# Grammars: Parse Trees and Ambiguity

CS236 - Discrete Structures
Instructor: Brett Decker
Fall 2021

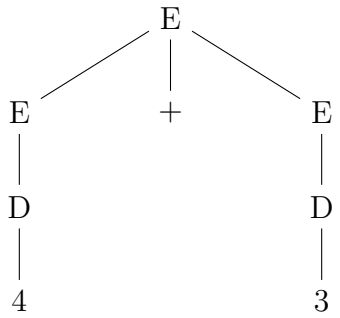
## Parse Trees

After introducing formal grammars, we learned how to derive valid strings by using the production rules. To take derivations further, we will create *parse trees*, a tree that identifies the production rules used to derive a valid string. Here is the idea of parse trees summed up in one sentence by Dr. Goodrich: "*Building a parse tree for a string is all about figuring out which production will turn nonterminals into terminals in a way that matches the string.*"

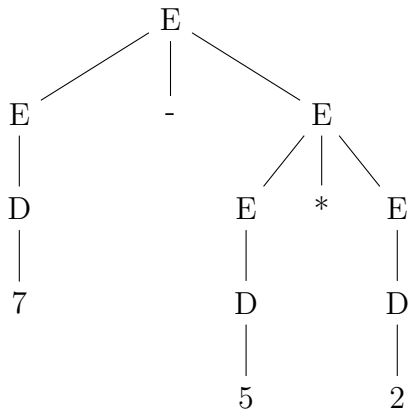
Parse trees will also give us a way to evaluate valid strings in a language. Parse trees are evaluated from bottom to top. Let's illustrate these concepts using the simple arithmetic language from our previous discussion:

$G = (V, T, S, P)$ where
$N = \{E, D\}$
$T = \{0, 1, 2, \ldots, 9, +, -, *, /, (, )\}$
$S = E$
$P = \{$
$E \to D \mid (E) \mid E + E \mid E - E \mid E * E \mid E/E$
$D \to 0 \mid 1 \mid 2 \mid \ldots \mid 9$
$\}$

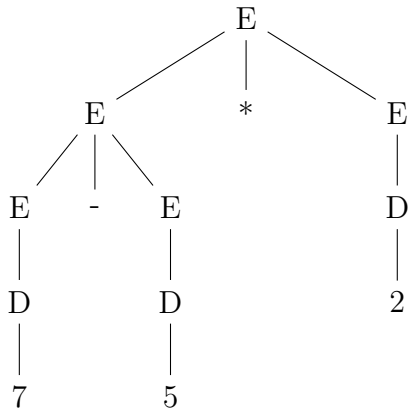
Recall that $V = N \cup T$ (all elements in both sets). Now, let's derive the terminal string $4+3$ using a parse tree:




When we evaluate from bottom to top, we mean that we start by identifying the leaf nodes ('4' and '3'). They are the operands. Then we find the operator to apply to them, the '+' node. We now evaluate $4 + 3$ and get '7' as the result. Thus, the string '$4 + 3$' evaluates to '7.' So far our simple arithmetic language seems fine. Let's do another example, '$7 - 5 * 2$':




Evaluated from bottom to top, we have $5 * 2 = 10$ and $7 - 10 = -3$. Our terminal string '$7 - 5 * 2$' evaluates to '$-3$.' Let us consider another valid parse tree for the terminal string '$7 - 5 * 2$':




Evaluated from bottom to top, we have $7 - 5 = 2$ and $2 * 2 = 4$. Our terminal string '$7 - 5 * 2$' now evaluates to '4.' This is problematic. We have two, valid parse trees for one terminal string that evaluate to two different results. When we have a grammar that allows for multiple parse trees for the same terminal string it is said to be *ambiguous*. This is important. This means that any grammar that has more than one possible derivation (or parse tree) for one terminal string is then an ambiguous grammar. Ambiguity is problematic. You've probably already experienced this before: why doesn't my C++ program work when compiled with g++? It worked in Visual Studio (complex languages like C++ sometimes have gaps in their specifications which leads to ambiguity, and thus compiler differences). Therefore, ambiguity is bad.

## Grammar Ambiguity: Definition

A grammar is ambiguous if there exists a terminal string that has more than one possible parse tree.

# Fixing Ambiguous Grammars

There are three causes of ambiguity: *precedence*, *left associativity*, and *right associativity*. Precedence defines the order of operations. Left associativity defines when operators should be evaluated from left to right. Right associativity defines when operators should be evaluated from right to left. Consider the following mathematical expression: '1 + 2 * 3.' What is the precedence of the different operations? According to mathematics, the multiplication has higher precedence and thus is executed first. We need our grammar for our simple arithmetic language to incorporate precedence to enforce this. Consider this expression: '6 / 2 * 3.' Is there a precedence issue with these operators? No, multiplication and division have the same precedence, but depending on which operation we perform first the result will be different. This is a case of left associativity. In mathematics, left associativity is the rule for these operators. We need our grammar to enforce this. Finally, let's consider '2^3^2.' What associativity do we use for exponentiation? Mathematics defines right associativity. We need our grammar to do likewise.

## Fixing Precedence

How do we fix *precedence* ambiguity? Consider the following general pattern: *the number of nonterminals from the start to a nonterminal* represents have far "down" in a parse tree a nonterminal can appear. Remember that we evaluate our parse trees from bottom to top. Thus, nonterminals further "down" in the tree have "higher" precedence. nonterminals further "up" in the tree have "lower" precedence. Thus by introducing new nonterminals into our grammar that get our symbols executed correct level of the tree allows us to fix the precedence.

## Fixing Precedence Example:

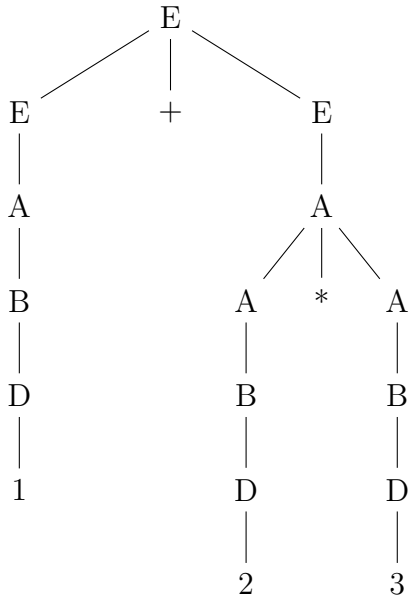
Let's do an example. We'll modify the production rules to enforce precedence according to mathematics with our simple arithmetic language:

$$
\begin{aligned}
P = \{ & \\
& E \to A \mid E + E \mid E - E \\
& A \to B \mid A * A \mid A/A \\
& B \to D \mid (E) \\
& D \to 0 \mid 1 \mid 2 \mid \ldots \mid 9 \\
\} &
\end{aligned}
$$

There is only one step from $E$ to the $+$ and the $-$ operator. They therefore are "high" in the tree, which means they get executed last. We then have to go from $E$ to $A$ to the $*$ and the / operators, which puts them lower in the tree than plus and minus. Thus, multiply and divide get executed before plus and minus. We have to go from $E$ to $A$ to $B$ before we get to matching parentheses. Thus, matching parentheses are lowest in the tree, so they get executed first.

Now let's create the parse tree for '1 + 2 * 3.' Notice that we must derive '$E + E$' first because there is no longer a production rule from the start symbol, $E$, to '$E * E$.' If we first

produce an $A$, and then the multiplication, we'll be stuck, because we have no production rule to include the remaining addition. Thus we have now enforced precedence. Here is the non-ambiguous parse tree for '1 + 2 * 3':




Recall that we evaluate the parse trees from bottom to top, so the multiplication will be evaluated first. This is what is meant by pushing higher precedence operators lower in the tree – the lower in the tree, the sooner to be evaluated. This modified grammar fixes precedence, but it doesn't fix left or right associativity (try out a derivation if you don't see why).

## Fixing Associativity

To fix *associativity* ambiguity we also need to introduce new nonterminals. We also have a general pattern for fixing associativity: right recursion (in a production) leads to right associativity (in a binary operation), and left recursion leads to left associativity. Why does this work? Left recursion enforces left associativity by pushing the first character of the right-hand side of the production lower in the parse tree than the last character of the right-hand side. Thus for some nonterminal $Z$ we have the production '$Z \to Zxxx$' for left associativity and '$Z \to xxxZ$' for right associativity.

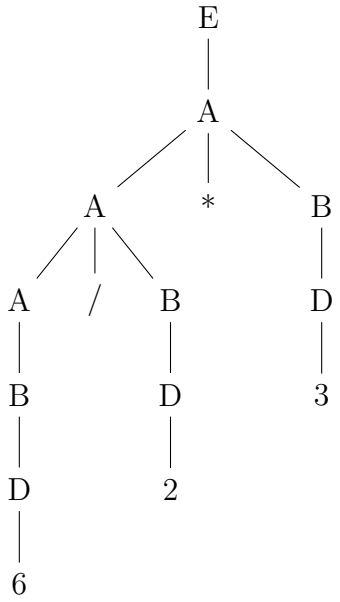
## Fixing Associativity Example:

Let's do an example. We'll start with the simple arithmetic grammar with precedence fixed. Note that since we already added new nonterminals to fix precedence we don't need to add more, but if we started with the original grammar we *would* need to add new nonterminals. We now need to change the production rules that we have to enforce associativity where we want it. Recall that we had the rule $A \to A * A$ above. We're using the same nonterminal, $A$, on both sides of the operator. To enforce associativity, we need to change this (using the pattern discussed above). We want left associativity, thus we keep the nonterminal on the

left as $A$, but change the $A$ on the right to a new nonterminal that will be lower in the parse tree. That means we change our rule from $A \to A * A$ to $A \to A * B$. Here is what our grammar production rules are now:

$$
\begin{aligned}
P = \{ & \\
& E \to A \mid E + A \mid E - A \\
& A \to B \mid A * B \mid A/B \\
& B \to D \mid (E) \\
& D \to 0 \mid 1 \mid 2 \mid \ldots \mid 9 \\
\} &
\end{aligned}
$$

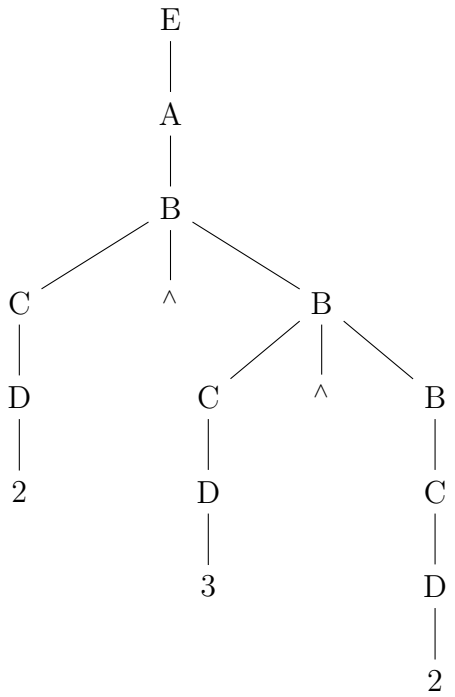
Now let's create the parse tree for '6 / 2 * 3':




Our parse tree now evaluates the operators left to right. Finally, let's add exponentiation to our grammar to show how right associativity works:

$$
\begin{aligned}
P = \{ & \\
& E \to A \mid E + A \mid E - A \\
& A \to B \mid A * B \mid A/B \\
& B \to C \mid C \hat{} B \\
& C \to D \mid (E) \\
& D \to 0 \mid 1 \mid 2 \mid \ldots \mid 9 \\
\} &
\end{aligned}
$$

Note that we had to add a nonterminal only because exponentiation has higher precedence than multiplication and division. Let's create the parse tree for '2^3^2':



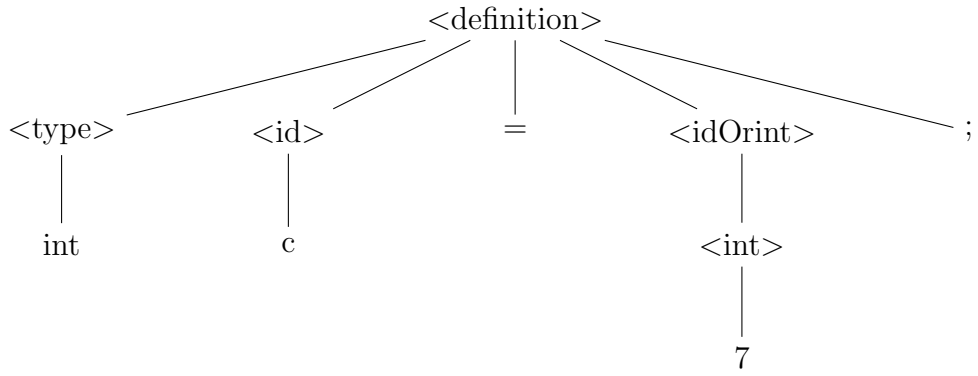

We have now fixed all ambiguity in our grammar for our simple arithmetic language. Here is the complete grammar:
$G = (V, T, S, P)$ where
$N = \{E, A, B, C, D\}$
$T = \{0, 1, 2, \ldots, 9, +, -, *, /, (, ), \hat{}\}$
$S = E$
$P = \{$
$E \to A \mid E + A \mid E - A$
$A \to B \mid A * B \mid A/B$
$B \to C \mid C\hat{}B$
$C \to D \mid (E)$
$D \to 0 \mid 1 \mid 2 \mid \ldots \mid 9$
$\}$

## Parse Trees with BNF

Consider again the BNF grammar seen previously:

```
<definition> ::= <type> <id> = <idOrint>;
<type> ::= int | byte
<idOrint> ::= <id> | <int>
<id> ::= a | b | c | ... | x | y | z
<int> ::= 0 | 1 | 2 | ... | 9
```

Let's see a parse-tree for the terminal string `int c = 7;`




# Conclusion

Understanding parse trees, and fixing ambiguity in grammars are important formal language topics and are used when creating, evaluating, and analyzing programming languages. Thanks to Dr. Michael Goodrich for the simple arithmetic grammar and examples.