Analyse lexicale

Mirabelle Nebut

Bureau 223 - extension M3
mirabelle.nebut at lifl.fr

2010-2011

Analyse lexicale

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

Ce cours est essentiellement tiré de [Wilhelm et Maurer].

Pré-requis

Notions de théorie du langage :

- automates à nombre fini d'état (AF) ;
- langages réguliers ;
- expressions régulières.

$\Rightarrow$ à réviser d'ici le TD de la fin de semaine

- voir cours AL en LS4 ;
- voir fiche de rappel des définitions sur le portail COMPIL ;
- voir avec moi si vous partez de rien ou presque rien.

Rappels

- Premier module du compilateur, au contact du texte source ;
- décompose la suite de caractères d'entrée en une suite de symboles.

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	
...	...

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

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

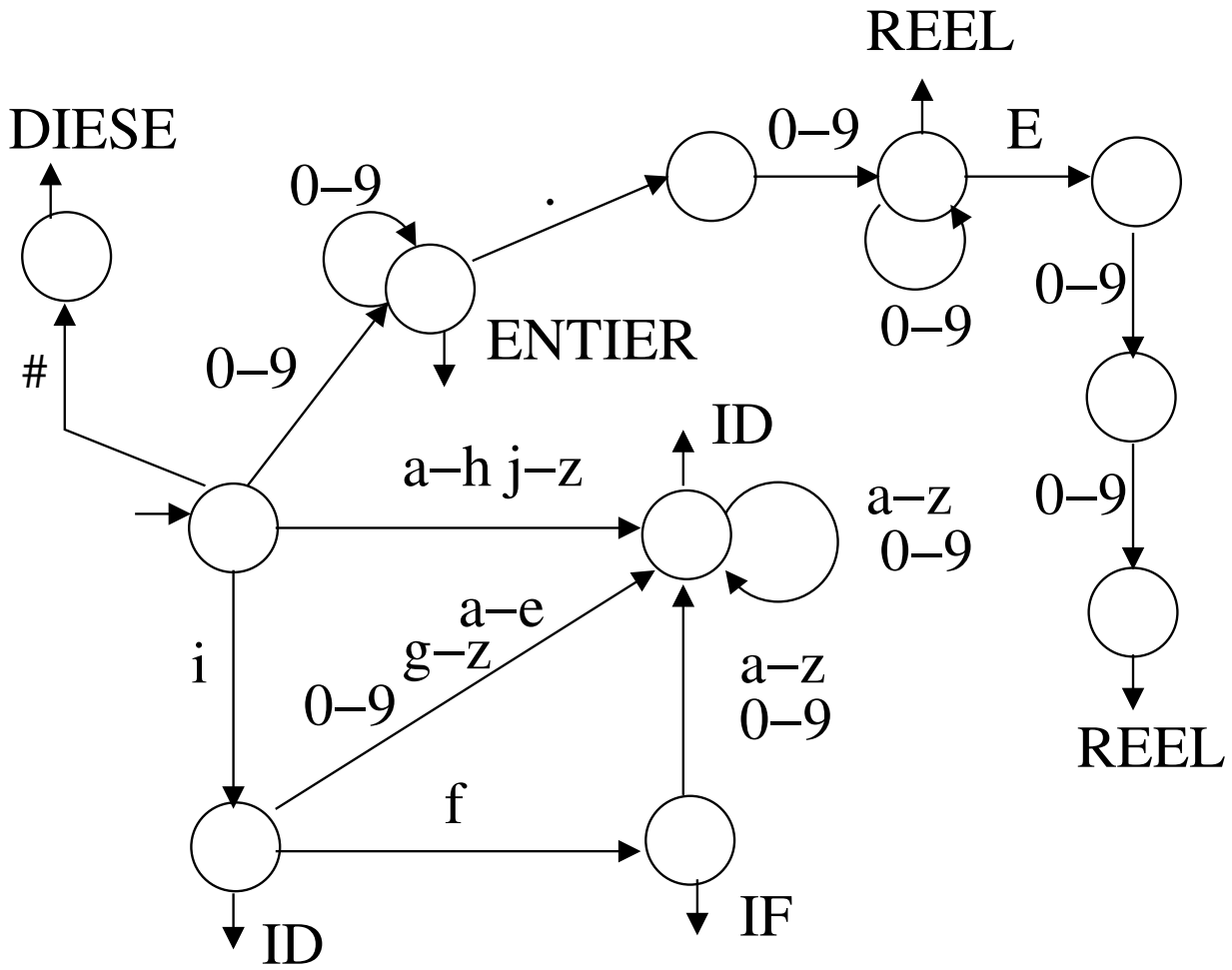
À vos claviers là comme ça tout de suite ?

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.

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

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.

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)

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 :

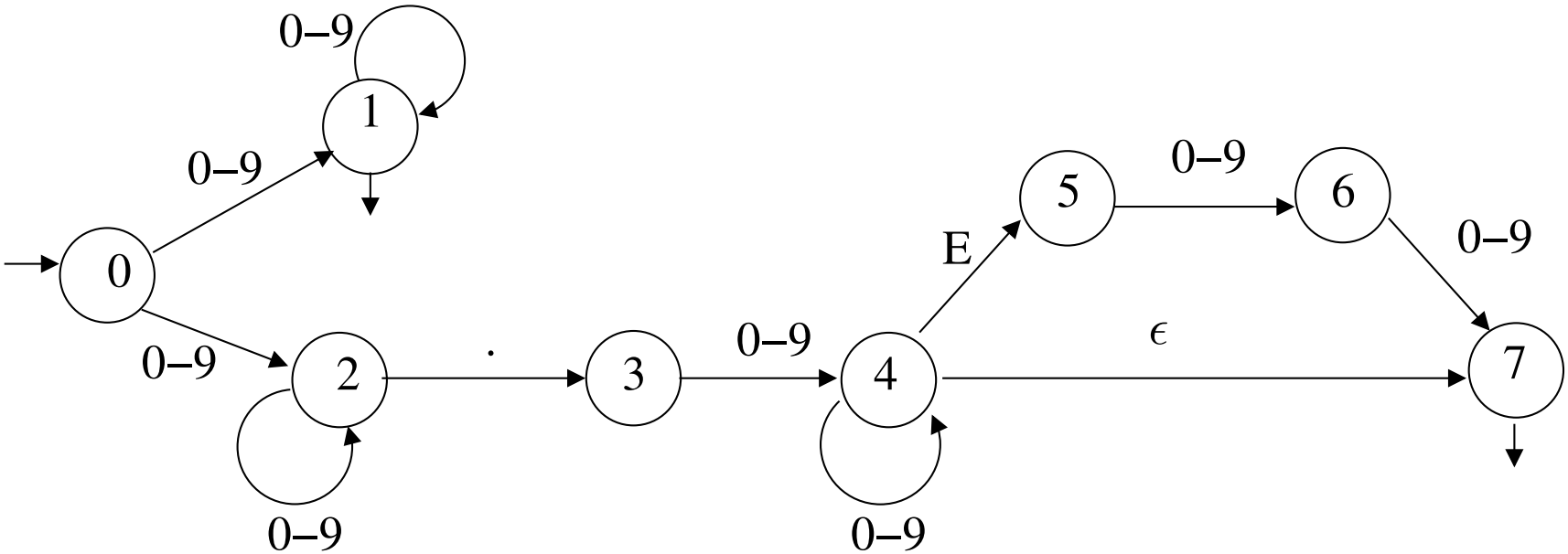
- pour transformer une expression régulière en AFND ;
- pour déterminiser un AFND ($\Rightarrow$ AFD).

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

Un AFND : exemple des constantes entières ou réelles

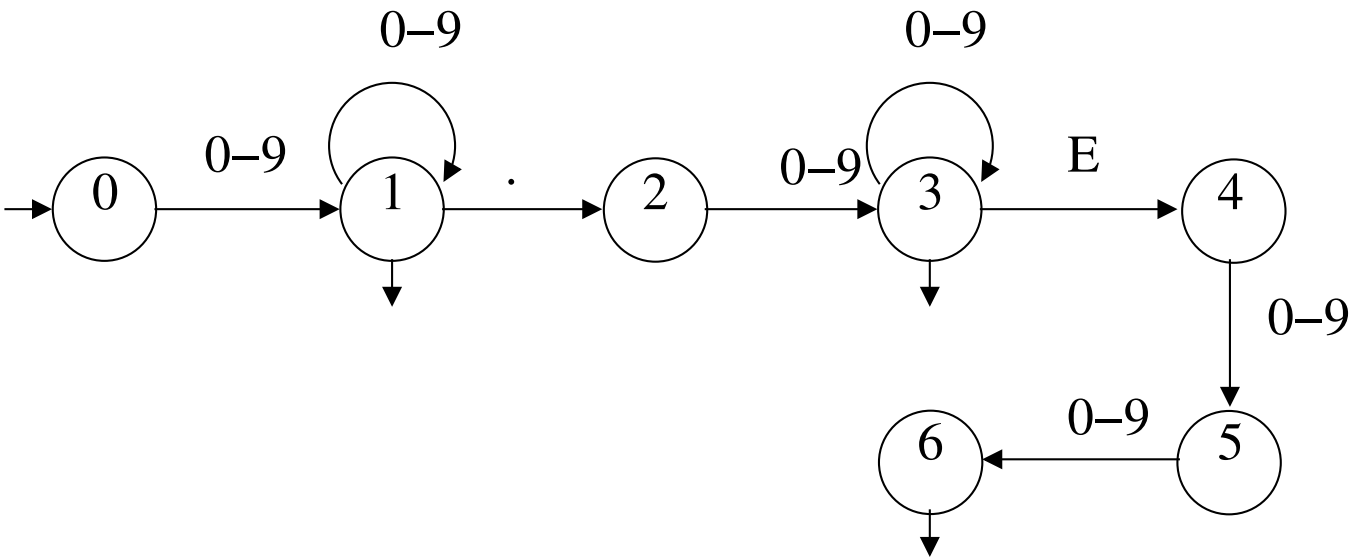
Deux sources de non-déterminisme :

- ϵ-transition ;
- transitions sortant d'un même état porteuses de la même étiquette.



AFD : exemple

Suppression du non-déterminisme.



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 $\Sigma \cup \{\text{nom}_1, \dots, \text{nom}_{\mathbf{i-1}}\}$.

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.