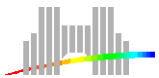
Master Informatique Fondamentale - M1 Compilation

Généralités, Analyse lexicale

Paul Feautrier

ENS de Lyon
Paul.Feautrier@ens-lyon.fr
perso.ens-lyon.fr/paul.feautrier

24 janvier 2007



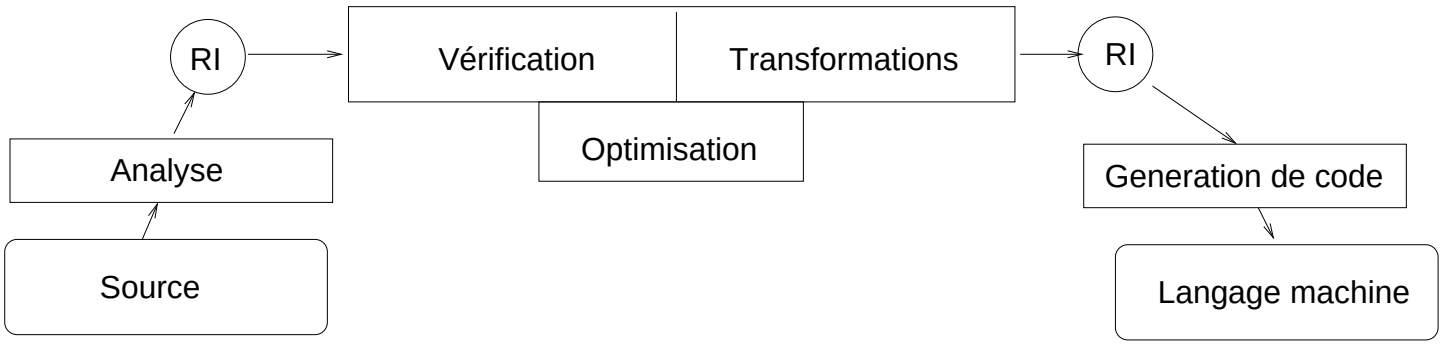
A quoi sert un compilateur

- Un ordinateur exécute du code binaire
- Les programmeurs veulent un langage de haut niveau :
  - clarté, mnémotechnie, compacité
  - redondance (permet les vérifications)
  - proximité avec les usages (mathématique, transactionnel, schématique, etc
- Il faut une traduction, et donc un programme traducteur.

Compilateur ou Interpréteur ?

- Un interpréteur simule une *machine virtuelle* dont le langage considéré est le langage machine.
- Il est en général plus facile de fabriquer un interpréteur qu'un compilateur, car il est plus facile d'exécuter un calcul que de fabriquer le code qui l'exécute.
- On utilise en général des schémas mixtes :
  - Un compilateur vérifie le code de haut niveau et le transcrit en binaire, plus compact.
  - Un interprète exécute le code binaire.
- Exemples : Java, Ocaml.

Comment marche un compilateur



- Le code source est analysé suivant une grammaire formelle et codé en binaire (représentation intermédiaire).
- On vérifie que le programme est sémantiquement correct. Exemple : un identificateur est utilisé conformément à sa déclaration.
- On applique des transformation successives qui rapprochent le programme de haut niveau d'un programme en langage machine. Exemples :
  - Boucle $\Rightarrow$ combinaison de tests et de `goto`.
  - Eclatement des expressions complexes : code à un opérateur par instruction.
- Lorsque le code est suffisamment proche du langage machine, on le transcrit en langage d'assemblage.
- Au passage, on essaie d'améliorer le programme (optimisation).

Pourquoi un cours de compilation ?

- C'est un sujet de recherche actif, parce que les processeurs modernes devienennt de plus en plus complexes, et demandent des compilateurs de plus en plus sophistiqués.
- En particulier les processeurs *Dual Core* et multicore posent un problème non résolu : paralléliser un programme arbitraire.
- Les compilateurs les plus utilisés (`gcc`, `javac`, `...` sont des logiciels libres.
- Beaucoup de problèmes se résolvent élégamment en définissant un langage spécialisé et en écrivant son compilateur (exemple : la synthèse de matériel).

Langages formels, I

- Un langage est un ensemble de conventions entre un émetteur et un récepteur qui leur permet de se comprendre.
- Emetteur : le programmeur. Récepteur : le compilateur.
- Comme le récepteur n'est qu'un programme, le langage doit être spécifié de façon précise : lanage *formel*.
- On distingue :
  - La *syntaxe* (ou grammaire) : quel sont les programmes corrects.
  - La *sémantique* : quels sont les programmes qui on un sens.
- Analogie avec les langues naturelles.

Langages formels, II

- La syntaxe peut être entièrement formalisée à l'aide d'automates mathématiques ou de grammaires formelles.
- Il existe des outils permettant de traduire une grammaire formelle en un programme d'analyse grammaticale.
- La formalisation de la sémantique est beaucoup plus difficile.
- Il existe des sémantiques formelles (dénotationnelles, opérationnelles, axiomatiques, etc, voir les cours correspondants) mais aucun outils permettant de les exploiter automatiquement.

Langages formels, III

- Un langage formel est un sous ensemble de tous les mots possibles sur un alphabet donné A. Un langage peut être infini.
- Une grammaire est une représentation finie d'un langage.
- Une grammaire formelle se définit à l'aide d'un alphabet auxiliaire N (les non terminaux) et de *règles de réécriture* $a \longrightarrow b$ où a et b sont des mots sur l'alphabet $A \cup N$.
- On passe d'un mot u à un mot v à l'aide de la règle $a \longrightarrow b$ en trouvant dans u une occurence de a et en la remplaçant par b.
- Le langage associé est l'ensemble des mots terminaux engendrés par réécriture à partir d'un non terminal Z (l'axiome).

Langages formels, IV : classification

- $X, Y, Z, \ldots$ sont des symboles terminaux,
- $u, v, w, \ldots$ sont des mots terminaux
- $a, b, c \ldots$ sont des mots arbitraires.
- Grammaires régulières : règles $X \longrightarrow u$ ou $X \longrightarrow Yv$.
- Grammaires algébriques ou hors contexte : règles $X \longrightarrow a$.
- Grammaires sensibles au contexte : règles $aXb \longrightarrow acb$.
- Grammaires générales : pas de restrictions sur les règles.

Seules les deux premières catégories engendrent des langages décidables.

Exemple : la grammaire des identificateurs

$$\begin{array}{llll} \mathcal{I} \to a, & \ldots & \mathcal{I} \to z \\ \mathcal{I} \to A, & \ldots & \mathcal{I} \to Z \\ \mathcal{I} \to \mathcal{I}a, & \ldots & \mathcal{I} \to \mathcal{I}z \\ \mathcal{I} \to \mathcal{I}A, & \ldots & \mathcal{I} \to \mathcal{I}Z \\ \mathcal{I} \to \mathcal{I}0, & \ldots & \mathcal{I} \to \mathcal{I}9 \end{array}$$

C'est une grammaire régulière.

Expressions arithmétiques : une grammaire simplifiée

$$
\begin{aligned}
\mathcal{E} &\rightarrow \mathcal{I} \\
\mathcal{E} &\rightarrow \mathcal{N} \\
\mathcal{E} &\rightarrow (\mathcal{E} + \mathcal{E}) \\
\mathcal{E} &\rightarrow (\mathcal{E} * \mathcal{E})
\end{aligned}
$$

C'est une grammaire hors contexte. Si l'on omet les parenthèses, la grammaire devient ambigüe. Il y a plusieurs suites de réécritures pour engendrer le même mot.

Outillage

- Une grammaire peut être utilisée soit comme générateur, soit comme analyseur.
- Un mot étant donné, reconstituer la suite des productions qui permet de l'engendrer à partir de l'axiome, ou dire qu'il n'est pas dans le langage.
- Ceci n'est possible que pour les deux premiers types de grammaires.
- Pour des raisons d'efficacité, on emploie les deux types, bien que les langages réguliers soient inclus dans les langages hors contexte.
- Mais la puissance des grammaires hors contexte est insuffisante pour exprimer toutes les règles. On doit programmer des contrôles supplémentaires.

Lec et Yacc

- Analyse lexicale = découpage en *mots* à l'aide de grammaires régulières.
- Analyse syntaxique = découpage en phrases à l'aide d'une grammaire hors contexte.
- On peut inventer des langages formels permettant de décrire les grammaires :
  - liste de productions,
  - conventions lexicales pour distinguer les terminaux, les non terminaux et les opérateurs propres
- On peut écrire des compilateurs pour ces langages : à partir de la grammaire, ils engendrent un analyseur.
- Les deux plus connus sont Lex (pour les grammaires régulières) et Yacc (pour un sous ensemble des grammaires hors contexte).
- Il en existe de nombreuses implémentations qui diffèrent par le langage cible (C, Ocaml, Java, ...) et le statut juridique.

Le langage Ocaml, I

Le langage Ocaml est particulièrement bien adapté pour l'écriture de compilateurs. C'est un langage mixte – fonctionnel et impératif – à typage fort.

- Structure de données récursives :

```
type operon = {op : string;
               args : expression list;
              }
and expression = None
               | Var of string
               | Const of int
               | SubExpression of expression
;;
```

Le langage Ocaml, II

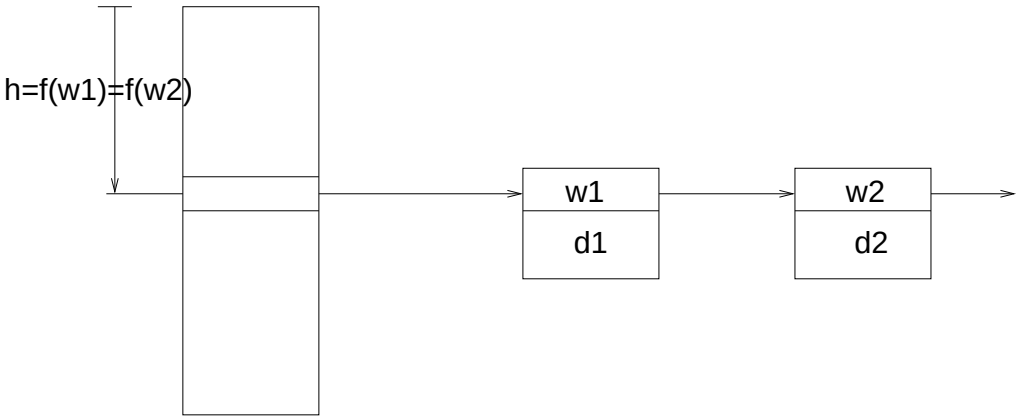
- *pattern matching*

```
match x with
    None -> ...
  | SubExpression e -> ... (List.fst e.args) ...
  | _ -> ()
```

- récursion, listes, chaînes de caractères, tableaux, piles, FIFO, fonctions anonymes, sérialisation ...
- objets et foncteurs : utilisation délicate

Le langage Ocaml, III

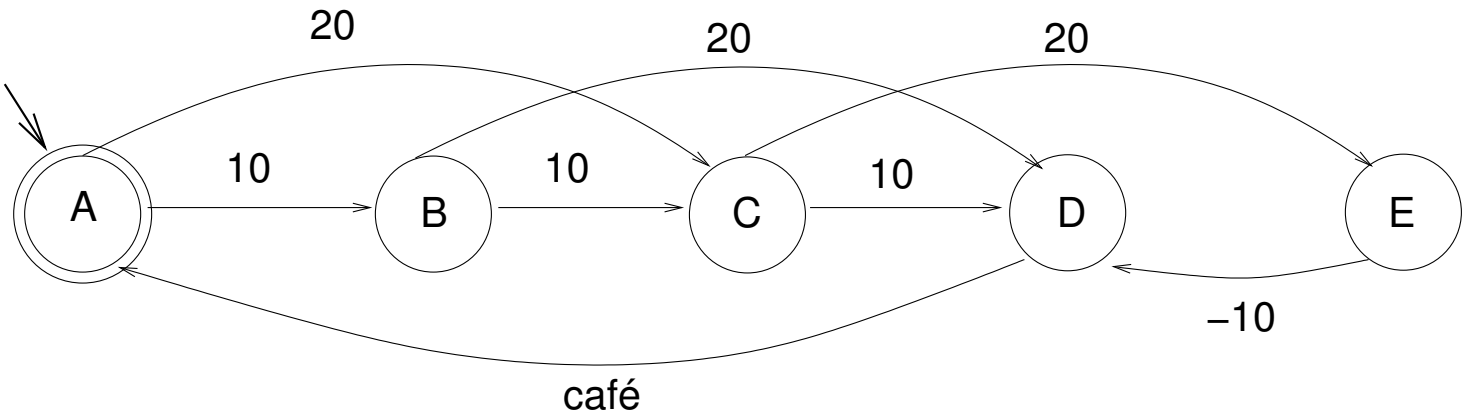
Tables hashcodées.



Interface :

```
let h = Hashtbl.create 127;
Hashtbl.add h w d;
let d = try Hashtbl.find h w
        with Not_found -> ....
in ...;
```

Automates finis



L'automate fini de la machine à café.

- Un alphabet d'entrée (ici $\{10, 20\}$) et éventuellement un alphabet de sortie (ici $\{-10, \mathrm{cafe}\}$).
- Un ensemble fini d'états (ici, $\{A, B, C, D, E\}$).
- Un état initial, A qui ici est également terminal.
- Une relation de transition :

$$A.10 \to B \quad A.20 \to C \quad B.10 \to C \quad B.20 \to D$$
$$C.10 \to D \quad C.20 \to E \quad D \to A/\mathrm{cafe} \quad E \to D/-10$$

Définitions

- Un mot est accepté par l'automate s'il existe un chemin étiqueté par les lettres du mot allant de l'état initial à un état terminal.
- L'ensemble des mots acceptés est le langage reconnu par l'automate.
- Le mot de longueur nulle se note ϵ.
- Un automate fini est *déterministe* si la relation de transition est une fonction.
- Un automate peut avoir des ϵ-transitions ; il est alors indéterministe.
- Un état est *accessible* s'il existe un chemin passant par cet état et allant de l'état initial à un état terminal. On peut éliminer les états non accessibles.
- Un automate est complet si pour tout état il y a au moins une transition par lettre de l'alphabet. En général, on complète un automate par des transitions vers un état d'erreur.

Notations

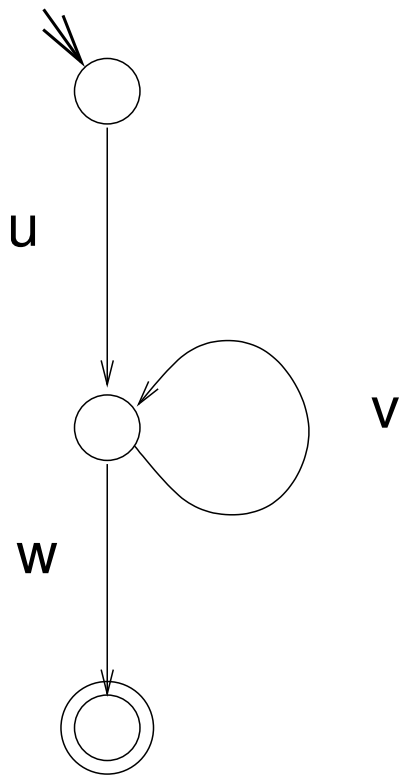
Un automate fini est un tuple $\{A, Q, i, T, \delta\}$.

- A l'alphabet (d'entré).
- Q l'ensemble (fini) des états.
- $i \in Q$ l'état initial.
- $T \subseteq Q$ l'ensemble des état terminaux.
- $\delta \subseteq Q \times A \times Q$ la relation de transition.
- On note $p.a \longrightarrow q$ pour $\langle p, a, q \rangle \in \delta$.
- Si w est un mot sur A de longueur n, on note $p.w \longrightarrow q$ pour

$$p.w_1 \longrightarrow q_1, \ldots, p_{n-1}.w_n \longrightarrow q_n = q.$$

En particulier, $p.\epsilon \longrightarrow p$.

Lemme de pompage



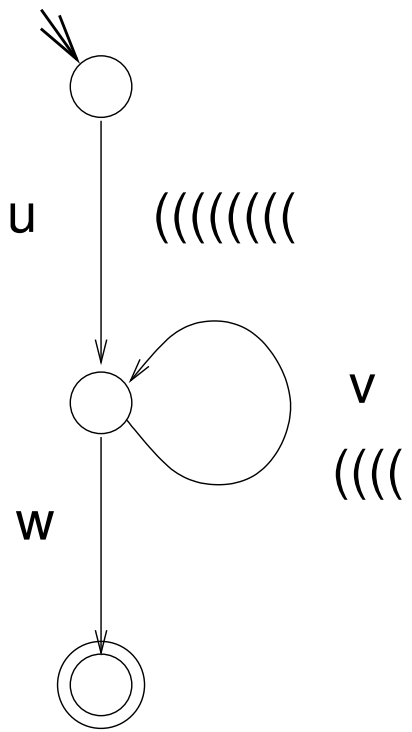
- Si le graphe de l'automate est acyclique, l'ensemble des mots est fini.

  Dans un graphe acyclique on peut définir un plus long chemin, et l'ensemble des mots de longueur bornée est fini.

- On suppose maintenant que le graphe contient un cycle accessible. Il existe trois mots u allant de l'état initial en un point du cycle, v le mot du cycle et w allent du point initial du cycle à l'état terminale. Alors tous les mots $u.v^n.w$ sont dans le langage de l'automate (lemme de pompage).
- Un automate fini "ne sait pas compter".

Le langage de Dick

Un exemple de ce qu'il est impossible de faire avec un automate fini : vérifier qu'un système de parenthèses est bien équilibré.



- On choisit un mot $u.v$ commençant par plus de parenthèses ouvrantes que l'automate n'a d'état.
- On revient donc nécessairement à un état déja vu.
- Soit w un mot allant de cet état à l'état terminal.
- Si $u.w$ est équilibré, $u.v.w$ ne peut pas être équilibré.

Expression régulières

Syntaxe abstraite :

$$\mathcal{E} \to \mathcal{L} \quad \mathcal{E} \to \mathcal{E}.\mathcal{E}$$
$$\mathcal{E} \to \mathcal{E}^* \quad \mathcal{E} \to \mathcal{E}|\mathcal{E}$$

La syntaxe est *abstraite* parce que l'on suppose que les termes sont des arbres et non des chaînes de caractères.

- $\mathcal{L}$ est l'ensemble des lettres de l'alphabet.
- Le point représente la concaténation.
- La barre représente l'alternation.
- L'étoile représente l'itération.
- $a.(a+b)^*$ se lit "un a suivi d'un nombre arbitraire de a ou de b" et désigne donc les mots sur l'alphabet $\{a, b\}$ qui commencent par un a.
- on va montrer que les automates finis et les expressions régulières definissent la même classe de langages.

Identités remarquables

- La concaténation est associative.
- Elle distribue par rapport à l'alternation.
- L'alternation est associative et commutative.
- $X^* = \epsilon | (X.X^*)$.

Equivalence, I

Etant donné un automate fini, trouver l'expression rationelle équivalente.

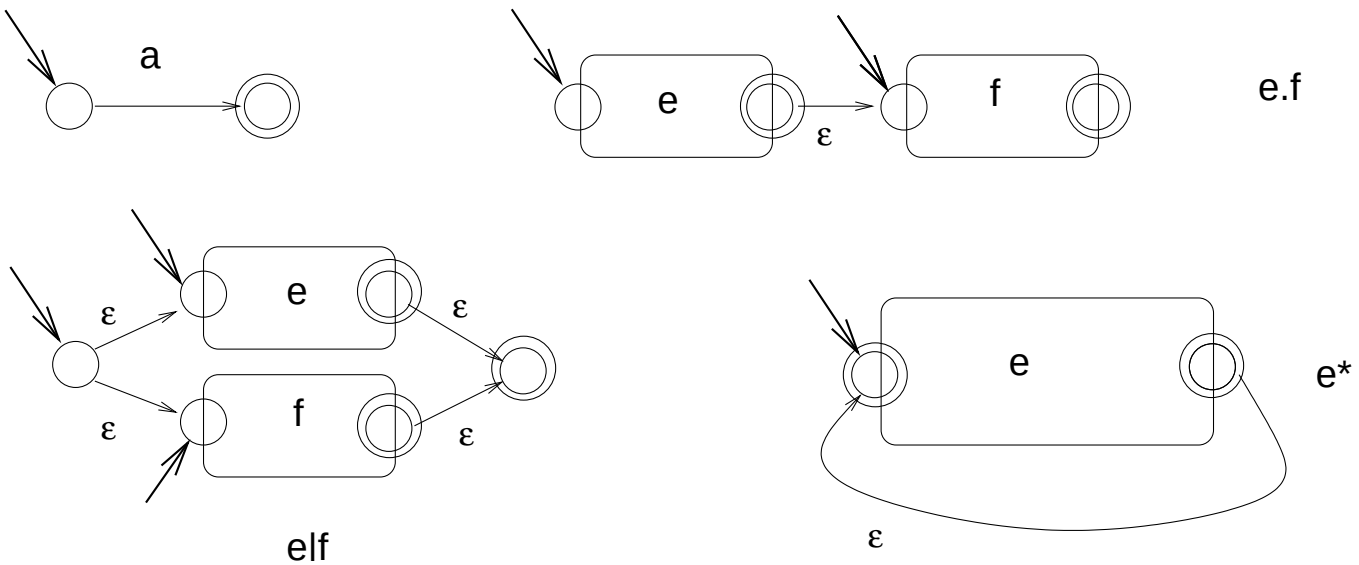
- Pour chaque transition $p.a \longrightarrow q$, écrire une équation $p = aq$.
- Résoudre par la méthode de Gauss.
- A chaque étape, les équations sont de la forme

$$p = e_1 q_1 | e_2 q_2 | \ldots | e_n q_n$$

  où les e_i sont des expressions rationelles.
- Si $p = q_k$, alors $p = e_k^*.(e_1 q_1 | \ldots | e_n q_n)$.
- Si p est différent de tous les q_k, alors on peut l'éliminer en le remplaçant par sa valeur dans toutes les équations.
- L'algorithme se termine quand tous les états restants sont terminaux.
- La complexité du résultat peut être exponentielle.

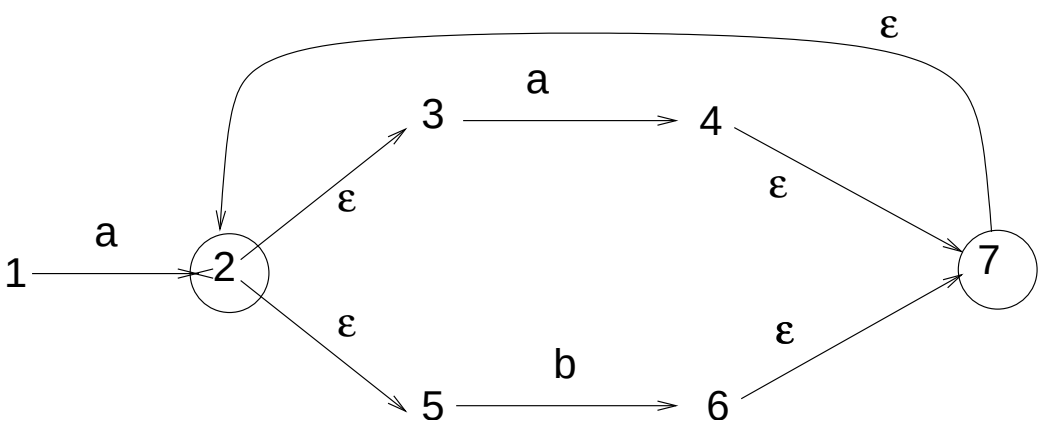
Equivalence, II

Etant donné une expression rationelle, trouver l'automate fini équivalent.
On procède par induction sur la structure de l'expression rationelle.



Exemple, I

$$a.(a|b)^*$$

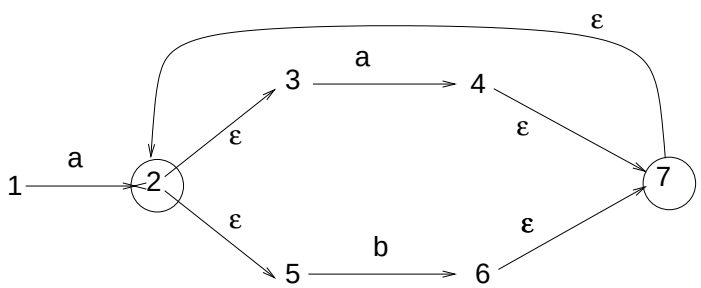


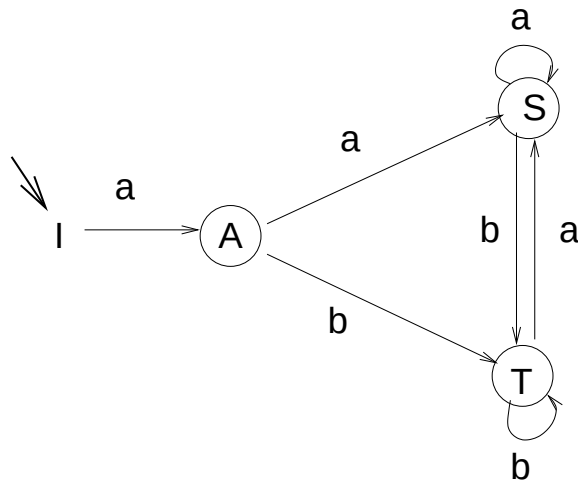
L'automate obtenu n'est pas déterministe.

Détermination

- On construit un autre automate dont les états sont les ensembles d'états de l'automate initial.
- Un état P est ϵ-clos ssi $x \in P, x.\epsilon \rightarrow y \Rightarrow y \in P$. On ne considère que les états ϵ-clos. Il existe une opération notée $\mathcal{E}$ qui calcule la ϵ cloture.
- L'état initial est $\mathcal{E}(i)$.
- Un état est terminal s'il contient un état terminal de l'automate initial.
- On crée une transition étiquetée par a entre P et $\mathcal{E}(\{y \mid \exists x \in P : x.a \rightarrow y\})$.

Exemple, II





$$I = \{1\},\ I.a \rightarrow \{2,3,5\} = A,\ A.a \rightarrow \{4,7,2,3,5\} = S$$

$$A.b \rightarrow \{6,7,2,3,5\} = T,\ S.a \rightarrow S,\ S.b \rightarrow T$$

$$T.a \rightarrow S,\ T.b \rightarrow T$$

L'automate résultant n'est pas minimal.

Minimisation : Equivalence de Nerode

- Langage engendré par un sommet :
  $\mathcal{L}(p) = \{w \mid p.w \rightarrow q, q \in T\}$
- Equivalence de Nerode : $p \approx q \Leftrightarrow \mathcal{L}(p) = \mathcal{L}(p)$.
- C'est une équivalence, comme toute relation image inverse de l'égalité.

Automate minimal d'un automate déterministe

- On considère l'automate quotient dont les états sont $Q/\approx$ ...
- Il y a un arc de $\overline{p}$ vers $\overline{q}$ étiqueté par a, si il existe $p \in \overline{p}, q \in \overline{q}, p.a \rightarrow q$.
- Si l'automate de départ est déterministe, l'automate quotient l'est aussi.
- L'automate quotient engendre le même langage que l'automate initial.
- L'automate quotient est minimum.

Calcul de l'équivalence de Nerode par approximation successives

On pose :

$$\mathcal{L}_k(p) = \{w \mid p.w \rightarrow q, q \in T, |w| \leq k\}$$

$$p \approx_k q \Leftrightarrow \mathcal{L}_k(p) = \mathcal{L}_k(q)$$

On a :

$$\mathcal{L}_k(p) = \mathcal{L}_{k-1}(p) \cup \bigcup_a a.\mathcal{L}_{k-1}(p.a)$$

De plus, la décomposition est unique. Il s'en suit

$$p \approx_k q \Leftrightarrow p \approx_{k-1} q \ \ \& \ \bigwedge_a p.a \approx_{k-1} q.a$$

Calcul de l'équivalence de Nerode, II

- L'équivalence $\approx_0$ divise l'ensemble des état en 2, les terminaux et les non terminaux.
- Connaissant $\approx_{k-1}$ on peut calculer $\approx_k$ par la formule ci-dessus.
- Si $\approx_k = \approx_{k-1}$ on a atteint un point fixe.
- Sinon le nombre de classe a augmenté, et comme il est borné par le nombre d'états, on finiit toujours par s'arréter.
- On peut accélérer l'algorithme :
  - On ne calcule $p \approx_k q$ que si $p \approx_{k-1} q$.
  - On ne calcule pas $p \approx_k q$ si on peut le déduire par transitivité de couples déja calculés (algorithme de Fisher-aller).

Lex, flex, ocamllex

- Classification : on se donne une chaîne de caractères et une liste d'expressions rationelles.
- Partitionner la chaîne en "jetons" dont chacun est une instance de l'une des expressions rationelles.
- Il ne doit pas y avoir de portion de chaîne non classifiée.
- Pour chaque jeton, indiquer le "nom" de l'expression rationelle en cause et le texte du jeton (sauf s'il n'y a aucune ambiguité).
- Méthode : construire et minimiser l'automate qui reconnait $(e_1|e_2|\ldots|e_N)^*$ et signaler un jeton chaque fois que l'on passe par l'état terminal.

Expressions régulières augmentées

- Le “programme” soumis à lex est de la forme :

$$\begin{array}{ll} & e_1\{j_1\} \\ | & e_2\{j_2\} \\ | & \ldots \end{array}$$

- Les expressions rationelles e_i utilisent des raccourcis. Par exemple, `'a'-'z'` représente `'a' | 'b' | 'c' ... | 'z'`.
- Le jeton est décrit par un bout de code dans le langage hote.
- L'automate est construit, déterminisé, minimisé et implémenté sous forme de tables.
- Attention, la taille de l'automate croit très vite avec la longueur des expressions rationelels e_i. Utiliser des astuces (table de mots clef)