

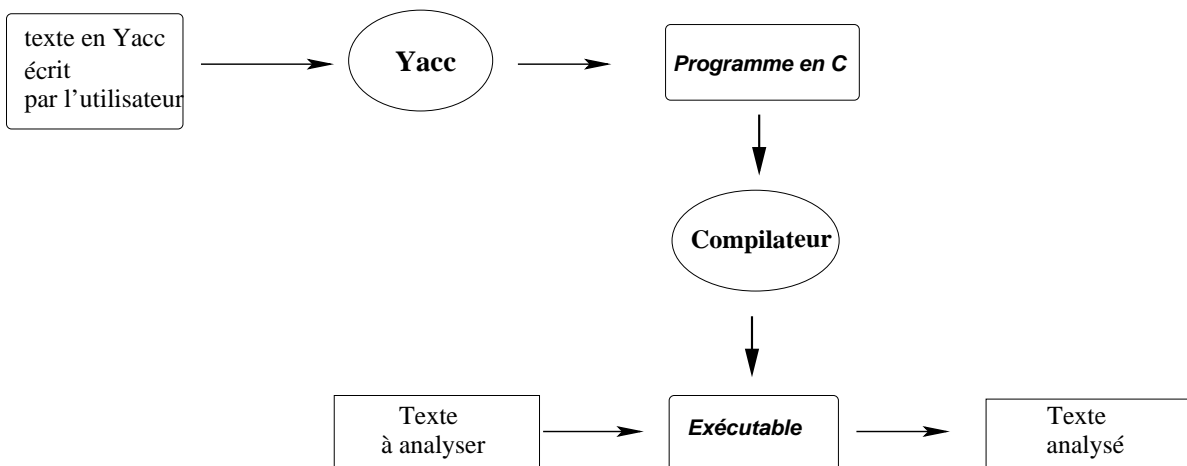
Une introduction au logiciel Yacc

Robert Cori

Octobre 2007

1 Principe général

Lex est un outil logiciel qui permet de générer des analyseurs syntaxiques. Pour réaliser un tel analyseur il faut décrire d'abord l'outil à construire dans un texte `monanalyseur.y`, ce texte fourni à **Yacc** permet d'obtenir ensuite un analyseur syntaxique en langage C qui a pour nom `y.tab.c`. Il convient d'ajouter un texte lex qui va engendrer l'analyseur lexicale `lex.yy.c` qu'il faudra inclure dans le programme principal. Une fois obtenu cet analyseur il faut le compiler et l'on a alors un exécutable qui effectue l'analyse syntaxique d'un texte suivant les instructions données dans `monanalyseur.y`. ce fonctionnement peut être schématisé par la figure suivante :



Les instructions nécessaires à la réalisation sont alors les suivantes

```
lex monanalyseur.lex
yacc monanalyseur.y
gcc -o monanalyseur y.tab.c
./monanalyseur < texteAAAnalyser > resultat
```

En fait il est plus convenable de réaliser l'ensemble de ces trois opérations au sein d'un Makefile comme par exemple:

```
%.lex %.y
    flex $*.lex
    yacc -t -v $*.y
```

```

mv y.output $*.automate
mv y.tab.c $*.c
gcc -Wall $*.c -lfl -o $@

```

qui permet de générer un analyseur à partir de `monanalyseur.lex` par la commande :

```
make monanalyseur
```

On se propose ici de décrire en détail comment construire un texte `monanalyseur.y` que nous appellerons un “programme en Yacc”.

2 Organisation d’un programme en Yacc

Un programme en **Yacc** se divise en trois parties séparées par des `%`. la première contient des options, et des déclarations et initialisations de variables utiles pour le programme en C généré. La seconde contient la grammaire du langage et les actions à effectuer lorsque au cours de l’analyse on a réduit par une règle, enfin la troisième partie contient l’enveloppe du programme C généré.

2.1 Grammaire et actions

C’est le cœur du programme **Yacc**, il s’agit ici de donner les règles de la grammaire le symbole \rightarrow du cours est remplacé par `:` et n’est écrit qu’une fois pour un non terminal, les différentes règles pour ce même non-terminal sont séparées par le symbole `|`. Les actions à effectuer sont à indiquer après chaque règle et doivent être exprimées dans le langage C.

2.2 Le programme en C

La troisième partie doit contenir le programme principal `main()` qui doit en général faire un appel à la fonction `yyparse()` qui est créée par Yacc, et il doit aussi contenir un `#include ‘‘lex.yy.c’’`.

2.3 Un premier exemple

Un exemple simple est le suivant, c’est un analyseur qui lit une suite de **a** et de **b** représentant un mot de parenthèse (**a** ouvrante, **b** fermante) et qui donne les règles qui l’ont engendré suivant la grammaire habituelle :

$$S \rightarrow aSbS \quad S \rightarrow aSb \quad S \rightarrow abS \quad S \rightarrow ab$$

Le point virgule sert de marqueur de fin

Voici le programme Yacc

```

%token PVIRG PARO PARF
%%
s0 :
    exp PVIRG {printf("fini\n"); exit(0);}
    ;
exp: PARO exp PARF exp {printf(" regle1 \n");}
    |PARO exp PARF {printf(" regle2 \n");}
    |PARO PARF exp {printf(" regle3 \n");}
    |PARO PARF {printf(" regle4 \n");}
    ;
%%

```

```

#include <ctype.h>
#include <stdio.h>
#include "lex.yy.c"
main(){
    yyparse();
}
yyerror(char *s){
    printf ("%s\n",s);
}

```

et le programme Lex qui va avec

```

%option noyywrap
%%
; {return PVIRG;}
a {return PARO;}
b {return PARF;}
%%
#include <stdlib.h>
#include <stdio.h>

```

3 Exemples avec ETF

3.1 Evaluation d'une expression

```

%{
#define YYSTYPE int
#include <math.h %}
%token NUMBER PLUS POINTV MULT PARO PARF ID

%%
statement : expression POINTV {
            printf(" val = %d \n", $1);
            exit(0);}
;
expression : expression PLUS terme  {$$ = $1 + $3;}
            | terme {$$ = $1;}
;
terme      : terme MULT facteur  {$$ = $1 * $3;}
            | facteur {$$ = $1;}
;
facteur    : PARO expression PARF {$$ = $2;}
            | NUMBER {$$ = $1;}
;
%%
#include <ctype.h>
#include <stdio.h>
#include "lex.yy.c"

```

```

main()
{
    yyparse();
}
yyerror(char *s) {
    printf ("%s\n",s);
}

```

3.2 Construction de la traduction en code machine

```

%{
#define YYSTYPE char*
#include <ctype.h>
#define LMAX 1000
#include <stdlib.h>
#include<string.h>
    void trans (char *u){
        int i;
        for (i = 0; u[i] != '\0'; i++)
            if(isdigit(u[i])) u[i]++;
    }
}%
%token NUMBER PLUS PVIRG  MULT PARO PARF ID EGAL
%%
statement : expression PVIRG {
    printf("\t %s \n", $1);exit(0);}
;
expression : expression PLUS terme {
    trans($3);
    strcat(strcat ($1,$3), "add R1 R2 \n") ;
    $$ = $1;}
| terme {
    $$ = $1;}
;
terme:      terme MULT facteur {
    trans($3);
    strcat(strcat($1,$3), "mult R1 R2 \n");
    $$ = $1;}
| facteur {$$ = $1;}
;
facteur : ID {$$ = (char *) malloc(LMAX);
    strcpy($$, "load " );
    strcat($$, $1); strcat($$ , " R1 \n");
    }
| PARO  expression PARF  {$$ = $2;}
;

```

4 Calcul de l'automate d'analyse

Avec l'option `-v` Yacc crée un fichier texte qui contient des informations sur la grammaire lue et la table de l'automate *SLR(1)* construit. En voici un exemple avec le langage *E, T, F*.

```
0 $accept : statement $end
1 statement : expression PVIRG
2 expression : expression PLUS terme
3           | terme
4 terme : terme MULT facteur
5       | facteur
6 facteur : PARO expression PARF
7       | ID

state 0
$accept : . statement $end (0)
ID shift 1
PARO shift 2
. error
statement goto 3
expression goto 4
terme goto 5
facteur goto 6
state 1
facteur : ID . (7)
. reduce 7
state 2
facteur : PARO . expression PARF (6)
ID shift 1
PARO shift 2
. error
expression goto 7
terme goto 5
facteur goto 6
state 3
$accept : statement . $end (0)
$end accept
state 4
statement : expression . PVIRG (1)
expression : expression . PLUS terme (2)
PLUS shift 8
PVIRG shift 9
. error
state 5
expression : terme . (3)
terme : terme . MULT facteur (4)
MULT shift 10
PLUS reduce 3
PARF reduce 3
PVIRG reduce 3
state 6
```

```

terme : facteur . (5)
. reduce 5
state 7
expression : expression . PLUS terme (2)
facteur : PARO expression . PARF (6)
PLUS shift 8
PARF shift 11
. error
state 8
expression : expression PLUS . terme (2)
ID shift 1
PARO shift 2
. error
terme goto 12
facteur goto 6
state 9
statement : expression PVIRG . (1)
. reduce 1
state 10
terme : terme MULT . facteur (4)
ID shift 1
PARO shift 2
. error
facteur goto 13
state 11
facteur : PARO expression PARF . (6)
. reduce 6
state 12
expression : expression PLUS terme . (2)
terme : terme . MULT facteur (4)
MULT shift 10
PLUS reduce 2
PARF reduce 2
PVIRG reduce 2
state 13
terme : terme MULT facteur . (4)
. reduce 4
8 terminals, 5 nonterminals
8 grammar rules, 14 states

```