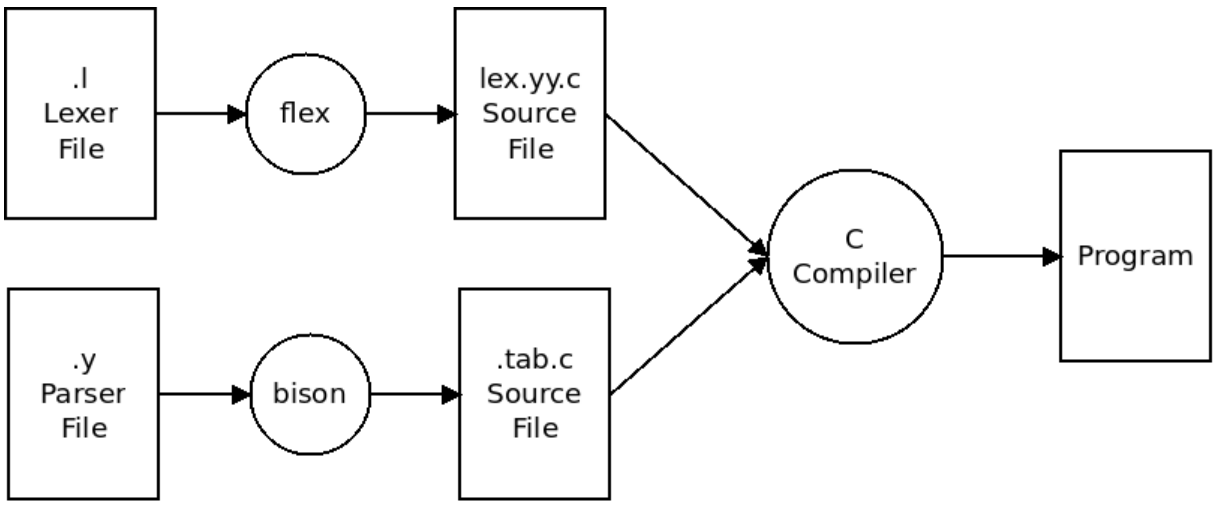
Fascicule de TP Flex

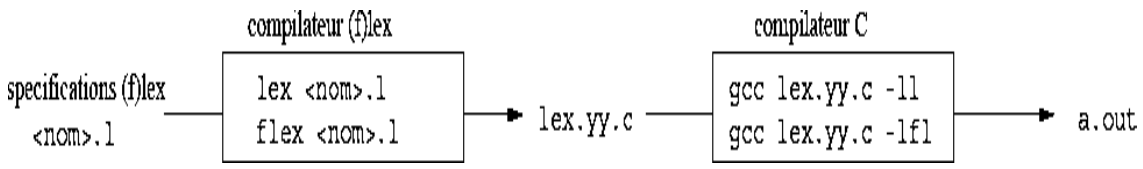
Installation

```
sudo apt-get update

sudo apt-get upgrade

sudo apt-get install flex bison
```

Introduction :

Lex est un analyseur lexical basé sur les expressions régulières comme le programme grep (voir aussi egrep et fgrep). Le programme grep vous permet de retrouver une expression régulière dans un texte alors que lex vous permet de créer un programme qui pour chaque expression régulière retrouvée pourra exécuter les commandes (écrites en C) que vous aurez prédéfinies. Vous pouvez par exemple, très facilement, faire un programme qui reprend un texte et mets en minuscule tous les mots qui sont en majuscule. Ou bien de mettre en majuscule toutes les premières lettres d'un mot qui est précédé d'un point. etc etc....





Le format :

Le format d'un fichier lex est composé de trois parties séparées par '%%'.

```
Les définitions
%%
Les règles
%%
Le code utilisateur
```

Si vous n'avez pas de code utilisateur le dernier '%%' peut-être omis. Voici le plus petit programme lex :

```
%%
```

qui n'a pour simple tâche que d'afficher sur la sortie standard caractère par caractère tous les caractères qui viennent de l'entrée standard. Vous pouvez tester ce programme en faisant :

```
$ lex rien.lex
$ gcc lex.yy.c -lfl -o rien
$ rien
```

Les définitions :

- Dans la partie des définitions, vous pouvez mettre deux types de définitions. Soit des déclarations ou des #define pour votre code C, ou bien des définitions d'expression régulières pour lex. Lex différencie les deux types de définitions par la mise en forme que vous donnez à vos lignes. Par défaut les définitions pour lex sont des définitions qui commencent au premier caractère de chaque ligne. Les définitions pour le code C, par exemple des #include, des déclarations de variables, types, ou constante (#define) doivent être mis en forme différemment. Vous avez la possibilité entre deux méthodes soit faire précéder d'une indentation votre ligne, soit l'encapsuler entre '%{' et '%}'. Vous pouvez aussi insérer des commentaires encadrés par '/*' et '*/' et précédé d'une tabulation (ou un espace).

```
        /* -- exemplevide.lex
        Cet exemple ne fait rien
        Il contient seulement des definitions
        Utilisation :
                lex exemplevide.lex
                gcc lex.yy.c
        */
%{
#define PI              3.14159 /* une constante */
int val;        /* Une variable */
%}
```

```
    int value;          /* une autre variable */
ENTIER   [0-9]+
BLANC    [ \t\n]
BLANCS   [ \t\n]+
%%
```

Les règles :

Dans la section des règles, vous pouvez définir une action pour chacune des expressions régulières que vous recherchez.

Les expressions régulières suivent un format précis, que vous avez vu en cours. En cas d'"oubli", vous pouvez retrouver la syntaxe dans la documentation de la commande 'grep', est-il nécessaire de rappeler où se trouve la documentation ? Vous pouvez soit faire 'man grep', soit aller jeter un coup de clique sur le site de gnu, dans le manuel de la commande 'grep' et plus précisément dans la section 'Regular Expressions'. Les expressions reconnues par lex (et flex) sont un sur-ensemble des expressions régulières classiques... Là aussi, vous pouvez, jeter un clique dans la documentation de flex, dans la section des règles ('Pattern'), ou bien faire..... 'man lex' !!!!!!!!!!.

Les actions, quant à elles, sont du code C, avec quelques particularités. Dans le code lex, vous avez à votre disponibilité plusieurs variables globales, ainsi que quelques fonctions vous permettant de modifier le comportement de l'automate lex.

Nous pouvons à l'aide des règles reconnaître toutes sortes d'expressions régulières. Par exemple, le programme suivant reconnaît les entiers et les mots.

```
        /* -- lire1.lex --
        Ce programme reconnait des mots et des entiers
        */

%%
[0-9]+      printf("entier ");
[a-zA-Z]+   printf("mot ");
```

Exemple d'utilisation :

Le texte "*12 AZ 12*" donne "*entier mot entier*", et le texte "*monsieur le num 723 a 34 euros*" donne comme sortie "*mot mot mot entier mot entier mot*"

L'automate 'lex' peut nous donner plus de renseignement sur ce que l'on a trouvé comme texte par l'intermédiaire de la variable globale 'yytext' (char yytext[YYLMAX];) qui contient la chaîne reconnu par la règle. Par exemple, avec le programme précédent, on voudrait afficher la valeur des entiers trouvés ainsi que les mots reconnus.

Le programme suivant illustre la variable yytext, il consiste à inverser la casse des lettres rencontrés, et laisse intacte les autres caractères.

```
        /* -- inversecasse.lex --
        Ce programme inverse la casse de toutes les lettres
        */
    #include <ctype.h>
%%
[a-z]   printf("%c", toupper(yytext[0]));
[A-Z]   printf("%c", tolower(yytext[0]));
```

Ce qui donne à l'exécution :

```
$ lex inversecasse.lex
$ gcc lex.yy.c
inversecasse
$ inversecasse
azerty1234+/.!#;WXCVBN
AZERTY1234+/.!#;wxcvbn
```

Jusqu'à maintenant, les actions étaient simples, et les programmes ne faisaient qu'une reconnaissance d'expressions simples et il n'y a aucun lien entre chaque action. En déclarant des variables globales dans votre programme, vous pouvez écrire des outils plus intéressants, imaginons par exemple un programme qui vérifie le parenthésage.

```
    /* -- par.lex --
    Ce programme compte le nb de parentheses ouvrantes et
    verifie si le nb de parenthese fermante correspond.
    */

    int nbParOuv=0; /* Variable globale du nb de par ouvrantes */
%%
"("     {nbParOuv++; ECHO;}
\)      {
        if (nbParOuv>0)
            {
            nbParOuv--;
            ECHO;
            }
        else
            printf ("Votre parenthesage est incorrect\n");
        }
```

Vous noterez au passage les différentes écritures possibles de caractères spéciaux.

Le code utilisateur :

Les programmes que nous avons écrit jusque-là affichaient des résultats par le moyen des actions. Mais nous pouvons écrire dans les actions des simples calculs et afficher le résultat à la fin de l'analyse. Pour cela on doit introduire un code utilisateur dans la dernière partie du fichier lex. Afin de lancer l'analyseur vous devez appeler la fonction 'yylex'.

Voici l'exemple d'un programme qui fait des statistiques sur un fichier.

```
               /* -- nlnc.lex --
               Ce programme compte le nb de lignes et
               le nb de caracteres
               */
          int nbLignes = 0, nbCaracteres = 0;

%%
\n      ++nbLignes; ++nbCaracteres;
.       ++nbCaracteres;

%%
main()
          {
          yylex();
          printf( "Nb de lignes = %d, Nb de caracteres = %d\n", nbLignes,
nbCaracteres );
          }
```

Les 'Token' :

Voici un programme d'évaluation d'expression postfixée, qui utilise les notions vu précédemment.

```
    /* -- postfix_eval.lex --
       Lit et evalue une expression postfixee.
    */

%{
#define MAXSP 100
int value;      /* valeur associee a la chaine reconnue */
int st[MAXSP];  /* pile */
int sp=-1;      /* pointeur de pile */

#define STOP 0
%}

%%

[0-9]+      {
            value=atoi(yytext);
            if (sp >= MAXSP) return STOP;
            st[++sp] = value;
            }

[-+*/]      {
            int a1, a2;
            value=yytext[0];
            if (sp < 0)
                {
                printf("Pas assez d'arguments pour l'operation
'%c'\n",value);
                return STOP;
                }
```

```
            else
                a2 = st[sp--];      /* on depile le 2em arg */

            if (sp < 0)
                {
                printf("Premier argument manquant pour l'operation
'%c'\n",value);
                return STOP;
                }
            else
                a1 = st[sp--];      /* on depile le 1er arg */

            switch (value)
                {
                case '-':
                    st[++sp] = a1 - a2;
                    break;
                case '*':
                    st[++sp] = a1 * a2;
                    break;
                case '+':
                    st[++sp] = a1 + a2;
                    break;
                case '/':
                    st[++sp] = a1 / a2;
                    break;
                }
            }

[ \t\n]+    ;
.           printf("car. non reconnu: (%c)\n",yytext[0]);

%%
main()
    {
    /* Analyse de l'expression */
    yylex();

    if ( sp != 0 )
        printf("sp!=0 : expr. postfixee incorrecte ou erreur prog. C\n");
    else
        printf("Resultat : %d\n", st[sp--] );
    }
```

Ce programme 'lex' peut-être écrit autrement en utilisant le concept de 'token'. Les programmes 'lex' s'écrivent autrement, les règles reconnaissent des expressions, et les actions renvoient une indication ('token'). C'est ensuite la fonction (le main généralement) qui appelle l'analyseur (yylex()) qui se charge de faire 'avancer' un automate qui peut-être très complexe. Avec ce format d'utilisation de 'lex', les règles s'écrivent plus simplement, et on sépare l'analyseur lexical de l'automate de calcul.

Les tokens sont définis comme étant des constantes. Les actions renvoient l'un de ces tokens à la fin de l'évaluation d'une chaîne.

L'évaluation d'une expression postfixée peut s'écrire ainsi :

```
    /* -- postfix_evalv2.lex --
       Lit et evalue une expression postfixee.
    */

%{
#define TEOF 0 /* Fin de fichier */
#define TNB 1 /* Token : entier */
#define TOP  2 /* Token : operation */
int value;      /* valeur associee a la chaine reconnue, ou bien le car. de
l'operation */
%}

%%

[0-9]+      { value=atoi(yytext);   return TNB; }
[-+*/]      { value=yytext[0];      return TOP; }

[ \t\n]+    ;
.           printf("car. non reconnu: (%c)\n",yytext[0]);

%%

#define MAXSP 100
int st[MAXSP];  /* pile */
int sp=-1;      /* pointeur de pile */
main()
    {
    int  token;
    int error = 0;
    int a1, a2;

    /* Analyse de l'expression */
    while (! error && (token=yylex())!=TEOF)
        {
        switch (token)
            {
            case TNB :
                if (sp >= MAXSP)
                    error=1;
                else
                    st[++sp] = value;
                break;

            case TOP :
                if (sp < 0)
                    {
                    printf("Pas assez d'arguments pour l'operation
'%c'\n",value);
                    error = 1;
                    }
                else
                    a2 = st[sp--];      /* on depile le 2em arg */

                if (sp < 0)
                    {
                    printf("Premier argument manquant pour l'operation
'%c'\n",value);
                    error = 1;
                    }
                else
                    a1 = st[sp--];      /* on depile le 1er arg */
```

```
                switch (value)
                    {
                    case '-':
                        st[++sp] = a1 - a2;
                        break;
                    case '*':
                        st[++sp] = a1 * a2;
                        break;
                    case '+':
                        st[++sp] = a1 + a2;
                        break;
                    case '/':
                        st[++sp] = a1 / a2;
                        break;
                    }
                break;
            }
        }

    if ( sp != 0 || error)
        printf("sp!=0 : expr. postfixee incorrecte ou erreur prog. C\n");
    else
        printf("Resultat : %d\n", st[sp--] );
    }
```

Conclusion :

Vous avez maintenant une bonne idée de ce qu'est 'lex'. Ces quelques lignes vous ont montré quelques notions de 'lex', mais nous n'avons pas abordé toutes les capacités de cet outil.

Yacc :

Introduction :

'lex' est un analyseur lexical, ce n'est pas un analyseur grammatical. C'est-à-dire que 'lex' ne vous permet pas de saisir une grammaire complexe d'un langage et l'analyser. C'est le rôle de 'yacc'. 'yacc' vous permet de faire ce que l'on appelle un 'parser'. Vous pouvez reconnaître tout type de langage à partir du moment où vous avez sa grammaire, exemple un fichier .ps, .html, .c, .pas, etc...

'yacc' est un analyseur grammatical qui a été spécialement conçu pour utiliser 'lex' comme analyseur lexical, ils marchent ensemble et chacun à son niveau. La partie 'lex' lit le fichier, décode des tokens et les renvoie à la partie 'yacc'. La partie 'yacc' ne lit pas le fichier en entrée mais seulement les tokens renvoyés par 'lex', au travers de la fonction 'yylex()'. La communication entre les deux se passe aux niveaux des tokens et aussi de la variable 'yylval' que nous verrons plus tard.

Le format :

Le format d'un fichier yacc est identique au format de 'lex', il est composé de trois parties séparées par '%%'.

```
Les définitions
%%
Les règles
%%
Le code utilisateur
```

Le format des règles :

Dans la partie des règles de 'yacc', contrairement à 'lex' où vous donnez un ensemble d'expressions régulières, vous entrez l'ensemble des règles qui définissent votre grammaire.

Exemple :

Soit une grammaire au format BNF :

```
<EXPR_CALCS> ::= EXPR_CALC
<EXPR_CALCS> ::= EXPR_CALCS EXPRE_CALC

<EXPR_CALC> ::= <EXPR_NUM> =

<EXPR_NUM> ::= <FACTEUR>
<EXPR_NUM> ::= <EXPR_NUM> + <FACTEUR>
<EXPR_NUM> ::= <EXPR_NUM> - <FACTEUR>

<FACTEUR> ::= <TERME>
<FACTEUR> ::= <FACTEUR> * <TERME>
<FACTEUR> ::= <FACTEUR> / <TERME>

<TERME> ::= <NB>
<TERME> ::= ( <EXPR_NUM> )
```

Suivant cette grammaire, vous pouvez reconnaître les expressions de la forme '2+3=', ou encore '2*(5-2)/(8+4)='

Le programme 'yacc' qui reconnait les expressions et les affiche s'écrit :

```
    /* -- eval.yac --
       Evaluation d'une expression
       Partie analyseur grammatical.
    */

%token  Tnb
%start  EXPR_CALCS

%%
EXPR_CALCS : EXPR_CALC
{printf ("EXPR_CALCS1\n");}
           | EXPR_CALCS EXPR_CALC
{printf ("EXPR_CALCS2\n");}
           ;
```

```
EXPR_CALC : EXPR_NUM '='
{printf ("EXPR_CALC ");}
          ;

EXPR_NUM : FACTEUR
{printf ("facteur(expr num) ");}
         | EXPR_NUM '+' FACTEUR
{printf ("ADDITION ");}
         | EXPR_NUM '-' FACTEUR
{printf ("SOUSTRACTION ");}
         ;

FACTEUR  : TERME
{printf ("terme(facteur) ");}
         | FACTEUR '*' TERME
{printf ("PRODUIT ");}
         | FACTEUR '/' TERME
{printf ("DIVISION ");}
         ;

TERME    : Tnb
{printf ("ENTIER ");}
         | '(' EXPR_NUM ')'
{printf ("EXPR entre (). ");}
         ;

%%

#include <stdio.h>
#include "lex.yy.c"

yyerror () {}

main()
{
 if ( yyparse() != 0 )
    { fprintf(stderr,"Syntaxe
incorrecte\n"); return 1; }
 else
    return 0;
}
```

La communication entre 'lex' et 'yacc' (1) : les 'tokens'

Mais qu'est-ce qu'un 'Tnb' ? Et d'où viennent les caractères des opérations ? Ils sont analysés par 'lex' et fourni à 'yacc'. Chaque élément trouvé est renvoyé tel quel (les opérateurs), soit vous renvoyez sous formes de constantes représentant un type de terminal lexical (Tnb).

Le fichier 'lex' correspondant est très simple car il ne s'occupe que de rechercher les opérations et les nombres :

```
    /* -- eval.lex --
       Evaluation d'une expression
       Partie analyseur lexical.
    */
```

```
BLANC [ \n\t]
%%

[0-9]+      return Tnb;
[-+*/()=]   return yytext[0];   /*
caracteres unites lexicales */
{BLANC}+    ;
.           fprintf(stderr,
"Caractere (%c) non reconnu\n",
yytext[0]);
```

Ce qui donne :

```
$ lex eval.lex
$ yacc eval.yac
$ gcc y.tab.c -lfl -o eval
$ eval
2 + 3 =
entier terme(facteur) facteur(expr num) entier terme(facteur) addition
expr_calc expr_calcs1
```

La communication entre 'lex' et 'yacc' (2) : 'yylval'

Les tokens reconnus par 'lex' peuvent correspondrent à plusieurs chaînes diffrentes, par exemple dans le programme précédent, le token Tnb représente un nombre entier, mais il peut représenter aussi bien le nombre 2, ou bien le nombre 539... Pour réaliser un programme d'évaluation d'expression de la forme '2 + 539 =', la partie 'yacc' doit connaître la valeur des nombres entiers lus. Pour cela, en plus des tokens, la communication entre 'lex' et 'yacc' est complétée par la variable globale : 'yylval', elle remplace la variable (int value) que nous avions déclarée dans le fichier 'lex' pour l'évaluation d'une expression postfixée. Par défaut 'yylval' est un entier.

Le fichier 'lex' doit être modifié comme suit pour retourné la valeur d'un entier:

```
    /* --  evalv2.lex  --
       Evaluation d'une expression
       Partie analyseur lexical.
    */

BLANC [ \n\t]
%%

[0-9]+      yylval = atoi(yytext);
return Tnb;
[-+*/()=]   return yytext[0];   /*
caracteres unites lexicales */
{BLANC}+    ;
.           fprintf(stderr,
"Caractere (%c) non reconnu\n",
yytext[0]);
```

Dans 'yacc', les tokens, ainsi que chaqu'une des règles, peuvent correspondre à des valeurs. Cette spécificité nous permet de faire le calcul à la volée de façon simple dans l'écriture des actions. La syntaxe est un peu particulère, elle est inspiré des arguments d'un script shell (ou perl). La valeur du premier élément de la règle est représentée par '$1', celle du deuxième est représentée par '$2' et ainsi de suite. La valeur de retour est représentée par '$$'.

Le fichier 'yacc' correspondant est :

```
    /* --  evalv2.yac  --
       Evaluation d'une expression
       Partie analyseur grammatical.
    */

%token  Tnb
%start  EXPR_CALCS

%%
EXPR_CALCS : EXPR_CALC
           | EXPR_CALCS EXPR_CALC
           ;

EXPR_CALC : EXPR_NUM '='
{printf ("%d\n", $1);}
          ;

EXPR_NUM : FACTEUR
         | EXPR_NUM '+' FACTEUR
{$$ = $1 + $3;}
         | EXPR_NUM '-' FACTEUR
{$$ = $1 - $3;}
         ;

FACTEUR  : TERME
         | FACTEUR '*' TERME
{$$ = $1 * $3;}
         | FACTEUR '/' TERME
{$$ = $1 / $3;}
         ;

TERME    : Tnb
{$$ = $1;}
         | '(' EXPR_NUM ')'
{$$ = $2;}
         ;

%%

#include <stdio.h>
#include "lex.yy.c"

yyerror (char * error) {}

int main()
{
 if ( yyparse() != 0 )
    { fprintf(stderr,"Syntaxe
incorrecte\n"); return 1; }
```

```
 else
    return 0;
}
```

Dans l'exemple suivant, nous voulons étendre notre programme à l'évaluation d'expression comportant des fonctions, par exemple la fonction absolue, ou négation (du signe). Le fichier 'lex' est modifié pour reconnaître un nom de fonction suivant l'expression régulière :

```
[a-zA-Z_]+      return Tid;
```

et le fichier 'yacc' contient les règles supplémentaires suivantes :

```
TERME    : Tnb
{$$ = $1;}
         | '(' EXPR_NUM ')'
{$$ = $2;}
         | APPEL_FONC
{$$ = $1;}
         ;


APPEL_FONC : Tid '(' EXPR_NUM ')'
{$$ = ??? }
           ;
```

Le problème est de savoir quel est le nom de la fonction reconnue. Le fichier 'lex' reconnaît l'identificateur (qui est contenu dans yytext) mais ne peut renvoyer que le token (Tid). On peut s'inspirer de la méthode utilisée pour les nombres entiers, mais nous avons vu que la variable yylval était un entier. En fait, on peut changer son type et utilisé par exemple le mot clef %union de 'yacc'. Ainsi, yylval devient une union de champ dont vous précisez la liste entre accolades.

```
%union  { char chaine[256]; int
valeur; }
```

En utilisant le mot clef '%union' de 'yacc', la variable yylval peut prendre plusieurs types (char [], ou bien int)... Dans ce cas-là, 'yacc' ne peut déterminer ce que doivent faire vos règles ($$ = $1 + $2, ...). Vous devez alors préciser dans le fichier 'yacc', pour chaque token, le type (le champ utilisé de l'union) entre '<' et '>' et pour chaque règle en utilisant le mot clef '%type'.

Voici le fichier 'lex' complet :

```
    /* --  evalv3.lex  --
       Evaluation d'une expression
       Partie analyseur lexical.
    */

BLANC [ \n\t]
%%
[a-zA-Z_]+  strcpy(yylval.chaine,
```

```
yytext);  return Tid;
[0-9]+      yylval.valeur =
atoi(yytext);   return Tnb;
[-+*/()=]   return yytext[0];   /*
caracteres unites lexicales */
{BLANC}+    ;
.           printf("Caractere (%c)
non reconnu\n", yytext[0]);
```

Et le fichier 'yacc' complet :

```
    /* --  evalv3.yac  --
       Evaluation d'une expression
       Partie analyseur grammatical.
    */
%{
int appel_fonc (char *, int, int *);
%}

%union  { char chaine[256]; int
valeur; }
%token <chaine> Tid
%token <valeur> Tnb
%type  <valeur> EXPR_NUM FACTEUR
TERME APPEL_FONC
%start  EXPR_CALCS

%%
EXPR_CALCS : EXPR_CALC
           | EXPR_CALCS EXPR_CALC
           ;

EXPR_CALC : EXPR_NUM '='
{printf ("%d\n", $1);}
          | error '='
{yyerrok;}
          ;

EXPR_NUM : FACTEUR
         | EXPR_NUM '+' FACTEUR
{$$ = $1 + $3;}
         | EXPR_NUM '-' FACTEUR
{$$ = $1 - $3;}
         ;

FACTEUR  : TERME
         | FACTEUR '*' TERME
{$$ = $1 * $3;}
         | FACTEUR '/' TERME
{$$ = $1 / $3;}
         ;

TERME    : Tnb
{$$ = $1;}
         | '(' EXPR_NUM ')'
{$$ = $2;}
         | APPEL_FONC
{$$ = $1;}
         ;
```

```
APPEL_FONC : Tid '(' EXPR_NUM ')'
{if (! appel_fonc ($1, $3, &$$))
YYERROR;}
           ;
%%

#include <stdio.h>
#include <string.h>
#include "lex.yy.c"

yyerror ()
{ fprintf(stderr,"Erreur\n"); }

main()
{
    if ( yyparse() != 0 )
        { fprintf(stderr,"Syntaxe
incorrecte\n"); return 1; }
    else
        return 0;
}

int appel_fonc (char *chaine, int
valeur, int * val)
{
    if (strcmp(chaine, "abs")==0)
        *val = abs(valeur);
    else if (strcmp(chaine,
"neg")==0)
        *val = - valeur;
    else
        { fprintf(stderr,"Fonction
inconnue\n"); return 0; }
    return 1;
}
```