Générer un analyseur avec *Flex&Bison*

Généralités
Analyse lexicale avec Flex
Analyse syntaxique avec Bison
Association de Flex et Bison

Fabrice Harrouet
École Nationale d'Ingénieurs de Brest
harrouet@enib.fr
http://www.enib.fr/~harrouet/

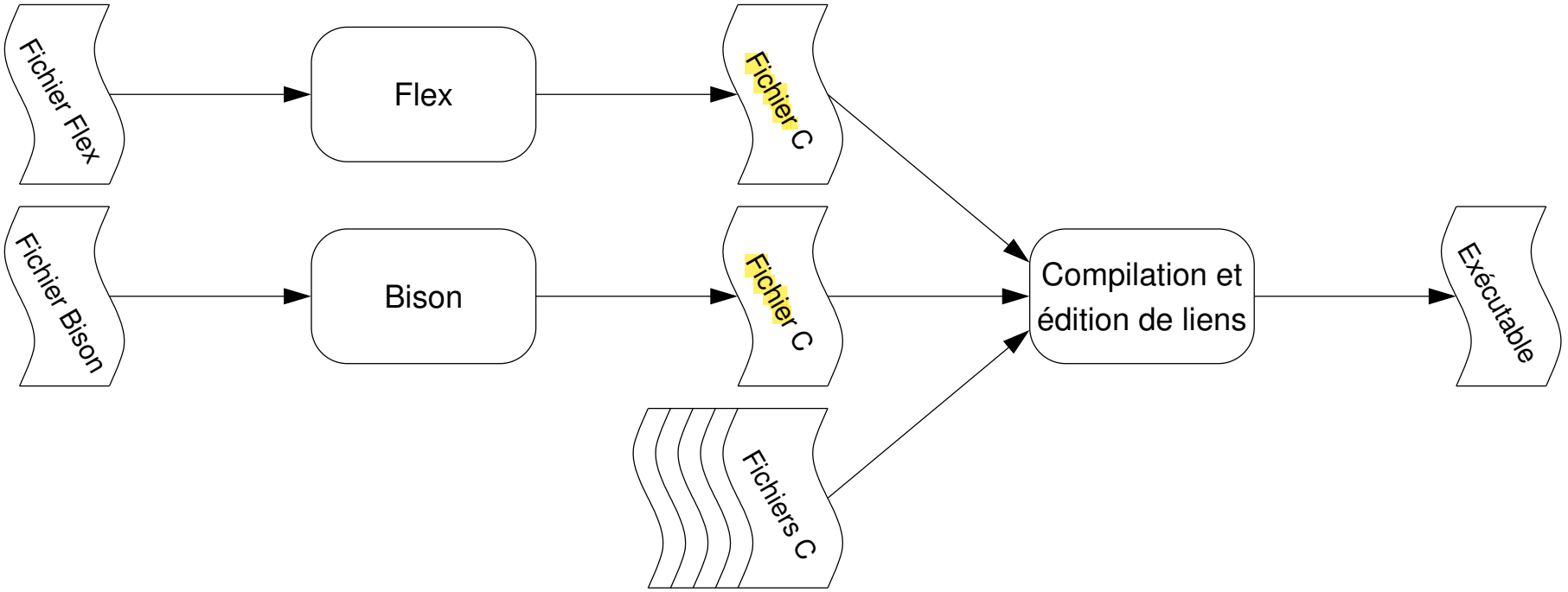
Origine des outils

- ▷ *Lex* **&** *Yacc*
  - ◇ Générateurs d'analyseurs lexicaux/syntaxiques en *C*
  - ◇ $\simeq$ années 1970, laboratoires *Bell*
  - ◇ Outils *UNIX* (*Posix*) standards
  - ◇ + de nombreuses variantes (commerciales ou non)
- ▷ *Flex* **&** *Bison*
  - ◇ Version *GNU* de *Lex*&*Yacc*
  - ◇ *Flex* : *"fast Lex"* *Bison* : jeu de mot sur *Yacc*
  - ◇ Beaucoup de possibilités supplémentaires !
  - ◇ Disponible pour un très grand nombre de plateformes
- ▷ $\exists$ de nombreux outils différents ayant le même propos
  - ◇ *AntLR*, *Spirit* ...
  - ◇ On retrouve des choses issues de *Lex*&*Yacc*

Principe des outils

- Lex/Flex : ***LEXical analyzer***
  - *Reconnaisseur de langages réguliers*
  - Expressions rationnelles → code C d'un analyseur lexical
  - Permet de reconnaître les *mots* d'un langage
- Yacc/Bison : ***Yet Another Compiler Compiler***
  - *Reconnaisseur de langages non contextuels*
  - Grammaire non contextuelle → code C d'un analyseur syntaxique
  - Permet de reconnaître les *phrases* d'un langage

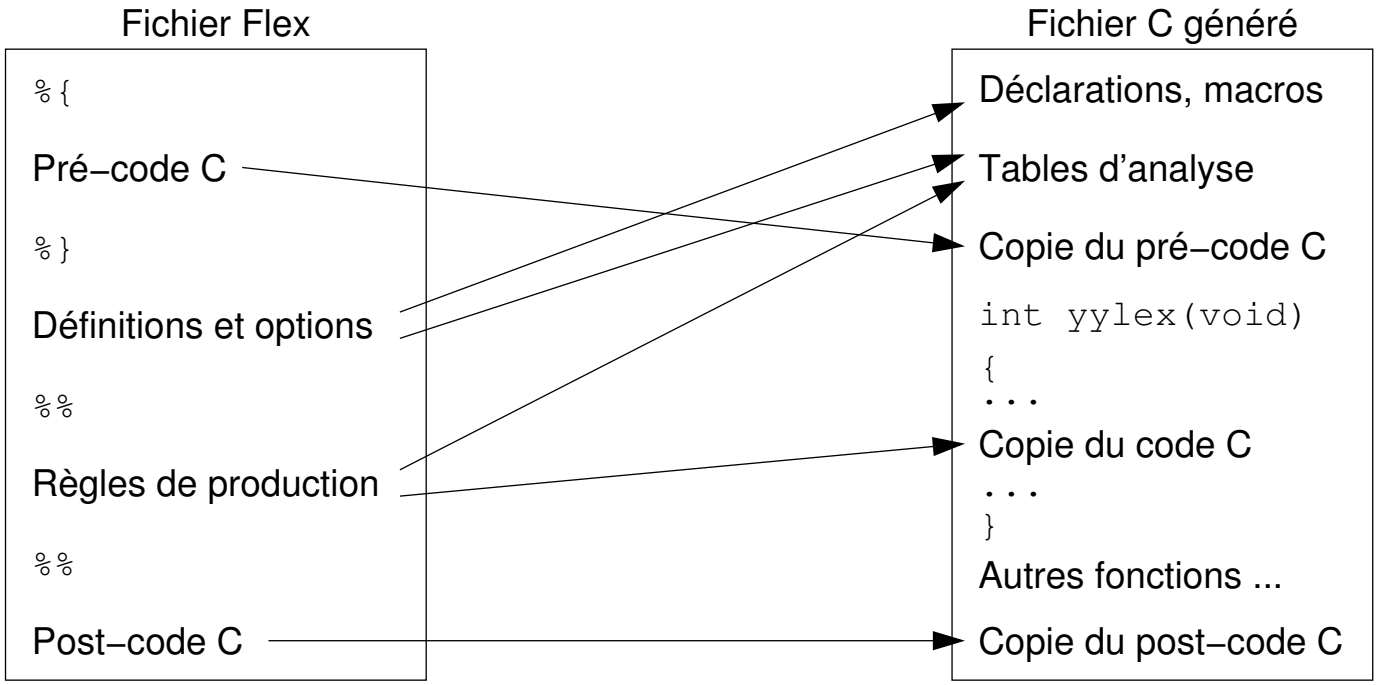
Principe des outils



▷ **Génération d'analyseurs statiques et non dynamiques**

- ◇ Le code *C* produit est spécifique au langage à reconnaitre → efficacité
- ◇ Compilation du code généré comme le reste de l'application
- ◇ Modif du langage → regénération du code des analyseurs

Structure d'un programme *Flex*



- ▷ **Pré/post-code *C* :** du *C* tout à fait classique
- ▷ **Options :** `%quelquechose` pour paramétrer le fonctionnement
- ▷ **Définitions :** Expressions rationnelles auxquelles on attribue un nom
- ▷ **Règles de production :** Associations *ER* $\rightarrow$ code *C* à exécuter

Expressions rationnelles en *Flex*

- **Comprend les *ER POSIX***
  - Voir le cours correspondant
- **Quelques possibilités supplémentaires**
  - Un *atome* peut être une chaîne *C* littérale
    - ex : `"ab"{3}` → `ababab`
  - Les caractères spéciaux du *C* sont reconnus
    - ex : `Hello\tWorld` , `Hello\123World` , `Hello\x2aWorld`
  - La notation { } permet aussi de réutiliser une *ER* nommée
    - ex : `({integer}|{real})?` (`integer` et `real` définies avant)
  - L'*atome* `<<EOF>>` représente la fin de fichier
  - Les *start-conditions* (voir plus loin)

Syntaxe *Flex*

▷ **Définition des *ER* nommées**

◇ Un nom **collé à gauche**, une ou plusieurs espaces, une *ER*

◇ ex : `integer` `[1-9][0-9]*|0`

◇ ex : `indent` `[a-zA-Z_][0-9a-zA-Z_]*`

▷ **Règles de production**

◇ Une *ER* **collée à gauche**, une ou plusieurs espaces, du code *C*

◇ Code sur une seule ligne ou bloc avec { sur la même ligne que l'*ER*

◇ ex : {`integer`} `cerr << "INTEGER" << endl;`

◇ ex : {`indent`}
```
{
cerr << "IDENT" << endl;
}
```

◇ Les commentaires **ne doivent pas** être collés à gauche ($\rightarrow$ *ER*!)

L'analyseur généré

- **La fonction `int yylex(void)`**
  - Extrait des caractères du flux `yyin` (`stdin` par défaut)
  - Confronte les séquences aux *ER* des règles de production
    - Test de haut en bas, **la plus longue** correspondance est retenue
  - Exécute les actions sémantiques (code *C*) associées
    - Variable `yytext` : chaîne de caractères correspondant à l'*ER*
    - Variable `yyleng` : longueur de `yytext`
  - Écrit les *non-correspondances* sur `yyout` (`stdout` par défaut)
  - Jusqu'à la fin de fichier ou un `return` dans les actions *C*
- **La valeur de retour**
  - `0` en cas de fin de fichier
  - Un code numérique (`#define`, `enum`) indiquant l'*ER* reconnue
    → Analyse lexicale

Compteur de lignes, mots et caractères

```
%{
int nbChar=0,nbWord=0,nbLine=0;
%}

 /* doesn't need yywrap() */
%option noyywrap

endOfLine       \n
character       [^ \t\n]

%%
{endOfLine}     { ++nbChar; ++nbLine; }
{character}+    { nbChar+=yyleng; ++nbWord; }
.               { ++nbChar; }
%%

int
main(void)
{
yylex();
fprintf(stderr,"%d\t%d\t%d\n",nbLine,nbWord,nbChar);
return(0);
}
```

```
$
$ flex -oprog.c prog.l
$ gcc -oprog prog.c
$ ./prog < prog.l
24      39      347
$
```

Un analyseur trivial

```
%{
%}
%option noyywrap
integer    [0-9]+
real       [0-9]+\.[0-9]*|\.[0-9]+
ident      [a-zA-Z_][0-9a-zA-Z_]*
%%
{real}     { fprintf(stderr,"REAL [%s]\n",yytext); }
{integer}  { fprintf(stderr,"INTEGER [%s]\n",yytext); }
{ident}    { fprintf(stderr,"IDENT [%s]\n",yytext); }
\n         { fprintf(stderr,"NEW_LINE [%s]\n",yytext); }
.          { fprintf(stderr,"UNKNOWN [%s]\n",yytext); }
%%

int
main(void)
{
yylex();
return(0);
}
```

```
$ cat file.txt
var1=123*45.67;
_attr+=var1;
$ ./prog < file.txt
IDENT [var1]
UNKNOWN [=]
INTEGER [123]
UNKNOWN [*]
REAL [45.67]
UNKNOWN [;]
NEW_LINE [
]
IDENT [_attr]
UNKNOWN [+]
UNKNOWN [=]
IDENT [var1]
UNKNOWN [;]
NEW_LINE [
]
$
```

Un analyseur plus conventionnel — 1/2

```
%{
#include <string.h>
enum {INTEGER=1,REAL,IDENT,NEW_LINE,UNKNOWN};
char globalValue[0x100];
%}

%option noyywrap

integer    [0-9]+
real       [0-9]+\.[0-9]*|\.[0-9]+
ident      [a-zA-Z_][0-9a-zA-Z_]*

%%

{real}     { strcpy(globalValue,yytext); return(REAL); }
{integer}  { strcpy(globalValue,yytext); return(INTEGER); }
{ident}    { strcpy(globalValue,yytext); return(IDENT); }
\n         { strcpy(globalValue,yytext); return(NEW_LINE); }
.          { strcpy(globalValue,yytext); return(UNKNOWN); }

%%
```

```
$ cat file.txt
var1=123*45.67;
_attr+=var1;
$ ./prog < file.txt
IDENT [var1]
UNKNOWN [=]
INTEGER [123]
UNKNOWN [*]
REAL [45.67]
UNKNOWN [;]
NEW_LINE [

]
IDENT [_attr]
UNKNOWN [+]
UNKNOWN [=]
IDENT [var1]
UNKNOWN [;]
NEW_LINE [

]
END_OF_FILE
$
```

Un analyseur plus conventionnel — 2/2

```
int
main(void)
{
int token;
do
  {
  token=yylex();
  switch(token)
    {
    case 0:        fprintf(stderr,"END_OF_FILE\n");               break;
    case INTEGER:  fprintf(stderr,"INTEGER [%s]\n",globalValue);  break;
    case REAL:     fprintf(stderr,"REAL [%s]\n",globalValue);     break;
    case IDENT:    fprintf(stderr,"IDENT [%s]\n",globalValue);    break;
    case NEW_LINE: fprintf(stderr,"NEW_LINE [%s]\n",globalValue); break;
    case UNKNOWN:  fprintf(stderr,"UNKNOWN [%s]\n",globalValue);  break;
    }
  } while(token);
return(0);
}
```

Une calculette à pile — 1/2

```
%{
#include <iostream>
#include <vector>
using namespace std;
enum {VALUE=1,PLUS,MINUS,MULT,DIV};
double val;
%}

%option noyywrap

integer    [0-9]+
real       [0-9]+\.[0-9]*|\.[0-9]+
value      {integer}|{real}

%%
{value}    { sscanf(yytext,"%lf",&val); return(VALUE); }
"+"        { return(PLUS); }
"-"        { return(MINUS); }
"*"        { return(MULT); }
"/"        { return(DIV); }
[ \t\n]+   { /* nothing to be done */ }
.          { cerr << "lexical error: " << yytext << endl; }
%%
```

```
$ cat calc.txt
1 2 +
4 *
5 10 - @/
$ ./prog calc.txt
--> 1
--> 1 2
--> 3
--> 3 4
--> 12
--> 12 5
--> 12 5 10
--> 12 -5
lexical error: @
--> -2.4
--> -2.4
$
```

Une calculette à pile — 2/2

```
int main(int argc,char ** argv)
{
vector<double> s;
int token;
if(argc>1) yyin=fopen(argv[1],"r"); // check result !!!
do
  {
  double x1,x2;
  token=yylex();
  switch(token)
    {
    case VALUE: s.push_back(val); break;
    case PLUS:  x2=s.back(); s.pop_back(); x1=s.back(); s.pop_back();
                s.push_back(x1+x2); break;
    case MINUS: /* ... x1-x2 ... */ break;
    case MULT:  /* ... x1*x2 ... */ break;
    case DIV:   /* ... x1/x2 ... */ break;
    }
  cerr << "-->"; for(size_t i=0;i<s.size();i++) { cerr << " " << s[i]; } cerr << endl;
  } while(token);
return(0);
}
```

Les *start-conditions*

- **Conditions inclusives**
  - `%s name` dans les options, `<name>ER` dans les règles de production
  - Les règles qui commencent par `<name>` ou sans `<>` sont valables quand on est dans la condition `name`
- **Conditions exclusives**
  - `%x name` dans les options, `<name>ER` dans les règles de production
  - Seules les règles qui commencent par `<name>` sont valables quand on est dans la condition `name`
- **Changement de condition**
  - `BEGIN(name)` dans le code *C* place l'analyseur dans la condition `name`
  - `BEGIN(INITIAL)` pour revenir dans l'état de départ (règles sans `<>`)

Reconnaître des chaînes *C* — 1/3

```
%{
#include <iostream>
#include <string>
using namespace std;
enum {STRING=1,INTEGER};
string val;
int nbLines;
%}

%option noyywrap

%x strEnv

integer    [0-9]+

%%
```

```
$
$ cat string.txt
123
"ab\tcd\n456\"hello"
789 "toto
$ ./prog string.txt
INTEGER[123]
STRING[ab       cd
456"hello]
INTEGER[789]
multi-line strings not allowed
STRING[toto]
3 lines
$
```

Reconnaître des chaînes *C* — 2/3

```
{integer}       { val=yytext; return(INTEGER); }
"\""            { val.clear(); BEGIN(strEnv); }
<strEnv>"\""    { BEGIN(INITIAL); return(STRING); }
<strEnv>"\\a"   { val+='\a'; }
<strEnv>"\\b"   { val+='\b'; }
<strEnv>"\\f"   { val+='\f'; }
<strEnv>"\\n"   { val+='\n'; }
<strEnv>"\\r"   { val+='\r'; }
<strEnv>"\\t"   { val+='\t'; }
<strEnv>"\\v"   { val+='\v'; }
<strEnv>"\\\\"  { val+='\\'; }
<strEnv>"\\\""  { val+='\"'; }
<strEnv>"\n"    { cerr << "multi-line strings not allowed" << endl; ++nbLines; }
<strEnv>"\\\n"  { val+='\n'; ++nbLines; } // line cut by \
<strEnv>"\\".   { val+=yytext[1]; }
<strEnv><<EOF>> { BEGIN(INITIAL); return(STRING); }
<strEnv>.       { val+=yytext[0]; }
[ \t]+          { /* nothing to be done */ }
"\n"            { ++nbLines; }
.               { cerr << "lexical error: " << yytext << endl; }
```

Reconnaître des chaînes *C* — 3/3

```
%%
int main(int argc,char ** argv)
{
int token;
if(argc>1) yyin=fopen(argv[1],"r"); // check result !!!
do
  {
  token=yylex();
  switch(token)
    {
    case STRING: cerr << "STRING[" << val << "]" << endl; break;
    case INTEGER: cerr << "INTEGER[" << val << "]" << endl; break;
    }
  } while(token);
cerr << nbLines << " lines" << endl;
return(0);
}
```

Quelques remarques

▷ **Utilisation de variables globales (internes et applicatives)**

◇ Chaque invocation analyse la suite du flux, l'état est sauvé

◇ Plusieurs analyseurs différents → PB à l'édition de liens

◇ Non réentrant → PB si plusieurs analyses simultanées

▷ **L'ordre des règles est important**

◇ Commencer par le plus spécifique, finir par le plus général
ex : {`ident`} doit être pris en compte **après** les mot-clefs

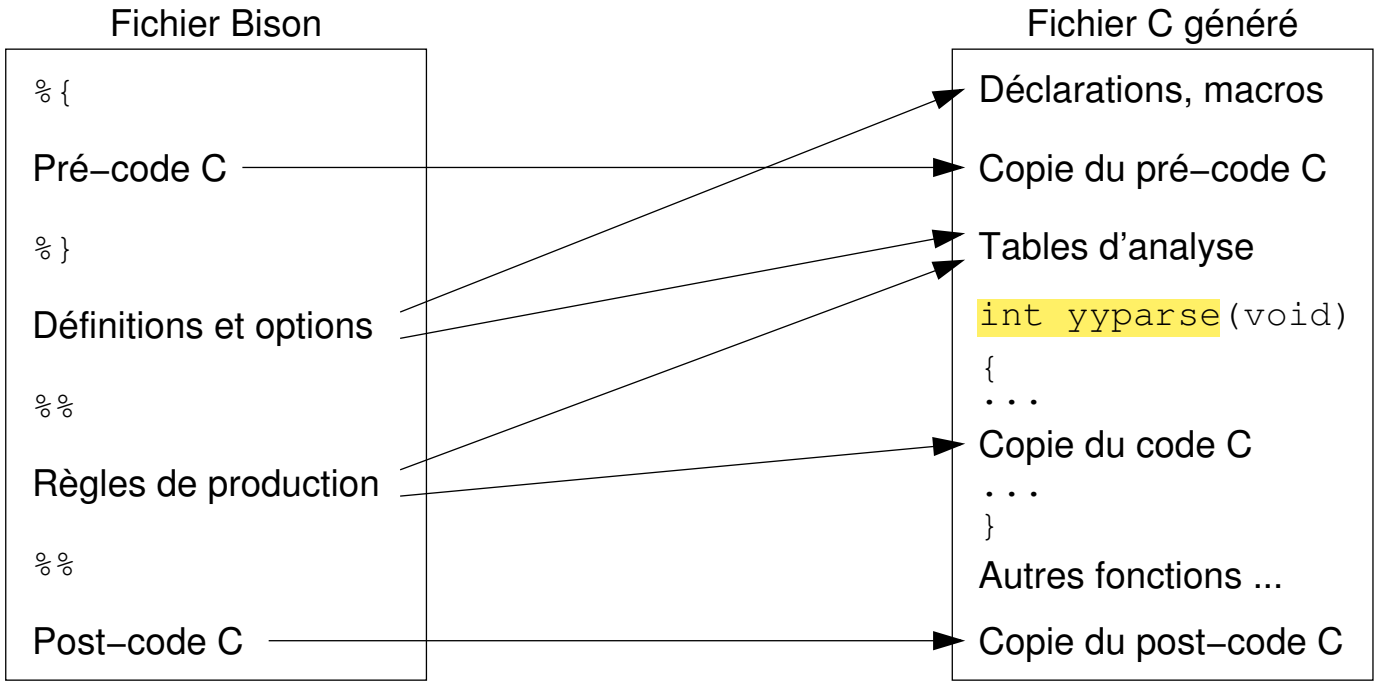
◇ La plus longue correspondance est tout de même preférée
ex : `form` est reconnu comme un {`ident`} même si le mot-clef `for` est testé avant.

▷ **Beaucoup d'autres options, fonctions, macros, variables**

◇ Le minimum est vu ici, très proche du *Lex* originel

◇ Permettent des horreurs, ou de contourner certaines limitations

Structure d'un programme *Bison*



- ▷ **Pré/post-code *C* :** du *C* tout à fait classique
- ▷ **Options :** `%quelquechose` pour paramétrer le fonctionnement
- ▷ **Définitions :** `%autrechose` pour définir les lexèmes, les priorités ...
- ▷ **Règles de production :** Règles de grammaires et code *C* à exécuter

Syntaxe *Bison*

▷ Règles de production

◇
```
non_terminal : sequence_de_symboles { /* code C */ }
             | autre_sequence { /* code C */ }
             | ...
             ;
```

◇ Les symboles sont représentés par des identificateurs

- ○ Terminaux : lexèmes provenant de l'analyse lexicale
- ○ Non-terminaux : symboles internes à l'analyseur syntaxique

◇ Le code *C* est optionnel et est exécuté lorsque la règle est réduite

```
program : START instList END { cerr << "program" << endl; }
;
instList : instList inst
         | inst
;
inst : IDENT ASSIGN expr SEMICOLON { cerr << "inst" << endl; }
;
expr : INTEGER { cerr << "integer expr" << endl; }
     | REAL    { cerr << "real expr" << endl; }
     | IDENT   { cerr << "ident expr" << endl; }
;
```

L'analyseur généré

- **La fonction** `int yyparse(void)`
  - Consomme les lexèmes obtenus par des appels à `yylex()` (à fournir)
    `int yylex(void);`
  - Vérifie (*LALR(1)*) si la séquence de lexèmes permet de réduire l'axiome de la grammaire exprimée
    (`%start non_terminal` dans les définitions)
  - Exécute les actions sémantiques (code *C*) associées aux règles réduites
  - Signale les erreurs à l'aide de la fonction `yyerror()` (à fournir)
    `void yyerror(const char * msg);`
    - Possibilité de récupération sur erreur pour poursuivre l'analyse
  - Jusqu'à ce que l'axiome de la grammaire soit reconnu ou erreur
- **La valeur de retour**
  - `0` si ok, `1` si erreur
  - Résultat de l'analyse complète → une seule invocation

Association *Flex / Bison*

- ▷ *Flex* **fourni les lexèmes à** *Bison*
  - ◇ *Bison* invoque la fonction `yylex()` produite par *Flex*
  - ◇ `yylex()` doit renvoyer des constantes connues de *Bison*
    → `%token IDENT INTEGER ...` dans les définitions de *Bison*
  - ◇ *Bison* genère un `.h` definissant ces constantes (et d'autres choses)
  - ◇ Le pré-code *C* de *Flex* inclu ce `.h`
- ▷ **Étapes de la construction :**
  - ◇ `$ bison -d -oprogY.cpp prog.y`
    → Produit le code *C* `progY.cpp` depuis le fichier *Bison* `prog.y`
    → Option `-d` pour générer le `.h` `progY.hpp`
  - ◇ `$ flex -oprogL.cpp prog.l`
    → Produit le code *C* `progL.cpp` depuis le fichier *Flex* `prog.l`
    → Le pré-code *C* doit inclure `progY.hpp`
  - ◇ `$ g++ -oprog progL.cpp progY.cpp`

Analyse syntaxique simple — 1/4

```
%{ /*-------- prog.l --------*/
extern int lineNumber;          // definie dans prog.y, utilise par notre code pour \n
void yyerror(const char * msg); // definie dans prog.y, utilise par notre code pour .
#include "progY.hpp"            // genere par prog.y --> constantes START, END ...
%}
%option noyywrap
integer         [0-9]+
real            [0-9]+\.[0-9]*|\.[0-9]+
ident           [a-zA-Z_][0-9a-zA-Z_]*
%%
"start"    { return(START); }
"end"      { return(END); }
":="       { return(ASSIGN); }
";"        { return(SEMICOLON); }
{ident}    { return(IDENT); }
{real}     { return(REAL); }
{integer}  { return(INTEGER); }
"\n"       { ++lineNumber; }
[ \t]+     { /* nothing to be done */ }
.          { char msg[0x20]; sprintf(msg,"lexical error <%s>",yytext); yyerror(msg); }
%%
```

Analyse syntaxique simple — 2/4

```
%{ /*-------- prog.y --------*/
#include <stdio.h>
#include <iostream>
using namespace std;

int yylex(void);                  // defini dans progL.cpp, utilise par yyparse()
void yyerror(const char * msg); // defini plus loin, utilise par yyparse()

int lineNumber;                   // notre compteur de lignes
extern FILE * yyin;               // defini dans progL.cpp, utilise par main()
%}

%token START END                  // les lexemes que doit fournir yylex()
%token ASSIGN SEMICOLON
%token IDENT REAL INTEGER

%start program                    // l'axiome de notre grammaire
%%
```

Analyse syntaxique simple — 3/4

```
program : START instList END { cerr << "program" << endl; }
;
instList : instList inst
         | inst
;
inst : IDENT ASSIGN expr SEMICOLON { cerr << "inst" << endl; }
;
expr : INTEGER { cerr << "integer expr" << endl; }
     | REAL    { cerr << "real expr" << endl; }
     | IDENT   { cerr << "ident expr" << endl; }
;
%%
void yyerror(const char * msg)
{
cerr << "line " << lineNumber << ": " << msg << endl;
}
int main(int argc,char ** argv)
{
if(argc>1) yyin=fopen(argv[1],"r"); // check result !!!
lineNumber=1;
if(!yyparse()) cerr << "Success" << endl;
return(0);
}
```

Analyse syntaxique simple — 4/4

```
$ bison -d -oprogY.cpp prog.y
$ flex -oprogL.cpp prog.l
$ g++ -oprog progL.cpp progY.cpp
$
$ cat file1.txt
start
a := 1 ;
b := 2.3 ;
c := a ;
end
$ ./prog file1.txt
integer expr
inst
real expr
inst
ident expr
inst
program
Success
$
```

```
$ cat file2.txt
start
a := -1 ;
b := 2,3 ;
c := a ;
end
$ ./prog file2.txt
line 2: lexical error <->
integer expr
inst
integer expr
line 3: lexical error <,>
line 3: parse error
$
```

```
$ cat file3.txt
start
a := 1 ;
b := 2.3 ;
c := a ;
end
and then ...
$ ./prog file3.txt
integer expr
inst
real expr
inst
ident expr
inst
program
line 6: parse error
$
```

Valeurs associées aux symboles

- **Chaque symbole peut contenir une valeur**
  - Au-delà de l'analyse syntaxique → **analyse sémantique**
  - `%union{ ... }` dans les définitions de *Bison* (union *C*)
- **Valeur d'un symbole → un seul champ de l'union**
  - `%type<nom_champ> symboles` dans les définitions de *Bison*
- **Accès aux valeurs depuis** *Bison*
  - Dans le code associé à `non_terminal : symboles`
  - `$$` valeur associée au non-terminal de la partie gauche (écriture)
  - `$1` valeur associée au premier symbole de la partie droite (lecture)
  - `$2` , `$3` ...pour les autres symboles de la partie droite
  - Si pas de code *C* → équivalent de `{$$=$1;}` par défaut
    - Attention aux `%type` → *warnings*
    - Inhiber avec du code vide {}

Initialiser la valeur associée à un lexème

- **Variable globale `yylval` de type `%union`**
  - Déclarée dans le `.h` généré par *Bison*
  - `yylex()` doit initialiser cette variable
  - `yyparse()` récupere cette valeur juste après l'appel à `yylex()`
- **Choix du champ de l'union**
  - `yylex()` ne connait pas les `%type`
  - Initialiser dans `yylval` le champ indiqué par le `%type` du lexème
    - Aucune vérification de la cohérence par les outils !
    - Responsabilité du programmeur

Analyse syntaxique avec passage de valeurs — 1/2

```
%{ /*-------- prog.l --------*/
/* idem ‘‘Analyse syntaxique simple’’ */
%}
%option noyywrap
integer    [0-9]+
real       [0-9]+\.[0-9]*|\.[0-9]+
ident      [a-zA-Z_][0-9a-zA-Z_]*
%%
"start"    { return(START); }
"end"      { return(END); }
":="       { return(ASSIGN); }
";"        { return(SEMICOLON); }
{ident}    { sprintf(yylval.str,"%s",yytext);
             return(IDENT); } // %type<str> IDENT
{real}     { sscanf(yytext,"%lf",&yylval.real);
             return(REAL); } // %type<real> REAL
{integer}  { sscanf(yytext,"%d",&yylval.integer);
             return(INTEGER); } // %type<integer> INTEGER
"\n"       { ++lineNumber; }
[ \t]+     { /* nothing to be done */ }
.          { char msg[0x20]; sprintf(msg,"lexical error <%s>",yytext); yyerror(msg); }
%%
```

```
$ bison -d -oprogY.cpp prog.y
$ flex -oprogL.cpp prog.l
$ g++ -oprog progL.cpp progY.cpp
   $ cat file.txt
   start
   a := 1 ;
   b := 2.3 ;
   c := a ;
   end
   $ ./prog file.txt
   integer 1
   assign a with i:1
   real 2.3
   assign b with r:2.3
   ident a
   assign c with s:a
   program
   Success
   $
```

Analyse syntaxique avec passage de valeurs — 2/2

```
%{ /*-------- prog.y --------*/
/* idem ‘‘Analyse syntaxique simple’’ */
%}
%token START END ASSIGN SEMICOLON IDENT REAL INTEGER
%union { char str[0x100]; double real; int integer; }
%type<str> IDENT expr
%type<real> REAL
%type<integer> INTEGER
%start program
%%
program : START instList END { cerr << "program" << endl; }
;
instList : instList inst
         | inst
;
inst : IDENT ASSIGN expr SEMICOLON { cerr << "assign " << $1 << " with " << $3 << endl; }
;
expr : INTEGER { cerr << "integer " << $1 << endl; sprintf($$,"i:%d",$1); }
     | REAL    { cerr << "real " << $1 << endl; sprintf($$,"r:%g",$1); }
     | IDENT   { cerr << "ident " << $1 << endl; sprintf($$,"s:%s",$1); }
;
%%
/* idem ‘‘Analyse syntaxique simple’’ */
```

Une calculette à pile ... sans pile ! — 1/2

```
%{ /*-------- prog.l --------*/
/* idem ‘‘Analyse syntaxique simple’’ */
%}
%option noyywrap
integer    [0-9]+
real       [0-9]+\.[0-9]*|\.[0-9]+
value      {integer}|{real}
%%
{value}    {
           sscanf(yytext,"%lf",&yylval.val);
           return(VALUE);
           }
"+"        { return(PLUS); }
"-"        { return(MINUS); }
"*"        { return(MULT); }
"/"        { return(DIV); }
"\n"       { ++lineNumber; }
[ \t]+     { /* nothing to be done */ }
.          { char msg[0x20]; sprintf(msg,"lexical error <%s>",yytext);
           yyerror(msg);
           }
%%
```

```
$ bison -d -oprogY.cpp prog.y
$ flex -oprogL.cpp prog.l
$ g++ -oprog progL.cpp progY.cpp
$ echo "1 2 + 4 * 5 10 - /" | ./prog
value=1
value=2
1+2=3
value=4
3*4=12
value=5
value=10
5-10=-5
12/-5=-2.4
Result: -2.4
Success
$
```

Une calculette à pile ... sans pile ! — 2/2

```
%{ /*-------- prog.y --------*/
/* idem ‘‘Analyse syntaxique simple’’ */
%}
%token VALUE PLUS MINUS MULT DIV
%union
  {
  double val;
  }
%type<val> VALUE expr
%start calc
%%
calc : expr { cerr << "Result: " << $1 << endl; }
;
expr : VALUE           { $$=$1; cerr << "value=" << $1 << endl;}
     | expr expr PLUS  { $$=$1+$2; cerr << $1 << "+" << $2 << "=" << $$ << endl; }
     | expr expr MINUS { $$=$1-$2; cerr << $1 << "-" << $2 << "=" << $$ << endl; }
     | expr expr MULT  { $$=$1*$2; cerr << $1 << "*" << $2 << "=" << $$ << endl; }
     | expr expr DIV   { $$=$1/$2; cerr << $1 << "/" << $2 << "=" << $$ << endl; }
;
%%
/* idem ‘‘Analyse syntaxique simple’’ */
```

Les conflits de la grammaire

- **Conflit réduire/réduire**
  - À un certain stade de l'analyse, le lexème suivant ne permet pas de dire s'il faut réduire une règle ou bien une autre
    → modifier la grammaire pour lever l'ambiguité
  - Doit toujours être éliminé !
- **Conflit décaler/réduire**
  - À un certain stade de l'analyse, le lexème suivant ne permet pas de dire s'il faut réduire une règle ou bien continuer à décaler pour reconnaitre une règle plus longue
    → modifier la grammaire pour lever l'ambiguité
  - Doit toujours être éliminé (sauf circonstances exceptionnelles) !
    → *Bison* fait le choix du décalage !
- **L'analyseur regarde toujours un lexème à l'avance**
  - Permet de retarder le choix

Exemples triviaux de conflits réduire/réduire

```
%token A1 B1
%start g1
%%
g1 : e1 B1
   | f1 B1
;
e1 : A1
;
f1 : A1
;
```

```
%token A2 B2 C2 D2
%start g2
%%
g2 : e2 B2 C2
   | f2 B2 D2
;
e2 : A2
;
f2 : A2
;
```

```
%token A3 B3 C3
%start g3
%%
g3 : e3 B3
   | f3 C3
;
e3 : A3
;
f3 : A3
;
```

- ▷ **Cas 1 : réduire `A1` en `e1` ou `f1` ?**
  - ◇ Après `A1` : `e1 -> A1 .` et `f1 -> A1 .`
  - ◇ Le lexème suivant (`B1`) ne permet pas de choisir !
- ▷ **Cas 2 : réduire `A2` en `e2` ou `f2` ?**
  - ◇ Idem, il faudrait repousser le choix à deux lexèmes (`C2` ou `D2`)
- ▷ **Cas 3 : pas de conflit !**
  - ◇ Le lexème suivant `A3` (`B3` ou `C3`) permet de choisir !

Exemples triviaux de conflits décaler/réduire

```
%token A1 B1
%start g1
%%
g1 : e1 B1
   | A1 B1
;
e1 : A1
;
```

```
%token A2 B2 C2 D2
%start g2
%%
g2 : e2 B2 C2
   | A2 B2 D2
;
e2 : A2
;
```

```
%token A3 B3 C3
%start g3
%%
g3 : e3 B3
   | A3 C3
;
e3 : A3
;
```

- ▷ **Cas 1 : réduire** `A1` **en** `e1` **ou décaler pour** `g1 -> A1 B1` **?**
  - ◇ Après `A1` : `g1 -> A1 . B1` et `e1 -> A1 .`
  - ◇ Le lexème suivant (`B1`) ne permet pas de choisir !
- ▷ **Cas 2 : réduire** `A2` **en** `e2` **ou décaler pour** `g2 -> A2 B2 D2` **?**
  - ◇ Idem, il faudrait repousser le choix à deux lexèmes (`C2` ou `D2`)
- ▷ **Cas 3 : pas de conflit !**
  - ◇ Le lexème suivant `A3` (`B3` ou `C3`) permet de choisir !

Exemples de conflits classiques

```
%token A1 B1
%start list
%%
list : list elem
     | /* empty */
;
elem : A1
     | B1
     | /* empty */
;
```

```
instr : ifInstr SEMICOLON
      | LB instrList RB
      | ...
;
instrList : instrList instr
          | /* empty */
;
ifInstr : IF LP cond RP instr
        | IF LP cond RP instr ELSE instr
;
```

▷ **Zéro ou plusieurs éléments**

◇ Deux fois le cas vide (réduire/réduire) → retirer `elem -> ` ε

▷ **Décaler/réduire dans** `if/else`

◇ `if(c1)if(c2)f();else g();` → `if(c1){if(c2)f();}else g();` ?

→ `if(c1){if(c2)f();else g();}` ?

◇ Le décalage par défaut est correct ! → on ne change rien !

Repérer les conflits

▷ **Générer une description du procédé d'analyse**

- ◇ `bison -v -oprogY.cpp progY.l` produit `progY.output` (`-v`)
- ◇ Début du fichier : conflits avec numéro de l'état où ils se situent
- ◇ Juste après : récapitulatif des règles, terminaux et non terminaux
- ◇ Toute la suite : les différents états de l'analyseur
  - ○ État d'avancement dans la reconnaissance des règles
  - ○ Actions possibles (décaler, réduire) selon les symboles reconnus

```
State 1 contains 1 shift/reduce conflict.
...
state 1
    g1  ->  A1 . B1   (rule 2)
    e1  ->  A1 .   (rule 3)

    B1          shift, and go to state 3
    B1          [reduce using rule 3 (e1)]
    $default    reduce using rule 3 (e1)
...
```

Associativité et précédence

- **Permettent d'éliminer facilement les conflits**
  - `%left`, `%right` ou `%nonassoc` dans les définitions de *Bison*
  - Indique l'associativité du ou des lexèmes indiqués sur la ligne
  - Ces lignes sont triées selon les priorités croissantes
  - `%prec` pour substituer une priorité dans la grammaire
  - Les conflits résolus sont tout de même reportés dans le `.output`
  - ex : expressions arithmétiques et logiques

```
%left OR
%left AND
%right NOT
%nonassoc LT GT LE GE EQ NEQ
%left PLUS MINUS
%left MULT DIV MODULO
%right UMINUS
```

Expressions arithmétiques et logiques — 1/3

```
%token OR AND NOT
%token LT GT LE GE EQ NEQ
%token PLUS MINUS
%token MULT DIV MODULO
%token VALUE LP RP

%start expr
%%
```

```
expr : expr OR expr
     | expr AND expr
     | NOT expr
     | expr LT expr
     | expr GT expr
     | expr LE expr
     | expr GE expr
     | expr EQ expr
     | expr NEQ expr
     | expr PLUS expr
     | expr MINUS expr // - binaire
     | expr MULT expr
     | expr DIV expr
     | expr MODULO expr
     | MINUS expr      // - unaire
     | VALUE
     | LP expr RP
;
```

```
$ bison -v prog.y
prog.y contains 195 shift/reduce conflicts.
$
```

Expressions arithmétiques et logiques — 2/3

```
%token PLUS MINUS
%token MULT DIV MODULO
%token VALUE LP RP

%start expr
%%
expr : expr PLUS prod              // associativite a gauche
     | expr MINUS prod             // operations les moins prioritaires
     | prod
;
prod : prod MULT factor            // associativite a gauche
     | prod DIV factor             // operations un peu plus prioritaires
     | prod MODULO factor
     | factor
;
factor : VALUE                     // constructions les plus prioritaires
       | LP expr RP
;


$ bison -v prog.y                 --> pas de conflits mais demarche tres fastidieuse !!!
$
```

Expressions arithmétiques et logiques — 3/3

```
%token OR AND NOT
%token LT GT LE GE EQ NEQ
%token PLUS MINUS
%token MULT DIV MODULO
%token VALUE LP RP

%left OR
%left AND
%right NOT
%nonassoc LT GT LE GE EQ NEQ
%left PLUS MINUS
%left MULT DIV MODULO
%right UMINUS

%start expr
%%
```

```
expr : expr OR expr
     | expr AND expr
     | NOT expr
     | expr LT expr
     | expr GT expr
     | expr LE expr
     | expr GE expr
     | expr EQ expr
     | expr NEQ expr
     | expr PLUS expr
     | expr MINUS expr  // - binaire
     | expr MULT expr
     | expr DIV expr
     | expr MODULO expr        // yylex() ne renvoie
     | MINUS expr %prec UMINUS // jamais UMINUS !!!
     | VALUE                   // --> subst. de prio.
     | LP expr RP
     ;
```

```
$ bison -v prog.y        --> conflits resolus facilement par les %left %right %nonassoc
$
```

Récupération sur erreurs

▷ **Entrée non-conforme à la grammaire**

◇ Appel automatique de `yyerror("parse error")`

◇ `yyparse()` se termine en renvoyant `1`

◇ La suite du flux d'entrée n'est pas analysée !

▷ **Le symbole spécial** `error`

◇ Permet d'enrichir la grammaire

◇ Lorsqu'aucune règle ne permet de poursuivre l'analyse
→ réduire celle qui utilise `error`

◇ Synchronisation sur le lexème qui suit le symbole `error`
→ consommation des lexèmes intermédiaires

◇ Utilisation de la macro `yyerrok;` dans le code *C*
→ évite la sortie de `yyparse()` après le message

```
assign : IDENT ASSIGN expr SEMICOLON { /* ... */ }
       | IDENT ASSIGN error SEMICOLON { yyerrok; }
;
```

Démarche usuelle pour la mise en œuvre

- **Mise au point de l'analyse lexical**
  - Fichier *Flex* complet : reconnaître tous les lexèmes
  - Fichier *Bison* avec une grammaire dégénérée
    - Accepter tous les lexèmes dans un ordre quelconque
    - Actions sémantiques : afficher le lexème reçu
- **Mise au point de l'analyse syntaxique**
  - Grammaire complète du langage à reconnaître
  - Actions sémantiques : afficher les constructions reconnues
- **Mise en place des actions sémantiques**
  - Construire/calculer les données utiles pour l'application
- **Envisager la récupération sur erreurs**
  - Rend la grammaire plus complexe ! $\rightarrow$ À effectuer en dernier lieu