Analyse lexicale

Mirabelle Nebut

Bureau 332 - M3
mirabelle.nebut at lifl.fr

2011-2012

Analyse lexicale

Rappels
- Vocabulaire, mots, langages
- Langages réguliers
- Expressions régulières
- Automates à nombre fini d'états

Comment se comporte un analyseur lexical ?

Construction d'un analyseur lexical

Spécification d'un analyseur lexical
- Bases théoriques (pourquoi c'est possible et ça marche)
- Des expressions aux descriptions régulières

La partie analyse lexicale est essentiellement tirée de [Wilhelm et Maurer].

À propos de ces rappels

Cette partie du cours est à travailler pour le TD1.

Pour ceux qui découvrent : appeler à l'aide si brasse coulée.

Référence

La partie rappels est essentiellement tirée du poly de Daniel Herman, Théorie des Langages et Compilation, poly 111 de l'ex IFSIC (Institut de Formation Supérieure en Informatique et Communication), Université de Rennes1.

Langages

Le français et Java sont des langages.

La théorie du langage fournit des modèles pour :

- décrire un langage

  formalisme de description

- fabriquer une machine qui sait dire si un texte appartient à un langage donné

  formalisme exécutable pour la reconnaissance

Langage, vocabulaire et mots

Un "texte" Java est obtenu en mettant les uns derrière les autres des briques de base (mots-clés, identificateurs, etc).

Le langage = tous les textes

Un mot du langage = un texte

Un symbole du vocabulaire = une brique de base

Vocabulaire et mots

Vocabulaire	**Mots**
{a,b}	aaaa, ba
{public, Toto, class, {, }}	public class Toto { } class class

Un vocabulaire ou alphabet V ou Σ est un ensemble fini de symboles.

Un mot sur un vocabulaire V est une séquence finie d'élements de V.

Mots

On note $| m |$ la longueur du mot m, c'est à dire le nombre de symboles qui le composent.

Sur $\{a, b\}$, $| aab | = 3$

On note ϵ le mot vide, tel que $| \epsilon | = 0$.

On note V^* l'ensemble des mots sur V.

Concaténation de mots

L'opération de concaténation est définie par :

$$\begin{array}{rcl} \bullet : V^* \times V^* & \rightarrow & V^* \\ m_1, m_2 & \mapsto & \text{le mot commençant par les symboles de } m_1 \\ & & \text{suivi par les symboles de } m_2 \\ & & \text{en préservant leur ordre} \end{array}$$

On écrit $m_1 m_2$ pour $m_1 \bullet m_2$.

On écrit a^n le mot composé de n occurrences de a.

On a donc $a^0 = \epsilon$

Et retour aux langages

Un langage L sur V est une partie de V^*.

$L \in \mathcal{P}(V^*)$

Un langage est un ensemble de mots, fini ou infini.

$\{a^n |$ n est pair$\}$: infini
$\{a^n |$ n est pair et $n \leq 100\}$: fini

À quels langages s'intéresse-t-on ?

Pas tous les langages !

Ex : entités lexicales d'un langage de programmation
V=alphabet anglais $\cup$ { (,), {, ;, ...}.

On veut pouvoir exprimer :

- un langage fini réduit à un symbole : { `i` }, { `f` }
- la notion de concaténation : { `i` } puis { `f` }
- la notion de répétion (non bornée) : un entier = des chiffres répétés
- la notion de choix : un entier signé commence par + ou -

Opérateurs sur langages : choix et concaténation

notion de choix $\rightarrow$ union ensembliste de langages
$L_1 \cup L_2 = \{m \mid m \in L_1 \vee L_2\}$

notion de concaténation $\rightarrow$ produit de concaténation sur langages
$L_1 L_2 = \{m_1 m_2 \mid m_1 \in L_1 \wedge m_2 \in L_2\}$

Opérateur de répétition

notion de répétion / itération $\rightarrow$ fermeture d'un langage

La fermeture L^* de L contient les mots des langages :

- $L^0 = \{\epsilon\}$
- L^1 : mots obtenus en concaténant un mot de L^0 avec un mot de L, donc $L^1 = L^0 L$
- ... $L^i = L^{i-1} L$
- ...

On a donc $L^* = \bigcup_{i \geq 0} L^i$

Une fermeture est une union infinie, et produit toujours un langage infini.

Langages réguliers

Les langages réguliers sont ceux construits à partir de ces opérateurs.

Les langages réguliers sont les langages construits à partir de langages finis, en utilisant un nombre fini de fois des opérations d'union, de produit, et de fermeture de langages.

régulier = rationnel

Les langages réguliers, formellement

Les langages réguliers sur V sont définis récursivement :

- $\emptyset$, $\{\epsilon\}$ sont des langages réguliers ;
- pour tout $a \in V$, $\{a\}$ est un langage régulier ;
- si L_1 et L_2 sont des langages réguliers sur V alors $L_1 \cup L_2$, $L_1 L_2$ et L_1^* sont aussi des langages réguliers sur V ;
- il n'y a pas d'autre langage régulier sur V.

Expressions régulières : exemple

- Le chiffre 1 : 1
- Un chiffre quelconque : 0|1|2|3|4|5|6|7|8|9
- Un nombre :

  (0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*
- Un exposant éventuel :

  ϵ | E(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)

Expressions régulières

Description des langages réguliers sur V : expr régulières (er) sur V

- $\emptyset$ est une er qui décrit le langage $\emptyset$;
- ϵ est une er qui décrit le langage $\{\epsilon\}$;
- a (pour $a \in V$) est une er qui décrit le langage $\{a\}$.

Si e_1 et e_2 sont des er sur V, décrivant les langages $L(e_1)$ et $L(e_2)$, alors les expressions suivantes sont aussi des er :

- $e_1 \mid e_2$ décrit le langage $L(e_1) \cup L(e_2)$;
- $e_1 e_2$ décrit le langage $L(e_1)L(e_2)$;
- ${e_1}^*$ décrit le langage $L(e_1)^*$.

Il n'y a pas d'autre expression régulière.

Opérateurs dérivés

Les bibliothèques d'expressions régulières fournissent d'autres constructions :

- répétition finie, pour n fixé : $a^3 = aaa$
- construction du type "tout sauf ces symboles"
- construction du type "au moins une fois" : $a^+ = a(a*)$
- construction du type « là ou pas » : $a? = a|\epsilon$
- etc

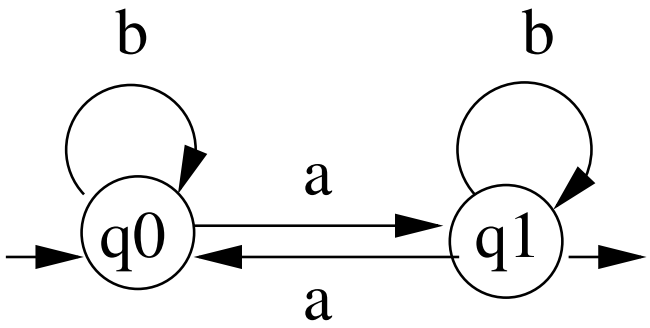
À propos des automates

Les automates à nombre fini d'état reconnaissent exactement les langages réguliers.

C'est pourquoi on les utilise dans ce cours.

Plus largement, les machines à états sont un moyen de modéliser les systèmes à état très répandu (cf statecharts UML).

Intuition



Sur $V = \{a, b, c\}$: reconnaît le langage des mots sans c qui contiennent un nombre impair de a.

Notion de mot sur un ruban + tête de lecture

Exemples de fonctionnement pour les 3 cas d'arrêt possible :

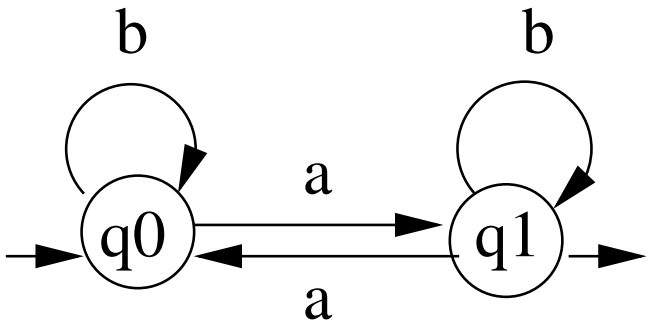
- mot $m_1 = ab$: acceptation
- mot $m_2 = aab$: rejet, arrêt ds état pas final
- mot $m_3 = bca$: rejet, blocage

AFD, définition

Un automate à nombre fini d'états déterministe (AFD) est un tuple $A = (V, Q, \Delta, q_0, F)$ dans lequel :

- V est l'alphabet d'entrée ;
- Q est un nombre fini d'états ;
- $q_0 \in Q$ est l'état initial ;
- $F \subseteq Q$ est l'ensemble des états finaux ;
- Δ est une fonction de transition de $Q \times V$ vers Q.

Fonction de transition, exemple

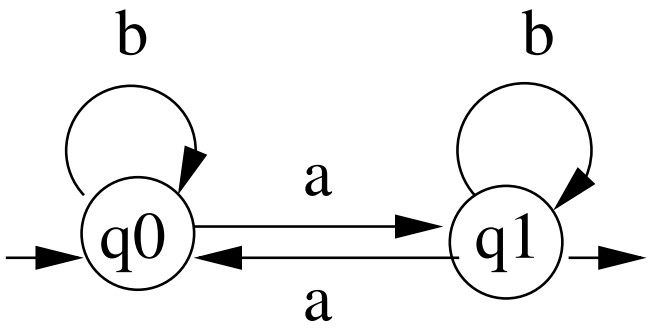


Δ	**a**	**b**
q_0	q_1	q_0
q_1	q_0	q_1

Notion de configuration, intuition

Configuration = situation courante de l'automate
= état courant + mot restant à lire

Ex config initiale : (q_0, ab)
Ex config finale : (q_1, ϵ)
Ex config intermédiaire : (q_1, b)



Configuration, définition

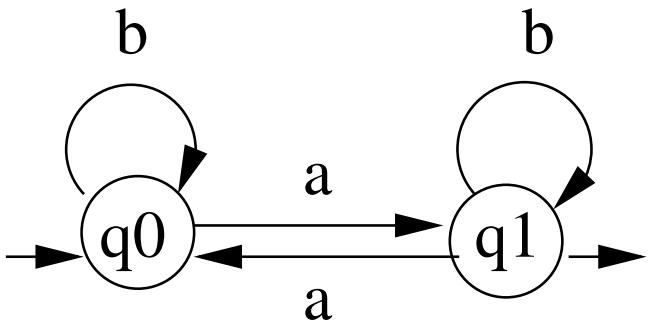
Soit $A = (V, Q, \Delta, q_0, F)$ un AFD :

- un couple (q, w) avec $q \in Q$ et $w \in V^*$ est une configuration de A ;
- (q_0, w) est une configuration initiale ;
- (q_f, ϵ) avec $q_f \in F$ est une configuration finale.

Relation de dérivation, intuition et définition

Une transition d'un automate relie 2 configurations successives.

$(q_0, ab) \vdash_A (q_1, b)$ et $(q_1, b) \vdash_A (q_1, \epsilon)$

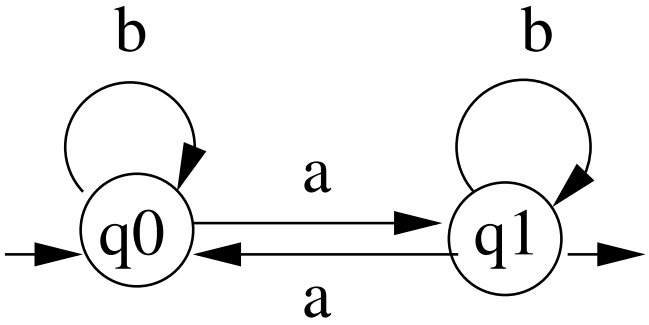


On définit la relation de dérivation, notée $\vdash_A$:

$$(q, aw) \vdash_A (q', w) \text{ ssi } \Delta(q, a) = q' \text{ avec } a \in V$$

Séquence de transitions

L'automate effectue des séquences de transitions.



Ex : $(q_0, ab) \vdash_A^* (q_1, \epsilon)$

On note $\vdash_A^*$ la clôture réflexive et transitive de $\vdash_A$
$(q_s, m_1) \vdash_A^* (q_c, m_2)$ ssi :

- soit $q_c = q_s$ et $m_1 = m_2$
- soit $(q_s, m_1) \vdash_A (q_i, m_3)$ et $(q_i, m_3) \vdash_A^* (q_c, m_2)$

Langage accepté par un automate

Le langage accepté par A est :

$$L(A) = \{m \in V^* \mid (q_0, m) \vdash_A^* (q_f, \epsilon) \text{ avec } q_f \in F\}$$

Vers des automates non déterministes : AFND

On n'a pas encore dit qu'un langage régulier peut se reconnaître par un AFD.

De fait, quand on essaie de faire correspondre les opérations de produit, union, etc à des combinaisons simples d'automates, on tombe sur des automates non déterministes : AFND.

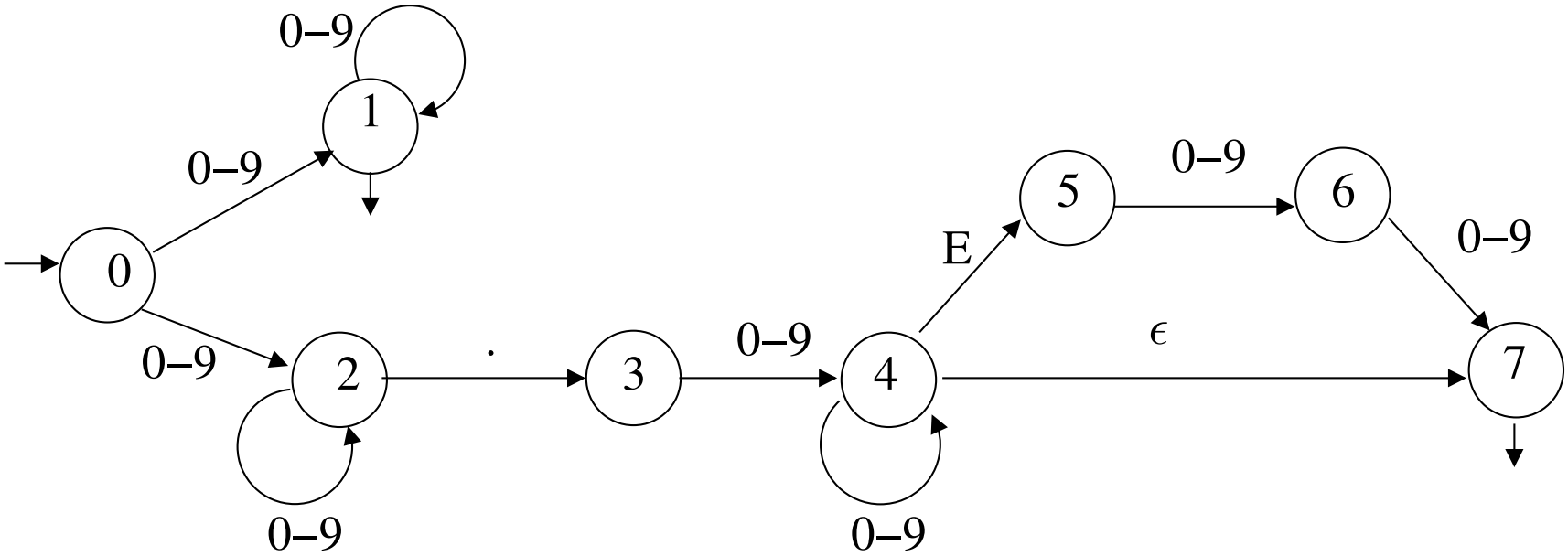
Non-déterminisme = choix de transiter dans différentes configurations.

Un AFND accepte un mot si, parmi toutes les séquences de transitions possibles pour ce mot, il en existe une qui aboutit dans un état final.

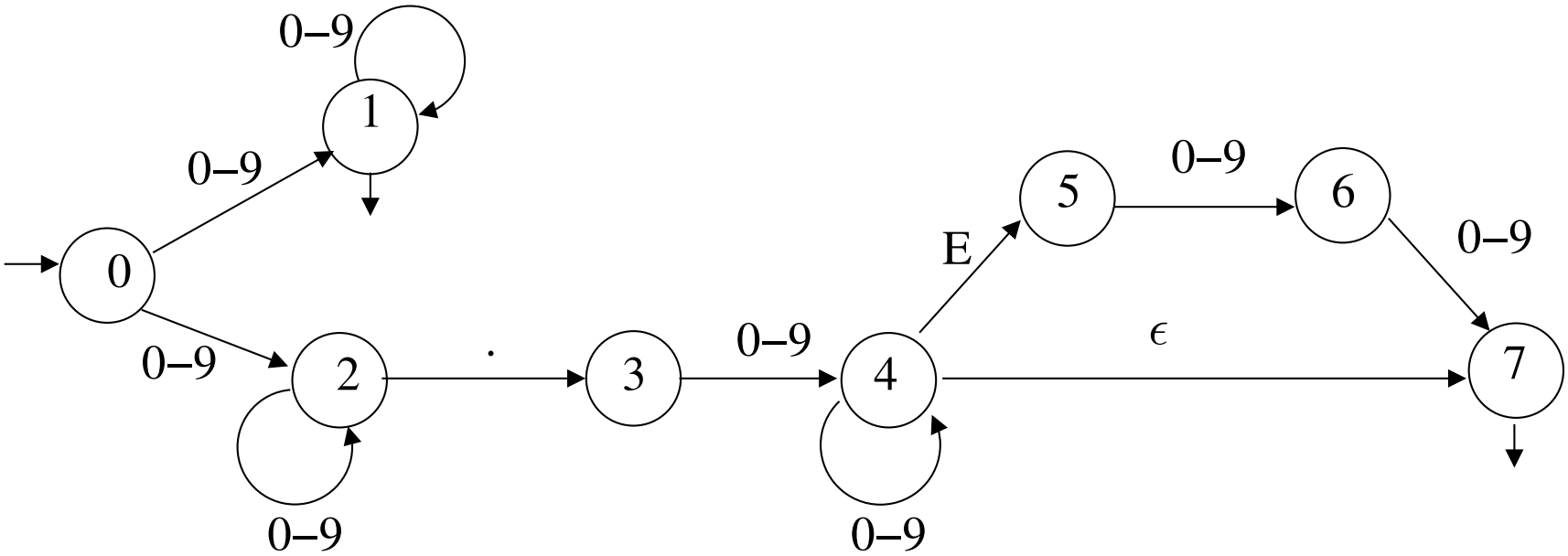
Sources de non-déterminisme

- soit plusieurs transitions possibles pour un symbole $a \in V$.
- soit ϵ-transition

Ex des constantes entières ou réelles :



Fonction de transition non déterministe



$\Delta(q_0, 9) = \{q_1, q_2\}$: q_0 a plusieurs successeurs par le symbole 9.

Pour un AFND, Δ est une fonction de transition de $Q \times (V \cup \{\epsilon\})$ vers Q ou $\mathcal{P}(Q)$.

ϵ-transition

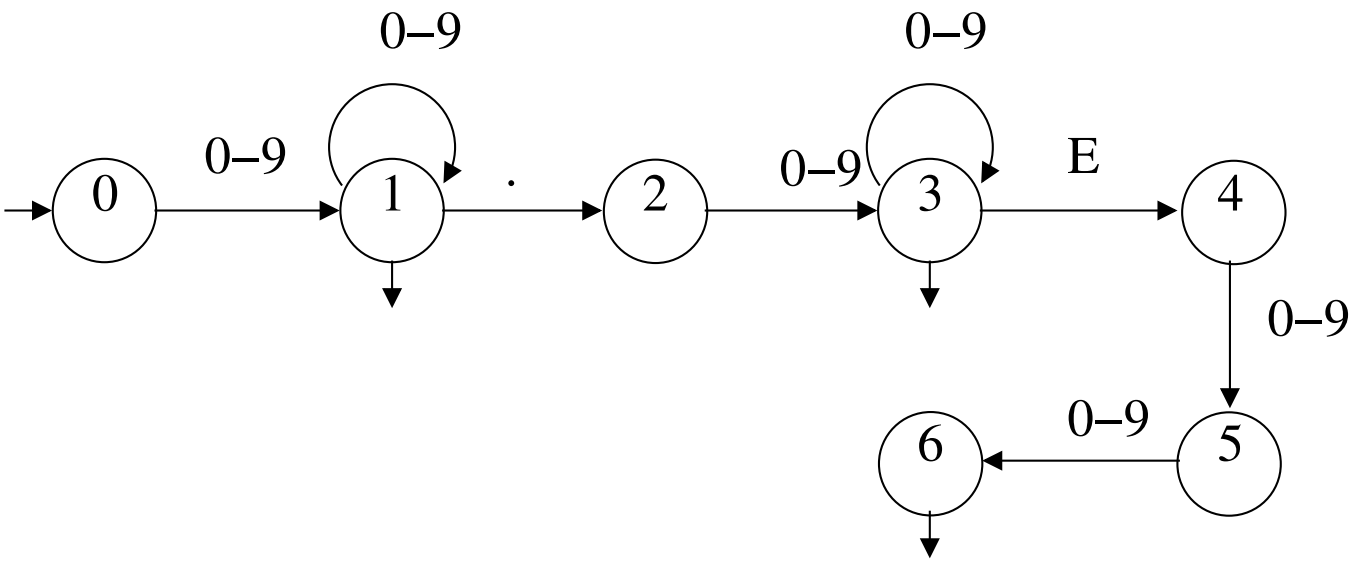
Une transition étiquetée par ϵ ne fait pas bouger la tête de lecture.

Elle peut être toujours déclenchée, qq soit la tête de lecture.

Si $q_2 \in \Delta(q_1, \epsilon)$, alors $(q_1, m) \vdash (q_2, m)$

Déterminisation

Il existe un algorithme qui calcule à partir d'un AFND A un AFD A' tel que $L(A') = L(A)$.



Retour aux langages réguliers

Un langage est régulier ssi il est reconnu par un AFND.

On a donc l'équivalence entre langages réguliers, expressions régulières et automates à nombre finis d'états.

Rappels

- Premier module du compilateur, au contact du texte source ;
- décompose la suite de caractères en une suite de symboles.
- peut filtrer les symboles non pertinents pour la suite.

Ex :

```
program pgm;
int x,y;
```

Cas d'un scanner jouant le rôle de filtre.

symbole reconnu	classe de symbole retournée
`program`	PROGRAM
espace	
`pgm`	IDENT
`;`	FININSTR
saut de ligne	
`int`	DECLINT
espace	
...	...

Exemple pour ce cours

On veut reconnaître les classes de symboles (ou unités lexicales) :

- `ENTIER` : les constantes entières (`3`, `11110`) ;
- `REEL` : les constantes réelles, avec un exposant optionnel à exactement deux chiffres (`3.14`, `22.67E01`) ;
- `IDENT` : les identificateurs à la Java sans `_` (`x`, `m5`, `j2sdk`) ;
- `IF` : le mot-clé `if` ;
- `DIESE` : le dièse `#`.

Sous quelle forme produire les symboles ?

Deux approches possibles pour retourner les symboles :

- un appel à une méthode qui produit une liste de `Token` ;
- des appels répétés à une méthode qui retourne un symbole à la fois (appelée par l'analyseur syntaxique) ←

```
Token next_token()
```

Ex : pour l'entrée `"toto#1.2if"`

1. `next_token()` retourne `IDENT("toto")` ;
2. `next_token()` retourne `DIESE` ;
3. `next_token()` retourne `REEL("1.2")` ;
4. `next_token()` retourne `IF`.

Levée des ambiguïtés

Pour l'entrée `"ti80"` :

- `IDENT("ti") ENTIER("80")` ?
- `IDENT("ti80")` ?

Pour l'entrée `"if#"` :

- `IF DIESE` ?
- `IDENT DIESE` ?

Reconnaissance du plus long préfixe

= tant qu'on peut continuer, on continue.

Pour l'entrée `"ti80"` :

$\Rightarrow$ `IDENT("ti80")`

et non `IDENT("ti") ENTIER("80")`

Prise en compte de priorités

Pour l'entrée `"if#"` :

- `"if"` respecte à la fois la description des symboles `IF` et `IDENT` ;
- le mot-clé `IF` est prioritaire.

$$\Rightarrow \texttt{IF DIESE}$$

On doit pouvoir indiquer des priorités.

Marche arrière en cas d'échec (1)

Soit l'entrée `"2.3E5xy"`, et `c` le caractère courant :

- `c=getChar()` : `'2'` ⇒ reconnaissance d'un `ENTIER`
- `c=getChar()` : `'.'` ⇒ tentative pour reconnaître un `REEL`
- `c=getChar()` : `'3'` ⇒ reconnaissance d'un `REEL`
- `c=getChar()` : `'E'` ⇒ tentative pour reconnaître un `REEL`
- `c=getChar()` : `'5'` ⇒ tentative pour reconnaître un `REEL`
- `c=getChar()` : `'x'` ⇒ `"2.3E5x"` pas un `REEL` !

`return Token.REEL("2.3")` : dernier symbole reconnu

⇒ mémorisation du dernier symbole reconnu

Marche arrière en cas d'échec (2)

- Entrée à analyser : `"2.3E5xy"` ;
- Entrée analysée : `"2.3E5x"` ;
- Entrée reconnue : `"2.3"` ;

- l'analyse reprend sur `'E'` ;
- $\Rightarrow$ il faut remettre `"E5x"` (déjà lus) dans la chaîne à analyser

```
return Token.IDENT("E5xy")
```

$\Rightarrow$ remettre les caractères lus mais pas reconnus dans le flot d'entrée

Mise en œuvre

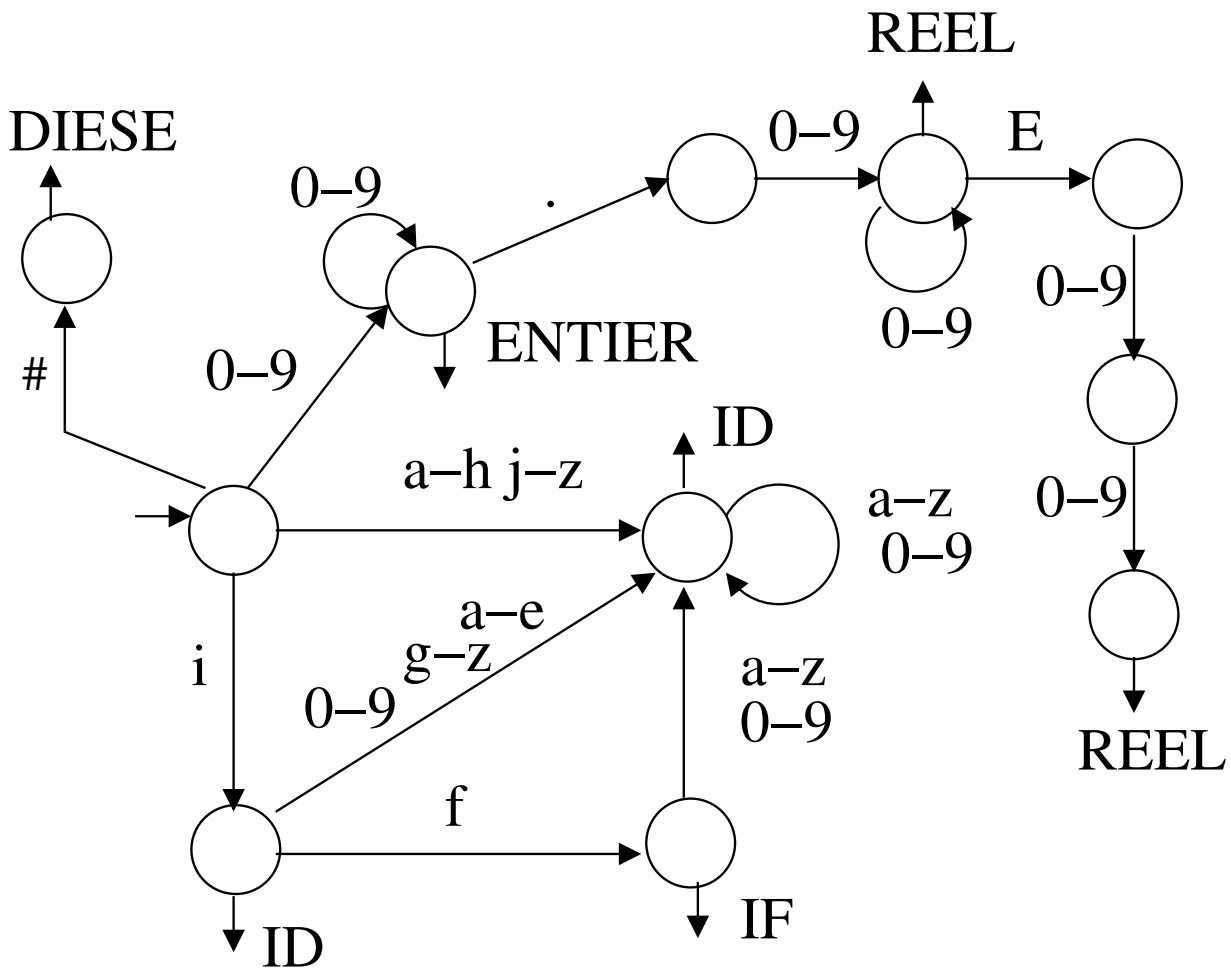
À vos claviers là comme ça tout de suite ?

Faisable. . . mais périlleux !

Plus sûr :

- construire l'automate à nombre fini d'états sous-jacent (incluant les priorités) ;
- coder cet automate :
  - reconnaissance du plus long préfixe ;
  - mémorisation du dernier symbole reconnu.
- optimiser (représentation des tables de l'automate, techniques de compression, pas dans ce cours).

Exemple d'AF



(majuscules omises, supposées incluses dans a-z)

Principes d'un an.lex. basé sur un AFD - 1

Un analyseur lexical ne fonctionne pas exactement comme un automate classique.

But de l'automate classique :

- reconnaître un langage ;
- accepter ou rejeter un mot d'entrée.

But de l'analyseur :

- saucissonner un mot d'entrée en sous-mots ;
- associer un symbole à chaque sous-mot ;
- accepter ou rejeter le mot d'entrée.

On parlera dans ce cours d'automate fonctionnant comme un analyseur lexical.

Principes d'un an.lex. basé sur un AFD - 2

États finals = états de reconnaissance d'un symbole.

Reconnaissance du plus long préfixe : tant qu'on peut transiter sur un caractère, on le fait.

Mémorisation du dernier état final traversé + sous-mot associé.

Principes d'un an.lex. basé sur un AFD - 3

Quand on ne peut pas transiter sur un caractère :

1. si état courant $=$ état final :
   - émission du symbole associé ;
   - retour dans l'état initial.
2. si état courant $\neq$ état final :
   - si $\exists$ un état mémorisé :
     - émission du symbole associé ;
     - repositionnement de la tête de lecture ;
     - retour dans l'état initial.
   - sinon erreur, rejet mot d'entrée.

Mise en œuvre

À vos claviers maintenant ?

Tout à fait faisable. . . mais :

- construction automate + codage = long et fastidieux ;
- en cas d'ajout d'un symbole, il faut tout recommencer !

Idée :

- générer automatiquement le code de l'automate ;
- à partir d'une description semi-formelle (ou spécification) du comportement de l'an.lex.

= écrire/utiliser un compilateur qui génère un module de compilateur !

Comment décrire un an. lexical (semi) formellement ?

Théorie du langage :

- description formelle du langage associé aux unités lexicales ;
- utilisation d'expressions régulières.

Génie logiciel : description (pas formelle)...

- du type des symboles ;
- du type de l'analyseur lexical ;
- du nom de la méthode `next_token()` ;
- de ce qu'il faut faire en cas d'erreur ;
- etc.

Analyse lexicale et langages réguliers

- le langage des nombres réels ;
- le langage des identificateurs à la Java ;
- le langage contenant un mot-clé ;
- etc

sont des langages réguliers.

Le langage représentant une classe de symboles est régulier.

Les langages réguliers sont clôts par union.

$\Rightarrow$ le langage reconnu à l'an.lex. est un langage régulier.

Analyse lexicale et expressions régulières

Un langage régulier peut être décrit par une expression régulière.

$\Rightarrow$ justifie l'utilisation des expressions régulières pour décrire un analyseur lexical.

Analyse lexicale et AF

Tout langage régulier est reconnu par un automate à nombre d'état fini (AF - AFD si déterministe, AFND sinon).

$\Rightarrow$ justifie l'utilisation d'un automate pour construire un an.lex.

Il existe des algorithmes :

- pour transformer une expression régulière en AFND ;
- pour déterminiser un AFND ($\Rightarrow$ AFD).

$\Rightarrow$ permet la génération automatique de code à partir des expr. reg.

De la spécification à l'analyseur

- donner une expression régulière E_i pour chaque unité lexicale ;
- indiquer éventuellement des priorités ;
- engendrer un AFND fonctionnant comme un an.lex qui reconnaît le langage :

$$(L(E_1) \mid \ldots \mid L(E_n))*$$

- déterminiser l'AFND (algorithme standard) ;
- minimiser l'AFD (algorithme standard) ;
- enrober tout ça dans un programme suivant le paramétrage logiciel indiqué.

C'est ce que fait l'outil JFLEX pour Java (appréciable, non ?).

Les expressions régulières c'est bien...

... mais c'est lourd !

Les constantes réelles :

$$(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*$$
$$.(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*$$
$$(\epsilon|E(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9))$$

- $\Rightarrow$ besoin d'augmenter le confort de spécification ;
  - lisibilité ;
  - concision.
- sans toucher à l'expressivité (garder un langage régulier).

$\Rightarrow$ classes de caractères et descriptions régulières.

Classes de caractères

Pour rassembler et nommer des ensembles de caractères.

Ex :

- *chiffre* = [0 - 9]
- *lettre* = [a-z A - Z]
- *etoile* = "*"

On respire déjà mieux :

$$\textit{chiffre chiffre}* \ \ .\textit{chiffre chiffre}* \ \ (\epsilon | \texttt{E}\ \textit{chiffre chiffre})$$

Descriptions régulières - 2

Pour nommer des expressions régulières et s'en resservir :

$$\begin{aligned} \textit{const\_entiere} &= \textit{chiffre chiffre}* \\ \textit{const\_reelle} &= \textit{const\_entiere}.\textit{const\_entiere}(\mathtt{E}\ \textit{chiffre chiffre} \mid \epsilon) \end{aligned}$$

Formellement : pour un alphabet d'entrée Σ

$$\begin{aligned} \text{nom}_1 &= ER_1(\Sigma) \\ \text{nom}_2 &= ER_2(\Sigma, \text{nom}_1) \\ &\dots \\ \text{nom}_n &= ER_n(\Sigma, \text{nom}_1, \dots, \text{nom}_{n-1}) \end{aligned}$$

où :

- les nom_i sont des noms distincts 2 à 2 ;
- les ER_i sont des expr. rég. sur $\Sigma \cup \{\text{nom}_1, \dots, \text{nom}_{i-1}\}$.

Descriptions régulières - 2

Pas de descriptions récursives !

Pas de *const_entiere* = *chiffre* | *chiffre const_entiere*

Pas non plus de récursivité indirecte.

Le langage engendré par une description régulière avec récursivité n'est pas nécessairement régulier.

Spécifications à la JFlex

- nommage de classes de caractères et descriptions régulières ;

$$\begin{aligned} \textit{ident} &= \textit{lettre}\,(\textit{lettre} \mid \textit{chiffre})* \\ \textit{blanc} &= \texttt{[\ \ \textbackslash n\textbackslash t]} \end{aligned}$$

- symboles du + au - prioritaires, association d'une action à une classe de symbole ;

```
"if" {return new Symbol(IF);}
          // IF prioritaire sur IDENT
ident {return new Symbol(IDENT);}
blanc { // rien, retenu par le crible}
```

Rendez-vous en TD et TP !

Aparté (1) sur la génération automatique de code

Avant d'écrire un générateur, se demander si c'est :

1. possible ;
2. rentable.

Possible :

- existence d'un algorithme fondé rigoureusement ;
- oui dans le cas d'un analyseur lexical, grâce aux fondements de théorie du langage (sem. opérationnelle).

Rentable :

- le générateur devra être utilisé plusieurs fois !
- modifications demandant regénération, ou génération de différents analyseurs.

Aparté (2) sur la génération automatique de code

Dans votre cas, le générateur est déjà écrit.

Choisir le plus rentable en considérant :

- temps d'apprentissage de l'outil ;
- utilisation des interfaces imposées par l'outil ;
- débogage pas toujours évident ;
- versus tout faire à la main, mais tout contrôler.

En TP je vous force la main : tout le monde utilisera JFlex.

Apparté (3) sur les spécifications formelles

Avantage : oblige à tout décrire, lever toute ambiguïté.

Ex :

- pour les constantes réelles : chiffres avant le . ?
- pour les constantes réelles : chiffres après le . ?
- pour les identificateurs : majuscules ?
- etc.