A *yacc* tutorial

Victor Eijkhout

August 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...
```

definitions As with *lex*, all code between `%{` and `%}` is copied to the beginning of the resulting C file. There can also be various definitions; see section **??**.

rules As with *lex*, a number of combinations of pattern and action. The patterns are now those of a context-free grammar, rather than of a regular grammar as was the case with *lex*.

code This can be very elaborate, but the main ingredient is the call to `yyparse`, the grammatical parse.

3 Motivating example

It is harder to give a small example of *yacc* programming than it was for *lex*. Here is a program that counts the number of *different* words in a text. (We could have written this particular example in *lex* too.)

First consider the *lex* program that matches words:

```
%{

#include "words.h"
int find_word(char*);
extern int yylval;
%}

%%

[a-zA-Z]+ {yylval = find_word(yytext);
   return WORD;}
.         ;
\n        ;

%%
```

The lexer now no longer has a main program, but instead returns a `WORD` return code. It also calls a routine `find_word`, which inserts the matched word in a list if it is not already there.

The routine `find_word` is defined in the *yacc* code:

```
%{

#include <stdlib.h>
#include <string.h>
  int yylex(void);
#include "words.h"
  int nwords=0;
#define MAXWORDS 100
  char *words[MAXWORDS];
%}

%token WORD

%%

text : ;
       | text WORD  ; {
            if ($2<0) printf("new word\n");
            else printf("matched word %d\n",$2);
                     }

%%

int find_word(char *w)
{
  int i;
```

```
  for (i=0; i<nwords; i++)
    if (strcmp(w,words[i])==0) {
      return i;
    }
  words[nwords++] = strdup(w);
  return -1;
}

int main(void)
{
  yyparse();
  printf("there were %d unique words\n",nwords);
}
```

Other things to note:

- The `WORD` token that was used in the *lex* code is defined here in the definitions section; *lex* knows about it through including the `words.h` file.
- The *lex* rule also sets a variable `yylval`; this puts a value on the stack top, where *yacc* can find it with `$1`, `$2`, et cetera.

All of this will be explained in detail below.

4 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; see section **??**.

5 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` every time 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.

5.1 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 -o`, a header file `<file>.h`[1] 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 : NUMBER '+' NUMBER
```

The *yacc* code now recognizes a `sum` if *lex* returns in sequence a `NUMBER` token, a plus character, and another `NUMBER` token.

See example **??** for a worked out code.

5.2 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* parser can return a value that is put on top of the stack, so that *yacc* can access it. This symbol is returned in the variable `yylval`. By default, this is defined as an `int`, so the *lex* program would have

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

See section **??** for how the stack values are used by *yacc*.

If more than just integers need to be returned, the specifications in the *yacc* code become more complicated. Suppose we are writing a calculator with variables, so 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:

---

1. If you leave out the `-o` option to *yacc*, the file is called `y.tab.h`.

```
%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 **??**.

6 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 defines coming from `%token` definitions in the *yacc* program or character values; see section **??**.

A simple example illustrating the ideas in this section can be found in section **??**.

6.1 Rule actions

The example in section **??** had such rules as:

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

The action belonging to the different right hand sides refer to $\$n$ quantities and to `$$`. The latter refers to the stack top, so by assigning to it a new item is put on the stack top. The former variables are assigned the values on the top of the stack: if the right hand side has three terms, terminal or nonterminal, then `$1` through `$3` are assigned and the three values are removed from the stack top.

7 Operators; precedence and associativity

The example in section **??** had separate rules for addition/subtraction and multiplication/division. We could simplify the grammar by writing

```
expr:
        expr '+' expr ;
        expr '-' expr ;
        expr '*' expr ;
        expr '/' expr ;
        expr '^' expr ;
        number ;
```

but this would have `1+2*3` evaluate to 9. In order to indicate operator precedence, we can have lines

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

The sequence of lines indicates increasing operator precedence and the keyword sets the associativity type: we want `5-1-2` to be 2, so minus is left associative; we want `2^2^3` to be 256, not 64, so exponentiation is right associative.

Operators that can be both unary and binary are handled by declaring a non-associative token, and explicitly indicating its precedence.

```
%left '-' '+'
%nonassoc UMINUS
%
expression : expression '+' expression
    | expression '-' expression
    | '-' expression %prec UMINUS
```

8 Further remarks

8.1 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 **??**.

Exercise 1. Try to write *lex* or *yacc* programs for the following languages:
$a^n b^m$, $a^n b^n$, $a^n b^n c^n$
Discuss the theoretical power of *lex* and *yacc*.

8.2 Errors and tracing

So far we have assumed that the input to *yacc* is syntactically correct, and *yacc* need only discover its structure. However, occasionally input will be incorrect.

8.2.1 Tracing

If you assign `yydebug=1;`, *yacc* will produce trace output. While its states may not make sense to you, at least you will see which tokens it matches, and which rules it can reduce.

8.2.2 Syntax errors

Sometimes, *yacc* reports '`syntax error`' and stops processing. This means that an unexpected symbol is found. A common source for this is the case that you have made a typo in your grammar, and the symbol it is trying to match is not defined. Example: suppose we have just matched an open token:

```
group : open body close
bodytext : ;
         | character bodytext
```

If you are tracing *yacc*'s workings, you will probably see it matching the `character`, then giving the `syntax error` message.

The 'syntax error' message is actually *yacc*'s default implementation of the `yyerror` routine, but it can be redefined at will. For example, suppose we have a declaration

```
int lineno=1;       /* in yacc */
extern int lineno; /* in lex */
```

and every line with `\n` in *lex* increases this variable. We could then define

```
void yyerror(char *s)
{
  printf("Parsing failed in line %d because of %s\n",
         lineno,s);
}
```

8.2.3 Error recovery

Error recovery in *yacc* is possible through the `error` token. In the rule

```
foo : bar baz ;
    | error baz printf("Hope for the best\n");
```

recognizing any token but `bar` will make *yacc* start skipping tokens, hoping to find `baz` and recover from that point. This is not guaranteed to work.

8.2.4 Semantical errors

Both *lex* and *yacc* are stronger than simple finite-state or pushdown automata, for instance if they are endowed with a symbol table. This can be used to detect semantic errors. For instance, while you would like to write

```
array_slice : array_name '[' int_expr ']'
```

you may be limited to

```
array_slice : ident '[' int_expr ']'
                {if (!is_array($1)) { ....
```

There are a couple of tools here:

`yyerror(char*)` is a default write to `stderr`; you can redefine it.
`YYABORT` is a macro that halts parsing.

8.3 Makefile rules for *yacc*

The `make` utility knows about *lex* and *yacc*, but if you want to do things yourself, here are some good rules:

```
# disable normal rules
.SUFFIXES:
.SUFFIXES: .l .y .o

# lex rules
.l.o :
        lex -t $*.l > $*.c
        cc -c $*.c -o $*.o

# yacc rules
.y.o :
        if [ ! -f $*.h ] ; then touch $*.h ; fi
        yacc -d -t -o $*.c $*.y
        cc -c -o $*.o $*.c ;
        rm $*.c

# link lines
lexprogram : $(LEXFILE).o
        cc $(LEXFILE).o -o $(LEXFILE) -ll
yaccprogram : $(YACCFILE).o $(LEXFILE).o
        cc $(YACCFILE).o $(LEXFILE).o -o $(YACCFILE) -ly -ll
```

8.4 The power of *yacc*

Theoretically, *yacc* implements an LALR(1) parser, which is essentially an LR parser with one token look-ahead. This describes a large class of useful grammars. As an example of a grammar with *two* tokens look-ahead, consider

phrase $\longrightarrow$ CART_ANIMAL and cart | WORK_ANIMAL and plow
CART_ANIMAL $\longrightarrow$ horse | goat
WORK_ANIMAL $\longrightarrow$ horse | ex

Now to distinguish between `horse and cart` and `horse and plow` from the word `horse` takes two tokens look-ahead.

Exercise 2. Use the TEX parser you wrote in *lex* to parse LATEX documents. The parser should

- Report the documentclass used;
- Check that `\begin{document}` and `\end{document}` are used, with no text before the begin command;
- Know about some commands with one argument, such as `\textbf`, and properly recognize that argument
- Recognize proper matching of begin/end of an environment.

Bonus: let your parser interpret `\newcommand` correctly. You can limit yourself to the the case of commands with one argument, that is `\newcommand{\foo}[1]{ ...}`

9 Examples

9.1 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 "calc1.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, the following *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 "calc1.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.

9.2 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);}
```

9.3 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 NVARS 100
char *vars[NVARS]; double vals[NVARS]; 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);}
```

Contents