# Compilation — Mémo (f)lex

**Alain KETTERLIN (alain@unistra.fr)**

## Présentation de lex

Lex est un générateur d'analyseur lexical : il prend en entrée une description lexicale (en résumé, un ensemble de couples expression-régulière/action) et produit une fonction d'analyse lexicale. Cette fonction est écrite en C et s'appelle `yylex()`. Elle peut ensuite être intégrée à n'importe quel programme.

Le manuel de l'utilisateur se trouve à flex.sourceforge.net/manual/.

Une description Lex a la forme suivante :

```
%{
...  /* Déclarations C */
%}
...  /* Déclarations Lex */
%%
...  /* Règles lexicales */
%%
...  /* Code C additionnel */
```

Les deux sections *Déclarations C* et *Code C additionnel* seront recopiées textuellement dans le fichier résultat. Les deux autres sections (*Déclarations Lex* et *Règles lexicales*) permettent de détailler le fonctionnement de l'analyseur lexical.

### Règles lexicales

La troisième partie est la plus importante : elle contient la définition de l'analyseur lexical sous la forme d'un ensemble de couples contenant chacun une expression régulière et une action.

```
exp-reg-1    action-1
exp-reg-2    action-2
...
```

Pour les expressions régulières, les constructions possibles sont résumées dans la table **??**. Il ne doit pas y avoir de blanc avant l'expression régulière. Une action commence sur la même ligne que l'expression régulière à laquelle elle est associée, et se poursuit jusqu'à la fin de la ligne, ou jusqu'à l'accolade fermante correspondante si l'action commence par une accolade ouvrante. Le code C constituant l'action est reproduit textuellement par Lex dans le code C résultant, y compris s'il contient des erreurs (de syntaxe ou autres).

Lex est gourmand (*greedy*) : il reconnaîtra le plus long texte possible. Si le texte reconnu correspond à plusieurs expressions, Lex choisit celle qui apparaît en premier dans la description. Si le texte d'entrée ne correspond à aucune expression, ou si l'action associée est omise, Lex affiche le texte lu sur la sortie standard.

À partir d'une telle description, Lex produit un fichier C (appelé par défaut `lex.yy.c`), contenant une fonction `yylex()` qui ressemble à :

```
int yylex()
{
  while ( non fin de fichier )
  {
    trouver le plus long préfixe correspondant
      à une des expressions régulières
    si le lexème correspond à exp-reg-1 alors
      action-1
    sinon si le lexème correspond à exp-reg-2 alors
      action-2
    sinon ...
      ...
    sinon /* action par défaut (optionnelle) */
      afficher le premier caractère sur stdout
    éliminer les caractères utilisés de l'entrée
  }
  return 0;
}
```

Remarquez que l'action associée à un type de lexèmes peut contenir une instruction `return` (ce qui termine l'appel courant de `yylex()`). Si ce n'est pas le cas, `yylex()` continue à s'exécuter en recherchant le lexème suivant. Par convention, `yylex()` renvoie 0 lorsqu'elle atteint la fin du fichier, mais c'est au reste du programme de déterminer combien de fois elle doit être appelée.

Notez aussi que par défaut, Lex lit son entrée standard, et les actions peuvent écrire sur la sortie standard (par exemple en phase de mise au point). En fait, la fonction `yylex()` lit un fichier stocké dans la variable globale `yyin` (de type `FILE*`). Il est donc possible d'initialiser cette variable avant le premier appel à `yylex()`. Pour changer de fichier d'entrée en cours d'analyse, cherchez dans la documentation les règles d'usage de `yy_create_buffer(FILE*,int)`, `yy_switch_to_buffer(...)` et `yy_delete_buffer(...)`. Notez que la déclaration de ces trois fonctions est insérée par Lex lui-même : elle ne

| **Un seul caractère** | | |
|---|---|---|
| $c$ | le caractère $c$ | |
| `.` | n'importe quel caractère | (sauf fin de ligne) |
| `[`$c_1c_2\ldots c_k$`]` | $c_1$ ou $c_2$ … ou $c_k$ | $\equiv (c_1\|c_2\|...\|c_k)$ |
| `[`$c_1$`-`$c_2$`]` | un caractère entre $c_1$ et $c_2$ | |
| `[^`$c_1c_2\ldots c_k$`]` | ni $c_1$ ni $c_2$ … ni $c_k$ | |
| `[^`$c_1$`-`$c_2$`]` | un caractère qui n'est pas entre $c_1$ et $c_2$ | |
| `[[:`$X$`:]]` | avec $X$ dans `alpha`, `lower`, `upper`, `blank`, `space`, `digit`, `alnum`, `punct`, `graph`, `print`, `xdigit`, `cntrl`. | `man isalpha` etc. |
| `\0` | le caractère de code 0 | |
| `\`$c$ | si $c$ est dans `nrtfvab` alors le caractère correspondant en C, sinon le caractère $c$ | `\*` représente le caractère `*` |
| `\`$o_1o_2o_3$ | code *octal* $o_1o_2o_3$ | |
| `\x`$x_1x_2$ | code *hexadécimal* $x_1x_2$ | |
| `<<EOF>>` | la fin du fichier | |
| **Une chaîne de caractères** | | |
| $\alpha\beta$ | concaténation | |
| `(`$\alpha\beta$`)` | groupement | |
| $\alpha \mid \beta$ | alternative | |
| $\alpha*$ | répétition (quelconque) | |
| $\alpha+$ | répétition (non vide) | $\equiv \alpha\alpha*$ |
| $\alpha?$ | option | $\equiv \alpha \mid \varepsilon$ |
| $\alpha\{n,m\}$ | répétition (entre $n$ et $m$ fois) | |
| $\alpha\{n\}$ | répétition (exactement $n$ fois) | $\equiv \alpha\{n,n\}$ |
| $\alpha\{m,\}$ | répétition (au moins $m$ fois) | $\equiv \alpha\{m\}\alpha*$ |
| `^`$\alpha$ | $\alpha$ au début d'une ligne | |
| $\alpha$`$` | $\alpha$ à la fin d'une ligne | |
| `"`$c_1c_2\ldots c_k$`"` | la chaîne $c_1c_2\ldots c_k$ littérale | `"a.b*"` $\equiv$ `a\.b\*` |
| `{`$\lambda$`}` | l'expression correspondant à la définition $\lambda$ | |

TABLE 1 – Syntaxe des expressions régulières Lex

peuvent donc être utilisée que dans le fichier Lex.

### Déclarations Lex

La section des *Déclarations Lex* contient deux types d'informations distincts, des options et des définitions de noms.

Les options modifient le code produit. Il est rare d'avoir à en connaître le détail (ne serait-ce que pour des raisons de portabilité). Deux d'entre elles sont cependant utiles pour des raisons pratiques avec la version GNU de Lex (appelée `flex`). Elle s'écrivent :

```
%option nounput
%option noyywrap
```

Leur signification tient au fonctionnement interne de `flex` (voir la documentation). Sans ces options, les programmes contenant une fonction `yylex()` produite par `flex` doivent être liés à `libfl.a` (via l'option `-lfl`).

Les définitions de noms quant à elles permettent de donner un nom à certaines parties d'expressions régulières. La syntaxe est immédiate. Voici un exemple :

```
DIGIT    [0-9]
%%
{DIGIT}+              return ENTIER;
{DIGIT}+"."{DIGIT}*   return FLOTTANT;
```

Notez que le nom défini doit être utilisé entouré d'accolades.

### Accès au texte reconnu

Lorsque Lex déclenche une règle, cela signifie qu'il a trouvé une chaîne de caractères qui correspond à une des expressions régulières. Il exécute alors l'action associée. Pendant l'exécution de ce code (et seulement à ce moment là), le texte reconnu est placé dans la variable `yytext` (de type `const char *`). Le nombre de caractères de la chaîne est placé dans la variable `yyleng`, de type `int`.

### Invocation

Si votre fichier s'appelle `lexeur.lex`, et que vous lancez

```
flex lexeur.lex
```

vous obtenez un fichier `lex.yy.c` contenant la fonction `yylex()`. Je vous recommande un appel de la forme :

```
flex -olexeur.c -s -p -p -v lexeur.lex
```

(Voir la documentation pour la signification des options.)