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 of division.
Parsing, syntax analysis or syntactic analysis is the process of analysing a string of symbols conforming to the rules of a grammar.

The term parsing comes from Latin pars meaning part.

A parse tree represents the syntactic structure of a string according to some grammar.

A grammar is a set of production rules for strings in a language. The rules describe how to form strings from the language's alphabet that are valid according to the language's syntax.

A grammar does not describe the meaning of the strings or what can be done with them only their form.
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 $1 \times 2 + 3 \times 4 + 5$ would have the parse tree
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
```

```
<exp> ::= <factor>
| <exp> + <factor>

<factor> ::= NUMBER
| <factor> * NUMBER
```

Diagram:
```
<exp> ::= <factor>
| <exp> + <factor>
```

```
<factor> ::= NUMBER
| <factor> * NUMBER
```

```
Start symbol
```

```
Production Rule
```

```
Nonterminal
```

```
Terminal
```

```
metasymbol
```

```
1 2 3 4 5
```
1 * 2 + 3 * 4 + 5

An expression is a whole load of things added together. If these things are NUMBERS then just do the adding

```
<exp> ::= <factor>
   | <exp> + <factor>
```

However these things might each be two or more things multiplied together and in this case do the multiplication(s) first

```
<factor> ::= NUMBER
   | <factor> * 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.
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.

Default action $$ = $1

Bison programs handle nesting
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.
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.

```
#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; }

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

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 defined in the grammar
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.

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

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