Compilation

IUT Informatique
Université Paris-Saclay

CHAPITRE 3 : BASES DE LEX

Analyse lexicale (rappel)

- L'analyse lexicale consiste à déterminer le, « statut » de chaque mot, c'est-à-dire **l'unité lexicale** (ou **token,** ou **lexème**) qui lui correspond.
- Les unités lexicales sont généralement définies par des expressions rationnelles.
- Le problème de base à résoudre est donc :
  - Données : un mot w, un langage rationnel L (donné par une expression rationnelle)
  - Question : **w ∈ L ?**

Analyse lexicale (suite)

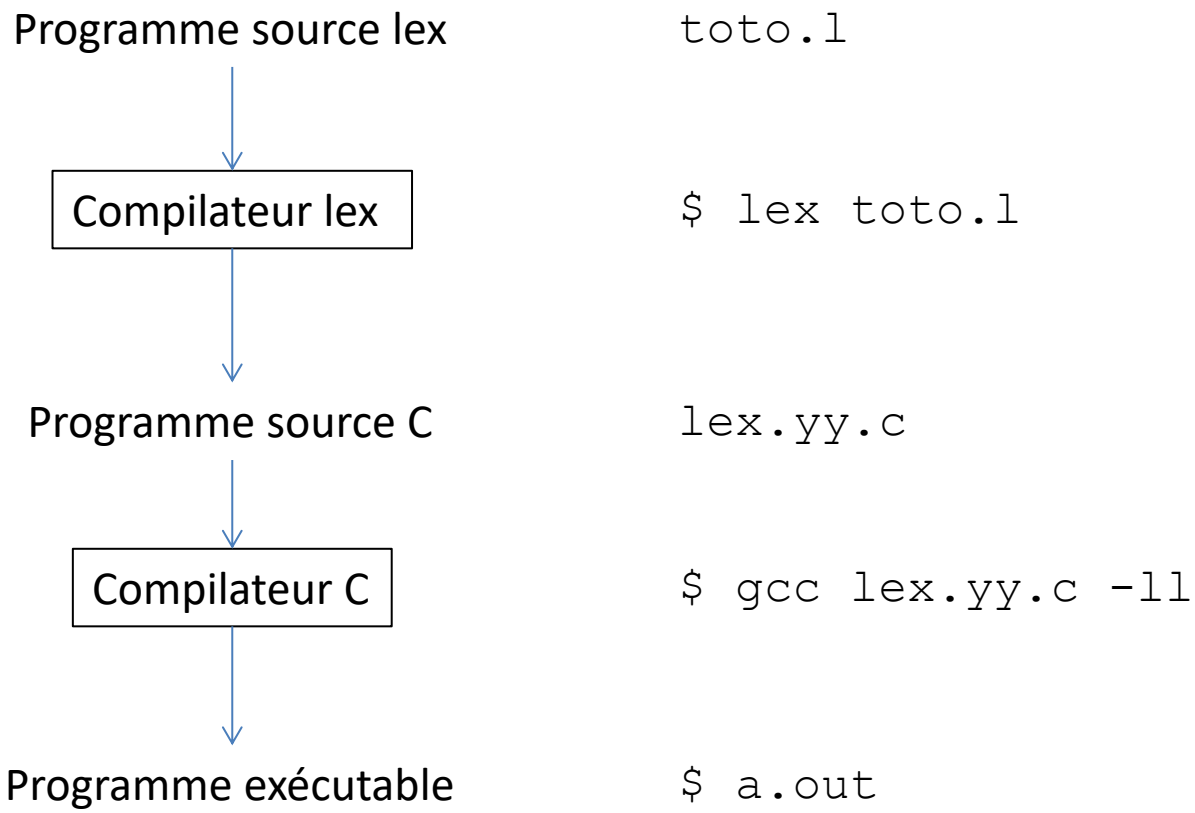
- Le problème plus général de l'analyse lexicale est :
  - Données :
    - un (long) mot w (= le texte à analyser)
    - Un ensemble de langages L_1, L_2, …, L_k définis par des expressions rationnelles.
  - Problème : décomposer w en une concaténation de mots $w_1w_2…w_n$ de sorte que chacun des w_i appartient à un L_j.
- Si une telle décomposition n'est pas possible, le texte est lexicalement incorrect.
- Si la décomposition est possible, l'analyseur effectuera une ou des actions pour chaque w_i. (La décomposition doit être unique.)

Analyse lexicale : méthode

- Idée générale : construire un automate fini à partir de l'expression rationnelle.
- (Rappel du théorème de Kleene : tout langage rationnel est reconnaissable.)

- Lex permet de programmer des analyseurs lexicaux en s'affranchissant totalement des problèmes de création et de gestion des automates.

Lex : schéma d'utilisation



Lex : généralités

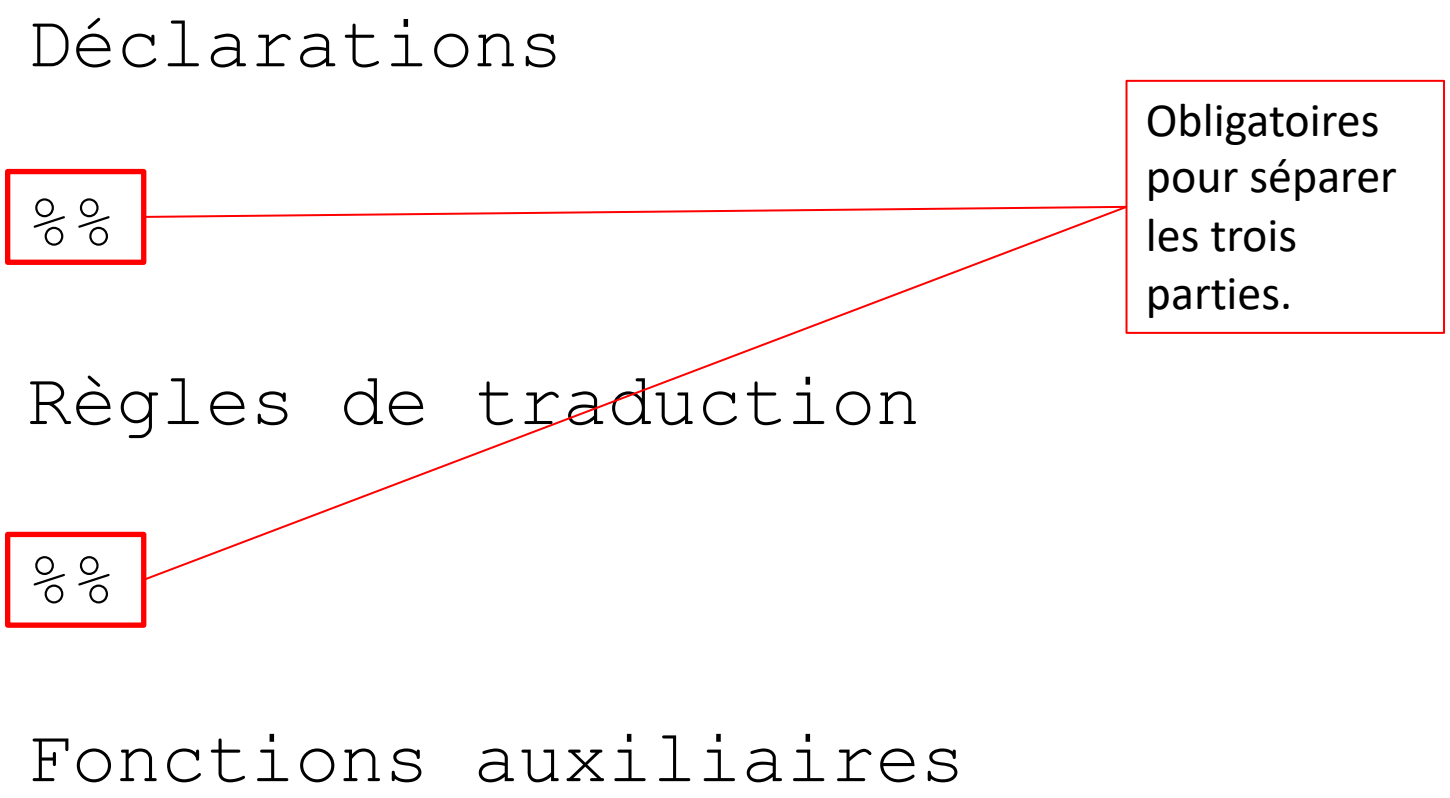
- Histoire
  - 1975 développé aux laboratoires Bell
  - A l'origine un outil d'Unix, aujourd'hui aussi pour Windows
  - A l'origine pour C, aujourd'hui aussi pour Java
  - Coopère généralement avec *Yacc*

- Versions actuelles
- flex version GNU de Lex (pour C)
  http://www.gnu.org/software/flex/
- JLex version Java (avec légères différences dans la syntaxe)
  http://www.cs.princeton.edu/~appel/modern/java/JLex/
- CsLex version C# , dérivé de JLex

Structure d’un programme lex

```
Déclarations

%%

Règles de traduction

%%

Fonctions auxiliaires
```

Structure d'un programme lex



Partie 1 : Déclarations

- Cette partie peut comprendre
  - des déclarations de variables et de constantes en C, délimitées par « %{« et « %} » (les délimitateurs sont toujours en début de ligne).
  - des « définitions régulières » qui permettent de donner des noms à des expressions régulières qui seront utilisées dans les règles de traduction.

- (Cette partie peut aussi être vide.)

Partie 2 : Règles de traduction

- Elles sont de la forme suivante :

  m_1 {action 1}
  m_2 {action 2}
  ...
  m_n {action n}

  où les m_i sont des expressions régulières lex et les actions sont des suites d'instructions en C. Elles décrivent ce que l'analyseur lexical doit faire lorsqu'il reconnaît un lexème (un mot du texte) qui appartient au langage m_i correspondant.

Fonctionnement général

- L'analyseur lexical produit par lex

- commence à chercher les lexèmes au début du code source ;

- après chaque lexème reconnu, recommence à chercher les lexèmes **juste après.**

Exemple : si *piraté* est reconnu, *raté* n'est pas reconnu

- **Si aucun lexème n'est reconnu à partir d'un point du code source,** l'analyseur affiche le premier caractère sur la sortie standard et recommence à chercher à partir du caractère suivant

Gloutonnerie et priorité

- Lex est **glouton** : il essaie toujours de reconnaître le mot le plus long possible.

- Lorsqu'un même mot peut correspondre à deux expressions m_i différentes, c'est la première qui est prise en compte.

Expressions régulières en lex

- Alphabets:
  - codes ISO, ASCII, etc

- Expressions régulières
  - forme de Kleene via des méta-opérateurs

    (concaténation, le • est omis)

    | *(alternative)*

    ✶ *(répétition)*

    ( )
  - exemple: les identificateurs C
    - *Id* = (a|b|..|z|A|..|Z|_) (a|b|..|z|A|..|Z|_|0|..|9)✶

Expressions régulières en lex

- Expressions régulières étendues
  - méta-opérateurs
    - [ ] - + ? • ^
    - " " \
  - exemples
    - *les entiers:* [0-9]*+*
    - *les identificateurs C:* [a-zA-Z_] [a-zA-Z0-9_]✶
    - *les chaînes de caractères sans " :* \" [^"]✶ \"
    - *les commentaires lignes Java:* "/ /"•✶

Expressions régulières en lex

- Les caractères terminaux
  - tous les caractères, sauf les spéciaux
  - l ’espace est significatif
  - les spéciaux doivent être protégés par ″ " ou \
- Les caractères spéciaux (méta-)

Les expressions autorisées

Opérations rationnelles	
e?	*0 ou 1 fois l'exp e*
*e**	*0 ou n fois l'exp e (n quelconque)*
e+	*1 ou n fois l'exp e (n quelconque)*
e f	*l'exp e f*
e | f	*l'exp e ou l'exp f*
e{n,m}	*l'exp e répétée entre n et m fois*
(e)	*l'exp e*
{D}	*l'exp obtenue par substitution de D (macro)*
Sensibilité au contexte	
e / f	*l'exp e si suivie de l'exp f*
^e	*exp e en début de ligne*
e$	*exp e en fin de ligne*
<E>e	*L'exp e si dans l'état E*
Caractères	
.	*tout caractère sauf \n*
\c	*le caractère c , même spécial*
"abc"	*la chaine de caractères abc*
[abc]	*le caractère a ou b ou c*
[^abc]	*un des caractères sauf a, b, c*
[a-c]	*le caractère a ou b ou c*

Partie 3 : Fonctions auxiliaires

- Ce sont des fonctions C qui sont utilisées pour l'analyse lexicale. En général, cette partie contient la fonction main().

- Toutefois cette partie peut être vide. Dans ce cas, tout se passe comme si on y avait écrit :

```
main() {
      yylex()
}
```

- La fonction `yylex()` est la fonction « standard » d'analyse lexicale. Son rôle est de lire le texte en entrée standard *lettre par lettre* et d'appliquer les règles définies dans la partie 2 du programme lex.

Spécification d'un programme

- On veut écrire un programme qui prend en entrée un texte sur l'entrée standard, et qui reconnaît les constantes entières (non signées) et réelles (avec le « . » comme séparateur) dans le texte.
- Exemple : si on donne en entrée le texte suivant :

```
234 Ab 56.9 -10
```

il doit écrire sur la sortie standard :

```
Un entier : 234
Un réel : 56.9
Un entier : 10
```

Exemple de programme lex

```
%%

[0-9]+                  {
                            printf(‘’Un entier : %s\n ’’, yytext);
                        }

[0-9] +’’.’’[0-9]*      {
                            printf(‘’Un réel : %s\n ’’, yytext);
                        }

.                       {}

%%
```

Variables spéciales de lex

Ces variables sont globales, ce sont les attributs de l'unité lexicale (lexème) en cours de traitement.

- `yytext` : contient le lexème.
- `yyleng` : longueur de yytext.
- `yylval` : la valeur du lexème, si celui-ci a une valeur numérique.
- `yylineno` : numéro de ligne du lexème.

Compilation et exécution du programme

```
> lex nombres.l
> gcc lex.yy.c -ll
> cat texte1.txt
5666 788 78.9
> a.out < texte1.txt
Un entier : 5666
Un entier : 788
Un réel : 78.9
```

```
> cat texte2.txt
234Ab56.9,87 -10
> a.out < texte2.txt
Un entier : 234
Un réel : 56.9
Un entier : 87
Un entier : 10
```

Exemple de programme lex

```
%%

[0-9]+                          {
                                        printf(‘’Un entier : %s\n ’’, yytext);
                                }

[0-9] +’’.’’[0-9]*    {
                                        printf(‘’Un réel : %s\n ’’, yytext);
                                }

.                               {}

%%
```

Supposons qu’on supprime cette ligne. Que se passe-t-il ?

Compilation et exécution du programme

```
> lex nombres.l
> gcc lex.yy.c -ll
> cat texte1.txt
5666 788 78.9
> a.out < texte1.txt
Un entier : 5666
 Un entier : 788
 Un réel : 78.9
```

```
> cat texte2.txt
234Ab56.9,87 -10
> a.out < texte2.txt
Un entier : 234
AbUn réel : 56.9
,Un entier : 87
 -Un entier : 10
```

Variante du programme

```
DIGIT    [0-9]

%%

{DIGIT}+                    {
                                printf(‘’Un entier : %s\n ’’, yytext);
                            }

{DIGIT} +’’.’’{DIGIT}*      {
                                printf(‘’Un réel : %s\n ’’, yytext);
                            }

.                           {}

%%
```

Commentaires en lex

```
%{
/* ce programme lex reconnaît les entiers et les réels. */
%}

DIGIT    [0-9]

%%

{DIGIT}+                    {
                                /* traitement d'un entier */
                                printf(''Un entier : %s\n '', yytext);
                            }

{DIGIT} +''.''{DIGIT}*    {
                                /* traitement d'un réel */
                                printf(''Un réel : %s\n '', yytext);
                            }

.                           {}

%%
```