

Introduction to Yacc

- **General Description**
- **Input file**
- **Output files**
- **Parsing conflicts**
- **Pseudovariables**
- **Examples**



General Description

- A **parser generator** is a program that takes input a specification of a syntax, and produces as output a procedure for recognizing that language.
- Historically, they are also called **compiler-compilers**.
- YACC (yet another compiler-compiler) is a LALR(1) (LookAhead, Left-to-right, Rightmost derivation producer with 1 lookahead token) parser generator.
- YACC was originally designed for being complemented by Lex.



Input File

- Yacc input file is divided in three parts.

```
/* definitions */
```

```
...
```

```
%%
```

```
/* rules */
```

```
...
```

```
%%
```

```
/* auxiliary routines */
```

```
...
```



Input File

- The definition part includes information about the tokens used in the syntax definition:

`%token NUMBER`

- Yacc automatically assigns numbers for tokens, but it can be overridden by

`%token NUMBER 621`

- Yacc also recognizes single characters as tokens. Therefore, assigned token numbers should no overlap ASCII codes.



Input File

- The definition part can also include C code external to the definition of the parser, within `%{` and `%}` in the first column.
- It can also include the specification of the starting symbol in the grammar:

`%start nonterminal`



Input File

- The rules part contains grammar definition in a modified BNF form.

```
<non_term> : <token_list> <action>  
            { <token_list> <action> }* ;
```

- Actions are normal C sentences (can be a compound sentence between {}).



Input File

- The auxiliary routines part is only C code.
- It includes function definitions for every function needed in rules part
- It can also contain the `main()` function definition if the scanner is going to be run as a program.
- The `main()` function must call the function `yyparse()`.



Input File

- If `yylex()` is not defined in the auxiliary routines sections, then it should be **#included** in the definitions part.
- Yacc input file generally finishes with `.y`



Output Files

- The output of Yacc is a file named `<name>.tab.c`, with the same name of the input (except for the extension `.y`).
- If it contains the `main()` definition, it must be compiled to be executable.
- Otherwise, the code can be an external function definition for the function

```
int yyparse() ;
```



Output Files

- If called with the `-d` option in the command line, Yacc produces as output a header file `<name>.tab.h` with all its specific definition (particularly important are token definitions to be included, for example, in a Lex input file).
- If called with the `-v` option, Yacc produces as output a file `y.output` containing a textual description of the LALR(1) parsing table used by the parser. This is useful for tracking down how the parser solves ambiguities and inaccuracies of the grammar.



Parsing conflicts

- Yacc has built-in disambiguities rules, so it can parse grammars even in the presence of conflicts.
- The file `y.output` describes eventual conflicts and how they are solved.
- Conflicts shift/reduce will be solved in general by giving preference to shift.
- Conflicts reduce/reduce will be solved in general by giving preference to the first grammar rule listed in the input file.



Parsing conflicts

- Yacc also provides way to define precedence and associativity of operators.
- In the definition part, we can include

```
%left '+' , '-'
```

```
%left '*' , DIVISION
```

- This line states that + and – have the same precedence and are left associative, and that the * and DIVISION (defined as token) have the same precedence but higher than + and -, and are also left associative.
- The other possible declarations are `%right` and `%nonassoc`.
- These definitions have the advantages of simplifying the grammar, and reducing the table so that the parser is more efficient.



Pseudovariabes

- In action part in rules, every symbol (token or non-terminal) in a grammar rule is associated with a **pseudovariabes**.
- This pseudovariabes are name $\$1, \$2, \dots$ for the first, second, \dots symbol in the recognized rule.
- It is possible also to assign the pseudovariabes $\$\$$ that refers to the non-terminal symbol that defines the rule.

exp : exp PLUS term $\{\$\$ = \$1 + \$3\}$ |
exp MINUS term $\{\$\$ = \$1 - \$3\}$ |
term $\{\$\$ = \$1\}$;



Pseudovariabes

- The pseudovariabes assigned to a token is defined in the scanner `yylex()` by assigned the global variabes `yylval`.
- Pseudovariabes are normally integers, but it is possible to assign a different type.
- In case it is needed several types for different pseudovariabes, a union type must be defined.

```
%union { double val; char op; }
```

```
%type <val> exp term factor
```

```
%type <op> addop mulop
```



Examples Parenthesis (.y)

```
%{
#include <ctype.h>
#include <stdio.h>
#define YYSTYPE double /* double type for yacc stack */
}%
%%
lines : lines S '\n' { printf("OK \n"); }
      | lines '\n'
      | /* empty */
      | error '\n' { yyerror("Error: reenter last line:");
                    yyerrok; } ;
S     : S '(' S ')' S
      | S '[' S ']' S | ;
```



Examples Parenthesis (.y)

```
%%  
#include "lex.yy.c"  
  
void yyerror(char * s)  
/* yacc error handler */  
{  
    fprintf (stderr, "%s\n", s);  
}  
  
int main(void) {return yyparse();}
```



Examples Parenthesis (.lex)

```
%%  
[ \t]    { /* skip blanks and tabs */ }  
\n|.     { return yytext[0]; }
```



Examples – Roman numbers (.y)

```
%{  
#include "rom.h"  
#define          YYTRACE  
  
int last=0;  
%}  
  
%token GLYPH WHITE  
%start file  
%%
```



Examples - Roman numbers (.y)

```

file          /* file uses the more efficient left-recursion */
: number
    { printf ("Got <%d>\n", $1); $$=$1; last=0; }
| file WHITE number
    { printf ("Got <%d>\n", $3); $$=$3; last=0; };

number /* number uses right-recursion and a state variable */
: GLYPH          { last = $$=$1; }
| GLYPH number  { if ($1 >= last)
                  $$ = $2 + (last=$1);
                  else  $$ = $2 - (last=$1); } ;

```

%%



Examples Roman numbers (.lex)

```
%{  
#include "ytab.h"  
#include <stdlib.h>  
#include <stdarg.h>  
#include <stdio.h>  
#include <string.h>  
void count ();  
%}  
  
%%
```



Examples Roman numbers (.lex)

```
I      { count(); yylval = 1; return GLYPH; }
V      { count(); yylval = 5; return GLYPH; }
X      { count(); yylval = 10; return GLYPH; }
L      { count(); yylval = 50; return GLYPH; }
C      { count(); yylval = 100; return GLYPH; }
D      { count(); yylval = 500; return GLYPH; }
M      { count(); yylval = 1000; return GLYPH; }

[ \t\n] { count(); return (WHITE); }

.      { count();
        yyerror("Unknown character (%c)", *yytext); }
```

%%



Examples Roman numbers (.lex)

```
int column = 0; int line_num = 1; char error_line_buffer[256];
void count() {
    int i; static char *elb=error_line_buffer;
    for (i = 0; yytext[i] != '\0'; i++) {
        if (yytext[i] == '\n') {
            column = 0;
            line_num++;
            elb = error_line_buffer;
        } else {
            *elb++ = yytext[i];
            if (yytext[i] == '\t')
                column += 8 - (column & (8-1));
            else
                column++; }
        *elb = 0; } }
```



Examples Roman numbers (.lex)

```
void yyerror(char *fmt, ...) {
    /* extern int yylineno; */
    extern int column, line_num;
    extern char error_line_buffer[256];
    va_list va;

    va_start(va, fmt);
    fprintf (stdout, "%s\n", error_line_buffer);
    fprintf (stdout, "\n%*s", column, "^");
    vfprintf(stdout, fmt, va);
    fprintf (stdout, ":Line %d\n", line_num);
    error_code = 26;
}
```



Examples Roman numbers (.c)

```
#include "rom.h"
int error_code = 0;
main (int c, char **v) {
    /* Initialize */
    yyparse ();
    if (error_code)
        printf ("C2C Error: %d\n", error_code);
    return error_code;
}
```



Examples Calculator (.y)

```
%{  
#include <string.h>  
#include "calc.h"  
%}  
  
%union { int value; struct symtab *symlink; }  
%token <symlink> NAME  
%token <value> NUMBER  
%left '-' '+'  
%left '*' '/'  
%nonassoc UMINUS  
  
%type <value> expression  
%%
```



Examples Calculator (.y)

```

sentence:  stlist ';' expression '.'
           { printf("Result: %d\n", $3); exit(0); }
|  expression '.'
           { printf("Result: %d\n", $1); exit(0); }
;

stlist:   stlist ';' statement
|  statement
;

statement: NAME '=' expression
           { $1->value=$3; $1->initialized=YES; }
;

```



Examples Calculator (.y)

```
expression: expression '+' expression { $$ = $1 + $3; }
| expression '-' expression { $$ = $1 - $3; }
| expression '*' expression { $$ = $1 * $3; }
| expression '/' expression { if($3 == 0)
                               yyerror("divide by zero");
                               else $$ = $1 / $3; }
| '-' expression %prec UMINUS { $$ = -$2; }
| '(' expression ')' { $$ = $2; }
| NUMBER { $$ = $1; }
| NAME { if ($1->initialized==YES)
          $$ = $1->value;
        else yyerror("variable without valid value used"); } ;
%%
```



Examples Calculator (.y)

```
struct symtab *symlook(char *str) {
    char *p;
    struct symtab *sp;
    for(sp = symtab; sp < &symtab[MAX_SYM]; sp++){
        if(sp->name && !strcmp(sp->name, str)) /* return pointer */
            return sp;
        if(!sp->name) { /* create new variable */
            sp->name = strdup(str);
            sp->initialized = NO;
            return sp;
        }
    }
    yyerror("Too many variables.");
    exit(1);
}
```



Examples Calculator (.lex)

```
%{
#include <string.h>
#include "calc.h"
#include "y.tab.h"
}%
id ([a-zA-Z_][a-zA-Z0-9_]*)
%%
[0-9]+ { yylval.value = atoi(yytext); return NUMBER; }
{id}   { yylval.symlink = symlook(yytext); return NAME; }
[ \t\n] ; /* ignore white space */
[=+\-*/;.()] return yytext[0];
. { printf("Error: Illegal character! (%s)\n",yytext);
  exit(1); }
%%
```

