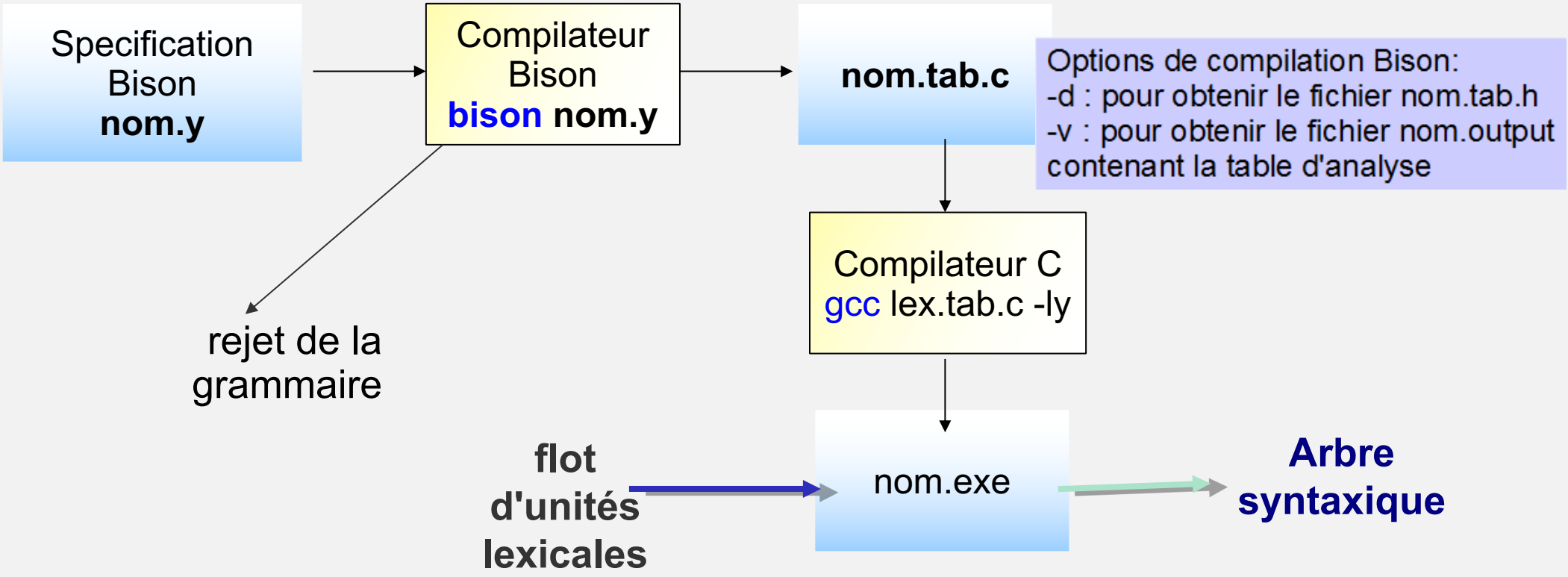
Analyseur syntaxique

L'outil Bison

L'outil Bison

- Bison est un générateur automatique d'analyseurs syntaxiques
- Il accepte en entrée les langages décrits par les grammaires non-contextuelle (**liste des productions**) et **règle de suppression d'ambiguité**
- Il produit un programme écrit dans un langage de haut niveau : analyseur syntaxique
- Il fait une analyse ascendante : utilise le modèle **décallages/réductions**



Spécification en Bison

Un programme Bison consiste en trois parties :

```
%{
    déclaration en C de variables, constantes, inclusions de fichiers, …
%}
    déclaration des unités lexicales utilisées
    déclaration de priorités (règle de suppression d'ambiguité) et de types

%%

    règles de production (avec éventuellement des actions sémantiques )

%%

    bloc principal et fonctions auxiliaires
```

Spécification en Bison

- Les symboles **non-terminaux** : suite d'une ou plusieurs lettres majuscules et/ou miniscules

```
non-terminal :   prod1
               |  prod2
                 …
               |  prod3
               ;
```

- Les symboles **terminaux** sont :
  - **Les unités lexiacles** : déclarer dans la première partie par : ***%token nom_de_UL***

    %token MC_Sinon

    %token NOMBRE
  - **Les caractères** entre quotes : ***'+', 'a',…***
- La partie trois doit contenir une fonction ***yylex()*** qui retourne les *UL* et les caractères reconnus. On peut :
  - Soit écrire cette fonction
  - Soit utiliser la fonction produite par un compilateur flex !!! (A suivre...)

Actions sémantiques

- **Les actions sémantiques** sont des instructions en C insérées dans les règles de production **S : E *PLUS E { printf(" %d", $1+$3;}* ;**
  - Elles sont exécutées chaque fois qu'il y a réduction par la production associée
  - ***S : A {printf("reduction par A"); } T {printf("reduction par T"); } 'a'***
- Les attributs peuvent être utilisés dans les actions sémantiques
  - Le symbole ***$$*** référence la valeur de l'attribut associé au non-terminal de la partie gauche
  - ***$i*** référence la valeur associée au **i_ème symmbole** (action sémantique)
  - ***expr : expr '+' expr {$$=$1+$3;};***

S : E / *E { if ($3==0) printf(" erreur : erreur division par zero "); } ;*

Variable *yyval*

- L'attribut d'un symbole terminal est la valeur contenue dans la variable globale ***yylval***
  - Il faudra donc penser à affecter correctement *yylval* lors de l'analyse lexicale :
    - *[0-9]+ {yylval=atoi(yytext); return NOMBRE;}*
- ***yylval*** est de type int par défaut.
  - On peut changer ce type par la déclaration d'une union dans la partie 1 (just avant les tokens)

```
%union{
    int entier;
    double reel;
    char * chaine;
}
```

```
%token <entier> NOMBRE
%token <chaine> ID CHAINE COMMENT
On peut changer le type des non-terminaux de la même façon :
%token <entier> S
%token <chaine> expr
```

Il faut préciser le champ utilisé de **yylval** au niveau analyseur lexical :
{nombre} {**yylval.entier**=atoi(yytext); return NOMBRE; }

Fonctions prédéfinies en Bison

int yyparse()	Fonction qui lance l'analyseur syntaxique
YYACCEPT	Instruction permettant de stopper l'analyseur syntaxique
yylerror(char *)	Fonction appelée chaque fois que l'analyseur est confronté à une erreur
int main()	Fonction qui contient entre autre l'appel à ***yylex()***
%start non-terminal	Action pour désignier l'axiome. Par défaut, c'est le premier non-terminal décrit dans les règles de production
...	

Conflits
shift/reduce & reduce/reduce

- Bison résoud les conflis 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
- On peut modifier cette façon de résoudre les conflits en donnant des associativités (droite ou gauche) et des priorités aux symboles terminaux :
  - ***%left '+' '-'*** : indique que + et – ont la **même priorité** et sont **associatifs à gauche**
  - **%left '*' '/'**
  - ***%right '^'*** : indique que **^ est associatif à droite**
    - + et – ont la plus faible priorité et ^ a la priorité la plus élevée
- La priorité d'une production est celle de **son terminal le plus à droite**
  - On peut modifier cet ordre de priorité en faisant suivre la production de la clause **: *%prec terminal*** (voir exemple)

Exemple 1

- L'exemple suivant reconnait les mots de la forme $\omega\, c\omega'$ où ω et ω' sont des mots sur $\{a,b\}$

```
%{
        void yyerror(char const *);
%}
%%
E : S '$'              {     printf (" Mot Accepté ");          YYACCEPT;}
S : 'a' S 'a'
        | 'b' S 'b'
        | 'c'
        ;
%%
int main(){
                yyparse();
 }
// gestion des erreurs
void yyerror(char const *s){
                                printf("erreur %s", s);
                                }
int yylex(){
        char ch=getchar();
        if(ch=='a' ||ch=='b' ||ch=='c' || ch=='$') { return(ch);}
                        else printf(" ERREUR : caractere non reconnu : %c ", ch);
 }
```

Démo 1

Tester ce programme :

Les étapes :

- Installer Bison (c\gnuwin32\...)
- Saisir le fichier nomfichier.y
- Compiler nomfichier.y
- Compiler nomfichier.tab.y.c
- Tester les chaines suivantes :
  - c$
  - aca$
  - abcba$
  - ...

TP : analyse syntaxique

Exemple2

```
%{
#include<ctype.h>
#include<stdlib.h>
int yylval;
static void yyerror(char const *s);
int yylex();
%}
%token CHIFFRE

%%
ligne : expr '\n'     { printf ("  = %d \n ", $1);      YYACCEPT;};
expr : expr '+' terme {$$ = $1 + $3;}
       | terme
       ;
terme : terme '*' facteur {$$ = $1 *  $3;}
       | facteur
       ;
facteur: '(' expr ')' {$$ = $2;}
       |      CHIFFRE      {$$ = $1;}
       ;
%%
```

```
int main(){
            yyparse();
           }
void yyerror(char const *s){
                                printf("erreur %s", s);
                            }
int yylex(){
            int c;
            c=getchar();
            if(isdigit(c)) {
                                yylval=c-'0';
                                return CHIFFRE;
            }
           return(c);
           }
```

Exemple2

```
%{
#include<ctype.h>
#include<stdlib.h>
int yylval;
static void yyerror(char const *s);
int yylex();
%}
%token CHIFFRE

%%
ligne : expr '\n'     { printf ("  = %d \n ", $1);
expr : expr '+' terme {$$ = $1 + $3;}
       | terme
       ;
terme : terme '*' facteur {$$ = $1 *  $3;}
       | facteur
       ;
facteur: '(' expr ')' {$$ = $2;}
       |      CHIFFRE      {$$ = $1;}
       ;
%%
```

Tester les chaines suivantes :
3+6
3+5*(2+2)
3(2*1)
(3+3)+3

Démo 2

```
    YYACCEPT;};

int main(){
            yyparse();
           }
void yyerror(char const *s){
                                printf("erreur %s", s);
                            }
int yylex(){
            int c;
            c=getchar();
            if(isdigit(c)) {
                                yylval=c-'0';
                                return CHIFFRE;
            }
          return(c);
          }
```

Exercice

Écrire la spécification bison de la grammaire suivante :

E → E + E | E - E | E / E | E * E | F

F → (E) | nb

Tester les mots suivants :
1+3-4
(2+3)*2
3(1*2+3)-5
2+3*3/2
...