

Introduction – 1

Le générateur de *parsers* YACC/BISON



INFO010 – Théorie des langages – Partie pratique

S. COLLETTE et G. GEERAERTS

- YACC est un outil qui permet de générer automa{t|g}iquement des *parsers*.
- Son entrée est une **spécification** qui décrit la grammaire du langage et les **action** à effectuer tout au long du *parsing*.
- Sa sortie est un morceau de code C qui implémente le *parser* sous forme d'une fonction : `yyparse()`.

Le *parser* généré

Le *parser* généré par YACC a les caractéristiques suivantes :

- Il analyse la grammaire en *bottom-up* ;
- Il fait appel à la fonction `yylex()` comme analyseur lexical. Celle-ci peut provenir de LEX. (mais ce n'est pas obligatoire) ;
- Il fait appel à la fonction `yyerror()` en cas d'erreur (à implémenter) ;
- Il effectue les **actions** spécifiées pour chaque règle lors du *reduce* de la règle.

Spécification YACC

- Spécification en trois parties (séparées par `%%`) :
 - Partie 1 : les déclarations
 - Déclarations C entre `%{` et `%}`
 - Déclaration des *tokens* grâce au mot-clef `%token` ;
 - **Associativité** des opérateurs : `%left`, `%right` et `%nonassoc`.

Spécification YACC

- Spécification en trois parties (séparées par %%):
 - Partie 2 : les règles.
 - Sont de la forme :

```
symbol      : definition1 {action1}
              | definition2 {action2}
              ...
              ;
```
 - Les actions sont effectuées à chaque fois que la définition correspondante est **réduite**.

Spécification YACC

- Spécification en trois parties (séparées par %%):
 - Partie 2 : les règles.
 - Les définitions sont formées de **parties gauches d'autres règles** et **d'identifiants de tokens**;
 - Les *tokens* auront été **déclarés dans la première partie** grâce à %token;
 - Pour que LEX utilise les mêmes *tokens*, on fait appel à YACC avec l'option -d, pour générer **y.tab.h**.

Spécification YACC

- Spécification en trois parties (séparées par %%):
 - Partie 3 : le code C. Il peut contenir les fonctions suivantes :
 - **main()**, qui fait alors appel à **yparse()**;
 - **yyerror()**... sans commentaires!
 - **yylex()**, qui sera en général générée par LEX dans un fichier séparé.

Attributs

- YACC permet d'**associer des attributs à chaque symbole**, grâce aux variables \$\$, \$1, \$2,...
- \$\$ est l'attribut de la **partie gauche** de la règle;
- \$i est l'attribut du **ième symbole de la partie droite** de la règle;
- Par défaut, ces attributs sont des entiers;
- On peut y accéder dans le code :

```
expression : expression '+' term
            { $$ = $1 + $3 } ;
```

Attributs : les terminaux

- À quoi correspondent les attributs `$i` des terminaux ?
- Il s'agit de la valeur mise dans `yylval` par la fonction `yylex()`.

Attributs : types plus complexes

- Les entiers c'est bien... mais on peut faire mieux!
- On peut déclarer les *tokens* comme étant des **union** au sens de C :

```
%union {
    int cmd ;
    char * text ;
    double val ;
}
```
- Cet union sera aussi le type de `yylval`!

Attributs : types plus complexes – 2

- On peut choisir d'associer un terminal ou un non-terminal à un élément particulier de l'union.
- Pour les **terminaux**, cela se fait grâce à **%token** :

```
%token<val> REAL
%token<text> IDENTIFIER
```

Attributs : types plus complexes – 2

- On peut choisir d'associer un terminal ou un non-terminal à un élément particulier de l'union.
- Pour les **non-terminaux**, cela se fait grâce à **%type** :

```
%type<val> expression
%type<val> term
...
expression : expression + term
{ $$ = $1 + $3 ; }
// $$, $1 et $3 : de type double
```

Attributs : types plus complexes – 3

- **Attention!** dans le *scanner* LEX, il faut toujours déclarer explicitement quel élément de l'union on manipule!
- Pour un *token* associé au champ *c* de l'union, on devra donc stocker la valeur de l'attribut dans `yy1val.c`.

Exemple – Spécification LEX

```
%{
#include "y.tab.h"
}%
integer    [0-9]*
real       ([0-9]*"."[0-9]+)
nl         \n
%%
[ \t]+    ;
{integer} { sscanf(yytext, "%d", &yy1val.integer) ;
            return INTEGER ; }
{real}    { sscanf(yytext, "%lf", &yy1val.real) ;
            return REAL ; }
{nl}      { extern int lineno ; lineno++ ;
            return '\n' ; }
.         { return yyext[0] ; }
%%
```

Exemple – Spécification YACC

```
%{
#include <stdio.h>
}%

%union {
    double  real ;
    int     integer ;
}

%token <real> REAL
%token <integer> INTEGER

%type <real> rexpr
%type <integer> iexpr

%left '+' '-'
%left '*' '/'
```

Exemple – Spécification YACC

```
%%
iexpr:  INTEGER { $$ = $1 ; }
        | iexpr '+' iexpr
          { $$ = $1 + $3 ; }
        | iexpr '-' iexpr
          { $$ = $1 - $3 ; }
        | iexpr '*' iexpr
          { $$ = $1 * $3 ; }
        | iexpr '/' iexpr
          { if ($3) $$ = $1 / $3 ;
            else { printf ("error: division by zero\n") ;
                    yyerror() ; } }
        | '(' iexpr ')'
          { $$ = $2 ; }
        ;
rexpr:  REAL
        | rexpr '+' rexpr
          [ ... ]
```

Exemple – Spécification YACC

```
char * progname ;
int lineno ;

main(int argc, char ** argv)
{
    progname = argv[0] ;
    yyparse() ;
}

yyerror(char * s)
{
    fprintf(stderr, "%s: %s", progname, s) ;
    fprintf(stderr, " line %d\n", lineno) ;
}
```

Exercices

Voir feuilles volantes distribuées au T.P.