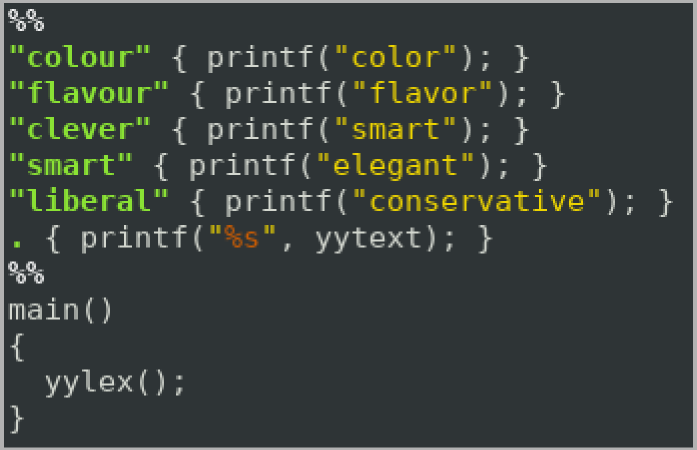
flex is not a bad tool to use for doing modest text transformations and for programs that collect statistics on input.



```
%%
"colour" { printf("color"); }
"flavour" { printf("flavor"); }
"clever" { printf("smart"); }
"smart" { printf("elegant"); }
"liberal" { printf("conservative"); }
. { printf("%s", yytext); }
%%
main()
{
  yylex();
}
```

More often than not, though, you'll want to use flex to generate a scanner that divides the input into tokens that are then used by other parts of your program.

Associativity and Precedence

Operator precedence. Operator precedence specifies the manner in which operands are grouped with operators. For example, 1 + 2 * 3 is treated as 1 + (2 * 3), whereas 1 * 2 + 3 is treated as (1 * 2) + 3 because the multiplication operator has a higher precedence than the addition operator. You can use parentheses to override the default operator precedence rules.

Operator associativity. When an expression has two operators with the same precedence, the operators and operands are grouped according to their associativity. For example 72 / 2 / 3 is treated as (72 / 2) / 3 since the division operator is left-to-right associate. You can use parentheses to override the default operator associativity rules.

Operators may be associative (meaning the operations can be grouped arbitrarily), left-associative (meaning the operations are grouped from the left), right-associative (meaning the operations are grouped from the right) or non-associative (meaning operations cannot be chained, often because the output type is incompatible with the input types).

Associativity and Precedence examples

(1-2)-3 = -4
1-(2-3) = 2

In order to reflect normal usage, addition, subtraction, multiplication, and division operators are usually left-associative

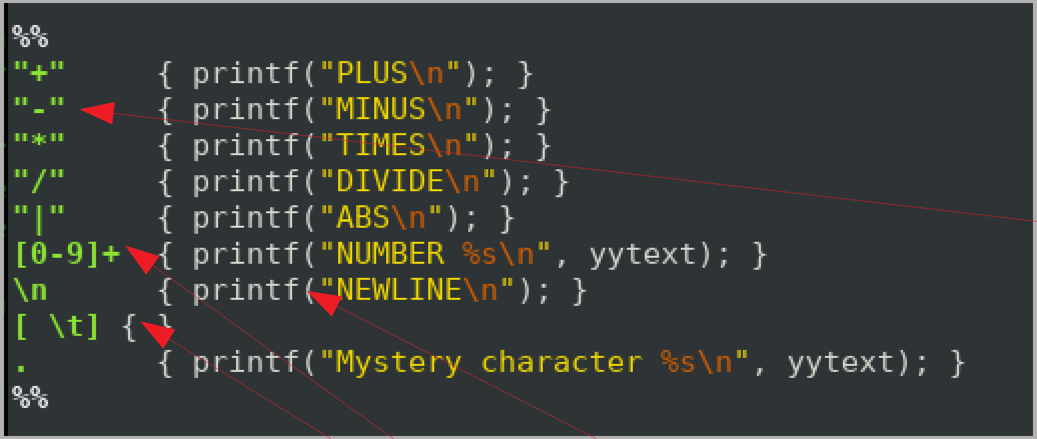
2*(3+4) = 14
(2*3)+4 = 10

(2^3)^4 = 4096
2^(3^4) = 2417851639229258349412352

for an exponentiation operator there is no general agreement

Calculator Program

We'll start by recognizing only integers, four basic arithmetic operators, and a unary absolute value operator



```
%%
"+"     { printf("PLUS\n"); }
"-"     { printf("MINUS\n"); }
"*"     { printf("TIMES\n"); }
"/"     { printf("DIVIDE\n"); }
"|"     { printf("ABS\n"); }
[0-9]+  { printf("NUMBER %s\n", yytext); }
\n      { printf("NEWLINE\n"); }
[ \t] { }
.       { printf("Mystery character %s\n", yytext); }
%%
```

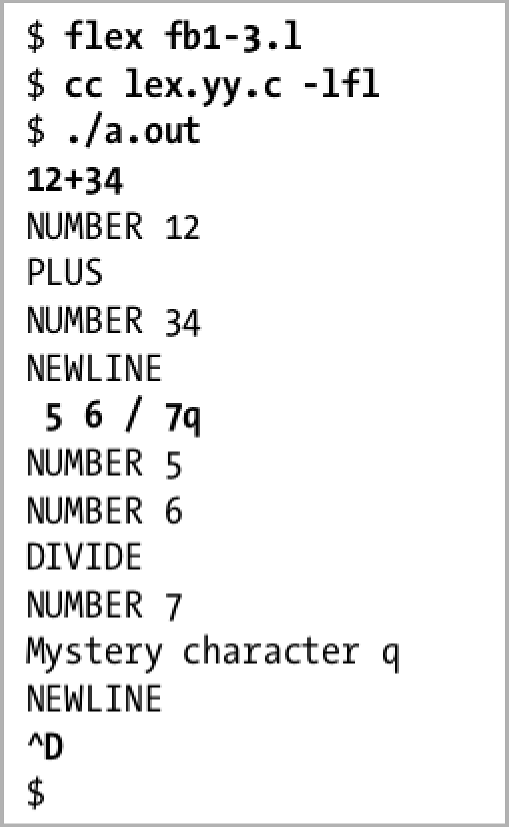
The first five patterns are literal operators, written as quoted strings, and the actions, for now, just print a message saying what matched. The quotes tell flex to use the strings as is, rather than interpreting them as regular expressions.

The sixth pattern matches an integer. The bracketed pattern [0-9] matches any single digit, and the following + sign means to match one or more of the preceding item, which here means a string of one or more digits. The action prints out the string that's matched, using the pointer yytext that the scanner sets after each match.

The seventh pattern matches a newline character, represented by the usual C \n sequence.

The eighth pattern ignores whitespace. It matches any single space or tab (\t), and the empty action code does nothing.

In this simple flex program, there's no C code in the third section. The flex library (-lfl) provides a tiny main program that just calls the scanner, which is adequate for this example.



```
$ flex fb1-3.l
$ cc lex.yy.c -lfl
$ ./a.out
12+34
NUMBER 12
PLUS
NUMBER 34
NEWLINE
 5 6 / 7q
NUMBER 5
NUMBER 6
DIVIDE
NUMBER 7
Mystery character q
NEWLINE
^D
$
```

By default, the terminal will collect input from the user until he presses Enter/Return.

Then the whole line is pushed to the input filestream of your program.

This is useful because your program does not have to deal with interpreting all keyboard events (e.g. remove letters when Backspace is pressed).

Scanner as Coroutine

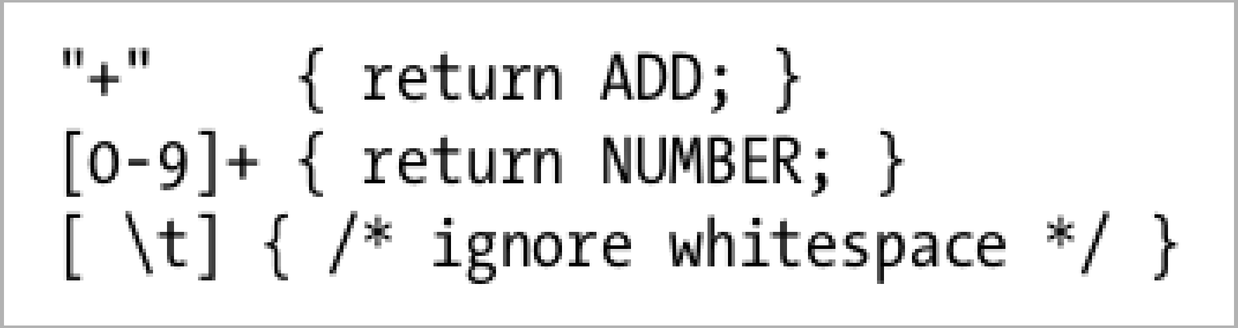
Coroutines are computer-program components that allow multiple entry for suspending and resuming execution at certain locations.

Most programs with flex scanners use the scanner to return a stream of tokens that are handled by a parser.

Each time the program needs a token, it calls yylex() , which reads
a little input and returns the token.

When it needs another token, it calls yylex() again. The scanner acts as a coroutine; that is, each time it returns, it remembers where it was, and on the next call it picks up where it left off.

The rule is actually quite simple: If action code returns, scanning resumes on the next call to yylex() ; if it doesn’t return, scanning resumes immediately.

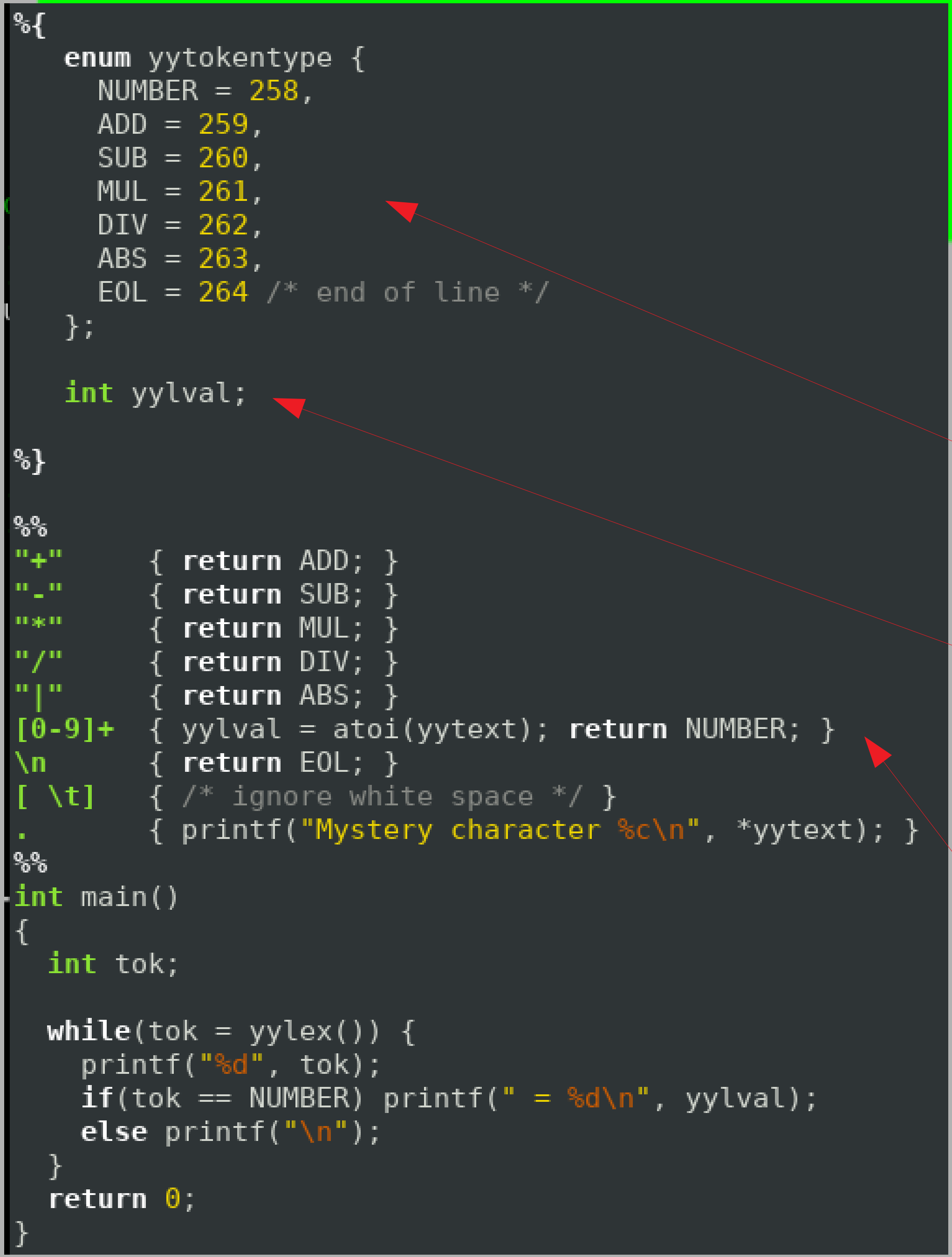


```
"+"    { return ADD; }
[0-9]+ { return NUMBER; }
[ \t] { /* ignore whitespace */ }
```

Tokens and Values

When a flex scanner returns a stream of tokens, each token actually has two parts, the token and the token's value. The token is a small integer. The token numbers are arbitrary, except that token zero always means end-of-file. When bison creates a parser, bison assigns the token numbers automatically starting at 258 (this avoids collisions with literal character tokens, discussed later) and creates a .h with definitions of the tokens numbers. But for now, we'll just define a few tokens by hand:

```
NUMBER = 258,
ADD = 259,
SUB = 260,
MUL = 261,
DIV = 262,
ABS = 263,
EOL = 264 end of line
```

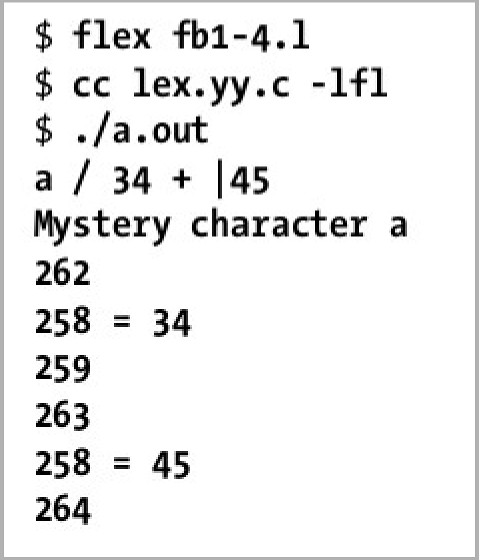


```
%{
   enum yytokentype {
     NUMBER = 258,
     ADD = 259,
     SUB = 260,
     MUL = 261,
     DIV = 262,
     ABS = 263,
     EOL = 264 /* end of line */
   };

   int yylval;

%}

%%
"+"     { return ADD; }
"-"     { return SUB; }
"*"     { return MUL; }
"/"     { return DIV; }
"|"     { return ABS; }
[0-9]+  { yylval = atoi(yytext); return NUMBER; }
\n      { return EOL; }
[ \t]   { /* ignore white space */ }
.       { printf("Mystery character %c\n", *yytext); }
%%
int main()
{
  int tok;

  while(tok = yylex()) {
    printf("%d", tok);
    if(tok == NUMBER) printf(" = %d\n", yylval);
    else printf("\n");
  }
  return 0;
}
```

We define the token numbers in a C enum

make yylval , the variable that stores the token value, an integer, which is adequate for the first version of our calculator.

For each of the tokens, the scanner returns the appropriate code for the token; for numbers, it turns the string of digits into an integer and stores it in yylval before returning



```
$ flex fb1-4.l
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
```