

Bison

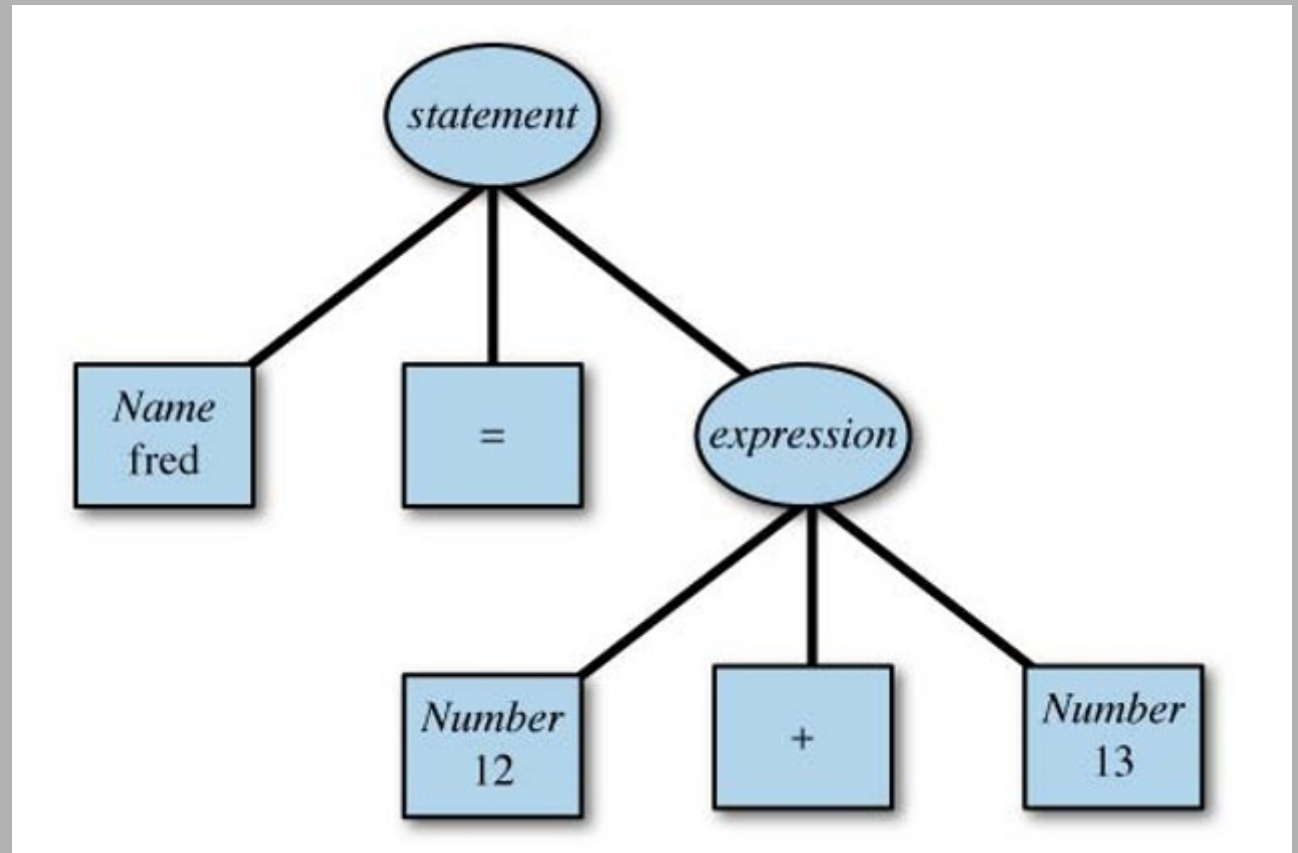
statement:

NAME '=' expression


expression:

NUMBER '+' NUMBER
| NUMBER '-' NUMBER

fred = 12 + 13



Every grammar includes a start symbol, the one that has to be at the root of the parse tree. In this grammar, statement is the start symbol.



A bison parser works by looking for rules that might match the tokens seen so far.

When bison processes a parser, it creates a set of states, each of which reflects a possible position in one or more partially parsed rules.

As the parser reads tokens, each time it reads a token that doesn't complete a rule, it pushes the token on an internal stack and switches to a new state reflecting the token it just read. This action is called a shift.

When it has found all the symbols that constitute the right-hand side of a rule, it pops the right-hand side symbols off the stack, pushes the left-hand side symbol onto the stack, and switches to a new state reflecting the new symbol on the stack. This action is called a reduction

Calculator that builds an AST: header fb3-1.h

```
extern int yylineno; /* from lexer */
void yyerror(char *s, ...);
int yyparse();
int yylex();

/* nodes in the Abstract Syntax Tree */
struct ast {
    int nodetype;
    struct ast *l;
    struct ast *r;
};

struct numval {
    int nodetype;
    double number;
}; /* type K */


/* build an AST */
struct ast *newast(int nodetype, struct ast *l, struct ast *r);
struct ast *newnum(double d);

/* evaluate an AST */
double eval(struct ast *);

/* delete and free an AST */
void treefree(struct ast *);
```

yyerror is slightly enhanced to take multiple arguments in the style of printf

The AST consists of nodes, each of which has a node type. Different nodes have different fields, but for now we have just two kinds, one that has pointers to up to two subnodes and one that contains a number.



The ellipses mean that there are a variable number of arguments following.
The place you will have used them are the printf family of functions.

They allow you to create functions of that style where the parameters are not known beforehand, and you can use the varargs functions (va_start, va_arg and va_end) to get at the specific arguments.

Union

A union is a special data type available in *C* that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

```
union Data {  
    int i;  
    float f;  
    char str[20];  
} data;
```

```
data.f = 220.5;  
printf( "data.f : %f\n", data.f);
```

```

%{
# include <stdio.h>
# include <stdlib.h>
# include "fb3-1.h"
%}

%union {
    struct ast *a;
    double d;
}

/* declare tokens */
%token <d> NUMBER
%token EOL

%type <a> exp factor term

%%

calclist: /* nothing */
| calclist exp EOL {
    printf("= %4.4g\n", eval($2));
    treefree($2);
    printf("> ");
}

| calclist EOL { printf("> "); } /* blank line or a comment */
;

exp: factor
| exp '+' factor { $$ = newast('+', $1,$3); }
| exp '-' factor { $$ = newast('-', $1,$3); }
;

factor: term
| factor '*' term { $$ = newast('*', $1,$3); }
| factor '/' term { $$ = newast('/', $1,$3); }
;

term: NUMBER { $$ = newnum($1); }
| '|' term { $$ = newast('|', $2, NULL); }
| '(' exp ')' { $$ = $2; }
| '-' term { $$ = newast('M', $2, NULL); }
;

%%

```

double in either normal or exponential notation, whichever is more appropriate for its magnitude

In a bison parser, every symbol, both tokens and nonterminals, can have a value associated with it. By default, the values are all integers, but useful programs generally need more sophisticated values. The %union construct is used to create a C language union declaration for symbol values. In this case, the union has two members; a, which is a pointer to an AST, and d, which is a double precision number.

%option nodefault at the top of the scanner to tell it not to add a default rule and rather to report an error if the input rules don't cover all possible input

```
%option noyywrap nodefault yylineno
%{
# include "fb3-1.h"
# include "fb3-1.tab.h"
%}

/* float exponent */
EXP      ([Ee][-+]?[0-9]+)

%%
"+"      |
"_"      |
"*"      |
"/"      |
"|"      |
"("      |
")"      { return yytext[0]; }
[0-9]+ "." [0-9]* {EXP}? |
"."? [0-9]+ {EXP}? { yylval.d = atof(yytext); return NUMBER; }

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

The %yylineno option tells flex to define an integer variable called yylineno and to maintain the current line number in it.

Lex and flex have always come with a small library now known as -lfl that defines a default main routine, as well as a default version of yywrap, a wart left over from the earliest days of lex.

```

%option noyywrap nodefault yylineno
%{
# include "fb3-1.h"
# include "fb3-1.tab.h"
%}

/* float exponent */
EXP      ([Ee][-+]?[0-9]+)

%%
"+"      |
"-"      |
"*"      |
"/"      |
"|"      |
"("      |
")"      { return yytext[0]; }
[0-9]+ "." [0-9]* {EXP}? |
"."? [0-9]+ {EXP}? { yylval.d = atof(yytext); return NUMBER; }

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

```

Rather than giving every token a name, it's also possible to use a single quoted character as a token, with the ASCII value of the token being the token number. (Bison starts the numbers for named tokens at 258, so there's no problem of collisions.) By convention, literal character tokens are used to represent input tokens consisting of the same character; for example, the token '+' represents the input token +, so in practice they are used only for punctuation and operators.


```

struct ast *
newast(int nodetype, struct ast *l, struct ast *r)
{
    struct ast *a = malloc(sizeof(struct ast));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = nodetype;
    a->l = l;
    a->r = r;
    return a;
}

struct ast *
newnum(double d)
{
    struct numval *a = malloc(sizeof(struct numval));

    if(!a) {
        yyerror("out of space");
        exit(0);
    }
    a->nodetype = 'K';
    a->number = d;
    return (struct ast *)a;
}

```

```

double
eval(struct ast *a)
{
    double v;

    switch(a->nodetype) {
        case 'K': v = ((struct numval *)a)->number; break;

        case '+': v = eval(a->l) + eval(a->r); break;
        case '-': v = eval(a->l) - eval(a->r); break;
        case '*': v = eval(a->l) * eval(a->r); break;
        case '/': v = eval(a->l) / eval(a->r); break;
        case '|': v = eval(a->l); if(v < 0) v = -v; break;
        case 'M': v = -eval(a->l); break;
        default: printf("internal error: bad node %c\n", a->nodetype);
    }
    return v;
}

void
treefree(struct ast *a)
{
    switch(a->nodetype) {

        /* two subtrees */
        case '+':
        case '-':
        case '*':
        case '/':
            treefree(a->r);

        /* one subtree */
        case '|':
        case 'M':
            treefree(a->l);

        /* no subtree */
        case 'K':
            free(a);
            break;

        default: printf("internal error: free bad node %c\n", a->nodetype);
    }
}

```

```
void
yyerror(char *s, ...)
{
    va_list ap;
    va_start(ap, s);

    fprintf(stderr, "%d: error: ", yylineno);
    vfprintf(stderr, s, ap);
    fprintf(stderr, "\n");
}

int
main()
{
    printf("> ");
    yyparse();
    return 0;
}
```

sends formatted output to a stream using an argument list passed to it.

```
bison -d fb3-1.y
flex fb3-1.l
cc fb3-1.tab.c lex.yy.c fb3-1funcs.c
```