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:

- 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`.
- 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.
- 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:
       /* Vide */
     | 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 -ll -lm [and maybe -lfl]
```

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 (usage and definition of variables and functions, many data types, etc.).

Previous: Syntaxical analysis with Yacc **Table of contents** **Conclusion**

Copyright © 2001-2002 Etienne Bernard (eb at pltplp dot net)