

A Yacc Tutorial

Victor Eijkhout

July 2004

1 Introduction

The unix utility *yacc* (Yet Another Compiler Compiler) parses a stream of token, typically generated by *lex*, according to a user-specified grammar.

2 Structure of a *yacc* file

A *yacc* file looks much like a *lex* file:

```
...definitions...
%%
...rules...
%%
...code...
```

In the example you just saw, all three sections are present:

definitions All code between `%{` and `%}` is copied to the beginning of the resulting C file.

rules A number of combinations of pattern and action: if the action is more than a single command it needs to be in braces.

code This can be very elaborate, but the main ingredient is the call to `yylex`, the lexical analyser. If the code segment is left out, a default main is used which only calls `yylex`.

3 Definitions section

There are three things that can go in the definitions section:

C code Any code between `%{` and `%}` is copied to the C file. This is typically used for defining file variables, and for prototypes of routines that are defined in the code segment.

definitions The definitions section of a *lex* file was concerned with characters; in *yacc* this is tokens. These token definitions are written to a `.h` file when *yacc* compiles this file.

associativity rules These handle associativity and priority of operators.

4 Lex Yacc interaction

Conceptually, *lex* parses a file of characters and outputs a stream of tokens; *yacc* accepts a stream of tokens and parses it, performing actions as appropriate. In practice, they are more tightly coupled.

If your *lex* program is supplying a tokenizer, the *yacc* program will repeatedly call the *yylex* routine. The *lex* rules will probably function by calling *return* everytime they have parsed a token. We will now see the way *lex* returns information in such a way that *yacc* can use it for parsing.

4.1 The shared header file of return codes

If *lex* is to return tokens that *yacc* will process, they have to agree on what tokens there are. This is done as follows.

- The *yacc* file will have token definitions
%token NUMBER
in the definitions section.
- When the *yacc* file is translated with *yacc -d*, a header file *y.tab.h* is created that has definitions like
#define NUMBER 258
This file can then be included in both the *lex* and *yacc* program.
- The *lex* file can then call *return NUMBER*, and the *yacc* program can match on this token.

The return codes that are defined from %TOKEN definitions typically start at around 258, so that single characters can simply be returned as their integer value:

```
/* in the lex program */
[0-9]+ {return NUMBER}
[-+*/] {return *yytext}
```

```
/* in the yacc program */
sum : TERMS '+' TERM
```

See example 7.1 for a worked out code.

4.2 Return values

In the above, very sketchy example, *lex* only returned the information that there was a number, not the actual number. For this we need a further mechanism. In addition to specifying the return code, the *lex* parse can return a symbol that is put on top of the stack, so that *yacc* can access it. This symbol is returned in the variable *yyval*. By default, this is defined as an *int*, so the *lex* program would have

```
extern int yyval;
%%
[0-9]+ {yyval=atoi(yytext); return NUMBER;}
```

If more than just integers need to be returned, the specifications in the *yacc* code become more complicated. Suppose we want to return double values, and integer indices in a table. The following three actions are needed.

1. The possible return values need to be stated:
%union {int ival; double dval;}
2. These types need to be connected to the possible return tokens:

```
%token <ival> INDEX
%token <dval> NUMBER
```

3. The types of non-terminals need to be given:

```
%type <dval> expr
%type <dval> mulex
%type <dval> term
```

The generated .h file will now have

```
#define INDEX 258
#define NUMBER 259
typedef union {int ival; double dval;} YYSTYPE;
extern YYSTYPE yylval;
```

This is illustrated in example 7.2.

5 Rules section

The rules section contains the grammar of the language you want to parse. This looks like

```
name1 :   THING something OTHERTHING {action}
        | othersomething THING          {other action}
name2 :   .....
```

This is the general form of context-free grammars, with a set of actions associated with each matching right-hand side. It is a good convention to keep non-terminals (names that can be expanded further) in lower case and terminals (the symbols that are finally matched) in upper case.

The terminal symbols get matched with return codes from the *lex* tokenizer. They are typically defined coming from %token definitions in the *yacc* program or character values; see section 4.1.

A simple example illustrating these ideas can be found in section 7.1.

6 User code section

The minimal main program is

```
int main()
{
    yyparse();
    return 0;
}
```

Extensions to more ambitious programs should be self-evident.

In addition to the main program, the code section will usually also contain subroutines, to be used either in the *yacc* or the *lex* program. See for instance example 7.3.

7 Examples

7.1 Simple calculator

This calculator evaluates simple arithmetic expressions. The *lex* program matches numbers and operators and returns them; it ignores white space, returns newlines, and gives an error message on anything else.

```
%{
#include <stdlib.h>
#include <stdio.h>
#include "calcl.h"
void yyerror(char*);
extern int yylval;

}%

%%

[ \t]+ ;
[0-9]+    {yylval = atoi(yytext);
           return INTEGER;}
[-+*/]    {return *yytext;}
"("       {return *yytext;}
")"       {return *yytext;}
\n        {return *yytext;}
.         {char msg[25];
           sprintf(msg, "%s <%s>", "invalid character", yytext);
           yyerror(msg);}
```

Accepting the *lex* output, this *yacc* program has rules that parse the stream of numbers and operators, and perform the corresponding calculations.

```
%{
#include <stdlib.h>
#include <stdio.h>
int yylex(void);
#include "calcl.h"
}%

%token INTEGER

%%

program:
    line program
    | line

line:
    expr '\n'          { printf("%d\n", $1); }
    | 'n'

expr:
    expr '+' mulex     { $$ = $1 + $3; }
    | expr '-' mulex   { $$ = $1 - $3; }
```

```

| mulex          { $$ = $1; }
mulex:
    mulex '*' term    { $$ = $1 * $3; }
| mulex '/' term      { $$ = $1 / $3; }
| term                { $$ = $1; }
term:
    '(' expr ')'      { $$ = $2; }
    | INTEGER         { $$ = $1; }

%%

void yyerror(char *s)
{
    fprintf(stderr, "%s\n", s);
    return;
}

int main(void)
{
    /*yydebug=1;*/
    yyparse();
    return 0;
}

```

Here we have realized operator precedence by having separate rules for the different priorities. The rule for plus/minus comes first, which means that its terms, the `mulex` expressions involving multiplication, are evaluated first.

7.2 Calculator with simple variables

In this example the return variables have been declared of type `double`. Furthermore, there can now be single-character variables that can be assigned and used. There now are two different return tokens: double values and integer variable indices. This necessitates the `%union` statement, as well as `%token` statements for the various return tokens and `%type` statements for the non-terminals.

This is all in the `yacc` file:

```

%{
#include <stdlib.h>
#include <stdio.h>
int yylex(void);
double var[26];
%}

%union { double dval; int ivar; }
%token <dval> DOUBLE
%token <ivar> NAME
%type <dval> expr
%type <dval> mulex
%type <dval> term

```

```

%%

program:
    line program
    | line
line:
    expr '\n' { printf("%g\n",$1); }
| NAME '=' expr '\n' { var[$1] = $3; }
expr:
    expr '+' mulex { $$ = $1 + $3; }
    | expr '-' mulex { $$ = $1 - $3; }
| mulex { $$ = $1; }
mulex:
    mulex '*' term { $$ = $1 * $3; }
| mulex '/' term { $$ = $1 / $3; }
| term { $$ = $1; }
term:
    '(' expr ')' { $$ = $2; }
| NAME { $$ = var[$1]; }
| DOUBLE { $$ = $1; }

%%

void yyerror(char *s)
{
    fprintf(stderr,"%s\n",s);
    return;
}

int main(void)
{
    /*yydebug=1;*/
    yyparse();
    return 0;
}

```

The *lex* file is not all that different; note how return values are now assigned to a component of *yylval* rather than *yylval* itself.

```

%{
#include <stdlib.h>
#include <stdio.h>
#include "calc2.h"
void yyerror(char*);
%}

%%

[ \t]+ ;

```

```

([0-9]+(\.[0-9]*)?)|([0-9]*\.[0-9]+)) {
    yylval.dval = atof(yytext);
    return DOUBLE;}
[-+*/=]    {return *yytext;}
"("        {return *yytext;}
")"        {return *yytext;}
[a-z]      {yylval.ivar = *yytext;
            return NAME;}
\n         {return *yytext;}
.          {char msg[25];
            sprintf(msg,"%s <%s>", "invalid character",yytext);
            yyerror(msg);}

```

7.3 Calculator with dynamic variables

Basically the same as the previous example, but now variable names can have regular names, and they are inserted into a names table dynamically. The *yacc* file defines a routine for getting a variable index:

```

%{
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int yylex(void);
#define NVAR 100
char *vars[NVAR]; double vals[NVAR]; int nvars=0;
%}

%union { double dval; int ivar; }
%token <dval> DOUBLE
%token <ivar> NAME
%type <dval> expr
%type <dval> mulex
%type <dval> term

%%

program:
    line program
    | line

line:
    expr '\n' { printf("%g\n", $1); }
| NAME '=' expr '\n' { vals[$1] = $3; }

expr:
    expr '+' mulex { $$ = $1 + $3; }
    | expr '-' mulex { $$ = $1 - $3; }
    | mulex { $$ = $1; }

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

```

```

term:
    '(' expr ')'      { $$ = $2; }
    | NAME            { $$ = vals[$1]; }
    | DOUBLE          { $$ = $1; }

```

```
%%
```

```

int varindex(char *var)
{
    int i;
    for (i=0; i<nvars; i++)
        if (strcmp(var,vars[i])==0) return i;
    vars[nvars] = strdup(var);
    return nvars++;
}

```

```

int main(void)
{
    /*yydebug=1;*/
    yyparse();
    return 0;
}

```

The *lex* file is largely unchanged, except for the rule that recognises variable names:

```

%{
#include <stdlib.h>
#include <stdio.h>
#include "calc3.h"
void yyerror(char*);
int varindex(char *var);
}%

%%

[ \t]+ ;
((([0-9]+(\.[0-9]*)?)|([0-9]*\.[0-9]+)) {
    yylval.dval = atof(yytext);
    return DOUBLE;}
[-+*/=] {return *yytext;}
"(" {return *yytext;}
")" {return *yytext;}
[a-z][a-z0-9]* {
    yylval.ivar = varindex(yytext);
    return NAME;}
\n {return *yytext;}
. {char msg[25];
    sprintf(msg,"%s <%s>","invalid character",yytext);
    yyerror(msg);}

```