université
PARIS-SACLAY

Cours de Langage Formel 2025-2026 Frédéric Gruau

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`. Pour les machines de Turing, j'ai intégré quelque notes du cours d'informatique théorique de Laurent Rosaz. 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 Deroulement, synchro sur TD

Le cours est fait au tableau, cela est plus vivant, et plus interactif. Ce support ne comprends 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 également structuré par rapport aux examens. Chaque cours fait l'objet d'un TD, et d'un exercice au partiel puis à l'examen.

Le déroulement pour chacune des 12 semaines est le suivant : Il y à 12 cours(1h30) suivi de 12 TD(2h).

a- Rappel

1. Cours : Panoramique, Langage formels, Expression rationnelle, lemme d'Arden, def. automate d'état fini.
   TD : Egalité langage, Expression rationnelle, Automates simples.
2. Cours : Automate non-déterministe, epsilon transitions, théorème de Kleene
   TD : Automates, suite et fin, détermination, résolution d'équation (début).

b- Approfondissement automates

3. Cours : minimisation d'un automate
   TD : résolution d'équation (autre exemple), construction d'automates, construction directe de l'automate minimal a partir du langage.
4. Cours pompage, clôture, décidabilité.
   TD pompage, clôture.

c- Grammaires Hors Contexte

5. Cours : grammaires hors contexte, arbre de dérivation, ambiguité, réécriture droite.
   TD : grammaire d'un langage, langage d'une grammaire, désambiguïser.
6. Cours : nettoyage de grammaire, FN Chomsky, décidabilité, cloture, analyse lexicale.
   TD : analyse lexicale, grammaire d'un vrai langage.

d- Automates à piles

7. Cours pompe algébrique, preuve, l'automate à piles qui reconnaît $a^n b^n$. TD : Automate a pile, q1, q2 est-il-algébrique, début.
8. Cours : automate à pile, toutes les définitions, Équivalence -grammaire, premier et suivant.
   TD : est-il-algébrique automate à pile (fin), premier et suivant.

e- Analyse Ascendente

9. Cours Analyse ascendente à la main, Analyse ascendente LR(0), SLR(1) sur exemple TD Analyse ascendante à la main, premier et suivant exemple simple
10. Cours : SLR(1), définition, automate LR(1) général, LALR(1),
    TD : analyse ascendante suite et fin.

f- Machine de Turing

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.

11. Cours et TD machine de Turing en TD on fait les trois premier exo, on débriefe le 4ème qui est à finir chez soi.
12. cours et TD décidabilité.

0.2 Prérequis

Les étudiants qui n'ont pas encore étudié les automates d'états finis ou les expressions rationnelles, ont un retard important sur les autres. Ce retard doit être comblé par un travail supplémentaire très significatif durant les deux premières semaines ou ces deux notions seront reprises rapidement. Ces étudiants retiendront l'attention du tdman qui s'en occupera spécialement en les envoyant au tableau systématiquement.

0.3 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). On considérera de plus un temps et un espace mémoire raisonnable (théorie de la complexité) .

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 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^{ieme} 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, Σ^* est un monoïde libre sur Σ : 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 de 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 $P(v)$.

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

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 au 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. $\{aa, a\}.\{ab, aab\} = \{aab, aaab, aaaab\}$.
- 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 de Kleene permet de passer a 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. 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.

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 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 $A.L + B$ dans le membre droit, on obtient $L = A^nL + A^{n-1}B + A^{n-2}B + ... + B$ 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 $L = A^{l+1}L + A^lB + \ldots + A^{l-1}B + A^{l-2}B + \ldots + B$. Si A ne contient pas epsilon, alors les mot de $A^{l+1}.L$ sont de longueur $> l$. Donc u n'est pas dans $A^{l+1}.L$. Donc il est dans le reste qui est une partie de A^*B.

2 Expressions rationnelles

On fait l'usage systématique de l'union, produit et étoile de Kleene. On considère les mots écrit sur Σ, complété des signes '+' (union aussi noté |) '.' '*', '(',')'

le langage Rat des expressions rationnelles sur l'alphabet Σ est définit par induction :

— 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.

Toute expression rationnelle dénote un langage dit rationnel, 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
  — Lang$(e_1 + e_2)$ = Lang$(e_1)\cup$Lang$(e_2$)
  — Lang$(e_1.e_2)$ = Lang(e_1) . Lang$(e_2$)
  — Lang$(e^*) = (\text{Lang}(e))^*)$.
  — Lang$((e))$ = Lang(e).

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

Definition par induction On peut définir des fonction booléennes sur des expressions rationnelles par inductions, par exemple pour savoir si le langage décrit contient au moins un mot non vide, la fonction booléenne "contient" sera définie par

— contient(ϵ)= faux
— pour tout lettre l contient (l)= vrai
— pour tout exressions e_1, e_2
  — contient(e_1 + e_2)=contient(e_1) ou contient(e_2)
  — contient($e_1.e_2$)=(contient(e_1) et nonVide(e_2)) ou (contient(e_2)et nonVide(e_1))
  — contient(e_1^*)) = contient (e_1)

Identités remarquables : Deux expressions rationnelles distinctes peuvent dénoter le même langage. Exemple : $(a + b)^*$ et $(a^*b^*)^*$ dénotent toutes deux le langage des mots quelconques sur l'alphabet $\{a, b\}$.

1.$r + s = s + r$
2.$(r + s) + t = r + (s + t)$
3.$(rs)t = r(st)$
$ExamLF2019.tex4.r(s + t) = rs + rt$
5.$(r + s)t = rt + st$
6.$\emptyset^* = \epsilon$
7.$(r^*)^* = r^*$
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 physique, des circuit digitaux...

3.1 Automates déterministes

Definition 4 *Un automate fini déterministe A est un triplet (Σ, Q, δ) où*
1. Σ est le vocabulaire 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 a trois états qui reconnaît les entier en binaire. $q_0 - 1 -> q_1 - 0, 1 -> q_1; q_0 - 0 -> q_2; q_1, q_2$ finaux

Definition 5 *Etant 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 $q_0 - u_1 -> q_1 -> ... - u_n -> q_n$. on écrit $q_0 - u -> q_n$*

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 bloqué 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 :

Theorem 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 $q - u -> q'$*

On peut alors étendre l'application δ aux mots, en posant $\delta(q, u) = q' \in Q$ tel que $q - u -> q'$. Un automate qui n'est pas complet, le devient si on ajoute un nouvel état appelé poubelle vers lequel vont toutes les transitions manquantes. 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 $F \subseteq Q$;*

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. $\delta : Q \times \Sigma \to P(Q)$ la fonction de transition.

Exemple : L'Automate des mots qui contiennent "aa" : $q_0 - a, b -> q_0 -> a -> q_1 - a -> q_2 - a, b -> q_2; q_2$ 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 et d'autres pas, si un calcul échoue, cela veut rien dire, il faut tout explorer. Reconnaissance = au moins un calcul démarrant sur l'état initial, arrive sur 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 $P(Q)$, 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 etats finaux sont $\{K \in P(Q) | K \cap F \neq \emptyset\}$*

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

Exercice : Déterminiser l'automate qui reconnaît les mot 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, éventuellement. Principe de la preuve de déterminisation : Les états sont étiquetés par l'ensemble des noms des états de l'automate non-déterministe qui le constituent. S'il est possible dans l'automate non-déterministe d'atteindre les états $q_1, \ldots, q_n$ depuis l'état q en lisant la lettre a, alors, dans l'automate déterminisé, depuis tout état contenant l'état q, en lisant cette même lettre a, on atteindra un état contenant $q_1, \ldots, q_n$

Les automates déterministes, et non déterministes reconnaissent les même langages. les automates déterministes sont parfois exponentiellement plus gros, la borne étant atteinte, c'est montré dans le TD.

3.3 Automates avec transitions vides

C'est une autre façon d'exprimer le non-déterminisme, ces automates sont aussi appelés asynchrone. Les transitions étiquetées par ϵ, dénotent l'absence de lettres lue lors de la transition.

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", ou une occurrence du mot "11.
$q_0 - 0, 1 - > q_0 - \epsilon - > q_1, q_2$.
$q_1 - 0 - > q'_3 - 0 - > q_3 - \epsilon - > q_5$
$q_2 - 1 - > q'_4 - 1 - > q_4 - \epsilon - > q_5$
$q_5 - 0, 1 - > q_5$
Morale : avec des epsilon-transitions, on construit 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 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'_4 lorqu'on dés-epsilonne.

Formulation avec les epsilon cloture. La notion de epsilon cloture permet une autre formulation sans doute plus claire :

1. On calcule les epsilon clotures de chaque états q : La cloture de q est l' ensemble des états que l'on peut atteindre depuis q en lisant epsilon. On peut possiblement lire plusieurs epsilon de suite, pour cette raison, on appelle cela une "cloture transitive".
2. l'état initial est la cloture de q_0, les étas finaux sont ceux dont la cloture contient un final
3. une transition de q_1 vers q_2 par la lettre x conduit a ajouter la même transition mais de cloture(q_1) vers cloture(q_2)

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. A chaque état $q_i, i = 0..n-1$, on associe un langage L_i appellé futur de q_i, qui contient les mot 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 mot 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 $L_0 = (ab^*a + b)^*$

5 Minimisation d'un automate déterministe

5.1 Exemple fil conducteur

Soit le langage $L = (a+b)*aba(a+b)*$. L'automate reconnaissant ce langage est $q_0 - b- > q_0 - a- > q_1 - a- > q_1 - b- > q_2 - a- > q_3 - a, b- > q_3$ et la transition $q_2 - b- > q0$. Cet automate est minimal déterministe complet, on rajoute des états pour le rendre non minimal : on dé-triple q_1 en rajoutant $q_1 - a- > q_1' - a- > q$"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 ont tous même "futur", ils attendent les même mots et devront donc être fusionné.

Definition 10 *Le futur d'un état est l'ensemble des mots qui mènent de cet état à un final.*

Futur$(q) = \{m/\delta(q, m) \in F\}$. Le futur de q_0 est le langage reconnu. Les futurs ont déjà été utilisé pour poser un système de n équations a n inconnues permettant de calculer le langage reconnu par un automate donné, avec Arden.

Pour notre automate : dire les futurs :
futur0= L,
futur1=ba(a+b)* + L
futur2=a (a+b)* + L,
futur3=(a+b)*

5.2 Minimisation d'automate.

Idée clef : les états ayant même futur peuvent être fusionnés.

On va donc calculer la relation d'équivalence $\sim$ telle que $q \sim q'$ si q, q' ont même futur. On calcule progressivement en considérant $\sim_k$ qui est vrai pour "ont même futur avec seulement les mots de longueur $\leq k$. On augmente progressivement k en partant de zéro. Pour $k = 0$, les classes d'equivalence sont les finaux pour qui epsilon à un futur, et les non-finaux dont le futur est vide. Pour passer de $\sim_k$ à $\sim_{k+1}$, on remarque que $\sim_{k+1}$ est plus fine que $\sim_k$ donc les classes de $\sim_{k+1}$ s'obtiennent en subdivisant les classes de $\sim_k$. Deux mot de $k+1$ lettre ont même futur, si aprés la lecture d'une lettre, on tombe dans un ensemble d'états qui ont même futur de k lettres, c'est à dire dans

une classe de $\sim_k$; On considére 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 ne peut subdiviser a l'infini, donc au bout d'un moment on aura $\sim_{k+1}=\sim_k$ Sur l'exemple, on sépare d'abord $q_3, q'3$, puis $q_2, q'2$ etc. Au bout d'un moment l'algorithme converge, car on ne peut pas éternellement scinder. A ce moment la, on peut fusionner ensemble les états de chaque sous-ensemble de la partition. Cela va donner un automate i.e. l'image par δ sera cohérente. Montrez sur l'exemple, que si on fusionne avant convergence, l'image par δ n'est pas cohérente, on n'obtient pas un automate.

5.3 Calcul direct du minimal.

La notion de futur, peut se définir sur des états, mais aussi directement sur des mots : Le futur d'un mot w est l'ensemble des mots u qui si on le rajoute après w, permet d'obtenir un mot du langage

Definition 11 *Soit L un langage, w un mot. $futur_L(w) = \{u/wu \in L\}$*

Nous avons donc deux notion de futur : futur d'un état et futur d'un mot. Ces deux notions sont liées. Pour tout automate qui reconnaît le langage : Le futur d'un mot, est le futur de l'état vers lequel ce mot mène. Futur(m) =Futur$\delta(q_0, m)$ Vice versa,le futur d'un état et le futur de n'importe quel mot qui y mène. Lexique : on appelle aussi cela "résidu".

Definition 12 *$u \sim_L v$ iff u, v ont même futur dans L*

Property 1 *$\sim_L$ est une relation d'équivalence.*

Rappel de ce qu'est une relation d'équivalences : trois propriétés : reflexivité,symetrie est transitivité.

ex1 : $x \sim y$ si x et y on même plafond (nombre entier au dessus), quelles sont les classes ?
ex2 : $x \sim y$ si $x = y$ modulo 2, puis modulo 4. Quelles sont les classes ?

Une relation d'equivalence est plus fine qu'une autre, si elle est impliquée par cette autre : mosulo 4 est plus fine que modulo 2, et cela implique que les classes de modulo 4 subdivise les classes de modulo 2.

donner les 4 classes d'équivalence pour $\sim_L$
(a+b)*bb \ L ; futur= L ,
(a+b)*a \ L ; futur=ba(a+b)* +L
(a+b)*ab \ L ; futur=a (a+b)* +L
L ; futur=(a+b)*

Remarque : Les états d'un automate déterministe partitionnent les mots, car un mot ne peut mener que vers un seul état. Idée clef : Dans le minimisé, tout les mots qui ont même futur devrait mener vers le même état ; "Tout les mots qui ont même futur" c'est une classe d'équivalence de $\sim_L$. Donc, il y a une correspondance bi-univoque entre les classe de $\sim_L$ et les états du minimisé.

Pour obtenir directement les états du minimisé à partir du langage, Il suffit de prendre pour les étas, les classes d'équivalence de $\sim_L$.

5.4 Existence et unicité d'un automate minimal

Theoreme 5 *(Myhill-Nerode) Si L est rationnel, alors $\sim_L$ à un nombre fini de classe, soit A_L l'automate défini par*
1- les états sont les classes de $\sim_L$, on note $[u]$ la classe de u.
2- L'état initial est $[\epsilon]$
3- $\delta([u], a) = [ua]$ est bien définie
4- Les états finaux sont ceux dont le futur contient ϵ.
A_L reconnaît L avec le moins d'états possible.

Preuve simplifiée :
Step1 δ est bien défini, il faut montrer que la transition ne dépends pas du représentant choisi dans la classe de u car $\sim_L$ est une congruence droite : si $u \sim_L v$ alors $ua \sim_L va$ pour toute lettre a.

Cela vient du fait que si deux mots u, v on même futur F, alors ua et va aussi ; car ce sont les mots de F qui commencent par a, moins ce premier a.

Step2 Prouvons que A_L est minimal. Soit A un autre automate qui reconnnait L et la relation$\sim_A$ définie par $u \sim_A v$ si u, v mènent vers le même état de A. Il y a une correspondance bijective entre les classes de $\sim_A$ et les états de A. Idée clef : si deux mots mènent vers le même état de A, alors ils ont forcément même

futur. Cela s'écrit $u \sim_A v \Rightarrow u \sim_L v$. Cela corresponds a la définition de être plus fin : $\sim_A$ est plus fine que $\sim_L$; Reparler de l'exemple de la relation "est egal modulo 4", qui est plus fine que "est egal modulo 2". Cela implique que les classe de "est égal modulo 2" s'obtiennent en fusionnant les classes de "est égal modulo 4". De même, $\sim_A$ est plus fine que $\sim_L$ implique que les états de A_L s'obtiennent en fusionnant des états de A. Comme A a été pris quelconque, cela montre bien que A_L à moins d'états qu'un automate quelconque, et il est donc minimal en nombre d'états.

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 simple : 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 $\phi(u.v) = \phi(u).\phi(v)$

Les clôtures ont deux utilités :
1- montrer que certains langages sont bien reconnaissables,
2- montrer que certains langages ne sont pas reconnaissables, (contraposée pompe, ou démonstration par l'absurde). Traiter l'exemple nombre de 'a' egale au nombre de 'b'.

7 Pompage des langages reconnaissables

Le langage $L = \{a^n b^n, n \in N\}$ n'est pas reconnaissable. Par l'absurde, si il l'était, soit un automate $A = (\Sigma, Q, \delta, q_0, F)$ qui le reconnaît. Considérons l'application $\phi : n \mapsto \delta(q0, a^n)$. 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 $\delta(q_0, a^n b^n) = \delta(q_0, a^m b^n)$ absurde, l'un appartient a 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 $c = q_0 \to q_{|m|} \in F$ 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 : $c = q_0 - u \to q_i - v \to q_j - w \to q_{|m|} \in F$ et l'on vérifie immédiatement les conditions $m = uvw, v \neq \epsilon, |uv| \leq N$. De plus, comme $q_i = q_j$, le calcul $i = q_0 - u \to q_i - v^k \to q_j - \to q_{|m|} \in F$ 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". On ne peut donc pas déduire d'un langage qu'il est reconnaissable en montrant qu'il satisfait le "Lemme de la pompe". Mais l'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 pous-

ser les négations à l'intérieur des formules en changeant les quantificateurs,

¬ 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 transitions vides vers un automate non-déterministe sans transitions vides en temps linéaire en la taille, sans rajouter d'états ⇒ 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 13 *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 $q_0 - w \to q$; productif, s'il existe w tel que $q - w \to f \in F$; utile, s'il est à la fois accessible et productif.*

Un automate est dit réduit si tous ses états sont utiles. Nettoyer = enlever les état 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 $q' \in \delta(q, a)$. ⇒ 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 précédent 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.

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

9 Décision et temps lineaire.

Nettoyer sert à montrer que certain problèmes sont décidables et de plus rapidement. Un problème est décidable, si il se présente sous la forme d'une question oui/non, et que il existe un programme qui réponds oui ou non, au bout d'un temps fini. Rapide signifie ici en temps linéaire, proportionnel à la taille du problème Un problème deux fois plus gros prends un temps deux fois plus gros.

Décision du vide Le langage reconnu est vide ssi l'automate nettoyé n'a plus d'état ormis l'état initial.

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

Décision du plein Le langage reconnu est dit plein, si il contient tout les mots possible. 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.

Theoreme 9 *Le plein du langage reconnu par un automate déterministe (resp. non-déterministe) est décidable en temps linéaire (resp. exponentiel).*

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.

10 Grammaire.

- Décrit des langages
- Inspirés des grammaires de langage naturel, exemple : Phrase → sujet verbe complément.
- Différence : une grammaire peut générer des mots arbitrairement long, les phrases restent de longueur raisonnable.

Definition 14 *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.

Exemple1 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 verticale 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 engendre les mots du langage, en récrivant un mot $u \in \Sigma^\star$ en un nouveau mot $v \in \Sigma^\star$. On remplace une occurrence (quelconque) d'un membre gauche de règle présent dans u par le membre droit de cette règle.

Definition 15 *Étant donnée une grammaire $G = (\Sigma_T , \Sigma_N , S, R)$, 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 16 *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 : 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.

Exemple3 (Grammaires hors-contexte) 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 représente des constantes ou des identificateurs. Pour l'analyse syntaxique, on considère toutes les constantes (resp. tout les identificateurs) comme le même terminal. Faire deux dérivation 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 17 *Étant donnée une grammaire $G = (\Sigma_T , \Sigma_N , S, R)$, 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 différent. On n'aime pas, on va définir l'ambiguïté comme suit, et chercher ensuite a l'éviter.

Definition 18 *Une grammaire G est ambiguë s'il existe un mot $w \in Lang(G)$ 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 dérivations. 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'autre 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 19 *Réécriture droite (resp gauche) on réécrit le plus a droite (resp. gauche)*

La réécriture droite (resp. gauche) corresponds a un parcours droit (resp. gauche) de l'arbre de dérivation. On parle aussi de dérivation droite et dérivation gauche. Il y a une correspondance bi-univoque 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).

10.5 Nettoyage de grammaire.

Definition 20 *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 ateignable.

10.6 Decidabilité

- Lang(G) = vide ? on nettoie on regarde si il reste qqc
- Lang(G) infini ? on nettoie et on regarde si 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 vu en PIL.
- G est elle ambiguë ? indécidable, on fera la démo en TD, mais oui.

11 Analyse lexicale

Lesétudiantsonttoustrèsbon

l'analyse lexicale decoupele texte source en mots appeles des tokens :
Les étudiant sont tous très bon

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

Role 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 $x \to x + 1$ ou sont les frontières ?

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

Les blanc ne sont pas toujours nécessaire (entre x, + et 1 par exemple). Les blancs n'apparaissent pas dans le flot de tokens renvoyé.

11.1 Notion de Token.

Un token comprends un numéro de classe qui corresponds 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 a partir de la séquence de chiffres représentant la constante. Pour la chaine "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" $0 - f - > 1 - u - > 2 - n - > 3$
constante entière 0 -chiffre $- >$1-chiffre $- > 1$
identificateur 0-lettre$- >$1-lettre+chiffre$- > 1$

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

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

Résolution de deux types d'ambiguités. Ambiguïté 1 : le mot funx est reconnu par l'expression régulière des identificateurs, mais contient un préfixe reconnu par une autre expression régulière (fun)
⇒ 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 clef "fun" mais aussi par celle des identificateurs
⇒ 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 memorisé ⇒ echec.
2. On a lu le préfixe w de l'entrée wv, avec w la chaîne reconnue par le dernier état final rencontré. ⇒ 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 expression a, ab, bc, l'analyse lexicale échoue sur abc (ab est reconnu, comme plus long, puis echec sur c) pourtant le mot abc appartient au langage $(a|ab|bc)*$

11.2 Analyse lexicale avec Ocamllex

un fichier ocamllex porte le suffixe .mll et a la forme suivante

```
{
% ocamllex lexer.mll
... code OCaml arbitraire ...
}
rule f1 = parse
| regexp1 { action1 }
| regexp2 { action2 }
| ...
and f2 = parse
...
and fn = parse
...
{... code OCaml arbitraire ...}
```

La compilation de ce fichier produit un fichier OCaml lexer.ml qui définit une fonction pour chaque analyseur, $f_1, \ldots, f_n$:
val $f1$: Lexing.lexbuf $\rightarrow \tau_1$
val $f2$: Lexing.lexbuf $\rightarrow \tau_2$
...
val f_n : Lexing.lexbuf $\rightarrow \tau_n$
le type Lexing.lexbuf est celui de la structure de donnees qui contient l'etat d'un analyseur lexical. la fonction from_channel permet de connstruire un lexbuf a partir d'un fichier ouvert :
val from_channel :
Pervasives.in_channel → lexbuf

Opérateur pour décrire des expression réguliére lex :

```
_  n'importe quel caractere
'a' le caractére 'a'
 "foobar" la chaîne "foobar"
[ caracteres ] ensemble de caracteres
 (par ex. [ 'a'-'z' 'A'-'Z'])
[^caracteres] complementaire
     (par ex. [^ '"'])
r1 | r2 alternative
r1 r2 concatenation
r*  etoile
r+ une ou plusieurs occurence =rr*
r? une ou zero occurence =epsilon | r
eof  la fin de l'entree
```

On peut nommer des expressions regulières :

```
let letter = ['a'-'z' 'A'-'Z']
    digit = ['0'-'9']

rule token = parse
| letter (letter | digit | '_')*
      as s{ Tident s }
| digit+ as s{Tconst int_of_string s}
```

On se donne un type Caml pour les tokens

```
type token =
| Tident of string
| Tconst of int
| Tfun
```

on peut récupérer la chaîne reconnue, ou les sous-chaînes reconnues par des sous-expressions régulières, a l'aide de la construction caml "as". Dans une action, il est possible de rappeler récursivement l'analyseur. Le tampon d'analyse lexicale doit être passé en argument ; il est contenu dans la variable lexbuf. Il est ainsi facile de traiter les blancs :

```
rule token = parse
| [' ' '\t' '\n']+ { token lexbuf }
```

Pour traiter les commentaires, on peut utiliser une expression régulière ... ou un analyseur dédié :

```
rule token = parse
| "(*" { comment lexbuf }
| ...
‘
and comment = parse
| "*)" { token lexbuf }
| _ { comment lexbuf }
| eof{failwith
      "comment non terminé"}

| ...
```

Avantage : on traite correctement l'erreur liée a un commentaire non fermé. Autre intérêt : on traite facilement les commentaires imbriqués. 1- Avec un compteur.

```
rule token = parse
| "(*" { level := 1;
     comment lexbuf;
     token lexbuf }
| ...
and comment = parse
| "*)" { decr level;
      if !level > 0 then
           comment lexbuf }
| "(*" { incr level;
         comment lexbuf }
| _ c{ comment lexbuf }
| eof {failwith
      "comment non terminé" }
```

2- Voire même, sans compteur, en utilisant la pile des appels :

```
 rule token = parse
  | "(*" {comment lexbuf;
          token lexbuf}
  | ...
and comment = parse
| "*)" { () }
| "(*" { comment lexbuf;
         comment lexbuf }
| _ { comment lexbuf }
| eof {failwith
     "comment non terminé"}
```

Note : Le langage des commentaires imbriqués n'est pas rationnel. Le fait d'utiliser des actions permet donc de dépasser la puissance des expressions rationnelles.

12 Démontrer qu'un langage est ou n'est pas algébrique

12.1 Forme normale de Chomsky

On va l'appeler "FNC" dans la suite. c'est cette forme qu'on utilise pour l'algorithme CYK vue en PIL en L2. Nous allons utiliser nous, la FNC pour démontrer une nouvelle version du théorème de pompe étendues pour les langages algébriques.

Theoreme 11 *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* $S \rightarrow \epsilon, P \rightarrow a, ouP \rightarrow MN$ *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 $A \rightarrow a | a \in \Sigma$.
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 $X \rightarrow X1...Xn$ pour $n > 2$ par $X \rightarrow X1YetY \rightarrow X2...Xn$ 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 temps qu'il n'y ait plus que deux non-terminaux en membre droit.

Exemple : notre fameuse grammaire $S \rightarrow \epsilon, S \rightarrow aSb$ qui génére $\{a^n b^n, n \geq 0\}$. On choisis celle la 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 $S' \rightarrow S|\epsilon, S \rightarrow aSb|ab$.

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

12.2 Pompage des algébriques

Expliquons sur $S \to aSb|\epsilon$ le principe du pompage. Ca 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 a dire ou S apparaît a la fois à gauche et a 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 mot pompé, le deuxième sous mot pompé, et le mot aprés le deuxième sous mot pompé.

Theoreme 12 *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* $ux^ivy^iw \in L$.

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 à 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 longeur inférieur à $2^h - 1$. Le mot total est donc bien inférieur a $2 * (2^h - 1) = 2^{h+1} - 1$.

Preuve pompe. Soit $N = 2^{|\Sigma_N|+1}$. si un mot m a une longueur$\geq N$ alors d'apres 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 met 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 longeur fait $|\Sigma_N|+1$. On a une première dérivation $S- > uAw$ une deuxième $A- > xAy$ ensuite le deuxième A se réécrit à son tour $A- > v$. On peut ici supposer sans perte de généralité que u, x, v, y, w ne contiennent que des terminaux (s'il 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 $A- > xAy$ avant de faire la réécriture non récursive $A- > v$.

On doit rappliquer le lemme encore une fois : Grâce au fait qu'on a choisit 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 inférieur 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 champs des possibles.

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

Theoreme 13 *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 ca 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 $u = a^N b^N c^N$
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 choisit donc $k = 2$. On a $ux^k vy^k w \notin L$ 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.

12.3 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 $a^n b^n c^m$ et $a^m b^n c^n$ intersecté donne $a^n b^n c^n$. 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))

13 Automates à pile

13.1 Motivation

Un automate à pile est un automate d'état fini qui a une pile en plus, avec donc un ensemble étalement fini de symboles pouvant être dans la pile. La fonction de transition de l'automate, en plus de lire une lettre et detérminer un nouvel état, peut lire le sommet de la pile (pour se faire on suppose que on le dépile) pour décider de sa transition, et peut empiler de nouveau 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 il représente un bon compromis entre simplicité et puissance.

- Ils sont suffisamment puissant pour réaliser l'analyse syntaxique d'un programme
- Ils restent suffisamment simple 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 différent comportements possibles. Ce n'est que lorsque on les génère automatiquement, pour reconnaître un vrai langage de programmation, que ces automates vont devenir gros, avec plus que 10 états.

13.2 Langages reconnaissables par automates à pile

Definition 21 *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'autre 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

Definition 22 *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 notée* $(q, \alpha) - a, X, \beta \to (q', \alpha')$, *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 23 *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* $(q_0, \alpha_0) - a_1, X_1, \beta 1 \rightarrow (q_1, \alpha_1)...(q_{n-1}, \alpha_{n-1}) - a_n, X_n, \beta_n \rightarrow (qn, \alpha_n)$*. L'entier n est la longueur du calcul et* $a_0 a_1 ... a_n$ *est le mot lu par le calcul, en abregé* $(q_0, \alpha_0) - a_0 ... a_n \rightarrow (q_n, \alpha_n)$

Plusieurs notion de reconnaissance. toutes équivalentes,

Definition 24 *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 14 *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 (si 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 grammaires hors-contexte.

Theoreme 15 *Un langage est hors-contexte si et seulement si 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 $S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow aaabbb$. A l'étape aaSbb le préfixe du mot reconnu sera le début du mot généré jusqu'au premier non terminal, c'est a 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 $N \to \beta$, on fait correspondre la transition $q - \epsilon, N, \beta \to q$
Pour tout terminal a on utilise la règle $q - a, a, \epsilon \to q$ Une récurrence simple montre que les dérivations dans la grammaire correspondent très exactement aux calculs de l'automate. Plus précisément, la grammaire dérive $S - >^* uX\alpha - >^* m$ si et seulement si l'automate peut générer la configuration $q, \gamma_0, m - >^* q, X\alpha, v$ 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.

14 Analyse syntaxique ascendante

14.1 Premier et Suivant

On définit deux sortes d'ensemble de lettres associés aux non-terminaux :
$premier(N \in V) = \{a \in \Sigma_T | \exists \beta \in \Sigma^\star N \to^\star a\beta\}$
$premier(\alpha \in \Sigma^\star) = \{a \in \Sigma^\star | \exists \beta \in \Sigma^\star, \alpha \to^\star a\beta\}$
$suivant(N \in V) = \{a \in \Sigma | S \to^\star \alpha N a \beta\}$

Ces ensembles se calculent en résolvant un système d'équations : Pour les premier, l'équation d'un non-terminal X s'écrit en regardant les règles qui réécrivent X, et en cherchant mentalement toutes les lettres qui commencent les membres droits de ces règles, ou qui commencent les mots dérivables à partir de ces membres droits. Ça fera des équations i.e. les premiers de X sont définis a partir des premiers de Y, Z . . .si ces membres droits commencent par des non-terminaux Y, Z

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 DROITS. Et, de plus, si jamais ces occurrences se trouvent à la toute fin du membre droit, on va rajouter les suivants du non-terminal réécrit dans cette règle,i.e. le membre gauche. On obtient ainsi une fois de plus un système d'équations.

Dans les deux cas, le système s'écrit comme une équation $X = f(X)$ ou 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 inclue 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 d'union, et l'union conserve l'inclusion, si A inclus dans A' et B inclus dans B', alors A union B est inclus dans A' union B'. La résolution se fait en itérant la fonction f depuis l'élément minimum (le vecteur d'ensemble vide) jusqu'à obtention d'un point fixe. La suite des itérations croit, dans un espace borné, donc ça va fatalement converger.

Exemple : On Applique 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 rajoute $S \to E\#$ pour être sûr que tout le monde a un suivant. On écrit la fonction f, on résout l'équation par itération. On obtient pour les premier, 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 j'itère f en démarrant de (∅,∅,∅,∅,)
on obtient donc la suite de vecteurs d'ensembles : (∅,∅,∅,∅)
(∅,∅,{ '(', id, cte },∅)
(∅,{ '(', id, cte } ,{ '(', id, cte },∅)
({ '(', id, cte },{ '(', id, cte } ,{ '(', id, cte },∅)
({ '(', id, cte },{ '(', id, cte } ,{ '(', id, cte },{ '(', id, cte })
Après ça bouge plus, c'est le point fixe. Trouver les équations pour les suivants est plus compliqué que pour les premiers, surtout en présence de non-terminaux "annulable", i.e. qui peuvent dériver ϵ : L'algorithme suivant résume ce que j'ai déjà dit d'une manière plus formelle : Pour chaque non-terminal E, pour chaque occurrence de E dans un membre droit des pro-

ductions, il faut regarder ce qui suit E :

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

Cet algorithme donne le système :
suivant(E) = { +, #, ')' } ;
suivant(F) = { + } union Suivant(E)
suivant(G) = Suivant(F)
suivant(S) = ∅
Avec les mêmes conventions de notation que pour premier, le système s'écrit
f(E,F,G,S) = ({ +, #, ')' } , { + } union E), F, ∅)
et l'itération donne :
(∅,∅,∅,∅)
({ +, #, ')' }, { + } ,∅,∅)
({ +, #, ')' },({ +, #, ')' } ,∅,∅)
({ +, #, ')' },({ +, #, ')' } ,{ +, #, ')' },∅)

14.2 L'analyse ascendante, Keskecé ?

Avec l'analyse descendante vue en PIL l'an dernier, 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 vu jusqu'à maintenant.

Caractère d'avance. De plus, cet automate d'analyse utilise un "caractères d'avance", pour décider de ses transitions. Le caractére d'avance désigne le prochain caractère du mot à lire. On dit caractère d'avance parceque 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" a 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 compile 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'autre mots, l'ensemble des langage 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 compilateur.

14.3 L'automate à deux états.

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

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 caractères. L'appellation shift vient du fait que on déplace le prochain caractère depuis le mot, sur la pile. Reduce lui, se comprends comme diminuer la pile vu que on remplace le membre droit qui est dessus et qui est en général plus long que un caractère, par le membre gauche qui fait un caractère. L'automate ascendant à deux états

avec le formalisme d'automate a pile, l'état 2 est final. Écrire cet automate pour notre exemple fil conducteur, en notant les action 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 que on construit effectivement l'arbre de dérivation en partant des feuilles, en réduisant dés que c'est possible. Et aussi que c'est bien une dérivation droite. Faire constater que le mot de pile corresponds à la liste des racines des branches que 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 $S- > \epsilon$. 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 $S- > \epsilon$ 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 égale au fond de pile. Il y aussi un conflit réduction/réduction entre $S- > \epsilon$ et $S- > aSb$. On résous ce conflit en choisissant systématiquement $S- > aSb$.

Pour pouvoir mener l'analyse automatiquement il faut enlever le non déterminisme automatiquement, i.e. 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 régle qui précise quel choix faire. En d'autre terme 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 possible entre shift ou reduce ;
2. Avec les automate SLR(1) puis LR(1) on montre comment utiliser le caractère d'avance.

14.4 L'automate LR(0).

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

Definition 25 (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 concatenation 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 26 (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 27 (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 $X \rightarrow \gamma$ concernée.

Fonctionnement d'un analyseur LR Sur un état d'analyseur $(\alpha \quad , \quad xw)$ 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** $X \rightarrow \gamma$ (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 28 *Un* ITEM LR(0) *pour une grammaire G est une production $X \rightarrow \gamma$ de G plus une position j dans γ. Cela est noté, $X \rightarrow \alpha \cdot \beta$*

L'intuition est qu'on est dans l'état $A \rightarrow \alpha \cdot \beta$ 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 $S' \rightarrow S\#$, le symbole # sert à marquer la fin du mot. L'état initial de l'automate LR(0) est l'item $S' \rightarrow .S\#$ on attend de lire sur l'entrée une séquence dérivable à partir de $S\#$.

Les transitions de l'automate LR(0)

Definition 29 (GOTO) *Supposons d'être dans un état $A \rightarrow \alpha \cdot X\beta$, 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(`$A \rightarrow \alpha \cdot X\beta, X$`)` $= A \rightarrow \alpha\ X \cdot \beta$

$\epsilon-$transitions de l'automate LR(0) Si on est dans l'état $A \rightarrow \alpha \cdot X\beta$, 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 $A \rightarrow \alpha \cdot X\beta$ vers tout les états $\{(X \rightarrow \cdot\gamma)\}$ ou $X \rightarrow \gamma$ 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, notion vue dans le cours de PIL, l'an dernier. 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 30 ($\epsilon-$Clôture LR(0))
`Clôture(`I`) =`
`répéter tant que` I `grandit`
`pour tout item` $A \rightarrow \alpha \cdot X\beta$ `dans I`
  `pour toute production` $X \rightarrow \gamma$
    $I \leftarrow I \cup \{(X \rightarrow \cdot\gamma)\}$
`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)

Theoreme 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 est cependant pas évident de voir cela, c'est simplement parceque l'automate LR(0) ne fait que mimer des dérivations ; Ce théoreme sera donc simplement admis, sa preuve etant soit considérée comme complétement triviale, ou ou contraire tout a fait obscure ! On se limite a 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 $S + S\# + a^\star + a^\star S + a^\star Sb$

Theoreme 17 *Les états de l'automate LR(0) précise quand on peut lire un terminal, il faut un point avant ce non 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 a la fin : $A \to \beta$.*

Ce théoreme va nous permettre de décider des actions possible, simplement en regardant les items contenus dans chaque états :

La construction de la table LR(0) Soit G une grammaire augmentée avec $S' - > S\#$, 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 $_i$ 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.

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

Note 2 : On remarque que dans l'automate LR(0) une réduction est décidée uniquement depuis l'état, i.e. 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-determinisme que nous appelerons conflit. Il existe deux sortes de conflits : entre une reduction 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 colonne pour l'état courant, on suppose qu'il est toujours 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 sur 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.

14.5 Automates supérieur à 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 le caractère d'avance

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 tout 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 caractére d'avance doit etre aussi un caractére 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 ($A \to \alpha \cdot X\beta, z$), 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 31 (Cloture LR(1) d'un état I) *Pour tout item* $(N \to \alpha.X\beta, b) \in I$, *pour toute règle* $X \to \gamma$ *et tout terminal* $a \in Premier(\beta b)$, *alors* $(X \to .\gamma, a) \in I$.

Action LR(1) : Une réduction $N \to \gamma$ 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 $(N \to \gamma., a)$. 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 $->$ 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 état obtenus par cloture, on regroupe ensemble les item 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)

i.e. 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.)

14.6 Différence LALR(1)-LR(1)

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

S	::= (X	X	::= F]
S	::= E]	E	::= A
S	::= F)	F	::= A
X	::= E)	A	::= ϵ

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 $\{[F \rightarrow A.,]], [E \rightarrow A.,)]\}$ (transition par '(' puis par A depuis l'état initial) l'autre des items $\{[F \rightarrow A.,)], [E \rightarrow A.,]]\}$.(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 $\{[F \rightarrow A.,)]], [E \rightarrow A.,])]\}$ 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 précédences ou d'associativité permettant de désambiguïser automatiquement lors du calcul de l'automate LALR (sans transformer la grammaire à priori). Les générateurs d'analyseurs d'aujourd'hui permettent ce type de désambiguïsation .

15 Machine de Turing

C'est la troisiéme partie du cours qui aborde les langages les plus compliqués, ceux que l'on peut reconnaitre avec n'importe lequel algorithme, la notion d'algorithme étant modélisé par des programmes très elementaire sur les machine "de Turing".

Definition 32 *Une machine de Turing est un triplet (Q,Σ,δ) Q est l'ensemble des états, Σ est l'alphabet , avec un symbole spécial $\perp$ "case vide". δ la fonction de transition : $Q \times \Sigma \mapsto \{G, D\} \times \Sigma \times Q$*

Une configuration comprends 1- un ruban bi-infini vide presque partout (fonction f de Z dans Σ ou l'ensemble des x tels que $f(x)$ non vide est fini) 2- Un pointeur vers une case de ce ruban appelé tête de lecture 3- Un état.
Une transition comporte la lettre lue, la nouvelle lettre posée, et le mouvement de la tête de lecture ; Pour simplifier l'écriture, si la nouvelle lettre est la même que l'ancienne -ce qui arrive souvent- on n'indique pas de lettre ; Pour exécuter une transition :
1- on remplace la lettre couramment pointée par la nouvelle lettre spécifiée dans la transition
2- on déplace la tete de lecture, 3- On change d'état.

Definition 33 *Une machine de Turing est dite déterministe, si on a jamais deux transitions possibles simultanément.*

Pour le moment, on ne considère que des Machine de Turing Déterministe (MTD), car le non-déterminisme est difficile à appréhender ;

Protocole reconnaissance langage :
1- On démarre avec un mot, depuis un état initial q_0, la tête de lecture pointe sur la première lettre.
2- On transitionne, jusqu'à ce que cela bloque (plus de transitions possible).
3- Si l'état dans lequel on se trouve à ce moment la, est final, alors le mot est reconnu.

Exemple : une MTD qui reconnaît $a^n b^n c^n$ On utilise la plupart du temps des transitions qui ne font que bouger la tête, et ne modifie pas le ruban.

Algorithme :

1. On va à droite du mot en vérifiant la forme $a^* b^* c^*$ (4 état) puis on fait des balayages :

2. Sur le trajet droite-gauche, on enlevé à chaque fois le premier 'c' rencontré (on le remplace par un dièse) puis le premier 'b', puis le premier 'a'
3. On repart vers la droite jusqu'à vide
4. si lorsqu'on revient à gauche, il n'y a plus que des dièses (en passant les dièses, on tombe pas sur 'c' mais sur vide) c'est gagné, sinon on recommence un tour de suppression. si ca bloque c'est perdu ;

Pour construire une machine de Turing sans se tromper, il faut tout d'abord écrire ce type d'algorithme qui mentionne des "balayages", puis ensuite faire les transitions, et en même temps, dessiner l'évolution du ruban.

Definition 34 *Soit M un Machine de Turing, pas forcément déterministe. Le langage reconnu par cette machine est l'ensemble des mots u tels qu'il existe une exécution de la MT qui commence avec u sur la bande et qui termine dans un état final.*

15.1 Langage décidable

Definition 35 *Un langage est dit "semi-decidable" si il existe une MTD qui le reconnaît, il est dit "décidable" si de plus cette MTD s'arrête pour toutes les entrées.*

Contrairement aux automates à pile, on peut simuler une MT Non déterministe, par une MT déterministe donc la décidabilité ne change pas si on considère le non déterminisme.

Comme elle ne "consomme" pas les lettres du mot reconnu, une machine de Turing peut facilement boucler. Exemple : la machine $q1 - a, D \to q1 - b, G \to q1 - \perp, D \to q2$ final ; Elle reconnaît a^* mais elle boucle si il y a un b a la fin d'un groupe de a ; la tête se met a faire un va et vient infini gauche droite gauche droite gauche .... Le problème si on boucle sur un mot en entrée, c'est que si jamais le mot n'est pas dans le langage, on ne le saura jamais, on saura seulement si le mot est dans le langage. D'ou l'appellation "semi-decidable".

La décidabilité complète permet de s'assurer que on reconnaît et les mots du langage, et les mots hors du langages, car la MT qui reconnaît est "clean", elle s'arrête toujours, donc elle s'arrêtera sur un état pas final si le mot est hors du du langage.

15.2 Langage non décidables.

1. Pour les FSA $a^n b^n$ pas reconnaissable
2. Pour les automates a piles, $a^n b^n c^n$ pas reconnaissable
3. Pour les MTD, l'équivalent de pas reconnaissable est "pas décidable", ou "indécidable" qui signifie il n'existe pas de machine qui reconnaisse et qui ne boucle jamais.

On va montrer que les MTD sont aussi puissantes que les ordinateurs. Si on ne peut pas décider si un mot donné est dans le langage ou pas, cela signifie donc qu' il n'existe pas un ALGORITHME au sens large, qui s'arrête partout, et répond oui ou non si le mot est dedans ou pas dedans.

15.3 Machine universelle

Il faut considérer un langage dont les mots codent des machine de Turing ayant certaines propriétés. Notion de codage, il faut commencer par coder l'alphabet de la machine, puis coder chaque état chaque transition, et également la configuration initiale.

Definition 36 *Une machine de Turing universelle, est une machine capable de simuler toutes les autres à partir d'un codage.*

Appelons un tel langage de mots codant des machines, un "langage de "machine". On construit des langage indécidable en considérant de tels langages de machines. NB on pourrait aussi dire un "langage de programmation", parce-que pour une machine universelle simulant une autre machine quelconque M, M est comme un programme.

15.4 Le problème de l'arrêt

On parle de problème indécidable plutôt que de langage indécidable. Puisque les mots représentent des machines, on parle d'un problème portant sur une propriété des machines. la star des problèmes indécidable = la machine encodée va t'elle s'arrêter, ou va t'elle boucler ? , les automates à pile, peuvent aussi boucler, mais on peut détecter s'il bouclent, car cela implique que le mot n'est pas consommé, donc seulement des epsilon transitions sont prises, et la pile va se mettre a augmenter

de façon régulière, lorsqu'on on oublie le cas d'arrêt dans un appel récursif, on obtient une pile d'exécution qui croit, jusqu'au message d'erreur "stack overflow".

Theoreme 18 *Le problème de l'arrêt pour les machine de Turing est indécidable.*

Le langage des mots représentant des couples machine + état initial du ruban, qui s'arrêtent, n'est pas décidable.

Preuve par l'absurde : (la preuve la plus célébre de l'informatique) Supposons qu'il existe un programme surnommé "Halt" qui décide le problème de l'arrêt. C'est à dire :

- Halt prends un programme prog, et une entrée m (son entrée est en deux parties)
- Si prog(m) s'arrête, alors Halt accepte (prog, m) en un temps fini (réponds oui, i.e s'arrete sur un état final) ;Il doit donc simuler prog sur m
- Si prog(m) ne s'arrête pas, alors Halt s'arrête quand même, et refuse (prog, m) en un temps fini (réponds non).

Pour simplifier, on utilise un machine avec des etats d'acceptation et aussi des états de rejet, et on rajoute des transitions pour que lorsque la machine s'arréte, c'est tourjours ou bien sur un état d'acceptation ou bien sur un état de rejet. Cela est un peu équivalent à rajouter une état puis lorsqu'on veut obtenir un automate d'états finis complet, à partir d'un automate qui bloque par manque de transitions.

Notons que cette machine Halt est une machine universelle, elle doit pouvoir simuler d'autres machines via leur code ; NB si prog ne représente pas une machine, que se passe t'il ? Réponse : m va bloquer pour des raisons de syntaxe, donc Halt réponds oui, mais a la limite on s'en fiche.

On construit une MTD "diagonale" qui utilise Halt, et rajoute quelques états (faire au tableau). Diagonale prends un entrée un code de MTD x, fait tourner Halt également sur x, et "inverse" le résultat : elle se met a boucler si Halt réponds oui

```
diagonale(x):
    si Halt accepte (x,x) boucle infinie
    sinon terminer en acceptant
```

Mais, on obtient une contradiction pour l'entrée x =diagonale. En effet, diagonale(diagonale) boucle si et seulement si Halt accepte (diagonale, diagonale) si et seulement si diagonale(diagonale) termine. Cela prouve donc par l'absurde que Halt n'existe pas.

Voici maintenant une démonstration dans le cadre hors Machine de Turing, peut etre plus claire : Enoncé du problème : On se demande s'il existe un programme H, par exemple en java, qui pourrait décider, pour n'importe quel autre programme P en java et entrée E, si P s'arrêtera ou tournera indéfiniment sur E. En particulier, rien n'empéche de considérer que E peut aussi représenter un programme, et que dans ce cas, P fait tourner ce programme.

Autrement dit :

- H(P, E) = "oui" si P s'arrête sur E,
- H(P, E) = "non" sinon.

Hypothèse (par l'absurde) Supposons que ce programme H existe.

On construit alors un nouveau programme spécial, qu'on va appeler D.

Construction du paradoxe Le programme D prend un programme en entrée (disons X) et fait ceci :

Il appelle H(X, X) pour savoir si X s'arrête quand on lui donne lui-même comme entrée.

- Si H(X, X) dit que X s'arrête, alors D entre dans une boucle infinie.
- Si H(X, X) dit que X boucle infiniment, alors D s'arrête.

Autrement dit :

```
D(X):
    si H(X, X) répond "oui":
        boucle infinie
    sinon:
        arrêter
```

Et maintenant le twist : appelons D(D) Que se passe-t-il si on appelle D avec lui-même en entrée ? Donc D(D) ?

- Si H(D, D) dit que D s'arrête, alors D boucle infiniment.
- Si H(D, D) dit que D boucle infiniment, alors D s'arrête

Contradiction totale ! Donc H ne peut pas exister.

Conclusion Le Halting Problem est indécidable. Il n'existe aucun programme qui puisse toujours dire si un autre s'arrête ou non.

Exemple de problèmes indécidable simple : Savoir si le jeu de la vie va terminer sur un

cycle. Savoir si une grammaire n'est pas ambiguë. On peut le faire au cas par cas, mais il faut de l'intelligence a chaque fois. Exemple marrant issu des math : une équation à coefficients entiers, admet-elle une solution entière ? ; Exemple : $x^n + y^n = z^n$ a des solutions pour $n = 2$, en effet : 3*3+4*4=5*5. Par contre il n'y a pas de solutions si $n > 2$ (fermat énnonce ce résultat en 1670 et dit que y a une preuve simple, mais qu'il n'a pas la place de le noter dans la marge, c'est démontré en 1994 avec des math super compliquées) De telles équation s'appellent des équation diophantiennes. Ce problème est le 10éme problème de Hilbert (1900). Matiyasevich prouve qu'on ne peut pas décider , il n'existe pas d'algorithme qui peu répondre oui ou non (1971).

Relation avec la logique mathématique : Une preuve est un peu comme un programme ; Théoréme d' incomplétude de Godel : toutes théorie mathématique suffisement compliquée (avec les nombres entiers) contient des énoncés vrai mais indémontrables.

15.5 Pour montrer l'indécidabilité ?

On fait une démonstration par l'absurde. On établit une correspondance avec les machines de Turing, de telle sorte que décider du problème permettrai aussi de décider de l'arrêt ce qui est absurde. Les problème non décidable répondent oui ou non au sujet d'une propriété sur des objets un peu compliqués, suffisamment compliqués pour permettre de faire du calcul. Plus précisément on va montrer que on peut associer a une machine de Turing quelconque, une instance du problème considéré taillée sur mesure qui permet de simuler l'exécution de cette machine de Turing, de telle façon que la machine s'arrête si et seulement si la réponse au problème est oui. Ainsi pouvoir répondre oui, permettrait de prédire l'arrêt d'une machine de Turing quelconque, ce qui est absurde. Nous allons faire cette démarche en TD pour les systèmes de Post, qui sont plus facile ensuite a réutiliser a leur tour pour démontrer par l'absurde que le problème de l'ambiguïté est non décidable.

15.6 La thèse de Church Turing

On montre que tous les modèles de calculs usuels qui semblent plus expressifs, sont pourtant équivalents à la machine de Turing, du point de vue de la décidabilité puis du point de vue de la calculabilité.

Point de vue de la décidabilité : Le fait qu'on sache répondre oui ou non, ne dépends pas du modèle de machine utilisé ;

Modèle 1 : Les machine RAM. Elles ont un nombre fixe de N cases contenant des instructions, un nombre infini de cases mémoires dont l'une est distinguée et s'appelle R, ces cases contiennent des entiers arbitrairement grand.

Les instructions réalisent des calculs en utilisant R et une case de la mémoire désignée par une adresse. Par exemple l'instruction store x, range le contenu de R dans la case d'adresse x. Intérêt : c'est plus expressif, on n'a pas besoin de se promener tout le temps sur le ruban, on peut directement adresser des cases.

machine RAM = langage assembleur hyper simplifié pour usage théorique. Elle sert à démontrer que la possibilité d'adresser ne change pas la décidabilité.

Modèle 2 : des programmes écrit dans un langage évolué, par exemple C++ tournant sur un ordinateur avec mémoire infinie (sinon c'est un automate d'états fini).

Point technique, la machine RAM opère sur des entiers arbitrairement grands. On peut coder de tel entiers sur des mots finis (il faut les dénombrer) et vice versa. Par exemple si l'alphabet à deux lettres O et 1, les mots seront les nombres écrit en binaire.

Theoreme 19 *Les machines de Turing, les machines RAM, les programme C++ sont équivalents du point de vue de la la décidabilité.*

preuve : C++ simule Turing qui simule RAM qui simule C++.
1- C++ simule Turing : on écrit un programme C++ qui simule une machine de Turing.
2- RAM simule C++ : c'est un compilateur
3- Turing simule RAM : Il faut coder l'état d'une machine RAML sur un rubant de Turing. On réprésente R, et la suite des contenu des cases mémoire. le programme lui, est transformé en un diagramme d'états, en utilisant plusieurs états pour chaque instruction.

Thèse de Church Turing : toute machine calculante et équivalente a la MT. Ce Postulat n'a jamais été contredit, mais il est par nature indémontrable, donc on appelle cela une thèse.