COMPILATION

L'ANALYSE SYNTAXIQUE ASCENDANTE

EMSI - 4ÈME IIR

2017/2018

Prof. M. D. RAHMANI

L'analyse ascendante

Plan:

1- Le but d'un analyseur ascendant

2- Implantation d'une grammaire LR

3- Le langage YACC (LALR)

L'analyse ascendante (LR, RR)

1- <u>But</u>:

L'***analyse ascendante*** consiste à ***réduire*** la chaîne d'entrée en un symbole initial, qui est l'***axiome***.

Il s'agit de la construction de l'arbre syntaxique en partant des éléments de la phrase à analyser (***les terminaux***) qui sont les feuilles de l'arbre, vers la racine de l'arbre qui correspond à l'***axiome*** de la grammaire.

L'analyse ascendante (LR, RR)

Remarques:

- Ce processus de construction de l'arbre est une ***réduction***, qui se traduit par le remplacement de la partie droite des productions par la partie gauche.

- Les grammaires LR (ascendantes) sont plus générales que les grammaires LL (descendantes).

En fait pour les algorithmes ascendants, la décision d'effectuer une réduction donnée est prise lorsque l'on dispose de l'ensemble de son membre droit pour le remplacer par le non terminal à gauche,

- ✓ Par contre, dans le cas d'un algorithme descendant, la décision est prise lorsque l'on dispose d'un symbole de son membre gauche.
- ✓ Dans le 1er cas, on dispose en général de plus d'informations que dans le second.

L'analyse ascendante LR

2- <u>Implantation de la méthode LR</u>:

Il s'agit d'une méthode d'analyse générale appelée "***analyse par décalage-réduction***" (*Shift Reduce Parsing*)

	Pile	Tampon
état initial:	$	ω$
état final:	$S	$

avec ω la chaîne à analyser et **S** l'axiome de la grammaire.

A l'état final, la chaîne ω est réduite en l'axiome **S** à moins qu'une erreur soit détectée pendant l'analyse.

L'analyse ascendante LR

Les opérations à effectuer sont:

1- **`Décalage`**: le symbole d'entrée courant est inséré dans la **pile**,

2- **`Réduction`**: l'analyseur reconnait la partie de droite d'une production au sommet de la **pile**, elle la remplace alors par le non-terminal correspondant,

3- **`Acceptation`**: la chaîne est réduite en l'axiome,

4- **`Erreur`**: détection d'une erreur de syntaxe et appel d'une routine de traitement d'erreur.

L'analyse ascendante LR

Exemple de fonctionnement: soit la chaîne ω = "**abbcde**"

La grammaire:

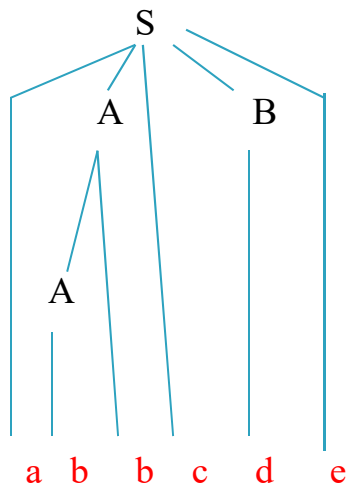
S → **aAcBe**
A → **Ab | b**
B → **d**

(1) conflit décalage/ réduction

(2) conflit réduction/ réduction

Pile	Tampon	Action
$	**abbcde**$	***décalage***
$a	bbcde$	***décalage***
$ab	bcde$	***réduction*** A → b (1)
$aA	bcde$	***décalage***
$aAb	cde$	***réduction*** A → Ab (1,2)
$aA	cde$	***décalage***
$aAc	de$	***décalage***
$aAcd	e$	***réduction*** B → d (1)
$aAcB	e$	***décalage***
$aAcBe	$	***réduction*** S → aAcBe
$S	**$**	**ACCEPTATION**

Arbre syntaxique



Exemples d'analyse ascendante LR

- Soit la grammaire: la phrase à analyser: « **id+id+id** »

  E ⟶ **E + T | T**

  T ⟶ **id**

- Soit la grammaire: la phrase à analyser: « **id+id+id** »

  E ⟶ **E + E | id**

L'analyse ascendante LR

Les inconvénients:

La méthode n'est pas déterministe et exige un gros travail d'analyse,
Peu de langages de programmation utilisent ces méthodes.

3 méthodes dérivent de la méthode générale:

- **SLR** (Simple LR): facile à implanter et non ambigüe mais adaptée à une classe limitée de grammaires,
- **LR canonique**: est la plus efficace car la plus générale mais la plus couteuse en temps et en mémoire,
- **LALR** (***Look Ahead***) est intermédiaire entre les 2 méthodes précédentes et prédictive,

La dernière méthode a donné naissance à l'outil ***YACC***.

Le langage YACC

Conçu au début des années 70 par Johnson, le langage YACC, utilitaire d'unix, veut dire, ***Yet Another Compiler Compiler*** *(encore un autre compilateur de compilateurs)*.

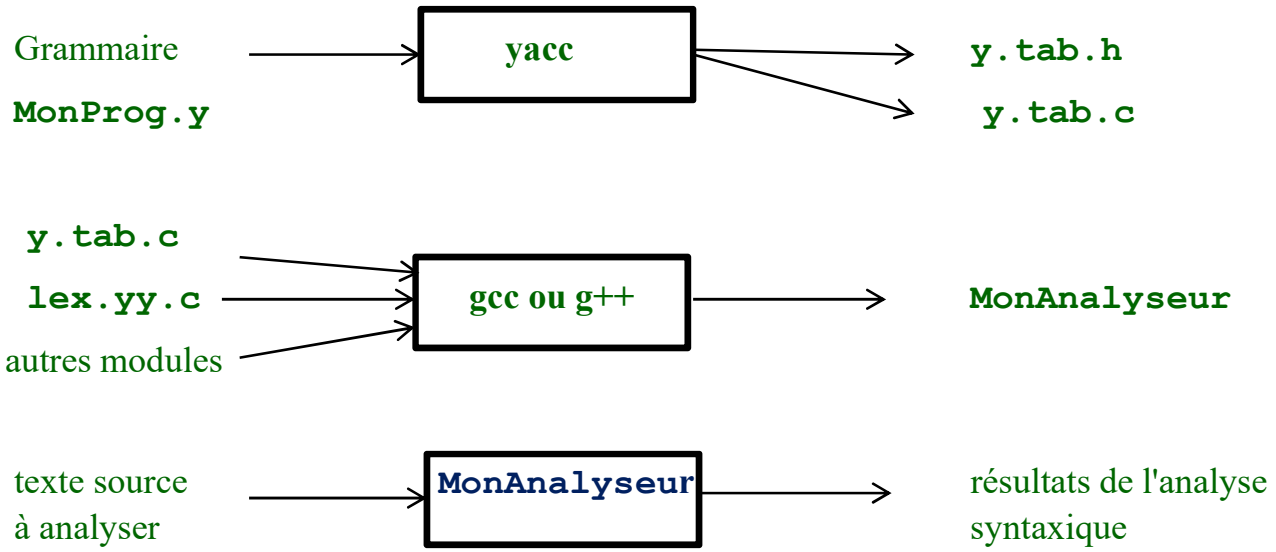
1- <u>But</u>:

Construire à partir d'une grammaire une fonction **C** nommée ***yyparse()***, qui est un analyseur syntaxique qui reconnait les constructions du langage décrit par la grammaire.

Un programme écrit en langage ***YACC*** prend en entrée un fichier source constitué essentiellement des **productions d'une grammaire *G*** et produit un ***programme C*** qui, une fois compilé, est un analyseur syntaxique pour le **langage *L(G)***.

Le langage YACC

Utilisation du Langage YACC



Le langage YACC

2- <u>Structure d'un code en YACC</u>:

Le générateur d'analyseurs syntaxiques **YACC** a une structure analogue à celle du **LEX**.

Le code source **YACC**, doit avoir un nom terminé par "**.y**". Il est composé de 3 sections délimitées par deux lignes "**%%**".

```
1- déclarations
%%
2- règles de traduction
%%
3- routines annexes en langage C
```

Le langage YACC

2.1- La section des déclarations:

Elle est constituée de 2 parties:

1- Une partie entre `%{` et `%}` des **déclarations à la C**, *des variables, unions, structures, ...*

2- Une partie constituée de la déclaration de

- l'**axiome**, avec `%start`
- des **terminaux**, avec `%token`
- des **opérateurs**, en précisant leur *associativité*, avec

  `%left`, `%right`, `%nonassoc`

Remarques:

- *Le symbole* `%` *doit être à la 1ère colonne*
- *Tout symbole non déclaré dans cette partie par* `token` *est considéré comme un* ***non terminal***

Le langage YACC

2.2- La section des règles de traduction:

Après %% nous avons dans cette partie **une suite de règles de traduction**.

Chaque règle est constituée par une **production de la grammaire** associée aux phrases du langage à analyser et éventuellement une action sous forme d'**une règle sémantique**.

La règle sémantique peut prendre la forme d'un ***schéma de traduction*** ou d'***une définition dirigée par la syntaxe***.

Si on a la règle: **E** ⟶ **E + T | E - T | T**

en YACC, on écrit:

```
E : E '+' T
  | E '-' T
  | T
  ;
```

Le langage YACC

2.3- La section des fonctions annexes en C:

Après %% nous avons dans cette partie une suite de fonctions écrite en langage C et utilisables par la 2ème partie des règles de traduction et le bloc principal sous la forme:

```
int main() {
        yyparse();  // la fonction principale du yacc
        return 0;
}

int yyerror(char *message) {
        printf("<< %s", message);
        return 1;
}
```

Remarques :

- Sous Unix la 1ère et la 3ème parties sont facultatives.

- L'analyseur syntaxique se présente comme une fonction yyparse(void), qui rend O si la chaîne d'entrée est acceptée.

- Le programme ***yacc*** peut communiquer avec un programme ***lex*** qui lui fournit les terminaux qui peuvent constituer les phrases.

Application: Calculatrice scientifique (1)

Nous allons écrire un code complet en **Flex** et **Bison**, d'une calculatrice scientifique, qui va lire une expression arithmétique, l'évaluer et afficher le résultat.

La lecture et l'affichage se fait par des règles sémantiques sous forme de schéma de traduction.

L'évaluation se fait par des règles sémantiques sous forme de définition dirigée par la syntaxe qui calculent des attributs.

L'expression à évaluer est **E** et le résultat de l'évaluation est dans **R**.

Soit la grammaire:

```
R  ⟶  E  fin
E  ⟶  E + T | E - T | T
T  ⟶  T * F | T /  F | F
F  ⟶  (E) | nombre
```

Application: Calculatrice scientifique (2)

Un programme en YACC: ***calc.y***

1ère partie

```
%{
#include<stdio.h>
%}
%token nombre fin       // priorité des opérateurs augmente
%left plus moins        // plus et moins ont la même priorité
%left fois div          // fois et div ont la même priorité
%start R
%%
```

Application: Calculatrice scientifique (3)

Un programme en YACC (suite):

2ème partie *sans règles sémantiques*

```
R : E  fin          // 1ère règle de grammaire
  ; // séparateur des règles de grammaire
E : E plus T        // 2ème règle de grammaire
  | E moins T
  | T
  ; // séparateur des règles de grammaire
T : T fois F        // 3ème règle de grammaire
  | T div F
  | F
  ; // séparateur des règles de grammaire
F : '(' E ')'       // 4ème règle de grammaire
  | nombre
  ; // séparateur des règles de grammaire
%%
```

Application: Calculatrice scientifique (4)

2ème partie ***avec les règles sémantiques***

- A chaque symbole de la grammaire (terminal ou non) est associé une valeur (de type entier par défaut).
- Cette valeur peut être utilisée dans les actions sémantiques.
- Le symbole $$ référence la valeur de l'attribut associé au non-terminal de la partie gauche d'une production de la grammaire.
- Le symbole $i référence la valeur associée au i-ème symbole (terminal ou non-terminal) de la partie droite d'une production de la grammaire.

Application: Calculatrice scientifique (5)

2ème partie *avec les règles sémantiques*

```
R : E fin           { printf("le résultat est: %d", $1);}
  ;
E : E plus T        { $$ = $1 + $3;}
  | E moins T       { $$ = $1 - $3;}
  | T               { $$ = $1; // facultatif }
  ;
T : T fois F        { $$ = $1 * $3;}
  | T div F         { if ($3==0) printf ("division par zéro interdite") else $$ = $1 / $3;}
  | F               { $$ = $1;}
  ;
F : '(' E ')'       { $$ = $2;}
  | nombre          { $$ = $1;}
  ;
%%
```

Application: Calculatrice scientifique (6)

3ème partie:

```
%%
int main(void) {
        printf("début de l'analyse");
        if(yyparse() == 0)
           printf("texte valide");
}
int yyerror(char *message) {
     printf("<< %s", message);
     return 1;
}
```

Remarque :

L'analyseur syntaxique se présente comme une fonction `int yyparse(void)` qui rend 0 si la chaîne est acceptée, non nulle dans le cas contraire.

Application: Calculatrice scientifique (7)

Un programme en Lex: ***analex.l***

```
%{
#include<stdlib.h>
#include "calc.tab.h"  // fichier d'interface fourni par le fichier yacc
%}
nombre [0-9]+
%%
[ \t]            {/* ne rien faire */}
{nombre}         {yylval = atoi(yytext); return(nombre);}
"\n"             {return (fin);}
"+"              {return (plus);}
"-"              {return (moins);}
"/"              {return (div);}
"*"              {return (fois);}
%%
```

Application: Calculatrice scientifique (8)

Communication avec l'analyseur lexical: `yylval`

- L'analyseur syntaxique et l'analyseur lexical peuvent communiquer entre eux par l'intermédiaire de la variable `int yylval`.
- Dans une action lexicale du fichier ***flex***, l'instruction `return(unité)` permet de renvoyer à l'analyseur syntaxique l'unité lexicale `unité`, dont la valeur est rangée dans `yylval`.
- L' analyseur prend automatiquement le contenu de `yylval` comme valeur de l'attribut associé à cette unité lexicale.
- La valeur `yylval` est de type `YYSTYPE`, déclarée dans la bibliothèque **bison** qui est un `int` par défaut. On peut changer ce type par:

```
#define YYSTYPE autre_type_C
```

exemple: `#define YYSTYPE double`

Application: Calculatrice scientifique (9)

Conflits en bison `décalage-réduction` et `réduction-réduction`:

- Lorsque l'analyseur syntaxique est confronté à des conflits, il rend compte du type et du nombre de conflits rencontrés:
  - `bison exemple.y` **le résultat** `conflicts: 6 shift/reduce, 2 reduce/reduce`
- Il y'a un conflit **`reduce/reduce`** lorsque le compilateur a le choix entre aux moins deux productions pour réduire une chaîne.
- Les conflits **`shift/reduce`** apparaissent lorsque le compilateur a le choix entre réduire par une production et décaler, c'est à dire déplacer un symbole du tampon vers la pile.
- yacc/bison résout les conflits de la manière suivante:
  - **`conflit reduce/reduce`**: la production choisie est celle apparaissant en premier dans la spécification.
  - **`conflit shift/reduce`**: c'est le shift qui est effectué en premier.

Pour voir comment bison résout les conflits, il faut compiler avec l'option **`-v`** pour produire **`nom.output`**

Application: Calculatrice scientifique (10)

Les étapes de compilation:

1 étape: > ***bison -d calc.y***

en sortie on a, ***calc.tab.c*** et ***calc.tab.h***

2ème étape: > ***flex analex.l***

en sortie on a: ***lex.yy.c***

3ème étape: > ***gcc -c lex.yy.c -o cal.l.o***

gcc -c calc.tab.c -o calc.y.o

gcc -o calc calc.l.o calc.y.o -lfl -lm

lfl: ***Library Fast Lex*** *(la librairie du Flex)*

lm: la librairie de *Bison*

TP: Calculatrice scientifique (1)


```
/* test avec bison b.y */
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token FIN NOMBRE
%left PLUS MOINS
%start R
%%
R : E FIN {printf("Resultat = %d",$1);return 0;}
  ;
E : E PLUS T  {$$=$1+$3;}
  | E MOINS T {$$=$1-$3;}
  | T         {$$=$1;}
  ;
T : NOMBRE
  ;
```

```
%%
int yyerror(char *s) {
  printf("%s",s);
  return 1;
}
int main() {
  printf("Donnez une expression : ");
  yyparse();
  return 0;
}
```

A compiler avec: >bison -d b.y
en sortie on a : b.tab.c et b.tab.h

TP: Calculatrice scientifique (2)


```
/* test avec Flex a.l */
%{
#include <math.h>
#include <stdlib.h>
#include "b.tab.h"
%}
nombre [0-9]+
%%
{nombre} {yylval=atoi(yytext); return NOMBRE;}
"\n"     {return FIN;}
"+"      {return PLUS;}
"-"      {return MOINS;}
 .       { }
```

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

A compiler avec >flex a.l

en sortie on a : lex.yy.c

TP: Calculatrice scientifique (3)


```
/* test avec Flex et bison a.l */
%{
#include <math.h>
#include <stdlib.h>
#include "b.tab.h"
%}
nombre [0-9]+
%%
{nombre} {yylval=atoi(yytext); return NOMBRE;}
"\n"     {return FIN;}
"+"      {return PLUS;}
"-"      {return MOINS;}
.        { }
%%
int yywrap(void) {
  return 1;
}
```

```
/* test avec bison b.y */
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token FIN NOMBRE
%left PLUS MOINS
%start R
%%
R : E FIN {printf(" Resultat = %d",$1);}
  ;
E : E PLUS T  {$$=$1+$3;}
  | E MOINS T {$$=$1-$3;}
  | T         {$$=$1;}
  ;
T : NOMBRE
  ;
%%
int yyerror(char *s) {
  printf("%s",s);
  return 1;
}
int main() {
  printf(printf(" Donnez une expression arithmetique: ");
  yyparse();
}
```

TP: Calculatrice scientifique (4)

Les étapes de compilation:

1 étape: > ***bison -d b.y***

en sortie on a, ***b.tab.c*** et ***b.tab.h***

l'analyseur syntaxique: ***b.tab.c***

et l'interface avec flex: ***b.tab.h***

2ème étape: > ***flex a.l***

en sortie on a l'analyseur lexical: ***lex.yy.c***

3ème étape: > Production des codes objets

gcc -c lex.yy.c -o a.o

gcc -c a.tab.c -o b.o

gcc -o ab a.o b.o

en sortie on a : ***ab.exe***

Test du + et - entre entiers

Voir le contenu du fichier ***b.tab.h***

```
#define FIN=258
# define NOMBRE=259
#define MOINS=260
#define PLUS=261
#define MOINS=262
typedef int YYSTYPE;
extern YYSTYPE yylval;
```

TP: Calculatrice scientifique (5)

Les étapes suivantes:

1- L'ajout de la multiplication pour vérifier la priorité (***a1.l*** et ***b1.y***). Exemple: 3 + 5 * 2 = ?

2- L'ajout des parenthèses pour forcer une priorité (***a2.l*** et ***b2.y***). Exemple: (3+5) * 2 = ?

3- L'ajout de la division et les réels (***a3.l*** et ***b3.y***). l'ajout du fichier global.h, atof:et le format d'affichage.

Inclure le fichier ***global.h*** dans les 2 programmes:

```
/* le fichier global.h */
#define YYSTYPE double
extern YYSTYPE yylval;
```

Test du + et - entre entiers

Voir le contenu du fichier ***b.tab.h***

```
#define FIN=258
# define NOMBRE=259
#define MOINS=260
#define PLUS=261
#define MOINS=262
typedef int YYSTYPE;
extern YYSTYPE yylval;
```

TP: Calculatrice scientifique (6)


```
/* Ajout de la multiplication b1.y */
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token FIN NOMBRE
%left PLUS MOINS
%left FOIS
%start R
%%
R : E FIN {printf(" Resultat = %d",$1);}
  ;
E : E PLUS T    {$$=$1+$3;}
  | E MOINS T {$$=$1-$3;}
  | T               {$$=$1;}
  ;
T : T FOIS F {$$=$1*$3;}
  | F             {$$=$1;}
  ;
F : NOMBRE
  ;
%%
int yyerror(char *s) {
  printf("%s",s);
  return 1;
}
int main() {
  printf("donner une expression : ");
  yyparse();
  return 0;
}
```

A compiler avec: >bison -d b1.y
en sortie on a : b1.tab.c et b1.tab.h

TP: Calculatrice scientifique (7)


```
/* Ajout de la multiplication a1.l */
%{
#include <math.h>
#include <stdlib.h>
#include "b1.tab.h"
%}
nombre [0-9]+
%%
{nombre} {yylval=atoi(yytext); return NOMBRE;}
"\n"     {return FIN;}
"+"      {return PLUS;}
"-"      {return MOINS;}
"*"      {return FOIS;}
 .       { }
```

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

A compiler avec >flex a1.l
en sortie on a : lex.yy.c

Les autres étapes de compilation:

```
> gcc -c lex.yy.c -o a1.o
> gcc -c b1.tab.c -o b1.o
> gcc -o ab1 a1.o b1.o
```

Lancer l'exécutable: ab1.exe

TP: Calculatrice scientifique (8)


```
/* Ajout des parenthèses b2.y */
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token FIN NOMBRE PO PF
%left PLUS MOINS
%left FOIS
%start R
%%
R : E FIN {printf(" Resultat = %d",$1);}
  ;
E : E PLUS T    {$$=$1+$3;}
  | E MOINS T {$$=$1-$3;}
  | T               {$$=$1;}
  ;
T : T FOIS F {$$=$1*$3;}
  | F             {$$=$1;}
  ;
F : NOMBRE
  | PO E PF       {$$=$2;}
  ;
%%
int yyerror(char *s) {
  printf("%s",s);
  return 1;
}
int main() {
  printf("donner une expression : ");
  yyparse();
  return 0;
}
```

A compiler avec: >bison -d b2.y
en sortie on a : b2.tab.c et b2.tab.h

TP: Calculatrice scientifique (9)


```
/* Ajout des parenthèses a2.l */
%{
#include <math.h>
#include <stdlib.h>
#include "b2.tab.h"
%}
nombre [0-9]+
%%
{nombre} {yylval=atoi(yytext); return NOMBRE;}
"\n"     {return FIN;}
"+"      {return PLUS;}
"-"      {return MOINS;}
"*"      {return FOIS;}
"("      {return PO;}
")"      {return PF;}
 .       { }
```

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

A compiler avec >flex a2.l
en sortie on a : lex.yy.c

Les autres étapes de compilation:

```
> gcc -c lex.yy.c -o a2.o
> gcc -c b2.tab.c -o b2.o
> gcc -o ab2 a2.o b2.o
```

TP: Calculatrice scientifique (10)


```
/* Ajout de la division et les réels: b3.y */
%{
#include <stdio.h>
#include <stdlib.h>
#include "global.h"
%}
%token FIN NOMBRE PO PF
%left PLUS MOINS
%left FOIS DIV
%start R
%%
R : E FIN {printf(" Resultat = %f",$1); return 0;}
   ;
E : E PLUS T    {$$=$1+$3;}
   | E MOINS T {$$=$1-$3;}
   | T          {$$=$1;}
  ;
T : T FOIS F    {$$=$1*$3;}
   | T DIV F    {if ($3==0) printf("division par zero interdite !")
                 else  $$=$1/$3;}
   | F          {$$=$1;}
  ;
```

```
F : NOMBRE
   | PO E PF       {$$=$2;}
   ;
%%
int yyerror(char *s) {
  printf("%s",s);
  return 1;
}
int main() {
  printf("donner une expression : ");
  yyparse();
  return 0;
}
```

```
/* inclusion d'un fichier d'interface global.h */
#define YYSTYPE double
extern YYSTYPE yylval;
```

A compiler avec: >bison -d b3.y

en sortie on a : b3.tab.c et b3.tab.h

TP: Calculatrice scientifique (11)


```
/* Ajout de la division et des réels: a3.l */
%{
#include <math.h>
#include <stdlib.h>
#include "global.h"
#include "b3.tab.h"
%}
nombre [0-9]+(\.[0-9]+)?((e|E)(\+|\-)?[0-9]+)?
%%
{nombre} {yylval=atof(yytext); return NOMBRE;}
"\n"     {return FIN;}
"+"      {return PLUS;}
"-"      {return MOINS;}
"*"      {return FOIS;}
"/"      {return DIV;}
"("      {return PO;}
")"      {return PF;}
 .       { }
```

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

A compiler avec >flex a3.l
en sortie on a : lex.yy.c

Les autres étapes de compilation:

```
> gcc -c lex.yy.c -o a3.o
> gcc -c b3.tab.c -o b3.o
> gcc -o ab3 a3.o b3.o
```

TP: Calculatrice scientifique (12)


```
/* Ajout du plus et du moins unaires: b4.y */
%{
#include <stdio.h>
#include <stdlib.h>
#include "global.h"
%}
%token FIN NOMBRE PO PF
%left PLUS MOINS
%left FOIS DIV
%nonassoc POS NEG
%start R
%%
R : E FIN {printf(" Resultat = %f",$1); return;}
   ;
E : E PLUS T    {$$=$1+$3;}
   | E MOINS T {$$=$1-$3;}
   | T          {$$=$1;}
  ;
T : T FOIS F    {$$=$1*$3;}
   | T DIV F    {if ($3==0) {printf("division par zero interdite !");
                  exit(0);} else  $$=$1/$3;}
   | F          {$$=$1;}
  ;
F : NOMBRE      {$$=$1;}
   | PO E PF    {$$=$2;}
   | MOINS F  %prec NEG {$$= -$2;}
   | PLUS F     %prec POS {$$= +$2;}
%%
int yyerror(void) {
  fprintf(stderr,"erreur de syntaxe\n");
  return 1;
}
int main() {
  printf("donner une expression : ");
  yyparse();
  return 0;
}
```

```
/* inclusion d'un fichier d'interface global.h */
#define YYSTYPE double
extern YYSTYPE yylval;
```

A compiler avec: >bison -d b4.y
en sortie on a : b4.tab.c et b4.tab.h

TP: Calculatrice scientifique (13)


```
/* Ajout du plus et du moins unaires: a4.l */
%{
#include <math.h>
#include <stdlib.h>
#include "global.h"
#include "b4.tab.h"
%}
nombre [0-9]+(\.[0-9]+)?((e|E)(\+|\-)?[0-9]+)?
%%
{nombre} {yylval=atof(yytext); return NOMBRE;}
"\n"     {return FIN;}
"+"      {return PLUS;}
"-"      {return MOINS;}
"*"      {return FOIS;}
"/"      {return DIV;}
"("      {return PO;}
")"      {return PF;}
 .       { }
%%
int yywrap(void) {
  return 1;
}
```

A compiler avec >flex a4.l
en sortie on a : lex.yy.c

Les autres étapes de compilation:

```
> gcc -c lex.yy.c -o a4.o
> gcc -c b4.tab.c -o b4.o
> gcc -o ab4 a4.o b4.o
```

Lancer l'exécutable: ab4.exe

TP: Calculatrice scientifique (14)

/* Autres améliorations: b5.y et a5.l */

Ajout des fonctions mathématiques:

- **absolue,**
- **racine carré**
- **puissance**
- **exponentielle et logarithme**
- **trigonométriques**

TP: Calculatrice scientifique (15)

/* Autres améliorations: a5.l */

Fonctions arithmétiques en C

Le fichier en-tête ***<math.h>*** *déclare des fonctions mathématiques. Tous les paramètres et résultats sont du type* ***double****; les angles sont indiqués en radians.*

double sin(double X) — **sinus de X**

double cos(double X) — **cosinus de X**

double tan(double X) — **tangente de X**

double asin(double X) — **arcsin(X) dans le domaine [-π/2, π/2], x[-1, 1]**

double acos(double X) — **arccos(X) dans le domaine [0, π], x[-1, 1]**

double atan(double X) — **arctan(X) dans le domaine [-π/2, π/2]**

double exp(double X): fonction exponentielle : e^X

double log(double X): logarithme naturel : ln(X), X>0

double log10(double X): logarithme à base 10 : $\log_{10}(X)$, X>0

double pow(double X, double Y): X exposant Y : X^Y

double sqrt(double X): racine carrée de X : $\sqrt{X}$, X≥0

double fabs(double X): valeur absolue de X : |X|

double floor(double X): arrondir en moins: int(X)

double ceil(double X): arrondir en plus

Fonctions arithmétiques en C

*Le fichier en-tête **<math.h>** déclare des fonctions mathématiques. Tous les paramètres et résultats sont du type **double**; les angles sont indiqués en radians.*

Exemple1 : *valeur absolue*

```
#include <stdio.h>
#include <math.h>
main() {
    double nbre = -1234.0;
    printf("Valeur absolue de %lf =  %lf\n", nbre, fabs(nbre));
    return 0;
}
/*-- résultat de l'exécution ---------------------------------
Valeur absolue de -1234.000000 =  1234.000000
------------------------------------------------------------*/
```

Exemple 2 : arrondis

```
#include <stdio.h>
#include <math.h>

main() {
   double nbre = 1234.56;

   printf("ceil de  %lf = %lf\n", nbre, ceil(nbre) );
   printf("floor de %lf = %lf\n", nbre, floor(nbre) );
   return 0;
}
/*-- résultat de l'exécution ----------------------------
ceil de  1234.560000 = 1235.000000
floor de 1234.560000 = 1234.000000
------------------------------------------------------------*/
```

Fonctions arithmétiques en C

printf("%f", 100.123);	==>	100.123000
printf("%12f", 100.123);	==>	__100.123000
printf("%.2f", 100.123);	==>	100.12
printf("%5.0f", 100.123);	==>	__100
printf("%10.3f", 100.123);	==>	___100.123
printf("%.4f", 1.23456);	==>	1.2346

```
float N = 12.1234;

double M = 12.123456789;

long double P = 15.5;
```

printf("%f", N);	==> 12.123400
printf("%f", M);	==> 12.123457
printf("%e", N);	==> 1.212340e+01
printf("%e", M);	==> 1.212346e+01
printf("%Le", P);	==> 1.550000e+01