Cours de Langages Formels
2020
Frédéric Gruau

Introduction

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. Dans certain cas bien précis, une partie du TD précéde le cours.

0.1 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.

0.2 Panoramique

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

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

TABLE 1 – Les trois niveaux de langages principaux.

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 $(A \cup B).C = (A.C) \cup (A.B)$
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, completé 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.

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.$(rs)t = r(st)$
4.$r(s + t) = rs + rt$
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 appellée 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 reconnait 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 :

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

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'etat 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* $\{K \in P(Q) | K \cap F \neq \emptyset\}$

Theoreme 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. Faire 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 théoreme 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.

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 reconnait 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 meme finaux ca reconnait L_1^+ apres 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 : $L_i = Y_1 + X_{i,0}.L_0 + X_{i,1}.L_1 + ... + X_{i,n-1}.L_{n-1}$ 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 $q_2 - b- > q1$. Cet automate est minimal, on rajoute des états pour le rendre non minimal : on détriple 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 méne de cet états à un final*

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 une partition des états regroupant dans un même sous ensemble les états ayant même futur. On calcule progressivement cette partition en considérant "ont même futur" avec seulement les mots de longueur k. On augmente progressivement k en partant de zéro. Pour $k = 0$, p_0 sépare les finaux des non-finaux. Pour passer de p_k à p_{k+1}, avoir même futur à $k+1$ lettre c'est lire une lettre et tomber dans le même sous ensemble d'états qui on même futur de k lettres ; On considére une à une chaque lettre a de l'alphabet, et chaque classe C de p_k, et on scinde en deux C si l'image des éléments de C par δ arrive dans des classes

différentes de p_k. On scinde en regroupant les états de C qui ont la même image ;

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 de l'automate minimal.

La notion de futur, peut se définir sur des états, mais aussi directement sur des mots :

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

Pour tout automate qui reconnaît le langage : Le futur d'un mot, et le futur de l'état vers lequel ce mot mène. 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.

ex1 : $x \sim y$ si x et y on meme 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 ?

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.

Donc, pour obtenir directement les états du minimisé à partir du langage, on prends états= classes d'équivalence de $\sim_L$ (mots qui ont même futur).

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 reconnait 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 mot on même futur en rajoutant un suffixe ca reste vrai.

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 ménent vers le même état de A. Idée clef : si deux mots ménent 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.

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)

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 reconnait. 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* $m = uvw, v \neq \epsilon, |uv| \leq N$
$\forall k \in N, uv^k w \in L$

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 un mot de 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 pousser les négations à l'intérieur des formules en changeant les quantificateurs,

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

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 $\Rightarrow$ la complexité des transformations est inchangée.

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

Definition 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 $q' \in \delta(q, a)$. $\Rightarrow$ algorithme simple en temps linéaire, ajouts successifs d'états productifs à un ensemble réduit initialement à F .

Un état q est accessible ssi $q = q_0$ ou bien il existe un état accessible q' et une lettre $a \in (\Sigma \cup \epsilon)$ tels que $q \in \delta(q', a)$. algorithme similaire, temps linéaire, par ajouts successifs d'états accessibles à un ensemble réduit initialement à q_0.

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

9 Décision et temps lineaire.

Nettoyer sert a montrer que certain problèmes sont décidables et de plus rapidement. Être décidable signifie 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.

Theorem 8 *Le vide du langage reconnu par un automate quelconque est décidable en temps linéaire.*

Décision du plein Si l'automate est déterministe, ce problème a la même complexité que le problème du vide, puisqu'ils s'échangent par permutation des états acceptants et non-acceptants. Cela n'est pas vrai des automates non-déterministes.

Theorem 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 (à un facteur près). Pour les automates non-déterministes, il est nécessaire de déterminiser au préalable, d'où le saut de complexité.

10 Grammaire hors contexte.

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

Definition 14 *Une grammaire G hors contexte 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. *$R \subseteq \mathcal{P}(\Sigma_N \times \Sigma^\star)$ 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 $S \to aSBC, CB \to BC, aB \to ab, bB \to bb, bC \to bc, cC \to cc$. Elles est hors-contexte, car les membres gauches contiennent plusieurs lettres. Faire une dérivation, montrer que cela génère $a^n b^n c^n, n > 0$.

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 $v = w_1 d w_2$;*

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 $w_0, w1, \ldots, w_n$ 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 itéré 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).*

Classification de Chomsky des grammaires :

- 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 a Pile)
- Type 3, régulier : membre droit contient un seul non terminal toujours tout a la fin (Automate d'etat fini)

Exemple2 (Grammaires hors-contexte) La grammaire $S \to \epsilon | aSb$ génére le langage $\{a^n b^n | n \geq 0\}$.

Exemple3 (grammaire réguliére) : La grammaire $N \to 0|1M, M \to 0M|1M|\epsilon$ génére les entiers naturels en représentation binaire, sans zéros redondants : par exemple, 01 n'est pas dedans.

10.2 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 $N \to \alpha_1 \ldots \alpha_n$ est une règle de la grammaire G.*

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

10.3 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 plu- sieurs arbres de dérivation s dans G.*

Comment enlever l'ambiguité ? (Exo de TD)

Méthode 1, bricolage, on introduit d'autre non terminaux pour forcer un ordre : $E \to E + F|F$
$F \to F - G|G$
$G \to id|cte|(E)$
Méthode 2, la méthode utilisée en pratique, car plus simple, et plus élégante. On utilise des méta régles, 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) correspond 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).

10.4 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 $S \to aMN, M \to a$.

10.5 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 ambigüe ? indécidable, on fera la démo en TD, mais oui.

11 Analyse lexicale

Lesétudiantsonttoustrèsbon

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

De même que dans les langues naturelles, ce découpage en mots facilite 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.

Les commentaires jouent le role de blancs et sont aussi virés.

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 classes de token. Pour chaque classe :

— On donne une expressions rationnelle

— L'analyseur génère l' automate associé.

Exemples :
le mot clef "fun" $0 - f - > 1 - u - > 2 - n - > 3$ final
constante entière 0 -chiffre $- >$1-chiffre $- > 1$ final
identificateur 0-lettre$- >$1 -lettre+chiffer $- >$ 1 final.

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)
$\Rightarrow$ on fait le choix de reconnaître le token le plus long possible

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

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

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

L'algorithme des fois, ne marche pas alors que il pourrait. Avec les trois 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 (pour le TP)

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 Automates à pile

12.1 Langages reconnaissables par automates à pile

Un automate à pile est une machine munie d'une bande de lecture, d'une bande de travail organisée en pile, et d'un contrôle. Donnez directement l'automate qui reconnait $a^n.b^n$

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

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* $(q_0, \gamma_0) - u_1 ... u_n \to (q_n, \alpha_n)$ *d'origine* (q_0, γ_0) *tel que :*

1. *reconnaissance par état final :* $q_n \in F$ *;*
2. *reconnaissance par pile vide :* $\alpha_n = \epsilon$ *;*

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

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

Preuve : Soit A un automate à pile reconnaissant par état final. On construit un automate reconnaissant à la fois par état final et pile vide : il suffit pour cela d'ajouter de nouvelles transitions sur chaque état final de manière à vider la pile. Cette transformation ne préserve pas forcément le déterminisme (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)

12.2 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 12 *Un langage est hors-contexte si et seulement si il est algèbrique.*

Preuve : on utilise la double inclusion.
1- Montrons que HorsContexte est inclus dans Algébrique.
Pour reconnaitre 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 reconnait $L(G)$ par pile vide. l'alphabet de pile contient non terminaux ET terminaux.

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

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

13.1 Forme normale de Chomsky

Theoreme 13 *Pour tout langage hors-contexte L, il existe une grammaire propre G qui l'engendre dont toutes les règles sont de l'une des trois formes $S \rightarrow \epsilon, P \rightarrow a, ouP \rightarrow MN$ 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 commence par rajouter une copie de Σ_T à Σ_N (on notera A la copie de a), puis l'ensemble de règles $A \rightarrow a|a \in \Sigma$. On remplace ensuite les symboles terminaux a figurant dans les membres droits de règles initiales par le non-terminal correspondant A, puis on remplace récursivement la règle $X \rightarrow X1...Xn$ pour $n > 2$ par $X \rightarrow X1YetY \rightarrow X2...Xn$ en ajoutant un nouveau non-terminal Y à chaque fois. La FNC est utilisée pour démontrer un théoréme de pompe pour les langages algébriques. Exemple : donner la FNC de la grammaire $S \rightarrow \epsilon, S \rightarrow aSb$ attention, il faut introduire un nouvel axiome, pour obtenir une grammaire propre, car dans une grammaire propre, l'axiome n'apparaît pas en partie droite.

13.2 Pompage des algébriques

Expliquer sur $S \rightarrow aSb|\epsilon$ le principe du pompage (on re-génére S depuis S).

Revenons à notre grammaire de règles $S \rightarrow \epsilon, S \rightarrow aSb$ engendrant les mots de la forme $a^n b^n$, et considérons l'arbre syntaxique du mot ab : Le long du chemin central, le non-terminal S apparaît deux fois. On peut donc obtenir de nouveaux arbres syntaxiques pour la même grammaire en supprimant, ou en répétant un nombre arbitraire de fois le motif central $S(a, , b)$. Par exemple on en peut

déduire que le mot aabb appartient au langage. etc

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

Preuve : on choisit une grammaire $G = (\Sigma_T, \Sigma_N, S, R)$ en forme normale de Chomsky qui engendre le langage L.

Lemme si N est le nombre de non-terminaux dans une grammaire Chomsky, et un arbre syntaxique a une hauteur h, alors le mot des feuilles est de longueur $2^h - 1$ au plus.

Preuve du Lemme par récurrence.
cas $h = 1$ une branche $2^h - 1 = 1$.
Cas $h+1$ on applique la formule sur chacun des deux sous arbres dont la profondeur est h

Preuve pompe. On choisit $N = 2^{|\Sigma_N|+1}$. 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 $\Sigma_N + 1$. soit C un chemin dans cet arbre de taille $|\Sigma_N| + 1$. il existe deux nœuds étiquetés par le même non-terminal A (faire dessin). 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). 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 $2^{\Sigma_N+1} = N$, yes.

Comme pour les rationnels, on utilise la contraposée de la pompe, pour montrer qu'un langage n'est pas algébrique. Exemple a^p , p premier n'est pas algébrique. (même démonstration que pas rationnel) $a^nb^nc^n$ n'est pas algébrique. je choisis $u = a^Nb^Nc^N$ et une décomposition quelconque, comme $|xvy| < N$ il ne peut contenir à la fois a, b et c

13.3 Clôture

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

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

14 Analyse syntaxique ascendante

14.1 C'est quoi l'analyse ascendante ?

On construit l'arbre de dérivation en partant des feuilles vers la racine. L'analyse descendante vue en PIL l'an dernier, construit l'arbre de la racine vers les feuilles. Les grammaires qui marchent pour l'analyse Descendante (resp. ascendante) s'appelle LL(1) premier L : left pour mot lus de gauche a droite, deuxième L= Left pour dérivation gauche (resp. LR(1) le R veut dire Right pour dérivation droite). Le chiffre 1 signifie un caractère d'avance.

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, LL(1) est inclus dans LR(1), mais pas le contraire. l'analyse ascendante est donc plus générale que l'analyse descendante. C'est celle utilisée en vrai dans les compilateur.

14.2 L'automate à un seul état.

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

L'automate d'analyse ascendante se construit ainsi : Soit $G = (\Sigma_T, \Sigma_N, S, R)$ la grammaire hors-contexte qui nous intéresse, et qui est cette fois augmentée de la règle $S' \to S\#$. On rajoute aussi le caractère # à la fin du mot. L'automate est $A = (\Sigma_T \cup \{\#\}, \Sigma_N \cup \{S'\}, \gamma_0, \{1,2\}, \{2\}, T)$, 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 $1 - \epsilon, S\#, S' \to 2$ descendant

L'automate ascendant à deux états avec le formalisme d'automate a pile, l'état 2 est final. Ecrire cet automate pour notre exemple fil conducteur, en notant des action reduce et shift. Faire tourner cet automate, pour analyser la chaine *aabb*. Ecrire la pile a gauche le mot a droite, et une colonne pour les actions.

L'automate est non-déterministe puisque lecture et réduction $S- > \epsilon.$ peuvent être fait simultanément. On parle de conflit lecture/reduction. Il y aussi des conflits réduction/réduction entre $S- > \epsilon$ et $S- > aSb$. L'automate ne peut être déterministe qu'en l'absence de tels conflits.

Enlever le non déterminisme = dire s'il faut lire ou réduire, et pour réduire, avec quelle règle, (peut y en avoir plusieurs). On dira que l'on "résous des conflits". Les conflits peuvent être entre un shift et un reduce, ou entre deux actions reduce ; Nous allons les résoudre grâce à deux techniques :

1. Un automate "MAGIQUE" dont les états vont déjà préciser quelles sont les actions possible shift ou reduce ;
2. La lecture d'un caractère d'avance.

14.3 L'automate Magique.

Theoreme 15 *le langage des mot de pile possible est un langage régulier. On peut facilement construire l'automate d'état finis "magique" le reconnaissant. L'état dans lequel on se trouve après lecture du début du mot, renseigne sur l'action qu'il faut faire pour la prochaine étape d'analyse ;*

Construction de l'automate Magique :

- item=règle + point dedans (ou on est dans la règle)
- états = ensemble d'items
- clôture d'un ensemble d'item : Règle du "rebondissement".
- transition : on déplace le point,

Règle du rebondissement : Si un état contient l' item $M \to \alpha.N\beta$ ou N est non terminal alors on rajoute tout les item $N \to .\gamma$ pour toute règle $N \to \gamma$ de G ; Attention : on peut re-rebondir, et re-re-rebondir etc...

Transition par une lettre x : dans tous les items ou y a un point avant le x, on déplace le point aprés le x, et on recopie ces (potentiellement plusieurs) items modifiés dans un nouvel état, après quoi, on cloture, et on rebondit, re-rebondi et re-re-rebondi ...L'ensemble de ces états et de ces transition constitue ce qu'on appelle l'automate LR(0).

Faire l'automate LR(0) de l'exemple. Cet automate reconnaît les mots de pile d'une analyse réussie. 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 16 *Si on fait tourner l'automate pour reconnaître un mot m mélangeant terminaux et non terminaux, et que on ne bloque pas, m est un mot de pile possible. De plus, si on arrive dans un étant q, il doit contenir un item $N \to \alpha.\beta$ tel que α est suffixe de m , $m = m'\alpha$ et il existe t un mot de terminaux et une dérivation droite de $S- > m'Nt- > m'.\alpha.\beta.t$*

Ce théorème renseigne sur le prochain pas à faire pour réaliser l'analyse ascendante de $m\beta$ qui remonte la dérivation.

1. si $\beta = \epsilon$ alors α est appelé un "manche" de l'analyse ascendante, i.e. la dernière étape de la dérivation droite est de la forme $N \to \alpha$. On en déduit que la prochaine étape de l'analyse ascendante (qui va dans l'autre sens) est l'action reduce($N \to \alpha$). Plus généralement, on pourra faire une action reduce ($N \to \alpha.$) depuis un état donné, seulement si l'état contient un item $N \to \alpha.$ avec le point au bout,
2. si au contraire $\beta \neq \epsilon$, la prochaine étape de l'analyse doit augmenter la pile en poussant la première lettre de β sur la pile. En

en déduit que l'on peut faire des shift a seulement si l'état contient un item avec la lettre a précédée du point.

14.4 L'automate Super-Magique

Les deux sortes d'actions possible dans un état donné sont donc :
1- reduce ($X \to \alpha.$) si il y a un item $x \to \alpha.$ avec un point final.
2- shift a (lecture en francais), si y a un item avec la lettre a après le '.'

L'automate magique, en cache un autre "SuperMagique" qui manœuvre l'automate magique. Il gére une pile intercalant les états du magique entre les symboles de piles incluant terminaux et non-terminaux. Ces états sont utilisé avec l'algo suivant :

- Empile l' état courant avant de faire un shift.
- L'état empilé avant α est utilisé sur une action reduce ($N \to \alpha$),
- Il est utilisé en faisant une transition de l'automate magique sur N vers le nouvel état qui sera empilé aprés N.

Faire fonctionner l'automate super-magique sur l'exemple. On note que il y a toujours deux états ou il a le même conflit shift a / reduce $S \to \epsilon$ précedement évoqué ; Donc on n'a pas finit de résoudre le non-déterminisme ;

14.5 Le caractère d'avance SLR(1)

Sur les états où il y a un conflit shift/reduce, on essaie de résoudre les conflits de la facon suivante : on utilise le fait que le mot de pile concaténé avec le reste du mot lu, est toujours 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 confit reduce/reduce L'automate avec les réductions plus ciblées, s'appele "l'automate SLR(1)". Pour mettre en évidence le fait qu'il n'y a plus de conflit, on représente l'automate sous forme d'un tableau, en indiquant les actions possible dans chaque case ; Sur notre exemple, on constate que cela marche parce-que suivant (S)=$\{b, \#\}$, ne contient pas la lettre a. L'automate ainsi que la grammaire s'appelle "SLR(1)".

14.6 Premier et suivant (vu en PIL)

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

Ces ensembles se calculent en résolvant un système d'equations (fait en TD) Cette résolution, se fait en itérant une fonction jusqu'à obtention d'un point fixe. On utilise pour cela le théorème du point fixe de Kleene, qui s'énonce comme suit :

Theoreme 17 *Soient L un ensemble munit un ordre partiel complet, 0 son élément minimum, et $f : L \to L$ une application croissante qui conserve les bornes sup. Alors f a un plus petit point fixe qui est le sup de la suite croissante suivante :* $0 <= f(0) <= f^2(0) <= ... <= f^n(0) \dots$

Précisons que un ordre partiel complet est un ensemble partiellement ordonné qui possède un élément minimum, et dont toutes les chaînes, (suite de valeurs croissantes) ont un sup.

Preuve : Elle est relativement simple et marrante ; Appelons $c = 0 <= f(0) <= f^2(0) <= ... <= f^n(0) \dots$, la chaine infinie Le fait que f soit croissante assure que c est croissante car $0 <= f(0) \Rightarrow f(0) <= f^2(0) \Rightarrow f^i(0) <= f^{i+1}(0)$. Soit x le sup de c $f(x) = f(sup(c)) = sup(f(c)) = sup(c) = x$. Si y est un autre point fixe, alors on montre facilement par recurence que $y > f^i(0)$ pour tout i donc par définition du sup, on aura $y > x$.

Exemple : On Aplique donc ce théoreme 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 sur que tout le monde a un suivant. Ecrire la fonction f, montrer qu'elle est croissante, résoudre l'équation par itération.

Trouver les équation pour les suivants est plus compliqué que pour les premiers, surtout en présence de non-terminaux "annulable", i.e. qui peuvent dériver ϵ : On éxecute cet algorithme : Pour chaque non-terminal E, pour

chaque occurrence de E dans un membre droits des productions, 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, ajouter Suivant(A) à Suivant(E), où A est le membre gauchede la production.

14.7 Le cas LR(1) général

On peut faire mieux que SLR(1). On améliore en ajoutant une lettre aux items, 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 résume l'historique connu. Pour compacter l'automate, on regroupe les item de même structure LR(0), donc un item transporte un ensemble de lettres minuscules.

L'ensemble initial d'items est la clôture de l'ensemble $S' \rightarrow .S, \#$. On rebondit comme avant, sauf que il faut tenir compte des premiéres lettres possibles qui suivent le nom terminal, au sein de l'item d'ou on rebondit : pour tout item $(N \rightarrow \alpha.X\beta, b) \in I$, pour toute règle $X \rightarrow \gamma$ et tout terminal $a \in Premier(\beta b)$, alors $(X \rightarrow .\gamma, a) \in I$.

Regle plus précise qu'avant : Une réduction $N \rightarrow \gamma$ ne pourra être effectuée dans un état q en présence du caractère d'avance a, qu'à la condition que q contienne l'item terminal $(N \rightarrow \gamma., a)$. C'est plus précis que SLR(1), car a peut appartenir à suivant de N mais ne pas être caractére de retour ; car, on l'a déjà dit, l'état spécifique de l'automate permet de préciser l'historique connu.

14.8 Le cas LALR(1), utilisé en vrai par les compilateurs.

On réduit le nombre d'états en regroupant les états de même structure LR(0) ; i.e. qui contiennent des items identiques, au caractère de retour près. C'est l'automate que construit Yacc, seul des grammaires très bizarre sont LR(1) et pas LALR(1), donc cela ne pose pas de problème en pratique. (voir exemple donné à la section suivante, pour comprendre.)

14.9 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) il faut construire l'automate LR(1). Cet automate n'a pas de conflits, ce qui permet de vérifier que la grammaire est bien LR(1). Il contient deux états l'un formé des items $\{[F \rightarrow A.,]], [E \rightarrow A.,)]\}$ l'autre des items $\{[F \rightarrow A.,)], [E \rightarrow A.,]]\}$ 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.

14.10 Génération d'analyseurs syntaxiques 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 sera binop $+$(binop $*$(id x)(id y))(cte 3)

Metode cannonique pour traiter 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 (voir TP).

15 Machine de Turing

(Cette partie intègre quelque notes du cours d'informatique théorique de Laurent Rozas)

Definition 25 *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 executer 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'etat.

Definition 26 *Une machine de Turing est dite détérministe, si on a jamais deux transitions possibles simultanément.*

Pour le moment, on ne considère que des Machine de Turing Détérministe (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 machine de Turing déterministe qui reconnaît $a^n b^n c^n$ On utilise la pluspart 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 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 lorqu'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.

15.1 Langage récursif

Definition 27 *Un langage est dit récursivement énumérable si il existe une MTD qui le reconnaît, il est dit "récursif", ou "décidable" si de plus cette MTD s'arrête pour toutes les entrées.*

Origine de cette dénomination : si un langage est reconnaissable par une MTD on peut effectivement énumérer les mots. Il existe un programme qui les énumère, i.e. qui les affiche toutes. Idée de la preuve : on imagine que la machine de Turing peut afficher des caractères (on pourrais aussi les stoquer sur le ruban). On utilise la machine de Turing universelle, qu'on va voir bientôt, Celle ci peut simuler n'importe quelle autre machine. On va simuler un nombre croissant de d'executions, chacune démarrant sur un nouveau mot. Dés que l'une d'entre elle s'arrête et que l'état est final, on affiche le mot sur lequel on avait démarré pour cette execution.

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 n'est pas récursive, car 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 a gauche droite gauche droite gauche .... Le problème si on boucle, 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.

Remarque simple, et utile pour les examens : un langage est décidable ssi lui et son complémentaire sont reconnaissable. preuve, on fait tourner en même temps les deux machines, il y en a une qui s'arrêtera un jour.

15.2 Langage non décidables.

1. Pour les FSA $a^n b^n$ pas reconnu
2. Pour les automates a piles, $a^n b^n c^n$ pas reconnu
3. Pour les MTD, "pas reconnu", signifie "pas décidable",

Si on ne peut pas décider si un mot donné est dans le langage ou pas, cela signifie qu' il n'existe pas un algorithme qui s'arrête partout, et répond oui ou non si le mot est dedans ou pas dedans. Il faut donc chercher beaucoup plus loin pour exhiber un langage pas décidable, parce que cela signifie qu'on ne peut pas calculer.

15.3 Machine universelle

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

Definition 28 *Une machine de Turing universelle, est une machine capable de simuler toutes les autres à partir d'un codage.*

Appelons un tel langage de mot 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 ?

Theoreme 18 *Le problème de l'arrêt 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 surnmommé "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 ;
- Si prog(m) ne s'arrête pas, alors Halt refuse (prog, m) en un temps fini.

NB si prog ne représente pas une machine, que se passe t'il ? Réponse : cela bloque pour des raisons de syntaxe.

Notons que cette machine Halt est une machine universelle, elle doit pouvoir simuler d'autres machines via leur code ; On construit une MTD "diagonale" qui utilise Halt, et rajoute quelques états (faire cela au tableau). Diagonale prends un entrée un code de MTD x :

diagonale(x) : si Halt accepte (x, x) boucle infinie sinon accepter

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.

ex de problèmes indécidable simple : 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 issus des math : une équation diophantienne (a coefficients entiers) admet-elle une solution entière ? ; C'est le 10éme problème de Hilbert (1900). Matiyasevich prouve qu'on ne peut pas décider (1971).

Exercice : Montrer que l'ensemble des mots représentant des machine de Turing qui s'arrêtent est reconnaissable. on peut répondre OUI si cela s'arrête, on ne peut pas savoir si cela ne s'arrête pas.

15.5 La thèse de Church Turing

On montre que tous les modèles de calculs usuels qui semblent plus expressif, sont pourtant équivalent à la machine de Turing, du point de vue de la décidabilité puis dans le prochain chapitre 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 mémoire 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. On peut coder des mot par des entiers (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 interpréter 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 la thèse de Church Turing, et pas le théorème de Church-Turing.

16 la NP complétude

16.1 Calculabilité

On peut facilement faire plus que reconnaître des mots avec des MTD : On peut calculer des fonctions de mots, qui associent un mot à un autre. Si ce mot utilise des chiffres, on va calculer une fonction de N dans N.

Definition 29 *Une MTD M calcule une fonction $f : D \subset \Sigma^\star \mapsto \Sigma^\star$ si lorsque le ruban démarre avec $x \in D$, lorsque la machine M blocque $f(x)$ est écrit sur le ruban.*

Alternativement, on s'arrête non pas quand la machine bloque, mais lorsque elle rentre dans un état final. Exo de td, montrez que ces deux définitions sont équivalentes. Résultat simple et marrant sur l'existence de fonctions non calculables :

Theoreme 20 *Soit la fonction max qui à n associe le max des temps pris par toutes les machines de n états, qui ne prennent rien en entrée et qui s'arrêtent. max n'est pas calculable.*

Preuve : si elle l'était, le problème de l'arrêt serait décidable. Il suffirait de simuler une machine de n états, un nombre de pas egal à $\max(n)$; si elle ne s'est toujours pas arrêtée elle ne s'arrêtera plus par définition de max.

Décidabilité = calculabilité pour des fonctions booléennes. On parlera à présent, lorsqu'on le peut, de fonctions et de calculabilité, plutôt que de fonction booléennes et de décidabilité, car...c'est plus parlant.

16.2 Les machine de Turing Non Déterministes.

Definition 30 *Une machine de Turing Non déterministe a plusieurs transitions possibles, pour certaine configurations . Pour que un mot soit reconnu, il suffit que une exécution parmi tous celles possibles, mènent vers un état final.*

Intuitivement, une MTND est une MTD avec un oracle qui guide le bon choix a faire, lorsqu'il y a un choix. Pour résoudre un problème, une MTND a juste besoin de vérifier qu'une solution potentielle est bien une solution, car l'oracle peut écrire la solution ;

Theoreme 21 *On peut simuler une MTND par une MTD*

On fait une exploration en largeur d'abord de l'arbre des exécutions, cela oblige à représenter les états de plusieurs machine sur un seul ruban. ca va prendre un temps exponentiel. Donc

NP est inclus dans EXP qui dénote la classe des problèmes qui prennent un temps exponentiel.

On ne s'intéresse à partir de maintenant qu'aux machines de Turing qui s'arrêtent tout le temps, et donc aux ensembles décidables reconnus par elles.

16.3 Complexité polynomiale, exponentielle

On défini la complexité, d'abord pour une exécution, ensuite pour un programme en considérant un ensemble de données de taille n, ensuite pour un problème, en considérant le programme le plus rapide qui le résout.

Definition 31 *On appelle complexité en temps d'une exécution d'une machine de Turing le nombre de transitions effectuées lors de cette exécution; et complexité en espace le nombre de cases de la bande qui ont été utilisées pendant l'exécution. La complexité en temps de l'exécution d'un programme C++ le nombre d'opérations effectuées sur des bits, et complexité en espace le nombre de bits utilisé lors de l'exécution.*

On considère le max (c'est ce qu'on appelle communément la complexité au pire) pour un ensemble de données de taille n. On calcule la complexité à un facteur constant prés (notion de grand O) On peut alors parler de MTD de complexité linéaire quadratique, polynomiale, exponentielle,... d'un programme.

Definition 32 *Soit f une fonction calculable, on appelle complexité par MTD, (resp. par C++), de f, la complexité de la MTD, (resp. du programme C++) le plus rapide calculant f, lorsque les données de f ont une taille n.*

Exemple le tri de n entiers a une complexité $nlog(n)$. Puisque l'on a une équivalence Turing-C++ pour la calculabilité, on voudrait aussi des équivalences sur les complexités. Notation : Soit f une fonction calculable, on note CT(f) et CP(f) les complexités de f respectivement par Turing et par C++ On peut clairement simuler une MTD par un programme C++ qui aura la même complexité, la complexité par C++ est donc toujours au moins aussi bonne que celle par MTD, c'est-à-dire que l'on a CP(f) $\leq$ CT (f).

On a aussi CT(f) $\leq$ CP(f)^2.

Preuve : Au temps t, on a besoin de parcourir le ruban seulement un nombre de fois fini pour simuler une opération de 1bit du programme C++. D'autre part, au temps t, l'espace occupé est $< t$ plus la taille du mot initialement sur le ruban. donc la simulation du pas de temps numéro t, de C++, prends au max t unité de temps. La somme des entiers de 1 à n est de l'ordre de n^2. (Faire le détail des calculs au tableau)

Finalement un programme C++ calcule f en temps polynomial ssi la MT aussi, ce qui change c'est juste le degré du polynôme. Idem pour exponentiel, car exp(t)$*$exp(t)=exp($2t$). Les notions de complexité en temps polynomiales et exponentielles ont donc en sens indépendamment de la machine considérée.

Definition 33 *Classe P = probleme polynomiaux en temps, indépendamment de la machine Classe EXP = idem, mais exponentiel,*

Exemple de problèmes de la classe P : produit matrice vecteur, matrice matrice, tri bulle. Exemple de problèmes de la classe EXP : exploration de l'espace des config d'un jeu, après N coups.

16.4 La fameuse classe NP

Complexité pour une MT non déterministe : on considère l'exécution la plus rapide qui reconnaît un mot, parmi toutes les exécutions possibles.

Definition 34 *Classe NP = polynomiaux en temps pour une machine NON DETERMINISTE*

NP= dieu écrit la solution, on a juste à la vérifier, et ce en un temps polynomial. C'est plus facile de vérifier une solution, que de la trouver.

Exemple : le problème des équations diophantiennes, si les solutions sont restreintes a des entiers codés sur n bits, devient NP, construire la machine qui propose une solution avec quelques états. Quelque inclusions simple :
P est dans EXP : trivial
NP est dans EXP : déjà démontré, cf simulation de MTND par MTD
P est dans NP : trivial. On ne sait pas si l'inclusion est stricte, c'est le grand mystère de toute l'informatique.

16.5 Le problème SAT de la satisfiabilité

On se donne des variables $(V_i)i \in 1 \dots v$ et une formule logique sur ces variables en forme normale conjonctive. C'est à dire une formule $C_1 \wedge C_2 \wedge ... \wedge C_c$ ou chaque C_i est une clause $C_i = L_{i,1} \vee L_{i,2} \vee ...L_{i,l_i}$, Chaque littéral L étant une variable ou sa Négation. Le problème SAT pour SATisfiabilité est de savoir si l'on peut affecter des valeurs booléennes aux variables pour que la formule soit vraie Exemple, les variables sont $V1, V2, V3$ et la formule $(V1 \vee V2 \vee V3) \wedge (\neg V1 \vee \neg V2 \vee \neg V3) \wedge (V1 \vee \neg V2) \wedge (V2 \vee \neg V3)$ La formule est satisfiable en utilisant les assignations : $V1 \leftarrow true$, $V2 \leftarrow false$, $V3 \leftarrow false$.

Le problème SAT est clairement dans NP : On obtient un algorithme non-déterministe polynomial qui répond OUI quand la formule est satisfiable, qui consiste à parachuter les affectations, puis à vérifier qu'elles forment bien une solution. Par contre, il ne semble pas que SAT soit dans P, l'algorithme naïf déterministe consiste à essayer toutes les possibilités, et comme il y en a 2^v, si v est le nombre de variables, on obtient bien un algorithme exponentiel. On a des améliorations de cet algorithme naïf, mais, de fait, tout ce qu'on (on = tous les chercheurs en informatique, et même les autres) a trouvé comme algorithme reste de complexité exponentielle.

16.6 Réduction vers SAT.

Qu'est ce qu'une réduction ?.

Definition 35 *Un langage L' est dit réductible en L ssi il existe une MT M de L' vers L, telle que pour tout x, M(x) est dans L ssi x est dans L'.*

Pour savoir si un mot w est dans L', le problème se transforme en le test si $M(w)$ est dans L,

Si de plus M fonctionne en temps polynomial, on dit que L' est polynomial réductible en L.

Theoreme 22 *Tout langage NP, est polynomial réductible en SAT. SAT est donc un problème avec une sorte d'universalité, on dit que SAT est NP-complet.*

Corrolaire : SAT $\in$ P ssi P = NP.

preuve=codage : On considère une MT non déterministe M quelconque, et on introduit un tas de littéraux (en nombre polynomial), et un tas de formules (en nombre polynomial) qui simule le fonctionnement de M (un nombre de steps seulement polynomial est nécessaire). On s'arrange pour que cette formule vérifie que :

M arrive sur un état final ssi la formule est satisfiable.

De plus, si M fonctionne en temps polynomial, la taille de la formule est également polynomiale. On a réduit le pb de décider si le mot est reconnu au pb de satisfaire une formule. La transformation est de nature algorithmique, donc elle même faisable avec une machine. Elle prends également un temp polynomial.

Soit $E \in$NP , soit une machine de Turing M non déterministe qui reconnaît E. il existe un entier F tel que $u \in E$ ssi il existe une exécution qui reconnaît u en un temps inférieur à $|u|^F$. Le nombre de transitions étant majoré par $|u|^F$, le pointeur ne pourra se déplacer que sur les cases de $B(-|u|^F)$ à $B(|u|^F)$; la case $B(0)$ étant la case pointée initialement. On peut alors définir un nombre polynomial de variables et écrire une formule logique de taille polynomiale qui traduise le fait qu'il existe une telle exécution. On a donc les équivalences : $u \in E$
ssi Il existe une éxecution de M en moins de $|u|^F$ transitions et qui n'utilise que les cases $B(-|u|^F)$à $B(|u|^F)$ qui reconnaît u
ssi Une certaine formule logique de taille polynomiale en $|u|$ est satisfiable

Cette formule s'obtient de la facon suivante : Les variables sont (on note $T = |u|^F$, "au temps t" signifie "après avoir effectué t transitions") :

- Pour tout état q tout $t \in 0 \dots N$, $Q(q,t)$ signifie au temps t, on est dans l'etat q"
- Pour tout $c \in -T \dots T$ et tout $t \in 0 \dots N, P(c,t)$ qui signifie "au temps t, la tête de lecture pointe sur la case c".
- Pour toute transition f et tout $t \in 1 \dots T, F(f,t)$ qui signifie "du temps $t-1$ au temps t, on passe par la flèche f "
- Pour toute lettre l, tout $c \in -T \dots T$ tout $t \in 0 \dots N, C(l,c,t)$ qui signifie "au temps t, la case c contient l".
- Pour tout $t \in 0 \dots N, A(t)$ qui signifie "au temps t, on stoppe et on est en état final".

"Il existe une execution" est alors traduit

en un ensemble de clause, dont on donne ici quelques exemples (Exercice, donnez la liste complète des clauses)

— A tout moment, on est au moins dans un état : $\forall t, Q(q0, t) \vee Q(q1, t) \vee ... \vee Q(q|Q|, t)$
— A tout moment, on n'est pas dans deux états à la fois : $\forall t, q = q', \neg Q(q, t) \vee \neg Q(q', t)$
— Initialement, le pointeur pointe sur la case 0 : $P(0, 0)$

Note : on considère que si l'on est en état final au temps t_0 , alors on n'en bouge plus jusqu'au temps T , donc on parle tout de même des moments t avec $t \geq t_0$.

16.7 Pourquoi NP-complet veut dire difficile ?

SAT est NP, de plus il est représentatif de tout les NP, car il simule une machine de Turing non deterministe en temps polynomial. Un tel problème est dit **NP complet**.

Si SAT est dans P, alors NP=P car tout les problèmes NP deviennent P, en les simulant dans SAT.

Si TRUC est NP-complet, alors TRUC $\in$ P ssi P $\in$ NP.

Il y a beaucoup,beaucoup d'autre problèmes qui, tout comme SAT, sont NP-complet : clique d'un graphe, hamiltonien, homomorphisme de graphe, voyageur de commerce ; Si on montre que un pb est NP-complet c'est la preuve qu'il est difficile, car tous ces problèmes réputés difficile sont équivalent modulo une réduction polynomiale.

16.8 Exemple de problème NP-complets

Le problème CLIQUE prend en entrée un graphe non-orienté G et un nombre k et demande si l'on peut choisir un ensemble E de k sommets qui forme une clique (i.e. que $\forall x, y \in E$, il y a une arrête entre x et y).

Le problème du voyageur de commerce demande par quelle ville passer, pour minimiser un trajet parcourant un ensemble donné de villes.