Analyse Lexicale

Ralf Treinen

Université Paris Cité
UFR Informatique
Institut de Recherche en Informatique Fondamentale

treinen@irif.fr

6 novembre 2023

Analyse lexicale et Analyse syntaxique

- *L'analyse syntaxique* est le processus de la traduction d'un texte d'entrée (par exemple un code source écrit en Java ou OCaml) en une représentation abstraite, appelée la *syntaxe abstraite*.
- Deux objectifs principaux :
  - faire abstraction autant que possible des détails d'écriture inutiles pour la suite (par exemple nombre d'espaces, les commentaires) ;
  - représenter explicitement la structure du texte analysé, par exemple sous forme d'un *arbre*.
- C'est en particulier la première partie d'un *compilateur*.

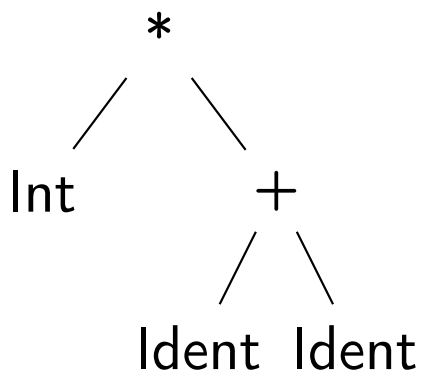
Entrée et résultat de l'analyse syntaxique

Exemple : expressions arithmétiques.

Texte d'entrée :

(	7	5	6	e	2		*		(	e	5	e	7		+	v	2	)	)

Résultat attendu de l'analyse syntaxique :



Les limites de l'analyse lexicale

- Les automates ne sont pas suffisants pour réaliser tout ce programme d'analyse syntaxique.
- Les automates ne peuvent pas reconnaître des structures imbriquées (par exemples des expressions parenthésées).
- On verra la semaine prochaine plus précisément pourquoi pas.
- Les automates sont utiles pour faire une première étape d'*analyse lexicale* : découper un texte d'entrée en morceaux, appelés des *lexèmes*, selon une spécification sous forme d'expressions rationnelles.

Spécification d'une analyse lexicale

- Dans un premier temps on peut imaginer qu'une spécification consiste en :
  - une liste d'expressions rationnelles pour les *lexemes* qu'on souhaite reconnaître,
  - et pour chaque expression rationnelle une valeur symbolique à engendrer, le *jeton* associé (anglais : *token*).
- En plus on spécifie sous forme d'une expression rationnelle les morceaux de texte pour lesquelles on ne veut pas engendrer de jetons (espaces, commentaires, etc.)

Associer des jetons à des expressions rationnelles

- On écrit les expressions rationnelles "à la grep".
- Sur l'exemple des expressions arithmétiques :

expression rationnelle	jeton
(0|[1..9][0..9]*)(e[0..9]+)?	INT
[a..zA..Z][a..zA..Z0..9]*	IDENT
(	PARG
)	PARD
*	MULT
+	PLUS

- en plus il faut spécifier qu'on souhaite "sauter" les espaces.

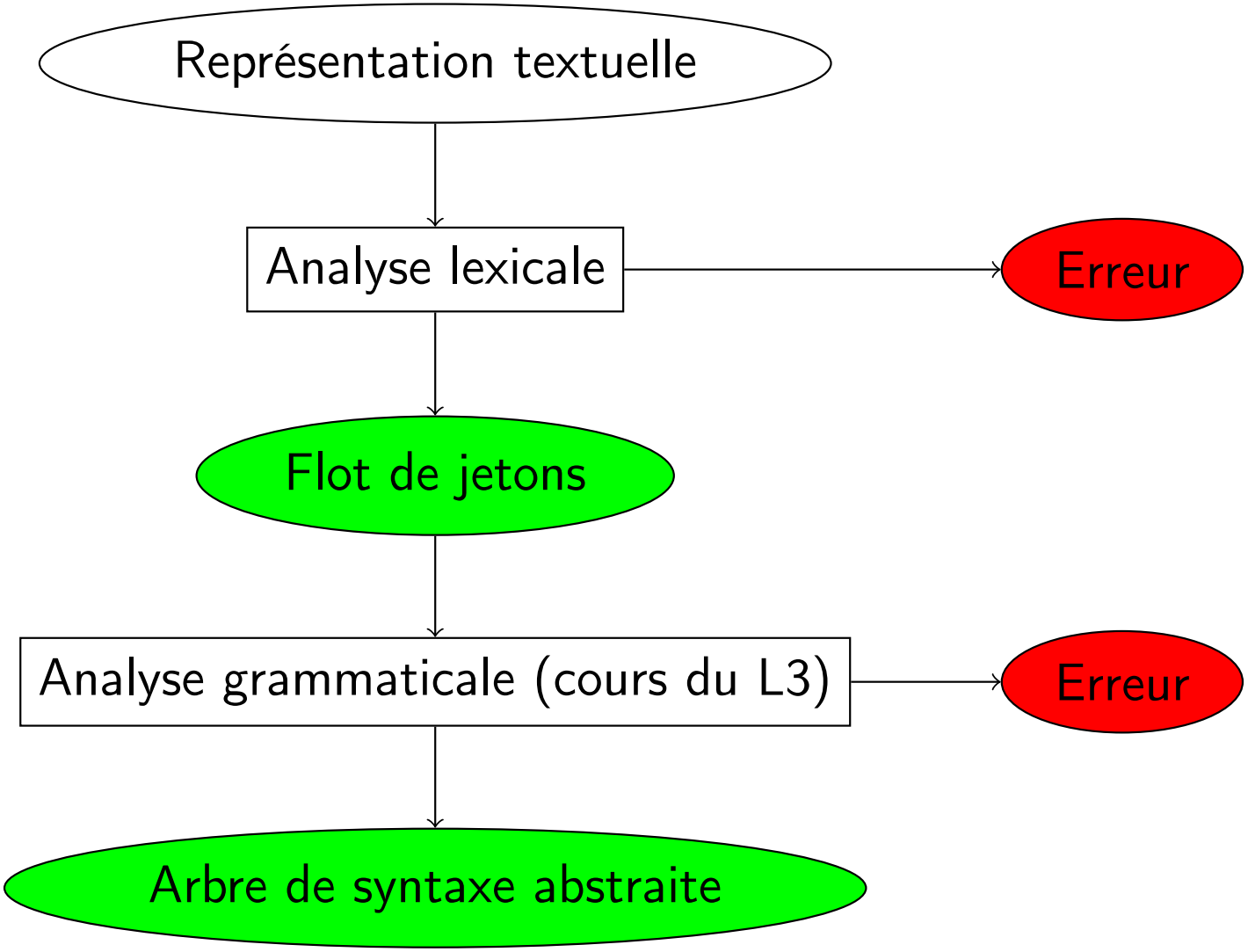
Exemple

- Texte d'entrée :

(	7	5	6	e	2		*		(	e	5	e	7		+		v	2	)	)

- Jetons trouvés par l'analyse lexicale :
  PARG INT MULT PARG IDENT PLUS IDENT PARD PARD

Les deux phases de l'analyse syntaxique



Jetons avec arguments

- En réalité, on veut aussi garder certaines informations avec les jetons, comme la valeur d'une constante entière, ou le nom d'un identificateur.
- Certains jetons doivent donc avoir un argument :
  - IDENT(string)
  - INT(int) (c'est bien `int` et pas `string` !)
- Séquence des jetons obtenue sur l'exemple :
  PARG INT(75600) MULT PARG IDENT("e5e7") PLUS IDENT("v2") PARD PARD

Ignorer des informations pas pertinentes

L'analyse lexicale sert aussi à faire abstraction de certaines informations dans le texte d'entrée qui ne sont pas pertinentes pour l'analyse du texte. Souvent il s'agit de :

- Les espaces : sont utiles pour indiquer la fin d'un mot. Les espaces sont utiles *pour* l'analyse lexicale, mais une fois le découpage fait on peut les oublier.
- Les commentaires : souvent l'analyse lexicale vérifie l'écriture correcte des commentaires, mais ne les représente pas dans sa sortie.

Exemple

Différents textes d'entrée qui peuvent donner la même séquence de jetons :

- `34 * (x + y)`
- `34*(x+y)`
- `34 *( x+ y)`
- `34 * (x+y) /* Ceci est un commentaire */`

Quelle information retenir dans les jetons

- On retient dans les jetons seulement l'information qui est utile pour la suite.
- La distinction entre information utile/inutile dépend de l'application.
- Par exemple : Les commentaires peuvent être utiles à retenir pour certaines applications.
- Il peut être utile de conserver avec les jetons aussi des informations de *localisation* : nom du fichier source, numéro de ligne, numéro de colonne.

Résoudre les ambiguïtés

- L'analyse lexicale va, pour produire le jeton suivant, chercher un *préfixe* du reste du texte qui correspond à une des expressions rationnelles, et construire le jeton correspondant.
- Il y a deux sources d'ambiguïtés :
  - Des préfixes de longueurs différentes peuvent être reconnus.

    Cette ambiguïté vient du fait que le problème est plus complexe que l'acceptation par un automate : maintenant on veut *découper* l'entrée selon des expressions rationnelles.
  - Les expressions régulières peuvent avoir une intersection. non vide.

    Cette ambiguïté vient du fait qu'on veut savoir par laquelle des expressions rationnelles un mot est reconnu.

Préfixes de longueur différentes reconnues

Exemple :

- Spécification :

expression rationnelle	jeton
`[a..z]+`	IDENT

- Texte d'entrée :

  `xyz`

- Plusieurs possibilités de découpage :
  1. `IDENT("x") IDENT("y") IDENT("z")`
  2. `IDENT("xy") IDENT("z")`
  3. `IDENT("x") IDENT("yz")`
  4. `IDENT("xyz")`

- La règle normale est : on cherche le préfixe *maximal*. Dans l'exemple ça donne `IDENT("xyz")`.

Plusieurs expressions régulières s'appliquent

Exemple :

- Spécification :

expression rationnelle	jeton
`public`	`PUBLIC`
[a..z]+	`IDENT`
[0..9]+	`INT`

- Texte d'entrée :

  `public12345`

- Plusieurs possibilités de création de jetons :
  1. `PUBLIC INT(12345)`
  2. `IDENT("public") INT(12345)`
- La règle normale est : à longueur égale du mot reconnu, c'est le premier cas dans la liste qui gagne.

Construction d'un automate

- Étant données les expressions rationnelles $r_1, \ldots, r_n$ dans la spécification.
- Il faut faire attention qu'on obtient à la fin l'information *laquelle* des expressions s'est appliquée.
- Construire des automates $A_1, \ldots, A_n$ pour les expressions, selon Thompson (par exemple).
- Créer un nouvel état initial, avec des ϵ-transitions vers les états initiaux de $A_1, \ldots, A_n$.
- Éliminer les ϵ-transitions.
- Déterminiser.
- Mais attention comment cet automate sera exécuté ...

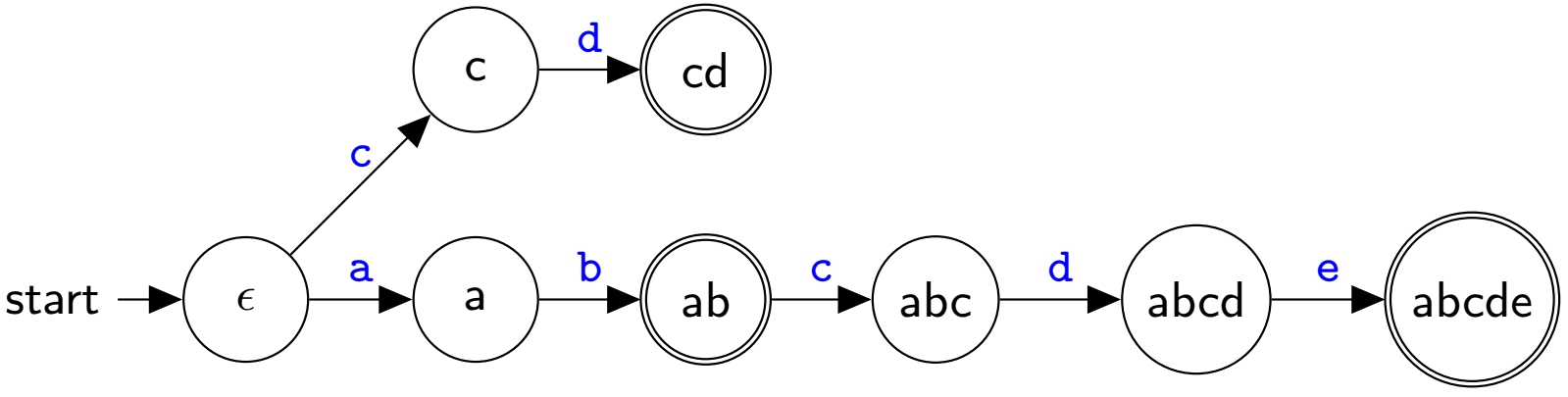
Exécution de l'automate pour le recherche d'un préfixe maximal

Exemple (artificiel) :

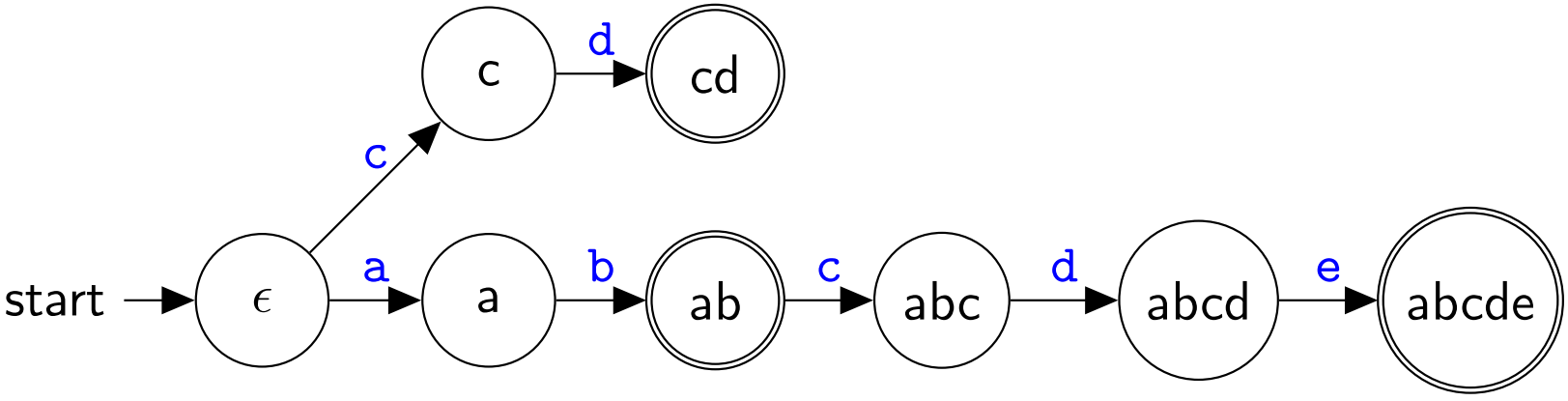
- Spécification :

expression rationnelle	jeton
ab	AB
cd	CD
abcde	ABCDE

- Automate déterministe :



Exemple



- Supposons qu'on cherche le préfixe maximal de l'entrée qui est reconnu par l'automate (cas normal).
- Entrée : `abcd`
- L'automate consomme tout le mot, et arrive dans l'état `abcd`.
- Il aurait dû trouver `ab` !

Exécution de l'automate pour le recherche d'un préfixe maximal

- Si on cherche le préfixe le plus *court* reconnu on doit s'arrêter dès qu'on arrive dans un état acceptant.
- Si on cherche le préfixe le plus *long* reconnu on doit continuer à lire tant que possible, et quand on passe par un état acceptant :
  - mémoriser l'état acceptant ;
  - mémoriser la position dans le mot d'entrée.

  Si l'automate ne peut plus continuer : remettre le pointeur de lecture dans l'entrée à la position où on a vu le dernier état acceptant.

Conclusion jusqu'ici : 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. Ces valeurs seront attachées au jetons.
- *Abstraire* l'entrée : ignorer des détails non pertinents pour la suite (espaces, commentaires, ...)

Implémenter une analyse lexicale

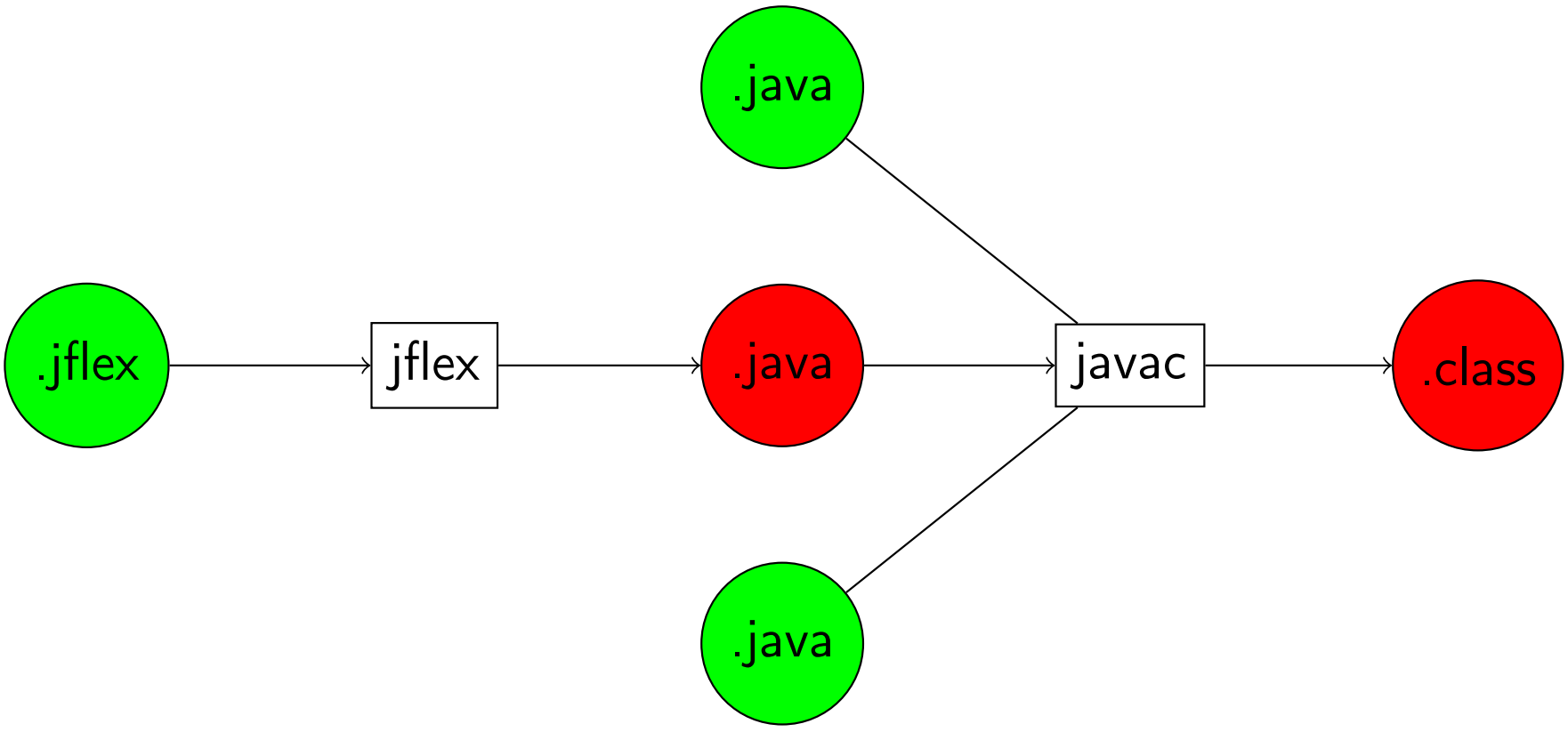
Il y a deux possibilités :

- Soit *écrire un programme* (Java ou autre) à la main, basé sur un automate fini.
  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.
  C'est la technique utilisée pour l'écriture des compilateurs modernes.

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.

Utilisation de jflex



Vert : écrit par le programmeur. Rouge : engendré.

Le code engendré dans le cas 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 a 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 %% :

1. Code utilisateur
2. Options et déclarations
3. Règles lexicales

Fichier arith.flex I

```
// partie code utilisateur : vide

%%
// partie options et déclarations
%public
%class Lexer
%unicode
%type Token

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

%%
// partie règles lexicales
{Ch}+                 {return new IntToken(Sym.INT, Integer.parse
{Le}({Le}|{Ch})*      {return new StringToken(Sym.IDENT, yytext()
"("                   {return new Token(Sym.PARG);}
```

Fichier `arith.flex` II

```
")"                 {return new Token(Sym.PARD);}
"*"                 {return new Token(Sym.MULT);}
"+"                 {return new Token(Sym.PLUS);}
{EspaceChar}+       {}
```

La partie *Code utilisateur*

- copiée simplement au début du fichier engendré (devant 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 (recommandé).
- `%type` *t* : le type de résultat de la méthode principale de l'analyse lexicale `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)

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`).

La partie *Options et Déclarations*

Macros : système de définitions d'expressions rationnelles

- mettre les mots entre apostrophes " et "
- pour utiliser une expression rationnelle préalablement définie, par exemple de nom *r* : `{r}`.
- classes de caractères, par exemple `[a-z]`
- quelque classes de caractères prédéfinies, par exemple `[:letter:]`, `[:digit:]`, `[:uppercase:]`, `[:lowercase:]`.

La partie *Règles Lexicales*

- Séquence de
  expression-rationnelle { *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 rationnelle qui s'applique.
- S'il n'y a pas d'instruction **return** dans cette action (et pas d'erreur) alors on continu à chercher le lexeme suivant.
- Cas normal : pas de **return** pour les espaces car les espaces ont le seul rôle de *séparer* les lexemes qui donnent lieu à l'envoi d'un jeton.

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.
- Regardons le code !

Ce que JFlex fait pour vous

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

Ce que JFlex fait pour vous

- Création d'un automate non-déterministe.
- Éliminer les ϵ-transitions.
- Déterminiser l'automate.
- Minimiser l'automate.
- On peut demander à `jflex` de montrer ces trois automates (option `-dot`, visualiser les automates avec `xdot` par exemple)
- Il y a également un mode autonome (*standalone*) pour des applications simple (pas de génération de jetons), voir le TP.

Les états de l'analyseur lexical

- Par défaut (comme sur le premier exemple), votre analyseur lexical a un seul état.
- Il peut être utile d'avoir plusieurs états — dans chaque état, JFlex peut utiliser des expressions rationnelles différentes.
- Pour en avoir plusieurs :
  - les déclarer à l'aide de `%state` (sauf YYINITIAL, qui est l'état par défaut) ;
  - mettre toutes les règles dans le contexte d'un état ;
  - dans les actions : changer d'état à l'aide de yybegin.
- Ne pas confondre les états de Flex avec les états de l'automate fini obtenu à partir des expressions rationnelles.

Pourquoi utiliser plusieurs états ?

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

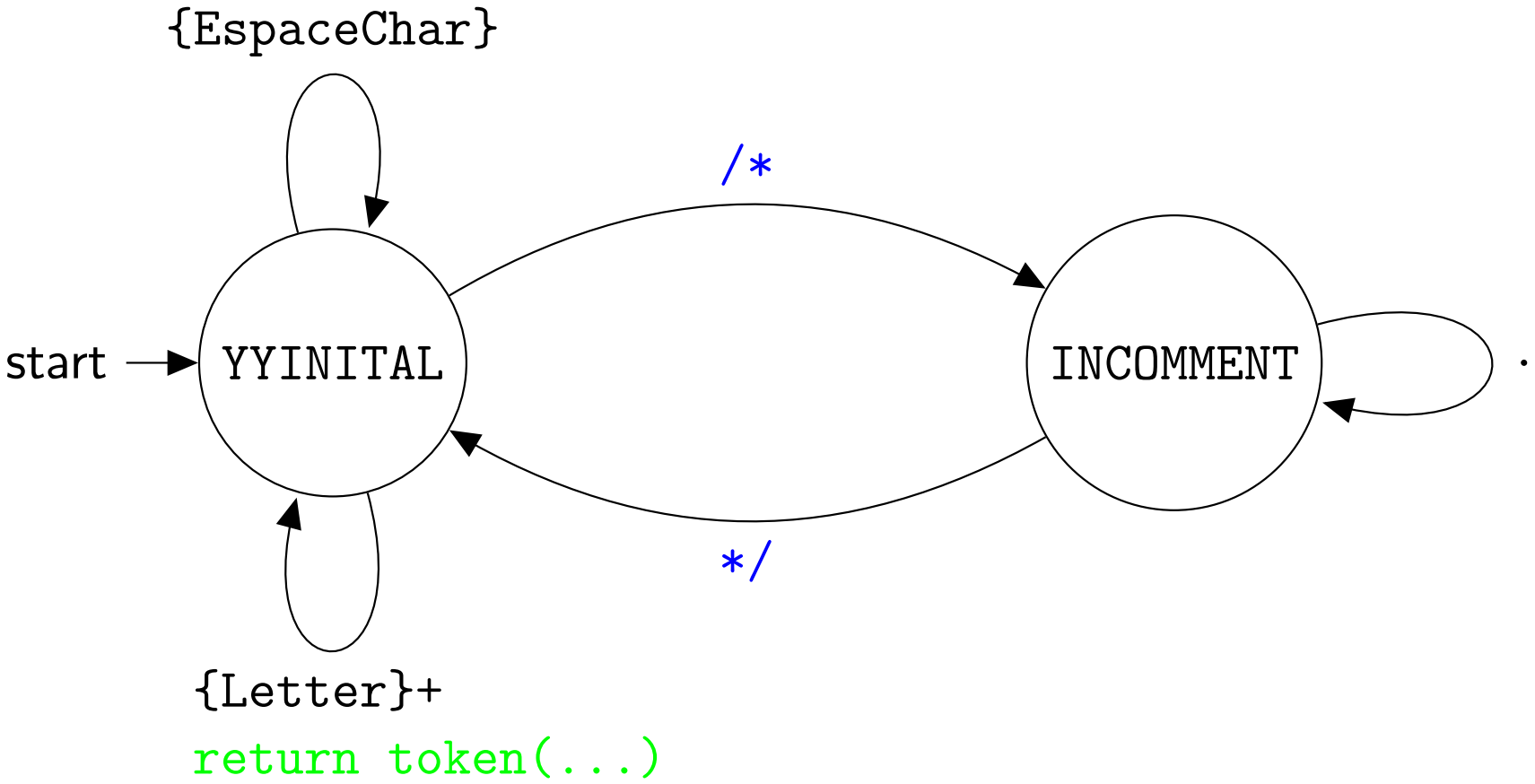
Reconnaître les commentaires (simplifié)

```
%%
%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 important que la dernière règle s'applique à un mot de longueur 1 seulement (c.-à-d. expression rationnelle `.`, et pas `.+`)

Les états de Flex dans l'exemple des commentaires



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 (devant le “e”) dans une variable, puis on va dans un deuxième état pour lire exposant.

Nouvelle version de arith.flex I

```
%%

%public
%class Lexer
%type Token
%unicode
%state EXPONENT

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

Nouvelle version de arith.flex II

```
            result=result*10;
        }
        return result;
    }
    %}

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

    %%

    <YYINITIAL> {
      {Ch}+                 {return new IntToken(Sym.INT,
                                 Integer.parseInt(yytext()));}
      {Le}({Le}|{Ch})*      {return new StringToken(Sym.IDENT,
                                 yytext());}
      "("                   {return new Token(Sym.PARG);}
```

Nouvelle version de `arith.flex` III

```
  ")"                 {return new Token(Sym.PARD);}
  "*"                 {return new Token(Sym.MULT);}
  "+"                 {return new Token(Sym.PLUS);}
  {EspaceChar}+       {}
  {Ch}+ "e"           {intbuff=Integer.parseInt(chop(yytext()));
                              yybegin(EXPONENT);}
}

<EXPONENT> {
  {Ch}+         {yybegin(YYINITIAL);
                 return new IntToken(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.

Attention la taille de l'automate engendré !

Contrôler la taille de l'automate

- Techniques utilisés par le générateurs :
  - Utiliser des classes de caractères 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 ident_or_keyword(String lexeme) {
     Sym s = keys.get(lexeme);
     if (s == null) { /* not a keyword */
        return new StringToken(Sym.IDENT, lexeme);
     } else {  /* keyword */
        return new Token(s);
     }
  }
%}

%%
{Letter}+      {return ident_or_keyword(yytext());}
{EspaceChar}   {}
```