

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

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.

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

## 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] { }  
.  
%%
```

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 fb1-3.1
$ 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 fb1-4.1
$ cc lex.yy.c -lfl
$ ./a.out
a / 34 + |45
Mystery character a
262
258 = 34
259
263
258 = 45
264
```





An expression is a finite combination of symbols that is well-formed according to some rules

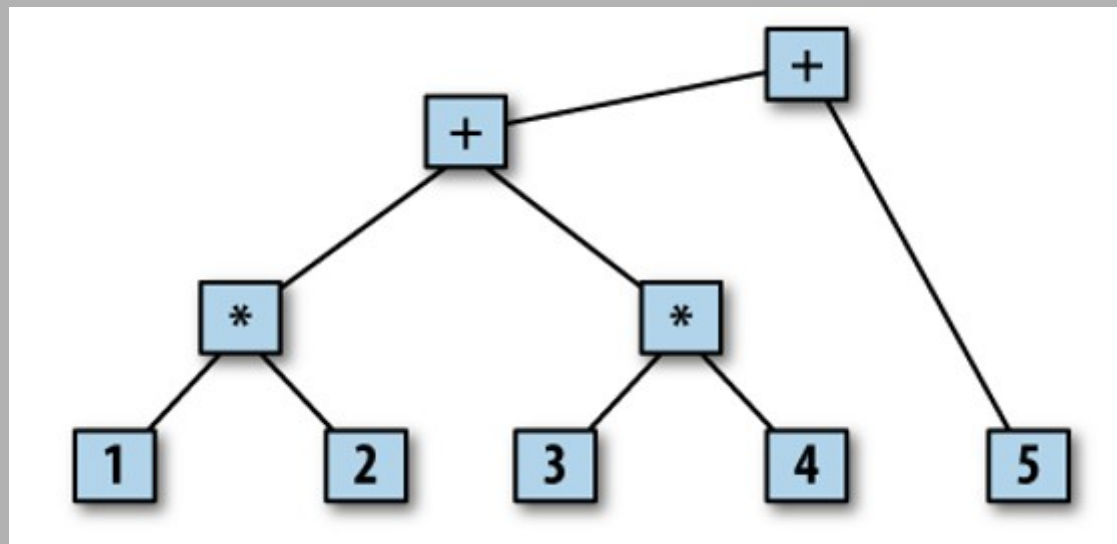
Terms are those parts of the expression between addition signs and subtraction signs.

Factors are the separate parts of a multiplication or division.

## Grammars and Parsing

The parser's job is to figure out the relationship among the input tokens. A common way to display such relationships is a parse tree.

For example, under the usual rules of arithmetic, multiplication has higher precedence than addition, the arithmetic expression  $1 * 2 + 3 * 4 + 5$  would have the parse tree



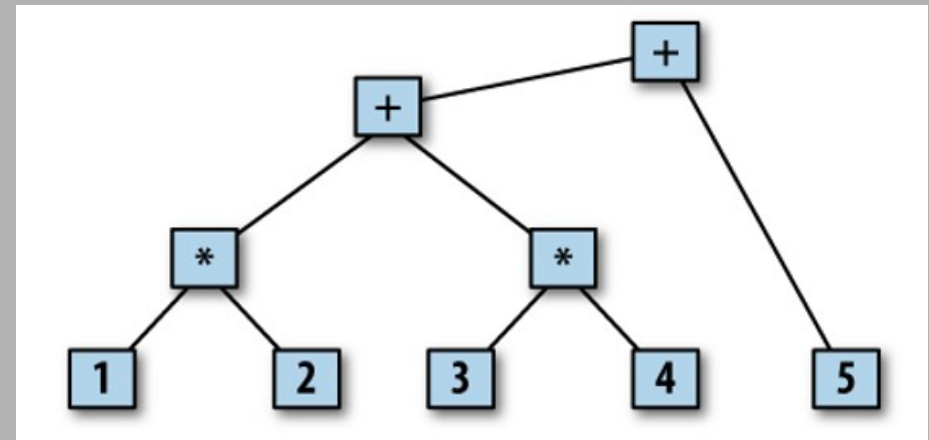
## Backus-Naur Form


Backus-Naur Form (BNF), created around 1960 to describe Algol 60 and named after two members of the Algol 60 committee

In order to write a parser, we need some way to describe the rules, the grammar, the parser uses to turn a sequence of tokens into a parse tree.

1 \* 2 + 3 \* 4 + 5

$\langle \text{exp} \rangle ::= \langle \text{factor} \rangle$   
          |  $\langle \text{exp} \rangle + \langle \text{factor} \rangle$   
 $\langle \text{factor} \rangle ::= \text{NUMBER}$   
          |  $\langle \text{factor} \rangle * \text{NUMBER}$






A parser is a software component that takes input data (frequently text) and builds a data structure - often some kind of parse tree, abstract syntax tree or other hierarchical structure, giving a structural representation of the input while checking for correct syntax

Bison is a general-purpose parser generator that converts a grammar description (Bison Grammar Files) into a C program to parse that grammar.

The Bison parser is a bottom-up parser. It tries, by shifts and reductions, to reduce the entire input down to a single grouping whose symbol is the grammar's start-symbol.



As Bison reads tokens, it pushes them onto a stack along with their semantic values. The stack is called the parser stack. Pushing a token is traditionally called shifting.

When the last  $n$  tokens and groupings shifted match the components of a grammar rule, they can be combined according to that rule. This is called reduction.

Those tokens and groupings are replaced on the stack by a single grouping whose symbol is the result (left hand side) of that rule.

Running the rule's action is part of the process of reduction, because this is what computes the semantic value of the resulting grouping.



In order for Bison to parse a language, it must be described by a grammar.

This means that you specify one or more syntactic groupings and give rules for constructing them from their parts.

For example, in the *C* language, one kind of grouping is called an 'expression'.

One rule for making an expression might be, "An expression can be made of a minus sign and another expression".

Another would be, "An expression can be an integer". As you can see, rules are often recursive, but there must be at least one rule which leads out of the recursion.

```

%{
# include <stdio.h>
int yylex();
void yyerror(char *s);
%}

/* declare tokens */
%token NUMBER
%token ADD SUB MUL DIV ABS
%token EOL
%%
calclist: /* nothing */
| calclist exp EOL { printf("= %d\n> ", $2); }
;

exp: factor
| exp ADD factor { $$ = $1 + $3; }
| exp SUB factor { $$ = $1 - $3; }
;

factor: term
| factor MUL term { $$ = $1 * $3; }
| factor DIV term { $$ = $1 / $3; }
;

term: NUMBER
| ABS term { $$ = $2 >= 0? $2 : - $2; }
;
%%
int main()
{
printf("> ");
yyparse();
return 0;
}

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

```

Bison programs have the same three-part structure as flex programs, with declarations, rules, and C code.

token declarations, telling bison the names of the symbols in the parser that are tokens.

By convention, tokens have uppercase names, although bison doesn't require it.

Any symbols not declared as tokens have to appear on the left side of at least one rule in the program.

```

%{
# include <stdio.h>
int yylex();
void yyerror(char *s);
%}

/* declare tokens */
%token NUMBER
%token ADD SUB MUL DIV ABS
%token EOL
%%
calclist: /* nothing */
| calclist exp EOL { printf("= %d\n> ", $2); }
;

exp: factor
| exp ADD factor { $$ = $1 + $3; }
| exp SUB factor { $$ = $1 - $3; }
;

factor: term
| factor MUL term { $$ = $1 * $3; }
| factor DIV term { $$ = $1 / $3; }
;

term: NUMBER
| ABS term { $$ = $2 >= 0? $2 : - $2; }
;
%%
int main()
{
    printf("> ");
    yyparse();
    return 0;
}

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

```

The second section contains the rules in simplified BNF. Bison uses a single colon rather than ::= , and since line boundaries are not significant, a semicolon marks the end of a rule.

Again, like flex, the C action code goes in braces at the end of each rule.





Each symbol in a bison rule has a value;

the value of the target symbol (the one to the left of the colon) is called \$\$ in the action code, and

the values on the right are numbered \$1 , \$2 , and so forth, up to the number of symbols in the rule.

The values of tokens are whatever was in yylval when the scanner returned the token;

the values of other symbols are set in rules in the parser.

In this parser, the values of the factor, term, and exp symbols are the value of the expression they represent.

```

%{
#include "fb1-5.tab.h"
void yyerror(char *s);
}%

%%
"+"      { return ADD; }
"-"      { return SUB; }
"*"      { return MUL; }
"/"      { return DIV; }
"|"      { return ABS; }
"[0-9]+" { yylval = atoi(yytext); return NUMBER; }

\n       { return EOL; }
[ \t]    { /* ignore white space */ }
.        { yyerror("Mystery character\n"); }
%%

```

Rather than defining explicit token values in the first part, we include a header file that bison will create for us, which includes both definitions of the token Numbers and a definition of `yylval`.

We also delete the testing main routine in the third section of the scanner, since the parser will now call the scanner.

```
bison -d fb1-5.y ; flex fb1-5.l ; gcc fb1-5.tab.c lex.yy.c -lfl
```

`-d` write an extra output file containing macro definitions for the token type names


One of the nicest things about using flex and bison to handle a program's input is That it's often quite easy to make small changes to the grammar.

Our expression language would be a lot more useful if it could handle parenthesized expressions, and it would be nice if it could handle comments, using `//` syntax.

To do this, we need only add one rule to the parser and three to the scanner.

```
%token OP CP in the declaration section
...
%%
term: NUMBER
    | ABS term { $$ = $2 >= 0? $2 : - $2; }
    | OP exp CP { $$ = $2; } New rule
    ;
```

```
"("      { return OP; }
")"      { return CP; }
"//".*   /* ignore comments */
```



Since a dot matches anything except a newline, `.*` will gobble up the rest of the line.