# Lex

R. Raffin

Université de Bourgogne

janvier 2022

# Plan intermédiaire

1 Introduction
2 premier exemple
3 Exemples plus complexes
4 Expressions régulières Lex
5 Règles
6 Définitions Lex

# What's Lex ?

Lex est un générateur d'analyse lexicale. Il a été écrit en 1975 par Mike Lesk et Eric Schmidt.

La version open-source libre (GNU) est `flex`. La commande `lex` existe toujours et fait appel à `flex`.

Son rôle n'est pas de reconnaître des symboles ou des mots mais de construire un analyseur permettant de reconnaître les unités lexicales.

# Processus de création de l'analyseur

On doit donc (une fois le langage analysé), définir des règles d'analyse, définir les actions associées, produire un automate, l'utiliser dans un programme.

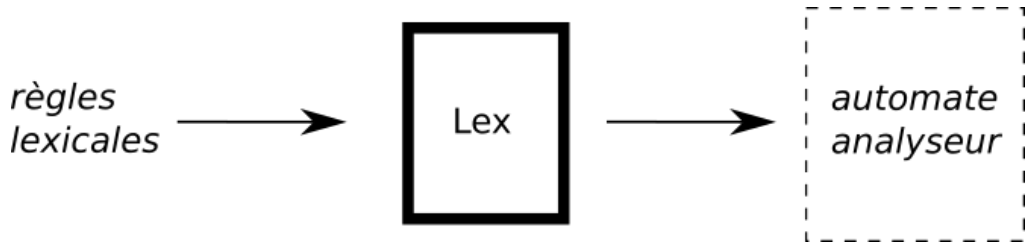
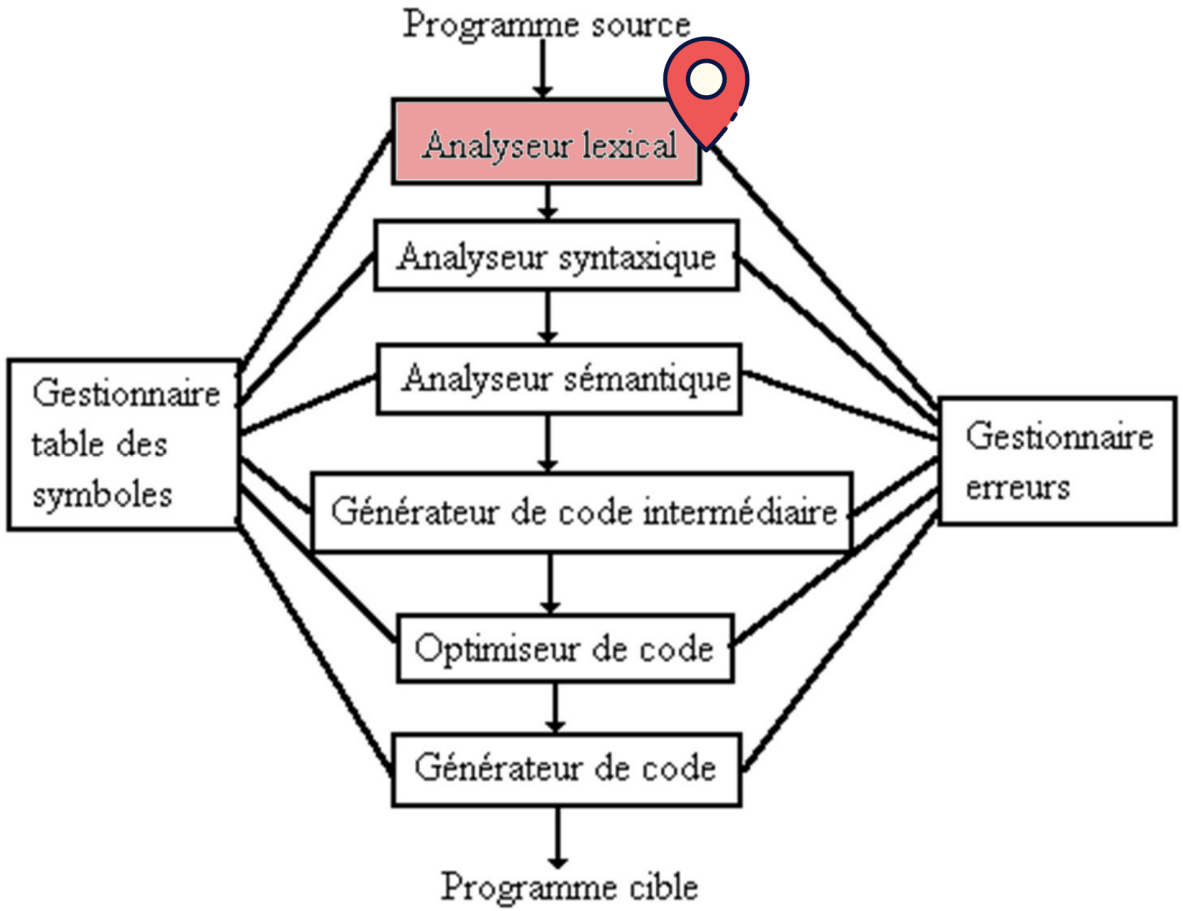



Figure – Étapes de construction

# Dans une chaîne de compilation...

(cf. le cours d'I. Foucherot)



# Reconnaissance et actions

L'analyse et la reconnaissance d'expressions est faite grâce à un automate fini déterministe.

Lex produit une liste d'expressions régulières, dans un programme. Ce dernier analysera séquentiellement une entrée et produira des actions dès qu'une chaîne sera reconnue.
Le programme produit par Lex est écrit en C.

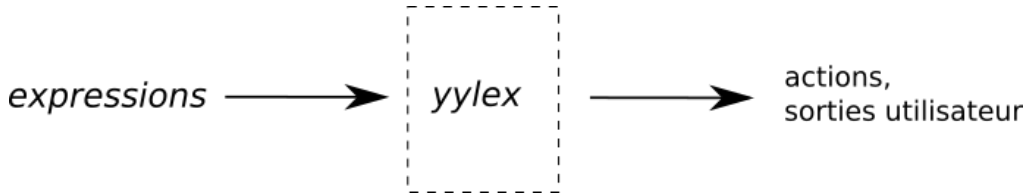



Figure – Utilisation

# Et ensuite ?

Un autre outil (Yacc), que nous utiliserons par la suite, nous permettra d'effectuer une analyse syntaxique d'une source.

Il s'interface assez facilement avec Lex pour construire un processus d'analyse ou un début de compilateur.

# Plan intermédiaire

1 Introduction

2 premier exemple

3 Exemples plus complexes

4 Expressions régulières Lex

5 Règles

6 Définitions Lex

# Premier exemple

Pour pouvoir avancer sur Lex, essayons un premier exemple :

Listing 1 – Premier essai

```
%{
#include <stdio.h>
%}
%%
gentil {
        printf("\tbien !\n");
}
méchant {
        printf("\tpas bien ¡\n");
}
%%
```

Objectif : isoler les unités lexicales dans ce programme, comprendre les actions effectuées.

# Explications

## Blocs et syntaxe

On a vu dans le source précédent des blocs de code, définis par % ou %% et 2 mots : `gentil` et `méchant`.

Les instructions % ... % seront utilisés par le code C produit. On y met les `include`, des variables globales...

Listing 2 – Premier essai

```
%{
#include <stdio.h>
%}
```

**Attention :**

Les % ... % doivent débuter le fichier Lex, pas de commentaire avant !

# Explications

## Blocs et syntaxe

On a vu dans le source précédent des blocs de code, définis par % ou %% et 2 mots : `gentil` et `méchant`.

Les instructions %% ... %% seront utilisés construire l'analyseur. Chaque mot est listé, avec les actions à entreprendre (code C entre {}).

Listing 3 – Premier essai

```
%%
gentil {
        printf("\tbien !\n");
}
méchant {
        printf("\tpas bien ¡\n");
}
%%
```

### Mots

1. le mot "gentil" doit déclencher le `printf("bien !");`
2. le mot"méchant" doit déclencher le `printf("pas bien ¡");`

# Explications

## Vous n'avez pas tout vu...

Le source précédent est dans le fichier « premierEssai.lex ». Ce fichier est analysé par le programme `lex` :

**construction analyseur**

```
lex -v premierEssai.lex
```

Celui-ci produit un fichier source en C : « lex.yy.c ». Il faut ensuite compiler ce fichier C pour obtenir l'analyseur exécutable :

**compilation analyseur**

```
gcc -Wall -o premierEssai lex.yy.c -lfl
```

Le fichier C produit est « lisible »...

# Plan intermédiaire

1 Introduction

2 premier exemple

3 Exemples plus complexes

4 Expressions régulières Lex

5 Règles

6 Définitions Lex

# Variables C, yytext, yyleng, yywrap()

Les 2 mots sont ici « debut » et « fin ». On veut compter le nombre de « debut ».

Listing 4 – Plus d'interactions

```
%{
#include <stdio.h>
int cpt=0;//une variable dans le programme en C
%}

%%
debut {
        cpt++; //incrémenté àchaque "debut"
        printf(" (%d) --> debut ok text: %s len: %d\n", cpt, yytext, yyleng);//yytexte le texte du token est
                trouvé (ici "debut"), yyleng sa longueur (ici 5)
}

fin printf("--> fin ok\n");

%%
yywrap(){ //fonction appelée àla fin de l'analyse, recopie àla fin de lex.yy.c
        printf("%d\n", cpt);
        return (1);//1 signifie "c'est terminé"
}
```

# Plan intermédiaire

1 Introduction

2 premier exemple

3 Exemples plus complexes

4 Expressions régulières Lex

5 Règles

6 Définitions Lex

# Opérateurs

La liste est la suivante :

```
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >
```

- `"` indique un ensemble de caractères interprétés comme du texte : `"a+b = ?"`, au lieu de `a\+b = \?`

Attention : pour utiliser un opérateur comme un caractère, il faut « l'échapper » : `\?` ou `\+`. Les `\n`, `\t`, `\b` (*backspace*) du C sont reconnus.

# Opérateurs

La liste est la suivante :

`" \ [ ] ^ - ? . * + | ( ) $ / { } % < >`

- `[]` crée des **classes** de caractères : `[a-zA-Z]`, `[0-9]`, `[abc]` (1 seul caractère, parmi `a`, `b` ou `c`). 3 caractères ont une action spéciale :
  - `-` indique un intervalle, attention à l'interprétation de `[a-9]`, cela dépend de l'encodage. Le signe « moins » doit être placé au début : `[-+0-9]`
  - `.` indique un caractère quelconque,
  - `^`, positionné au début, indique le complément : `[^abc]` reconnaît tous les caractères sauf `a`, `b` ou `c`,

# Opérateurs

La liste est la suivante :

`" \ [ ] ^ - ? . * + | ( ) $ / { } % < >`

- **options** :
  - + et * les répétitions habituelles, {} nombre d'occurences : `[0-9]{3}` des chiffres par centaines. Pour un ensemble de chiffres (entre 1 et 3 occurences) `[0-9]{1,3}`,
  - ? s'applique à l'élément précédent, indique qu'il n'est pas nécessaire : `[-+]?[0-9]+` les nombres, signés ou non,
  - | indique l'alternative dans une classe, `[a-z|A-Z]` lettre minuscule ou majuscule,
  - () pour grouper les expressions,

# Opérateurs

La liste est la suivante :

`" \ [ ] ^ - ? . * + | ( ) $ / { } % < >`

- **contextes** :
  - `^` et `$` début et fin de ligne,
  - `/` dépendance du suivant (contraire de `?`) : `12/34` 1 ou 2 est reconnu si 3 ou 4 suit, `ab/\n` ≠ `ab$`,
  - `%` caractère spécial de Lex (section).

# Plan intermédiaire

1 Introduction

2 premier exemple

3 Exemples plus complexes

4 Expressions régulières Lex

5 Règles

6 Définitions Lex

# Actions possibles

Les règles représentent des actions confiées au programme en C construit par compilation. Ce programme à accès aux opérations habituelles (entrées-sorties, manipulation de données...) et peut interagir avec l'analyseur par des fonctions exposées :

- `yytext` est le texte reconnu (`char *`), `yyleng` le nombre de caractères,

# Actions possibles

- `REJECT` permet de finalement rejeter la règle et de recommencer, sans vider le *buffer* d'analyse. Le même ensemble de caractères peut être analysé plusieurs fois

Listing 5 – REJECT

```
%%
1234 { printf("N0 %s, %d", yytext, yyleng);
 REJECT;
}
12 { printf("N1");
 REJECT;
}
\n |
. ;
%%
```

# Actions possibles

- `yymore()` permet de sauvegarder le buffer reconnu courant, et de l'ajouter au prochain reconnu. `yyless()` permet, une fois la règle reconnue, de reculer dans le buffer courant.

Listing 6 – yymore()

```
%%
1234 { printf("N0 %s", yytext);
  yymore();
}
56 { printf("Complet %s", yytext);
}
\n |
. ;
%%
```

# En cas d'ambigüité ?

2 règles :

1. la chaîne la plus longue est préférée,
2. si 2 règles (ou plus) utilisent le même nombre de caractères, la règle donnée en premier est utilisée.

# Actions possibles

Plus globalement des fonctions sont accessibles :

- `yywrap()` est une fonction lancée à la fin de l'analyse Lex
- `yylex` démarre l'analyse (on peut définir un `main()`),
- `input()` retourne le prochain caractère,
- `output(c)` écrit le caractère 'c' dans la sortie courante,
- `unput(c)` remet le caractère 'c' dans le *buffer* pour la suite de l'analyse

# Exemple

```
%{
#include <stdio.h>
int c;
%}
%%
[a-z]+ { printf("[%s](%d)", yytext, yyleng);
 c++;
 //printf("prochain car: %c \n", input());
 }
%%

int main() {
c=0;
yylex();
return 0;
}

int yywrap() {
printf("nb de mots %d",c);
return 1;
}
```

# Exemple

```
%{
#include <stdio.h>
int c;
%}
%%
[a-z]+ { printf("[%s](%d)", yytext, yyleng);
 c++;
 //printf("prochain car: %c \n", input());
 }
%%

int main() {
c=0;
yylex();
return 0;
}

int yywrap() {
printf("nb de mots %d",c);
return 1;
}
```

Petit sondage : combien de mots seront trouvés dans ce fichier ?

```
abc
123
you and me
abc 123
et le piège 12332aab
```

# Plan intermédiaire

1 Introduction

2 premier exemple

3 Exemples plus complexes

4 Expressions régulières Lex

5 Règles

6 Définitions Lex

**Avant** la partie %% on peut inclure des définitions de mots qui seront traduits par Lex dans la partie de règles. Par exemple :

Listing 7 – Définitions régulières

```
INT [-+]?[1-9][0-9]*
TXT [a-z]+
P "."
REAL {INT}{P}{INT}
AFFECT "="
AFFECTATION {TXT}{AFFECT}({REAL}|{INT})

%%
{INT} { printf("int");
}
{TXT} { printf("txt");
}
{REAL} { printf("real");
}
^{AFFECTATION}$ { printf("affectation");
}
%%
```

# Construction

De l'utilité d'un makefile, d'un script ou autre pour construire son projet (une passe `lex` puis une passe `gcc`) :

Listing 8 – exemple de Makefile

```
default: $(EXE)

%: %.o
	$(CXX) -o $@ $^ $(LDFLAGS)

%.o: %.c
	$(CXX) -c $(CFLAGS) $<

%.c: %.lex
	$(LEX) -o $@ $<

clean:
	$(RM) -fv $(EXE) core.* *~ *.o
```

Fichtre ! Ce sont encore des expressions et des règles ☺