + |v|) + 1 = (|w| + 1) + |v| = |wa| + |v|$. Donc $|(wa) \cdot v| = |wa| + |v|$. Donc par induction, la propriété vaut pour tout $u \in A^*$.

Vocabulaire: préfixe, sufixe, facteur Préfixe : le mot u est un préfixe du mot w si et seulement s'il existe un mot v tel que $w = uv$ (le mot u est un début de w) Le mot ab est un préfixe du mot abba mais n'est pas un préfixe du mot aabb Suffixe : le mot u est un suffixe du mot w si et seulement s'il existe un mot v tel que $w = vu$ (le mot u est une fin de w) Le mot ba est un suffixe du mot abba mais n'est pas un préfixe du mot aabb Le mot vide ϵ est un préfixe de n'importe quel mot. Le mot u est un facteur du mot w si et seulement s'il existe deux mot v_1, v_2 tel que $w = v_1uv_2$ (le mot u apparait au milieu de w)

Definition 3 *: Un langage est un ensemble de mots*

Ceci est un fait simple, mais beaucoup d'élèves l'oublient. Le plus souvent, on travaillera avec des ensembles infinis, car c'est là où il y a de la structure. Les langages finis sont tous tristement équivalents d'un point de vue théorique. Le langage vide noté $\emptyset$ ne possède aucun mot. Le langage unité $\{\epsilon\}$ est réduit au mot vide. On définit à nouveau des opérations sur les langages, opérations ensemblistes classiques: $L_1 \cup L_2$ désigne l'union des langages L_1 et L_2, $L_1 \cap L_2$ désigne l'intersection des langages L_1 et L_2, $\Sigma^* \setminus L$ désigne le complémentaire dans Σ^* du langage L. La concaténation peut s'étendre à l'ensemble de mots:

- $L_1.L_2 = \{u.v | u \in L_1 \text{ et } v \in L_2\}$ désigne le produit des langages L_1 et L_2. Attention, ce n'est pas le produit ensembliste. $\{aa, a\}.\{ab, aab\} = \{aab, aaab, aaaab\}$. On n'a pas l'égalité $|L_1.L_2| = |L_1||L_2|$. mais l'inégalité $|L_1.L_2| \leq |L_1||L_2|$
- L^n tel que $L^0 = \{\epsilon\}$ et $L^{n+1} = L.L_n$ désigne la puissance n^{ieme} du langage L
- $L^* = \cup_{i/i \in N} L^i$ désigne l'itéré du langage L. L'étoile s'appelle " l'étoile de Kleene » du nom de son inventeur. Elle permet de passer à l'infini $(aa)^* = \epsilon, aa, aaaa, aaaaaa, \ldots =$ mot ayant un nombre pair de a $= \{u \in a*/|u|_a mod\ 2 = 0\}$.

$L^+ = \cup_{i>0} L_i$ désigne l'itéré strict du langage L. L'étoile de Kleene est ce qui est extrêmement nouveau et particulièrement difficile à comprendre et à maîtriser. Il faut mémoriser que L^* est l'ensemble des séquences de mots de L. Plusieurs exercices du TD sont dédiés à comprendre l'étoile de Kleene. **Exemple** a$(aa)^*$ nombre impair de a

On démontre l'égalité entre deux langages de deux manières:
1-Par des manipulation algébriques en utilisant les propriétés ensemblistes de l'union et de du produit, par exemple: L'associativité de l'union, $(A \cup B) \cup C = A \cup (B \cup C)$, la distributivité de l'union sur le produit $(A \cup B).C = (A.C) \cup (B.C)$
2- Par la double inclusion. $L_1 = L_2$ ssi $L_1 \subset L_2$ et $L_2 \subset L_1$

Exemple : démontrer la distributivité de l'union sur le produit avec double inclusion.

— semaine 2—

Le lemme suivant sera utilisé pour montrer comment calculer le langage reconnu par un automate. C'est une preuve pas trop compliquée et un joli exemple de double inclusion.

Theoreme 1 *Lemme d'arden: Si A et B sont deux langages, l'équation $L = A.L \cup B$ admet A^*B comme solution , de plus Si A ne contient pas ϵ, cette solution est unique.*

Preuve. A^*B est une solution. Le vérifier. Elle est unique? Soit L une solution, montrons que $L = A^*.B$, comment? Par double inclusion!

1- $A^*.B \subset L$ en substituant $n-1$ fois L par $A.L \cup B$ dans le membre droit, on obtient $L = A^nL \cup A^{n-1}B \cup A^{n-2}B \cup ... \cup B$ On voit apparaître $A^*.B$, qui est donc inclus dans L.

2- $L \subset A^*B$. Soit u dans L de longueur l on choisit $n = l + 1$ dans l'équation précédente $L = A^{l+1}L \cup A^lB \cup \ldots \cup A^{l-1}B \cup A^{l-2}B \cup \ldots \cup B$. Si A ne contient pas epsilon, alors les mots de A^{l+1} sont de longueur $> l$. Donc u n'est pas dans A^{l+1}. Donc il est dans le reste qui est une partie de A^*B.

2 s2 Expressions rationnelles

On fait l'usage systématique de l'union, du produit et de l'étoile de Kleene. On considère les mots écrits sur un certain alphabet de base, Σ, complété des signes + (union aussi notée |), ., *, (,)

Le langage « Rat » des expressions rationnelles sur l'alphabet Σ est défini par induction comme suit:

— une lettre de Σ est dans RAT
— ϵ est dans RAT
— Si e_1 et e_2 désignent des expressions rationnelles, alors
  — $e_1 + e_2$ est dans RAT (somme)
  — $e_1.e_2$ est dans RAT (produit)
  — e_1^* est dans RAT (itérée de e_1)
  — (e) est dans RAT

Précédence: étoile>produit > somme. Avec les expressions rationnelles, nous avons un langage de langages, car Chaque expression rationnelle dénote un langage définit aussi par induction:

— si u est une lettre ou ϵ, Lang$(u) = \{u\}$.
— Si e_1 et e_2 désignent des expressions rationnelles, alors
  1. Lang$(e_1|e_2)$ = Lang$(e_1)\cup$Lang$(e_2$)
  2. Lang$(e_1.e_2)$ = Lang(e_1) . Lang$(e_2$)
  3. Lang(e^*) = (Lang(e))*).
  4. Lang$((e))$ = Lang(e).

Abus de notation important: une expression rationnelle est identifiée au langage qu'elle dénote.

Définition de propriété par induction sur une expression rationnelle On peut définir des fonctions booléennes sur des expressions rationnelles par induction, par exemple pour savoir si le langage décrit contient au moins un mot non vide, la fonction booléenne "cont" sera définie par

— cont(ϵ)= faux
— pour toute lettre l cont(l)= vrai
— pour toutes expressions e_1, e_2
  cont$(e_1|e_2)$=cont(e_1)ou cont(e_2)
  cont$(e_1.e_2)$=(cont(e_1)et nonVide(e_2)) ou (cont(e_2)et nonVide(e_1))
  cont(e_1^*)) = cont(e_1)

On constate que définir cont oblige aussi à définir nonVide qui est vrai si une expression rationnelle contient quelque chose.

Identités remarquables : Deux expressions rationnelles distinctes peuvent dénoter le même langage. Exemple: $(a|b)^*$ et $(a^*b^*)^*$ dénotent tous deux le langage des mots quelconques sur l'alphabet $\{a, b\}$.
1.$r|s = s|r$ commutativité union
2.$(r|s)|t = r|(s|t)$ assoicativité union
3.$(rs)t = r(st)$assoicativité produit
4.$r(s|t) = rs|rt, (r|s)t = rt|st$ distributivité
6.$\emptyset^* = \epsilon$
7.$(r^*)^* = r^*$ séquence de séquence = séquence
8.$(r^*s^*)^* = (r|s)^*$

3 Automates d'états finis.

C'est un formalisme très général qui peut modéliser des dispositifs automatiques, des systèmes réactifs, des objets mathématiques ou physiques, des circuits digitaux, le contrôle d'un microprocesseur, la solidité d'un mariage...

3.1 Automates déterministes

Definition 4 *Un automate fini déterministe A est un triplet (Σ, Q, δ) où*
1. Σ est l'alphabet de l'automate ;
2. Q est l'ensemble fini des états de l'automate
3. $\delta : Q \times \Sigma \to Q$ est une application partielle appelée fonction de transition de l'automate.

Si $\delta(q, a) = q'$, on peut noter cela $q - a \to q'$. Exercice: Donner l'automate à trois états qui reconnaît les entiers en binaire, sans zéro redondant. $q_0 - 1 \to q_1 - 0, 1 \to q_1; q_0 - 0 \to q_2; q_1, q_2$ finaux. On peut représenter un automate par le graphe de ses états, ou bien en listant les valeurs prises par la fonction de transitions.

Definition 5 *Étant donné un automate déterministe $A = (\Sigma, Q, \delta)$, et un mot $u = u_0, u_1, ...u_n$ On appelle calcul associé au mot une suite de transitions $q_0 - u_1 \to q_1 \to ... - u_n \to q_n$. On écrit $q_0 - u \to q_n$*

Le calcul produit par la lecture d'un mot u par un automate fini déterministe est automatique : La lecture des lettres composant le mot provoque des transitions bien définies jusqu'à être bloquée en cas de transitions manquantes, ou bien jusqu'à atteindre un certain état après la lecture complète du mot. Lorsque δ est totale, l'automate est dit complet. On en déduit la propriété fondamentale des automates déterministes complets :

Theoreme 2 *Soit $A = (\Sigma, Q, \delta)$ un automate fini déterministe complet. Alors, pour tout mot $u \in \Sigma^*$ et tout état $q \in Q$, il existe un unique état $q' \in Q$ tel que $q - u \to q'$*

On peut alors étendre l'application δ aux mots, en posant $\delta(q,u) = q' \in Q$ tel que $q - u \to q'$. Un automate qui n'est pas complet le devient si on ajoute un nouvel état appelé poubelle vers lequel vont toutes les transitions manquantes. Et quand on est dans la poubelle, on y reste. Exercice: Compléter l'automate précédent. C'est pas forcément une bonne idée de rajouter une poubelle.

Definition 6 *Un automate déterministe vient avec la donnée d'un état initial $q_0 \in Q$ et d'un ensemble d'états finaux $F \subseteq Q$;*

On repère l'état initial par une flèche entrante sans source, et les états finaux par des flèches sortantes sans cible, ou bien un double cercle. Un mot w est reconnu par l'automate S'il existe un calcul dit réussi issu de l'état initial q_0 et terminant dans un état final après avoir lu le mot w. On note Lang(A) le langage des mots reconnus par l'automate A. Un langage reconnu par un automate est dit reconnaissable. On appelle REC l'ensemble des langages reconnaissables. Il est clair que l'ajout d'une poubelle ne change pas les mots reconnus. Si l'automate A est complet, on peut reformuler la condition d'acceptation des mots comme $u \in \text{Lang}(A)$ ssi $\delta(q_0, u) \in F$.

3.2 Automates non déterministes

Definition 7 *Un automate non-déterministe A est un triplet (Σ, Q, δ) où*
1. Σ est l'alphabet de l'automate
2. Q est l'ensemble des états de l'automate
3. $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ la fonction de transition.

Exemple: L'Automate des mots qui contiennent "aa": $q_0 - a, b \to q_0 \to a \to q_1 - a \to q_2 - a, b \to q_2; q_2$ final.

On notera comme précédemment $q - \alpha \to q$ pour $q \in \delta(q, \alpha)$ avec $\alpha \in \Sigma$. $\delta(q,u)$ est l'ensemble (peut-être vide) des états atteignables depuis q en lisant le mot u. Automate déterministe = cas particulier d'automate non déterministe. La notion de calcul est la même, non déterministe/déterministe. Il peut y avoir plusieurs calculs issus de l'état initial q_0, qui lisent un mot w donné, dont certains peuvent se bloquer. D'autre part, si un calcul échoue, cela ne veut rien dire ; il faut tout explorer. Reconnaissance = au moins un calcul démarrant sur l'état initial, arrivant à un final.

Déterminisation. Si Q est l'ensemble des états d'un automate non-déterministe, l'ensemble des états de l'automate déterministe associé Sera $\mathcal{P}(Q)$, l'ensemble des parties de Q. Il y en a exponentiellement plus.

Definition 8 *Soit $A = (\Sigma, Q, \delta, q_0, F)$ un automate non déterministe. On définit Det(A) comme l'automate $(\Sigma, P(Q), \delta_{det})$ où $\delta_{det}(K, a) = \cup_{q \in K} \delta(q, a)$.. L'état initial est $\{q_0\}$. Les états finaux sont $\{K \in \mathcal{P}(Q) | K \cap F \neq \emptyset\}$*

Theoreme 3 *Soit A un automate non déterministe. Alors Det(A) est déterministe et reconnaît le même langage que A.*

Sketch de la Preuve: par double inclusion:
Si A accepte un mot w, alors il existe une suite d'états possibles. Par construction, cette suite d'états est contenue dans les ensembles visités par le Det(A), donc Det(A) accepte aussi w
Si Det(A) accepte w, cela veut dire que dans l'ensemble d'états atteints, il y a un état final q_f de A ; en remontant les ensembles d'états visités, on peut construire un chemin de q_0 vers q_f tel que A peut suivre ce chemin et aussi accepter w.

Algo de déterminisation: On n'a pas besoin, en pratique, de lister tous les ensembles d'états possibles, ça serait en effet très long. On liste seulement ceux qu'on peut atteindre en démarrant de l'état initial qui est le singleton $\{q_0\}$. Au fur et à mesure qu'on découvre de nouveaux ensembles, on recalcule les transitions qui en partent. Exercice : Déterminiser l'automate qui reconnaît les mots contenant aa. On fait trois colonnes contenant des ensembles d'états: colonne de gauche: les ensembles visités, colonne du milieu: transitions par 0, colonne de droite: transition par 1. Après, on peut renuméroter les états, et/ou dessiner le graphe de l'automate, sans oublier d'indiquer l'état initial, et les états finaux qui sont les ensembles qui contiennent un final.

Les automates déterministes et non déterministes reconnaissent les mêmes langages. les automates déterministes sont parfois exponentiellement plus gros, la borne pouvant être atteinte, c'est illustré dans le TD.

—Semaine 3 —

3.3 s3 Transitions vides

Une transition vide est une transition étiquetée par ϵ. Cela dénote qu'on a pas besoin de lire de lettre pour effectuer la transition.C'est une autre façon parfois très pratique, d'exprimer le non-déterminisme. Un automate avec transitions vide est aussi appelés "asynchrone".

Definition 9 *Un automate non déterministe A avec transitions vides est un triplet* (Σ, Q, δ) *où 1.* Σ *est l'alphabet de l'automate*
2. Q *est l'ensemble des états de l'automate*
3. $\delta : Q \times (\Sigma \cup \epsilon) \to P(Q)$ *est la fonction de transition de l'automate.*

On notera $q - \alpha \to q'$ pour $q' \in \delta(q, \alpha)$, avec $\alpha \in \Sigma$ ou $\alpha = \epsilon$.

Exercice: construire un automate non déterministe avec transitions vides reconnaissant le langage sur l'alphabet $\{0, 1\}$ contenant au moins une occurrence du mot "00", puis un autre similaire en remplaçant "00" par "11" puis faire la réunion des deux.

$q_0 - \epsilon \to q_1, q_2$.
$q_1 - 1 \to q_1 - 0 \to q_3 - 0 \to q_4$
$q_2 - 0 \to q_2 - 1 \to q_5 - 1 \to q_6$
$q_5 - 0, 1 \to q_5$
$q_6 - 0, 1 \to q_6$
q_0 initial, q_5, q_6 finaux
Moralité: avec des epsilon-transitions, on construit facilement un automate qui reconnaît la réunion de deux langages, à partir des deux automates qui reconnaissent chacun des langages. Cette technique sera utilisée pour la démonstration du théorème de Kleene;

Note : Les calculs d'un automate avec transitions vides autorisent le passage par un nombre quelconque de transitions vides au cours de l'exécution. Le nombre d'états parcourus peut donc être bien supérieur au nombre de lettres lues.

Élimination des transitions vides. Nous voulons maintenant montrer que le langage reconnu par un automate avec transitions vides peut également l'être par un automate non déterministe sans transitions vides. Il faut ajouter de nouvelles transitions dans l'automate :
1- Pour chaque chemin d'un état s à un état t formé de epsilon-transitions, et pour chaque transition de t à un état u portant une lettre a, ajouter une transition de s à u d'étiquette a ;
2- Pour chaque chemin d'un état s à un état t terminal formé de epsilon-transitions, ajouter s à l'ensemble des états terminaux ;
Notons que cette construction n'augmente pas le nombre d'états.

Exercice enlever les epsilon transitions de l'automate précédent. Attention, y a des transition de q_0 vers q_3 et q_5 lorqu'on dés-epsilonne.

Formulation avec les epsilon cloture. La notion d'épsilon-clôture permet une autre construction du désepsilonné, plus simple et plus puissante:

1. On calcule les epsilon-clôtures de chaque état q: L'epsilon-clôture de q est l'ensemble des états que l'on peut atteindre depuis q en lisant epsilon. On peut possiblement rebondir sur plusieurs epsilon de suite, pour cette raison, on appelle cela une "clôture transitive", ce qui est un concept générique que nous retrouverons. Les epsilon-clôtures forment les états du dé-epsilonné.
2. L'état initial est la clôture de q_0, les états finaux sont ceux dont la clôture contient un final
3. une transition de q_1 vers q_2 par la lettre x conduit à ajouter la même transition, mais de clôture(q_1) vers clôture(q_2)

Exemple, enlever à nouveau les epsilons avec cette nouvelle technique. Exemple 2, illustre l'intérêt des epsilon clôture: on gére les cycles : $q_0 - \epsilon \to q_a - \epsilon \to q_b - \epsilon \to q_c - \epsilon \to q_a$ et $q_x - x \to q_x$ Attention, il n'y a plus d'épsilon dans l'automate sur les clôtures, mais cela peut rester non déterministe quand même, comme illustré en TD.

4 Theoreme de Kleene.

Theoreme 4 *RAT=REC*

Preuve: on utilise la double inclusion

1-RAT $\subset$ REC.
Si E est une expression rationnelle, il existe un automate qui reconnaît le langage associé, demo par induction:
1-Cas de base
Si e=vide pas d'état
si $e = \epsilon$ un etat initial acceptant
si $e =$ une lettre, deux états

2-Induction
Faut montrer que si A reconnaît L_1 et A' reconnaît L_2 on peut construire un automate reconnaissant: $L_1|L_2, L_1.L_2, L_1^*$ faire des dessins.
Si $e = e_1|e_2$ on rajoute un état initial et un final et on recolle avec des epsilon, initial sur initiaux, finaux vers final (comme dans l' exo traité)
si $e = e_1.e_2$ suffit de rajouter un epsilon des finaux du premier vers l'initial du second.
si $e = e_1^*$ on rajoute des epsilon de final vers initial, en laissant les mêmes finaux ça reconnaît L_1^+ après, il faut rajouter epsilon en ajoutant un autre état initial et aussi final.

2-REC $\subset$ RAT. On calcule directement l'expression rationnelle du langage reconnu par un automate. il s'obtient par la résolution d'un ensemble d'équations à n inconnues, ou n est le nombre d'états.

Definition 10 *À chaque état $q_i, i = 0..n-1$, on associe un langage L_i appelé futur de q_i, qui contient les mots menant de q_i vers un état final.*

On a le système de n équations à n inconnues suivant: $L_i = Y_i|X_{i,0}.L_0|X_{i,1}.L_1|...|X_{i,n-1}.L_{n-1}$ ou $Y_i = \epsilon$ si q_i est dans F et $X_{i,j}=$ l'ensemble des lettres étiquetant les transition de q_i vers q_j. Ce système se résout avec le Lemme d'Arden, et par substitution. Cela donne n expressions rationnelles pour chacun des L_i et en particulier pour L_0 qui est le langage reconnu par l'automate.

Exemple: Soit L le langage des mots sur a, b contenant un nombre pair de a. L'automate est simple : Deux états q_0 et q_1, transition par b de q_0 vers q_0 et de q_1 vers q_1, Transition par a de q_0 vers q_1 et vice versa. q_0 à la fois initial et final. $L_0 = \epsilon|b.L_0|a.L_1$
$L_1 = b.L_1|a.L_0$
$L_1 = b^*a.L_0$ (arden)
on remplace $L_0 = \epsilon|b.L_0|a(b^*.a.L_0)$
On identifie pour appliquer Arden, une autre fois, on trouve $L_0 = (ab^*a|b)^*$

— semaine 4 —

5 s4 Minimisation d'un automate déterministe

5.1 relation d'équivalences

Il s'agit d'une relation binaire sur un ensemble donné vérifiant trois propriétés:

1. Réflexivité tout élément est en relation avec lui-même,
2. symétrie si un élément est en relation avec un autre, alors cet autre est aussi en relation avec lui.
3. transitivité. un élément est en relation avec un autre, lequel est aussi en relation avec un troisième, alors le premier est aussi en relation avec le troisième.

ex1 sur les nombres réels : $x \sim y$ si x et y ont même plafond (nombre entier au-dessus), quelles sont les classes?
ex2: sur les nombres entiers : $x \sim y$ si $x = y$ modulo2, puis modulo4. Quelles sont les classes?

Definition 11 *Une relation d'équivalence est dite "plus fine" qu'une autre, si elle est impliquée par cette autre.*

Exemple: modulo4 plus fine que modulo2

Theoreme 5 *Si une relation donnée est plus fine qu'une autre, ses classes subdivisent les classes de l'autre.*

Exemple: les classes de modulo4 subdivisent les classes de modulo2.

5.2 Exemple fil conducteur

Soit le langage $L = (a|b) * aba(a|b)*$. L'automate reconnaissant ce langage est $q_0 - b \to q_0 - a \to q_1 - a \to q_1 - b \to q_2 - a \to q_3 - a, b \to q_3$ et la transition $q_2 - b \to q0$. Cet automate est minimal déterministe complet, on rajoute des états pour le rendre non minimal: on dé-triple q_1 en rajoutant $q_1 - a \to q_1' - a \to q"1$ et on dédouble q_0, q_2, q_3, en donnant le même sens aux état de même indice: les états 0,1,2,3 attendent respectivement aba,ba,a,epsilon. Les états $q_1, q_1', q"1$ attendent les mêmes mots, ils ont même futur. Rappel: Futur$(q) = \{m/\delta(q, m) \in F\}$. Le futur de q_0 est le langage reconnu. Les futurs ont déjà été utilisés pour poser un système de n équations a n inconnues permettant de calculer le

langage reconnu par un automate donné, via le lemme d'Arden.

Pour notre automate: voici les quatre futurs possibles:

1. futur0= L,
2. futur1=ba(a|b)* | L
3. futur2=a (a|b)* | L,
4. futur3=(a|b)*

5.3 Minimisation d'automate.

Idée clef: les États ayant le même futur peuvent être fusionnés.

On va donc calculer la relation d'équivalence $\sim$ telle que $q \sim q'$ si q, q' ont le même futur. On va la calculer progressivement en considérant $\sim_k$, qui est vrai pour "ont même futur", en considérant seulement les mots de longueur $\leq k$. On augmente progressivement k en partant de zéro. Pour $k = 0$, les classes d'équivalence sont les finaux pour qui epsilon estLe futur, et les non-finaux dont le futur est vide. Les mot de longueur $\leq k+1$ contiennent les mots de longueur$\leq k$, cela implique que $\sim_{k+1}$ est plus fine que $\sim_k$ Pour passer de $\sim_k$ à $\sim_{k+1}$, les classes de $\sim_{k+1}$ s'obtiennent en subdivisant les classes de $\sim_k$. Deux mots de $k+1$ lettres ont le même futur, si après la lecture de la première lettre, on tombe dans un ensemble d'états qui ont le même futur de k lettres, c'est-à-dire dans une classe de $\sim_k$; On considère donc une à une chaque lettre a de l'alphabet, et chaque classe C de $\sim_k$, et on scinde en deux C si l'image des éléments de C par δ arrive dans des classes différentes de $\sim_k$. On scinde en regroupant les états de C qui ont la même image. On scinde donc pour passer de $k = 0$ à $k = 1$. On rescinde pour passer de $k = 1$ à $k = 2$...On ne peut pas subdiviser à l'infini, donc au bout d'un moment, on obtiendra $\sim_{k+1}=\sim_k$ L'algorithme aura convergé. À ce moment-là, on peut fusionner ensemble les états de chaque sous-ensemble de la partition. Cela va donner un automate, c.-à-d. l'image par δ sera cohérente. Sur l'exemple, on sépare d'abord $q_3, q'3$, puis $q_2, q'2$ etc. Montrez sur l'exemple, que si on fusionne avant convergence, l'image par δ n'est pas cohérente, on n'obtient pas un automate.

6 Propriétés de clôture des langages reconnaissables

Soit f une opération d'arité n sur les mots, et $L_1, ...L_n$ des langages. On définit le langage $f(L_1, \ldots, L_n) = f(u_1, \ldots, u_n)/u_i \in L_i$

On dit que les langages reconnaissables sont clos par une opération f si $f(L_1, \ldots, L_n)$ est reconnaissable lorsque les langages $L_1, \ldots, L_n$ le sont.

Les langages reconnaissables possèdent de très nombreuses propriétés de clôture, en particulier par les opérations ensemblistes simples: union, intersection, complémentaire. Par substitution (de lettres par des mots = homomorphisme), par homomorphisme inverse, par pompage, etc. (un homomorphisme ϕ de mots vérifie que $\phi(\epsilon) = \epsilon$ et $\phi(u.v) = \phi(u).\phi(v)$. Pour définir un morphisme, il suffit de connaître l'image de chaque lettre.

Les clôtures ont deux utilités :
1- montrer que certains langages sont bien reconnaissables,
2- montrer que certains langages ne sont pas reconnaissables, via une démonstration par l'absurde. Traiter l'exemple où le nombre de 'a' est égal au nombre de 'b'.

—Semaine 5—

7 s5 Pompage des langages reconnaissables

Le langage $L = \{a^n b^n, n \in N\}$ n'est pas reconnaissable. Par l'absurde, s'il l'était, soit un automate $A = (\Sigma, Q, \delta, q_0, F)$ qui le reconnaît. Considérons l'application ϕ: $n \mapsto \delta(q0, a^n)$. ϕ ne peut être injective, car son ensembe de départ est infini, et son ensemble d'arrivée est fini. Donc il existerai n, m tel que $\phi(n) = \phi(m) = q$. Mais alors $\delta(q_0, a^n b^n) = \delta(q_0, a^m b^n)$ est absurde : l'un appartient à F, l'autre pas.

Plus généralement, si un automate avec états dans Q, est complet sans transitions vide, tout calcul de plus de $n = |Q|$ états, passe deux fois par le même état durant les n premières transitions. Ce cycle peut etre supprimé, ou itéré. Faire un dessin.

Theoreme 6 *Tout langage L reconnaissable satisfait la propriété Pompe:*
$\exists N > 0$ *tel que*

$\forall m \in L$, *tel que* $|m| \geq N$
$\exists u, v, w \in \Sigma^*$ *tq* $m = uvw, v \neq \epsilon, |uv| \leq N$
$\forall k \in \mathbb{N}, uv^k w \in L$

Soit $m = uvw$ un mot sur le langage Σ . On dit que les mots de la forme $uv^k w$ pour $k \in N$ sont obtenus par pompage de v dans m.

Preuve: C'est le même argument que celui que l'on vient de développer pour montrer que le langage $\{a^n b^n | n \in N\}$ n'est pas reconnaissable. On se donne un automate A complet, sans transition vide reconnaissant le langage L, et on note $N = |Q|$ qui est positif strictement puisque A est complet. Soit maintenant $m \in L$ de taille au moins N , et $c = q_0 \rightarrow q_{|m|} \in F$ un calcul reconnaissant m. Comme l'automate n'a que $N \leq |m|$ états, il existe deux entiers i et j tels que $0 \leq i < j \leq N$ et $q_i = q_j$. Il existe donc trois mots u, v, w tels que : $c = q_0 - u \rightarrow q_i - v \rightarrow q_j - w \rightarrow q_{|m|} \in F$ et l'on vérifie immédiatement les conditions $m = uvw, v \neq \epsilon, |uv| \leq N$. De plus, comme $q_i = q_j$, le calcul $i = q_0 - u \rightarrow q_i - v^k \rightarrow q_j - \rightarrow q_{|m|} \in F$ reconnait le mot $uv^k w$ qui est donc dans L.

Contraposée du lemme de la pompe. Être pompable est une condition nécessaire, mais pas suffisante. Il existe des langages non reconnaissables qui satisfont le "Lemme de la pompe" (cf exercice optionnel TD). On ne peut donc pas déduire d'un langage qu'il est reconnaissable en montrant qu'il satisfait le "Lemme de la pompe". Par contre, on peut s'en servir pour montrer qu'un langage n'est pas reconnaissable, puisque, par contraposition, un langage qui ne satisfait pas le "Lemme de la pompe" ne peut pas être reconnaissable.

Exprimons donc la négation de la propriété de la pompe, cela se fait par application des règles usuelles de logique permettant de pousser les négations à l'intérieur des formules en inversant les quantificateurs $\forall, \exists$

$\neg$ Pompe =
$\forall N > 0$,
$\exists m \in L, |m| \geq N$tel que
$\forall u, v, w \in \Sigma^*$ telque $m = uvw, v \neq \epsilon, |uv| < N$,
$\exists k \in \mathbb{N}$ tel que $uv^k w \notin L$.

Traiter l'exemple du langage $a^n b^n$

8 Nettoyage des automates

On peut passer d'un automate non-déterministe avec des transitions vides vers un automate non déterministe sans transitions vides en temps linéaire en la taille, sans rajouter d'états $\Rightarrow$ la complexité des transformations est inchangée.

Qu'il soit ou non déterministe, un automate peut posséder des états retirables sans changer le langage reconnu.

Definition 12 *Étant donné un automate (déterministe ou pas, avec ou sans transitions vides)* $A = (\Sigma, Q, q_0, F, \delta)$*, l'état* $q \in Q$ *est accessible, s'il existe* w *tel que* $q_0 - w \rightarrow q$*; productif, s'il existe* w *tel que* $q - w \rightarrow f \in F$*; utile, s'il est à la fois accessible et productif.*

Un automate est dit réduit si tous ses états sont utiles. Nettoyer = enlever les états inutiles.

Un état q est productif ssi $q \in F$ ou bien il existe un état productif q' et une lettre $a \in (\Sigma \cup \epsilon)$ tels que $q' \in \delta(q, a)$. $\Rightarrow$ algorithme simple en temps linéaire, ajouts successifs d'états productifs à un ensemble réduit initialement à F. On marque les états. Pour obtenir un temps linéaire, il faut que les états puissent pointer vers les états précédents en plus des suivants, pour pouvoir remonter. Un état q est accessible ssi $q = q_0$ ou bien, il existe un état accessible q' et une lettre $a \in (\Sigma \cup \epsilon)$ tels que $q \in \delta(q', a)$. Algorithme similaire, temps linéaire, par ajouts successifs d'états accessibles à un ensemble réduit initialement à q_0.

Theorem 7 *Pour tout automate fini A, déterministe ou pas, il existe un automate fini de même nature sans états inutiles qui peut être obtenu à partir du premier en temps linéaire.*

9 Décision et temps lineaire.

Nettoyez sert à montrer que certains problèmes sont décidables et de plus rapidement. Un problème est décidable, s'il se présente sous la forme d'une question oui/non, et s'il existe un programme qui répond oui ou non, au bout d'un temps fini. Rapide signifie en général, en temps linéaire, c'est-à-dire proportionnel à la taille du problème Un problème deux fois plus gros prend un temps deux fois plus gros. On ne

considère que des problèmes décidables dans ce cours. En examen, si on vous pose la question « Ce problème est-il décidable ? » '? Il faut comprendre: donnez un algo pour résoudre ce problème. On formule les choses avec l'adjectif « décidable » pour vous préparer au fait que oui, il existe des problèmes indécidables, pour lesquels il n'y a pas d'algo qui puisse répondre. C'est le cours d'info théorique du L3.

Décision du vide

Theoreme 8 *Savoir si le langage reconnu par un automate quelconque est vide est décidable en temps linéaire.*

Preuve: Le langage reconnu est vide ssi l'automate nettoyé n'a plus d'état, hormis l'état initial.

Décision du plein

Theoreme 9 *Savoir si un automate déterministe (resp. non déterministe) reconnaît tous les mots est décidable en temps linéaire (resp. exponentiel).*

Le langage reconnu est dit plein, s'il contient tous les mots possibles. Si l'automate est déterministe complet, le problème « du plein » a la même complexité que le problème « du vide » que nous venons de considérer, puisqu'ils s'échangent par permutation des états acceptants et non-acceptants. Cela n'est pas vrai des automates non déterministes. Dans le cas des automates déterministes : le passage au complémentaire se fait en temps linéaire sans changer la taille de l'automate. Pour les automates non déterministes, il est nécessaire de déterminiser au préalable, d'où le saut de complexité.

— semaine 6 —

10 s6 Grammaire.

- Décrit des langages
- Inspirés des grammaires de langage naturel, exemple: Phrase → sujet verbe complément.
- Différence: une grammaire peut générer des mots arbitrairement longs ; les phrases restent de longueur raisonnable, pour pouvoir être traitées par nos petits cerveaux.

Definition 13 *Une grammaire G est un quadruplet (Σ_N,Σ_T, S, R) où*

1. Σ_T *= alphabet de symboles dits terminaux;*
2. Σ_N *alphabet de symboles non-terminaux;*
3. $\Sigma = \Sigma_N \cup \Sigma_T$ *= alphabet total de G ;*
4. *S est un non-terminal appelé axiome ;*
5. $R \subseteq \mathcal{P}(\Sigma^\star \times \Sigma^\star)$ *est un ensemble fini de règles notées* $g \to d$ *si* $(g, d) \in R$.

Notation: On utilisera des majuscules pour les non-terminaux et des minuscules pour les terminaux.

Exemple 1 La grammaire $S \to aSBC, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc$. Faire une dérivation, montrer que cela génère $a^n b^n c^n, n > 0$. Notation: on utilise un trait vertical pour regrouper les différents membres droits associés à un même membre gauche. Dans la grammaire ci-dessus, on peut ainsi noter $S \to aSBC|\epsilon$

10.1 Langage engendré.

On utilise les règles d'une grammaire, pour réécrire un mot $u \in \Sigma^\star$ en un nouveau mot $v \in \Sigma^\star$. Il suffit de repérer une occurrence (quelconque) d'un membre gauche de la règle présent dans u et de le remplacer par le membre droit de cette règle.

Definition 14 *Étant donné une grammaire G =* (Σ_T , Σ_N , S, R), *On dit que le mot* $u \in \Sigma^\star$ *se récrit en le mot* $v \in \Sigma^\star$ *dans G avec la règle* $g \to d$, *et on note* $u - g, d \to v$ *si* $u = w_1 g w_2$, *et* $v = w_1 d w_2$;

On peut aussi noter plus simplement $u \to v$. Plus généralement, on dit que le mot $v \in \Sigma^\star$ dérive du mot $u \in \Sigma^\star$, dans la grammaire G, et on note $u \to^* v$, (fermeture transitive de $\to$) s'il existe une suite finie $w_0, w_1, \ldots, w_n$ de mots de $\Sigma^\star$ telle que $w_0 = u, w_i \to w_{i+1}$ pour tout $i \in 0, \ldots n-1$, et $w_n = v$. On peut indiquer le non-terminal récrit en le soulignant. La réécriture est itérée à partir de l'axiome jusqu'à l'élimination complète des non-terminaux.

Definition 15 *Le langage engendré par la grammaire G est l'ensemble des mots de* $\Sigma_T^\star$ *qui dérivent de l'axiome de G, que l'on note par Lang(G) ou L(G).*

Classification de Chomsky:

- type 0: membre gauche arbitraire membre droit arbitraire (Machine de Turing)
- type 1: on passe
- type 2, hors contexte: membre gauche = un seul non-terminal (Automate à pile)
- Type 3, régulier: hors contexte + membre droit contient un seul non-terminal toujours tout à la fin (Automate d'état fini)

Exemple2 (grammaire régulière): La grammaire $N \to 0|1M, M \to 0M|1M|\epsilon$ génère les entiers naturels en représentation binaire, sans zéros redondants: par exemple, 01 n'est pas dedans.

10.2 Grammaires hors contexte.

Les grammaires régulières type 3 sont trop simples; les langages qu'elles génèrent sont les langages rationnels, pourquoi s'ennuyer à définir un outil puissant pour rester dans ce monde limité?

Les grammaires type 1 sont trop compliquées pour nous, elles peuvent générer n'importe quel langage dès l'instant où il existe un algorithme qui peut le faire.

On va se concentrer exclusivement sur les grammaires hors contexte, de type 2, elle sont suffisamment puissantes pour décrire la syntaxe des langages de programmation, et suffisamment simples pour autoriser automatiquement une analyse de cette syntaxe. On dit "hors-contexte", car le non-terminal du membre gauche décide tout seul (sans contexte) comment il souhaite se réécrire.

La grammaire de l'exemple 1 n'est pas hors contexte, car les membres gauches contiennent plusieurs lettres.

Exemple 3 (Grammaires hors contexte canonique) La grammaire $S \to \epsilon|aSb$ génère le langage $\{a^n b^n | n \geq 0\}$.

Si la grammaire est hors-contexte, le langage généré est dit ALGÉBRIQUE. On considère seulement ceux-là dans la suite. Ils incluent tous les langages de programmation usuels. Le langage $\{a^n b^n | n \geq 0\}$ est algébrique, on vient d'en donner une grammaire hors contexte. Par contre, le langage $\{a^n b^n c^n | n \geq 0\}$ ne l'est pas. On en a donné une grammaire, mais celle-ci n'était pas hors contexte.

10.3 Arbre de dérivation

Considérons la grammaire suivante: $\Sigma_T = \{+, *, (,), Id, Cte\}, \Sigma_N = \{E\}, E \to Id|Cte|E + E|E * E|(E)$. En vrai, Cte et Id représentent des constantes ou des identificateurs. Pour l'analyse syntaxique, on considère toutes les constantes (resp. tous les identificateurs) comme le même terminal. Faire deux dérivations toutes les deux droites de $x+4*y$ distinctes, montrer que le sens diffère;

Si la grammaire est hors contexte on peut représenter une dérivation par un arbre :

Definition 16 *Étant donné une grammaire G = (Σ_T , Σ_N , S, R), les arbres de dérivation de G sont des arbres avec la racine (resp. les nœuds internes, les feuilles) étiqueté(es) par l'axiome, (resp. des non-terminaux, des terminaux) vérifiant de plus que si les fils pris de gauche à droite d'un nœud interne étiqueté par le non- terminal N sont étiquetés par les symboles respectifs $\alpha_1, ..., \alpha_n$, alors $N \to \alpha_1 \dots \alpha_n$ est une règle de la grammaire G.*

Un arbre de dérivation résume plusieurs dérivations possibles, réalisées avec un ordonnancement différent.

10.4 Ambiguïté

Deux arbres différents, cela implique un non-déterminisme, et aussi deux calculs différents. On n'aime pas, on va définir l'ambiguïté comme suit, et chercher ensuite à l' éviter.

Definition 17 *Une grammaire G est ambiguë s'il existe un mot w ∈ Lang(G) qui possède plusieurs arbres de dérivation dans G.*

Démontrer l'ambiguïté est facile, il suffit d'exhiber deux arbres de dérivation pour un mot. Par contre, démontrer qu'une grammaire n'est pas ambiguë est difficile et considéré comme hors programme. Cela requiert une démonstration par récurrence sur la profondeur des arbres de dérivation. En TD, on se convaincra de la non-ambigüité d'une grammaire en construisant l'arbre de dérivation d'un mot représentatif, et en constatant qu'on n'a jamais de choix durant cette construction.

Comment résoudre l'ambiguité?(TD) Méthode 1, bricolage, on introduit d'autres non-terminaux pour forcer un ordre:
$E \to E + T|T$
$T \to T * F|F$
$F \to id|cte|(E)$
Méthode 2, la méthode utilisée en pratique, car plus simple et plus élégante. On utilise des méta règles, extérieures à la grammaire:

- priorité de * sur +
- Associativité à gauche de $*, +$.

Definition 18 *Réécriture droite (resp gauche) on réécrit le plus a droite (resp. gauche)*

La réécriture droite (resp. gauche) correspond à un parcours droit (resp. gauche) de l'arbre de dérivation. On parle aussi de dérivation droite et dérivation gauche. Il y a une correspondance biunivoque entre dérivation gauche (resp. droite) et arbre de dérivation.

Propriété: une grammaire G est non ambigüe si et seulement si tout mot a une seule dérivation droite (resp. gauche).

— semaine 7—

10.5 s7 Nettoyage grammaire.

Definition 19 *Une grammaire hors-contexte $G = (\Sigma_T , \Sigma_N , S, R)$ est dite propre si elle vérifie :*

1. $\forall N \to u \in R, u \neq \epsilon$ *ou* $N = S$
2. $\forall N \to u \in R$, *On n'a pas de S dans u*
3. *Les non-terminaux sont tous utiles, c'est-à-dire à la fois atteignable et productif.*
4. *Il n'y a pas de règles ou on remplace un non terminal par un autre.*

Un non-terminal est dit atteignable si on peut le générer depuis l'axiome, il est dit productif s'il peut générer une chaîne de terminaux; Donner des exemples négatifs pour illustrer;

Theoreme 10 *Pour toute grammaire hors contexte $G = (\Sigma_T , \Sigma_N , S, R)$, il existe une grammaire hors-contexte G' propre qui engendre le même langage.*

Preuve: La mise sous forme propre d'une grammaire hors contexte est la succession de 5 étapes qui terminent.

1. On rajoute une règle $S' \to S, S'$ devenant le nouvel axiome ;

2. Élimination des $M \to \epsilon$. Calculer l'ensemble $E = \{M \in \Sigma_N | M \to^* \epsilon\}$;
Pour tout $M \in E$ Faire
Pour toute règle $N \to \alpha M \beta$ Faire
Ajouter la règle $N \to \alpha\beta$
Fin Faire ;
Fin Faire
Enlever les règles $M \to \epsilon$ si $M \neq S'$

3. Élimination des règles $M \to N$.
Calculer les paires (M, N) telles que $M \to^* N$
Pour chaque paire (M, N) calculée, Faire
Pour chaque règle $N \to u$ Faire
Ajouter la règle $M \to u$
Fin Faire
Fin Faire ;
Enlever toutes les règles $M \to N$

4. Suppression des non terminaux non productifs : Calculer les non-terminaux productifs ; Enlever tous les autres

5. Suppression des non terminaux non atteignables : Calculer les non-terminaux atteignables ; Enlever tous les autres

On remarque que chaque étape ne remet pas en cause la précédente, et donc la grammaire obtenue est propre. Ce ne serait pas le cas si l'on inversait les deux dernières étapes, comme le montre l'exemple $S \to aMN, M \to a$. M est atteignable, il ne sera pas enlevé par l'étape 5, mais S n'est pas productif, donc il est enlevé par l'étape 4, puis M n'est plus atteignable.

10.6 Decidabilité

- Lang(G) = vide? On nettoie, on regarde s'il reste qqc
- Lang(G) infini? On nettoie et on regarde s'il y a un cycle (un non-terminal X tel que $X \to \alpha X \beta$)
- Un mot u est-il dans Lang(G)? On met sous FNC et on utilise l'algo CYK (hors programme) ou bien si la grammaire est « raisonnable », on utilise l'analyse syntaxique.
- G est-elle ambiguë? Indécidable, hors programme.

11 Analyse lexicale

Lesétudiantsonttoustrèsbon

Dans le processus de compilation, l'analyse lexicale découpe le texte source en mots appelés des tokens:
Les étudiants sont tous très bons

De même que dans les langues naturelles, ce découpage en mots rend possible le travail de la phase suivante, l'analyse syntaxique.

Rôle des séparateurs. Le texte source est une suite de caractères. Les blancs (espace, retour chariot, tabulation, etc.) permettent de séparer deux tokens

Exemple: pour le source camel: fun $x \to x+1$
Où sont les frontières?

funx = un seul token (l'identificateur funx) et fun x = deux tokens (le mot-clé fun et l'identificateur x)

Les blancs ne sont pas toujours nécessaires (entre x, + et 1 par exemple). Les blancs n'apparaissent pas dans le flot de tokens renvoyé. Les commentaires jouent le rôle de blancs et sont aussi éliminés.

11.1 Notion de Token.

Un token comprend un numéro de classe qui correspond à un terminal de la grammaire spécifiant la syntaxe du programme. Un token a aussi une valeur. Certaines classes de token ont une seule valeur possible: mot clef, opérateur binaire .... D'autres classes en ont plusieurs, dans ce cas la valeur est calculée a partir de la sous chaîne associée au token. Exemple: pour la classe cte, la valeur est l'entier calculé à partir de la séquence de chiffres représentant la constante. Pour la chaîne "32" il faut traiter les caractères '3' et '2', et calculer $3 * 10 + 2$.

Comment spécifier les classes de token?
Pour chaque classe:

- On donne une expressions rationnelle
- L'analyseur génère l'automate associé.

Exemples, les états finaux sont resp. 3,1,1
le mot clef "fun" $0 - f \to 1 - u \to 2 - n \to 3$
constante entière 0 -chiffre $\to$1-chiffre $\to$ 1
identificateur 0-lettre$\to$1-lettre+chiffer$\to$ 1

Le gros automate unique. L'analyseur lexical construit un automate d'états fini qui fait la réunion de tous ces petits automates.

Fonction de l'analyseur

- Décomposer un mot (le source) en une suite de mots reconnus. Ambiguïté possible.
- Construire les tokens: les états finaux contiennent des actions pour calculer les valeurs des tokens.

Résolution de deux types d'ambiguïté.
Ambiguïté 1: le mot funx est reconnu par l'expression régulière des identificateurs, mais contient un préfixe reconnu par une autre expression régulière (fun)
$\Rightarrow$ on fait le choix de reconnaître le token le plus long possible

Ambiguïté 2: le mot fun est reconnu par l'expression régulière du mot-clé "fun" mais aussi par celle des identificateurs
$\Rightarrow$ On classe les tokens par ordre de priorité.

L'algorithme qui gère le gros automate unique. L'analyseur lexical mémorise le dernier état final rencontré. Lorsqu'il n'y a plus de transition possible, de deux choses l'une :

1. Aucun état final mémoire $\Rightarrow$ échec.
2. On a lu le préfixe w de l'entrée wv, avec w la chaîne reconnue par le dernier état final rencontré. $\Rightarrow$ on renvoie le token w, et l'analyse redémarre sur v concaténé au reste de l'entrée.

L'algorithme des fois, ne marche pas alors que il pourrait. Avec les trois expressions a, ab, bc, l'analyse lexicale échoue sur abc (ab est reconnu, comme plus long, puis échec sur c). Pourtant le mot abc appartient au langage $(a|ab|bc)*$

12 arbre de syntaxe abstraite

C'est une notion à la frontière de ce cours. C'est l'arbre de dérivation résumé: On « enlève » les non-terminaux tout en conservant la structure. On obtient un truc quatre fois plus petit en taille et beaucoup plus clair.

Par exemple, si on considère juste une expression dans un programme, un arbre de la syntaxe abstraite est un arbre dont les nœuds internes sont marqués par des opérateurs et dont

les feuilles (ou nœuds externes) représentent les opérandes de ces opérateurs. Autrement dit, généralement, une feuille est une variable ou une constante.

Un arbre de la syntaxe abstraite est utilisé par un analyseur syntaxique comme un intermédiaire entre un arbre de dérivation et une structure de données. Il est utilisé comme la représentation intermédiaire interne d'un programme informatique pendant que les types sont vérifiés, pendant qu'il est optimisé et à partir duquel la génération de code est effectuée.

— semaine 8 —

13 s8 Automates à pile

Cette notion paraît simple mais l'expérience montre que ce formalisme prend un certain temps à s'approprier.

13.1 Motivation

Un automate à pile est un automate d'état fini qui a une pile en plus, avec donc un ensemble d'états finis de symboles pouvant être dans la pile. La fonction de transition de l'automate, en plus de lire une lettre et de déterminer un nouvel état, peut lire le sommet de la pile (pour ce faire, on suppose qu'on le dépile) pour décider de sa transition, et peut empiler de nouveaux symboles de pile. Un automate à pile n'est clairement pas un automate d'état fini, car la pile est non bornée. Cependant, cet infini reste « gentil », du fait qu'on ne peut y accéder que par le sommet de la pile. On s'intéresse aux automates à pile, car ils représentent un bon compromis entre simplicité et puissance.

— Ils sont suffisamment puissants pour réaliser l'analyse syntaxique d'un programme
— Ils restent suffisamment simples pour pouvoir être générés automatiquement.

En plus de réaliser des transitions d'état, l'automate à pile doit gérer la pile en dépilant le sommet de pile, (lequel l'aide a décider quelle transition prendre) puis en rempilant qqc sur la pile. Donnez directement l'automate qui reconnaît $a^n.b^n$.

Pour étudier les automates à piles, 4 ou 5 états produisent déjà un comportement complexe et suffisent à couvrir un pannel représentatif des différents comportements possibles. Ce n'est que lorsqu'on les génère automatiquement, pour reconnaître un vrai langage de programmation, que ces automates vont devenir gros, avec plus de 10 états.

13.2 Langages reconnaissables par automates à pile

Definition 20 *Un automate à pile non déterministe A est un quadruplet* $(\Sigma, Q, \Gamma, \delta)$ *où*

1. Σ *est le vocabulaire de l'automate*
2. Q *est l'ensemble des états de l'automate ;*
3. Γ *est le vocabulaire de pile de l'automate ;*
4. $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to P(\Gamma^\star \times Q)$ *est la fonction de transition de l'automate.*

L'automate sera dit déterministe s'il vérifie qu'il n'y a jamais de choix possible. Les epsilon transitions peuvent être déterministe, à condition que le symbole de pile ne soit pas utilisé dans d'autres transitions de l'état concerné.

On notera $q - a, X, \alpha \to q'$ pour $(q', \alpha) \in \delta(q, a, X)$. Lexicographie les lettres a, b, c, . . . pour les lettres de Σ , les lettres u, v, w, . . . pour les mots sur Σ, les lettres X, Y, Z pour les lettres de Γ et les lettres $\alpha, \beta \ldots$ pour les mots de pile sur Γ

Donnez l'exemple de l'automate qui reconnaît $a^n b^n$

— semaine 8 —

Definition 21 *Étant donné un automate* $A = (\Sigma, Q, \Gamma, \delta)$ *On appelle configuration toute paire* (q, α) *formée d'un état* $q \in Q$ *de l'automate et d'un mot de pile* $\alpha \in \Gamma^\star$. *On appelle transition de l'automate, la relation entre configurations notées* $(q, \alpha) - a, X, \beta \to (q', \alpha')$, *où* $a \in \Sigma, X \in \Gamma, \beta \in \Sigma^\star$ *,telle que* $(i)(q', \beta) \in \delta(q, a, X), (ii)\alpha = \lambda X, et (iii)\alpha' = \lambda\beta$

Definition 22 *Étant donné un automate A* $= (\Sigma, Q, \Gamma, \delta)$, *On appelle calcul d'origine* $(q0 \in Q, \alpha 0 \in \Gamma^+$*,une suite de transitions* $(q_0, \alpha_0) - a_1, X_1, \beta 1 \to (q_1, \alpha_1) ... (q_{n-1}, \alpha_{n-1}) - a_n, X_n, \beta_n \to (qn, \alpha_n)$. *L'entier n est la longueur du calcul et* $a_0 a_1 ... a_n$ *est le mot lu par le calcul, En abrégé* $(q_0, \alpha_0) - a_0 ... a_n \to (q_n, \alpha_n)$

Plusieurs notions de reconnaissance, toutes équivalentes,

Definition 23 *On dit que le mot $u = u_1...u_n$ est reconnu par l'automate $A + q_0, F, \gamma_0$ où $\gamma_0 \in \Gamma$ est appelé fond de pile de l'automate, q_0 est son état initial et $F \subseteq Q$ est l'ensemble des états finaux, s'il existe un calcul $(q_0, \gamma_0) - u_1...u_n \rightarrow (q_n, \alpha_n)$ d'origine (q_0, γ_0) tel que :*
1. Reconnaissance par état final : $q_n \in F$;
2. Reconnaissance par pile vide : $\alpha_n = \epsilon$;

On note par Lang(A) le langage des mots reconnus par l'automate A. Notons l'importance du symbole de fond de pile : la première transition de l'automate nécessite la lecture d'un symbole dans la pile, qui doit donc être initialisée avec γ_0.

Theoreme 11 *Les modes de reconnaissance sont équivalents pour les automates non déterministes.*

Preuve: Soit A un automate à pile reconnaissant par état final. On construit un automate reconnaissant à la fois par état final et pile vide : il suffit pour cela d'ajouter de nouvelles transitions sur chaque état final de manière à vider la pile. Cette transformation ne préserve pas forcément le déterminisme (s'il y a des transitions qui partaient de l'état final). Soit maintenant A un automate à pile reconnaissant par pile vide. On construit un automate reconnaissant à la fois par état final et pile vide, en ajoutant un nouvel état f final, un nouveau symbole de fond de pile γ'_0, puis les transitions : qui commencent par empiler ce nouveau fond de pile dessous l'ancien, et ensuite l'utilise pour aller vers f
$(i) delta'(q'_0, \gamma_0) = (q_0, \gamma'_0\gamma_0)$;
$(ii) \forall q \in Q \forall a \in \Sigma \cup \{\epsilon\} \forall X \in \Gamma, \delta'(q, a, X) = \delta(q, a, X)$;
$(iii) \forall q \in Q, delta'(q, \gamma' 0) = (f, \epsilon)$.
Ces transformations conservent le déterminisme. Comme le déterminisme est conservé en passant de la reconnaissance par pile vide à celle par état final, la bonne notion de reconnaissance par un automate déterministe, c'est-à-dire celle qui qui autorise la plus grande classe de langages reconnus, est basée sur la reconnaissance par état final. On choisit la reconnaissance par état final pour les automates déterministes. (si on a déterministe + état final, on ne peut pas en déduire déterministe + pile vide)

13.3 Automates à pile et grammaires hors-contexte

Les langages reconnaissables par un automate à pile (possiblement non-déterministes) sont appelés algébriques. Un langage est hors contexte s'il est généré par une grammaire hors contexte.

Theoreme 12 *Un langage est hors contexte si et seulement s'il est algébrique.*

Preuve: on utilise la double inclusion.
1- Montrons que HorsContexte est inclus dans Algébrique.
Pour reconnaître le langage engendré par une grammaire hors contexte, L'idée est de construire un automate à pile non-déterministe qui calque le calcul fait par la grammaire :

Exemple pour la grammaire $S \rightarrow epsilon|aSb$ on a la dérivation $S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow aaabbb$. À l'étape aaSbb le préfixe du mot reconnu sera le Début du mot généré jusqu'au premier non-terminal, c'est-à-dire : aa , la pile contiendra le reste: c'est a dire Sbb. Pour se faire, juste dans cette preuve, on convient que la pile tombe à gauche, (au lieu de a droite comme d'habitude);
L'automate a un unique état , on l'appelle automate "Marguerite"
règle: (epsilon, S,aSb) (a,a,epsilon) (b,b,epsilon)
Au début, on dépile S on empile aSb
ensuite, on dépile a en lisant a
ensuite on dépile S on rempile aSb
ensuite a nouveau on dépile a en lisant a
ensuite on dépile S on rempile que d'al.
ensuite on depile les 2 b en lisant les 2 b.
Dessiner l'automate à pile correspondant.

Preuve cas général soit G = $(\Sigma_T, \Sigma_N, S, R)$, on construit un automate à pile A = $(\Sigma_T, Q = \{q\}, q, \Sigma_N \cup \Sigma_T, S, T)$ qui reconnaît $L(G)$ par pile vide. l'alphabet de pile contient non terminaux ET terminaux.

À toute règle de la forme $N \rightarrow \beta$, on fait correspondre la transition $q - \epsilon, N, \beta \rightarrow q$
Pour tout terminal a on utilise la règle $q - a, a, \epsilon \rightarrow q$ Une récurrence simple montre que les dérivations dans la grammaire correspondent très exactement aux calculs de l'automate. Plus précisément, la grammaire dérive $S \rightarrow^* uX\alpha \rightarrow^* m$ si et seulement si l'automate peut générer la configuration $q, \gamma_0, m \rightarrow^*$

$q, X\alpha, v$ et $m = u.v$ est un mot du langage de la grammaire.

2-Montrons que Algébrique est inclus dans Hors-contexte: C' est plus complexe, et sans intérêt pratique : nous l'omettrons.

— semaine 9 —

14 s9 Est il algébrique?

Objectif: Savoir démontrer si un langage est ou n'est pas algébrique.

14.1 Forme normale de Chomsky

On va l'appeler "FNC" dans la suite. La FNC va servir à démontrer une nouvelle version du théorème de pompe étendu pour les langages algébriques.

Theoreme 13 *Pour tout langage hors contexte L, il existe une grammaire propre G qui l'engendre dont toutes les règles sont de l'une des trois formes* $S \to \epsilon, P \to a, ouP \to MN$ *avec M, N différents de S. (c'est la FNC Forme Normale Chomsky)*

Preuve: Partant d'une grammaire hors-contexte propre $G = (\Sigma_T\ , \Sigma_N, S, R)$ on appique deux étapes:

1. On rajoute une copie de Σ_T à Σ_N (on notera A la copie de a), puis l'ensemble de règles $A \to a | a \in \Sigma$.
2. On remplace les symboles terminaux a figurant dans les membres droits de règles initiales par le non-terminal correspondant A, puis on remplace la règle $X \to X1...Xn$ pour $n > 2$ par $X \to X1YetY \to X2...Xn$ en ajoutant un nouveau non-terminal Y. On obtient ainsi une nouvelle règle, avec un membre droit ayant un non-terminal en moins. On itère cette opération qui rajoute à chaque fois une nouvelle règle, mais avec de moins en moins de non-terminaux en membre droit, jusqu'à ce qu'il n'y ait plus que deux non-terminaux en membre droit.

Exemple: notre fameuse grammaire $S \to \epsilon, S \to aSb$ qui génère $\{a^n b^n, n \geq 0\}$. On choisit celle-là car elle est très simple et suffit pour illustrer. Il faut commencer par lui appliquer l'algorithme de nettoyage, car elle n'est pas propre: en effet , l'axiome apparaît en partie droite. Celui-ci donne la grammaire propre $S' \to S|\epsilon, S \to aSb|ab$.

1. On introduit non-terminaux A et B, on obtient la grammaire $S' \to S|\epsilon, S \to ASB|AB, A \to a, B \to b$.
2. on remplace la règle $S \to ASB$ par $S \to AC$ et $C \to SB$

14.2 Pompage des algébriques

Expliquons sur $S \to aSb|\epsilon$ le principe du pompage. Ça a l'air un peu contradictoire, mais pour expliquer Le principe : on n'a pas besoin de la FNC (on en a besoin pour démontrer l'existence d'une borne, précisément). Toute réécriture avec suffisamment de pas va comporter deux fois le symbole S. Considérons par exemple l'arbre de dérivation de la réécriture $S \to aSb \to ab$. Le long du chemin central, le non-terminal S apparaît deux fois: une fois parce que c'est l'axiome, une autre fois parce qu'on utilise la règle qui est récursive, c'est-à-dire où S apparaît à la fois à gauche et à droite. S est donc réécrit deux fois: la première fois récursivement, et la deuxième fois non récursivement. Ben, l'étape récursive peut être répétée autant de fois qu'on veut avant l'étape non récursive qui termine. En faisant cela on va répéter le même mot généré a gauche et à droite de l'occurrence de S dans le membre droit, ce mot dans le cas présent est réduit à seule lettre: 'a' pour la gauche, et 'b' pour la droite. A chaque fois on rajoute à la fois un 'a' et un 'b' donc, on "pompe" sur deux endroits en même temps. Faudra couper en cinq morceaux: avant le premier lieu de pompage, le premier sous-mot pompé, le bout qui sépare les deux sous-mots pompé, le deuxième sous-mot pompé, et le mot après le deuxième sous-mot pompé.

Theoreme 14 *Si L est algébrique, alors il satisfait la propriété*
$\exists N \in N$ *tel que*
$\forall m \in L$ *vérifiant* $|m| \geq N$,
$\exists u, x, v, y, w \in \Sigma_T^\star$ *tel que* $m = uxvyw$ *et* $|xy| > 0$ *et* $|xvy| < N$
Et on peut pomper, c'est-à-dire $\forall i \geq 0$ *on a* $ux^i vy^i w \in L$.

Preuve: on choisit une grammaire $G = (\Sigma_T, \Sigma_N, S, R)$ en forme normale de Chomsky

qui engendre le langage L. On va d'abord démontrer un petit lemme simple qu'on va devoir réutiliser deux fois.

Lemme: Dans une grammaire Chomsky, si un arbre de dérivation à une hauteur h, alors le mot des feuilles est de longueur $2^h - 1$ au plus.

Preuve du lemme par récurrence.
Cas $h = 1$: on a une branche, vu que la grammaire est FNC, c'est forcément une dérivation $A \to a$. Il y a donc une seule feuille et on vérifie $2^h - 1 = 1$.
Cas $h+1$, on applique la formule sur chacun des deux sous-arbres dont la profondeur est h. Leur mot de feuilles, par hypothèse de récurrence, est de longueur inférieure à $2^h - 1$. Le mot total est donc bien inférieur à $2 * (2^h - 1) = 2^{h+1} - 1$.

Preuve pompe. Soit $N = 2^{|\Sigma_N|+1}$. Si un mot m a une longueur $\geq N$, alors d'après le Lemme précédent, son arbre de dérivation a une hauteur $\geq |\Sigma_N| + 1$. On choisit un sous-arbre de profondeur EXACTEMENT $\Sigma_N + 1$. Je mets en majuscules, car c'est un détail qui a son importance par la suite. Soit C un chemin dans cet arbre de taille $|\Sigma_N| + 1$. Il existe deux nœuds étiquetés par le même non-terminal A puisqu'il y a $|\Sigma_N|$ non terminaux, et que la longueur fait $|\Sigma_N| + 1$. On a une première dérivation $S \to uAw$; une deuxième $A \to xAy$; ensuite, le deuxième A se réécrit à son tour $A \to v$. On peut ici supposer sans perte de généralité que u, x, v, y, w ne contiennent que des terminaux (s'ils contiennent aussi des non-terminaux, on les réécrit jusqu'au bout en terminaux). Si on veut pomper k fois, il suffit d'intercaler $k-1$ réécritures récursives supplémentaires $A \to xAy$ avant de faire la réécriture non récursive $A \to v$.

On doit réappliquer le lemme encore une fois : Grâce au fait qu'on a choisi un sous-arbre de profondeur EXACTEMENT $|\Sigma_N| + 1$, le mot xvy qui est un sous mot des feuilles de ce sous arbre a une longueur inférieure ou égale à $2^{\Sigma_N+1} = N$, qui finit de démontrer la partie finalement importante du théorème qui nous permettra ensuite de l'utiliser : c'est la condition $|xvy| < N$ qui restreint le champ des possibles.

Comme pour les rationnels, on utilise la contraposée de la pompe pour montrer qu'un langage n'est pas algébrique.

Theoreme 15 *Contraposée. Si L satisfait la propriété :*
$\forall N \in N$
$\exists m \in L$ *vérifiant* $|m| \geq N$ *tel que,*
$\forall u, x, v, y, w \in \Sigma_T^\star$ *vérifiant* $m = uxvyw$ *et* $|xy| > 0$ *et* $|xvy| < N$
On peut pas pomper, c'est-à-dire $\exists k \geq 0$ *tel que on a* $ux^kvy^kw \notin L$.
Alors ça prouve que L n'est pas algébrique

Exemple: Démontrons que $a^nb^nc^n$ n'est pas algébrique. Soit N un entier quelconque
je choisis $u = a^Nb^Nc^N$
Soit une décomposition quelconque $u, x, v, y, w \in \Sigma_T^\star$ vérifiant $m = uxvyw$ et $|xy| > 0$ et $|xvy| < N$,
Comme $|xvy| < N$ il ne peut contenir à la fois a, b et c Je choisis donc $k = 2$. On a $ux^kvy^kw \notin L$ soit parce qu'on a pas rajouté des a, soit parce qu'on a pas rajouté des c, et on a rajouté ou bien des a, ou bien des b, ou bien des c.

14.3 Clôture

- Clos par union $S \to S1|S2$
- Clos par étoile $S \to S.S1|\epsilon$
- Pas clos par intersection!.
- Pas clos par complémentaire!!.
- Clos par intersection avec rationnel
- Cloture par morphisme, morphisme inverse (rappeler ce qu'est un morphisme)

Attention, l'intersection de deux langages n'est pas algébrique contre-exemple $a^nb^nc^m$ et $a^mb^nc^n$ intersectés donne $a^nb^nc^n$. Pour montrer que l'intersection avec un rationnel reste algébrique, on fait le même produit synchronisé qu'on a déjà vu dans le cadre des automates d'états fini. Ce produit ne marche pas pour l'intersection de deux langage algébriques, car on obtiendrait deux piles à gérer, et on ne peut gérer deux piles avec une seule pile; Le complémentaire d'un algébrique, n'est pas algébrique, sinon l'intersection le serait. En effet on peut exprimer l'intersection à partir de l'union et du complémentaire : $A \cap B$ = complémentaire (complémentaire $(A) \cup$ complémentaire (B))

— semaine 10 —

15 s10 Analyse Syntaxique Descendante

Soit G une grammaire. L'analyse syntaxique consiste à construire l'arbre de dérivation d'un mot w qui appartient à $L(G)$. Ce faisant, on

va aussi en même temps vérifier que oui, on a bien $w \in L(G)$. Lorsque la grammaire G a de bonnes propriétés, on peut automatiquement construire un automate à pile qui fera ce travail.

On commence par étudier l'analyse descendante qui est plus facile (mais moins performante) que l'ascendante : on construit l'arbre de dérivation du haut vers le bas, c'est-à-dire depuis la racine étiquetée par l'axiome, vers les feuilles. On expande toujours le non-terminal le plus à gauche En lisant le mot lettre par lettre, et en observant quelle est la prochaine lettre , on va pouvoir décider quel membre droit choisir lors de l'expansion du non-terminal le plus à gauche, pour construire l'arbre.

15.1 Grammaire facile.

Considérons la grammaire suivante: $G1$: $S \to aSbT|cT|d \quad T \to aT|bS|c$. Analysons le mot $w = accbbadbc$ À chaque étape de l'analyse descendante, on a un symbole non-terminal courant à dériver (sur l'arbre), et une lettre courante à engendrer (dans le mot à analyser) $G1$ est particulièrement adaptée à l'analyse descendante car le membre droit de chaque règle commence toujours par un terminal. De plus, pour tout couple (X, x) où X est un non-terminal et x un terminal, au plus une seule règle a son membre droit commençant par x, c'est donc celle qu'on va appliquer si le prochain caractère est x, pour expandre un non-terminal courant X.

15.2 Utilisation d'un automate à pile

On refait l'analyse précédente, mais cette fois en utilisant un automate à pile. Cela se fait en utilisant une table avec trois colonnes: d'abord la pile, à gauche ensuite le mot au centre (qui va être lu lettre par lettre) et finalement une colonne de droite "action" . Attention: pour que cela roule facilement, il faut faire **tomber la pile à gauche**, i.e. le sommet de pile se trouve à gauche. Il y a deux actions possibles pour notre automate à pile, en fonction de la nature du sommet de pile:

1. Si c'est un non-terminal X, il sera réécrit à l'étape suivante en choisissant le bon membre droit α en fonction de la prochaine lettre du mot appelée **caractère d'avance**. L'action se note expand($X \to \alpha$), elle consiste à dépiler X et empiler α
2. Si c'est un terminal x, alors cela doit être aussi la prochaine lettre du mot, sinon, le mot n'est pas reconnu. On la dépile, en même temps qu'on avance sur le mot. L'action se note shift(x)

15.3 Table de transition

Elle indique pour chaque non-terminal à expandre, vers quel membre droit expandre, en fonction du caractère d'avance. Il y a donc les nom-terminaux possibles en ordonnée, les terminaux en abscisse, et des membres droits dans les cases. L'automate doit être déterministe pour pouvoir être exploitable. Il sera déterministe s'il n'y a pas de choix; c.-à-d. si au plus un membre droit possible par case. Pour notre grammaire facile, c'est facile! On range dans la case (X,x) le membre droit de la règle réécrivant X, et qui commence par x. Dans le cas général, on peut avoir des membres droits qui:

1. commencent par des non-terminaux,
2. sont réduits à epsilon.

Il va falloir que l'on passe par des calculs pour savoir comment choisir le bon membre droit.

15.4 Grammaire difficulté moyenne

Ces grammaires ont donc des membres droits qui ne commencent pas par des terminaux. Par exemple, dans notre grammaire simple, on construit $G2$ en remplaçant partout dans les membres droits 'a' par 'A' et en ajoutant la règle $A \to a$. Le raisonnement pour réussir l'analyse est le même, tenant compte du fait qu'à partir de 'A' on ne génère que des mots qui commencent par 'a'. Dans le cas général, il faudra calculer les "premiers" d'un non terminal 'X' qui sont les terminaux (lettres minuscules) commençant un mot pouvant être dérivé depuis X. Considérons une règle $X \to \alpha$. La notion de premier se généralise aux à des mots m. Premier(m), ce sont les terminaux commençant un mot pouvant être dérivé depuis m. En particulier, on considère Premier(α). Dans la table, on va écrire α dont les cases de coordonnées (X, x) où $x \in$ Premier(α). Cela généralise ce qu'on faisait pour les grammaires simples.

15.5 Calcul des "premiers"

On définit un ensemble de lettres associées à un non-terminal X par:
premier$(X) = \{a \in \Sigma_T | \exists \beta \in \Sigma^\star, X \to a\beta\}$
Ces ensembles se calculent en résolvant un système d'équations qui s'écrit en regardant les règles qui réécrivent X, et en cherchant mentalement toutes les lettres qui commencent les membres droits, ou qui commencent les mots dérivables à partir de ces membres droits. Ça produit des équations i.e. les premiers de X sont définis à partir des premiers de $Y, Z \ldots$si ces membres droits commencent par des non-terminaux Y, Z

Le système d'équations s'écrit comme $X = f(X)$ où X est un vecteur d'ensemble de mots, et f est une fonction croissante sur ces vecteurs d'ensemble pour l'ordre suivant: un vecteur d'ensemble est plus petit qu'un autre si chacune de ses composantes est incluse dans la composante correspondante de l'autre vecteur. Le fait que f est croissante pour nos systèmes est en fait très simple: f est définie seulement à partir de l'union, et l'union conserve l'inclusion, si A est inclus dans A' et B est inclus dans B', alors A union B est inclus dans A' union B'. Il existe de plus un élément minimum qui est le vecteur d'ensemble vide. La résolution se fait en itérant la fonction f depuis ce minimum. Cette suite est croissante, et le nombre de valeurs étant fini, elle converge vers un point fixe de f. Être un point fixe, ça signifie être une solution du système $f(X) = X$. De plus, on prouve facilement que c'est le plus petit point fixe. Cela montre que le système a une solution minimale unique. Une fois calculés les premiers des non-terminaux, on peut facilement en déduire les premiers des membres droit w qui sont définis exactement pareil:

premier$(w) = \{a \in \Sigma_T | \exists \beta \in \Sigma^\star, w \to a\beta\}$

Appliquons donc la méthode au calcul des suivants pour la grammaire: $E \to E{+}F|F, F \to F + G|G, G \to id|cte|(E)$

On écrit la fonction f ; on montre qu'elle est croissante ; on résout l'équation par itération.

On obtient pour les premiers, premier(E) = premier(F); premier(F) = premier(G); premier(G) = { '(', id, cte } ; premier (S) = premier (E).
Pour écrire le système je note X pour premier(X), j'utilise le même symbole du non-terminal pour dénoter l'ensemble de mots qu'on cherche; $f(E, F, G, S) = (F, G, \{'(', id, cte\}, E)$
Pour résoudre On obtient donc la suite de vecteur d'ensembles:
(∅,∅,∅,∅)
(∅,∅,{ '(', id, cte },∅)
(∅,{ '(', id, cte } ,{ '(', id, cte },∅)
({ '(', id, cte },{ '(', id, cte } ,{ '(', id, cte },∅)
({ '(', id, cte },{ '(', id, cte } ,{ '(', id, cte },{ '(', id, cte })

15.6 Grammaire les plus difficiles

Ces grammaires ont des règles avec des réécritures $X \to \epsilon$. Le non-terminal X est dit « nullifiable ». Un mot en entiers, sans terminaux, peut aussi être nullifiable, si toutes ses lettres le sont. On considère la grammaire $G3 = S \to aSb|\epsilon$. ϵ n'a pas de premier alors que faire? On choisira d'expandre S vers ϵ, si le prochain caractère du mot peut "suivre" S i.e. pour les caractères appartenant à suivant(S). En effet, lorsqu'on choisit la règle $S \to \epsilon$,le non-terminal S « s'en va », et donc le caractère d'avance suivra donc S. Faire l'exemple, on comprend limpide ce qui se passe sur cette grammaire. Il faut donc calculer ce deuxième type d'ensemble de lettres, appelé « Suivant ».

15.7 Calcul des suivants

Pour les suivants de X, il faut regarder les occurrences de X dans les membres droits ; en général, on cherche à regarder encore une fois les occurrences dans les membres gauches et on se trompe. Les suivants sont les premiers des sous-mots qui suivent ces occurrences dans les membres DROIT. Et, de plus, si jamais ces occurrences se trouvent à la toute fin du membre droit, on va rajouter les suivantes du non-terminal réécrit dans cette règle,i.e. le membre gauche. Pour chaque non terminal X on obtient encore une fois, un système d'équations: Suivant $(X) = \{a \in \Sigma | S \to \alpha X a \beta\}$.

Ce sont littéralement les minuscules qui peuvent suivre une majuscule dans une dérivation depuis l'axiome. Trouver les équations pour les suivants est plus compliqué que pour les premiers, surtout en présence de non-terminaux "annulables", i.e. qui peuvent dériver ϵ: On exécute l'algorithme suivant qui

résume ce qu'on vient de dire, plus formellement: Rajouter la règle $S' \to S\#$ pour marquer la fin des mots ; cela rajoute # dans Suivant(S) ou S est l'axiome. Puis, pour chaque non-terminal X, pour chaque occurrence de X dans un membre droit des productions, il faut regarder ce qui suit X :

1. si c'est un terminal 'x', ajouter 'x' à Suivant(X)
2. si c'est un non-terminal Y, ajouter Premier (Y) à Suivant(X)
3. de plus, si Y est annulable, reprendre à partir de ce qui suit Y
4. si rien ne suit, ou (plus généralement) si tout ce qui suit est annulable, ajouter Suivant(Z) à Suivant(X), où Z est le membre gauche de la production.

Pour la grammaires des expressions arithmétiques, cet algorithme donne le système : Suivant(E) = { +, #, ')' }; suivant(F) = { + } union Suivant(E); suivant(G) = Suivant(F) ; Suivant(S) = #; Avec les même convention de notation que pour premier, le système s'écrit
f(E,F,G,S) = ({ +, #, ')' } , { + } union E), F, $\emptyset$)
et l'itération donne:
($\emptyset$,$\emptyset$,$\emptyset$,$\emptyset$)
({ +, #, ')' }, { + } ,$\emptyset$,$\emptyset$)
({ +, #, ')' },({ +, #, ')' } ,$\emptyset$,$\emptyset$)
({ +, #, ')' },({ +, #, ')' } ,{ +, #, ')' },$\emptyset$)

15.8 Utilisation d'un automate à pile, bis

Armé d'une table, on peut formaliser l'algorithme de reconnaissance :

Entrée Grammaire LL(1) $G = (N, \Sigma, P, S)$, table M, mot $a_1 \dots a_n\$$

Sortie accepter / erreur

1. Initialiser la pile $P \leftarrow [\$, S]$ et $a \leftarrow a_1$.
2. Tant que $\top(P) \neq \$$ faire :
   (a) $X \leftarrow \top(P)$.
   (b) Si $X \in \Sigma$ alors
      i. si $X = a$ alors dépiler X, avancer a ;
      ii. sinon, *erreur*.
   (c) Sinon, si $X \in N$ alors
      i. si $M[X, a] = X \to Y_1 \cdots Y_k$ existe, dépiler X puis empiler $Y_k, \dots, Y_1$ (sauf ε) ;
      ii. sinon, *erreur*.
3. Si $a = \$$ alors *accepter*, sinon *erreur*.

15.9 Grammaire LL(1)

On dit que une grammaire est $LL(k)$ si lire k caractères d'avance permet d'obtenir une table d'analyse déterministe, i.e. au plus un membre droit possible pour chaque case. Dans la pratique, on utilise toujours comme on l'a fait ici, un seul caractère d'avance, donc $k = 1$. Si une grammaire est LL(1), alors l'automate d'analyse descendante pourra fonctionner.

— semaine 11 —

16 s11 Analyse syntaxique ascendante

16.1 Analyse ascendante, Keskecé?

Avec l'analyse descendante, l'analyseur construit l'arbre de dérivation "en descendant" de la racine vers les feuilles. Au contraire, avec l'analyse ascendante, on construit l'arbre en partant des feuilles et en « remontant » vers la racine. Le programme qui construit l'arbre s'appelle un analyseur syntaxique, et il se présente comme un automate à pile, avec de multiples états, contrairement aux automates à pile simple que nous avons vus jusqu'à maintenant.

Caractère d'avance. De plus, cet automate d'analyse utilise un « caractère d'avance » pour décider de ses transitions. Le caractère d'avance désigne le prochain caractère du mot à lire. On dit caractère d'avance parce qu'on va le consulter, mais sans le « consommer », il reste toujours disponible pour les transitions suivantes. L'automate d'analyse se construit automatiquement à partir de la grammaire par un processus qu'on peut qualifier de compilation, et qui peut réussir ou échouer suivant la grammaire.

LL(1)/LR(1) Les grammaires qui compile vers un analyseur descendant s'appellent LL(1) le premier 'L' est mis pour "left" pour mot lus

de "gauche" à droite, et le deuxième 'L' est aussi mis pour "Left", pour dérivation "gauche ". Le chiffre 1 signifie un seul caractère d'avance. Les grammaires qui compilent vers un analyseur ascendant s'appellent LR(1) le 'R' veut dire "Right" pour dérivation "droite", le premier 'L' et le chiffre 1 ont la même sens que dans LL(1). Par extension, un langage est dit LL(1) (resp. LR(1)) si il peut être généré par une grammaire LL(1) (resp. LR(1)) Une grammaire qui peut se compiler vers un analyseur descendant peut toujours aussi se compiler vers un analyseur ascendant, mais pas le contraire. En d'autres mots, l'ensemble des langages LL(1) est strictement inclus dans l'ensemble des langages LR(1). l'analyse ascendante est donc plus puissante que l'analyse descendante. Elle est utilisée en vrai dans les compilateurs.

16.2 L'automate à deux états.

Considérons comme exemple fil conducteur, la grammaire simplissime $S \to aSb|\epsilon$.

Soit $G = (\Sigma_T, \Sigma_N, S, R)$ une grammaire quelconque. On augmente d'abord la grammaire avec la règle $S' \to S\#$. Cela permet de marquer la fin du mot avec le caractère # . L'automate d'analyse ascendante à deux états se construit ainsi: $A = (\Sigma_T \cup \{\#\}, \Sigma_N \cup \{S'\}, \gamma_0, \{1,2\}, \{2\}, T)$, où la fonction de transition T est définie par:

- shift : $1 - a, \epsilon, a \to 1$ pour tout a $\in \Sigma$.
- reduce : $1 - \epsilon, w, N \to 1$ pour toute règle $N \to w \in R$ (on dépile un mot w et on empile N)
- succès $1 - \epsilon, S\#, S' \to 2$

Ici, on s'autorise un formalisme de pile étendu ou il est possible de dépiler en une seule fois plusieurs caractères, ou bien aussi zéro caractère. L'appellation shift vient du fait qu'on déplace le prochain caractère depuis le mot, sur la pile. Reduce lui, se comprend comme diminuer la pile vu qu'on remplace le membre droit qui est dessus et qui est en général plus long qu'un caractère, par le membre gauche qui fait un caractère. L'automate ascendant à deux états avec le formalisme d'automate à pile, l'état 2 est final. Écrire cet automate pour notre exemple fil conducteur, en notant les actions reduce et shift. Faire tourner cet automate, pour analyser la chaîne *aabb*. Écrire la pile à gauche, le mot à droite, et une colonne pour les actions. En même temps que l'on fait tourner, faire constater qu'on construit effectivement l'arbre de dérivation en partant des feuilles, en réduisant dès que c'est possible. Et aussi, c'est bien une dérivation droite. Faire constater que le mot de pile correspond à la liste des racines des branches qu'on a déjà remontées. Avec les shift, on remonte sur une feuille, et avec les reduce, on remonte sur un nœud branche interne.

Notion de conflits. L'automate est non déterministe puisque la réduction $S \to \epsilon$ peut être faite à tout moment, et donc simultanément avec une lecture. On parle de conflit lecture/reduction. L'automate ne peut être déterministe qu'en l'absence de tels conflits. On peut résoudre ce conflit à la main, en choisissant de faire la réduction $S \to \epsilon$ seulement si le caractère d'avance est b et il y a un 'a' sur la pile ou si c'est # et la pile est réduite au fond de pile. Il y aussi un conflit réduction/réduction entre $S \to \epsilon$ et $S \to aSb$. On résout ce conflit en choisissant systématiquement $S \to aSb$.

Pour pouvoir mener l'analyse automatiquement, il faut Enlever le non-déterminisme automatiquement, c.-à-d. décider s'il faut lire ou réduire en cas de choix, et si on réduit, avec quelle règle. Pour enlever le non-déterminisme, il faut une règle qui précise quel choix faire. En d'autres termes, cette règle enlève les choix. Un choix est donc quelque chose que l'on cherche à éviter, on les appelle "conflit" au lieu de "choix". On dira "résoudre les conflits" au lieu de "enlever le non-déterminisme". Les lectures sont appelées « shift » Les conflits peuvent être entre un shift et un reduce, ou entre deux actions reduce; Leur résolution utilise deux techniques:

1. Avec l' automate LR(0), on montre comment les états peuvent préciser quelles sont les actions possibles entre shift ou reduce;
2. Avec les automates SLR(1) puis LR(1) on montre comment utiliser le caractère d'avance.

16.3 L'automate LR(0).

Comme on l'a dit, à chaque instant de l'analyse ascendante, le mot de pile correspond à la liste des racines des branches qu'on a déjà remontées. Plus précisément,

Definition 24 (Protophrase) *Une protophrase est une séquence de symboles terminaux*

et non terminaux qui peut apparaître en cours d'une dérivation du symbole initial S d'une grammaire G.
On parle de protophrase droite *(resp.* gauche*) lorsque cette séquence peut apparaître dans une dérivation droite (resp. gauche) de G.*

Pour une configuration d'un analyseur ascendant, la concaténation du mot sur la pile, avec le mot restant à lire est toujours une protophrase *droite* de G (si l'analyse se termine avec succès).

Definition 25 (Poignée) *Dans une protophrase ϕ, la séquence γ est une* poignée *pour la grammaire G si elle est le membre droit d'une production $X \to \gamma$, et que cette production doit être appliquée à ϕ pour construire la protophrase précédente, dans une dérivation droite á partir de S vers ϕ avec la grammaire G.*

Lors d'une analyse ascendante, on progresse en identifiant une poignée en haut de la pile, et l'operation de réduction consiste à remplacer cette poignée γ, par le non-terminal X

Definition 26 (Préfixe viable) *Une séquence γ est un* préfixe viable *pour une grammaire G si γ est un préfixe de $\alpha\beta$, où $\phi = \alpha\beta w$ est une protophrase droite de G est β est une poignée dans cette protophrase.*
Autrement dit, un préfixe viable est un préfixe γ d'une protophrase ϕ, mais qui ne s' étend pas plus à droite d'une poignée β de ϕ.

un préfixe viable peut toujours se compléter en une protophrase droite. En d'autre termes, il n'y a pas d'erreurs au cours de l'analyse tant que le mot de pile est un préfixe viable. Il s'agit donc de reconnaitre ces préfixes viables.

Analyseur LR L'analyseur LR comprends:
- une pile et un flot d'entrée
- une table d'analyse: qui décrit un automate à états finis augmenté avec des actions à effectuer sur la pile

L'exécution de l'automate est censée décaler sur la pile des symboles terminaux, jusqu'à atteindre une préfixe viable maximal (i.e. pas extensible à droite, i.e. contenant une poignée γ en sommet de pile, puis réduire la poignée en la remplaçant avec la partie droite X de la production $X \to \gamma$ concernée.

Fonctionnement d'un analyseur LR Sur un état d'analyseur $(\alpha \ , \ xw)$ le fonctionnement de l'analyseur LR est le suivant:
- exécuter l'automate à partir de l'état initial s_1 sur la pile α, ce qui nous laisse sur un état s_k
- exécuter l'action décrite dans la table d'analyse associée au symbole terminal x en entrée pour l'état s_k

  shift (noté s) déplacer le prochain caractère lu x sur la pile,

  reduce $X \to \gamma$ (notée r). Sur le sommet de la pile il y a la partie gauche γ de la règle. dépiler γ et empiler X

  accept (noté a) arrêter avec succès

  error (noté par case vide) signaler erreur
- recommencer avec l' état suivant

Comment produire une table d'analyse? Il faut savoir reconnaître les préfixes viables, et savoir déterminer quelles productions utiliser pour les réductions. Pour reconnaître les préfixes viables, on définit un automate d'états fini dont les états sont des "items". Une première notion simple d'item sont les items LR(0) qui sont simplement une régle avec un "point" quelque part dans le membre droit. Ce seront les états de notre premier automate qui s'appellera l'automate LR(0)

Definition 27 *Un* ITEM LR(0) *pour une grammaire G est une production $X \to \gamma$ de G plus une position j dans γ. Cela est noté, $X \to \alpha \cdot \beta$*

L'intuition est qu'on est dans l'état $A \to \alpha \cdot \beta$ si on a déjà vu en entrée le préfixe α d'une protophrase et que l'on attend de lire sur l'entrée une séquence dérivable à partir de β.

Si S est l'axiome de notre grammaire, on rajoute la règle $S' \to S\#$, le symbole $\#$ sert à marquer la fin du mot. L'état initial de l'automate LR(0) est l'item $S' \to .S\#$ on attend de lire sur l'entrée une séquence dérivable à partir de $S\#$.

Les transitions de l'automate LR(0)

Definition 28 (GOTO) *Supposons d'être dans un état $A \to \alpha \cdot X\beta$, pour un symbole* terminal ou non terminal *X: on a donc*

déjà vu en entrée le préfixe α et on attende une séquence dérivable à partir de $X\beta$. Si maintenant l'on reconnaît X, alors on a vu αX et on attende une séquence dérivable à partir de β.
C'est cela que capture la notion suivante de transition qu'on appelle un "GOTO", ou on déplace le point après X.

`Goto(`$A \to \alpha\cdot X\beta, X) = A \to \alpha X\cdot\beta$

$\epsilon-$transitions de l'automate LR(0) Si on est dans l'état $A \to \alpha \cdot X\beta$, i.e. on a déjà vu en entrée le préfixe α et on attend une séquence dérivable à partir de $X\beta$, on est aussi en condition d'attendre une séquence dérivable depuis X, suivie d'une séquence dérivable depuis β. Il faut donc ajouter une $\epsilon-$transition des états de la forme $A \to \alpha\cdot X\beta$ vers tout les états $\{(X \to \cdot\gamma)\}$ ou $X \to \gamma$ est réecriture possible de X dans la grammaire que l'on considère.

Calcul direct des $\epsilon-$Clotures On souhaite avoir un automate déterministe, on va directement calculer ce dernier, en prenant pour état non pas les items LR(0), mais les $\epsilon-$clôtures. Cela consiste à faire grossir chaque états en ajoutant dedans, de proche en proche, tout les état accessible par une suite de $\epsilon-$transitions. On obtient ainsi de nouveau états qui ne sont plus des items, mais des ensembles d'items.

Definition 29 ($\epsilon-$Clôture LR(0)) `Clôture(`I`) =`
`répéter tant que` I `grandit`
`pour tout item` $A \to \alpha\cdot X\beta$ `dans I`
`pour toute production` $X \to \gamma$
$I \leftarrow I \cup \{(X \to \cdot\gamma)\}$
`retourner` I

L'état initial de l'automate LR(0) déterminisé est la clôture de l'item $S' \to .S\#$. En général, il contient beaucoup d'items.

Premier résultat sur l'automate LR(0)

Theorem 16 *Le langage des mots de pile possible (ie. des préfixes viables) durant une analyse ascendante réussie est un langage régulier. Il est reconnu par l'automate LR(0) si on met tout les états finaux.*

Ce théoreme découle directement de la méthode suivie pour construire l'automate LR(0). Il n'est cependant pas évident de voir cela, c'est simplement parce que l'automate LR(0) ne fait que mimer des dérivations; ce théoreme sera donc simplement admis, sa preuve étant soit considérée comme complètement triviale, soit, au contraire, tout à fait obscure! On se limite à constater que cela fonctionne sur l'exemple; réécrire les états avec juste un numéro, en les mettant tous finaux; On trouve le langage de pile $S + S\# + a^\star + a^\star S + a^\star Sb$

Theorem 17 *Les états de l'automate LR(0) précise quand on peut lire un terminal, il faut un point avant ce terminal, et quand on peut réduire par une régle $A \to \beta$, il faut que l'état contienne l'item correspondant avec un point tout à la fin : $A \to \beta$.*

Ce théorème va nous permettre de décider des actions possibles, simplement en regardant les items contenus dans chaque état:

La construction de la table LR(0) Soit G une grammaire augmentée avec $S' \to S\#$, pour laquelle on a construit l'automate LR(0). La table d'analyse a une ligne par état, une colonne par symboles (terminal, ou non-terminal) que l'on remplit de la façon suivante:
Pour tout état $s_i \in \mathcal{I}$, dans la case s_i, u on mettra:

- $shift(s_j)$ si s_i contient un item avec un point avant le u i.e. $(A \to \beta_1 \cdot u\beta_2, v) \in s_i$ et s_j est le nouvel état après la transition, i.e. $s_j \in Goto(s_i, u)$
- *reduce* $A \to \beta$ si s_i contient un item avec un "." tout à la fin. En effet, si $A \to \beta\cdot, u) \in s_i$, on a alors le manche β sur la pile, et on peut réduire.
- *accept* dans la case $s_i, \$$ si $(S' \to S\$\cdot) \in s_i$
- sinon on laisse vide, ce qui signale erreur.

Note 1 : Les lectures de non-terminaux seront utilisées lorsqu'on fera des réductions dans l'analyse. Afin de les distinguer, on les regroupe tous dans une demi-table, et on les appelle "goto" au lieu de "shift".

Note 2: On remarque que dans l'automate LR(0) une réduction est décidée uniquement depuis l'état, c.-à-d. le prochain caractère à lire noté u ne joue aucun rôle. C'est cela qui limite beaucoup sa généralité.

Grammaire LR(0) Si chaque case de la table LR(0) contient au plus une action, alors

l'automate LR(0) indique une seule action possible, il est donc deterministe, et on pourra mener l'analyse ascendante. La grammaire est dite LR(0). Si par contre la table LR(0) donne le choix entre plusieurs actions, alors il y aura non-déterminisme que nous appellerons conflit. Il existe deux sortes de conflits: entre une réduction et une lecture ou entre deux réductions. Il ne peut exister de conflits entre deux lectures, puisque alors le caractère lu serai différent et cela résoudrai d'emblée le conflit.

Dessiner la table LR(0) pour l'exemple fil conducteur. On note que il y a toujours deux états ou il a le même conflit shift a / reduce $S \to \epsilon$ précedement évoqué. On devra donc considérer des automates plus puissants.

Le méta-automate Lorsqu'on fait une reduction $X \to \beta$ il faut repartir de l'état s de l'automate LR(0) qu'on avait juste avant de commencer à lire β, et aller dans le nouvel état $Goto(s, X)$. C'est donc a ce moment précis, lors de la mise en oeuvre des réductions, qu'entre en scéne la partie droite de la table d'analyse, appellée les goto. Pour facilement récupérer l'état s, la solution naturelle consiste à empiler les états aussi. De cette maniére, la pile sera une alternance de symbole de grammaire et d'état. Pour trouver s, il suffira de dépiler β ainsi que les symboles pris en sandwich dedans, puis de lire l'état se trouvant juste avant. Au final, l'analyse se fait via un "méta-automate" (un automate d'automate) qui manœuvre globalement la pile, et l'automate LR(0). Cet automate se formalise toujours comme un "brave petit" automate à pile, donc on ne sort pas du cadre, il est sympathique d'en être conscient. Voici l'algo suivit par le méta-automate.

- Pas besoin de colonnes pour l'état courant, il se trouve au sommet de pile.
- Pour faire un shift, on empile la lettre lue puis le nouvel état.
- L'état s empilé juste devant un mot β est utilisé lors de reduce ($X \to \beta$), pour savoir d'ou on repart.
- Le nouvel état est $s' = \text{GOTO}(s, X)$.

Faire fonctionner le méta automate pour analyser aabb.

— semaine 12 —

17 s12 Automates plus puissants que LR(0)

L'automate LR(0) n'est pas utilisé du tout en pratique, car pas assez puissant. C'est pédagogique de commencer par lui, car il est plus simple. Les automates plus puissants vont utiliser 1 caractère d'avance, pour cela on les appelle SLR(1), LR(1), LALR(1)

L'automate SLR(1) Sur les états où il y a un conflit shift/reduce, on essaie de résoudre les conflits de la façon suivante : on utilise le fait que le mot de pile concaténé avec le reste du mot lu, est une protophrase, i.e. une étape dans la dérivation droite. On en déduit que pour pouvoir réduire par $X \to \beta$, le caractère d'avance doit appartenir à suivant(X).

On calcule donc le suivant du non-terminal vers lequel on réduit, et on va réduire seulement si le caractère d'avance appartient au suivant du non-terminal « vers-lequel-on-réduit ». Cette méthode peut aussi résoudre les conflits reduce/reduce qui réduisent vers des non-terminaux distincts. L'automate avec les réductions plus ciblées, s'appele "l'automate SLR(1)". La table SLR(1) est similaire à la table LR(0), la différence est que cette fois ci, on tient compte du caractère d'avance u pour indiquer une réduction. Sur notre exemple, on constate que cela marche parce-que comme suivant(S)=$\{b, \#\}$, ne contient pas la lettre 'a', il n'y aura plus de reduce pour la colonne de 'a'. On dira que la grammaire est "SLR(1)" si tous les conflits sont levés.

L'automate LR(1) On peut faire mieux que SLR(1). On améliore en ajoutant une lettre terminale aux items LR(0), on obtient des nouveaux items, appelés "item LR(1)". Ce caractère est appelé "caractère de retour". Il précise quels sont les suivants possible du membre gauche de l'item, en fonction de l'état spécifique de l'automate qui cette fois, utilise tout l' "historique de dérivation" connu. Ce faisant, il prédit avec plus de précision, les caractère d'avance possible. Pour réaliser une réduction, le caractère d'avance doit être aussi un caractère de retour. L'état initial est l'item $S' \to .S, \#$.

L'intuition de l'état $(A \to \alpha \cdot \beta, z)$ estque l'on a déjà vu en entrée le préfixe α d'une protophrase et que l'on attende sur l'entrée une

séquence dérivable à partir de βz.

Epsilon transitions et clôture LR(1) Si on a $(A \rightarrow \alpha \cdot X\beta, z)$, i.e. on a a déjà vu en entrée le préfixe α et on attende une séquence dérivable à partir de $X\beta z$, on est aussi en condition d'attendre une séquence dérivable depuis X, suivie d'une séquence dérivable depuis βz. Cela défini donc des nouvelles epsilons transitions entre item LR(1) (tenant compte des premiéres lettres possibles qui suivent le nom terminal vers lequel on réduit), et une nouvelle facon de calculer les clôtures correspondantes:

Definition 30 (Cloture LR(1) d'un état I) *Pour tout item* $(N \rightarrow \alpha.X\beta, b) \in I$, *pour toute règle* $X \rightarrow \gamma$ *et tout terminal* $a \in Premier(\beta b)$, *alors* $(X \rightarrow .\gamma, a) \in I$.

Action LR(1): Une réduction $N \rightarrow \gamma$ ne pourra être effectuée dans un état q en présence du caractère d'avance a, qu'à la condition que q contienne l'item LR(1) terminal $(N \rightarrow \gamma., a)$. C'est plus précis que SLR(1), car a peut appartenir à suivant de N mais ne pas être caractère de retour;

Exemple, faire l'automate LR(1) de la grammaire S $\rightarrow$ aSb,$|\epsilon$, montrer que au tout début, celui ci restreint les suivants à juste # au lieu de $b, \#$ préconisé par l'automate SLR(1), en effet si on veut générer juste ϵ, le caractère d'avance sera bien #. Notes: Pour compacter la représentation des états obtenus par clôture, on regroupe ensemble les items LR(1) ayant la même partie LR(0), en écrivant l'item LR(0) suivi de l'ensemble des lettres minuscules regroupées.

L'automate LALR(1). L'automate LR(1) est lourd, il contient beaucoup d'états en pratique. On réduit le nombre d'états en fusionnant ensemble les états de même partie LR(0), c.-à-d. qui contiennent des items identiques, au caractère de retour près. C'est l'automate que construit Yacc, et qui est utilisé en vrai par les compilateurs. Seules des grammaires très bizarres sont LR(1) et pas LALR(1), donc cela ne pose pas de problème en pratique. (voir exemple donné à la section suivante.)

17.1 Différence LALR(1)-LR(1)

On va montrer que la grammaire suivante et LR(1) mais pas LALR(1).

S	$\rightarrow$ (X	X	$\rightarrow$ F]
S	$\rightarrow$ E]	E	$\rightarrow$ A
S	$\rightarrow$ F)	F	$\rightarrow$ A
X	$\rightarrow$ E)	A	$\rightarrow \epsilon$

La table de transitions permet de voir simplement que cette grammaire est LL(1). Elle est donc a fortiori LR(1) (résultat du cours non démontré). Pour savoir si elle est LALR(1) on commence à construire l'automate LR(1) et on fusionne les nouveaux états crées par transitions, au fur et à mesure que c'est possible. Cet algorithme n'est pas si compliqué, mais tout de même, l'expérience montre que c'est l'un des points les plus difficile à capter pour vous les étudiants. L' automate contient deux états : l'un formé des items $\{[FA.,]], [EA.,)]\}$ (transition par '(' puis par A depuis l'état initial) l'autre des items $\{[FA.,)], [EA.,]]\}$.(transition par '(' puis par A depuis l'état initial)

Dans l'automate LALR(1) ces deux états ont même structure LR(0) et sont confondus en le même état $\{[FA.,)]], [EA.,])]\}$ qui est conflictuel, puisque il donne le choix entre deux réductions possible.

Analyseurs générés avec YACC En même temps qu'on reconnaît, on construit l'arbre de syntaxe abstraite qui est une version résumée de l'arbre de dérivation, suffisant pour compiler. Exemple $x * y + 3$ donne l'arbre binop +(binop *(id x)(id y))(cte 3)

Metode cannonique pour les expression arithmetiques : On ajoute des règles de précédence ou d'associativité permettant de désambiguïser automatiquement lors du calcul de l'automate LALR (sans transformer la grammaire a priori). Les générateurs d'analyseurs d'aujourd'hui permettent ce type de désambiguïsation .