

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.




$1 * 2 + 3 * 4 + 5$

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

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

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


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



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.

Default action $$$ = \1

Bison programs handle nesting

```

%{
# 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.



The Bison parser detects a syntax error (or parse error) whenever it reads a token which cannot satisfy any syntax rule. An action in the grammar can also explicitly proclaim an error, using the macro `YYERROR`

The Bison parser expects to report the error by calling an error reporting function named `yyerror`, which you must supply. It is called by `yyparse` whenever a syntax error is found, and it receives one argument. For a syntax error, the string is normally "syntax error".


```
%{
# 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 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.

```
%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.

Bottom-Up (Shift-Reduce) Parsing

In bottom-up parsing we start with the sentence and try to apply the production rules in reverse, in order to finish up with the start symbol of the grammar. This corresponds to starting at the leaves of the parse tree, and working back to the root.

Each application of a production rule in reverse is known as a reduction.
The r.h.s. of a rule to which a reduction is applied is known as a handle.


Thus the operation of a bottom-up parser will be as follows:

Start with the sentence to be parsed as the initial sentential form

Until the sentential form is the start symbol do:

Scan through the input until we recognise something that corresponds to the r.h.s. of one of the production rules (this is called a handle)

Apply a production rule in reverse; ie. replace the r.h.s. of the rule which appears in the sentential form with the l.h.s. of the rule
(an action known as a reduction)



In step 2(a) above we are shifting the input symbols to one side as we move through them; hence a parser which operates by repeatedly applying steps 2(a) and 2(b) above is known as a shift-reduce parser.

A shift-reduce parser is most commonly implemented using a stack, where we proceed as follows:

- start with the sentence to be parsed on top of the stack

- a "shift" action corresponds to pushing the current input symbol onto the stack

- a "reduce" action occurs when we have a handle on top of the stack.

- To perform the reduction, we pop the handle off the stack and replace it with the terminal on the l.h.s. of the corresponding rule.

In a bottom-up shift-reduce parser there are two decisions:

- Should we shift another symbol, or reduce by some rule?

- If reduce, then reduce by which rule?

Shift/reduce conflicts

The Dangling else is the classic problem with "C" style if statements. These statements are difficult to describe in a way which is not ambiguous. Consider:

```
if (expr1) if (expr2) statement1; else statement2;
```

We know that the else must match the second if, so the above is equivalent to:

```
if (expr1) { if (expr2) statement1; else statement2; }
```

But the grammar also matches the other possible parse, equivalent to:

```
if (expr1) { if (expr2) statement1; } else statement2;
```



Bison does the right thing here, by design: it always prefers "shift" over "reduce".

What that means is that if an else could match an open if statement, bison will always do that, rather than holding onto the else to match some outer if statement.

The problem with this solution is that you still end up with a warning about shift/reduce conflicts, and it is hard to distinguish between "OK" conflicts, and newly-created "not OK" conflicts.

Bison provides the %expect declaration so you can tell it how many conflicts you expect, which will suppress the warning if the right number are found, but that is still pretty fragile.