Hierarchical Grammars

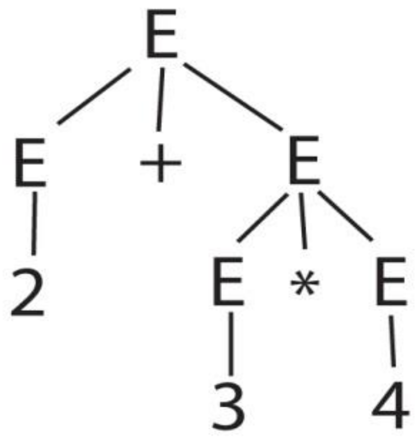
Ambiguity

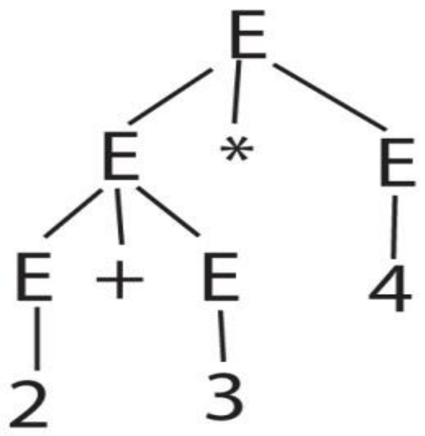
A grammar that can produce more than one parse tree for a given sentence is called *ambiguous*. Ambiguity is a very bad property for a grammar because it can lead to the sentence being interpreted in different and unexpected ways.

Example: Consider the grammar

E ::= E+E | E*E | num

Here are two parse trees for the sentence 2+3*4



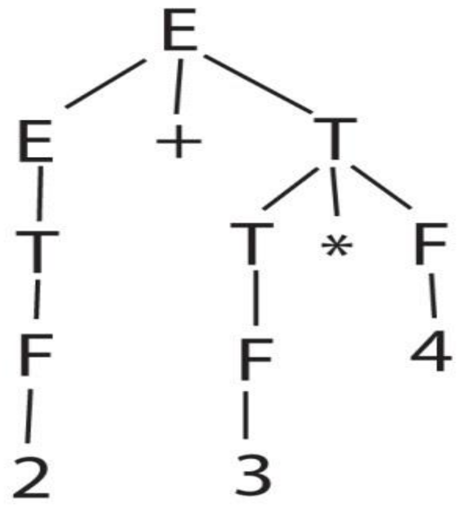


If we evaluate 2+3*4 from the parse trees, these two trees give different values: 14 on the left and 20 on the right. That is why ambiguity is bad.

Here is an unambiguous grammar that generates the same language as the previous ambiguous one:

```
        E ::= E+T | T
        T ::= T*F | F
        F ::= num
```

This time our sentence 2+3*4 has only one parse tree:



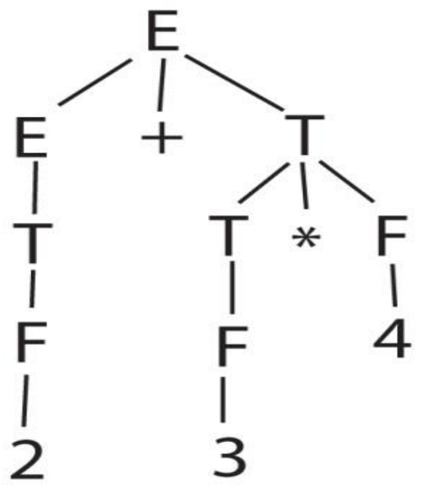
If there is a + in the sentence the rule E ::= E+T needs to be applied at the start, for if we begin E ::= T there is no + in the rules for T and below. The hierarchy of the rules eliminates the ambiguity.

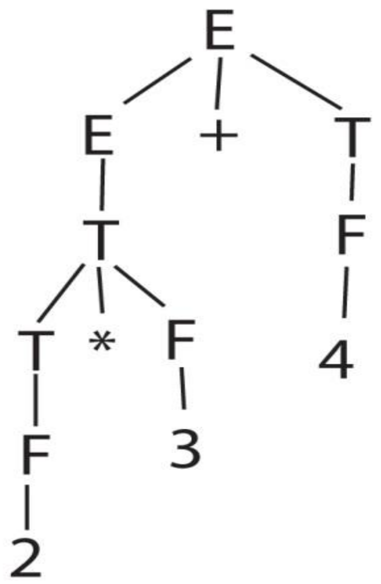
Precedence

Consider the grammar

```
E ::= E+T | T
T ::= T*F | F
F ::=  num
```

Here are parse trees for 2+3*4 and 2*3 + 4





Note that in both cases the grammar correctly gives multiplication precedence over addition.

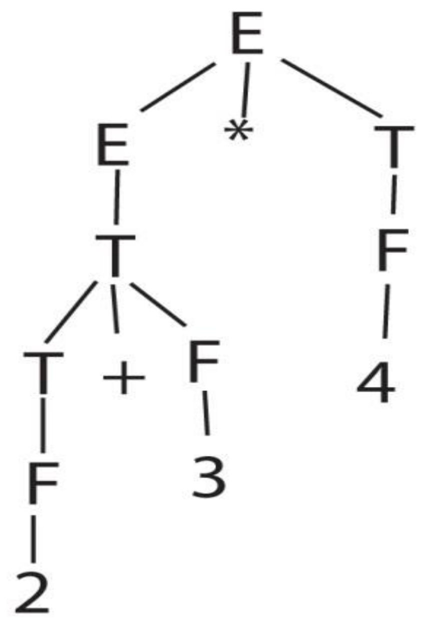
Now consider the grammar

E ::= E*T | T

T ::= T+F | F

F ::= num

The sentence 2+3*4 parses to



This gives addition precedence over multiplication!

Moral: In expression grammars we can determine operator precedence by using hierarchical grammars, with lower-precedence operators appearing higher in the grammar and higher-precedence operators farther down in the list of rules.

Associativity

In high school math classes you learned that the addition operator is *associative*: a+(b+c) = (a+b)+c.

This is not true of subtraction:

10-(7-3) = 6 (10-7)-3 = 0

We usually think of subtraction (and division as well) as being *left associative* in that the expression a-b-c is assumed to mean (a-b)-c and a-b-c-d-e is (((a-b)-c)-d)-e. This has not always been the case -- all operators in APL were right associative -- but left associativity is usually been preferred.

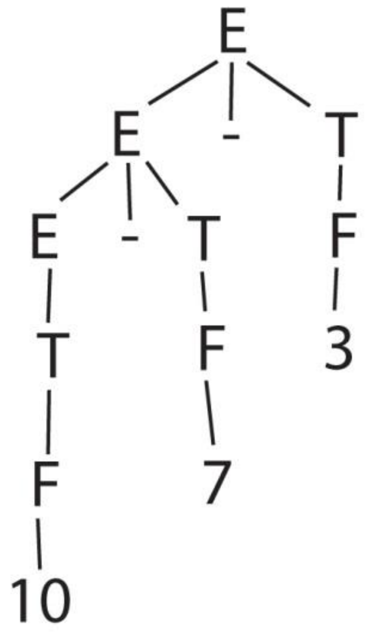
Let's extend our grammar to include subtraction and division:

E ::= E+T | E-T | T

T ::= T*F | T/F | F

F ::= num

The expression 10-7-3 parses to



Note that this associates from the left.

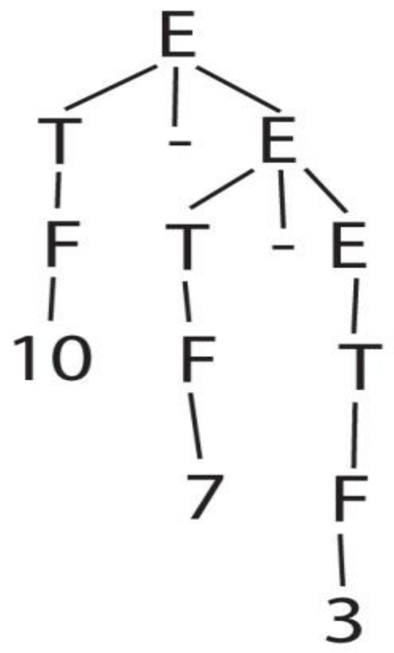
If we changed the grammar to

E ::= T+E | T-E | T

T ::= F*T | F/T | F

F ::= num

then 10-7-3 would parse to



This associates from the right.

A grammar rule

$$A ::= \alpha$$

is said to be *left recursive* if the symbol on the left side, A, is the leftmost symbol of the right side.

For example, our rule E ::= E+T is left recursive.

Similarly A ::= α is *right recursive* if A is the rightmost symbol on the right side. E ::= T+E is right recursive.

The last two examples show that left recursive rules make left associative operators, and right recursive rules make right associative operators.

Moral: For expression grammars we want hierarchical grammar rules to avoid ambiguity, operator precedence determined by the depth of the rule for the operator in the grammar, and associativity determined by the left recursive or right recursive nature of the rules.

Here is a full grammar for arithmetic expressions:

```
E ::= E+T | E-T | T
T ::= T*F | T/F | F
F ::= G**F | G
G ::= -H | H
H ::= (E) | id | num
```