Tutorial de Lex/Yacc[1]

Etienne Bernard (bernard@isia.cma.fr)
http://www.via.ecp.fr/~eb/textes/minimanlexyacc-english.html

Conteúdo

[1]Este tutorial foi convertido no formato LaTeX por Claudio Cesar de Sá (dcc2ccs@joinville.udesc.br), procurando manter o texto original em sua íntegra. Se alguém achar algum erro, grato em avisar-me.

1 The grammar used

Let's write a mini calculator. The grammar needed for such a program is quite simple. We will have to evaluate expressions that can be either a number, either an expression between parenthesis, either an expression added to an other, etc. So, as you can see, the grammar is recursive.

The analysis of the language is made in two passes. The first one is what is called lexical analysis. This is the job of Lex, through the function yylex(), which consums the tokens (see below). This function will inform the syntaxical analyser, generated by Yacc, through the function yyparse().

2 Use of Lex in syntaxical analysis

The purpose of the lexical analysis is to transform a series of symbols into **tokens** (a token may be a number, a "+"sign, a reserved word for the language, etc.). Once this transformation is done, the syntaxical analyser will be able to do its job (see below). So, the aim of the lexical analyser is to consum symbols at to pass them back to the syntaxical analyser. A Lex description file can be divided into three parts, using the following plan:

```
declarations
%%
productions
%%
additionnal code
```

in which no part is required. However, the first %% is required, in order to mark the separation between the declarations and the productions.

2.1 First part of a Lex file: decalarations

This part of a Lex file may contain:

- Code written in the target language (usually C or C++), embraced in %{ and %}, which will be placed at the top of the file that Lex will create. That is the place that we usually put the include files. Lex will put "*as is*"all that is written between these signs in the target file. The two signs will have to be placed at the beginning of the line.

- Regular expressions, defining non-terminal notions, such as letters, digits, and numbers. This specifications have the form:
notion	*regular expression*
  You will be able to use the notions defined this way in the end of the first part of the file, and in the second part of the file, if you embrace them between { and }.

Example :

```
%{
#include "calc.h"
#include <stdio.h>
#include <stdlib.h>
%}
/* Regular expressions */
/* ------------------- */

white           [\t\n ]+
letter          [A-Za-z]
digit10         [0-9]                   /* base 10 */
digit16         [0-9A-Fa-f]             /* base 16 */

identifier      {letter}(_|{letter}|{digit10})*
int10           {digit10}+
```

The example by itself is, I hope, easy to understand, but let's have a deeper look into regular expressions.

2.2 Regular expressions

```
     Symbol  |  Meaning
-------------+------------------
          x  |  The "x" character
          .  |  Any character except \n
      [xyz]  |  Either x, either y, either z
      [^bz]  |  Any character, EXCEPT b and z
      [a-z]  |  Any character between a and z
     [^a-z]  |  Any character EXCEPT those between a and z
         R*  |  Zero R or more; R can be any regular expression
         R+  |  One R or more
         R?  |  One or zero R (that is an optionnal R)
     R{2,5}  |  Two to 5 R
      R{2,}  |  Two R or more
       R{2}  |  Exactly two R
"[xyz\"foo"  |  The string "[xyz"foo"
   {NOTION}  |  Expansion of NOTION, that as been defined above in the file
         \X  |  If X is a "a", "b", "f", "n", "r", "t", or
             |  "v", this represent the ANSI-C interpretation of \X
         \0  |  ASCII 0 character
       \123  |  ASCII character which ASCII code is 123 IN OCTAL
       \x2A  |  ASCII character which ASCII code is 2A in hexadecimal
         RS  |  R followed by S
        R|S  |  R or S
        R/S  |  R, only if followed by S
         ^R  |  R, only at the beginning of a line
         R$  |  R, only at the end of a line
     <<EOF>> | End of file
```

So the definition identifier {letter}(_ | {letter} | digit10})* will recognise as identifiers the words “integer”, “a_variable”, “a1”, but not “_ident”nor “1variable”. Easy, isn’t it? As a last example, this is the definition of a real number:

```
digit           [0-9]
integer         {digit}+
exponant        [eE][+-]?{integer}
real            {integer}("."{integer})?{exponant}?
```

2.3 Second part of a Lex file: productions

This part is aimed to instruct Lex what to do in the generated analyser when it will encounter one notion or another one. It may contain:

- Some specifications, written in the target language (usually C or C++) surrounded by %{ and %} (at the beginning of a line). The specifications will be put at the beginning of the yylex() function, which is the function that consums the tokens, and returns an integer.
- Productions, having the syntax:

regular_expression	*action*

  If action is missing, Lex will put the matching characters as is into the standard output. If action is specified, it has to be written in the target language. If it contains more than one instruction or is written in more than one line, you will have to embrace it between { and }.

You should also note that comment such as /* ... */ can be present in the second part of a Lex file only if enclosed between braces, in the action part of the statements. Otherwise, Lex would consider them as regular expressions or actions, which would give errors, or, at least, a weird behaviour.

Finally, the yytext variable used in the actions contains the characters accepted by the regular expression. This is a char table, of length yyleng (ie, char yytext[yyleng]). Example:

```
%%
[ \t]+$         ;
[ \t]           printf(" ");
```

This little Lex file will generate a program that will suppress the space characters that are not useful. You can also notice with that little program that Lex is not reserved to interpreters or compilers, and can be used, for example, for searches and replaces, etc.

2.4 Third part: additional code

You can put in this part all the code you want. If you don’t put anything here, Lex will consider that it is just:

```
main() {
  yylex();
 }
```

2.5 Conclusion about Lex

As you can see it, Lex is quite easy to use (but it can be more complicated if you use start conditions). We have not seen all the possibilities of Lex, and I suggest that you read its man page for a more detailed information.

3 Syntaxical analysis with Yacc

Yacc (**Y**et **A**nother **C**ompiler **Compiler**) is a program designed to compile a LALR(1) grammar and to produce the source code of the syntaxical analyser of the language produced by this grammar. It is also possible to make it do s emantic actions. As for a Lex file, a Yacc file can be divided into three parts:

```
declarations
%%
productions
%%
additionnal code
```

and only the first %% and the second part are mandatory.

3.1 The first part of a Yacc file

The first part of a Yacc file may contain:

- Specifications written in the target language, enclosed between %{ and %} (each symbol at the begining of a line) that will be put at the top of the scanner generated by Yacc.
- Declaration of the tokens that can be encontered:

  %token TOKEN
- The type of the terminal, using the reserved word: %union
- Informations about operators' priority or associativity.
- The axiom of the grammar, using the reserved word %start (if not specified, the axiom is the first production of the second part of the file).

The yylval variable, *implicitely* declared of the %union type is really important in the file, since it is the variable that contains the description of the last token read.

3.2 Second part of a Yacc file

This part can not be empty. It may contain:

- Declarations and/or definitions enclosed between %{ and %}.
- Productions of the language's grammar. They look like:

```
nonterminal_notion:
                          body_1        { semantical_action_1 }
                        | body_2        { semantical_action_2 }
                        | ...
                        | body_n        { semantical_action_n }
                        ;
```

  provided that the "body_i"may be terminal or nonterminal notions of the language.

And finally...

3.3 Third part of a Yacc file

This part contains the additional code, must contain a main() function (that should call the yyparse() function), and an yyerror(char ∗message) function, that is called when a syntax error is found.

3.4 Conclusion about Yacc

This presentation is far from being exhaustive, and I didn't explain you everything. We will clarify some points in the following example.

4 An example: a little expression interpreter

This interpreter is able to calculate the value of any mathematical expression written with real numbers, the operators "+", "−"(binary and unary operators), "∗", "/", "^ "(power operator). This interpreter will not be able to use variables or functions.

4.1 The Lex part of the interpreter

Here is the source:

```
%{
#include "global.h"
#include "calc.h"
#include <stdlib.h>
%}

white           [  \t]+
```

```
digit            [0-9]
integer          {digit}+
exponant         [eE][+-]?{integer}

real             {integer}("."{integer})?{exponant}?

%%

{white}          { /* We ignore white characters */ }

{real}           {
                   yylval=atof(yytext);
                   return(NUMBER);
                 }

"+"            return(PLUS);
"-"            return(MINUS);

"*"            return(TIMES);
"/"            return(DIVIDE);

"^"            return(POWER);

"("            return(LEFT_PARENTHESIS);
")"            return(RIGHT_PARENTHESIS);

"\n"  return(END);
```

Explaination:

1. The first part includes the file calc.h, that will be generated later by Yacc and that will contain the definition for NUMBER, PLUS, MINUS, etc... We include the stdlib header, because we will use the atof() function after. We declare the real notion used in the second part. The global.h file contains only the #define YYSTYPE double declaration, because all the structures we will manipulate have the type double. By the way, this is the type of yylval.

2. The second part tells the syntaxical parser which type of token it encountered. If it is a number, we put its value in the yylval variable, in order to be used later.

3. Finally, the third part is empty, because we do not want Lex to create a main() function, that will be declared in the Yacc file.

4.2 The Yacc part of the interpreter

This is the most important, and the most interesting:

```
%{
```

```
#include "global.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
%}

%token  NUMBER
%token  PLUS    MINUS   TIMES   DIVIDE  POWER
%token  LEFT_PARENTHESIS        RIGHT_PARENTHESIS
%token  END

%left   PLUS    MINUS
%left   TIMES   DIVIDE
%left   NEG
%right  POWER

%start Input
%%

Input:
          /* Empty */
        | Input Line
        ;

Line:
          END
        | Expression END                { printf("Result: %f\n",$1); }
        ;

Expression:
          NUMBER                        { $$=$1; }
        | Expression PLUS Expression    { $$=$1+$3; }
        | Expression MINUS Expression   { $$=$1-$3; }
        | Expression TIMES Expression   { $$=$1*$3; }
        | Expression DIVIDE Expression  { $$=$1/$3; }
        | MINUS Expression %prec NEG    { $$=-$2; }
        | Expression POWER Expression   { $$=pow($1,$3); }
        | LEFT_PARENTHESIS Expression RIGHT_PARENTHESIS { $$=$2; }
        ;

%%

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

int main(void) {
```

```
  yyparse();
}
```

Well, it seems a little less simple, isn't it? In fact, it is not so complicated. We include the usual files, and we use the %token keyword to declare the tokens that we can find. There is, in this case, no particular order for the declaration.

Then we have the %left and %right keywords. This is used to tell Yacc the associativity of the operators, and their priority. Then, we define the operators in an increasing order of priority. So, "$1 + 2 * 3$"is evaluated as "$1 + (2 * 3)$". You will have to choose between "*left*"or "*right*"declaration for your operator. For a left-associative operator (%left - "+"in this example), a+b+c will be evaluated as $(a + b) + c$. For a right-associative operator ("^ "here), a^b^c will be evaluated as a^(b^c).

Then will tell Yacc that the axiom will be Input, that is its state will be such as it will consider any entry as an Input, at the begining. You should also note the recursivity in the definition of Input. It is used to treat an entry which size is unknown. For internal reason to Yacc, you should use:

```
Input:
                  /* Empty */
                | Input Line
                ;
instead of
Input:
                  /* Empty */
                | Line Input
                ;
```

(This permits a reduction as soon as possible).

Let's have a look to the definition of Line. The definition itself is quite simple, but you should ask yourself what represents the $1. In fact, $1 is a reference to the value returned to the first notion of the production. It is similar for $2, $3, ... And $$ is the value returned by the production. So, in the definition of Expression, $$=$1+$3 adds the value of the first expression to the value of the second expression (this is the third notion) and returns the result in $$.

If you have a look to the definition of the unary minus, the %*prec* keyword is used to tell Yacc that the priority is that of NEG.

Finally, the third part of the file is simple, since it just calls the yyparse() function.

4.3 Compiling and running the example

Provided that the Lex file is called calc.lex, and the Yacc file calc.y, all you have to do is:

```
>bison -d calc.y
>mv calc.tab.h calc.h
>mv calc.tab.c calc.y.c
>flex calc.lex
>mv lex.yy.c calc.lex.c
```

```
>gcc -c calc.lex.c -o calc.lex.o
>gcc -c calc.y.c -o calc.y.o
>gcc -o calc calc.lex.o calc.y.o -lfl -lm    [eventually -ll]
```

O sufixo "*-l*" nos parâmetros da compilação, indicam que bibliotecas serão adicionadas na etapa da linkedição. No caso: "*fl*" é a flex, e "*m*" é a de recursos matemáticos da linguagem C.

Please note that you need to create a file called global.h which will contain:

```
#define YYSTYPE double
extern YYSTYPE yylval;
```

I only have bison and flex instead of yacc and lex, but it should be the same, except the file names.

The call to bison with the "$-d$"parameter creates the calc.tab.h header file, which defines the tokens. We call flex and we rename the files we get. Then, you only have to compile, and do not forget the proper libraries. We get:

```
>calc
 1+2*3
 Result: 7.000000
 2.5*(3.2-4.1^2)
 Result: -34.025000
```

4.4 A better calculator

When there is a syntax error, the program stops. In order to continue, we may replace

```
Line:
          END
        | Expression END                { printf("Result: %f\n",$1); }
        ;
by
Line:
          END
        | Expression END                { printf("Result %f\n",$1); }
        | error END                     { yyerrok; }
        ;
```

but, of course, it is only an idea and there are many others (utilisation and definition of variables and functions, many data types, etc.).

5 Conclusion

I hope that this little tutorial about Lex and Yacc answers many of the questions you should ask yourself. However, it is not complete. Do not hesitate to read the man pages for Lex and Yacc, and if you have questions, you can send me a mail to bernard@isia.cma.fr.