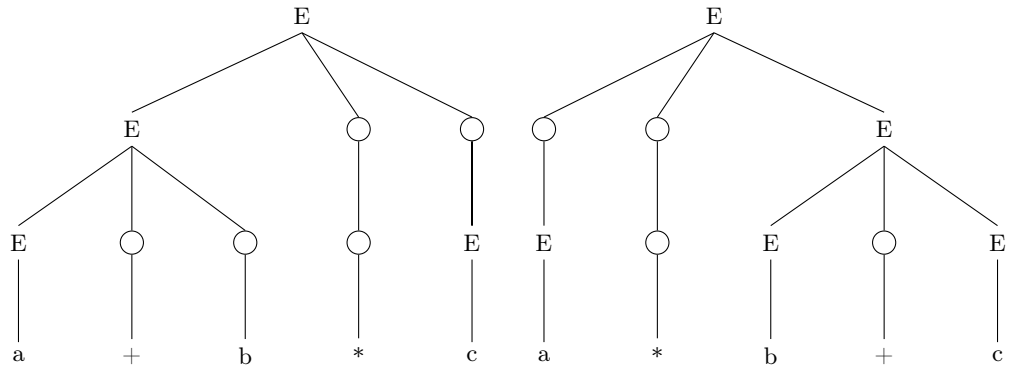
Consider a simple grammar:

E → E + E
E → E * E
E → (E)
E → id

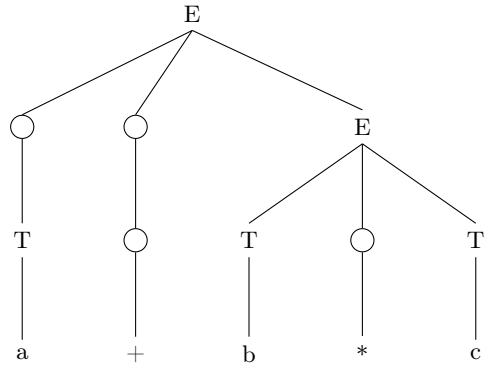
This has an ambiguity, as can be seen by the two parse trees below:



Parsers will build the left tree, whereas humans would build the right.
We must complicate the grammar to filter out the bad trees.

E → E * T
E → E + T
E → T
E → (E)
E → id

Now we have a single correct tree:



Our edits specify two new properties:

1. (*) has higher precedence than (+)
2. both (*) and (+) are left-associative

Let's now look at a subset of C's grammar:

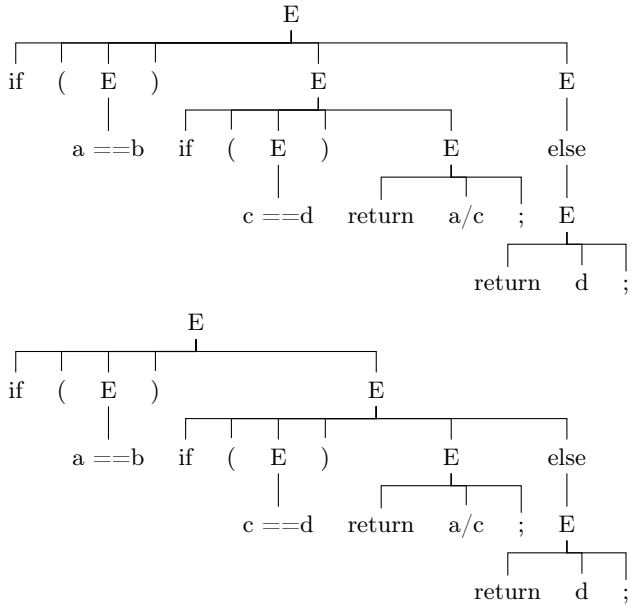
smt:
';'
'break' ';'
'return' ';'
'return' expr ';'
expr ';'
'goto' 'ID' ';'
'ID' ';' stmt
'while' '(' expr ')' stmt
'do' stmt 'while' '(' expr ')' ';'
'if' '(' expr ')' stmt
'if' '(' expr ')' stmt 'else' stmt

We may naturally wonder why some of these choices were made; for instance – why parentheses?
Consider the following altered right-hand-sides

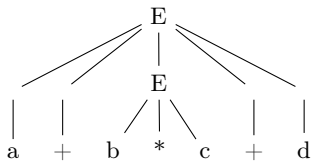
1. 'while' expr stmt:
   This is ambiguous – consider while i * p = = 3;

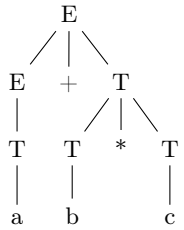
2. 'do' stmt 'while' expr ';'
   This is not ambiguous, since we know where our expr ends. Parentheses are used for consistency.

Not all ambiguities are as obvious as our original example, however; this C sub-grammar even has one!
It is one we have undoubtedly encountered.... The dangling else!
Consider the following two trees:



We know the latter is correct, but what rule is causing this?
As it turns out.. 'if '(' expr ')' stmt 'else' stmt *is too generous!*
We could fix this by adding a <full-stmt> nonterminal that requires an else, but C doesn't.
To see why, we must consider the following two trees for the statement a + b * c + d:





The left tree is an **abstract parse tree**, which uses an ambiguous grammar and smart compiler.
The right tree is a **concrete parse tree**, since its grammar is unambiguous.
Since abstract parse trees are smaller and simpler, the C compiler uses the abstract tree.
The C standard thus leaves fixing the ambiguity to the programmer, in "the usual way".

For an example of ambiguity, consider Prolog:
Prolog lets us define operators in the form :- op(precedence, associativity, [operators]),
where lower precedence means operating first, and y is the associative parameter. Thus:

```
:-op(500, yfx, [+, -]).
:-op(400, yfx, [*, /]).
% so for a*b*c -> (a+b)*c, we violate (*)'s precedence and associativity!
:-op(200, xfy, [**]).
:-op(200, fy, [+, -]).
/*      Therefore:
        a**b**c -> a**(b**c)
        a**(-b) -> -b**a -> -(b**a) */
:-op(700, xfx, [=, >=,...]).
% these are non associative ^^^ so a==b==c is an error!
if (a <= b <= c) ...; % valid code! but translates to
if ((a<=b) <= c)...; % which is equivalent to
if ([0,1] <= c)...; % which is not the expected behavior!
```

Ambiguity is not the only runtime error we may face though; consider the following C code:

```
int a, b;
int f(void) {return (a=1) + (b=2);}
// this is valid & harmless, albeit pointless, BUT
int g(void) {return (a=1) + (a=2);}
// this is undefined behavior in C!
// it could change a to 15, or return 42, or even dump core!
```

Compare how C and Java deal with this problem:
C – undefined behavior
Java – L → R semantics; this is valid, but prevents compiler optimization.
We can thus see side effects in expressions tend to be bad news, and programs with them are hard to maintain.
Undefined Behavior is dependent on semantics; it is the problem of competing side effects

These problems were evident to J. Backus after he made FORTAN ambiguous for performance's sake.
He regretted it, and thus proposed **functional programming**
He had two primary motivations in doing this:

1. clarity (use of well defined math notation)
   even the name of an imperative language (C++ -> C = C + 1) is nonsense!
2. performance (via parallelization)
   escape the von Neumann LOAD-EXECUTE-STORE model

One can now see Backus was a visionary:

1. IOT devices occasionally recompute a value rather than load from RAM to save power.
2. RAM access can have massive time cost when distance separated from CPU
3. modern computers run many CPU's in parallel

This makes his paradigm worth discussing, but some terms need to be defined first:
function ≡ a mapping from a domain to a range
domain ≡ a set of values
range ≡ a set of values
partial function ≡ a function which maps a subset of the domain
no side effects ≡ the same arguments to the same function will always the same result
functional form/higher-order function ≡ a function with a function as a parameter

So if functional languages can't modify state, how do they do I/O?
The short answer? They don't; it can't be done functionally.
The long answer? Consider getc(). It moves the stream pointer, so it has side effects.
We can thus emulate a pure functional I/O function by

```
define (f1, c) = getc(f)
```

Functional programming has a few very beneficial properties:

```
(* 1. evaluation order is not controlled by sequencing *)
# h(f(a), g(b))
        (* the order must be g,f -> h; since g \& f have no order, this is partial order *)
(* 2. referential transparency *)
#       let x = 1 in
        let f = fun f -> x + f in
        let x = 2 in
        f 2;;
- : int 3
(*      f uses the definition of x within its code block -- this x cannot change value.
        The variable/function f exist in different namespaces => they are unambiguous.
        Thus the CPU can cache and optimize even faster.
        This is so strong that there will be no errors, even on reuse: *)
# let square square = square * square;;
(* perfectly valid code -- function square \& variable square are in different namespaces *)
```