

# Corrigé TD 1

## Analyse Lexicale

### Lex / Flex


## 1 Commande upper

Pour mettre en majuscules nous pouvons écrire le fichier `lex` suivant:

```
1 %{
2     #include <ctype.h>
3     %}
4
5     %%
6     .|\n { putchar(toupper(yytext[0])); }
7     %%
8
9     int main() { return yylex(); }
10    int yywrap() { return 1; }
```

Ici, tous les caractères (c'est-à-dire `'.'` et `'\n'`) sont convertis par la fonction C `toupper`. Noter que le résultat reconnu par l'expression régulière à gauche est toujours rangé dans la variable `yytext`. Comme notre expression régulière ne reconnaît qu'un seul caractère, celui-ci est dans `yytext[0]`.

### Note:


La solution utilisée ici n'est pas optimale, puisque nous convertissons **tous** les caractères, y compris ce qui ne sont pas des minuscules. Une meilleure solution consiste à remplacer la ligne 6 du code précédent par 

```
[a-z] { putchar(toupper(yytext[0])); }
.\n { ECHO; } /* Règle inutile, car implicite. */
```

## 2 Commande wc

L'écriture de la commande `wc` est assez simple. Il suffit de déclarer 3 compteurs globaux et de reconnaître les séquences suivantes:

1. les **mots**: c'est une suite caractères qui ne sont pas des séparateurs. L'expression régulière qui les reconnaît: `[^ \t\n]`.  
Cette expression correspond à l'ensemble de tous les caractères (à cause des crochets `[...]`) (à cause du caractère `^`) sauf l'espace `' '`, tabulation (`'\t'`) et newline (`'\n'`).
2. les **lignes**, c'est-à-dire `'\n'`.
3. les **autres caractères**, c'est-à-dire `'.'`

On obtient donc 

```
%{
    int chars = 0;
    int words = 0;
    int lines = 0;
}%

%%
[^\n\t ]+ { words += 1; chars += strlen(yytext); }
\n        { chars += 1; lines += 1; }
.         { chars += 1; }
%%

int main() {
    yylex();
    printf("%3d %3d %3d\n", lines, words, chars);
}

int yywrap() { return 1; }
```

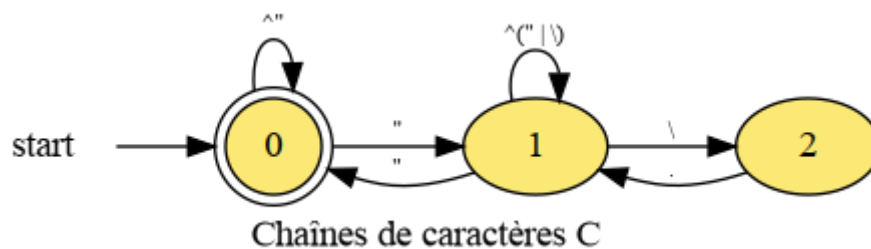
## 3 Chaînes de caractères C

### 3.1 Version 1: expressions régulières

#### Remarque:

On ne s'occupe pas ici des commentaires qui pourraient rendre notre programme incorrect si un commentaire contenait des guillemets.

L'automate de reconnaissance des chaînes de caractères de C est représenté ci-dessous:



L'expression régulière pour reconnaître une chaîne peut donc être: `"([^\\""]|\\.)*"`. L'écriture est ici un peu compliquée, car il faut mettre un `\"` devant les caractères `\"` et `"`. Ce que l'on a ici se lit donc comme:

1. un `"` de début de chaîne
2. puis une suite (éventuellement vide) de
  - 1 caractère qui n'est ni `\"` ni `"` OU
  - 2 caractères dont le premier est un `\"`
3. un `"` final

Lorsqu'une chaîne sera reconnue par cette expression régulière, la variable `yytext` contiendra donc la chaîne reconnue que l'on peut mettre en majuscules caractère par caractère (en faisant attention de ne pas mettre en majuscules le caractère qui suit un `\"`).

Quant aux caractères qui sont en dehors des chaînes, ils sont recopiés tels quels (avec la macro `ECHO` de lex) (voir [version1](#) 📄):

```
\"([^\\"\\]|\\.)*\"    { for (char *s = yytext; *s; s++) {
                        putchar(toupper(*s));
                        if (*s == '\\') putchar(*++s);
                    }
                }
.\|\\n                { ECHO; }
```

## 3.2 Version 2: Utilisation de contextes gauches

On déclare ici un nouveau contexte gauche **STRING**. On a donc les cas suivants:

- si on rencontre **'** dans le contexte **INITIAL**, on doit armer le contexte **STRING**
- si on est dans le contexte **STRING** :
  - si on rencontre un guillemet (c'est obligatoirement le guillemet fermant), on peut donc réarmer le contexte **INITIAL**
  - si on rencontre un caractère **'\\'** suivi d'un caractère quelconque, il faut les recopier tels quels (ils sont dans **yytext**).
  - tous les autres caractères sont affichés en majuscule.

Les règles sont pour les chaînes sont donc (voir [version 2](#) 📄):

```
\"                { putchar(''); BEGIN STRING; }
<STRING>\"        { putchar(''); BEGIN INITIAL; }
<STRING>\\.        { printf("%s", yytext); }
<STRING>.         { putchar(toupper(*yytext)); }

.\|\\n            { ECHO; }
```

## 4 Commentaires C

### 4.1 Première version

Il suffit de combiner ici le code de l'exercice précédent sur les chaînes et le code vu en cours. On obtient la [première version](#) 📄:

```
%X COMMENT
%%

/* strings */
\"([^\"]|\\.)*\"    { printf(\"%s\", yytext); }

/* Commentaires classiques */
\"/*\"                { BEGIN COMMENT; }
<COMMENT>\"*/\"        { putchar(' '); BEGIN INITIAL; }
<COMMENT>.\|\\n      { }

/* Commentaires C++ */
\"//\".*$              { }

/* Les autres caractères */
.\|\\n                ECHO;

%%

int main() { return yylex(); }
int yywrap() { return 1; }
```

## 4.2 Deuxième version

Pour traiter les commentaires emboîtés, il suffit d'avoir une variable `level` qui compte le niveau d'imbrication des commentaires. À chaque fois que l'on rencontre `/*` on incrémente ce compteur et quand on rencontre `*/`, on le décrémente. Si `level` est à zéro, c'est qu'on est en dehors d'un commentaire et les caractères doivent donc être affichés. Il faut pour cela, bien sûr, repasser dans le mode `INITIAL`.

Le code de la [seconde version](#) est donc:

```
\"/*\"                { level += 1; BEGIN COMMENT; }
<COMMENT>\"*/\"        { if (--level == 0) { putchar(' '); BEGIN INITIAL; } }
<COMMENT>\"/*\"        { level += 1; }
<COMMENT>.\|\\n      { }
```

Remarque:

On peut factoriser les deux lignes qui traitent l'ouverture des commentaires. En effet, il faut traiter l'ouverture dans les deux contextes (`INITIAL` et `COMMENT`). Comme le code est semblable, on peut écrire (au prix d'un passage inutile dans le contexte `COMMENT` si on y est déjà):

```
<INITIAL,COMMENT>\"/*\"    { level += 1; BEGIN COMMENT; }
```

## 4.3 Troisième version

Ici, on se contente de rajouter une ligne pour reconnaître la fin de fichier à l'intérieur d'un commentaire (c'est à dire dans le contexte gauche `COMMENT`). La règle s'écrit simplement:

```
<COMMENT><<EOF>>      { fprintf(stderr, \"EOF while reading a comment\\n\"); exit(1); }
```

La version finale du programme est disponible dans le fichier [uncomment3.1](#).

# 5 Reconnaissance de nombres

Pour les nombres entiers, l'expression régulière permettant de les reconnaître est:

```
[+-]?[0-9]+
```

c'est-à-dire:

- un signe (facultatif)
- suivi d'au moins un chiffre

Pour les nombres réels sans exposants, nous pouvons écrire:

```
[+-]?([0-9]+\.[0-9]*)|(\.[0-9]+)
```

L'alternative permet de reconnaître

- les nombres qui ne commencent pas par un point: `[0-9]+\.[0-9]*` Noter que le point et les chiffres après ce dernier sont facultatifs, afin de toujours reconnaître les entiers;
- les nombres inférieurs à 1 qui ne comportent pas de 0 devant le point: `\.[0-9]+`

Pour l'exposant, l'expression est `[eE][+-]?[0-9]+` (la lettre `e` ou `E` suivie d'un entier signé ou non)

La version finale (nombre avec exposant éventuel) est donc:

```
[+-]?((([0-9]+\.[0-9]*)|(\.[0-9]+))([eE][+-]?[0-9]+)?
```

Comme cette expression est un peu compliquée, on peut en simplifier l'écriture en écrivant les règles lex suivantes:

```
signe      [+ -]
digit      [0-9]
nombre     {digit}+
exposant   [eE]{signe}?{digit}+

reel       {signe}?(({nombre}\.[0-9]{digit}*)|(\.[0-9]{nombre}){exposant})?
```

Quant aux règles *lex* à appliquer, celles-ci sont:


```
%%
{reel}      printf("<NOMBRE '%s'>", yytext);
.|\\n      ECHO;
%%
```

[Code complet](#) 

## 6 Évaluation de nombres C

### 6.1 Une première version

Pour cet exercice, on peut bien sûr se contenter de reconnaître les trois syntaxes de nombres possibles et se reposer sur une fonction capable ensuite de calculer la valeur d'une chaîne dans la base donnée (la fonction Posix `strtol` par exemple peut être utilisée pour cela).

On obtiendrait :

```
%%
0x[0-9a-fA-F]+      { printf("Hexadecimal: %d\n", strtol(yytext+2, NULL, 16)); }
0[0-7]*              { printf("Octal: %d\n", strtol(yytext, NULL, 8)); }
[1-9][0-9]*          { printf("Decimal: %d\n", strtol(yytext, NULL, 10)); }

/* Les autres caractères */
.|\\n                { ; }
%%
```

Dans cette version:

- les chiffres des nombre hexadécimaux sont soit des chiffres soit des lettres comprises entre `a` et `f` (en majuscule ou en minuscule)
- les chiffres de nombre octaux sont compris entre `0` et `7`

- les chiffres décimaux ont compris entre 0 et 9, sauf le premier chiffre qui ne peut être 0.

## 6.2 Version utilisant des contextes gauches


La version précédente nécessite que l'on reconnaisse le nombre entièrement, que l'on stocke donc ses digits dans une chaîne et que l'on convertisse ensuite la chaîne reconnue.

L'utilisation de contextes gauches permet de calculer la valeur du nombre au fur et à mesure. On utilise pour cela 3 contextes gauches (un par base).

On a donc une variable `value` qui contient toujours la valeur courante du nombre en cours de lecture. Lorsqu'on rencontre un nouveau *digit*, on remplace `value` par `value*base+val(digit)` ou `val` permet de trouver la valeur entière d'un *digit*.

### Note:

Une version de cette méthode de calcul d'un nombre (pour la base 10 seulement) est donnée dans le cours 4-1 de *Programmation Procédurale* (fonction `atoi`).

Une version possible :

```
%{
    int value = 0;

    void afficher(char *type, int n) {
        printf("Reconnaissance d'un nombre en %s: %d(10) %o(8) %x(16)\n",
            type, n, n, n);
    }
}%

%x      OCTAL DEC HEXA

%%

"0x"      { value = 0; BEGIN HEXA; }
<HEXA>[0-9] { value = 16 * value + *yytext - '0'; }
<HEXA>[a-f] { value = 16 * value + *yytext - 'a' + 10; }
<HEXA>[A-F] { value = 16 * value + *yytext - 'A' + 10; }
<HEXA>[^0-9a-fA-F] { afficher("hexadecimal", value); BEGIN INITIAL; }

"0"      { value = 0; BEGIN OCTAL; }
<OCTAL>[0-7] { value = 8 * value + *yytext - '0'; }
<OCTAL>[^0-7] { afficher("octal", value); BEGIN INITIAL; }

[1-9]      { value = *yytext - '0'; BEGIN DEC; }
<DEC>[0-9] { value = 10 * value + *yytext - '0'; }
<DEC>[^0-9] { afficher("décimal", value); BEGIN INITIAL; }

/* Les autres caractères */
.|\\n      { ; }

%%
int main() { return yylex(); }
int yywrap() { return 1; }
```

## 7 Calculatrice

### 7.1 Version de base

Le corrigé de la calculatrice est simple. Il est donné ci-dessous

```

/*
 * calc.l          -- Un lexical simple pour une calculatrice avec variables
 */

%{
#include <stdio.h>
#include "calc.h"
%}

%option noyywrap

%%

[0-9]+      { yy1val.val = atoi(yytext); return NUMBER; }
[a-zA-Z_][a-zA-Z0-9_]+ { yy1val.var = strdup(yytext); return IDENT; }

"+"        { return PLUS; }
"-"        { return MINUS; }
"*"        { return MULT; }
"/"        { return DIV; }
"("        { return OPEN; }
")"        { return CLOSE; }
"="        { return EQUAL; }
\n         { return EOL; }

[ \t]      { }
.          { fprintf(stderr, "Unexpected character %c (%d)\n",
                        *yytext, *yytext); }

%%

```

**Notes:**

- Il faut inclure le fichier `"calc.h"` qui contient les définitions de constantes pour les tokens ( `PLUS` , `MINUS` , ...), ainsi que la déclaration de la variable partagée `yy1val` .
- Les seuls cas où il faut faire attention sont les identificateurs et les entiers: Dans ce cas, il faut remplir le bon champ de l'union `yy1val` avant de renvoyer le type token rencontré.
- Pour compiler (le source lex étant dans le fichier `calc.lex.c`

```

$ lex -o calc.lex.c calc.l
$ gcc -Wall -std=gnu99 calc.c calc.lex.c -o calc

```

**Attention:**

Lorsqu'on voit un identificateur, il faut penser à faire une copie "fraîche" de la valeur qui est dans `yytext` . Pour cela, on utilise la fonction `strdup` . Si on omet de le faire, l'analyseur syntaxique pointera toujours directement `yytext` , qui est une variable qui peut être modifiée ultérieurement par l'analyseur lexical. Cela veut dire que la lecture d'une nouvelle ligne pourra modifier les valeurs qui ont été rangées dans l'arbre d'analyse syntaxique (Pas bon du tout 😊).

**Analyseur lexical de la calculatrice** 📄

## 7.2 Version étendue

La version étendue de la calculatrice n'est pas compliquée. Il suffit de rajouter les règles suivantes:

```
%%
"quit"          { return EOF; }
"#".*$         { }
"^!".*$        { system(yytext+1);}
0x[0-9a-fA-F]+ { yylval.val = strtol(yytext+2, NULL, 16); return NUMBER;}
0[0-7]*        { yylval.val = strtol(yytext, NULL, 8); return NUMBER;}
...
```

1. La première règle simule l'arrivée d'une fin de fichier dès qu'on voit le mot **quit**
2. La seconde règle permet de sauter les caractères qui suivent un **'#'** jusqu'à la fin de la ligne. Le texte sauté est donc un commentaire.
3. la troisième règle permet de passer le texte qui suit le **'!'** en début de ligne à la primitive POSIX **system**.
4. Les deux règles suivantes correspondent aux règles que l'on avait vu dans l'exercice précédent.

#### Analyseur lexical de la calculatrice étendue

## 8 Espaces et langages de programmation

Avant de commencer, il faut savoir que l'expression `- - a` est valide en C: elle correspond à la valeur `(- (- a))`, c'est à dire `a`. Par contre, l'expression `--a` sans espace entre les deux tirets correspond à la pré-décrément de la variable `a`.

Par conséquent,

- l'expression `- -1` est valide et correspond à l'entier `+1`, alors que
- l'expression `--1` n'est pas valide puisque on ne peut pas décrémenter une constante.

La table demandée est donc:

Expression	Valeur	x	y	Valeur sans espace	x	y
<code>x - 1</code>	0	1	2	0	1	2
<code>x - -1</code>	-2	1	2	erreur		
<code>x - - 1</code>	2	1	2	erreur		
<code>x - - - 1</code>	0	1	2	0	0	2
<code>x - y</code>	-1	1	2	-1	1	2
<code>x - -y</code>	3	1	2	erreur		
<code>x - - y</code>	3	1	2	erreur		
<code>x ---y</code>	-1	0	2	-1	0	2
<code>-y</code>	-2	1	2	-2	1	2
<code>- -y</code>	2	1	2	-2	1	2
<code>-y</code>	1	1	1	1	1	1
<code>x - - - - y</code>	0	0	1	erreur		
<code>x-- + --y</code>	2	0	1	2	0	1

Comme on peut le voir, les espaces peuvent donc être **significatifs**, même si un analyseur lexical ne renvoie en général pas de lexème correspondant à un espace en tant que tel!