Compilation

Yacc / Bison

SI4 — 2018-2019

Erick Gallesio

Yacc / Bison

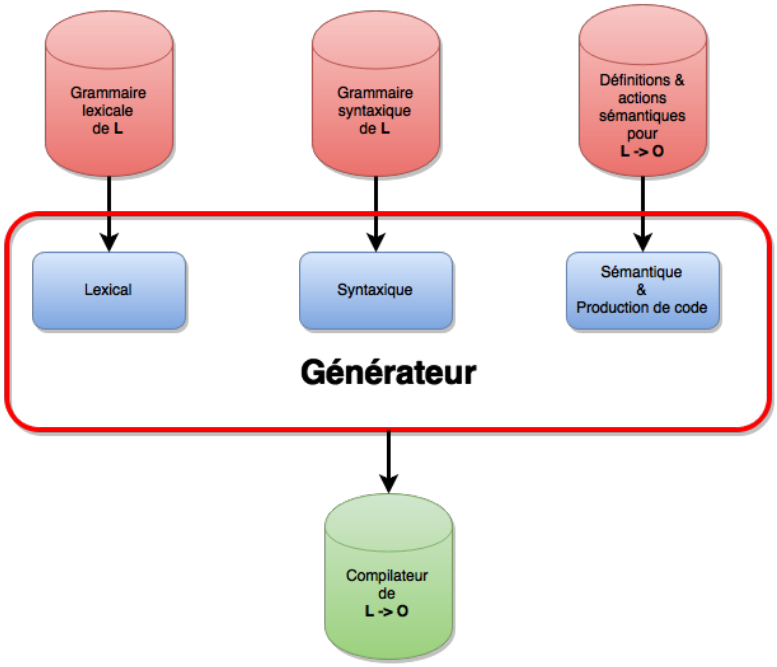
Yacc est un outil créé en 1974 par S.C Johnson:

- présent dans les premières versions de Unix
- permet de construire automatiquement des analyseurs syntaxiques
- peut être couplé facilement avec un analyseur lexical construit par **Lex/Flex**
- génération se fait à partir de grammaires sous-classes des LR(1)
- analyse **LALR** (Look-Ahead Left Recursive)
- très bonnes performances des analyseurs produits
- mais *pas terrible* pour la récupération des erreurs
- peut produire du code C ou C++ (mais des variantes de yacc existent pour beaucoup d'autres langages aussi)

Bison est la variante GNU de Yacc.

- C'est la version que l'on utilisera en TD.

Métacompilation (*compiler compiler*)



Principe de fonctionnement d'un compilateur de compilateurs

Fichier yacc

Un fichier **yacc** est assez semblable à un à un fichier **lex**:

- constitué de 3 parties
- les parties sont séparées par des `%%`

```
[ définitions ]
%%
[ règles ]
%%
[ fonctions ]
```

La partie *définitions* peut inclure du code C:

```
%token KWHILE KIF  // des définitions Yacc
%left '+' '-'
%{                 // code  qui ira au début de l'analyseur produit
   #include <stdio.h>
   void myerror(...) { ... }
%}
```

Analyse expression — version 0 (1/3)

On veut reconnaître la grammaire $\mathbf{ETF_0}$:

```
E →  E + T | T
T →  T * F | F
F → (E) | digit
```

Pour l'instant:

- on suppose que les nombres sont compris entre 0 et 9
- les espaces et les *newlines* ne sont pas significatifs

Conventions **Yacc**:

- l'analyseur lexical doit s'appeler `yylex()`
- lorsqu'une erreur se produit, la fonction `yyerror(str)` est appelée avec un message en paramètre (en général `"Syntax error"`)
- l'analyseur syntaxique produit s'appelle `yyparse()`

Analyse expression — version 0 (2/3)

Une version basique de la fonction `yylex` pourrait être:

```
int yylex(void) {
  int c;

  do
     c = getchar();
  while (c == ' ' || c == '\n' || c == '\t');
  return c;
}
```

De même, la fonction `yyerror` pourrait être:

```
void yyerror(char *msg) {
  fprintf(stderr, "ERROR: %s\n", msg);
}
```

Analyse expression — version 0 (3/3)

L'analyseur complet (dans `etf0.y`):

```
%{
    #include <stdio.h>
    int yylex(void) { ... }
    void yyerror(char *msg) { ... }
%}
%%
expr:       expr '+' term
    |       term ;

term:       term '*' factor
    |       factor ;

factor:     '(' expr ')'
    |       digit ;

digit:      '0' | '1' |'2' |'3' |'4' |'5' |'6' |'7' |'8' |'9' ;
%%
int main() { return yyparse(); }
```

Pour construire l'analyseur (qui reconnaît ETF_0 mais n'évalue rien):

```
$ bison -o etf0.c etf0.y
$ gcc -std=gnu99 -o etf0 etf0.c
```

Amélioration de l'analyseur v.0

La version actuelle de notre analyseur

- affiche un message d'erreur si la phase entrée est incorrecte
- n'affiche rien dans le cas contraire.

Yacc

- permet d'ajouter du code qui est exécuté lorsqu'une partie droite de règle est reconnue.
- Ce code est mis entre accolades.

On peut donc modifier notre fichier `etf0.y` en:

```
etf0:       expr        { printf("It's OK\n"); }
            ;

expr:       expr '+' term
    ......
```

Ainsi, lorsque on arrive à la réduction `etf0` , le message est affiché.

Analyse expression — version 1 (1/3)

Dans cette version on améliore notre analyseur lexical pour accepter des nombres quelconques.

Pour *yacc*, un lexème peut être représenté par:

- un caractère (donc une valeur dans **[0..255]** en C) pour les lexèmes simples
- une valeur **≥ 256** lorsque pour les lexèmes complexes (`'<='` , `'=='` , mots clés, …)

Lorsque l'analyseur rencontre un lexème complexe, il peut:

- renvoyer un code pour le lexème (qui est déclaré avec la directive **`%token`** dans le fichier yacc
- renseigner la variable globale **`yylval`** (par défaut de type entier) avec des informations sur le lexème trouvé.

Par exemple pour un entier, on peut renvoyer le token `INTEGER` et mettre la valeur de l'entier que l'on vient de lire dans `yylval`.

Analyse expression — version 1 (2/3)

Une nouvelle version de la fonction `yylex` :

```
int yylex(void) {
  int c;

  // Sauter les espaces
  while (isspace(c = getchar())) {
  }

  if (isdigit(c)) {
    ungetc(c, stdin); // car on a déjà lu le 1er caractère du nbre
    scanf("%d", &yylval);
    return INTEGER;
  }
  return c;
}
```

Pour que `INTEGER` soit défini, on doit ajouter la définition

```
%token INTEGER
```

en tête du fichier *yacc*.

Analyse expression — version 1 (3/3)

Le fichier etf1.y implémentant les expressions avec des nombres sera donc

```
%{
    #include <stdio.h>
    #include <ctype.h>
    int yylex(void);
    void yyerror(char *msg);
%}

%token INTEGER

%%
etf1:       expr          { printf("OK\n"); }
    ;
...
factor:     '(' expr ')'
    |       INTEGER       { printf("INT %d rencontré\n", yylval);}
    ;

%%
int yylex(void) { ... }
void yyerror(char *msg) { ... }
int main() {  return yyparse(); }
```

Couplage Lex / Yacc (1/3)

Écrire l'analyseur lexical à la main n'est pas très pratique et le plus souvent, on passe par **lex**.

Il faut donc trouver un moyen de faire communiquer **lex** et **yacc**.

- On suppose que:
  - *les règles lexicales sont* ***etf2-lex.l*** *et*
  - *les règles syntaxiques dans* ***etf2-synt.y***.
- Appeler **bison** pour produire les fichiers
  - ***`etf2-synt.c`*** *qui contient l'analyseur (****option -o****)*
  - ***`etf2-synt.h`*** *qui contient la definition de* `INTEGER` *(****option -d****)*
- Appeler **flex** pour produire
  - ***`etf2-lex.c`*** *qui contient l'analyseur lexical (****option -o****)*
- Compiler ensuite les deux fichiers C produits:

```
bison -d -o etf2-synt.c etf2-synt.y
flex -o etf2-lex.c etf2-lex.l
gcc -o etf2 etf2-lex.c etf2-synt.c -lfl
```

Couplage Lex / Yacc (2/3)

Le fichier **etf2-lex.l**:

```
%{
  #include <stdio.h>
  #include "etf2-synt.h"
%}

number      [0-9]+

%%

[ \t]|\n    { /* sauter les espaces, les newlines et les tabs */ }
{number}    {
               yylval = atoi(yytext);
               return INTEGER;
            }
.           { return yytext[0]; }

%%
```

Couplage Lex / Yacc (3/3)

Le fichier **etf2-synt.y**:

```
%{  // Inclusion C
    #include <stdio.h>
    #include <ctype.h>
    int yylex(void);
    void yyerror(char *msg);
%}
%token INTEGER              // Déclaration token lex

%%
etf2:       expr              { printf("OK\n"); }
    ;
expr:       expr '+' term
    |       term ;

term:       term '*' factor
    |       factor ;

factor:     '(' expr ')'
    |       INTEGER         {printf("INT %d rencontré\n",yylval);}
    ;

%%
void yyerror(char *msg) { fprintf(stderr, "ERROR: %s\n", msg); }
int main() { return yyparse(); }
```

Précédence des opérateurs (1/3)

La grammaire $\mathbf{ETF_2}$ précédente permet de régler les problème de priorité des opérateurs.

- Pour des langages complexes, cela devient vite impraticable
- e.g. C++ a 18 niveaux de priorité!

Les règles de précédence permettent

- d'écrire plus facilement les grammaires
- de préciser les règle d'associativité des opérateurs
- de résoudre les conflits.

On veut pouvoir écrire:

```
E → E + E | E - E | E * E | E / E | (E) | NUMBER
```

et préciser que '*' et '/' sont plus prioritaires que '+' et '-'.

On utilise pour cela des directives **yacc**:

- **%left**: associativité à gauche
- **%right**: associativité à droite
- **%nonassoc**: non associatif

Précédence des opérateurs (2/3)

L'ordre des déclarations d'associativité permet de déduire les priorités des opérateurs:
$\Longrightarrow$ les opérateurs déclarés le **plus tôt** sont **moins** prioritaires.

Ainsi, si on déclare

```
%rigth  '='             // moins prioritaire
%nonassoc '<' '>'
%left   '+' '-'
%left   '*' '/'         // plus prioritaire
```

alors

```
a = b = c * d - e  - f * g
```

est analysée comme:

```
a = ( b = (  ( (c*d) - e)  - (f*g) )  )
```

et

```
a < b < c
```

est interdit (puisque '<' est non associatif)

Précédence des opérateurs (3/3)

Une grammaire pour les expressions pourrait donc être

```
%nonassoc  '<'  '>'
%left  '+' '-'
%left  '*' '/'
%right  '^'         /* puissance */
%nonassoc  UMIMUS   /* opérateur '-' unaire */

%%  /* zone de règles */
E   : E '+' E   | E '-' E
    | E '*' E   | E '/' E
    | E '<' E   | E '>' E
    | E '^' E
    | '-' E  %prec UMINUS   /* ici '-' prioritaire sur '*' */
    | '(' E ')' | NUMBER
    ;
```

%prec permet ici de fixer la précédence du '-' unaire. Par conséquent,

- `-5^2` sera analysé comme `(-5)^2` et non pas comme `-(5^2)`
- `3 + -2` ne déclenchera pas d'erreur (mais `3 + +2` oui!)

Les attributs

En **Yacc** les attributs sont synthétisés.

On peut donc calculer la valeur de l'attribut de la partie gauche en fonction d'attributs de la partie droite (**mais pas le contraire**)

Lorsqu'un règle est réduite:

- yacc "remonte" un attribut. Par convention, celui-ci s'appelle `$$` .
- l'attribut synthétisé `$$` peut être construit avec les attributs synthétisés de la partie droite. Il s'appellent `$1` , `$2` , ...

Par exemple sur G0:

```
...

expr:  expr '+' term            { $$ = $1 + $3; }
    |  term                     { $$ = $1; }
    ;
...
```

permet une évaluation ascendante de la valeur de l'expression

Grammaire attribuée pour ETF

Pour évaluer les expressions de ETF de façon ascendante, la grammaire devient:

```
ETF:          expr              { printf("expr = %d\n", $1); }
    ;
expr:       expr '+' term       { $$ = $1 + $3; }
    |       term                { $$ = $1; }
    ;

term:       term '*' factor     { $$ = $1 * $3; }
    |       factor                                  // α
    ;

factor:     '(' expr ')'        { $$ = $2; }
    |       NUMBER                                  // β
    ;
```

En **α** et **β**, on utilise le traitement implicite `{ $$ = $1; }`

Typage des attributs (1 / 3)

Par défaut `yylval` est de type entier.

On peut fixer un autre type avec la macro C `YYSTYPE`

Par exemple:

```
%{
 #include <stdio.h>
 ...
 #define YYSTYPE float      // ⇒ yylval typée comme  un float
%}

%%
ETF:          expr               { printf("expr = %f\n", $1); }
    ;
...
```

Il n'est pas rare que l'on ait besoin de travailler avec des attributs de types différents.

Dans ce cas, on peut utiliser la construction **`%union`** de *Yacc*.

Typage des attributs (2 / 3)

Soit grammaire

```
E → E + E | E - E | E * E | E / E | E < E | E > E | (E) | NUMBER
```

On déclare:

```
%union{
    int val;
    struct S {
       enum {ent, boolean} type;
       int v;
    } eval;
}

%token <val> NUMBER     // constantes sont typées ent
%type <eval> expr term factor
...

%%

expr :   expr '<' expr  { $$.type = boolean;  $$.v = ($1.v < $3.v); }
     |   expr '>' expr  { $$.type = boolean;  $$.v = ($1.v > $3.v); }
     |   expr '+' term  { $$.type = ent;      $$.v = $1.v + $3.v; }
     ...
     |   '(' expr ')'   { $$.type = $2.type;  $$.v = $2.v ; }
     |   NUMBER         { $$ = $1 ; }
```

Typage des attributs (3 / 3)

En fait, la déclaration **`%union`** précédente:

```
%union{
    int val;
    struct S {
       enum {ent, boolean} type;
       int v;
    } eval;
}
```

provoque la déclaration C suivante dans l'analyseur:

```
typedef union {
    int val;
    struct S {
       enum {ent, boolean} type;
       int v;
    } eval;
} YYSTYPE;

extern YYSTYPE yylval;
```

- **%token** permet de spécifier le champ (et donc le type) utilisé dans l'union pour l'attribut associé à un terminal
- **%type** permet de spécifier le champ (et donc le type) utilisé dans l'union pour l'attribut associé à un non-terminal

Reprise sur erreurs (1/2)

- **yyparse** est la fonction d'analyse
  - *elle renvoie 0 si l'analyse réussit et 1 sinon*
  - *Deux macros permettent de forcer le retour de* ***yyparse***
    - `YYABORT` provoque un retour avec échec
    - `YYACCEPT` provoque un retour avec succès
- **yyerror** est appelée quand une erreur se produit
  - *par défaut: pas très bavard ("Syntax error" et c'est tout)*
  - *Si on fait* `#define YYERROR_VERBOSE 1` *c'est un peu mieux:*

    `+12` sur ETF0 produit
    `syntax error, unexpected '+', expecting NUMBER or '-' or '('`
  - *Si pas de reprise, arrêt de l'analyse*

Reprise sur erreurs (2/2)

- **error** est un pseudo non-terminal qui permet d'avancer dans la phrase jusqu'à un lexème donné.
- **yyerrok** permet de remettre la pile en état

Exemple d'utilisation classique:

```
statement:
           whilestatement                { .... }
         | ifstatement                   { ... }
         | var = expression ';'          { .... }
         | error ';'                     { yyerrok; }
         ;
```

Cela permet de se «rattraper» sur un ';' quand on tombe sur une erreur syntaxique.

Résolution des conflits (1/6)

Yacc accepte des grammaire ambiguës en entrée

- cela permet (souvent) de simplifier l'écriture de la grammaire
- les règles de priorité vu plus haut permettent de "diriger" l'analyse:
  - *priorité d'une règle est celle du dernier lexème (éventuellement, pas de priorité)*
  - *Si conflit shift/reduce si la règle et la fenêtre ont une priorité associée:*
    - *shift* si la fenêtre est plus prioritaire
    - *reduce* si la fenêtre est moins prioritaire
    - si égalité, on regarde associativité:
      - gauche ⟶ réduction
      - droite ⟶ décalage
      - non associatif ⟶ erreur
- En l'absence de règle de priorité:
  - *si conflit shift/reduce, yacc choisit le shift*
  - *si conflit reduce/reduce, yacc prend la règle la plus en haut parmi les règles en conflit.*

Résolution des conflits (2/6)

Garder des conflits n'est en général pas souhaitable.

Au pire, vérifier que le traitement choisi par *yacc* correspond à ce que l'on veut.

Utiliser l'option **`-v`** permet de construire un fichier suffixé par `.output`

Prenons la grammaire suivante dans le fichier **`G.y`** .

```
%%
expr:   var '=' expr
    |   var
    |   '(' expr ')'
    ;
var:    IDENT
    |   var '[' expr ']'
    |   '*' expr
    ;
```

La construction de l'analyseur avec l'option *-d*:

- produit un fichier **G.output**, et
- indique qu'il y 2 conflits de type *shift*/*reduce* à l'état 5

Résolution des conflits (3/6)

On a:

```
State 5

    1 expr: var . '=' expr
    2     | var .
    5 var: var . '[' expr ']'

    '='  shift, and go to state 9
    '['  shift, and go to state 10

    '='       [reduce using rule 2 (expr)]
    '['       [reduce using rule 2 (expr)]
    $default  reduce using rule 2 (expr)
```

- Si on a `var` en pile et `'='` ou `'['` en fenêtre, on peut
  - *choisir shift (le défaut de yacc)*
  - *choisir reduce vers `expr`*

Le comportement par défaut (décalage) semble correct ici.

Résolution des conflits (4/6)

Mais dans l'état 3, qui est l'état où l'on est après une '*', on a:

```
State 3

    6 var: '*' . expr

    '('    shift, and go to state 1
    IDENT  shift, and go to state 2
    '*'    shift, and go to state 3

    expr  go to state 7
    var   go to state 5
```

Cet état peut donc aller sur l'état 5 précédent. On a donc

- `*tab[i]` qui est interprété comme `*(tab[i])` (ce qui est correct)
- `*p=q` qui est interprété comme `*(p=q)` (ce qui est incorrect)

En fait, cette grammaire est **ambiguë**

Voyons comment traiter ce conflit.

Résolution des conflits (5/6)

Première solution: Modifier la grammaire

Introduction d'une règle supplémentaire.

```
%%
expr:       var '=' expr
    |       var
    ;                           // Reduce avec '=' en fenêtre

var:        id
    |       '*' var
    ;                           // Shift avec '[' en fenêtre

id:         IDENT
    |       '(' expr ')'
    |       id '[' expr ']'
    ;
```

- C'est souvent assez difficile.
- Attention à ne pas modifier le langage reconnu

Résolution des conflits (6/6)

Deuxième solution: utiliser les règles de priorité

```
%right '='
%right '*'
%left  '['

%%
expr:       var '=' expr
    |       var                     %prec '='   // ⟵ important
    |       '(' expr ')'
    ;
var:        IDENT
    |       var '[' expr ']'
    |       '*' expr
    ;
```

- c'est souvent plus simple (même si ce n'est pas toujours trivial)
- permet de conserver la grammaire originale

Méthodologie de développement avec *Yacc*

- Conventions de style d'un fichier *yacc*
  - *les TERMINAUX (TOKEN) en majuscules*
  - *les non terminaux en minuscules*
  - *règles de grammaires et actions sur des lignes séparées*
  - *regrouper les règles avec le même membre gauche (non terminal)*
  - *";" après la dernière règle d'un groupe et sur une ligne séparée*
  - *indentation: 2 tab pour les règles et 3 tab pour les actions*
- Récursions
  - *préférer les récursivités gauches*
- Méthodologie
  - *programmer la validation syntaxique(phase II)*
  - *programmer la validation lexicale (phase I)*
  - *programmer les phases sémantiques (phases III)*
