Analyse de Données Structurées - Cours 3

Ralf Treinen

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

treinen@pps.univ-paris-diderot.fr

3 février 2015

Objectif de l'analyse lexicale

- Lire le texte d'entrée, et faire un premier traitement en vue d'une simplification pour les étapes suivantes :
- *Découpage* de l'entrée en *lexèmes* (des mots élémentaires)
- *Classer* les lexèmes identifiés, création de jetons
- *Interpréter* les lexèmes quand pertinent, par exemple transformer une suite de chiffres en un entier
- *Abstraire* l'entrée : ignorer des détails non pertinents pour la suite (espaces, commentaires. ...)

Spécification d'une analyse lexicale

- Définition des catégories lexicales différentes : expressions régulières
- Définition de la stratégie : chercher le mot le plus long ou le plus court, comment résoudre des ambiguïtés
- Définir le type des jetons produits avec leurs arguments éventuels.

Implémenter une analyse lexicale

- Soit *écrire un programme* (Java ou autre) à la main, basé sur un automate fini : discuté au dernier cours.
  Les premiers compilateurs étaient effectivement écrits de cette façon (compilateur du langage FORTRAN, Backus et al. 1957 : 18 personnes-années).
- Soit faire *engendrer* un analyseur lexicale à partir d'une spécification (ce cours).
  C'est la technique utilisée pour l'écriture des compilateurs modernes.

L'interface de l'analyseur lexicale

- On pourrait imaginer que l'analyse lexicale va créer une liste Java avec tous les jetons créés lors de l'analyse.
- Problème : cette liste risque d'être très longue.
- Normalement, la phase suivante de l'analyse a seulement besoin de lire les jetons une fois dans l'ordre.
- Pour ces raisons, l'analyse lexicale crée les jetons un après l'autre *à la demande* : fonction qui renvoie le jeton suivant.

Différents générateurs

- Existent pour presque tous les langages de programmation.
- Le premier générateur était *lex*, publié en 1975 par Mike Lesk et Eric Schmidt. Engendre du code en C.
- Successeur : *flex*, 1987.
- Les générateurs modernes sont souvent issus de flex. Nous utilisons ici un générateur pour Java : *jflex*.
- Les générateurs pour des autres langages de programmation sont très similaires.

Le code engendré dans le case de jflex

- Une classe pour l'analyseur lexicale, le nom de la classe peut être défini dans la spécification (dans nos exemples : `Lexer`).
- La création d'un objet de cette classe (un analyseur) prend en argument un objet qui représente le *flot d'entrée*, par exemple un fichier, ou l'entrée standard.
- Il y une méthode pour demander le jeton suivant. Le nom de cette méthode, et le type des jetons, peuvent être définis dans la spécification.

La spécification

- Trois parties, séparées par des lignes `%%` :
  - code utilisateur
  - options et déclarations
  - règles lexicales
- Code utilisateur :
  - copié simplement au début du fichier engendré (avant la définition de la classe)
  - partie souvent vide (sauf commentaires, et `import ...`)

La partie *Options et Déclarations*

- Options : commencent avec le symbole %. Parmi les options les plus importantes :
  - `%class` *nom* : donne le nom de la classe engendrée.
  - `%public` : la classe engendrée est publique
  - `%type` *t* : le type de résultat de la fonction `yylex`.
  - `%unicode` : accepte des caractères Unicode en entrée de l'analyse lexicale (recommandé)
  - `%line` : compte lignes pendant l'analyse lexicale (disponible en `yyline`).
  - `%column` : compte colonnes pendant l'analyse lexicale (disponible en `yycolumn`).
  - `%state` *s* : Déclaration de l'état *s* (voir plus tard)

Analyse de Données Structurées - Cours 3
Générateurs d'analyse lexicale
Structure de la spécification et du code engendré

La partie *Options et Déclarations*

- Code entre `%{` et `%}` (peut être sur plusieurs lignes) :
  - copié au début de la classe engendré
  - ce code a donc accès aux champs de la classe (par exemple, `yyline`, `yycolumn`)
- Code entre `%eofval{` et `%eofval}` : code exécuté quand l'analyse lexicale arrive à la fin de l'entrée (défaut : `null`).

Analyse de Données Structurées - Cours 3
Générateurs d'analyse lexicale
Structure de la spécification et du code engendré

La partie *Options et Déclarations*

- Macros : système de définitions d'expressions régulières
  - mettre les mots entre apostrophes " et "
  - pour utiliser une expression régulières préalablement définie, par exemple du nom *r* : `{r}`.
  - classes de caractères, par exemple `[a-z]`
  - quelque classes de caractère prédéfinies, par exemple `[:letter:]`, `[:digit:]`, `[:uppercase:]`, `[:lowercase:]`.

Analyse de Données Structurées - Cours 3
Générateurs d'analyse lexicale
Structure de la spécification et du code engendré

La partie *Règles Lexicales*

- Séquence de
  expression-régulière { *code-java* }
- dans le cas le plus simple, le code java est un `return ...`
- Règles d'exécution : on cherche le lexeme le plus long possible, et on applique l'action de la première expression régulières qui s'applique.

Le premier exemple

- Analyse lexicale pour des expressions arithmétiques comme vu la dernière fois.
- Petite différence au premier exemple : les entiers ne contiennent pas d'exposant.
- Définition des classes pour les Symboles (type de jetons), puis pour les jetons éventuellement avec des arguments.
- Le fichier de spécification pour `jflex`.
- Un petit programme principal pour tester.

Fichier Sym.java

```
public enum Sym {
    INT, IDENT, PARG, PARD, MULT, PLUS;

    public String str(){
        switch (this) {
        case INT: return "INT";
        case IDENT: return "IDENT";
        case PARG: return "PARG";
        case PARD: return "PARD";
        case MULT: return "MULT";
        case PLUS: return "PLUS";
        default: return "impossible";
        }
    }
}
```


Fichier Token.java I

```
class Token {
    protected Sym symbol;
    public Token(Sym s) {
        symbol=s;
    }
    public Sym symbol() {
        return symbol;
    }
    public String str() {
        return (symbol.str());
    }
}

class StringToken extends Token {
    private String value;
    public StringToken(Sym c, String s) {
        super(c);
        value=s;
```


Fichier Token.java II

```
    }
    public String str() {
        return (symbol.str() + '(' + value + ')');
    }
}

class IntToken extends Token {
    private int value;
    public IntToken(Sym c, int i) {
        super(c);
        value=i;
    }
    public String str() {
        return(symbol.str() + '(' + value + ')');
    }
}
```

Fichier arith.flex I

```
%%
%public
%class Lexer
%unicode
%type Token

%{
  private Token token(Sym type) {
      return new Token(type);
  }
  private StringToken token(Sym type, String value) {
      return new StringToken(type, value);
  }
  private IntToken token(Sym type, int value) {
      return new IntToken(type, value);
  }
%}
```


Fichier arith.flex II

```
EspaceChar = [ \n\r\f\t]
Ch         = [0-9]
Le         = [a-zA-Z]

%%
{Ch}+              {return token(Sym.INT,
                          Integer.parseInt(yytext()));}
{Le}({Le}|{Ch})*   {return token(Sym.IDENT,yytext());}
"("                {return token(Sym.PARG);}
")"                {return token(Sym.PARD);}
"*"                {return token(Sym.MULT);}
"+"                {return token(Sym.PLUS);}
{EspaceChar}+      {}
```


Fichier Test.java

```
import java.io.*;

class Test {

    public static void main(String[] args) throws Exception {
        File input = new File(args[0]);
        Reader reader = new FileReader(input);
        Lexer lexer = new Lexer(reader);
        Token t;
        do {
            t=lexer.yylex();
            if (t != null) {System.out.println(t.str());}
        } while (t != null);
    }
}
```


L'automate créé

- Création des classes de caractères : tous les caractères qui ne sont jamais distingués par les expressions régulières sont groupés dans la même classe.
- Les classes crées doivent être disjointes.
- Exemple : expressions régulières :
  `"end"`
  `[a-z]*`
- Quatre classes de caractères disjointes : `[e]`, `[n]`, `[d]`, `[a-cf-mo-z]`

L'automate créé

- Création d'un automate non-déterministe pour l'union de toutes les expressions régulières.
- Déterminiser l'automate (et éliminer les ϵ-transitions).
- Minimiser l'automate.
- On peut demander à `jflex` de montrer ces trois automates (option `-dot`, visualiser les automates avec `xdot` par exemple)

Les états de l'analyseur lexical

- Par défaut (comme sur le premier exemple), votre analyseur lexical a un seul état.
- Pour en avoir plusieurs :
  - les déclarer à l'aide de `%state` (sauf YYINITIAL)
  - mettre toutes les règles dans le contexte d'un état
  - dans les actions : changer d'état à l'aide de `yybegin`.

Pourquoi utiliser plusieurs états ?

- Un premier exemple sont les commentaires : avec une expression régulières comme `"/*" .* "*/"` on a un problème quand il y a plusieurs commentaires dans le texte (pourquoi ?)
- Dans ce cas on veut on fait trouver le mot *le plus court* décrit par l'expression régulière. Cela peut être simulé en utilisant deux états.

Reconnaître les commentaires (simplifié)

```
%%
%type Token
 EspaceChar = [ \n\r\f\t]
 Letter     = [a-zA-Z]
%state INCOMMENT
%%
<YYINITIAL> {
  {Letter}+     {return token(Sym.IDENT, yytext());}
  {EspaceChar}  {}
  "/*"          {yybegin(INCOMMENT);}
}
<INCOMMENT> {
  "*/"   {yybegin(YYINITIAL);}
  .      {}
}
```

Reconnaître les commentaires

- `YYINITIAL` est l'état par défaut
- Il est crucial que la dernière règle s'applique à un mot de longueur 1 seulement.

Exemple : découper un mot en plusieurs parties

- Retour à notre premier exemple : on souhaite maintenant aussi reconnaître des entiers avec exposant (756e2, par exemple).
- On utilise deux états : quand on trouve un symbole "e" après une séquence de chiffres on stocke la valeur entière trouvée dans une variable, puis on va dans un deuxième état où on va lire exposant.

Nouvelle version de `arith.flex` I

```
%%

%public
%class Lexer
%type Token
%unicode

%{
  private Token token(Sym type) {
      return new Token(type);
  }
  private StringToken token(Sym type, String value) {
      return new StringToken(type, value);
  }
  private IntToken token(Sym type, int value) {
      return new IntToken(type, value);
  }
  int intbuff=0;
```


Nouvelle version de `arith.flex` II

```
  private String chop(String s) {
      return(s.substring(0,s.length()-1));
  }
  private int expo(int base, int ex) {
    int result=base;
    for(int i = 1; i<=ex; i++) {
      result=result*10;
    }
    return result;
}
%}

EspaceChar = [ \n\r\f\t]
Ch         = [0-9]
Le         = [a-zA-Z]

%state EXPONENT

%%
```

Nouvelle version de `arith.flex` III

```
<YYINITIAL> {
  {Ch}+              {return token(Sym.INT,
                                   Integer.parseInt(yytext()));}
  {Le}({Le}|{Ch})*   {return token(Sym.IDENT,yytext());}
  "("                {return token(Sym.PARG);}
  ")"                {return token(Sym.PARD);}
  "*"                {return token(Sym.MULT);}
  "+"                {return token(Sym.PLUS);}
  {EspaceChar}+      {}
  {Ch}+ "e"          {intbuff=Integer.parseInt(chop(yytext()));
                      yybegin(EXPONENT);}
}
<EXPONENT> {
  {Ch}+ {yybegin(YYINITIAL);
         return(token(Sym.INT,
                      expo(intbuff,Integer.parseInt(yytext()))));}
}
```


Mots clefs d'un langage de programmation

Solution naïve : une règle par mot clefs.

```
%%
%type Token
EspaceChar = [ \n\r\f\t]
Letter     = [a-zA-Z]
%%
"begin"       {return token(Sym.BEGIN)}
"end"         {return token(Sym.END)}
"class"       {return token(Sym.CLASS)}
{Letter}+     {return token(Sym.IDENT,yytext());}
{EspaceChar}  {}
```


Attention à l'ordre des règles

Entrée : `beg begin beginner`

- Premier appel à `yylex()` : seulement la quatrième règle s'applique ⇒ token IDENT.
- Deuxième appel à `yylex()` : les règles (2) et (4) s'appliquent au même lexeme `begin`, c'est donc la première parmi ces deux qui gagne ⇒ token BEGIN.
- Troisième appel à `yylex()` : les règles (2) et (4) s'appliquent mais la dernière reconnaît un lexeme plus long ⇒ token IDENT.

Mais regarder la taille de l'automate engendré !

Contrôler la taille de l'automate

- Techniques utilisés par le générateurs :
  - Utiliser des classes de caractères au lieu dans la représentation de l'automate.
  - Minimiser l'automate engendré à partir des expressions régulières.
- Optimisation dans la spécification : Éviter de créer une nouvelle classe lexicale pour chaque mot clef (Java : 46 mots clefs.)

Comment reconnaître les mots clefs sans catégories dédiées ?

- En Java (et pareil dans les autres langages de programmation) : tous les mots clefs sont des séquences de lettres en minuscules.
- Mettre une seule catégorie pour les identificateurs.
- Dans l'action associé, on cherche (par ex. dans une table de hachage) si le lexeme est un mot clefs, et crée un jeton en fonction.

Le fichier `keys.flex` I

```
import java.util.HashMap;
class Keys extends HashMap<String,Sym> {
        public Keys() {
                super();
                this.put("end",Sym.END);
                this.put("begin",Sym.BEGIN);
                this.put("class",Sym.CLASS);
        }
}

%%
%public
%type Token
%class Lexer
%unicode
EspaceChar = [ \n\r\f\t]
Letter     = [a-zA-Z]
```


Le fichier `keys.flex` II

```
%{
  private Keys keys = new Keys();
  private Token token(Sym type) {
       return new Token(type);
  }
  private Token token(Sym sym, String value) {
     Sym s = keys.get(value);
     if (s == null) { /* not a keyword */
        return new StringToken(sym, value);
     } else {  /* keyword */
        return new Token(s);
     }
  }
%}

%%
{Letter}+      {return token(Sym.IDENT, yytext());}
{EspaceChar}   {}
```