In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.

The syntax is "abstract" in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details. For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes. Likewise, a syntactic construct like an if-condition-then expression may be denoted by means of a single node with three branches.
while b ≠ 0
    if a > b
        a := a - b
    else
        b := b - a
return a
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.
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.

A double is a double-precision, 64-bit floating-point data type.
Variadic functions

The number of arguments passed is determined at runtime but the compiler operates at compile time.

It is thus impossible for any compiler, no matter how smart to know the number of arguments that will be passed to a variadic function.

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.
#include <stdarg.h>

double average(int count, ...) {
    va_list ap;
    int j;
    double tot = 0;
    va_start(ap, count); //Requires the last fixed parameter (to get the address)
    for(j=0; j<count; j++)
        tot+=va_arg(ap, double); //Requires the type to cast to.
        // Increments ap to the next argument.
    va_end(ap);
    return tot/count;
}
A union is a special data type available in C that allows you 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.

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

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

you need to always access the right type of data from within the union. If you write an int and try to read it back as a float, you'll get a meaningless value.
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.

```c
#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); }
```
Once the union is defined, we need to tell bison what symbols have what types of values by putting the appropriate name from the union in angle brackets (<>).

The token NUMBER, which represents numbers in the input, has the value <d> to hold the value of the number.

The new declaration %type assigns the value <a> to exp, factor, and term, which we’ll use as we build up our AST.

You don’t have to declare a type for a token or declare a nonterminal at all if you don’t use the symbol’s value. If there is a %union in the declarations, bison will give you an error if you attempt to use the value of a symbol that doesn’t have an assigned type.
%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

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.

The %yylineno option tells flex to define an integer variable called yylineno and to maintain the current line number in it.
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(6);
    }
    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(6);
    }
    a->nodetype = 'K';
    a->number = d;
    return (struct ast *)a;
}
depth-first traversal of the tree, recursively visiting the subtrees of each node and then the node itself.
sends formatted output to a stream using an argument list passed to it.

```c
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;
}
```

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