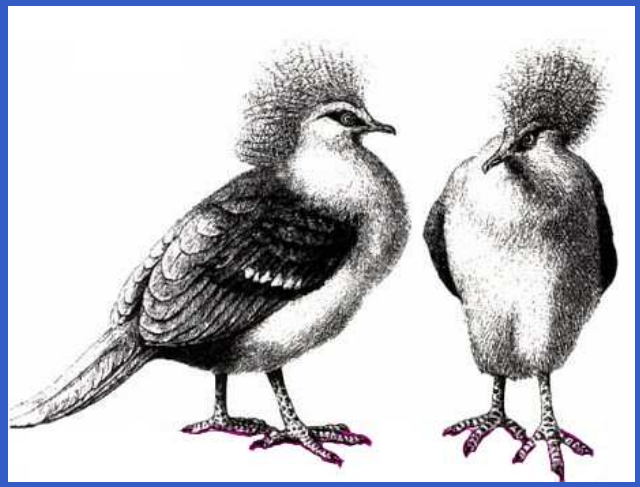
Le générateur de *scanners* (F)LEX



INFO010 – Théorie des langages – Partie pratique

S. COLLETTE et G. GEERAERTS

ULB Sciences – Informatique

Introduction – 1

- LEX est un outil qui permet de générer automatiquement un *scanner* à partir d'une spécification.

Introduction – 1

- LEX est un outil qui permet de générer automatiquement un *scanner* à partir d'une spécification.
- Il est souvent utilisé en conjonction avec YACC (générateur d'analyseur syntaxique), qui fera l'objet d'un prochain TP. . .

Introduction – 1

- LEX est un outil qui permet de générer automatiquement un *scanner* à partir d'une spécification.
- Il est souvent utilisé en conjonction avec YACC (générateur d'analyseur syntaxique), qui fera l'objet d'un prochain TP. . .
- Il existe une version GNU de LEX, qui s'appelle FLEX, ainsi que de YACC, qui s'appelle BISON.

Introduction – 2

- L'*input* de LEX est une spécification, formée de paires d'expressions régulières et de code C ;

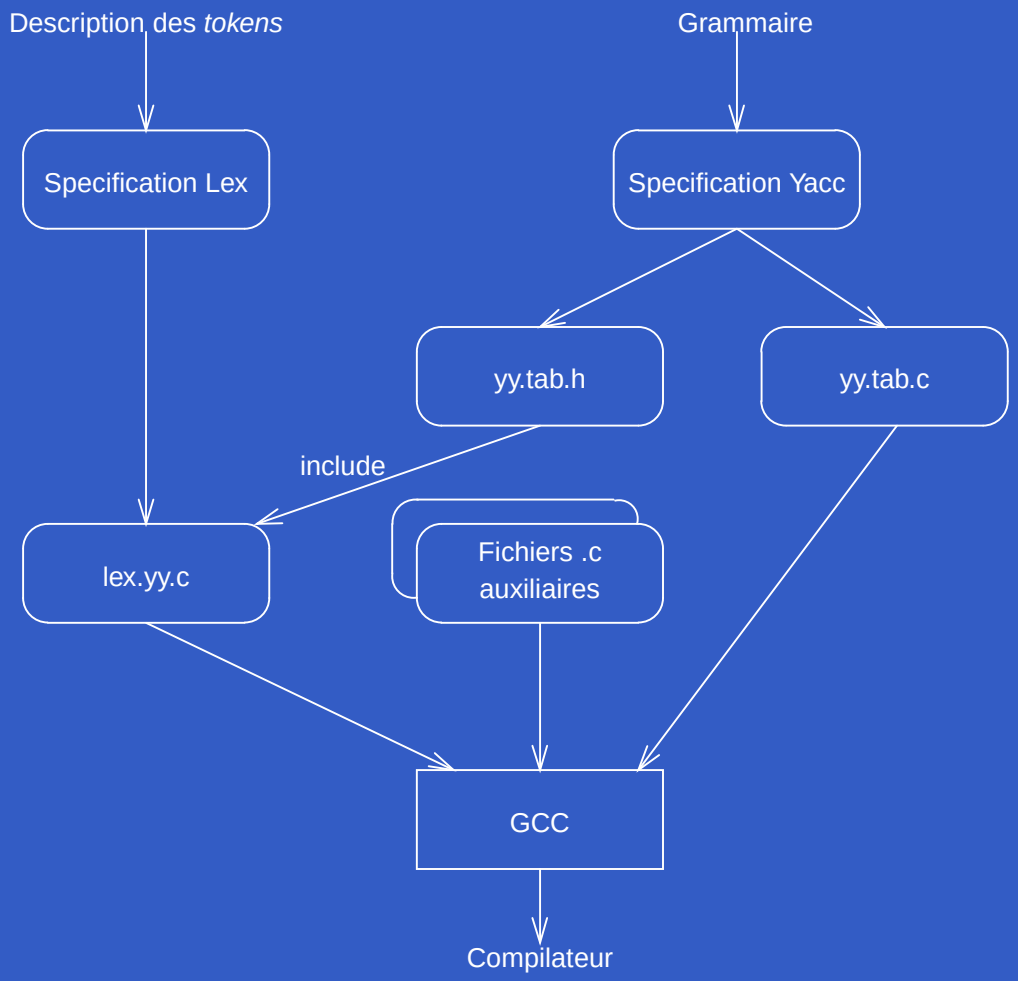
Introduction – 2

- L'*input* de LEX est une spécification, formée de paires d'expressions régulières et de code C ;
- LEX s'en sert pour générer un fichier source C, implémentant le *scanner* sour la forme d'une fonction : `yylex()` ;

Introduction – 2

- L'*input* de LEX est une spécification, formée de paires d'expressions régulières et de code C ;
- LEX s'en sert pour générer un fichier source C, implémentant le *scanner* sour la forme d'une fonction : `yylex()` ;
- L'exécutable, obtenu après compilation, analyse son *input* pour trouver des occurences des expressions régulières de la spécification, et exécute le code C associé.

Interaction LEX-YACC



Format d'une spécification LEX

- Format en trois parties, séparées par des %% :
  - Partie 1 : Déclaration de variables, de constantes et de définitions régulières ;
    - Les définitions régulières sont utilisées comme des « macros » dans les actions ;
    - Par exemple : `chiffre [0-9]`

Format d'une spécification LEX

- Format en trois parties, séparées par des %% :
  - Partie 2 : Règles de traduction, de la forme :

    `ExpReg  {Action}`
    - `ExpReg` est une expression régulière étendue ;
    - `Action` est un fragment de code C, qui sera exécuté chaque fois qu'un *token* satisfaisant `ExpReg` est rencontré.
    - Les actions peuvent faire appel aux expressions régulières de la partie 1 grâce aux `{}` ;
    - Par exemple : `{chiffre} {Action...};`

Format d'une spécification LEX

- Format en trois parties, séparées par des %% :
  - Partie 3 : Procédures de l'utilisateur.
    - Par exemple : `main()` si le *scanner* n'est pas utilisé avec YACC ou BISON.

Variables spéciales accessibles

- Dans les actions, on peut accéder à certaines variables spéciales :
  - `yyleng` : contient la taille du *token* reconnu ;
  - `yytext` : est une variable de type `char*` qui pointe vers la chaîne de caractères reconnue par l'expression régulière.
  - `yylval` : qui permet de passer des valeurs entières à YACC...
- Il existe aussi une action spéciale : `ECHO` qui équivaut à `printf("%s",yytext)`.

Exemple

```
chiffre [0-9]
lettre [a-z]|[A-Z]
%%
{lettre}({chiffre}|{lettre})*   {
        printf("J'ai reconnu l'identifiant:
        %s de longueur %d\n", yytext, yyleng) ;
        }

({chiffre})+    {
        printf("J'ai reconnu un nombre\n") ; ECHO ;
        }
%%
main(){ yylex() ; }
```

Compilation

- Pour obtenir l'exécutable du *scanner* :

Compilation

- Pour obtenir l'exécutable du *scanner* :
  - On commence par compiler le fichier `lex.yy.c` :

```
gcc -c lex.yy.c
```

Compilation

- Pour obtenir l'exécutable du *scanner* :
  - On commence par compiler le fichier `lex.yy.c` :

    ```
    gcc -c lex.yy.c
    ```

  - ... ce qui crée le fichier `lex.yy.o`;

Compilation

- Pour obtenir l'exécutable du *scanner* :
  - On commence par compiler le fichier `lex.yy.c` :

    ```
    gcc -c lex.yy.c
    ```

  - ... ce qui crée le fichier `lex.yy.o` ;
  - On compile les autres fichiers `.c` ;

Compilation

- Pour obtenir l'exécutable du *scanner* :
  - On commence par compiler le fichier `lex.yy.c` :

```
gcc -c lex.yy.c
```

  - ... ce qui crée le fichier `lex.yy.o` ;
  - On compile les autres fichiers `.c` ;
  - On *linke* le tout avec la librairie `libl` (`libfl` pour FLEX) :

```
gcc -o executable fichier1.o ...
    fichierN.o lex.yy.o -ll
```

Exercices

1. Écrire un filtre qui compte le nombre de caractères, de mots et de lignes d'un fichier ;
2. Écrire un filtre qui numérote les lignes d'un fichier (hormis les lignes blanches) ;
3. Écrire un filtre qui n'imprime que les commentaires d'un programmes. Ceux-ci sont compris entre `{ }` ;
4. Écrire un filtre qui transforme un texte en remplaçant le mot `compilateur` par `beurk` si la ligne début par `a`, par `schtroumpf` si la ligne débute par `b` et par `youpi !!` si la ligne débute par `c`.

Exercice 5

Écrire une fonction d'analyse lexicale à l'aide de LEX. Les *tokens* reconnus seront :

- Les nombres décimaux (en notation scientifique) ;
- Les identifiants de variables ;
- Les opérateurs relationnels (<, >, $\leq$, *etc*) ;
- Les mots-clefs `si`, `sinon` et `alors`.

Cette fonction a pour but d'être utilisée dans un analyseur syntaxique écrit en YACC.

Exercice 1 – Solution

```
chiffre [0-9]
lettre  [a-z]|[A-Z]
alphanum {chiffre}|{lettre}
blanc " "
%{
int Nb_Car, Nb_Word, Nb_Line;
%}
%%
({alphanum})+      {ECHO; Nb_Car += yyleng; Nb_Word++;}
({blanc})+         {ECHO; Nb_Car += yyleng;}
\n {ECHO; Nb_Line++;}
%%
```

Exercice 1 – Solution

```
#include <stdio.h>
int main()
{
  Nb_Car=Nb_Word=Nb_Line=0;
  yylex() ; /* pas de return */
  printf("\nNb_Car=%d\nNb_Word=%d\nNb_Line=%d\n\n",
                          Nb_Car,Nb_Word,Nb_Line);
  return(1);
}
```

Exercice 2 – Solution

```
blanc " "|\t
%{
#define yywrap() 1 /* GNU flex */
int Nb_Line = 1;
%}
%%
^({blanc})*     {}
\n              {}
^.*             {printf("[%d] %s\n",Nb_Line, yytext) ; N
%%
#include <stdio.h>
int main()
{ yylex(); return(1); }
```

Exercice 3 – Solution

```
%%
"{"([^{}])+"}" {ECHO;}
.|\n {}
%%
#include <stdio.h>
int main()
{
  yylex();
  return(1);
}
```

Exercice 4 – Solution

```
%{
int i;
%}
%%
\n {ECHO; i=0;}
^a {ECHO; i=1;}
^b {ECHO; i=2;}
^c {ECHO; i=3;}
compilateur {if(i==1) printf("beurk");
        if(i==2) printf("schtroumpf");
        if (i==3) printf("youpi!!");
        if(i==0) printf("compilateur");}
.  {ECHO;}
```

Exercice 4 – Solution

```
%%
#include <stdio.h>
int main()
{
  i=0;
  yylex();
  return(1);
}
```

Exercice 5 – Solution

```
%{
#define PPQ 1
#define PPE 2
/* etc... */
%}
/*Definition regulieres*/
delim   [ \t\n]
bl      {delim}+
lettre  [A-Za-z]
chiffre [0-9]
id      {lettre}+({lettre}|{chiffre})*
nombre  {chiffre}+(\.{chiffre}+)?(E[+-]?{chiffre}+)?
```

Exercice 5 – Solution

```
%%
{bl}     {/* Pas d'action, pas de retour */}
si       {return(SI);}
alors    {return(ALORS);}
sinon    {return(SINON);}
{id}     {yylval=RangerId(); return(ID);}
{nombre}{yylval=RangerNB(); return(NOMBRE);}
"<"      {yylval=PPQ; return(OPREL);}
"<="     {yylval=PPE; return(OPREL);}
"="      {yylval=EGA; return(OPREL);}
"<>"     {yylval=DIF; return(OPREL);}
```

Exercice 5 – Solution

```
">"      {yylval=PGQ; return(OPREL);}
">="     {yylval=PGE; return(OPREL);}
%%
RangerId() {
  /*
Ranger dans la table des symboles le lexeme yytext
  */
}
RangerNb() {
  /*Idem pour un nombre (Conversion)*/
}
```