SLecture 4

CSE 425: Concepts of Programming Languages

Group: WillCodeForFood

Nazmul Hossain, Sabbir Mollah, Pritom Kumar Saha

July 30, 2019

Contents

1 Parse Tree **2**

2 Abstract Syntax Tree (AST) **7**

3 Ambiguity **9**

4 Precedence **10**

5 Associativity **12**

6 Reference **14**

1 Parse Tree

Parse tree, also known as derivation tree or concrete syntax tree or parsing tree, represents the derivation grammar to yield strings. It represents some syntactic structure of a string according to some context-free grammar.

Parse tree may be generated for a natural language or even for a programming language. It is heavily used in the processing of programming languages as well as natural language processing (NLP).

In the early years of computer programming, parsing was one of the fundamental problems compiler writers had to face. Nowadays it has become automated thanks to some tools like the Lex which generates lexical analyzers and Yacc (Yet Another Compiler-Compiler) which is a Look Ahead Left-to-Right (LALR) parser generator. Parse trees can be constructed in both bottom-up and top-down method.

There are two categories of parse trees based on terminal nodes. These are:

Constituency-based parse trees:

The constituency-based parse trees distinguish between terminal and non-terminal nodes. The leaf nodes are labeled by terminal categories of the grammar and the interior nodes are labeled by the non-terminal categories.

Parse three depends on grammar. Even English grammars have parse trees. For instance: If we consider an English sentence: We learned the programming language from the professor.

⟨sentence⟩ → ⟨noun phrase⟩ ⟨verb phrase⟩

→ ⟨noun⟩ ⟨verb phrase⟩.
→ We ⟨verb phrase⟩.
→ We ⟨verb⟩ ⟨noun phrase⟩ ⟨prepositional phrase⟩.
→ We learned ⟨noun phrase⟩ ⟨prepositional phrase⟩.
→ We learned ⟨determiner⟩ ⟨noun⟩ ⟨prepositional phrase⟩.
→ We learned the ⟨noun⟩ ⟨prepositional phrase⟩.
→ We learned the programming language ⟨prepositional phrase⟩.
→ We learned the programming language ⟨preposition⟩ ⟨noun phrase⟩.
→ We learned the programming language from ⟨noun phrase⟩.
→ We learned the programming language from ⟨determiner⟩ ⟨noun⟩.
→ We learned the programming language from the ⟨noun⟩.
→ We learned the programming language from the professor.

The parse tree would look the tree below for the above grammar:

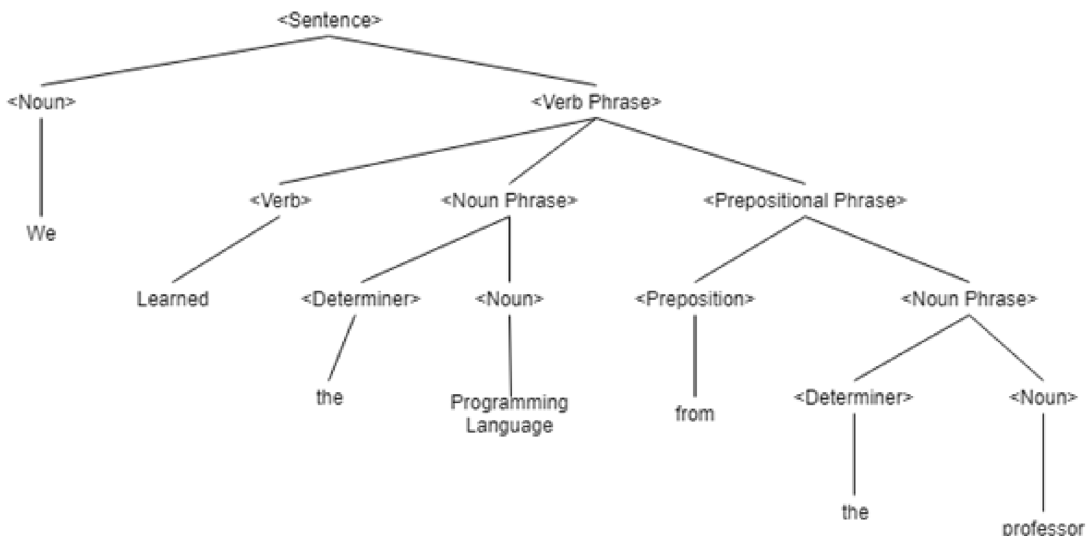


Figure 1: English Grammar Parse Tree (Constituency-based)

Dependency-based parse trees:

All nodes are considered as terminal int the dependency-based parse trees. They contain fewer nodes than constituency-based parse trees on average. An example of a dependency-based parse tree is as follows: We consider the English sentence: John hit the ball.

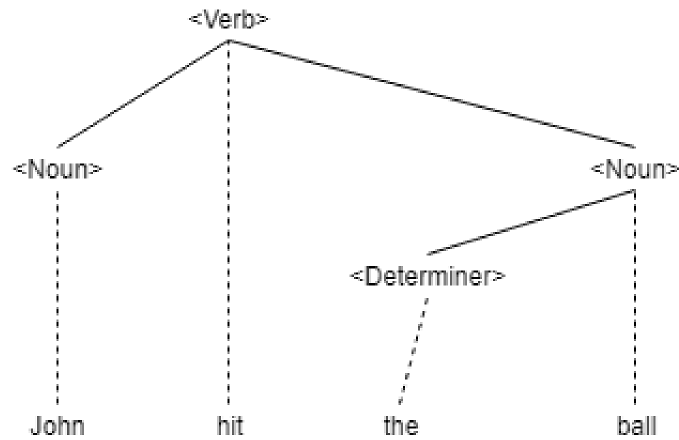


Figure 2: English Grammar Parse Tree (Dependency-based)

The constituency-based parse tree is the more popular grammar structure. This is used for programming languages too.

In a programming language, if we define grammar as follows,

$$\begin{aligned}\langle\text{expression}\rangle \rightarrow\ & \langle\text{expression}\rangle + \langle\text{expression}\rangle \mid \\ & \langle\text{expression}\rangle * \langle\text{expression}\rangle \mid \\ & \langle\text{number}\rangle \mid \langle\text{digit}\rangle\end{aligned}$$

$$\langle\text{number}\rangle \rightarrow \langle\text{number}\rangle\ \langle\text{digit}\rangle \mid \langle\text{digit}\rangle$$

$$\langle\text{digit}\rangle \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Then the derivation of the number 234 is as follows:

$$\begin{aligned}\langle\text{number}\rangle & \rightarrow \langle\text{number}\rangle\ \langle\text{digit}\rangle \\ & \rightarrow \langle\text{number}\rangle\ \langle\text{digit}\rangle\ \langle\text{digit}\rangle \\ & \rightarrow \langle\text{digit}\rangle\ \langle\text{digit}\rangle\ \langle\text{digit}\rangle \\ & \rightarrow 2\ \langle\text{digit}\rangle\ \langle\text{digit}\rangle \\ & \rightarrow 2\ 3\ \langle\text{digit}\rangle \\ & \rightarrow 2\ 3\ 4\end{aligned}$$

The number 234 will be constructed in a parse tree as follows:

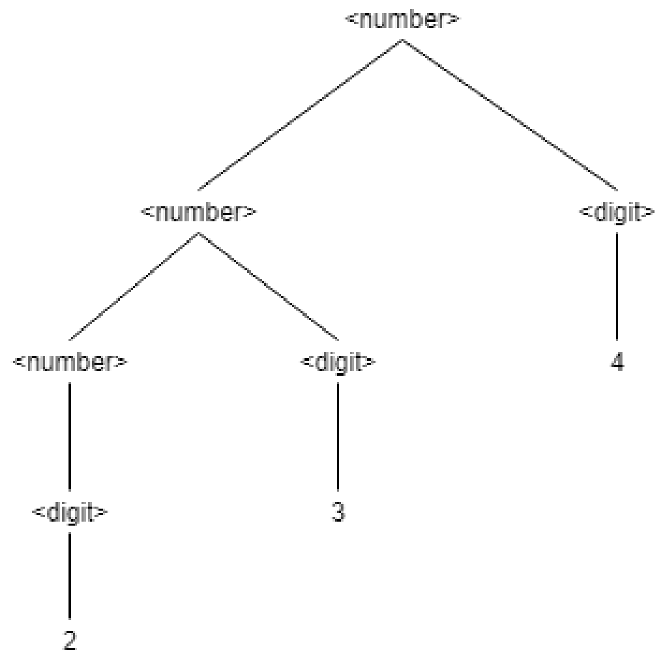


Figure 3: Parse tree for a number

Using the same grammar, we can derive the arithmetic expression 3 + 4 * 5 as follows:

(Two different leftmost derivation exists for the desired construct. This will be discussed in details in the ambiguity section. For now, we are showing one of the methods here.)

$$
\begin{aligned}
\langle \text{expression} \rangle &\rightarrow \langle \text{expression} \rangle * \langle \text{expression} \rangle \\
&\rightarrow \langle \text{expression} \rangle + \langle \text{expression} \rangle * \langle \text{expression} \rangle \\
&\rightarrow \langle \text{number} \rangle + \langle \text{expression} \rangle * \langle \text{expression} \rangle \\
&\rightarrow \langle \text{digit} \rangle + \langle \text{expression} \rangle * \langle \text{expression} \rangle \\
&\rightarrow 3 + \langle \text{expression} \rangle * \langle \text{expression} \rangle \\
&\rightarrow 3 + \langle \text{number} \rangle * \langle \text{expression} \rangle \\
&\rightarrow 3 + \langle \text{digit} \rangle * \langle \text{expression} \rangle \\
&\rightarrow 3 + 4 * \langle \text{expression} \rangle \\
&\rightarrow 3 + 4 * \langle \text{number} \rangle \\
&\rightarrow 3 + 4 * \langle \text{digit} \rangle \\
&\rightarrow 3 + 4 * 5
\end{aligned}
$$

If we don't use leftmost derivation and derive the arithmetic operation in the following way, it will also produce the exact same parse tree:

$$
\begin{aligned}
\langle\text{expression}\rangle &\rightarrow \langle\text{expression}\rangle * \langle\text{expression}\rangle \\
&\rightarrow \langle\text{expression}\rangle + \langle\text{expression}\rangle * \langle\text{expression}\rangle \\
&\rightarrow \langle\text{number}\rangle + \langle\text{expression}\rangle * \langle\text{expression}\rangle \\
&\rightarrow \langle\text{number}\rangle + \langle\text{number}\rangle * \langle\text{expression}\rangle \\
&\rightarrow \langle\text{number}\rangle + \langle\text{number}\rangle * \langle\text{number}\rangle \\
&\rightarrow \langle\text{digit}\rangle + \langle\text{number}\rangle * \langle\text{number}\rangle \\
&\rightarrow \langle\text{digit}\rangle + \langle\text{digit}\rangle * \langle\text{number}\rangle \\
&\rightarrow \langle\text{digit}\rangle + \langle\text{digit}\rangle * \langle\text{digit}\rangle \\
&\rightarrow 3 + \langle\text{digit}\rangle * \langle\text{digit}\rangle \\
&\rightarrow 3 + 4 * \langle\text{digit}\rangle \\
&\rightarrow 3 + 4 * 5
\end{aligned}
$$

The parse tree will look like this:

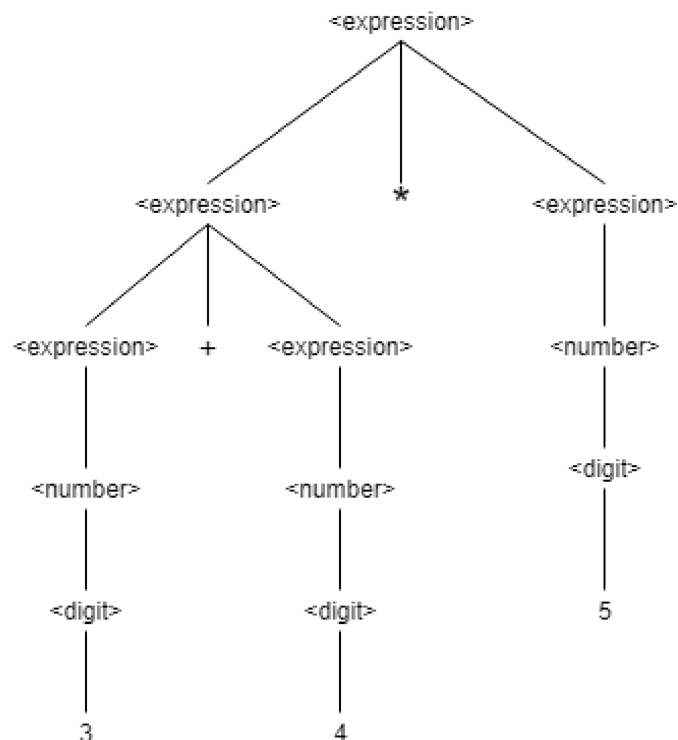


Figure 4: Parse tree for an arithmetic expression

2 Abstract Syntax Tree (AST)

Abstract syntax tree (AST) also known as syntax tree or context syntax tree is an abstract syntactic structure of a code written in a programming language. A condensed parse tree is also considered as an abstract syntax tree.

Differences between parse tree and abstract syntax tree are as follows:

Parse Tree	**Abstract Syntax Tree**
An ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.	Abstract syntax tree is an abstract syntactic structure of a code written in a programming language.
Interior nodes represent grammar rules and leaf nodes represents terminals.	Interior nodes represent operations and leaf nodes represent operands.
Characteristic information from the real syntax are provided.	Does not provide characteristic information.
Syntax trees are comparatively denser than parse trees.	Parse trees are comparatively less dense than syntax trees.

To build an abstract syntax tree, let us consider the previous arithmetic operation (3 + 4 * 5) again. If we collapse the production chains of the fig 4, the AST will be built. The final form of the AST is as the following figure:

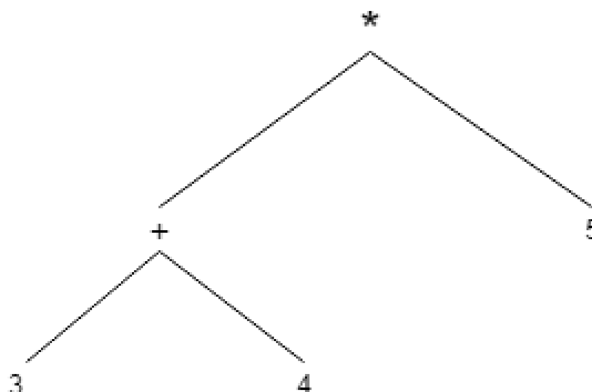


Figure 5: AST for an arithmetic expression

Let us now consider the following pseudo-code snippet:

if $($ number $\rangle 5)$
    returntrue;
else
    returnfalse;

The Abstract Syntax tree of this code snippet would look like the following figure:

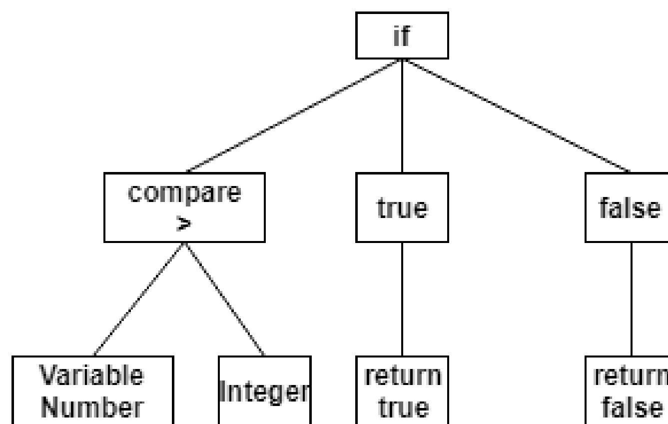


Figure 6: AST for a programming language syntax

3 Ambiguity

In programming languages ambiguity arises when a string in a context-free grammar definition has more than one leftmost derivation or parse tree. So, an unambiguous grammar would be when a string has only one unique derivation or parse tree.

The reference grammar often can be ambiguous. If an ambiguous grammar exists it is generally resolved by adding new rules like: precedence rules and context-sensitive parsing rules.

Let's look at an example of an ambiguous grammar:

Our rule is defined as:

$X \rightarrow X + X \mid X - X \mid y$

We are going to derive **y + y + y** using this rule.

$$\begin{aligned} X &\rightarrow X + X \\ &\rightarrow y + X \\ &\rightarrow y + X + X \\ &\rightarrow y + y + X \\ &\rightarrow y + y + y \end{aligned} \qquad \begin{aligned} X &\rightarrow X + X \\ &\rightarrow X + X + X \\ &\rightarrow y + X + X \\ &\rightarrow y + y + X \\ &\rightarrow y + y + y \end{aligned}$$

As we can see we have generated two unique derivations for the same string, we can say that this grammar rule is ambiguous.

We can use left most derivation as a flag to check if a grammar rule is ambiguous or not. If we find any grammar rule for which one token has more than one possible replacement in a leftmost derivation, then we can conclude that the grammar rule is ambiguous.

An ambiguous grammar rule can be fixed to be unambiguous by introducing new rules: Precedence and Associativity.

4 Precedence

In a programming language operator precedence is a set of rules that defines the order of evaluation of various operators. For example, in most of the languages (and in mathematical notation) the multiplication operator takes precedence over the addition operator.

So, 2+3*4 evaluates to 14 and not 20.

Why does it evaluate to 14?

Since the multiplication has a higher precedence, (3*4) gets evaluated first, so the result is 2+12=14. Setting up precedence can be a good way to remove ambiguity in a Parse tree. These rules help to keep the statements concise while removing ambiguity. Without precedence rules even a simple statement like 4*5 + 4*5*5 + 6 would have to be written as ((4*5) + ((4*5)*5)) + 6. One would agree that this is unnecessarily redundant.

In the context of a Parse Tree or AST, it is important that we maintain the precedence, so that the compiler can figure out what to evaluate first. For example, if X happens before Y, then we need to put X lower in the tree than Y. Let's take a look at the two possible AST of the expression 3 * 5 + 4.

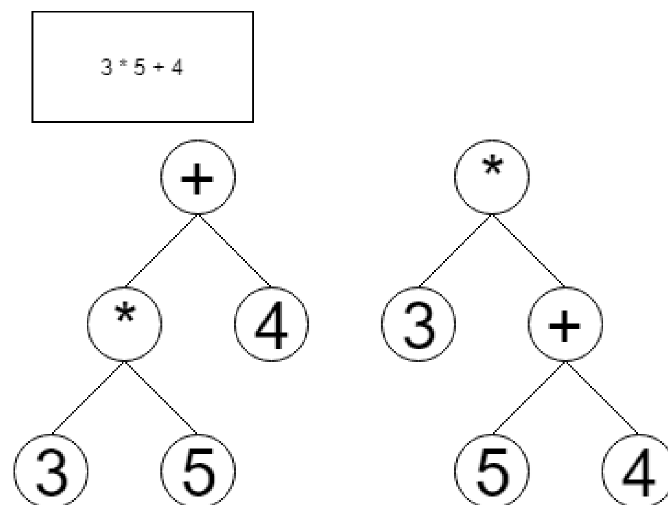


Figure 7: Two possible AST for a mathematical expression

Intuitively the left tree is giving precedence to the '+' symbol so that the expression will get evaluated as (3 * 5) + 4, while the right tree will evaluate it as 3 * (5 + 4). Not to argue that most people would prefer to give precedence to the multiplication.

The parse tree of this expression with multiplicative precedence will look like this:

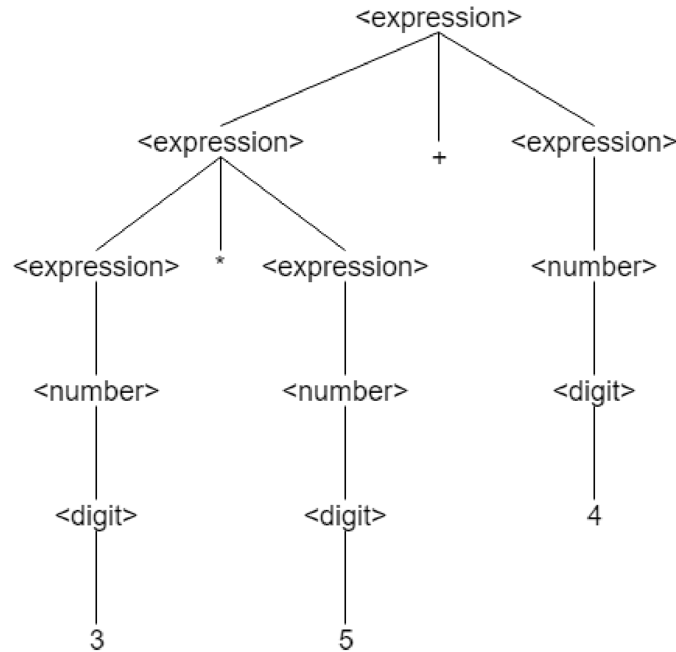


Figure 8: Two possible AST for a mathematical expression

Now one may be left wondering on how to define grammar with precedence in act. Here is an example of an unambiguous grammar that gives precedence to multiplication operator.

expression $\rightarrow$ expression + term | term
term $\rightarrow$ term * factor | factor
factor $\rightarrow$ (expression) | number
number $\rightarrow$ number digit | digit
digit $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

From this set of grammar, the multiplication will get defactored later than the addition.

5 Associativity

Associativity is a set of rules that helps to remove ambiguity in the case of same precedence operators grouped together when there are no parentheses.

For example, in the expression 1 + 2 + 3 there are two '+' signs. Since both of these two operators have the same precedence it can lead us to two possible AST.

Or in other words the expression can be evaluated either as 1 + (2+3) or (1+2) + 3.

To remove this ambiguity we have to define which of the two '+' should go lower in the tree. To do this the concept of operator associativity comes to light.

According to associativity operators can be either left associative or right associative. If an operator is left associative then the left most operator get evaluated first.

That is 1 + 2 + 3 + 4 will be evaluated as (((1 + 2) + 3) + 4)

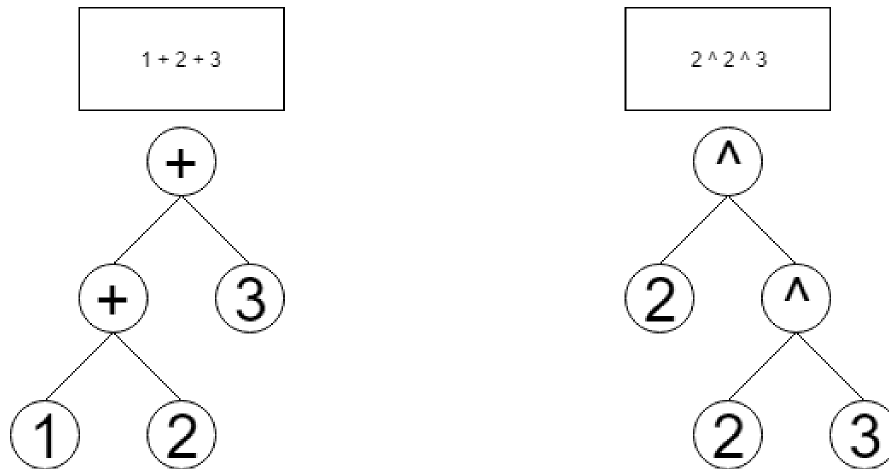
Associativity is bound in the grammar of the programming language. For example to signify left associativity for addition we would define a left-recursive grammar:

expression → expression + term | term

Similarly, for right associativity we would use a right-recursive grammar:

expression → term ˆ expression | term

The mathematical expression 1 + 2 + 3 should follow left associativity, such that it may evaluate as 1 + (2+3). On the other hand the expression 2ˆ2ˆ3 (ˆ symbolizes power) needs to follow right associativity.



6 Reference

1. https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Parsing
2. https://en.wikipedia.org/wiki/Yacc
3. https://en.wikipedia.org/wiki/Lex_(software)
4. https://en.wikipedia.org/wiki/Parse_tree
5. https://www.gatevidyalay.com/syntax-trees
6. https://en.wikipedia.org/wiki/Ambiguous_grammar
7. https://en.wikipedia.org/wiki/Operator_associativity
9. https://en.wikipedia.org/wiki/Order_of_operations