# Introduction à GNU Flex

Jérémy Cochoy

2 juillet 2010

**Résumé**

Cet article est une introduction au puissant Analyseur Lexical *Flex*. Y sont décrits les rudiments nécessaires à l'utilisation de cet outil, ainsi que quelques fonctionnalités avancées.

# Table des matières

# Première partie
# Découverte de Flex

## 1 Introduction

**Flex** est un générateur de *lexer*, terme anglais qui désigne un *Analyseur Lexical*.

Un Analyseur Lexical est un "programme"[1] qui analyse un flot de caractères et le segmente en lexèmes (en anglais on parle de "token" qui signifie jeton).

Comme exemple de lexème, on peut prendre les "mots" d'un langage de programmation. Ainsi, figurent dans les lexèmes : les nombres, les identifiants de variables et de fonctions, les mots clés, les opérateurs, etc...

On utilisera souvent un analyseur lexical avec un *Analyseur Syntaxique*. En effet, on peut considérer que :

- l'ensemble des caractères ASCII forme un alphabet, et les lexèmes sont des mots d'un langage L.
- les lexèmes obtenus par l'analyseur lexical peuvent être considérés comme les lettres d'un alphabet et forment un langage L' (sous-langage de L).

On retrouve typiquement ce type de considération dans la conception d'un compilateur, bien que Flex ne se limite pas qu'à ce seul domaine.

Dans cet article, on introduira les notions de base nécessaires à la manipulation d'un lexer du type 'lex', puis nous approfondirons certaines fonctionnalités de Flex.

## 2 Format d'un document .lex

### 2.1 Les sections

Un document .lex est divisé en trois sections :

1. Les définitions
2. Les règles
3. Le code additionel

Chacune de ces trois parties est séparée par un symbole %%. Seules les règles sont obligatoires, et on peut donc omettre le deuxième séparateur.

Ainsi le listing 1 est le contenu d'un document minimal compilable par Flex, qui ne fait rien.

Listing 1 – Document minimal compilable

```
1  %%
```

1. Pour être exact, ce n'est pas nécessairement un programme. Flex crée un ensemble de fonctions qui peuvent être réutilisé, par exemple par un *analyseur syntaxique* comme *Bison*. On devrait plutôt considérer un analyseur lexical comme une boite noire qui prend en entrée un flux de caractères et a pour sortie un flux de lexèmes.

## 2.2 Les définitions

Cette première section permet de définir des alias d'une expression régulière sous la forme *<name> <definition>*.

*<name>* doit être un identificateur commençant par une lettre, et ne comportant que des caractères alphanumériques, des underscores(_) et des tirets(-).

*definition* doit être une expression régulière valide (cf section 3).
Une définition peut faire référence à un nom précédemment défini.
Ainsi, la chaine *{name}* sera évaluée comme *(definition)*. (cf exemple 2)
Un exemple est donné dans la figure 2.

C'est aussi dans cette section que vous pouvez ajouter quelques lignes de C qui apparaitront avant l'implémentation des méthodes du lexer.
Ceci peut être extrêmement utile dans le cas où vous souhaitez inclure un header qui définit des constantes numériques (par exemple via un enum) correspondant aux différents lexèmes que vous souhaitez identifier.
Nous en reparlerons dans la section 4.2.

Notez que vous pouvez, via la balise *%top{}*, imposer à Flex de placer votre code avant la plupart des déclarations et définitions de Flex.

Listing 2 – Format et exemple de la section des définitions

```
%top{
        /* Ce code se retrouvera au début du fichier lex.yy.c */
}
%top{
        /* Celui-ci aussi */
}
%{
        /* Ce code se trouvera avant l'implémentation
        des méthodes, parfait pour définir quelques constantes */
        enum {
          NUMBER        = 256,
          SUM           = 257,
          DIF           = 258
        };
%}

/* Définit un chiffre */
chiffre [0-9]
/* Définit un entier */
entier  {chiffre}+
/* Définit un nombre réel */
reel    {chiffre}+"."{chiffre}*
```

### 2.3 Les règles

Cette section est capitale, car c'est ici que sont définis les lexèmes, à partir d'une expression régulière.

Chaque ligne se présente sous la forme <*pattern*> <*action*> où <*pattern*> est une expression régulière qui décrit un/des lexème(s) et <*action*> est le code C qui sera executé. On peut aussi y placer quelques macros définies par Flex, comme ECHO; qui affiche le lexème.

Bien que deux sections soient respectivement consacrées aux expressions régulières (section 3) et aux actions (section 4.2), la figure 3 représente un lexer qui extrait les nombres d'un texte.

### 2.4 Code additionel

Cette section est disponible pour que vous y placiez le code que vous souhaitez. Il y sera recopié tel quel à la fin du fichier *lex.yy.c*. La figure 3 en est un parfait exemple.

Listing 3 – Extrait les entiers et réels d'un fichier texte

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

chiffre [0-9]
entier  {chiffre}+
reel    {chiffre}+"."{chiffre}*
%%

                /* Les entiers et réels sont affichés */

{entier}        ECHO; printf("\n");
{reel}          ECHO; printf("\n");

                /* Les autres caractères, ignorés. */
[[:alnum:]]+    ;
[[:blank:]]+    ;
[\n]            ;

%%
/* À compiler avec "lex extract.lex; gcc lex.yy.c -lfl" */
int main(void)
{
  yylex();
  return 0;
}
```

## 3 Les expressions régulières

Pour définir le <*pattern*> de chaque lexème, Flex utilise une syntaxe d'expressions régulières ainsi que quelques classes de caractères prédéfinies.

## 3.1 Les règles élémentaires

Voici une liste des règles élémentaires des expressions régulières utilisées par Flex.
Les expressions sont encadrées de "'".

Flex cherchera à faire correspondre une certaine partie du texte source à une des règles. Si plusieurs règles correspondent, Flex "choisira" la plus longue, en nombre de caractères.

On nommera "classe" toute expression insérée entre crochets `[]`.

**Caractère simple**
`'x'`
Fait correspondre le caractère `x`.

**Tout caractère**
`'.'`
Fait correspondre n'importe quel caractère, excepté `\n`.

**Ensemble**
`'[xyz]'`
Fait correspondre à l'un des caractères `x`, `y` ou `z`.

**Intervalle**
`'[ai-kz]'`
Fait correspondre l'un des caractères `a`, `i`, `j`, `k` ou `z`.
Notez que les caractères spéciaux — voir ci-dessous — sont considérés comme de simples caractères, à l'exception de `^` en premier caractère, `\`, `]]` et `[[`, et enfin `-` entre plusieurs caractères.

**Négation**
`'[^A-Zc]'`
Fait correspondre tout caractère *qui n'est pas* dans la classe.
Dans ce cas :

1. Compris entre `A` et `Z` inclus
2. Le caractère `c`

**Etoile**
`'r*'`
De 0 à inf r. (où r est une expression).

**Plus**
`'r+'`
Une instance de r, ou bien plusieurs.

**Option** `'r?'`
L'expression r peut être ou ne pas être présente.

**Dénombrement**

1. `'r{2, 5}'` : 2 à 5 instances
2. `'r{2,}'` : 2 à inf instances
3. `'r{4}'` : exactement 4 instances

**Expension**

`'{name}'`

Se verra substituer l'expression nommée `name` dans la section des définitions.

**Chaine de caractères**

`'"[xyz]\"foo"'`

Fait correspondre la chaine de caractères `[xyz]\"foo`.

**La protection**

`'\X'`

1. `'\x'` correspond au caractère ASCII spécial `\x` où — en l'absence d'équivalence ASCII — au caractère `x`.
2. `'\0'` correspond au caractère ASCII NULL(0)
3. `'\123'` fait correspondre le caractère ASCII ayant pour code **octal** `123`.
4. `'\2a'` fait correspondre le caractère ASCII ayant pour code **hexadécimal** `2a`.

**Parenthèses**

`'(r)'`

Force l'évaluation de l'expression r avant les opérations de plus faibles précédences.

**Concaténation**

`'rs'`

Fait correspondre les deux expressions r et s, l'une après l'autre.

**Disjonction**

`'r|s'`

Fait correspondre soit r, soit s.

**Début de ligne**

`'^r'`

Fait correspondre r seulement si le caractère précédent était un `\n` ou bien qu'il s'agit de la première ligne.

**Fin de ligne**

`'r$'`

Fait correspondre r seulement si le caractère suivant est un `\n`.

**Fin de fichier**

`'<<EOF>>'`

Cette expression — qui contient uniquement la chaine `<<EOF>>` — correspond à la fin du flux de lecture. L'action associée sera donc exécutée, par exemple, à la fin de la lecture du fichier [2].

## 3.2 Non-interprétation des caractères spéciaux

Comme vous pouvez le constater, de nombreux caractères sont interprétés par Flex.

Ceci peut poser un problème si vous désirez exprimer une chaine qui utilise ces caractères, par exemple `[[:alnum:]]+.*`.

Pour pouvoir facilement préciser que ces caractères ne doivent pas être interprétés par Flex, il vous suffit de les encadrer par des guillemets, ce qui donnera pour l'exemple que nous avons cité : `"[[:alnum:]]+.*`

## 3.3 Précédence des opérateurs

Les expressions de cette liste sont classées par ordre de précédence décroissante [3].

Pour rappel, la précédence correspond à l'ordre d'évaluation des éléments considérés. Ainsi, dans l'expression arithmétique *3*2+7*3*, l'opérateur * sera d'abord évalué car de plus forte précédence. Par la suite, l'opérateur + sera évalué.

Ce seront donc les premiers symboles de l'expression qui seront interprétés en premier. Afin de forcer l'application d'un opérateur à une expression complexe, vous pouvez donc utiliser les parenthèses.

Par exemple, l'expression `foo|bar*` est équivalente à `(foo)|(ba(r*))`.

## 3.4 Alias de classes

Pour faciliter l'écriture d'ensembles de caractères couramment utilisés, Flex dispose des alias suivants :

| | | |
|---|---|---|
| `[:alnum:]` | `[:alpha:]` | `[:blank:]` |
| `[:cntrl:]` | `[:digit:]` | `[:graph:]` |
| `[:lower:]` | `[:print:]` | `[:punct:]` |
| `[:space:]` | `[:upper:]` | `[:xdigit:]` |

Ces alias peuvent être utilisés dans les opérateurs d'ensemble [4]. On aura par exemple `[[:digit:]x]` équivalent à `[0-9x]`.

Chacune de ces classes de la forme `[:XX:]` correspond à l'ensemble des caractères reconnus par la fonction C `isXX`.

Par exemple : `[[:alnum:][:digit:]]` équivaut à `[a-zA-Z0-9]`.

2. Dans le cas d'une entrée interactive, ce sera CTRL+D qui permettra d'indiquer la fin du flux.
3. Pour une description plus précise de la précédence des opérateurs, reportez-vous au manuel de Flex[4].
4. On peut aussi nier ces classes, de la manière suivante : *[:^blank:]*.

### 3.5 Modificateurs d'interprétation

Il est possible de modifier, localement à une expression, la façon dont est interprétée une expression.

La syntaxe est de la forme : `(?r-s:pattern)` où *pattern* est l'expression considérée, $r$ une option à appliquer et $s$ une option à ne pas appliquer.

Les options sont :

1. $i$ signifie case-insensitive (insensible à la casse)
2. $-i$ signifie donc case-sensitive (sensible à la casse)
3. $s$ altère le sens de `'.'` et permet de matcher tout caractère, y compris `\n`
4. $-s$ n'altère pas l'interprétation de `'.'`
5. $x$ ignore les commentaires C et les espaces.

Voici quelques expressions et leurs équivalences "classiques".

| | |
|---|---|
| `(?:foo)` | `(foo)` |
| `(?i:foo)` | `([Ff][Oo][Oo])` |
| `(?-i:fOo)` | `(fOo)` |
| `(?ixs: a . b)` | `([Aa][\x00-\xFF][Bb])` |
| `(?x: a /* comment */ b)` | `(ab)` |

De façon générale, je vous déconseille fortement d'utiliser ces options si elles ne vous apportent pas un meilleur confort d'écriture/lecture, ou rendent vos expressions obscures.
Ne les employez que pour alléger la lecture et n'hésitez pas à rappeler par un bref commentaire leurs significations.

Les cas qui reviendront le plus probablement dans leurs utilisations sont une insensibilité à la casse locale à une expression.
Par exemple : `i(?i:dentifian)t`.

### 3.6 Union et intersection de classes

Il est possible d'effectuer des intersections ou encore des unions de classes. Les opérateurs d'intersection et d'union sont respectivement `{-}` et `{+}`.
Leur précédence est la même que celle d'une classe et ils sont tous deux associatifs à gauche.

Ils peuvent être utiles pour composer simplement de nouvelles classes.
Par exemple : `[[:alnum:]]{-}[aAxXvV]{+}[%]`

## 4 Les actions

Les actions sont le coeur du lexer. Nous les avons déjà entrevues avec un exemple(figure 3).
Cette section se consacrera donc à la compréhension de comment ces actions nous permettent de segmenter le flux de caractères en lexèmes.

### 4.1 La correspondance

Avant d'approfondir les actions, il est nécessaire d'établir quelques points sur la façon dont le lexer généré fera correspondre les caractères aux regex.

### 4.1.1 Le flux d'entré

Flex lit les caractères un à un[5] et cherche à les faire correspondre à l'une des regex, jusqu'à ce qu'un caractère ne corresponde plus à aucune des regexp, ou bien qu'il atteigne la fin du flux.

Flex choisira donc l'expression qui fait correspondre le plus grand nombre de caractères. S'il y en a plusieurs, il choisira la **première** à apparaitre dans le fichier lex[6].

Notez que si Lex ne parvient pas à faire correspondre un caractère seul avec aucune de vos règles, il lui appliquera la *règle par défaut* qui consiste à afficher le caractère à l'écran.

### 4.1.2 L'accès aux caractères d'un lexème

Flex stocke chacun des caractères analysés les uns à la suite des autres dans une chaine de caractères qu'il rend accessible dans chaque action.

Le texte formant le lexème est pointé par la globale *yytext* et la longueur de la chaine est contenue dans la globale *yyleng*. Par défaut, `yytext` est une chaine de caractères allouée dynamiquement par le lexer.

En utilisant ce mécanisme vous êtes à l'abri d'un débordement mémoire si vous faite correspondre des chaines extrêmement longues[7].

Notez que l'utilisation de ce mode provoque l'altération de la chaine contenue dans `yytext` à l'utilisation de la directive `unput` (cf section 4.4.3).

Une autre possibilité, qui correspond à la version AT&T de Lex, est l'utilisation d'un tableau global de taille `YYLMAX`.

La valeur de `YYLMAX` par défaut est normalement suffisante pour bien des utilisations[8].
Vous pouvez toutefois définir la valeur de votre choix en définissant tout simplement vous-même la constante `YYLMAX` dans la section des définitions.

Un dépassement de la taille du tableau entraine l'arrêt du parseur.

Notez que vous ne pouvez pas coupler l'option `-array` avec l'option `-c++` (cf section 8).

Pour expliciter l'utilisation d'un pointeur — qui est, l'option par défaut de Flex — ou bien d'un tableau, vous pouvez utiliser, au choix, les directives `%pointer` et `%array` dans la section de définition de votre fichier lex, ou encore les options `-array` et `-pointer` à la compilation.

5. Il est bon de préciser que Flex utilise son propre buffer pour la lecture des caractères. S'ils ne sont pas réellement lut un à un pour des raisons de performances, ils sont bien traités un à un.

6. Nous verrons à la section 4.3.4 que nous pouvons forcer la correspondance avec une chaine plus courte sous certaines conditions.

7. Attention, ceci n'est pas sans impact sur les performances de votre lexer.
À chaque redimensionnement de `yytext`, le lexer doit recommencer la lecture de la chaine depuis les premiers caractères.
Je vous recommande donc de rester raisonnable dans la longueur des chaines que vous cherchez à matcher. Considérez 4096 octets comme un maximum.

8. Dans la version 2.5.35, la taille par défaut est de 8192 caractères.

## 4.2 Description d'une action

Une action est un bloc de code C [9] qui sera inséré dans le code du parseur et sera ainsi exécuté lorsqu'une chaine correspond à une expression.

Ce système est très flexible puisque vous pouvez faire référence à des variables globales ou encore locales [10], interrompre l'exécution d'yylex (cf section 4.5), appeler les fonctions de votre choix et bien sûr exécuter tous les traitements qui vous semblent nécessaires.

Une action se place sur la même ligne que l'expression régulière qui la déclenche. Dans le cas d'actions complexes, vous pouvez les écrire sur plusieurs lignes en les encadrant d'accolades. En effet, Flex gère parfaitement l'imbrication des accolades du code C pouvant être contenu dans les actions.

L'absence d'une action sera interprétée comme une action "vide", c'est-à-dire que la chaine reconnue sera simplement ignorée.

## 4.3 Macro-Action de flex

Flex fournit un jeu de macros qui s'intègre parfaitement au code C pour vous permettre de contrôler le comportement de votre lexer. Les voici.

### 4.3.1 Pipe : See Below

Afin d'alléger l'écriture et d'éviter la répétition d'une même action pour plusieurs règles successives, vous disposez de l'action /.

Le caractère / (aussi nommé "pipe") sera interprété comme *effectuer l'action identique à celle de la règle suivante.*

De plus, vous pouvez chainer les renvois, comme dans l'exemple 5.

### 4.3.2 ECHO

Produit la sortie de la chaine de caractères correspondant au lexème à l'origine de cette action. Cette macro "affiche" [11] donc le contenu de `yytext`.

Cette macro peut être suivie de code C qui sera exécuté. Le listing 4 duplique chacun des caractères non-blancs reçus en entrée.

Listing 4 – Duplique les caractères non-blancs

```
%%
[[:blank:]]     ECHO;
.               ECHO; ECHO; /* duplication */
```

9. Vous pouvez placer du code C++ et compiler le fichier obtenu avec G++.
Pour en savoir plus, référez-vous à la section C++ (cf section 8).

10. Vous pouvez déclarer une variable locale (cf section 13.2) en insérant le code C de sa déclaration après une tabulation.

11. Si l'entrée et la sortie sont, par défaut, l'entrée et la sortie standard, vous pouvez tout à fait les redéfinir pour par exemple lire et écrire directement depuis/dans des fichiers.
On préfèrera donc employer les termes "sortie" et "entrée".

```
4 \n                ECHO;
  %%
  int main(void)
  {
    yylex();
9   return 0;
  }
```

#### 4.3.3 BEGIN

Suivi d'une *condition de départ* (cf section 6), provoque le changement de l'état du lexer à la prochaine lecture.

#### 4.3.4 REJECT

L'instruction `REJECT` provoque le "rejet" d'une expression. C'est-à-dire que Flex considèrera avoir échoué lors de la tentative de correspondance du flux avec l'expression qui a déclenché ce rejet.

L'action `REJECT` vous permet donc de faire adopter à Flex le comportement qu'il aurait eu si la correspondance avec cette expression n'avait pas été possible tout en vous laissant la possibilité d'exécuter des instructions C.

Notez que `REJECT` est un *branchement*, c'est-à-dire qu'au même titre qu'un *break*, il termine l'exécution de l'action en cours et rend la main au lexer.

Observez la figure 5 et étudions le comportement avec la chaine *chapichapo*.
Le lexer commencera par faire correspondre *chapichapo*, l'enverra à la sortie, puis *chapicha*, *chapi*, et enfin *cha*.
La chaine obtenue sera donc *chapichapochapichachapicha*.

Attention, l'utilisation de `REJECT` n'est pas sans conséquences sur les performances de votre lexer. Il est recommandé de minimiser son utilisation si le temps fait partie de vos contraintes de développement.

Listing 5 – Exemple de rejet successif

```
  %%
  cha               |
  chapi             |
  chapicha          |
5 chapichapo        ECHO; REJECT;
  .|\n              /* Supprime tout autre caractère */
```

### 4.4 Influencer le flux

Flex vous fournit aussi quelques fonctions qui vous permettent d'intervenir sur le flux et d'y ajouter des caractères.

### 4.4.1 yymore

La fonction[12] `yymore` provoque l'ajout des caractères constitutifs du lexème actuellement reconnu en tant que préfixe du prochain lexème reconnu.

Par exemple, le texte de la figure 6, suite à la lecture de la chaine *super-panda* provoquera la sortie *super-super-panda*.

Notez que si vous utilisez `yymore`, vous ne devez pas modifier la valeur de yylen.
Il est bon aussi de savoir qu'`yymore` a un impact sur la vitesse de votre lexer, bien qu'il puisse être considéré comme négligeable.

Listing 6 – Illustration d'`yymore`

```
%%
super-   ECHO; yymore();
panda    ECHO;
```

### 4.4.2 yyless

La fonction `yyless`[13] est d'un certain point de vue "complémentaire" à `yymore`.

Elle permet de forcer la réexamination de tous les caractères situés après les n premiers de l'expression reconnue. Ces caractères seront donc placés à nouveau dans le flux de lecture de façon à être réexaminés dans le même ordre où ils se trouvaient à l'origine.

Suite à la lecture de la chaine *nutshell*, le lexer obtenu à partir de la figure 7 fera un unput des caractères `llehs` qui seront à nouveau identifiés par l'expression `shell`, ce qui engendrera la sortie *nutshell-shell*.

Listing 7 – Illustration d'`yyless`

```
%{
2 #include <stdio.h>
%}
%%
nutshell        ECHO; printf("-"); yyless(3);
shell           ECHO;
```

### 4.4.3 unput

La fonction `unput` permet d'ajouter des caractères en tête du flux de lecture.

Si vous souhaitez placer dans le buffer de lecture le mot *lin*, vous devez donc effectuer les opérations de la figure 8. Ainsi, la lecture de la chaine *michelin is michelin* aura pour sortie *lin is lin*.

Attention, l'utilisation d'unput provoque la destruction du contenu d'*yytext* si vous utilisez l'option `%pointer`(cf: section 4.1.2).

12. En réalité, pour un lexer C, `yymore` est défini par `#define yymore() ((yy_more_flag) = 1)`.
13. Tout comme `yymore`, `yyless` est aussi une macro.

Listing 8 – Unput d'un mot

```
%%
michelin        {
                unput('n');
                unput('i');
                unput('l');
                }
```

### 4.4.4 input

La fonction `input` permet de lire le prochain caractère appartenant au flux. Vous pourrez alors le traiter comme vous le désirez.

La figure 9 présente un moyen d'ignorer des commentaires C.

Listing 9 – Ignorer un commentaire avec input

```
%%
"/*"    {
        register int c;

        while (1)
        {
          /* Lecture des caractères jusqu'à une étoile */
          while ((c = input()) != '*' && c != EOF );

          /* Si l'étoile est suivie d'un slash */
          if (c == '*')
          {
            /* Lire toutes les étoiles */
            while ((c = input()) == '*');
            /* Si le caractère suivant est alors un slash,
               le commentaire se fini ici. */
            if ( c == '/' )
              break;
          }

          /* Fin inattendue */
          if ( c == EOF )
          {
            error( "EOF in comment" );
            break;
          }
        }
      }
```

### 4.4.5 yyterminate

La fonction[14] `yyterminate` permet de provoquer la fin d'exécution du lexer. Vous pouvez l'utiliser, par exemple, à la fin d'une action correspondant au symbole `<<EOF>>`.

## 4.5 Interruption d'`yylex()`

Jusqu'à présent, nous nous sommes contentés de laisser yylex s'exécuter et identifier tous les tokens les uns à la suite des autres sans interruption. Pourtant, il est tout à fait possible d'interrompre l'exécution d'`yylex()` pour rendre la main à la fonction appelante et obtenir par l'intermédiaire d'`yylex()` un flux de lexème.

Ce mécanisme est utilisé pour utiliser conjointement Flex et Bison (un puissant générateur d'analyseur syntaxique).

Pour ce faire, on déclarera les identifiants de chaque lexème comme un jeu de constantes dans un header, de préférence de valeur supérieure à 255, de cette façon un caractère simple constituant un lexème aura pour identifiant sa valeur ASCII.
Par la suite, chaque action consistera à retourner l'ID correspondant au lexème rencontré.

Pour un exemple, consultez l'extrait de code 10 qui illustre ces propos.

Listing 10 – Identifie les lexèmes pour une calculatrice élémentaire

```
%{
  /* Ce projet peut se compiler et s'exécuter avec les commandes :
    $>flex calcid.lex
    $>gcc lex.yy.c
    $>echo '(3*7.42) + 2^sin4' | ./a.out
  */

  /* Contenu du fichier id.h, correspondant à #include id.h */
  #define SIN 256
  #define COS 257
  #define EXP 258
  #define LN  259
  #define NUMBER  300
%}
/* Permet de se dispenser d'implémenter yywrap,
     ou de linker avec -lfl */
%option noyywrap

/* Définissons un nombre */
chiffre      [0-9]+
nombre       {chiffre}+("."{chiffre}+)?
%%

  /* Ignore les blancs */
[[:blank:]]|\n
```

14. La fonction `yyterminate()` n'est autre chose qu'une macro définie par `#define yyterminate() return YY_NULL`.

```
27    /* Identification des opérateurs */
[+\-*/^]  return (int)*yytext; /* Opérateurs +-/* */
[()]      return (int)*yytext; /* Parenthèses */

    /* Identification des nombres */
32  {nombre}  return NUMBER; /* Nombres réels */

    /* Identification des fonctions */
sin     return SIN;
cos     return COS;
37  exp     return EXP;
ln      return LN;

    /* Caractères inconnus (différent de \n) */
.     return (int)*yytext;
42
%%

int main(void)
{
47    int code;
    while((code = yylex()))
    {
      switch(code)
      {
52      case NUMBER:
          printf("Nombre : %s\n", yytext);
          break;
        case SIN:
        case COS:
57      case EXP:
        case LN:
          printf("Fonction : %s\n", yytext);
          break;
        case '+':
62      case '-':
        case '/':
        case '*':
        case '^':
        case '(':
67      case ')':
          printf("Opérateur : %c\n", code);
          break;
        default:
          printf("Inconnu : %c\n", code);
72        break;
      };
    }
```

```
  return (0);
}
```

## 5 Un exemple pour conclure

Pour conclure cette première partie, je vous propose un exemple de compilateur brainfuck pour les systèmes Unix disposant de la libc.

Listing 11 – Compilateur brainfuck

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

    /* Mode de compilation */
    #define BSD 1
    #define LINUX   0
    int mode = LINUX;

    /* Compteur de boucle */
    int loop = 0;

    /* Constantes */
    #define SHOW(x) printf("%s\n", (x))

    #define ST_ALIGN    "andl $-16, %esp"

    #define RIGHT       "subl $1, %esp"
    #define LEFT        "addl $1, %esp"
    #define PLUS        "incb (%esp)"
    #define MINUS       "decb (%esp)"

%}
%option noyywrap
%%
[>] SHOW(RIGHT);
[<] SHOW(LEFT);
[+] SHOW(PLUS);
[\-]    SHOW(MINUS);

\[  {
        loop++;
        printf(".L0%d\start:\n", loop);
        SHOW("cmpb  $0, (%esp)");
        printf("jz  .L0%d\end\n", loop);
    }

\]  {
        printf("jmp .L0%d\start\n", loop);
```

```
            printf(".L0%dend:\n", loop);
            loop--;
        }

[.,]    {
            /* On sauvegarde esp et on aligne la stack */
            SHOW("movl %esp, %ebp");
            SHOW(ST_ALIGN);
            /* Syscall write/read */
            if(*yytext == '.')
                SHOW("movl $4, %eax"); /* write */
            else
                SHOW("movl $3, %eax"); /* read */
            if (mode == BSD)
            {
                SHOW("pushl %ebx");
                SHOW("pushl %ebp");
                SHOW("pushl %edx");
                SHOW("pushl $0");
            }
            else
            {
                SHOW("movl $1, %ebx");
                SHOW("movl %ebp, %ecx");
                SHOW("movl $1, %edx");
            }
            SHOW("int $0x80");
            SHOW("movl %ebp, %esp");
        }

        /* Fin du fichier */
<<EOF>> {
                /* Toutes les boucles ne sont pas ouvertes/fermées! */
                if(loop)
                    fprintf(stderr,
                      "Wraning: All loop aren't closed!\n");
                yyterminate();
            }

.|\n        /* Ignore */

%%
int main(int argc, char *argv[])
{
    /* Paramètres */
    char* in = "in.bf";
    char* out = "out.s";
    int opt;
    while ((opt = getopt(argc, argv, "blho:")) != -1)
```

```
88      {
            switch(opt)
            {
                /* Sortie */
                case 'o':
93                  out = strdup(optarg);
                    break;
                /* BSD / LINUX */
                case 'b':
                    mode = BSD;
98                  break;
                case 'l':
                    mode = LINUX;
                    break;
                /* Utilisation */
103             case 'h':
                default:
                    fprintf(stderr,
                      "Usage: %s [-b|-l] -o destination source\n",
                        argv[0]);
108                 return EXIT_FAILURE;
            }
        }

        /* On charge le nom du fichier */
113     if (optind < argc)
            in = strdup(argv[optind]);
        if (optind + 1 < argc)
        {
            printf("Too many files!\n");
118         return EXIT_FAILURE;
        }

        /* On ouvre les IO */
        freopen(in, "r", stdin);
123     freopen(out, "w+", stdout);

        /***************/
        /* Compilation */
        /***************/
128
        /* Entry point */
        printf(".globl _start\n");
        printf("_start:\n");
        SHOW("addl $16, %esp");
133     SHOW(ST_ALIGN);
        while (yylex())
            ;
        /* EXIT(0) */
```

```
    printf("%s\n", "movl␣$1,␣%eax");
138 printf("%s\n", "movl␣$0,␣%ebx");
    printf("%s\n", "int␣$0x80");
    return 0;
}
```

# Deuxième partie
# Flex, utilisation avancée

## 6 Les conditions d'éxécution, ou "Start-Conditions"

Les *start-conditions* sont l'un des puissants mécanismes fournis par Lex dont vous pouvez user et abuser, sans craindre pour les performances de vos lexers.

Il s'agit de définir une série d'*états*, et d'associer à chacune de vos expressions un ou des *états* dans lesquels elles sont accessibles.
Ainsi, vous pouvez constater que pour une même expression vous pouvez adopter deux comportements totalement différents dans le cas de deux états différents.

Une expression qui n'est exécutée que dans les états *str* et *foo* sera de la forme du listing 12.

Listing 12 – start-conditions en action

```
/* ... */
%%

<str,foo>[^"]* { /* Consomme le contenu de la chaine */ }
```

Notez l'absence d'espace entre les états.

### 6.1 Déclarations

Les *start-conditions* sont déclarées dans la section des définitions.

Par défaut, on peut considérer qu'un seul état existe, l'état `<INITIAL>` qui correspond à l'état dans lequel se trouve votre lexer à la lecture du premier caractère.

Vous disposez des deux méthodes suivantes pour déclarer vos *start-conditions*.

**déclaration inclusive (`%s`) :** en utilisant ce type de déclaration, toutes vos expressions qui n'ont pas d'état défini explicitement seront exécutées — i.e. se verront affecter tous les états possibles — par votre lexer.
Vous ferrez donc ce choix si vous souhaitez restreindre certaines expressions à un état particulier.

**déclaration exclusive (`%x`) :** dans le cas de la déclaration exclusive, toute expression n'ayant pas ses états explicitement définis ne sera active que pour l'état `<INITIAL>`.
On appréciera ce mode dans le cas où votre lexer repose sur un enchainement d'états et que le comportement *normal* — comprenez par là, celui qui correspond à *aucun état particulier* — de votre lexer ne doit pas interférer avec un comportement spécifique.

Sachez que les start-conditions apparaissent dans le fichier *lex.yy.c* sous la forme de constantes et portent exactement le nom que vous leur définissez.
Elles n'ont donc pas d'espace de nom réservé et vous devez veiller à ce que leur nom n'entre pas en conflit avec vos propres macros, constantes ou fonctions.

## 6.2 Utilisation

Pour utiliser vos start-conditions, il vous suffit de faire précéder vos expressions régulières par les états associés comme dans le listing. 13.

Il est, bien entendu, possible de changer l'état de votre lexer. Pour ce faire,vous disposez de la macro `BEGIN()` qui prend en paramètre le nouvel état.

Voici un exemple permettant de consommer les commentaires d'un fichier de code C et d'y compter le nombre de retours à la ligne contenus en commentaire.

Listing 13 – Compte les lignes de commentaire d'un code C

```
%option noyywrap
%x cmt
%%
        int line = 0;

"/*"            BEGIN(cmt); /* Passe dans l'état "commentaire" */

<cmt>\n         {
                        /* Compte une ligne */
                        line++;
                }
<cmt>[^*\n]+    /* Consomme les caractères divers */
<cmt>"*"+       /* Consomme les '*' */
<cmt>"*"+"/"    BEGIN(INITIAL); /* Retourne à l'état initial */

<cmt><<EOF>>    {
                        fprintf(stderr, "Fin de fichier inattendu\n");
                        yyterminate();
                }
<<EOF>>         {
                        fprintf(stderr, "Nombre de \\n:%d\n", line);
                        yyterminate();
                }

        /* Les caractères hors des commentaires sont
           laissés tels quels */
%%
int main(void)
{
        yylex();
        return 0;
}
```

## 6.3 Manipulation

### 6.3.1 Start-conditions actuelles

Il est possible dans toute action de connaitre l'état actuel du lexer, cela grâce à la macro `YY_START`.

Ceci peut s'avérer extrêmement utile.

### 6.3.2 Stockage

Les start-conditions apparaissent dans le code source généré comme des constantes, plus exactement des entiers. Vous pouvez donc les stocker, les manipuler comme bon vous semble avant de les utiliser.

Le listing 14 vous présente une façon simple de mémoriser l'état dans lequel se trouvait le lexer avant de rentrer dans l'état de lecture de commentaire, puis le restore.

Listing 14 – Mémorisation d'un état avec une variable

```
%x cmt foo
%%
        comment_caller;

<INITIAL,foo>"/*"       {
                            comment_caller = YY_START;
                            BEGIN(cmt);
                        }

<cmt>\n             /* ... */
<cmt>[^*\n]+        /* ... */
<cmt>"*"+           /* ... */

<cmt>"*"+"/"        BEGIN(comment_caller);
```

### 6.3.3 Pile

Vous serez peut-être amené à mémoriser plusieurs états précédents, et cela vous amènera à souhaiter disposer d'un mécanisme de pile d'état afin de mémoriser l'état précédent à divers niveaux d'imbrication.

Flex vous fournit un mécanisme de pile dynamique — ainsi, vous n'avez pas de niveau maximum d'imbrication, à l'exception de la mémoire de votre système — ainsi qu'un jeu de trois fonctions pour y accéder.

Pour utiliser cette fonctionnalité, vous devez toutefois l'activer avec la directive `%option stack` ou l'option `-stack` à la compilation.

1. `void yy_push_state(int new_state)` cette fonction permet d'ajouter l'état actuel — qui, pour rappel, est accessible avec `YY_START` — au sommet de la pile, puis de passer à l'état `new_state` — ce qui peut se faire avec BEGIN(new_state).
2. `void yy_pop_state()` change l'état actuel en l'état stocké au sommet de la pile via `BEGIN` puis le retire de la pile.

3. `int yy_top_state()` permet de consulter l'état au sommet de la pile. Le contenu ne sera pas altéré.

## 6.4 Portée étendue

Il est possible de définir qu'un groupe d'expressions sera actif pour les mêmes start-conditions. Pour ce faire, il suffit d'encadrer les expressions par des accolades, après avoir définit les états qui leur sont associés, comme illustré dans le listing 15.

Listing 15 – Affectation d'une start-condition à un ensemble d'expressions

```
<ESC>    {
                "\\n"    return '\n';
                "\\r"    return '\r';
                "\\f"    return '\f';
                "\\0"    return '\0';
         }
```

## 6.5 Extraction de chaine C

Voici un exemple tiré du manuel de Flex qui permet d'extraire les chaines C contenues entre guillemets.

Leur taille est limitée à celle du buffer.

Listing 16 – Extraction d'une chaine de caractères C label

```
%{
#define ERRO(str)          {fprintf(stderr, "%s\n", str); exit(-1);}
#define CHK_SZ             {                                       \
                           if (string_buf_ptr - string_buf \
                               == MAX_STR_CONST)                   \
                           ERROR("Chaine trop longue")             \
                           }
#define MAX_STR_CONST      4096
%x str

%%
char string_buf[MAX_STR_CONST];
char *string_buf_ptr;

\"         {
           /* Début de la chaine */
           /* Initialisation du pointeur de chaine au début du buffer */
           string_buf_ptr = string_buf;
           BEGIN(str);
           }

<str>\" {
           /* Fin de la chaine */
           BEGIN(INITIAL);
```

```
        *string_buf_ptr = '\0';
        /* Renvoi du résultat à la fonction appelante */
        return string_buff;
        }

<str>\n {
        ERROR("Retour chariot inattendu\n");
        }

<str>\\[0-7]{1,3}       {
        /* Séquence d'échappement en octal */
        int result;

        sscanf(yytext + 1, "%o", &result);

        if (result > 0xff)
          ERROR("Valeur invalide");

        *string_buf_ptr++ = result;

        CHK_SZ;
        }

<str>\\[0-9]+   {
  /* Séquence invalide, par exemple '\48' ou '\0777777' */
                }

<str>\\n  *string_buf_ptr++ = '\n'; CHK_SZ;
<str>\\t  *string_buf_ptr++ = '\t'; CHK_SZ;
<str>\\r  *string_buf_ptr++ = '\r'; CHK_SZ;
<str>\\b  *string_buf_ptr++ = '\b'; CHK_SZ;
<str>\\f  *string_buf_ptr++ = '\f'; CHK_SZ;

<str>\\(.|\n)  *string_buf_ptr++ = yytext[1]; CHK_SZ;

<str>[^\\\n\"]+ {
        /* Suite de caractères simples, recopiée telle quelle */
        char *yptr = yytext;

        if (strlen(yytext) + string_buf_ptr + 1
            >= MAX_STR_CONST + string_buf)
          ERROR("Chaine trop longue")
        while (*yptr)
          *string_buf_ptr++ = *yptr++;
        }
```

# 7 Sources multiples

## 7.1 Chainage de flux d'entrée

Dans les exemples précédents, nous avons toujours linké (faire l'édition des liens) nos lexers avec l'option `-lfl` ou bien écrit nos scripts `.lex` avec la directive `%noyywrap`.

Ainsi, nous pouvions nous passer de définir la fonction yywrap — une version rudimentaire retournant `TRUE` est implémentée dans `libfl.a` — dans nos documents `.lex`.

### 7.1.1 La fonction `yywrap`

Cette fonction que nous avons ignoré est appelée par le lexer quand il rencontre la fin du flux (*EOF*). Elle doit, soit obtenir un nouveau flux d'entrée et retourner `FALSE`, soit renvoyer simplement `TRUE`, signifiant que la totalité des flux a été consommée et que le lexer a fini sa tâche.

Notez que si vous avez des expressions portant sur `<<EOF>>`, vous devrez vous-même appeler explicitement yywrap et vérifier sa valeur de retour.

Le listing 17 propose un exemple de fonction yywrap qui permet d'enchainer l'analyse d'une suite de fichiers contenue dans une liste chainée, décrivant ainsi la procédure.

Notez que le flux d'entrée est FILE *yyin. La méthode à respecter pour le modifier est toutefois particulière.

Listing 17 – Chainage d'entrée depuis une liste de fichiers

```
//Une liste de noms de fichiers
struct file_list
{
  struct file_list *next;
  char *name;
};

//Liste globale, initialisée par
//le point d'entrée du programme.
struct file_list *flist;

//La fonction yywrap
int yywrap(void)
{
 start:
  if(flist)
    {
      //Overture du fichier
      yyin = fopen(flist->name, "r");

      if(!yyin)
        goto yyin_err;
```

```
        //Charge le nouveau buffer associé au fichier
        // (Cette ligne est facultative car, pour des
        // raisons de compatibilité, Flex assure cette
        // tâche dans le cas où l'on se contente
        // de faire pointer yyin sur un nouveau FILE*.
        // Par souci d'homogénéitée on préfèrera
        // toutefois ajouter cette ligne.)
        //
        //NB: Une alternative possible est:
        //      yyrestart(yyin);
        yy_switch_to_buffer(yy_create_buffer(yyin, YY_BUF_SIZE));

        //Eventuellement, on réinitialise les
        //start-conditions.
        BEGIN(INITIAL);

        //Le prochain fichier sera le suivant
        flist = flist->next;
        //On laisse le lexer se charger du fichier ouvert
        return 0;
      }

  //Il ne reste aucun fichier
  return 1;


 yyin_err:
  //On affiche un message
  fprintf(stderr,
          "Warning: Impossible d'ouvrir \"%s\"\n",
          flist->name);
  //On passe au fichier suivant
  flist = flist->next;
  goto start;
}
```

### 7.1.2 La fonction `yyrestart`

La fonction `yyrestart`, prototypée **void** yyrestart(FILE* new_file) permet de réinitialiser le lexer de façon à ce que le flux d'entrée soit maintenant associé à `new_file`.

Attention, cela n'affecte en rien les start-conditions. C'est à vous d'éventuellement les réinitialiser si cela est nécessaire.

### 7.1.3 Remplissage du buffer

La lecture des caractères s'effectue par l'intermédiaire d'un buffer, crée avec la méthode `yy_create_buffer`.

Ce remplissage est effectué par la macro YY_INPUT() que vous pouvez redéfinir. L'exemple 18 redéfinit YY_INPUT() pour permettre de ne lire qu'un unique caractère à la fois.

Listing 18 – Redéfinition d'`YY_INPUT`

```
#define YY_INPUT(buf, result, max_size) \
        { \
        /* Lecture d'un unique caractère */
        int c = getchar(); \
        /* Si aucun caractère n'est lu, on renvoie 0 */ \
        if (c == EOF) \
                result = YY_NULL; \
        /* Dans le cas de n caractères lus, on renvoie n */ \
        else \
        { \
        buf[0] = c; \
                result = 1; \
        }
```

### 7.1.4 Modifier la sortie

Vous pouvez aussi modifier où pointe le flux de sortie. Pour ce faire, vous disposez de la variable globale FILE *yyout que vous pouvez réassigner comme bon vous semble.

## 7.2 Imbrication de flux d'entrée

Le mécanisme précédent fonctionne parfaitement pour enchainer des flux les uns *après* les autres, mais peut aussi être adapté pour imbriquer des flux. Comprenez par là, interrompre l'analyse d'un flux, puis en analyser un second, avant de reprendre l'analyse du premier.

On peut souhaiter répéter ce procédé sans limite d'imbrication maximale. La meilleur méthode s'avère alors d'utiliser une pile de contexte où chacun des contextes se révèle être le buffer associé à chaque `FILE*`.

Bien que vous pouvez manipuler vous-même une pile de buffer, Flex vous propose un jeu de fonctions mettant à disposition une pile *dynamique*.

C'est ce qu'illustre le listing 19 en réalisant un simple lexer qui se contente d'évaluer les inclusions `#include` du langage C. Cet exemple ne tient pas compte des directives prèprocesseur `#define` qui permettent d'éviter certains problèmes d'inclusion circulaire.

Listing 19 – Interprétation de l'include C/C++

```
/*
 * Les états "inc_begin" et "inc_end" correspondent
 * respectivement au début de la lecture d'une directive
 * d'inclusion et à la fin de cette directive
 */
%x inc_begin inc_end
%%
#include          BEGIN(inc_begin);
```

```
<inc_begin,inc_end>[[:blank:]]+ /* Consomme les blancs */

<inc_begin><.+>          |
<inc_begin>\".+\"        {
        /* Supprime le dernier '"' ou '>' */
        yytext[yyleng - 1] = '\0';
        /* Supprime le premier '"' ou '<' */
        yyin = fopen(yytext + 1, "r");

        if(!yyin)
          fprintf(stderr, "Warning: can't open file %s at line %d");

        /* Ajoute le nouveau buffer */
        yypush_buffer_state(yy_create_buffer(yyin, YY_BUFF_SIZE));

        BEGIN(inc_end);
                        }

<inc_begin>[[:^blank:]]          |
<inc_end>[^[:blank:]\n]          {
        fprintf(stderr, "Warning: caractere inatendu\n");
                                 }

<inc_end>\n             ECHO; BEGIN(INITIAL);

<*><<EOF>>      {
                /* Restore le buffer précédent */
                yypop_buffer_state();

                /* Si l'on vient de supprimer le premier buffer,
                   on peut alors terminer le lexer. */
                if(!YY_CURRENT_BUFFER)
                  yyterminate();
                }

 /* Le code n'est pas modifié */
[a-z]+          ECHO;
[^a-z\n]*       ECHO;
\n              ECHO;
```

## 7.3 Interface et outils fournis par lex

Voici un récapitulatif du prototype des différentes fonctions et macros employé dans les exemples précédents, avec leurs descriptions.

### 7.3.1 Manipulation des buffers

**Fonction:** YY_BUFFER_STATE yy_create_buffer(FILE *file, **int** size)

Cette fonction permet de créer un buffer associé au flux `file` pouvant contenir simultanément `size` octets.

Elle retourne un `YY_BUFFER_STATE` qui s'avère être un pointeur sur une instance de `struct yy_buffer_state`. Vous pouvez donc initialiser vos `YY_BUFFER_STATE` avec `((YY_BUFFER_STATE)0)`.

Le paramètre `file` est la valeur stockée dans yyin à l'apel de `YY_INPUT()`. Bien sûr, si vous redéfinissez `YY_INPUT` de manière à ne plus employer yyin, vous pouvez alors passer un pointeur nul en argument.

**Fonction:** YY_BUFFER_STATE yy_new_buffer(FILE *file, **int** size)

Un simple alias d'`yy_create_buffer`.

**Fonction:** **void** yy_delete_buffer(YY_BUFFER_STATE buffer)

Libère le buffer représenté par l'argument `buffer`.

Si `buffer` est nul, cette méthode est sans effet.

**Fonction:** **void** yy_switch_to_buffer(YY_BUFFER_STATE new_buffer)

Cette fonction permet de substituer `new_buffer` à `YY_CURRENT_BUFFER` et ainsi d'altérer la source du lexer.

Le buffer actuel ne sera pas détruit. Vous pouvez donc conserver une copie de la valeur d'`YY_CURRENT_BUFFER` avant l'appel de la fonction.

**Fonction:** **void** yy_flush_buffer(YY_BUFFER_STATE buffer)

La fonction `yy_flush_buffer` permet de vider le contenu actuellement contenu dans le buffer `buffer`.

À la prochaine lecture d'un lexème par votre lexer, la macro `YY_INPUT` sera appelé pour remplir le buffer.

**Fonction:** **void** yypush_buffer_state(YY_BUFFER_STATE buffer)

L'appel de cette fonction, dans un premier temps, pousse le buffer actuel — contenu dans `YY_CURRENT_BUFFER` — sur une pile interne dynamique — la taille de la pile augmente autant que nécessaire, et n'est limitée que par la mémoire de votre système — puis, dans un second temps, change le buffer courant en celui de l'argument, `buffer`.

Cette fonction permet donc de préserver l'état du buffer courant pour, par la suite, pouvoir le restaurer et continuer à procéder à l'analyse de ce dernier.

**Fonction:** **void** yypop_buffer_state()

Fonction réciproque de la précédente, `yypop_buffer_state` agit aussi en deux temps.

Elle commence par détruire le buffer courant en appelant `yy_delete_buffer(YY_CURRENT_BUFFER)`.

Enfin, elle supprime du sommet de la pile le dernier buffer qui s'y trouve, et change le buffer courant pour ce dernier.

Si l'on atteint la base de la stack, `YY_CURRENT_BUFFER` prendra alors la valeur `YY_NULL`.

### 7.3.2 Sources *en mémoire*

Voici quelques fonctions qui permettent de créer des buffers prenant leur source directement en mémoire.

Les deux premières créent une copie des données, étant donné que les lexers de Flex modifient le contenu du buffer durant le processus d'analyse.

La troisième méthode permet de ne pas effectuer de copie et suppose donc que les données sont modifiables.

**Fonction:** YY_BUFFER_STATE yy_scan_string(**const char** *str)
Cette fonction permet de créer un buffer à partir d'une chaine de caractères C standard, c'est-à-dire terminée par le caractère de fin de chaine `'\0'`.

**Fonction:** YY_BUFFER_STATE yy_scan_bytes(**const char** *bytes, **int** len)
Permet de créer un buffer à partir des `len` octets situés à l'adresse `bytes`.
La chaine peut contenir des caractères de fin de chaine (caractères nul).

**Fonction:** YY_BUFFER_STATE yy_scan_buffer(**char** *base, yy_size_t size)
Cette fonction crée un buffer dont le contenu commence à l'adresse `base` et mesure `size` octets.
Les deux derniers octets de `base` — c'est-à-dire `base[size - 1]` et `base[size - 2]` — doivent contenir le caractère `YY_END_OF_BUFFER_CHAR` qui n'est autre que le caractère de fin de chaine C, `'\0'`.
Le contenu analysé se situe donc de l'adresse `base + 0` à `base + (size - 2)`.

### 7.3.3 Quelques macro/constante

**Macro:** YY_INPUT(buf, result, max_size)
Cette macro est utilisée pour remplir le buffer courant une fois vide.
Ses arguments sont respectivement :
- un tableau de caractères `buf` qui se trouve être la destination
- le nombre de caractères lus, stocké dans `result`
- la taille maximale de buf et donc le nombre maximum de caractères

**Constante:** YY_CURRENT_BUFFER
Permet d'accéder à l'état du buffer courant.
Cette valeur ne doit **jamais** être utilisée à gauche d'une affectation.

# 8 Générer un analyseur lexical C++

Pour les cas qui pourraient vous amener à manipuler un langage orienté objet comme C++ et employer Flex pour générer un lexer, il est possible de compiler le code C tel quel avec g++.

Vous pourrez alors bien sûr profiter de tous les atouts du C++.
Malheureusement, les flux resteront des FILE* et non des flux — `std::stream`.
Vous devriez normalement pouvoir compiler votre lexer sans erreurs (reportez-vous à la section 13.6 si vous en rencontrez). Mais cela ne correspond pas vraiment à une approche objet.

Flex fournit donc un mode de génération C++ — ce projet est encore expérimental — où le fichier généré — lex.yy.cc au lieu de lex.yy.c — est alors l'implémentation d'une classe C++.

Notez que le fichier de sortie par défaut devient alors *lex.yy.cc* au lieu de *lex.yy.c*.

## 8.1 Comment générer un parseur C++

Pour forcer la génération du code de votre lexer en C++, vous pouvez ajouter l'option `%option c++` dans les déclarations de votre fichier .lex, ou bien ajouter l'option `-+` — qui revient à *−c++* — dans votre ligne de commande lors de la compilation.

## 8.2 L'interface C++

Flex déclare deux classes dans le header FlexLexer.h — que vous pouvez trouver dans `/usr/include` pour une installation par package sous une distribution Linux — qui forment l'interface que vous manipulerez pour utiliser le lexer généré.

## 8.3 La classe `FlexLexer`

La classe FlexLexer est une classe abstraite dont hérite tout lexer. Elle représente donc le squelette d'un lexer, fournissant des alternatives aux habituelles constantes et fonctions que nous avons rencontrés.
Voici les fonctions membres que vous pouvez y trouver.

**Fonction:** **const char*** YYText() **const**
Permet d'obtenir la chaine de caractères du dernier lexème identifié par le lexer.
C'est un équivalent de yytext.

**Fonction:** **int** YYLeng() const
Renvoie la taille du dernier lexème identifié par le lexer.
C'est l'équivalent de yyleng ou encore strlen(TTText()).

**Fonction:** **int** lineno() **const**
Dans le cas où l'option `%option yylineno` est activée (cf 13.4.1), cette fonction vous permet d'obtenir le numéro de la ligne — pour le buffer actif — où se trouve le lexème identifié. Si l'option n'est pas activée, la valeur est initialisée à 1.

**Fonction:** **void** set_debug(**int** flag)
Permet de changer les flags de débogage.
C'est une équivalence de `yy_flex_debug()`.
Pour plus d'informations, référez-vous au manuel.

**Fonction:** **int** debug() **const**
Permet d'obtenir les flags de débogage.

**Autres fonctions:** Les fonctions suivantes sont équivalentes à leurs alternatives C.
Notez l'utilisation de std::istream pour les flux.
- **virtual struct** yy_buffer_state* yy_create_buffer(std::istream* s, **int** size)
- **virtual void** yy_delete_buffer(**struct** yy_buffer_state* b)
- **virtual void** yy_flush_buffer(**struct** yy_buffer_state* b) [15]
- **virtual void** yy_switch_to_buffer(**struct** yy_buffer_state* new_buffer)
- **virtual void** yyrestart(std::istream* s)

15. En réalité, cette fonction est implémentée dans `yyFlexLexer`.

## 8.4 La classe `yyFlexLexer`

Cette seconde classe [16] correspond à l'implémentation de votre lexer.

L'implémentation de ses fonctions membres a lieu dans le fichier `lex.yy.cc`, généré par flex.

**Fonction:** yyFlexLexer(istream* arg_yyin = 0, ostream* arg_yyout = 0)

Le constructeur de la classe `yyFlexLexer`, qui prend en paramètres ses flux d'entrée et de sortie.

Si aucune option n'est spécifiée, les flux seront `cin` et `cout`.

**Fonction:** **virtual int** yylex() Cette fonction correspond à l'`yylex()` des analyseurs C. Pour rappel, cette fonction consomme le flux d'entrée, identifie les lexèmes, et exécute les actions associées jusqu'à interruption due, soit à une action — une instruction return que vous avez vous-même écrit dans le fichier `.lex` —, soit à la consommation de la totalité du flux.

Vous pouvez éventuellement avoir une classe `Y` qui hérite d'`yyFlexLexer` et souhaiter accéder aux données membres de `Y` dans la fonction `yylex()`. Dans ce cas, Flex permet d'implémenter la fonction membre Y::yylex() ainsi qu'une fonction yyFlexLexer::yylex() qui provoque l'appel de yyFlexLexer::LexerError() afin de prévenir son utilisation. Pour ce faire, vous devez utiliser l'option `%option yyclass="Y"`.

**Fonction:** **virtual void** switch_streams(istream* new_in = 0, ostream* new_out = 0)

Cette fonction permet de réaffecter les flux d'entrée et de sortie du lexer.

Dans le cas où l'un — ou les deux — des paramètres est nul, le flux correspondant ne sera pas affecté.

**Fonction:** **virtual int** yylex(istream* new_in, ostream* new_out)

Dans un premier temps, appelle switch_streams(new_in, new_out) puis dans un second passe la main à yylex().

**Fonction:** **protected**: **virtual int** LexerInput(**char** *buf, **int** max_size) Cette fonction lit jusqu'à `max_size` caractères depuis std :: istream* yyin et les stocke dans `buf`. Elle retourne le nombre de caractères placé dans buf. La valeur de retour 0 correspond à la fin du flux.

Sachez que les lexers interactifs définissent la macro `YY_INTERACTIVE`, et vous pouvez donc implémenter deux traitements différents selon si le lexer est interactif ou non, via les directives préprocesseur `#ifdef`, `#ifndef` et `#endif`.

Cette fonction correspond bien evidemment à la macro `YY_INPUT()` (cf section 7.1.3).

**Fonction:** **virtual void** LexerOutput(**const char*** buf, **int** size)

Cette fonction écrit les `size` caractères à partir de l'adresse `buf` vers la sortie std :: ostream* yyout.

La chaine de caractères `buf` est terminée par un caractère de fin de chaine `'\0'` mais peut éventuellement contenir des `'\0'` si votre analyseur contient des règles pouvant en contenir.

**Fonction:** **virtual void** LexerError(**const char*** msg)

Affiche un message d'erreur fatale à l'application.

L'implémentation fournie par défaut provoque la sortie de `msg` vers `cerr` puis termine l'application.

16. Pour faciliter l'approche vers les nouvelles fonctions, certaines fonctions déjà déclarées dans FlexLexer figurent dans cette partie.

Vous pouvez consulter le fichier FlexLexer pour la portée et les déclarations exactes des fonctions.

## 8.5 Quelques remarques

L'implémentation de yytext est un pointeur et vous ne pouvez pas utiliser l'option %array. Tenez compte de ces considérations si vous souhaitez utiliser `unput()`.

Sachez aussi que les instances d'`yyFlexLexer` possèdent leurs propres contextes et peuvent donc être utilisées comme des analyseurs réentrants (cf section 10).

Vous pouvez aussi utiliser plusieurs instances du même lexer sans risque de conflit, cela grâce à l'absence de variable globale ou partagée.

## 8.6 Une application

Afin de replacer ces informations dans leur contexte et d'illustrer par la pratique l'utilisation d'un lexer en C++, voici un exemple élémentaire qui permet en une lecture d'avoir un aperçu.

L'exemple 20 se contente d'identifier quelques éléments et de provoquer leur affichage avec une syntaxe C++ — utilisation du flux de sortie `std::cout`.

Listing 20 – Multiples lexers C++

```
%{
#include <iostream>

/* Compteur de lignes */
int num_ligne = 0;

/* Valeur actuelle du compteur dans une action */
#define NL      (const_cast<const int>(num_line))
%}

/* Blancs */
blancs    [[:blank:]]+

alpha     [[:alpha:]]
dig       [[:digit:]]
identif   ({alpha}|{dig}|\$)({alpha}|{dig}|[_.\-/$])*
num1      [-+]?{dig}+\.?([eE][-+]?{dig}+)?
num2      [-+]?{dig}*\.{dig}+([eE][-+]?{dig}+)?
/* Nombre */
nombre    {num1}|{num2}

/* Autre */
autre     [.]
%%

{blancs}          /* Ignore les espaces et tabulations */

"/*"              {
  int c;

```

```
  /* Tant que l'on parvient à lire un caractère */
  while((c = yyinput()) != 0)
    {
      /* C'est une ligne */
      if(c == '\n')
        ++num_ligne;

      /* C'est peut-être une fin de commentaire? */
      else if(c == '*')
        {
          /* Fin du commentaire */
          if((c = yyinput()) == '/')
            break;
          /* Le caractère est replacé dans le buffer
             pour être traité à nouveau par la
             condition de cette boucle. */
          else
            unput(c);
        }
    }
}

{nombre}        std::cout << NL << "nombre " << YYText() << '\n';
{identif}       std::cout << NL << "id: " << YYText() << '\n';
{chaine}        std::cout << NL << "chaine: " << YYText() << '\n';
{autre}         std::cout << NL << "autre:" << YYText() << '\n';

\n              num_line++; /* Tient compte de la ligne */

%%
/* Une utilisation triviale */
int main(int /* argc */, char** /* argv */)
{
  FlexLexer* lexer = new yyFlexLexer;
  while(lexer->yylex() != 0)
    ;
  return 0;
}
```

## 8.7 De multiples analyseurs

Comme indiqué dans le fichier *FlexLexer.h*, il est possible de faire cohabiter plusieurs lexers dans un même projet.

Il suffit dans un premier temps de redéfinir le préfixe (*yy* dans *yyFlexLexer*) avec l'option `%option prefix="zz"` — vous pouvez aussi utiliser le paramètre `-Pzz` à la compilation.

Par la suite, on utilisera le code suivant pour inclure la déclaration de chacune des classes :

Listing 21 – Multiples lexers C++

```
/* Utilisation des lexers xxFlexLexer et yyFlexLexer */
#undef yyFlexLexer
#define yyFlexLexer xxFlexLexer
#include <FlexLexer.h>

#undef yyFlexLexer
#define yyFlexLexer zzFlexLexer
#include <FlexLexer.h>

// ...

FlexLexer *lex1 = new xxFlexLexer();
FlexLexer *lex2 = new zzFlexLexer();
```

# 9 Sérialisation de l'automate

Flex génère des analyseurs lexicaux sous la forme d'*Automates Finis Déterministe*(cf section 12).

Ce mécanisme est extrêmement efficace et performant mais présente toutefois l'inconvénient de générer une table de correspondance particulièrement volumineuse.

Si vous utilisez plusieurs analyseurs où que l'analyse n'est qu'une phase d'exécution de votre programme, vous pouvez souhaiter libérer la mémoire utilisée par l'analyseur.

Par défaut, la table est stockée sous forme d'un tableau C inclus dans le binaire. Flex propose alors un moyen de charger dynamiquement des tables depuis des fichiers en mémoire, puis de les détruire une fois qu'elles sont devenues inutiles.

## 9.1 Génération de tables externes

La génération de tables externes s'effectue par l'ajout de l'option `-table-file=FILE` à la compilation ou encore `%option tables-file=FILE` dans votre fichier `.lex`.
*FILE* correspond ici au fichier de sortie. Si vous ne le spécifiez pas explicitement — c'est-à-dire en utilisant seulement l'option `%option tables-file` — le fichier de sortie par défaut sera `lex.yy.tables`, où `yy` se verra substitué par le préfixe de votre scanner (voir l'option `prefixe` à la section 13.10).

Si vous utilisez plusieurs lexers, et que vous souhaitez charger/décharger leur table en une fois, vous pouvez container plusieurs fichiers de sortie en un seul.

Le listing 22 génère deux analyseurs avec pour préfixe *c* et *cpp*. Pour chacun, un fichier de table externe est généré. Une fois les fichiers obtenus, ils sont concaténés en un unique fichier à l'aide de la commande *cat*.
A l'exécution chacun des lexers saura retrouver sa propre table à l'aide de son nom — il est en effet impossible de lier deux analyseurs avec les mêmes noms de fonction et de variables globales, et deux tables de même nom donneraient lieu à un conflit — imposé par le choix des préfixes.

Listing 22 – Concaténation de table

```
$flex --tables-file --prefix=cpp    cpp.l
$flex --tables-file --prefix=c      c.l
$cat lex.cpp.tables lex.c.tables > all.tables
```

## 9.2 Charger et libérer les tables

Après avoir généré des fichiers de table externe, il vous faut les charger en mémoire afin que vos analyseurs lexicaux puissent s'exécuter correctement.

Flex vous fournit les deux fonctions suivantes.

**Fonction:** **int** yytables_fload (FILE* fp [, yyscan_t scanner])
Cette fonction permet de charger en mémoire les tables contenues dans le flux pointé par `fp`.
Le second paramètre n'est utilisé que par les analyseurs réentrants (cf section 10).
La fonction renvoie `0` en cas de succès et une valeur non nulle en cas d'erreur.

**Fonction:** **int** yytables_destroy([yyscan_t scanner])
Cette fonction permet de libérer la mémoire allouée pour les tables.
Là encore, le paramètre `scanner` ne s'applique qu'aux analyseurs réentrants.
Tout comme la fonction précédente, la valeur de retour est `0` en cas de succès et une valeur non nulle en cas d'erreur.

Vous devez **explicitement** charger les tables **avant toute utilisation d'**`yylex()` et les libérer quand vous ne souhaitez plus les utiliser. Si vous ne faites pas appel à `yytables_destroy`, la mémoire ne sera pas libérée jusqu'à la fin de l'exécution du lexer.

Il est important de savoir que les fonctions `yytables_fload` et `yytables_destroy` **ne sont pas thread-safe**.
C'est à vous de vous assurer que leur appel sera unique.

Vous pouvez bien sûr charger et décharger des tables autant de fois que vous le désirez. Vous pouvez donc les détruire durant les périodes où votre application ne fait pas appel à votre analyseur lexical.

## 9.3 Une brève introduction au format de fichier

Nous ne décrirons pas en détail le format de fichier des tables externes. Nous ne ferons qu'une brève description et le lecteur pourra, s'il le désire, consulter la documentation.

Un fichier `.tables` peut contenir de nombreux jeux de tables juxtaposés les uns après les autres. On notera donc qu'il est nécessaire de lire complètement la première table pour savoir si une deuxième lui succède.

Un jeu de table est constitué d'une *en-tête* dont les quatre premiers octets sont un *magic-nomber* — un code permettant de reconnaitre le format du fichier — ayant pour valeur hexadécimale `0xF13C57B1`. Le header est aligné à 64 octets grâce à des "octets de bourrage", et a donc une taille fixe.
Suivent derrière cette en-tête les différents tableaux de données constituant la table du DFA.
Le listing 23, provenant de la documentation, représente les tables d'N automates.

Listing 23 – Cancaténation de table

```
                TABLE SET 1
               +-----------------------------------+
Header         | uint32          th_magic;         |
               | uint32          th_hsize;         |
               | uint32          th_ssize;         |
               | uint16          th_flags;         |
               | char            th_version[];     |
               | char            th_name[];        |
               | uint8           th_pad64[];       |
               +-----------------------------------+
Table 1        | uint16          td_id;            |
               | uint16          td_flags;         |
               | uint32          td_lolen;         |
               | uint32          td_hilen;         |
               | void            td_data[];        |
               | uint8           td_pad64[];       |
               +-----------------------------------+
Table 2        |                                   |
    .          .                                   .
    .          .                                   .
    .          .                                   .
    .          .                                   .
Table n        |                                   |
               +-----------------------------------+
                TABLE SET 2
                      .
                      .
                      .
                TABLE SET N
```

# 10 Analyseurs réentrants

Flex permet aussi de générer des analyseurs lexicaux réentrants. On peut qualifier de réentrante une fonction qui peut s'appeler *elle-même* sans que cela n'ait de conséquences sur son fonctionnement, ou encore être utilisées *simultanément*. Pour ce faire, elle ne doit agir que sur un contexte qu'elle reçoit en paramètre.

On peut considérer de toute fonction qui ne modifie aucune valeur non locale qu'elle est constante (un exemple est une fonction "**float** sinus(**float** a)" qui calculerait le sinus de `a` localement et retournerait le résultat) et donc a fortiori réentrante.

Une fonction réentrante est nécessairement *thread-safe*, sous réserve bien sûr que le contexte qu'elle reçoit en argument (nous parlerons d'environnement pour qualifier ces valeurs qu'elle s'autorise à modifier, et nécessaire au bon fonctionnement d'*yylex*) ne soit pas partagé par plusieurs threads.

L'utilisation d'analyseurs réentrants ne présente aucune difficulté supplémentaire mais demande le respect de certaines contraintes, dues à la modification de l'API induite par le critère de réentrance et la présence d'un *contexte* propre à chaque *instance* de lexer.

## 10.1 Quelques exemples qui peuvent amener à l'utilisation d'un analyseur réentrant

On peut déjà citer trois cas de figure où l'utilisation d'un analyseur réentrant peut être utile.

La comparaison de la structure de deux fichiers au niveau de lexème, et non des caractères eux-mêmes, nécessite l'utilisation simultanée du même lexer sur deux flux d'entrés. On pourrait en effet s'en passer, mais il serait alors nécessaire, soit de dupliquer l'analyseur, soit de mémoriser les résultats de la première analyse dans une file pour effectuer la comparaison durant la seconde, obligeant ainsi à analyser la totalité du premier fichier.

L'exemple du listing 24 présente un exemple simple permettant d'effectuer une différence entre deux fichiers.

Listing 24 – Diff par lexème

```
//...
// Initialisation des analyseurs
//...

//Les lexèmes identifiés
int token1;
int token2;

//Initialisation pour effectuer une première passe
token1 = 1;
token2 = 1;
//Tant que le dernier token lu n'est pas YY_NULL
while(token1 && token2)
{
  token1 = yylex(scanner_1);
  token2 = yylex(scanner_2);

  if(token1 != token2)
    printf("Les fichiers difèrent à la ligne %d\n",
                yyget_lineno(scanner_1);
}

//...
// Désallocation des ressources
//...
```

Le souhait de disposer d'un analyseur récursif justifie tout à fait l'utilisation du mode réentrant. On peut, par exemple, souhaiter analyser un lexème comme un flux. Si vous trouvez ceci obscur, considérez alors un lexème `eval` suivi d'une chaine de caractères à évaluer suivant le même

langage que celui que vous analysez actuellement. Voyez qu'un `eval` peut lui-même contenir un `eval`, et cela récursivement, avec une profondeur variable. L'exemple du listing 25 correspond à ce cas de figure.

Listing 25 – Diff par lexème

```
%option reentrant

%%
"eval(".+");"    {
        //L'analyseur
        yyscan_t scanner;
        //Le flux provenant du lexème en cours
        YY_BUFFER_STATE buf;

        //Initialisation de l'analyseur
        yylex_init(&scanner);

        //Création d'un flux
        yytext[yyleng-1] = ' ';
        buf = yy_scan_string(yytext + 5, scanner);

        //Analyse
        yylex(scanner);

        //Libération des ressources
        yy_delete_buffer(buf, scanner);
        yylex_destroy(scanner);
}

        //...
%%
```

Enfin, on peut souhaiter utiliser une ou plusieurs instances d'un analyseur dans un contexte *multi-thread*. La présence de variable globale ne permet pas d'avoir un comportement thread-safe et nécessite donc que vous preniez de multiples précautions pour vous assurer qu'il ne se produise pas d'erreurs. Il est alors beaucoup plus simple, et plus sûr, d'utiliser un analyseur réentrant, qui est alors *thread-safe*.

## 10.2 Déclaration, instanciation et libération

Avant toute chose, il est nécessaire de spécifier l'option `%option reentrant` — ou encore `-R` ou `-reentrant` à la compilation.

Par la suite, vous devez initialiser un environnement avant de procéder à l'analyse. Vous créez donc une structure — elle peut être dans la pile ou allouée dynamiquement — correspondant à une *intance* de l'analyseur, qu'il vous faut initialiser à l'aide de la fonction yylex_init.

Vous pouvez procéder à l'identification des lexèmes avec un appel à yylex qui prend maintenant en paramètre l'environnement de votre lexer.

Enfin, votre analyse effectuée et terminée, vous pouvez libérer les ressources mémoire utilisées à l'aide de la fonction yylex_destroy.

Voici les prototypes des différentes fonctions mentionnées :
- **int** yylex_init(yyscan_t *ptr_yy_globals)
- **int** yylex(yyscan_t yyscanner)
- **int** yylex_destroy(yyscan_t yyscanner)

## 10.3 Utilisation

L'utilisation d'une version réentrante d'un analyseur Flex diffère très peu de la version non réentrante. Tout d'abord, la totalité des globales que vous connaissiez sont remplacées par des macros du même nom correspondant à l'accès aux données de la structure yyscan_t. Vous écrirez donc vos actions exactement comme vous l'auriez fait avec un analyseur non réentrant.

Voici quelques unes des macros dont il est question :
- **#define** yyin yyg->yyin_r
- **#define** yyout yyg->yyout_r
- **#define** yyleng yyg->yyleng_r
- **#define** yytext yyg->yytext_r
- **#define** yylineno yyg->yylineno_r

Ensuite, sachez que les méthodes usuelles se voient toutes agrémentées d'un dernier paramètre supplémentaire, de type yyscan_t. Leur appel s'effectue donc comme auparavant, avec l'ajout de ce dernier paramètre. La structure yyscan_t, correspondant à l'instance de l'analyseur en cours, est accessible dans les actions via la variable locale yyscan_t yyscanner que vous avez vous-même passé en paramètre à l'appel d'yylex.

Voici un exemple mettant en scène un analyseur réentrant :

Listing 26 – Exemple d'utilisation d'un analyseur réentrant

```
%option reentrant stack noyywrap
%x COMMENT
%%

"//"                    yy_push_state(COMMENT, yyscanner);
.|\n                    /* Ne fait rien */

        /* Affichage du commentaire et retour à l'état précédent */
<COMMENT>\n             yy_pop_state(yyscanner);
<COMMENT>[^\n]+         fprintf(yyout, "%s\n", yytext);

%%

int main ( int argc, char * argv[] )
{
        /* Contiendra l'instance du lexer */
        yyscan_t scanner;

        /* Initialise le lexer */
        yylex_init(&scanner);

        /* Effectue l'analyse */
        yylex(scanner);
```

```
25          /* Destruction de l'instance */
            yylex_destroy(scanner);

            return 0;
}
```

## 10.4 Manipulation de l'environnement

Si vous disposez de macros permettant d'accéder aux données de l'environnement de l'analyseur à l'intérieur de vos actions, vous pouvez aussi y accéder à l'extérieur à l'aide de la structure yyscan_t de votre analyseur.

Flex fournit alors un jeu d'accesseurs et de mutateurs détaillés ci-dessous. Notez que si vous souhaitez modifier le contenu d'`yytext`, vous pouvez utiliser l'accesseur qui vous renvoie un pointeur modifiable.

- **char** *yyget_text(yyscan_t scanner)
- **int** yyget_leng(yyscan_t scanner)
- **int** yyget_lineno(yyscan_t scanner)
- FILE *yyget_in(yyscan_t scanner)
- FILE *yyget_out(yyscan_t scanner)
- **int** yyget_debug(yyscan_t scanner)
- **void** yyset_lineno(**int** line_number , yyscan_t scanner)
- **void** yyset_in(FILE *in_str , yyscan_t scanner)
- **void** yyset_out(FILE *out_str , yyscan_t scanner)
- **void** yyset_debug(**int** flag, yyscan_t scanner)

La variable locale à l'environnement `yylineno` n'est modifiée que si l'option `%yylineno` est activée. Dans le cas contraire, elle est initialisée à 1 et n'est plus modifiée.

## 10.5 Ajouter des données à l'environement

Vous pouvez partager des données entre votre application et les actions de votre analyseur. Vous fournissez alors un pointeur sur une structure qui contiendra vos données qui sera accessible par la macro `yyextra` au sein de vos actions.

L'initialisation devra alors utiliser la méthode yylex_init_extra. Vous pourrez par la suite modifier ce pointeur à l'aide des accesseurs et mutateurs yyset_extra et yyget_extra. Le type de ce pointeur, dans lequel vous pouvez convertir le vôtre, est défini par la ligne #**define** YY_EXTRA_TYPE **void***. Vous pouvez modifier cette définition par l'utilisation de l'option `%option extra-type="votre_type"`.

Les prototypes des trois méthodes mentionnées :

- **int** yylex_init_extra(YY_EXTRA_TYPE user_defined, yyscan_t *ptr_yy_globals)
- YY_EXTRA_TYPE yyget_extra(yyscan_t scanner)
- **void** yyset_extra(YY_EXTRA_TYPE user_defined , yyscan_t scanner)

Pour conclure, voici un exemple tiré de la documentation qui illustre l'utilisation de données supplémentaires adjointe à l'environnement.

Listing 27 – Utilisation d'un environement étendu

```
%{
        #include <sys/stat.h>
        #include <unistd.h>
%}
```

```
%option reentrant
        /* Pour plus d'informations, man 2 stat */
%option extra-type="struct stat *"
%%

        /* Remplace __filesize__ et __lastmod__
           par la taille du fichier / la dernière
           date de modification du fichier */
__filesize__    printf("%ld", yyextra->st_size);
__lastmod__             printf("%ld", yyextra->st_mtime);
%%
/* Analyse un fichier */
void scan_file(char* filename)
{
        yyscan_t scanner;
        struct stat buf;
        FILE *in;

        /* Ouvre un fichier */
        in = fopen(filename, "r");
        /* Obtient ses informations */
        stat(filename, &buf);

        /* Initialise l'analyseur */
        yylex_init_extra(buf, &scanner);
        /* Modifie le flux d'entrée */
        yyset_in(in, scanner);

        /* Effectue l'analyse */
        yylex(scanner);

        /* Libère les ressources */
        yylex_destroy(scanner);
        fclose(in);
}
```

# 11 Gestion de la mémoire

Nous savons maintenant que Flex fournit divers mécanismes pour nous permettre de réaliser aisément de puissants analyseurs lexicaux.

Nombre de ces derniers nécessitent l'allocation et la désallocation de mémoire.

Il peut donc être intéressant de comprendre la façon dont Flex gère la mémoire et comment vous pouvez redéfinir ce comportement.

## 11.1 La gestion effectuée par défaut

Les lexers générés par Flex allouent de la mémoire dans deux cas :

– le premier cas est l'initialisation du lexer, qui a lieu au premier appel de la fonction `yylex()`.

– le second cas est la nécessité d'allouer plus de mémoire pour par exemple contenir un important lexème dont la taille dépasse celle d'`yytext`.

Vous pouvez provoquer la libération de la mémoire utilisée par un lexer en appelant la fonction **int** yylex_destroy(). Notez que cela n'affecte pas la mémoire allouée par yytables_fload() Voici la liste des principales allocations effectuées :

**16 Ko (16384 octets) pour le buffer de lecture**

Flex alloue un buffer pour effectuer la correspondance des caractères aux expressions régulières. Ce buffer est étendu si nécessaire, chaque extension se traduisant par l'augmentation de sa taille selon un facteur 2. Il n'est pas nécessaire d'avoir un buffer pouvant contenir beaucoup plus de caractères que la plus longue de vos expressions. Si vous le souhaitez, vous pouvez redéfinir la taille de ce buffer a l'aide de la macro #**define** YY_BUF_SIZE.

**64 octets en cas d'utilisation de `REJECT`**

L'utilisation de `REJECT` nécessite une pile d'états pouvant contenir autant d'états que le buffer de lecture peut contenir de caractères.

La redéfinition de YY_BUF_SIZE provoque aussi une modification de la taille de cette pile d'états.

Ce mode de fonctionnement justifie l'impossibilité d'utiliser l'extension dynamique du buffer de lecture en cas d'utilisation de `REJECT`, la pile d'états étant statique.

**100 octets pour la pile des start-conditions**

La pile d'états des *start-conditions* (cf section 6) n'est présente que si vous spécifiez l'option `%option stack`. Les états sont de simples entiers et la pile peut donc déjà en contenir un nombre important. Toutefois, si cela est nécessaire, Flex réalloue la mémoire nécessaire.

**40 octets pour chaque YY_BUFFER_STATE**

Chacun de ces buffers représente un flux d'entrée (cf section 7) et contient un *buffer de lecture*. Flex ne détruit pas automatiquement les buffers alloués (a l'exception de ceux contenus dans la pile, et du buffer courant si vous faites appel à yylex_destroy()). Si vous détruisez vous-même YY_CURRENT_BUFFER, pensez à lui affecter la valeur nulle pour éviter une erreur si votre lexer tente de le détruire une seconde fois.

**84 octets pour l'environement d'un analyseur réentrant**

Chaque structure yyscan_t nécessite l'allocation de 84 octets qui seront libérés à l'appel de yylex_destroy.

## 11.2 Intervenir sur la gestion de la mémoire

Pour les opérations d'allocation, réallocation et désallocation, Flex utilise respectivement les fonctions yyalloc, yyrealloc et yyfree. Par défaut, ces fonctions sont de simples liens vers les fonctions malloc, realloc et free de la bibliothèque standard C.

Vous pouvez redéfinir ces fonctions pour utiliser votre propre mécanisme des gestion de la mémoire, comme par exemple un système de Gorbage Collector (cf libGC) ou encore un mécanisme de Memory Pool.

Les deux étapes suivantes sont alors nécessaires :

1. Supprimer l'implémentation par défaut de Flex via l'une des instructions suivantes :
   - %option noyyalloc
   - %option noyyrealloc
   - %option noyyfree
2. Implémenter vous-même les méthodes avec pour prototype :

```
// Pour un analyseur non réentrant
void *yyalloc(size_t bytes);
void *yyrealloc(void * ptr, size_t bytes);
void  yyfree(void * ptr);

// Pour un analyseur réentrant
void *yyalloc(size_t bytes, void * yyscanner);
void *yyrealloc(void * ptr, size_t bytes, void * yyscanner);
void  yyfree(void * ptr, void * yyscanner);
```

# 12 Théorie : Comprendre Flex

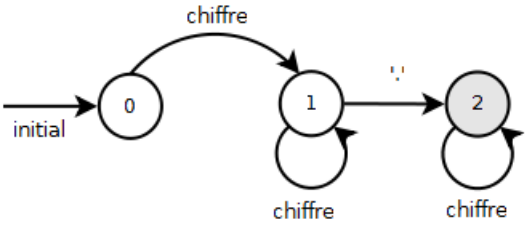
Flex génère des analyseurs lexicaux permettant la reconnaissance d'un langage rationnel. Sans entrer dans les détails, cela correspond exactement aux chaines de caractères pouvant être identifiées par des expressions régulières. Pour reconnaitre ce type de langage, on utilise des "Automates Fini Déterministe", aussi abrégés DFA.

## 12.1 Qu'est-ce qu'un DFA

Nous ne chercherons pas à décrire la théorie des langages rationnels, ni à détailler les différents type d'automates, où encore à définir mathématiquement ces concepts. Cette section s'adresse aux lecteurs qui n'ont aucune notion de ce que peut être un DFA. Si vous connaissez déjà ce mécanisme, vous pouvez passer à la section suivante.

Un DFA est un ensemble d'états, avec un état initial et des états finaux, ainsi qu'une fonction de transition qui permet de passer d'un état à l'autre. La figure 1 présente un simple automate à trois états, numérotés dans les cercles, dont un état initial et un état final en gris.

FIGURE 1 – Un DFA reconnaissant un nombre décimal



Pour représenter la fonction de transition, on nommera des flèches permettant de passer d'un état à un autre d'une étiquette. Cette étiquette est choisie parmi les caractères lus en entrée, c'est-à-dire l'une des 256 valeurs que peut prendre un char[17]. On remarque alors que si l'on suit les transitions indiquées par les caractères lus, pour passer d'état en état, on ne parvient à atteindre l'état final qu'en ne lisant une expression de la forme `[[:digit:]]+"."[[:digit:]]+`.

On démontre que les automates finis déterministes (et non déterministes) sont équivalent à un langage. On démontre aussi qu'ils reconnaissent exactement l'ensemble des langages reconnus par des expressions régulières, qui est exactement l'ensemble des langages rationnels.

### 12.2 Implémentation d'un DFA

Nous allons ici étudier une implémentation simpliste du DFA que nous avons observé, qui a le mérite de ressembler fortement à la façon dont Flex génère ses analyseurs.

La première chose à faire est de créer une table de transition qui fait correspondre à un état et une étiquette un nouvel état. Pour ce faire on peut utiliser un tableau bidimensionnel, de la forme etat_t transition [NB_ETATS][256]. On prend pour état initial l'état 0. Un changement d'état se traduira alors par la simple affectation état_courant = transitions[état_courant][caractère_lu]. Cette table de transition n'est rien d'autre que les fameuses tables que vous pouvez exporter dans un fichier `.tables`.

Quand on rencontre une absence de transition — supposons que notre tableau contienne la valeur -1 pour tout couple (état, caractères) qui ne donne pas lieu à une transition — on pourra alors considérer deux cas :

- on se trouve dans un état "final" et on a identifié une expression. Le dernier caractère lu (ne faisant donc pas partie de l'expression identifiée) sera replacé dans le buffer.
- on ne se trouve pas dans un état "final", le mot n'appartient pas au langage.

Pour identifier rapidement si un état est final, on pourra utiliser une simple table de hachage de la forme bool est_final[NB_ETATS].

Flex génère un unique DFA permettant de reconnaitre toutes les expressions et peut donc effectuer la correspondance de plusieurs expressions en parallèle, ne s'arrêtant que lorsque qu'il a reconnu (ou non) la plus longue expression possible. Les lexers générés par Flex mémorisent leur dernier passage par un état terminal, ainsi que la longueur de la chaine, de façon à pouvoir retrouver, si une plus faible portion de la chaine correspond à un lexème, l'état dans lequel se trouvait l'analyseur, et faire correspondre `yytext`.

## 13 Trucs et Astuces

### 13.1 Commentaires

Vous pouvez placer des commentaires c-style dans un document .lex. Ils seront recopiés tel quels dans le fichier source généré. Il est toutefois nécessaire de respecter certaines règles afin que Flex puisse les différencier des expressions régulières.

Pour les sections de définition et de code, vous pouvez les écrire librement, sous réserve de ne pas les placer après une ligne d'option dans la section de description. Pour la section des règles, afin que Flex puisse différencier vos expressions régulières de vos commentaires, ajoutez une tabulation avant ces derniers.

17. Si l'on se restreint à la table ASCII, alors les valeurs possible sont limitées à 127.

## 13.2 Variables locales

Il est possible de créer des variables locales à la fonction yylex (qui se trouveront donc sur la pile).

Pour ce faire, déclarez-les avant vos expressions régulières, au début de la section des règles, en les précédant d'une tabulation.
Par exemple : `[tabulation]int token_counter;`

Sachez que tout ce qui n'est pas considéré comme une expression est recopié tel quel dans le corps de la fonction *yylex*.

## 13.3 Options de Flex

## 13.4 Comment utiliser une option

Les options se placent dans les déclarations du fichier lex.
Une option est du type *%option <name>[=<value>]* et plusieurs peuvent se placer sur la même ligne. Par exemple : *%option opt1 opt2 opt3*

### 13.4.1 Numérotation des lignes

Flex fournit la possibilité de mémoriser le numéro de ligne du buffer courant.
La valeur correspond donc à `nombre_de_fin_de\n + 1`.

Pour activer cette fonction, il est nécessaire de l'indiquer à Flex avec l'option *%option yylineno* ou encore l'ajout de `-yylineno` à la ligne de compilation.

Vous pouvez alors accéder à cette valeur via la variable globale `yylineno`.

Dans le cas où vous n'avez pas activé l'option, cette variable est présente mais initialisée à 1.

## 13.5 Trailing context

Le *trailing context* est une forme particulière d'expression régulière, qui fait intervenir les caractères succédant au lexème recherché, mais n'en faisant pas partie.
Il est donc question du contexte dans lequel apparait le lexème.

Ainsi, un lexème `'chat'` suivit du caractère `'s'` se traduira par l'expression régulière `'chat/s'`.

Vous pouvez ainsi, via ce mécanisme, analyser certaines des grammaires les plus simples.(Un nombre de cas extrêmement limité, Flex n'étant pas conçu pour ce genre de choses).

Il est toutefois déconseillé d'employer ce mécanisme, altérant les performances et pouvant dans certains cas adopter un comportement indéterminé. Pour plus d'informations, référez-vous au manuel.

## 13.6 YY_BREAK

Flex définit la constante `YY_BREAK`, correspondant au délimiteur de fin d'action.
La constante est définie par #**define** YY_BREAK **break**; et vous pouvez bien sûr la redéfinir comme vous le désirez.

Cela peut s'avérer utile si vous réalisez, par exemple, un lexer que vous souhaitez compiler avec un compilateur C++, et que vos actions contiennent un return.

En effet, dans ce cas de figure vous obtiendrez un warning du compilateur vous indiquant que l'instruction `break` est inatteignable.

## 13.7 Déclaration d'`yylex()`

Il est possible de modifier la définition de la fonction `yylex()`.

Flex fournit en effet cette déclaration sous forme de la macro `YY_DECL` que l'utilisateur peut redéfinir.

Pour par exemple lui permettre de prendre de nouveaux paramètres et de modifier son type de retour, on peut écrire quelque chose de la forme #**define** YY_DECL **double** lexscan(**double** r, **char** k).

Remarquez l'absence de ; justifiée par le fait que la macro est directement suivie par l'implémentation entre accolades.

## 13.8 Aide au développement

Lors du développement de votre analyseur lexical, vous chercherez probablement à établir les bonnes expressions pour les flux que vous devez analyser.

Pour vous faciliter la tâche, sachez que vous disposez des options `-warn` — ou encore `%option warn` — pour afficher des avertissements dans le cas où, par exemple, certains motifs ne correspondent à aucune expression.

Vous disposez aussi des options `-perf-report` | `%option perf-report`, `-backup` | `%option backup` et `-debug` | `%option debug`. Pour plus d'informations, reportez-vous à la documentation.

## 13.9 Fichiers de sortie

Vous pouvez modifier le fichier de sortie de Flex — par défaut `lex.yy.c` — en utilisant l'option `-oFile` | `-outFile=FILE` | `%option outfile="FILE"` où *FILE* est le nom du fichier de sortie.

Pour générer un header de tous les prototypes de fonction et des variables globales, vous pouvez utiliser l'option `-header-file=FILE` | `%option header-file="FILE"`.
Ceci est incompatible avec le mode *C++*.

Enfin, l'option `-tables-file=FILE` que vous avez déjà rencontré à la section 9 permet de sélectionner le fichier de sortie de la table du *DFA*.

## 13.10 Préfixe

Comme expliqué dans la section 8, l'option `-Pzz | -prefixe=zz, %option prefix="zz"` permet de remplacer le préfixe `yy` par `zz`, ceci ayant divers impacts comme la modification des noms de fonctions et variables — par exemple `zzleng` et `zztext` — du lexer.

## 13.11 Performances

Pour améliorer les performances de votre analyseur lexical, vous disposez de plusieurs options comme `-fast | %option fast`. De même, pour réduire la taille de votre analyseur, vous pouvez compresser la table du *DFA* avec par exemple — il existe en effet plusieurs méthodes de compression — l'option `-Cm | %option meta-ecs`.

Pour connaitre plus exactement les effets de ces options, consultez la documentation.

# Troisième partie
# Pour conclure

## 14 Notes de l'auteur

Dans ce texte, nombre d'exemples de code sont extraits du manuel officiel — souvent légèrement modifiés et adaptés aux propos —, disponible en anglais sur sourceforge.com(cf [4]).

Je vous encourage vivement — si bien sûr vous comprenez l'anglais et en avez le temps — à le lire.

Vous y trouverez la liste complète des options, des problèmes d'incompatibilité avec Lex, et de nombreuses informations qui pourraient vous être utile.

Un grand remerciement à Benj. (Benjamin) qui a pris le temps de relire chaque ligne de ce document et de relever chaque faute, sans qui cette publication n'aurait pu avoir lieu.

## Références

[1] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. *Compilateurs : principes, techniques et outils*. Novembre 2007.

[2] Doug Brown, John Levine, and Tony Mason. *Lex & Yacc, Second Edition*. 1992.

[3] Henri Garreta. Techniques et outils pour la compilation. Janvier 2001. `ftp://ftp-developpez.com/general/cours/PolyCompil.pdf`.

[4] Gnu. *Flex manpage*, 2007. `http://flex.sourceforge.net/manual/index.html`.

# Index