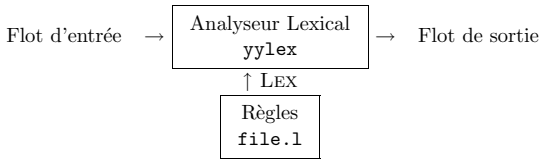
Petit manuel de Lex et Yacc

Mathieu Jaume François Schoubben

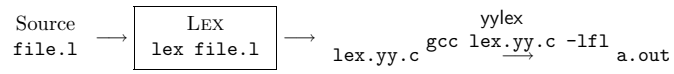
1 Lex – Générateur d'analyseur lexical

Lex est un outil qui permet d'engendrer un analyseur lexical (lexer), c'est à dire un programme qui, étant donné un flot d'entrée, reconnaît des expressions régulières et partitionne le flot d'entrée. Les expressions régulières reconnues sont celles spécifiées par l'utilisateur.

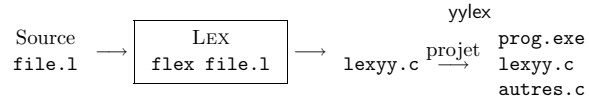


⇝ Langage C

- Linux



- BC++



Le fichier engendré contient la fonction :

```
function yylex :  Integer;
```

⇝ Utilisation

`file.l` contient la description de l'analyseur lexical qui doit être engendré sous la forme de règles. Une règle est la donnée d'une expression régulière e et d'une action (celle qui sera exécutée lorsqu'une expression "satisfaisant" e sera reconnue). Par exemple, avec la règle :

```
"integer" {printf("found keyword INT");}
```

à chaque fois que la chaîne `integer` sera reconnue, le message `found keyword INT` sera écrit sur la sortie standard. Une expression régulière spécifie un ensemble de chaînes. Elle est exprimée à partir de caractères et d'opérateurs sur ces caractères.

Caractères spéciaux

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

Pour utiliser ces caractères comme "caractères ordinaires", il faut les protéger en les plaçant dans une chaîne entourée de double-quotes (") ou en les plaçant après un `\`. Les caractères `\n`, `\t`, et `\b` correspondent comme en C au saut de ligne, à la tabulation et au *backspace*.

Spécification d'expressions régulières

- `"` Une chaîne de caractères entourée de double-quotes représente la chaîne elle-même.

`"abc"` spécifie la chaîne `abc`

- `[ ]` Une chaîne de caractères entre crochets représente un de ses éléments. Dans ce contexte, `-` indique un intervalle et `^` désigne l'exclusion.

`[xyz]` :	`x`, `y` ou `z`
`[a-zA-Z]` :	toutes les lettres minuscules et majuscules
`[^0-9]` :	tous les caractères sauf les chiffres

- `.` : tout caractère sauf `\n`
- `|` : Opérateur d'alternance

`x|y|z` :	équivaut à `[xyz]` (`x`, `y` ou `z`)
`[a-z]|[A-Z]` :	toutes les lettres minuscules et majuscules

- `* et +` : Opérateur de répétition (`*` : zéro ou plusieurs fois, `+` : une ou plusieurs fois)

`(x|y)*` :	chaîne de longueur positive ou nulle constituée des caractères `x` ou `y`
`[a-z]+` :	chaîne de longueur strict. positive constituée de lettres minuscules

- `?` : Opérateur d'occurrence zéro ou une fois

`ab?c` :	chaîne `abc` ou chaîne `ac`

- `/` : Condition de reconnaissance

`ab/cd`	chaîne `ab` seulement si elle est suivie de la chaîne `cd`

- `$ et ^` : Début de ligne et fin de ligne

`ab$`	chaîne `ab` en fin de ligne
`^ab`	chaîne `ab` seulement si elle est en début de ligne (après `\n` ou `$`)

- `{ }` : Opérateur de répétition bornée – Définition

`a{1,5}` :	chaîne de longueur comprise entre 1 et 5 constituée du caractère `a`
`a{2,}` :	chaîne de longueur supérieure ou égale à 2 constituée du caractère `a`
`a{2,2}` :	chaîne de longueur 2 constituée du caractère `a`
`{digit}` :	chaîne prédéfinie de nom `digit`

Exercice 1.1

1. *Comment spécifier la chaîne* `xyz++` *?*
2. *Quelle est la différence entre* `(ab|cd)` *et* `a(b|c)d` *?*
3. *Comment spécifier les chaînes représentant les entiers relatifs sachant que :*
   - *0 s'écrit de manière unique (−0 n'est pas correct)*
   - *les zéros non significatifs ne figurent pas dans la représentation des entiers*
4. *Comment spécifier tout caractère sauf une lettre majuscule ou une fin de ligne ?*

▶

1. `"xyz++"` ou `xyz\+\+`
2. `(ab|cd)` spécifie les chaînes `ab` et `cd` tandis que `a(b|c)d` spécifie les chaînes `abd` et `acd`.
3. `(-?[1-9][0-9]*)|0`
4. `[^A-Z\n]`

◀

Source Lex La forme générale d'un source Lex est :

{ définitions }
% %
{ règles }
% %
{ fonctions }

La spécification des **définitions**

- permet d'associer un nom à une spécification :

```
DIGIT [0-9]
...
{DIGIT}+"."{DIGIT}* ;
```

- permet de déclarer des "conditions de déclenchement"

En cas de conflit, Lex choisit toujours la règle qui produit le plus long lexème.

`prog` action1
`program` action2
La deuxième règle sera choisie.

Si plusieurs règles donnent des lexèmes de mêmes longueurs, Lex choisit la première.

`prog` action1
`[a-z]+` action2
La première règle sera choisie.

Si aucune règle ne correspond au flot d'entrée, Lex choisit sa règle par défaut implicite :

`.|\n {ECHO}` recopie le flot d'entrée sur le flot de sortie

Une **action** est un bloc d'instructions qui est exécuté lorsque la chaîne de caractères lue correspond à la chaîne spécifiée avant l'action. Ce bloc d'instructions peut utiliser certaines variables prédéfinies comme :

- `yytext` : chaîne reconnue correspondant à l'expression régulière
- `yylen` : longueur de `yytext`.
  Par exemple une règle qui permet de compter le nombre de mots et le nombre de caractères dans les mots peut s'écrire :

```
[a-zA-Z]+ { words++;chars=chars+yylen; }
```

- `yyin` : fichier d'entrée (`FILE *`) – en C
- `yyout` : fichier de sortie (`FILE *`) – en C

et peut contenir certaines macros prédéfinies :

- comme `|` qui indique que l'action à exécuter est la même que celle de la règle suivante
- comme `ECHO`

  `[a-z]+ {printf("%s",yytext);}` équivaut à `[a-z]+ {ECHO;}`

- comme `BEGIN` en C
- ou comme `REJECT` en C qui permet d'envisager la "deuxième meilleure" règle pour laquelle le flot d'entrée (ou un de ses préfixes) est reconnu

```
%%
frob {special();REJECT;}
[^ \t\n]+ {++word_count;}
```

compte le nombre de mots du flot d'entrée et
exécute `special` à chaque fois que la chaîne `frob` apparaît

Pour ceux qui utilisent C. Les actions peuvent invoquer certaines directives :

- `yymore()` indique que lors de la prochaine application d'une règle, la chaîne reconnue doit être concaténée à `yytext` qui ne doit donc pas écraser `yytext`

```
%%
mega- {ECHO;yymore();}
kludge {ECHO;}
```

Exemple : `mega-kludge` → `mega-mega-kludge`

- `yyless(n)` indique que les `n` derniers caractères de `yytext` doivent être pris en compte lors de la prochaine application d'une règle

```
%%
foobar {ECHO;yyless(3);}
[a-z]+ {ECHO;}
```

Exemple : `foobar` → `foobarbar`

Conditions de déclenchement On peut conditionner l'application d'une règle à l'aide de "conditions de déclenchement" ... si on préfixe l'expression régulière d'une règle par `<sc>`, alors cette règle ne sera appliquée que si l'on se trouve dans l'état `sc`. Les conditions de déclenchement sont déclarées dans la partie définitions du source (elles peuvent être inclusives (`%s`) ou exclusives (`%x`)). Une condition de déclenchement est activée à l'aide de la macro `BEGIN` en C et reste active jusqu'à la prochaine macro `BEGIN` ou `start`. Si la condition est inclusive alors les règles sans conditions restent actives sinon elles ne sont pas considérées. `INITIAL` en C est la condition par défaut.

- `<sc>[a-z]` : lettre minuscule seulement si on se trouve dans l'état `sc`
- `<sc1,sc2,sc3>[a-z]` : lettre minuscule seulement si on se trouve dans des états `sc1`, `sc1`, ou `sc3`.
- `<*>[a-z]` : lettre minuscule quel que soit l'état dans lequel on se trouve (la règle par défaut est implicitement préfixée par `<*>`)

Exemple en C : `123.456` vu comme `123`, `.` et `456` ou comme le réel `123.456` si précédé de `expect-floats`.

```
%{ #include <math.h> %}
%s expect
%%
"expect-floats"           {BEGIN(expect);}
<expect>[0-9]+"."[0-9]+   {printf("float=%f\n",atof(yytext));}
<expect>\n                {BEGIN(INITIAL);}
[0-9]+                    {printf("int=%d\n",atoi(yytext));}
"."                       {printf("dot\n");}
```

Exercice 1.2 *Ecrire un programme qui supprime toutes les occurrences de la chaîne* `zap me` *d'un fichier.*

▶

```
%%
  "zap me" {}
```

◀

Exercice 1.3 *Quel est l'effet du programme :*


```
%%
a      |
ab     |
abc    |
abcd   {ECHO;REJECT;}
.|\n
```

lorsque le flot d'entrée est `abcd` *?*

▶ `abcdabcaba` ◀

Exercice 1.4 *Ecrire un programme qui compte le nombre de lignes et le nombre de caractères d'un fichier.*

▶

```
  int num_lines = 0; num_chars=0;
%%
  \n   {++num_lines; ++num_chars;}
  .    {++numchars;}
%%
  main()
   { yylex();
     printf("# of lines = %d, # of chars =%d \n",
                          num_lines,num_chars);}
```

◀

Exercice 1.5 *Ecrire un programme qui remplace toute suite de caractères blancs ou de tabulations par un unique caractère blanc si cette suite n'apparaît pas en fin de ligne et qui la supprime sinon.*

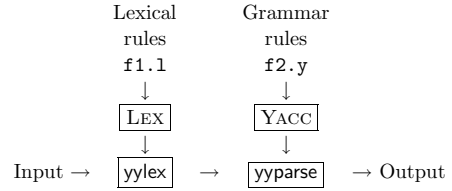
▶

```
%%
  [ \t]+   {putchar(' ');}
  [ \t]+$  {}
```

◀

2 YACC

Une fois l'analyse lexicale effectuée, l'analyse syntaxique permet, étant donnée une grammaire G, de vérifier qu'un "texte" est conforme à G et dans ce cas permet de construire l'arbre syntaxique correspondant. YACC est un outil qui étant donnée une spécification (*grosso modo* la description d'une grammaire) engendre un analyseur syntaxique (parser) pour cette grammaire.



La **structure générale d'une spécification** YACC est la suivante :

déclarations
%%
règles (grammaire)
%%
fonctions

Une **règle** s'écrit :

A : BODY ;

où A est le nom d'un non-terminal et où BODY est une suite (éventuellement vide) de noms (de non-terminaux ou de terminaux, les *tokens*) et de littéraux (i.e., caractères entre quotes). Lorsqu'il existe plusieurs règles de production pour un même non-terminal, l'opérateur | peut être utilisé pour séparer le corps de chaque règle. On peut par exemple écrire :

exp : NUM | exp '+' exp ;

La partie **déclarations** contient les déclarations "C" (entre `%{` et `%}`) ainsi que la déclaration des noms de *tokens* :

%token name1 name2 ...

Les noms qui n'ont pas été déclarés en tant que "nom de *token*" sont considérés comme des non-terminaux. Lorsque les *tokens* correspondent à des opérateurs pour lesquels on souhaite spécifier une propriété d'associativité, on utilise `left` et `right` à la place de `token`. Par exemple, on écrira :

```
%left '+' '-' /* addition et soustraction associatives a gauche */
%right '^' /* exponentiation associative a droite */
```


Il est aussi possible d'utiliser `%nonassoc` pour déclarer un *token*. Si l'on souhaite de plus spécifier la précédence entre opérateurs, on écrit la déclaration des *tokens* dans l'ordre inverse des priorités (sur l'exemple ci-dessus, l'exponentiation a une plus grande précédence que l'addition et la soustraction). On peut modifier la précédence d'un opérateur dans une règle. Par exemple, si l'on souhaite décrire le "moins unaire" aussi désigné par `'-'` comme un opérateur ayant même précédence qu'un *token* de nom `NEG`, on écrira dans la règle :

```
mexp :  '-' exp %prec NEG ...
```

afin de ne pas associer au "moins unaire" la même précédence que le "moins binaire".

Chaque non-terminal doit apparaître au moins une fois comme partie gauche d'une règle. Le symbole (non-terminal) de départ de la grammaire spécifiée peut être désigné explicitement dans la partie déclaration :

%start symbol

A défaut, le symbole de départ sera le non-terminal apparaissant dans la partie gauche de la première règle.

On peut associer à chaque terminal et à chaque non-terminal une valeur. Cette **valeur "sémantique"** n'est pas nécessairement utile du point de vue de la vérification de la correction syntaxique mais pourra s'avérer utile par la suite. Par exemple, même si la correction syntaxique d'une expression arithmétique ne dépend pas du résultat de son évaluation, il peut être avantageux d'évaluer les sous-expressions qui la composent tout en s'assurant que celles-ci sont correctes d'un point de vue syntaxique.

L'analyse syntaxique s'effectue après l'**analyse lexicale** qui permet de reconnaitre les *tokens* via des valeurs numériques. Si 0 est la valeur numérique retournée, le *token* correspond à la fin du fichier. Généralement, on ne considère pas ces valeurs numériques de manière explicite. Concrètement, YACC fait appel à `yylex` à chaque fois qu'un *token* est nécessaire à la suite de l'analyse syntaxique ; LEX "rend la main" à YACC dès qu'une unité lexicale est trouvée en retournant le lexème (i.e., le *token* pour YACC).

Les *tokens* sont éventuellement associés à des valeurs via la variable `yylval` qui est de type `YYSTYPE`, type qui peut être simplement défini dans la partie "déclarations" via la macro "`#define`". Pour les caractères non-reconnus, cette valeur correspond au code ASCII du caractère. Par défaut, `YYSTYPE` correspond au type `int`. Il peut être utile de pouvoir considérer des *tokens* dont les valeurs appartiennent à des types différents. Pour ce faire, on spécifie dans la partie "déclarations" l'"union" des types qui seront considérés. Par exemple, si l'on souhaite associer aux *tokens* des valeurs de type "entier" et "chaîne de caractères", on écrira :

```
%union {  int ival;
          char *str;
       }
```

On peut aussi, de manière équivalente, (re)définir dans la "partie C", le type `YYSTYPE` comme suit :

```
typedef union {  int ival;
                 char *str;
              }  YYSTYPE;
```

C'est d'ailleurs cette déclaration de type qui figurera dans le fichier `y.tab.h` (`y_tab.h` pour MS-DOS) qu'engendre YACC.

Une fois le type `YYSTYPE` spécifié, il faut indiquer quel membre de l'union utiliser pour les divers *tokens* et non-terminaux. On utilise pour cela la construction `<name>` dans la partie "déclaration". Par exemple, on écrira :

```
%token <ival>NUMBER
%token <str>IDENT
```

Pour les non-terminaux, on écrira :

```
%type <ival>exp
```

Puisque l'analyse syntaxique utilise le "résultat" de l'**analyse lexicale**, il faut que le fichier LEX utilisé dispose de certaines informations, notamment les noms de *tokens* ainsi que les types des valeurs qui seront associées aux *tokens*. Aussi, dans la partie "déclarations" du fichier LEX, on écrira :

```
#include "y.tab.h"
```

ce qui permettra de disposer du type `YYSTYPE` mais aussi des noms de *tokens* puisque pour chaque *token* de nom `name`, YACC fait figurer dans le fichier `y.tab.h` la directive :

`#define name` valeur_numérique_désignant_le_*token*

Enfin, les règles d'analyse lexicale doivent être écrites de manière à ce que YACC dispose de l'information relative au *token* reconnu et à sa valeur. On écrira par exemple :

```
[0-9]+    {yylval.ival=atoi(yytext); return NUMBER;}
[a-zA-Z]+ {yylval.str=strdup(yytext); return IDENT;}
```

A chaque règle peut être associée une **action** qui correspond à un bloc d'instructions C et qui sera exécuté lorsque le *parser* aura reconnu la structure spécifiée par la règle. Ces actions peuvent retourner une valeur et utiliser les valeurs retournées par les actions précédentes. Par exemple, on peut écrire la règle :

```
A : '(' B ')' { hello(1,"abc"); } ;
```

Pour retourner une valeur, on utilise le symbole $$: par exemple une action qui consiste seulement à retourner 1 s'écrit

```
{ $$ = 1; }
```

Pour utiliser les valeurs retournées par les actions précédentes (ou par l'analyseur lexical), une action peut invoquer les pseudo-variables $1, $2, ... qui correspondent aux valeurs retournées par les actions correspondant aux éléments de la partie droite de la règle. Par exemple, étant donnée la règle :

```
A : B C D { f($1,$2,$3) ;};
```

les valeurs de $1, $2 et $3 correspondent respectivement aux valeurs retournées par les actions spécifiées dans les règles utilisées reconnaissant `B`, `C` et `D`. L'exemple, plus concret, de l'évaluation d'une expression parenthésée s'écrit :


```
expr :  '(' expr ')' { $$ = $2 ;};
```

Par défaut, si aucune valeur n'est indiquée comme valeur de retour de l'action, c'est $1 qui est retournée. YACC permet d'exécuter des actions avant la fin de l'application d'une règle. Il est donc possible d'indiquer une action entre deux composants de la partie droite d'une règle. Par exemple, la règle :

```
A :  B
        { $$=1 ; }
     C
        { x=$2; y=$3; }
  ;
```

aura pour effet d'affecter 1 à `x` et la valeur retournée par la règle utilisée pour reconnaitre `C` à `y` puisqu'elle est considérée par YACC comme :

```
$ACT :  /* empty */
                        { $$=1 ; }
     ;
  A :  B $ACT C
                        { x=$2; y=$3; }
     ;
```

On peut spécifier le "type d'une valeur" dans une règle (ce qui peut être utile par exemple lorsqu'un *token* "mono-caractère" n'a pas été déclaré). Par exemple, on peut écrire :

```
rule :  aaa bbb { func($<ival>1); $<str>$="hello"; } ;
```

Lors de l'exécution de la fonction `yyparse`, si une **erreur de syntaxe** est détectée, la fonction `yyerror` est appelée afin qu'un message s'affiche. En l'absence de règle permettant de continuer l'analyse syntaxique après la détection d'une erreur, la fonction `yyparse` s'arrête après l'exécution de `yyerror` en retournant une valeur non nulle. Toutefois, il est souvent souhaitable de continuer l'analyse syntaxique afin de détecter d'autres erreurs éventuelles. Pour cela, il peut être préférable de reprendre la vérification "au début de la phrase suivante". Pour ce faire, YACC dispose d'un *token* de nom `error` qui peut être utilisé dans une règle et d'un mécanisme permettant de reprendre la vérification à partir d'un *token* donné. Par exemple, la règle :

```
stat :  error ';' ;
```

permet lorsqu'une erreur est détectée de reprendre la vérification à partir du *token* qui suit le prochain `';'`.

Enfin, signalons que les actions associées aux règles peuvent invoquer des "macros" comme `YYACCEPT` (resp. `YYABORT`) qui termine la fonction `yyparse` en retournant 0 (resp. 1).

Exercice 2.1 *Spécifier l'analyse lexicale et l'analyse syntaxique (en utilisant* LEX *et* YACC*) des expressions arithmétiques en représentation postfixe sur les "flottants". Durant l'analyse syntaxique les expressions seront évaluées. Modifier ensuite les règles pour traiter les expressions arithmétiques en représentation infixe.*

```
%{
#define YYSTYPE double
#include <math.h>
#include <stdio.h>
%}
%token NUM
%%
input : /* empty */ | input line ;
line  : '\n' | exp '\n' {printf("%f",$1)} ;
exp   : NUM         { $$=$1; }           |
        exp exp '+' { $$=$1+$2; } ;      |
        exp exp '-' { $$=$1-$2; } ;      |
        exp exp '*' { $$=$1*$2; } ;      |
        exp exp '/' { if ($2==0)
                        { yyerror("division by 0");
                          YYABORT; }
                      else $$=$1/$2; } ; |
        exp exp '^' { $$=pow($1,$2); } ; |
        exp 'n'     { $$=-$1; }   ; /* unary minus */
%%
main()
{ yyparse(); }
void yyerror(char *s)
{ fprintf(stderr,"%s \n",s); }
```

Fichier Lex

```
%{ #include "y.tab.h" %}
%%
" "             ;
[0-9]+"."[0-9]+ { yylval=atof(yytext); return NUM; }
%%
```

Représentation infixe des expressions.

```
%{
#define YYSTYPE double
#include <math.h>
%}
%token NUM
%left '-' '+'
%left '*' '/'
%left NEG /* unary minus */
%right '^'
%%
input : /* empty */ | input line ;
line  : '\n' | exp '\n' {printf("%f",$1)} ;
exp   : NUM         { $$=$1; }         |
        exp '+' exp { $$=$1+$3; } ;    |
        exp '-' exp { $$=$1-$3; } ;    |
        exp '*' exp { $$=$1*$3; } ;    |
        exp '/' exp { if ($3==0)
                        { yyerror("division by 0");
                          YYABORT; }
```



```
                      else $$=$1/$3; };|
        '-' exp %prec NEG { $$=-$2; }  |  /* unary minus */
        exp '^' exp { $$=pow($1,$3) ; }|
        '(' exp ')' { $$=$2; } ;
%%
...
```

3 Références

Plusieurs sites web peuvent vous être utiles :

- `http://www.linux-france.org/article/devl/lexyacc/minimanlexyacc.html` (mini-guide en français)
- `http://pltplp.net/lex-yacc/` (en français)
- `http://www-edu.gel.usherb.ca/nkoj01/gei443/exemples.html` (en français)
- `http://www.linuxdoc.org/HOWTO/Lex-YACC-HOWTO.html`(en anglais)
- `http://www.uman.com/lexyacc.shtml)` (pour télécharger les programmes et trouver la doc).
- avec les outils gnu (linux ou cygwin) : man flex et info bison

Le livre "*Lex & Yacc*" des éditions O'reilly (J Levine, T Mason & D Brown) est une référence dans le domaine.