# Partie II : Analyse lexicale

- Les langages réguliers
- Traduction d'un langage régulier en un programme
- Fonctionnement d'un analyseur lexical
- Génération automatique d'analyseurs lexicaux



# Les langages réguliers

**Définition** : Un langage est dit *régulier* si sa syntaxe peut être exprimée à l'aide d'une seule règle EBNF non récursive.

Comme il n'y a qu'une seule règle non récursive, tous les symboles dans la partie droite de la production doivent être des symboles terminaux. La partie droite est aussi appelée *expression régulière.*

L'intérêt des langages réguliers est qu'ils peuvent être reconnus par des machines à états finis.

Autre caractérisation : un langage est *régulier* si sa syntaxe peut être exprimée par *plusieurs* règles EBNF qui ne dépendent pas récursivement les unes des autres.

**Exemple** :

```
identifier = letter {letter | digit}
digit = "0" | ... | "9"
letter = "a" | ... | "z" | "A" | ... | "Z"
```

## Les langages réguliers et l'analyse lexicale

- La syntaxe d'un langage de programmation est habituellement donnée en deux parties.
- La *Micro-Syntaxe* décrit la forme des mots individuels ou lexèmes (tokens).
- La *Macro-Syntaxe* décrit comment les programmes sont formés à partir des lexèmes.
- La traduction d'un programme source en une séquence de lexèmes est la tâche principale d'un analyseur lexical dans un compilateur.
- La Micro-syntaxe est habituellement décrite par un langage régulier.
- Donc les analyseurs lexicaux peuvent être des machines à états finis.
- Quel type de programmes correspondent aux machines à états finis ?



## Exercice

Supposons que nous avons une fonction

```
char next ();
```

qui consomme et renvoie le caractère suivant lu en entrée.

Écrivez une fonction

```
boolean isIdent ()
```

qui teste si l'entrée est de la forme

```
input = identifiier '\n'.
```

Est-ce que la grammaire des identificateurs vous aide pour écrire cette fonction ?

De quelle manière ?

## Traduction d'un langage régulier en un programme

| K | Pr(K) |
|---|---|
| "x" | **if** (sym == "x") next(); **else** error(); |
| (exp) | Pr(exp) |
| [exp] | **if** (« sym in first(exp) ») { Pr(exp) } |
| {exp} | **while** (« sym in first(exp) ») { Pr(exp) } |
| $fact_1$ ... $fact_n$ | $Pr(fact_1)$ ; ... ; $Pr(fact_n)$ |
| $term_1$ \| ... \| $term_n$ | **switch** (sym) { |
| | **case** first($term_1$): $Pr(term_1)$; **break**; |
| | ... |
| | **case** first($term_n$): $Pr(term_n)$; **break**; |
| | **default**: error() |
| | } |



## Traduction d'un langage régulier en un programme (2)

**Hypothèses :**

- un symbole *lookahead*, stocké dans `sym`.
- **`next()`** consomme le symbole suivant et le stocke dans `sym`.
- **`error()`** quitte avec un message d'erreur.
- **`first(exp)`** est l'ensemble des symboles initiaux de `exp`.
- On suppose que la syntaxe donnée est analysable par la gauche (*left-parsable*).

# Traduction d'un langage régulier en un programme (3)

Cela signifie :

| K | Condition |
|---|---|
| $term_1$ \| ... \| $term_n$ | Les termes n'ont pas de symboles initiaux en commun. |
| $fact_1$ ... $fact_n$ | si $fact_i$ contient la séquence vide alors $fact_i$ et $fact_{i+1}$ n'ont pas de symboles initiaux en commun. |
| {exp}, [exp] | L'ensemble des symboles initiaux de exp ne peut pas contenir un symbole qui suit aussi les expressions {exp} or [exp]. |



# Exemple : un scanner pour les identificateurs

```
void ident () {
  if (isLetter(ch)) next(); else error();
  while (isLetterOrDigit(ch)) {
    switch (ch) {
    case 'a': ... case 'z':
    case 'A': ... case 'Z': letter(); break;
    case '0': ... case '9': digit(); break;
    }
  }}
```

avec

```
  boolean isLetter(char ch) {
    return
      'a' <= ch && ch <= 'z' || 'A' <= ch && ch <= 'Z '
  }

  boolean isDigit(char ch) {
    return '0' <= ch && ch <= '9';
  }

  boolean isLetterOrDigit(char ch) {
    return isLetter(ch) || isDigit(ch);
  }
```

# Exemple : un scanner pour les identificateurs (2)

```
void letter() {
    switch (ch) {
    case 'a': if (ch == 'a') next(); else error();
    ...
    case 'Z': if (ch == 'Z') next(); else error();
  }
 void digit() {
    switch (ch) {
    case '0': if (ch == '0') next(); else error();
    ...
    case '9': if (ch == '9') next(); else error();
  }
```

- **ou bien, en compactant un peu :**

```
void ident () {
  if ('a' <= ch && ch <= 'z' ||
      'A' <= ch && ch <= 'Z')
    next();
  else error();
  while ('a' <= ch && ch <= 'z' ||
         'A' <= ch && ch <= 'Z' ||
         '0' <= ch && ch <= '9')
   next();
```



# Le travail d'un analyseur lexical

- L'action de base d'un analyseur lexical est de lire une partie de l'entrée et de retourner un lexème:

```
Token sym;
void nextSym () {
 "ignore les espaces blancs et assigne le prochain
lexème à sym"
}
```

- Un espace blanc peut être
  - un caractère blanc, une tabulation, un retour à la ligne
  - plus généralement : n'importe quel caractère ≤ ' '
  - des commentaires : une séquence quelconque de caractères entre /* ... */.
- Un lexème consiste en une classe de lexème (token class) avec éventuellement d'autres informations.

## Quelques Lexèmes de Java

- Les classes de lexème

| | |
|---|---|
| IDENT | `foo, main,` |
| NUMBER | `0, 123, 1000` |
| FLOAT | `0.5  1.0e+3` |
| STRING | `"", "a", "*** error"` |
| CLASS | `class` |
| VOID | `void` |
| LPAREN | `(` |
| RPAREN | `)` |
| LBRACE | `{` |
| RBRACE | `}` |
| SEMICOLON | `;` |
| EOF | `\uFFFF  (i.e. (char)-1)` |
| ... | |

- Les classes de lexème sont représentées par des entiers (int) en Java.



## Exemple d'exécution d'un analyseur lexical

- Pour le programme suivant:

```
class Foo {
  void bar () {
    println ("hello world\n");
  }
}
```

- L'analyseur lexical doit retourner:

CLASS IDENT(Foo) LBRACE VOID IDENT(bar)
LPAREN RPAREN LBRACE IDENT(println) LPAREN
STRING("hello world\n") RPAREN SEMICOLON
RBRACE RBRACE EOF

# L'interface d'un analyseur lexical

```
class Scanner {
      /** Constructor */
      Scanner (InputStream in)

      /** The symbol read = tokenclass last */
      int sym;

      /** The symbol's character representation */
      String chars;

      /** Read next token into sym and chars */
      void nextSym ()

      /** Close input stream */
      void close()

 }
```



# Syntaxe lexicale de l'EBNF

La syntaxe des lexèmes de l'EBNF :

```
symbol      = {blank}
              (identifier | literal |
               "(" | ")" | "[" | "]" | "{" | "}" | "|" | "=" | ". " ).
Identifier  =  letter { letter | digit }.
literal     =  "\"" {stringchar} "\"".
stringchar  =  escapechar  |  plainchar.
escapechar  =  "\\" char.
plainchar   =  charNoQuote.
```

# Définition des symboles de l'EBNF

```
package ebnf;
    interface Symbols {
      static final int
        ERROR   = 0,
        EOF     = ERROR  +1, IDENT   = EOF    +1,
        LITERAL = IDENT  +1, LPAREN  = LITERAL+1,
        RPAREN  = LPAREN +1, LBRACK  = RPAREN +1,
        RBRACK  = LBRACK +1, LBRACE  = RBRACK +1,
        RBRACE  = LBRACE +1, BAR     = RBRACE +1,
        EQL     = BAR    +1, PERIOD  = EQL    +1;
    }
```

Notes sur Java :

- Les symboles sont regroupés dans une interface qui peut être « héritée » par les classes ayant besoin d'y accéder.
- +1 : astuce pour compenser l'absence d'énumérations en Java.



# Scanner EBNF (1)

```
package ebnf;
  import java.io.*;

class Scanner implements /*imports*/
  Symbols {

  /** the symbol recognized last */
  public int sym;

  /** if that symbol was an identifier
      or a literal, it's string
      representation */
  public String chars;

  /** the character stream being tokenized
  */
  private InputStream in;

  /** the next unconsumed character */
  private char ch;

  /** a buffer for assembling strings */
  private StringBuffer buf =
          new StringBuffer();

  /** the end of file character */
  private final char eofCh = (char) -1

  /** constructor */
  public Scanner(InputStream in) {
    this.in = in;
    nextCh(); }
```

```
  public static void error(String msg) {
    System.out.println(
          "**** error: "+msg );
    System.exit(-1);
  }

  /** print current character and read
      next character */
  private void nextCh() {
    System.out.print(ch);
    try {
      ch = (char)in.read();
    } catch (IOException ex) {
      error("read failure: " +
            ex.toString());
    }
  }

  /** read next symbol*/
  public void nextSym() {
    while (ch <= ' ') nextCh();
    switch (ch) {
      case 'a': . . . case'z':
      case 'A': . . . case'Z':
        buf.setLength(0);
        buf.append(ch); nextCh();
        while ('a' <= ch && ch <= 'z' ||
               'A' <= ch && ch <= 'Z' ||
               '0' <= ch && ch <= '9') {
          buf.append(ch); nextCh();}
```

# Scanner EBNF (2)

```
      sym = IDENT;
      chars = buf.toString();
      break;
    case '\"':
      nextCh();
      buf.setLength(0);
      while (' ' <= ch && ch != eofCh &&
             ch != '\"') {
        if (ch == '\\') nextCh();
        buf.append(ch); nextCh();
      }
      if (ch == '\"') nextCh();
      else
        error("unclosed string literal");
      sym = LITERAL;
      chars = buf.toString();
      break;
    case '(':
      sym = LPAREN; nextCh(); break;
    case ')':
      sym = RPAREN; nextCh(); break;
    case '[':
      sym = LBRACK; nextCh(); break;
    case ']':
      sym = LBRACK; nextCh(); break;
    case '{':
      sym = LBRACE; nextCh(); break;
    case '}':
      sym = LBRACE; nextCh(); break;
```

```
      case '|':
        sym = BAR; nextCh(); break;
      case '=':
        sym = EQL; nextCh(); break;
      case '.':
        sym = PERIOD; nextCh(); break;
      case eofCh:
        sym = EOF; break;
      default:
        error("illegal character: " + ch +
        "(" + (int)ch + ")");
      }
    }

  /** the string representation of a symbol*/
  public static String representation
          (int sym) {
    switch (sym) {
    case ERROR  : return "<error>";
    case EOF    : return "<eof>";
    case IDENT  : return "identifier";
    case LITERAL: return "literal";
    case LPAREN : return "`('";
    case RPAREN : return "`)'";
      . . .
    default     : return "<unknown>"; }
  }

  public void close() throws IOException {
    in.close(); }
```



# Un programme de test pour le scanner EBNF

```
package ebnf;
import java.io.*;
class ScannerTest implements Symbols {
  static public void main(String[] args) {
    try {
      Scanner s = new Scanner(new FileInputStream(args[0]));
      s.nextSym();
      while (s.sym != EOF) {
        System.out.println("[" + Scanner.representation(s.sym) + "]");
        s.nextSym();
      }
      s.close();
    } catch (IOException ex) {
      System.out.println(ex);
      System.exit(-1);
    }
  }
}
```

## Plus longue correspondance

- **Problème :**

  La syntaxe donnée pour EBNF est ambiguë
  (pourquoi ?)

- **Solution :**

  Le scanner détermine à chaque étape le *plus long* symbole qui correspond à la définition

  (« longest match rule »)



## Génération automatique d'analyseurs lexicaux

- Il y a un procédé systématique pour faire correspondre un analyseur lexical à n'importe quelle expression régulière.
- Trois étapes :
  - expression rationnelle -> automate à états finis non déterministe (AFND)
  - AFND -> automate à états finis déterministe (AFD)
  - AFD -> programme généré du scanner
- Cela peut être automatisé dans un *générateur de scanner*.

## Automate à états finis

- Consiste en un nombre fini d'*états* et de *transitions*.
- Les transitions sont étiquetées par les symboles d'entrée.
- Il y a un *état initial*.
- Un sous-ensemble des états sont les *états finaux*.
- Un automate à états finis démarre dans l'état initial, et pour chaque symbole lu suit une arête étiquetée par ce symbole.
- Il *accepte* une chaîne en entrée ssi il termine dans un état final.
- Exemples: Voir tableau, (voir aussi figure 2.3 dans Appel).

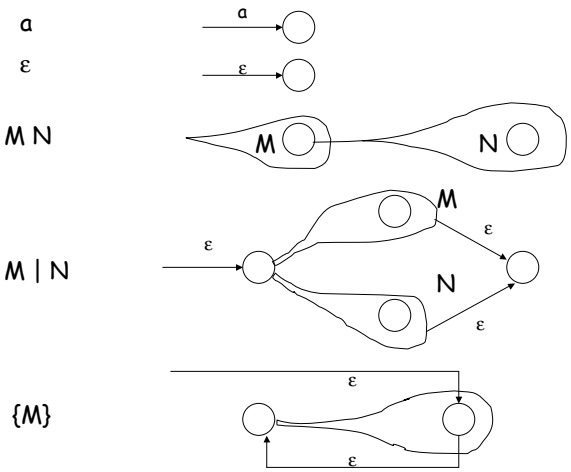


## Automates à états finis (non) déterministes

- Dans un *automate à états finis non déterministe*, il peut y avoir plus d'une arête partant d'un même nœud et étiquetée par un même symbole.
- Il peut y avoir une arête spéciale $\varepsilon$ qu'on peut suivre sans consommer de symbole en entrée.
- A l'inverse, dans un *automate à états finis déterministe* toutes les arêtes quittant un même nœud ont des ensembles d'étiquettes deux à deux disjoints, et il n'y a pas d'étiquette $\varepsilon$.

## Des expressions régulières aux AFNDs

- Voici une façon systématique de traduire n'importe quelle expression régulière en un AFND :






## Changer un AFND en un AFD

- Problème : exécuter un AFND requiert des retours arrière (*backtracking*), ce qui est inefficace.
- On aimerait le changer en un AFD.
- Idée de base : construire un AFD qui a un état pour chaque *ensemble d'états* dans lequel l'AFND pourrait se trouver.
- Un ensemble d'états est final dans l'AFD si il contient un état final de l'AFND.
- Comme le nombre d'états d'un AFND est fini (disons N), le nombre d'ensembles d'états possibles est lui aussi fini (borné par $2^N$)
- Souvent le nombre d'ensembles d'états effectivement accessibles est beaucoup plus petit.

## Algorithme pour changer un AFND en un AFD

- Voir Appel, Section 2.4
- Première étape : pour un ensemble d'états S, soit closure(S) le plus petit ensemble d'états tel que chaque état atteignable à partir d'un état de S en utilisant que des transitions ε soit dans closure(S).
- Algorithme pour calculer closure(S):

```
T := {S};
repeat
  T' := T;
  for each NFA-state s in T
    for each edge e from s to some state s'
      if (e is labelled with ε)
        T := T ∪ {s '}
until T = T '
```

- L'algorithme utilise le principe *d'itération jusqu'au point fixe* (*fixed point iteration*).



## Algorithme pour changer un AFND en un AFD (2)

- Deuxième étape : pour un ensemble d'états S et un symbole d'entrée c, soit DFAedge(S,c) l'ensemble des états qui peuvent être atteints à partir de S en suivant une arête étiquetée par c.
- Algorithme pour calculer DFAedge(S,c)

```
T := {}
for each state s in S
  for each edge e from s to some state s '
    if (e is labelled with c)
      T := T ∪ closure({s'})
```

## Simulation d'un AFD

- En utilisant la machinerie développée jusqu'ici, on peut déjà *simuler* un AFD, étant donné un AFND :
- Soit $s_1$ l'état initial de l'AFND et soit $c_1 \ldots c_k$ le flot d'entrée (*input-stream*) courant. Alors la simulation marche de la manière suivante :

```
d := closure({s1})
for i := 1 to k do
  d := DFAedge (d, ci)
 "accept if d contains an accepting NFA state"
```

- Manipuler ces ensembles pendant l'exécution est encore très inefficace.



## Construction de l'AFD

- Les états de l'AFD sont numérotés à partir de 0
- 0 est l'état d' erreur : l'AFD rentre dans l'état 0 ssi l'AFND est bloqué faute de trouver une arête correspondant au symbole d'entrée.
- Structures de données:

  `states` : un tableau qui fait correspondre à chaque état de l'AFD l'ensemble des états de l'AFND qu'il représente.

  `trans` : une matrice de transitions des numéros d'états vers les numéros d'états.

## Construction de l'AFD (2)

- Algorithme

```
states[0] := {}  // error state
states[1] := closure({s_1})
j := 1 ; p := 2
/* states[0..j) have been processed completely
   states[j..p) are as yet unprocessed
*/
while j < p do {
  for each input character c
    d := DFAedge (states[j], c)
    if (d == states[i] for some i < p)
      trans[j, c] := i
    else
      states[p] := d
      trans[j, c] := p
      p := p + 1
}
j := j + 1
```



## Exécuter un AFD

- Première possibilité : représenter l'AFD par une matrice :

  trans: Array [StateIndex, InputSymbol] of StateIndex

- Boucle de l'analyseur :

```
d := 1; // l'état initial de l'AFD
while ("encore des entrées") {
  c := "prochain caractère en entrée"
  d := trans[d, c]
}
"accepter si d contient un état final de l'AFND"
```

## Exécuter un AFD (2)

- Deuxième possibilité : représenter l'AFD par un branchement multi-indexé (case statement) :

```
d := 1;
while ("encore des entrées") {
  c := "prochain caractère sur l'entrée"
  switch (d) {
  case 0:
    switch (c) {
    case 'a': d := 3
    ...
    }
  ...
  }
}
```



## Résumé : l'analyse lexicale

- L'analyse lexicale transforme des caractères en entrée en lexèmes (tokens).
- La syntaxe lexicale est décrite par des expressions régulières.
- Nous avons vu deux façons de construire un analyseur lexical à partir d'une grammaire pour la syntaxe lexicale.
- A la main, en utilisant un schéma de programmation.
  - Ça marche si la grammaire est analysable par la gauche (left-parsable).
- A la machine, en passant d'une expression régulière à un AFND puis à un AFD.

## Générateurs de scanners

- Il y a de nombreux générateurs qui génèrent automatiquement un analyseur lexical à partir d'une description.
- La description énumère les classes de lexèmes et donne leur syntaxe sous forme d'expressions régulières.
- Exemples : Lex, JavaLex.
- Avantages à utiliser un générateur de scanners ?
- Désavantages ?