Analyse lexicale

Martin Odersky

version 1.7

Plan du cours

1. Les langages réguliers
   - Caractérisation d'un langage régulier
   - Analyse lexicale
   - Traduction d'un langage régulier en un programme
2. Comment construire un analyseur lexical ?
   - Un analyseur lexical en pratique
   - Exemple : Analyseur lexical EBNF
   - Syntaxes ambiguës
3. Génération automatique d'analyseurs lexicaux
   - Automates à états finis
   - Etapes de transformation d'une ER à un analyseur
   - Logiciels de génération automatique d'analyseurs

Les langages réguliers

Définition : Langage régulier

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 (ER).

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

Définition : Micro- et macro-syntaxe

Dans le cadre des langages de programmation, on parle de :

Micro-syntaxe décrit la forme des mots individuels c'est-à-dire des lexèmes (*tokens*).

Macro-syntaxe décrit comment les programmes sont formés à partir des lexèmes.

Définition : Analyseur lexical

L'analyseur lexical (en anglais *scanner*) traduit un programme source en une séquence de lexèmes définis par la micro-syntaxe.

Si une micro-syntaxe est décrite par un langage régulier, son analyseur lexical peut être une machine à états finis.

Exercice

Soit une fonction `def next: Character` qui consomme et renvoie le caractère suivant lu en entrée.

Écrivez une fonction `def ident: Boolean` qui teste si l'entrée est de la forme `input = identifier "\n"`.

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

Exercice

Quel type de programmes correspondent aux machines à états finis ?

Traduction d'un langage régulier en un programme

ER	*Pr*(*ER*)
`"x"`	`if (char == "x") next else error;`
`(exp)`	*Pr*(`exp`)
`[exp]`	`if (`⟨`char` *in* `first(exp)`⟩`)` *Pr*(`exp`);
`{exp}`	`while (`⟨`char` *in* `first(exp)`⟩`)` *Pr*(`exp`);
$\texttt{fact}_1$ `...` $\texttt{fact}_n$	*Pr*($\texttt{fact}_1$); `...`; *Pr*($\texttt{fact}_n$);

$\texttt{term}_1$ `| ... |` $\texttt{term}_n$	`char match {`
	  `case first(`$\texttt{term}_1$`) =>`
	    Pr($\texttt{term}_1$);
	  `...`
	  `case first(`$\texttt{term}_n$`) =>`
	    Pr($\texttt{term}_n$);
	  `case _ => error;`
	`}`

On considère les hypothèses suivantes :

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

Définition : Analysable par la gauche

Un grammaire est dite «analysable par la gauche» si :

ER	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]` ou `{exp}`.

Exemple : Analyseur pour les identifiants

```
def ident = {
  letter
  while (List('a',...,'z','A',...,'Z','0',...,'9') contains char) {
    char match {
      case 'a'|...|'z'|'A'|...|'Z' => letter
      case '0'|...|'9' => digit
}}}
def letter = char match {
  case 'a' => if (char == 'a') next else error
  ...
  case 'Z' => if (char == 'Z') next else error
}
def digit = char match {
  case '0' => if (char == '0') next else error
  ...
  case '9' => if (char == '9') next else error
}
```

Exemple : Analyseur simplifié pour les identifiants

```
def ident = {
  if (isLetter(char))
    next
  else error
  while (isLetterOrDigit(char))
    next
}
```

Le rôle 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 :

```
var token: Token
def nextToken: Unit = {
  token = «le prochain lexème, les espaces blancs étant ignorés»
}
```

Un espace blanc peut être :

- un caractère blanc, une tabulation, un retour à la ligne ;
- plus généralement : n'importe quel caractère $\leq$ '␣' ;
- des commentaires : une séquence quelconque de caractères entre /* ... */.

Qu'est-ce qu'un lexème ?

Un lexème consiste en une classe de lexème (*token class*) avec éventuellement d'autres informations.

Exemple : Quelques classes de lexèmes de Scala

IDENT	→	foo, main, ...
NUMBER	→	0, 123, 1000, ...
STRING	→	"", "a", "Tchô", ...
CLASS	→	class
EXTENDS	→	extends
LPAREN	→	(
RBRACE	→	}
SEMICOLON	→	;
EOF	→	\uFFFF

Les classes de lexème sont représentées

- par des valeurs d'une classe `Enumeration` en Scala,
- par des entiers (`int`) en Java.

Exemple : Une analyse lexicale

- Pour le programme suivant :

```
class Foo {
  def bar: Unit =
    println("Tchô");
}
```

- L'analyseur lexical doit retourner :

```
CLASS IDENT("Foo") LBRACE
  DEF IDENT("bar") COLON IDENT("Unit") EQUAL
    IDENT("println") LPAREN STRING("Tchô") RPAREN SEMICOLON
RBRACE EOF
```

Interface d'un analyseur lexical

```
class Scanner (in: InputStream) {
  /** Le lexème courant. */
  var token: Token
  /** La position du premier caractère du lexème. */
  var start: Int
  /** Si le lexème est paramétrisé, sa valeur. */
  var chars: String
  /** Une représentation texte du lexème. */
  def representation: String
  /** Passe au prochain lexème. */
  /** Les caractères blancs sont sautés. */
  def nextToken: Unit
}
```

Exemple : Analyseur lexical EBNF

Le code exécutable de l'analyseur lexical EBNF est disponible sur le site web du cours.

Exemple : Syntaxe EBNF

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

Exemple : Définition des lexèmes EBNF

```
object EBNFTokens extends Enumeration {
  type Token = Value
  val BAD     = Value("<bad>")
  val EOF     = Value("<eof>")
  val IDENT   = Value("ident")
  val LITERAL = Value("literal")
  val LPAREN  = Value("("); val RPAREN = Value(")")
  val LBRACK  = Value("["); val RBRACK = Value("]")
  val LBRACE  = Value("{"); val RBRACE = Value("}")
  val BAR     = Value("|")
  val EQL     = Value("=")
  val PERIOD  = Value(".")
}
```

Exemple : Analyseur lexical EBNF

```
class EBNFScanner (in: InputStream) {
  import EBNFTokens._
  private val EOFChar =
    (-1).asInstanceOf[Char]
  var token: Token = BAD
  var chars: String = ""
  private var char = '␣'
  private def nextChar =
   char = in.read().asInstanceOf[Char]
  def representation: String = token match {
    case IDENT | LITERAL =>
      token.toString() + "(" + chars + ")␣"
    case _ => token.toString() + "␣"
  }
  def close = in.close();
```

continue sur la page suivante ...

```
  def nextToken = {
    while (char <= '␣') nextChar
    if (Character.isLetter(char)) {
      val buf = new StringBuffer()
      buf.append(char); nextChar
      while (Character.isLetterOrDigit(char)) buf.append(char); nextChar
      token = IDENT; chars = buf.toString()
    } else char match {
      case '"' => {
        nextChar
        val buf = new StringBuffer()
        while (char != '"' && char != EOFChar &&  char >= '␣')
          if (char == '\\') nextChar
          buf.append(char); nextChar
        if (char == '"') {
          nextChar
          token = LITERAL; chars = buf.toString()
        } else error("Chaîne␣non-terminée")
      }
      case '(' => token = LPAREN; nextChar
      case ')' => token = RPAREN; nextChar
      case '[' => token = LBRACK; nextChar
      case ']' => token = RBRACK; nextChar
      case '{' => token = LBRACE; nextChar
      case '}' => token = RBRACE; nextChar
      case '|' => token = BAR; nextChar
      case '=' => token = EQL; nextChar
      case '.' => token = PERIOD; nextChar
      case EOFChar => token = EOF
      case _ => token = BAD; nextChar
}}}
```

Exemple : Programme de test pour analyseur EBNF

```
object EBNFScannerTest {
  import EBNFTokens._
  def main (args: Array[String]) = {
    val scan = new EBNFScanner(
      new FileInputStream(args(0)))
    scan.nextToken
    while (scan.token != EOF) {
      Console.print(scan.representation)
      scan.nextToken
    }
    Console.println(scan.representation)
    scan.close
  }
}
```

Le même analyseur lexical EBNF, en Java

Exemple Java : Définition des lexèmes EBNF

```
interface EBNFTokens {
  public static int BAD     = 0;
  public static int EOF     = 1;
  public static int IDENT   = 2;
  public static int LITERAL = 3;
  public static int LPAREN  = 4;
  public static int RPAREN  = 5;
  public static int LBRACK  = 6;
  public static int RBRACK  = 7;
  public static int LBRACE  = 8;
  public static int RBRACE  = 9;
  public static int BAR     = 10;
  public static int EQL     = 11;
  public static int PERIOD  = 12;
}
```

Exemple Java : Analyseur lexical EBNF

```
public class EBNFScanner implements EBNFTokens {
  public EBNFScanner (InputStream in) {
    this.in = in;
  }
  private InputStream in;
  private final char EOFChar = (char)-1;
  public int token = BAD;
  private String chrs = "";
  private char chr = '␣';
  private void nextChr() {
    try {
      chr = (char)in.read();
    } catch (IOException e) {
      throw new Error("Erreur␣d'E/S");
    }
  }
```

continue sur la page suivante . . .

```
public void nextToken() {
  while (chr <= ' ') {
    nextChr();
  }
  if (Character.isLetter(chr)) {
    StringBuffer buf = new StringBuffer();
    buf.append(chr);
    nextChr();
    while (Character.isLetterOrDigit(chr)) {
      buf.append(chr);
      nextChr();
    }
    token = IDENT;
    chrs = buf.toString();
  } else {
    switch (chr) {
      case '\u0022':
        nextChr();
        StringBuffer buf = new StringBuffer();
        while (chr != '\u0022' && chr != EOFChar && chr >= ' ') {
          if (chr == '\\') {
            nextChr();
          }
          buf.append(chr);
          nextChr();
        }
```

continue sur la page suivante . . .

```
        if (chr == '\u0022') {
          nextChr();
          token = LITERAL;
          chrs = buf.toString();
        } else {
          throw new Error("Chaîne␣non-terminée");
        }
        break;
      case '(': token = LPAREN; nextChr(); break;
      case ')': token = RPAREN; nextChr(); break;
      case '{': token = LBRACE; nextChr(); break;
      case '}': token = RBRACE; nextChr(); break;
      case '[': token = LBRACK; nextChr(); break;
      case ']': token = RBRACK; nextChr(); break;
      case '|': token = BAR; nextChr(); break;
      case '=': token = EQL; nextChr(); break;
      case '.': token = PERIOD; nextChr(); break;
      case EOFChar: token = EOF; break;
      default: token = BAD; nextChr(); break;
}}}}
```

Syntaxes ambiguës : Plus longue correspondance

Exercice

La syntaxe donnée pour EBNF est ambiguë ; expliquez pourquoi.

Problème :

- certaines grammaires sont ambiguës.

Solution :

- l'analyseur détermine à chaque étape le plus long lexème qui correspond à la définition (*longest match rule).*

Génération automatique d'analyseurs lexicaux

Il existe un procédé systématique pour faire correspondre un analyseur lexical à n'importe quelle expression régulière.

Ce procédé se compose de trois étapes :

1. expression régulière (ER) → automate à états finis non déterministe (AFND) ;
2. AFND → automate à états finis déterministe (AFD) ;
3. AFD → analyseur lexical généré.

Ce procédé peut être automatisé dans un générateur d'analyseurs lexicaux.

Automates à états finis

Définition : Automate à états finis

Un automate à états finis :

- consiste en un nombre fini d'états et de transitions ;
- où les transitions sont étiquetées par les symboles d'entrée ;
- un des état est l'état initial ;
- un sous-ensemble des états sont les états finaux ;
- il 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 disponibles au tableau et sur la figure 2.3 de l'*Appel*.

Dans un automate à états finis non déterministe (AFND) :

- Il peut y avoir plus d'une arête partant d'un même noeud et étiquetée par un même symbole.
- Il peut y avoir une arête spéciale ϵ qu'on peut suivre sans consommer de symbole en entrée.

A l'inverse, dans un automate à états finis déterministe (AFD) :

- Toutes les arêtes quittant un même noeud ont des ensembles d'étiquettes deux à deux disjoints.
- Il n'y a pas d'étiquette ϵ.

Transformation étape 1 : ER $\longrightarrow$ AFND

Algorithme : Traduction d'une expression régulière en un AFND

Expression	AFND	Expression	AFND
"a"	(transition a vers un état final)	ϵ	(transition ϵ vers un état final)
M N	(automate M suivi de l'automate N)		
M | N	(transitions ϵ vers M et N, puis transitions ϵ vers un état final)		
{M}	(automate M avec transitions ϵ de retour et de contournement)		

Transformation étape 2 : AFND → AFD

Exécuter un AFND requiert des retours arrière (*backtracking*), ce qui est inefficace.

On aimerait le changer en un AFD, ce qui est possible en se basant sur cette idée :

- On 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.

Pour un ensemble d'états `S`, on calcule `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 : Calcul de `closure(S)`

```
def closure(S : Set[State]) : Set[State] =
  var T = S
  do
    var T' = T
    for s ∈ T
      for s -ε-> t ∈ edges
        T = T ∪ {t}
  while (T ≠ T')
  T
```

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

Pour un ensemble d'états `S` et un symbole d'entrée `c`, on calcule `DFAedge(S,c)` l'ensemble des états qui peuvent être atteints à partir de `S` en suivant une arête étiquetée par `c`.

Algorithme : Calcul de `DFAedge(S,c)`

def $DFAEdge(S : Set[State], c : Character) : Set[State]$ =

  var $T := \{\}$

  for $s \in S$

    for $s \xrightarrow{c} t \in edges$

      $T := T \cup closure\{t\}$

  T

Solution intermédiaire : AFND → AFD par simulation

Soit `s1` l'état initial de l'AFND et soit `input` le flot d'entrée (*input-stream*) courant. Alors la simulation marche de la manière suivante :

Algorithme : Simulation d'un AFD

```
def simulate(s1 : State, input : Buffer[Character]) =
  var d := closure(List(s1))
  for c ∈ input
    d := DFAEdge(d, c)
  if (d ∈ acceptingStates) accept else fail
```

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

Solution optimale : AFND $\longrightarrow$ AFD par construction

- 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 sur les caractères des numéros d'états vers les numéros d'états.

Algorithme : Construction d'un AFD

```
states(0) := {}
states(1) := closure{s1}
var j := 1 // Les états [0..j) ont été traités
var p := 2 // Les états [j..p) sont encore non-traités
while (j < p)
  for c ∈ «tous les caractères d'entrée»
    d := DFAEdge(states(j), c)
    if (∃i ∈ 1..p−1 s.t. d = states(i))
      trans(j, c) := i
    else
      states(p) := d
      trans(j, c) := p
      p := p + 1
  j := j + 1
```

Transformation étape 3 : AFD → analyseur lexical

Première possibilité, représenter l'AFD par une matrice (`trans`).

Algorithme : Exécution d'un AFD

```
var d := 1 // l'état initial de l'AFD
for c ∈ input
  d := trans(d, c)
if (d ∈ acceptingStates) accept else fail
```

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

Algorithme : Exécution d'un AFD v.2

```
var d := 1 // l'état initial de l'AFD
for c ∈ input
  d match {
    case 0 => c match {
      case 'a' => d := 3
      case ...
    }
    case ...
  }
}
if (d ∈ acceptingStates) accept else fail;
```

Logiciels de génération automatique d'analyseurs

Il existe de nombreux logiciels qui génèrent automatiquement un analyseur lexical à partir d'une description.

- La description à fournir énumère les classes de lexèmes et donne leur syntaxe sous forme d'expressions régulières.
- Les plus connus : Lex (C), JFlex (Java) . . .

Exercice

Quels sont les avantages et les désavantages à utiliser un générateur d'analyseurs lexicaux ?

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.
    1. A la main, en utilisant un schéma de programmation. Fonctionne bien si la grammaire est analysable par la gauche (*left-parsable*).
    2. A la machine, en passant d'une expression régulière à un AFND puis à un AFD.