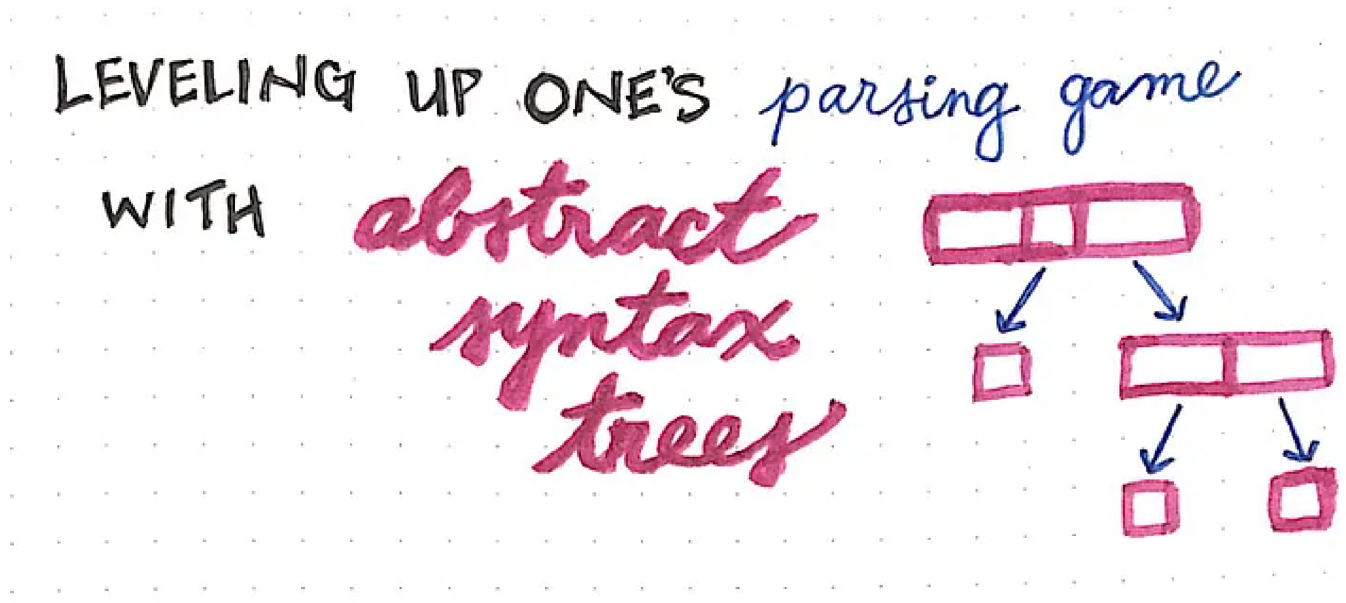
# Leveling Up One's Parsing Game With ASTs

Vaidehi Joshi · Follow
Published in basecs · 13 min read · Dec 4, 2017

2.9K 11



Leveling up one's parsing game with ASTs!

Before I started on this journey of trying to learn computer science, there were certain terms and phrases that made me want to run the other direction.

But instead of running, I feigned knowledge of it, nodding along in conversations, pretending like I knew what someone was referencing even though the truth was that I had no idea and had actually stopped listening entirely when I heard That Super Scary Computer Science Term™. Throughout the course of this series, I've managed to cover a lot of ground and many of those terms have actually become a whole lot less scary!

There is one big one, though, that I've been avoiding for awhile. Up until now, whenever I had heard this term, I felt paralyzed. It has come up in casual conversation at meet ups and sometimes in conference talks. Every single time, I think of machines spinning and computers spitting out strings of code that are indecipherable except that everyone else around me *can actually decipher them* so it's actually just me who doesn't know what's going on (whoops how did this happen?!).

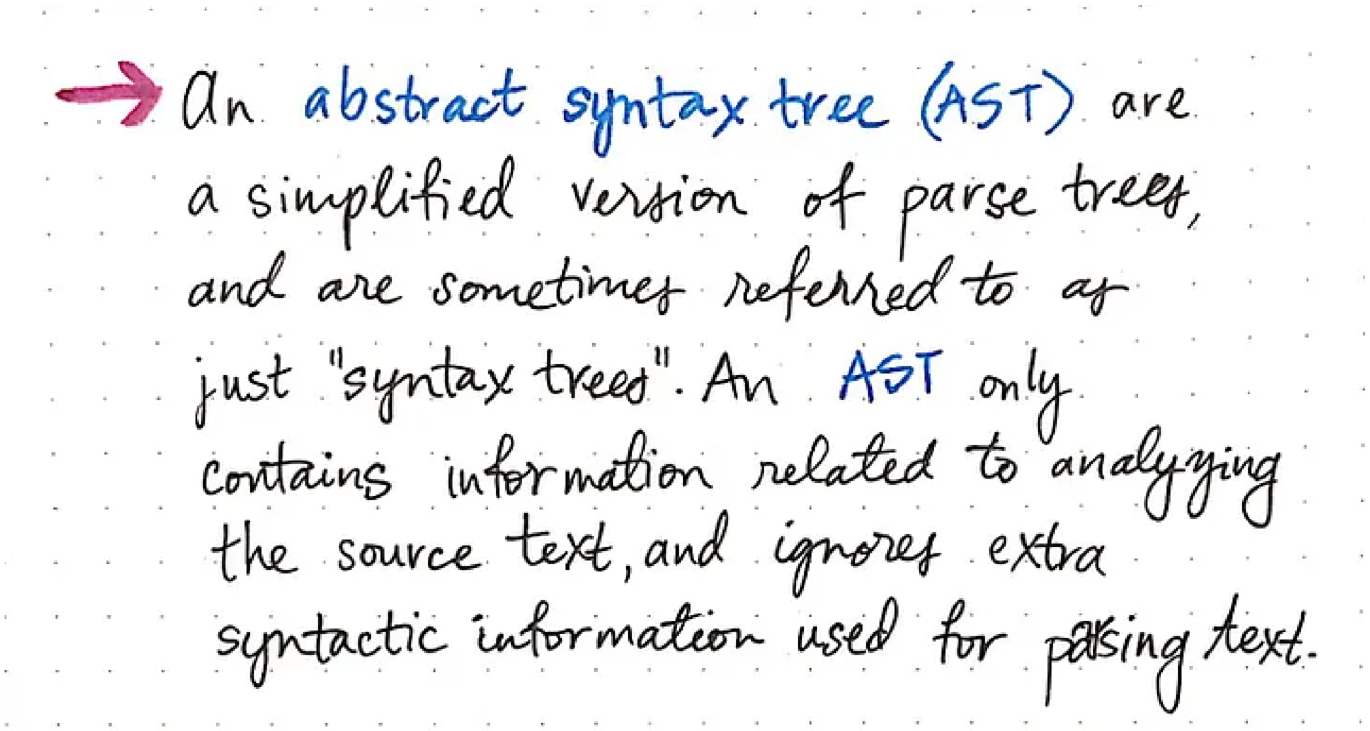
Perhaps I'm not the only one who has felt that way. But, I suppose I should tell you what this term actually is, right? Well, get ready, because I'm referring to ever-elusive and seemingly confusing *abstract syntax tree*, or *AST* for short. After many years of being intimidated, I'm excited to finally stop being afraid of this term and truly understand what on earth it is.

It's time to face the root of the abstract syntax tree head on — and level up our parsing game!

## From concrete to abstract

Every good quest starts with a solid foundation, and our mission to demystify this structure should begin in the exact same way: with a definition, of course!

An ***abstract syntax tree*** (usually just referred to as an *AST*) is really nothing more than a simplified, condensed version of a parse tree. In the context of compiler design, the term *"AST"* is used interchangeably with ***syntax tree***.



An abstract syntax tree (AST) are a simplified version of parse trees, and are sometimes referred to as just "syntax trees". An AST only contains information related to analyzing the source text, and ignores extra syntactic information used for parsing text.

Abstract syntax tree: a definition

We often think about syntax trees (and how they are constructed) in comparison to their parse tree counterparts, which we're already pretty familiar with. We know that *parse trees* are tree data structures that contain the grammatical structure of our code; in other words, they contain all the syntactic information that appears in a code "sentence", and is derived directly from the grammar of the programming language itself.

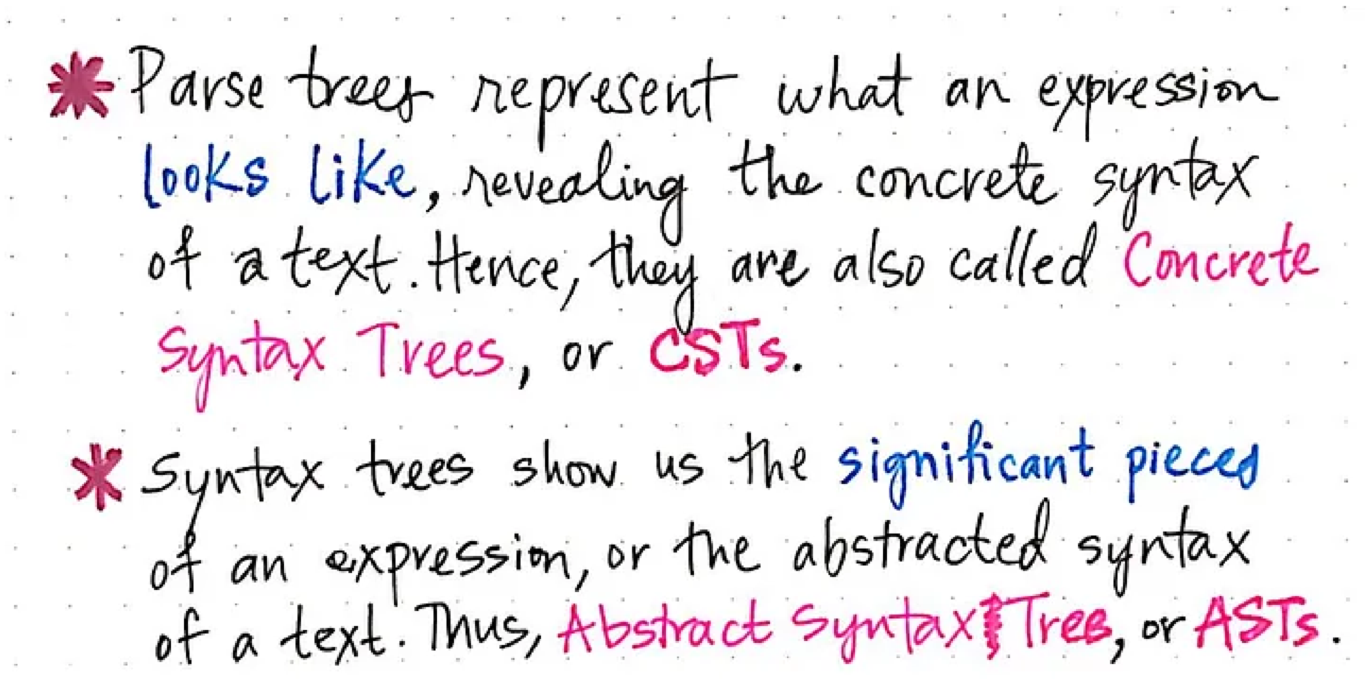
A *abstract syntax tree,* on the other hand, ignores a significant amount of the syntatic information that a parse tree would otherwise contain.

> By contrast, an AST only contains the information related to analyzing the source text, and skips any other extra content that is used while parsing the text.

This distinction starts to make a whole lot more sense if we focus in on the "abstractness" of an AST.

We'll recall that a ***parse tree*** is an illustrated, pictorial version of the grammatical structure of a sentence. In other words, we can say that a parse tree represents exactly what an expression, sentence, or text looks like. It's basically a direct translation of the text itself; we take the sentence and turn every little piece of it — from the punctuation to the expressions to the tokens — into a tree data structure. It reveals the concrete syntax of a text, which is why it is also referred to as a ***concrete syntax tree***, or ***CST***. We use the term concrete to describe this structure because it's a grammatical copy of our code, token by token, in tree format.

But what makes something concrete versus abstract? Well, an abstract syntax tree doesn't show us *exactly* what an expression looks like, the way that a parse tree does.

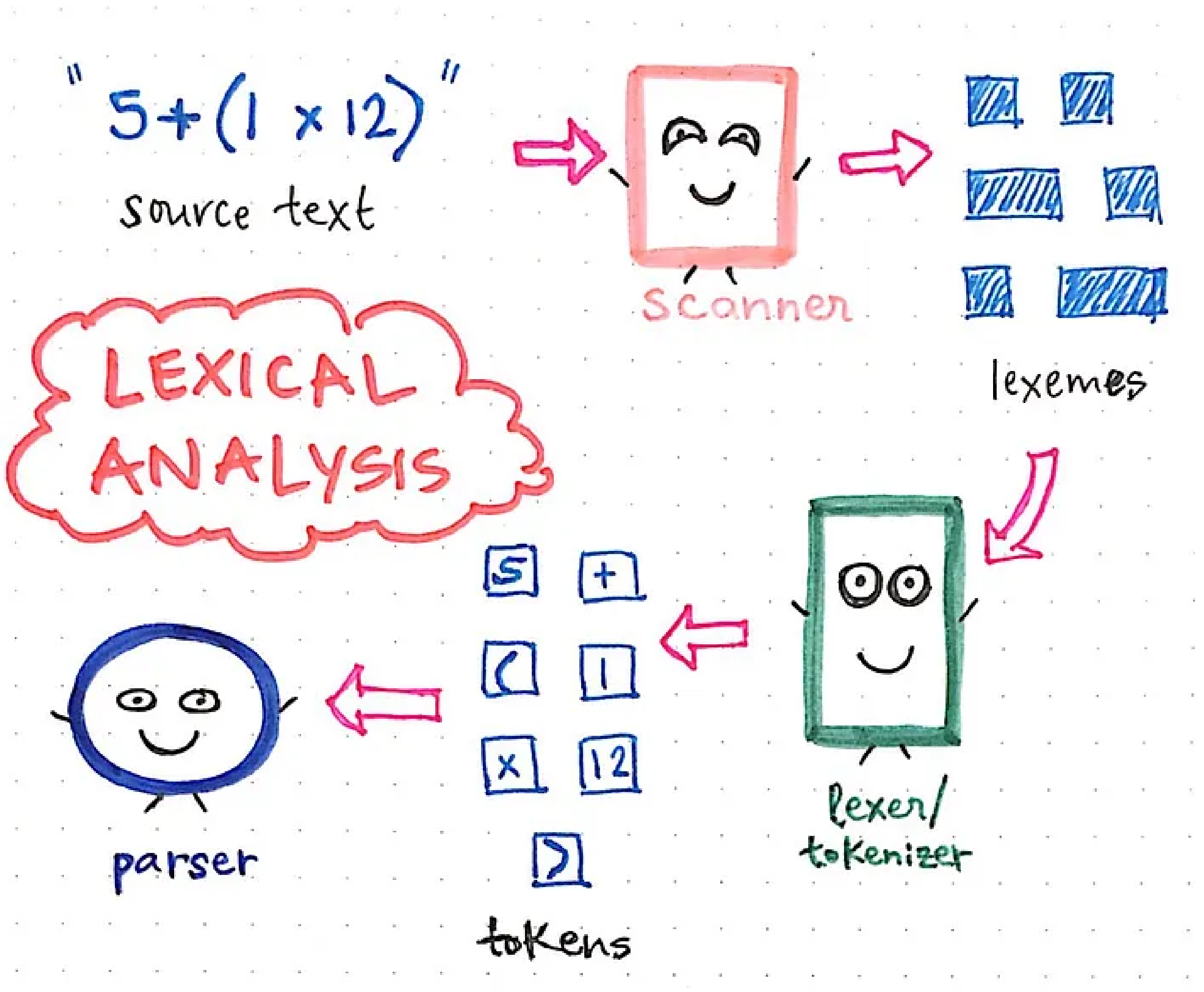


- Parse trees represent what an expression looks like, revealing the concrete syntax of a text. Hence, they are also called Concrete Syntax Trees, or CSTs.
- Syntax trees show us the significant pieces of an expression, or the abstracted syntax of a text. Thus, Abstract Syntax Trees, or ASTs.

Concrete versus abstract syntax trees

Rather, an abstract syntax tree shows us the “important” bits — the things that we really care about, which give meaning to our code “sentence” itself. Syntax trees show us the significant pieces of an expression, or the *abstracted* syntax of our source text. Hence, in comparison to concrete syntax tress, these structures are are abstract representations of our code (and in some ways, less exact), which is exactly how they got their name.

Now that we understand the distinction between these two data structures and the different ways that they can represent our code, it’s worth asking the question: where does an abstract syntax tree fit into the compiler? First, let’s remind ourselves of everything that we know about the compilation process as we know it so far.



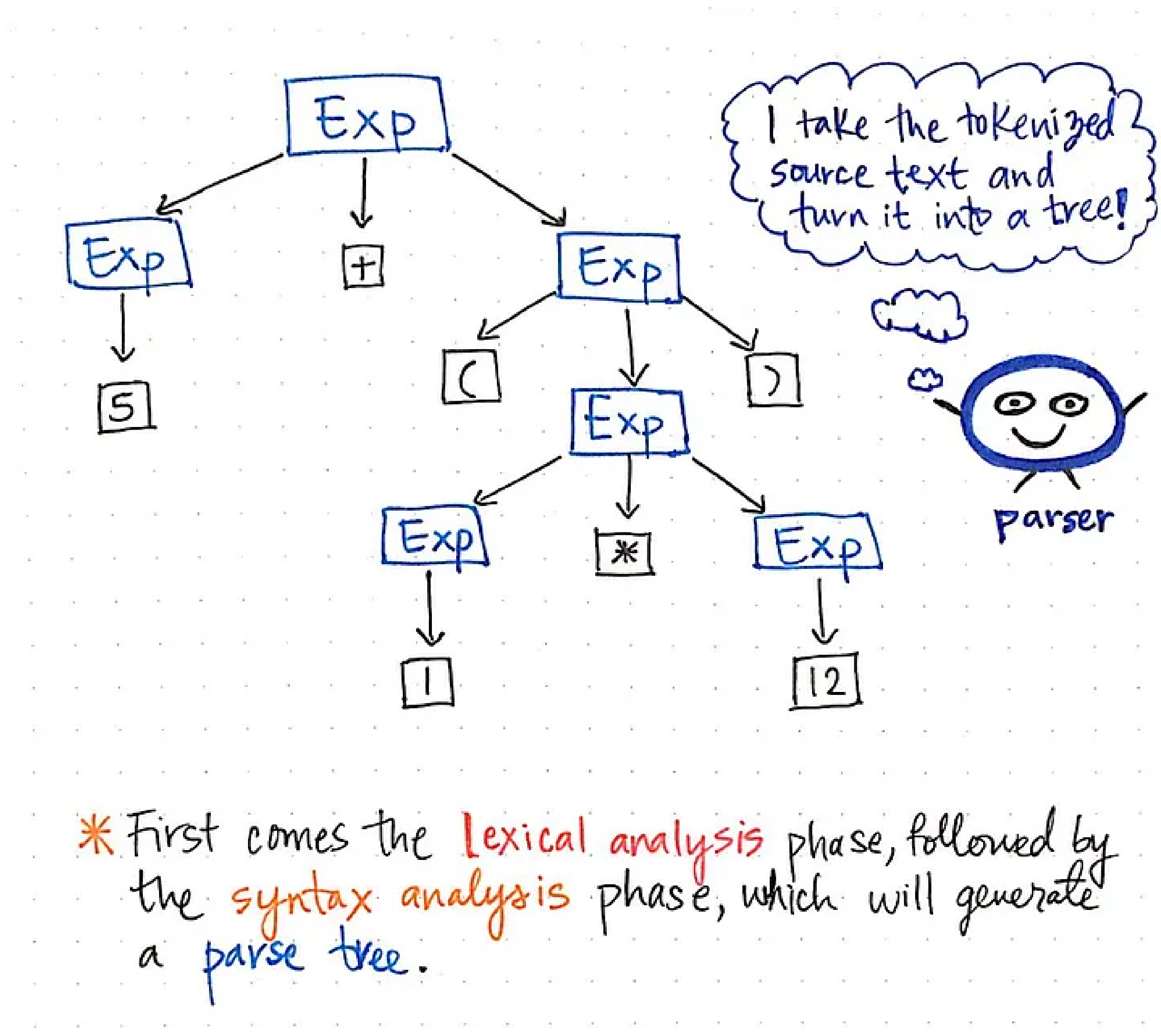
Revisiting the events leading up to parsing!

Let's say we have a super short and sweet source text, which looks like this: `5 + (1 x 12)`.

We'll recall that the first thing that happens in the compilation process is the *scanning* of the text, a job performed by the *scanner*, which results in the text being broken up into its smallest possible parts, which are called *lexemes*. This part will be language agnostic, and we'll end up with the stripped-out version of our source text.

Next, these very lexemes are passed on to the lexer/tokenizer, which turns those small representations of our source text into *tokens,* which will be specific to our language. Our tokens will look something like this: `[5, +, (, 1, x, 12, )]`. The joint effort of the scanner and the tokenizer make up the *lexical analysis* of compilation.

Then, once our input has been tokenized, its resulting tokens are is passed along to our parser, which then takes the source text and builds a parse tree out of it. The illustration below exemplifies what our tokenized code looks like, in parse tree format.


Exp
Exp
+
Exp
5
(
Exp
)
Exp
*
Exp
1
12
I take the tokenized source text and turn it into a tree!
parser
* First comes the lexical analysis phase, followed by the syntax analysis phase, which will generate a parse tree.

The syntax analysis phase generates the parse tree

The work of turning tokens into a parse tree is also called parsing, and is known as the *syntax analysis* phase. The syntax analysis phase depends directly on the lexical analysis phase; thus, lexical analysis must always come first in the compilation process, because our compiler's parser can only do its job once the tokenizer does it's job!

> We can think of the parts of the compiler as good friends, who all depend on each other to make sure that our code is correctly transformed from a text or file into a parse tree.

But back to our original question: where does the abstract syntax tree fit into this friend group? Well, in order to answer that question, it helps to understand the *need* for an AST in the first place.

## Condensing one tree into another

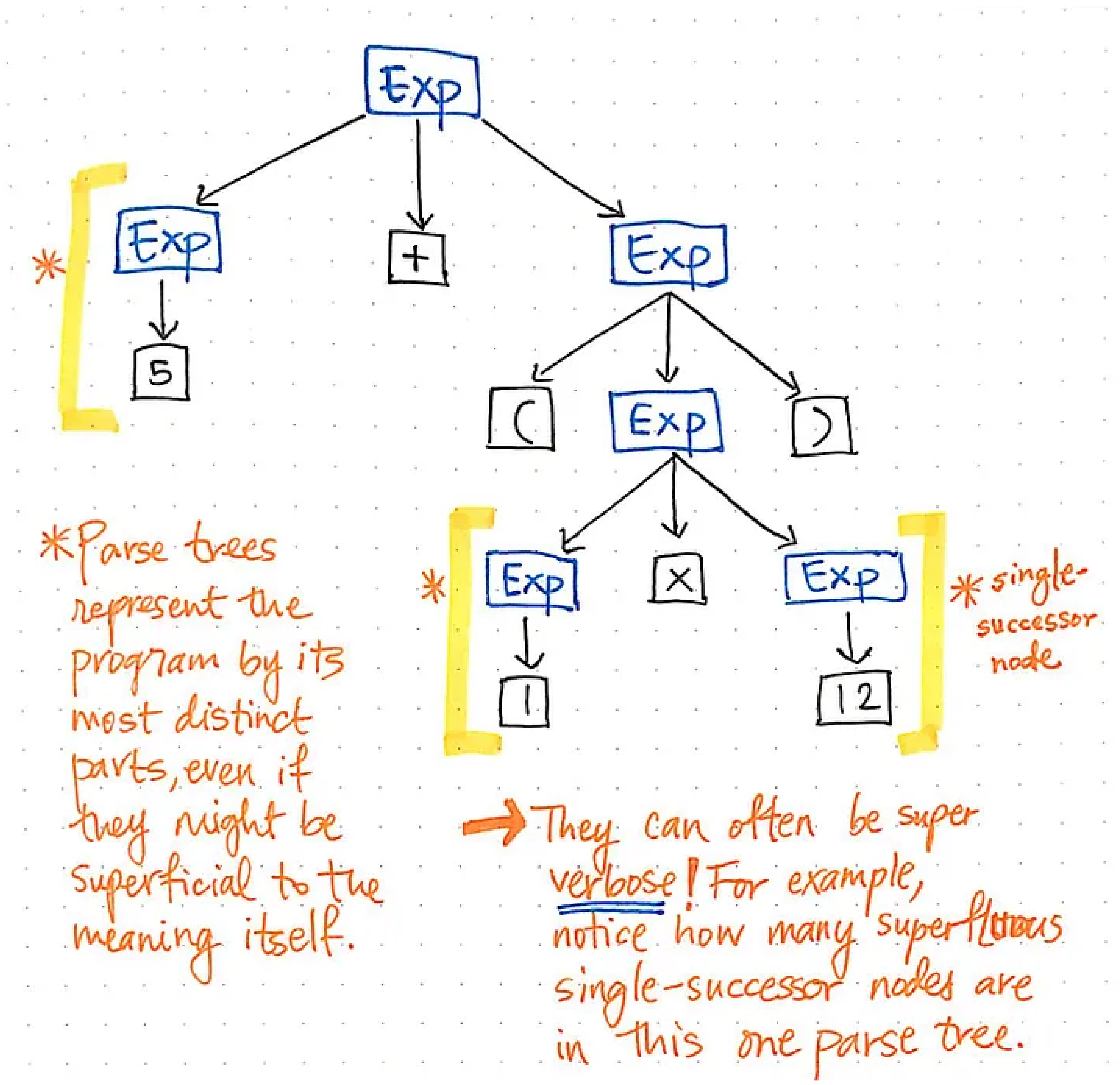
Okay, so now we have two trees to keep straight in our heads. We already had a parse tree, and how there's yet another data structure to learn! And apparently, this AST data structure is just a simplified parse tree. So, why do we need it? What even is the point of it?

Well, let's take a look at our parse tree, shall we?

We already know that parse trees represent our program at its most distinct parts; indeed, this was why the scanner and the tokenizer have such important jobs of breaking down our expression into its smallest parts!

What does it mean to actually represent a program by its most distinct parts?

As it turns out, sometimes all of the distinct parts of a program actually aren't all that useful to us all the time.



Exp
Exp
+
Exp
5
(
Exp
)
Exp
x
Exp
1
12
*
*
*Parse trees represent the program by its most distinct parts, even if they might be superficial to the meaning itself.
* single-successor node
→ They can often be super verbose! For example, notice how many superfluous single-successor nodes are in this one parse tree.

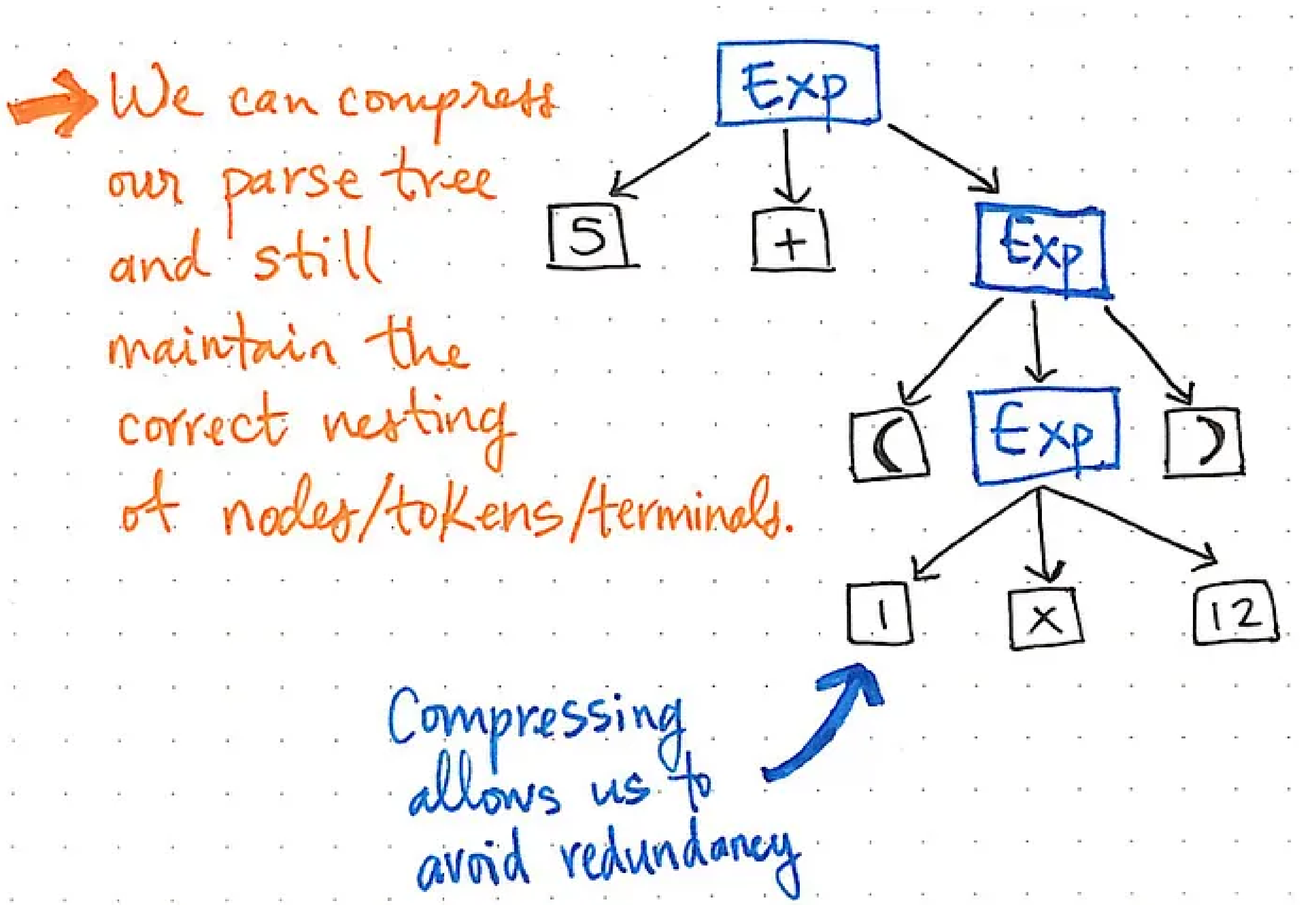
Parse tree can often be super verbose.

Let’s take a look at the illustration shown here, which represents our original expression, `5 + (1 x 12)`, in parse tree tree format. If we take a close look at this tree with a critical eye, we’ll see that there are a few instances where one node has exactly one child, which are also referred to as ***single-successor nodes,*** as they only have one child node stemming from them (or one “successor”).

In the case of our parse tree example, the single-successor nodes have a parent node of an `Expression`, or `Exp`, which have a single successor of some value, such as `5`, `1`, or `12`. However, the `Exp` parent nodes here aren’t actually adding anything of value to us, are they? We can see that they contain token/terminal child nodes, but we don’t really care about the “expression” parent node; all we really want to know is what *is* the expression?

The parent node doesn’t give us any additional information at all once we’ve parsed our tree. Instead, what we’re really concerned with is the single *child*, single-successor node. Indeed, that is the node that gives us the important information, the part that is significant to us: the number and value itself! Considering the fact that these parent nodes are kind of unnecessary to us, it becomes obvious that this parse tree is a kind of verbose.

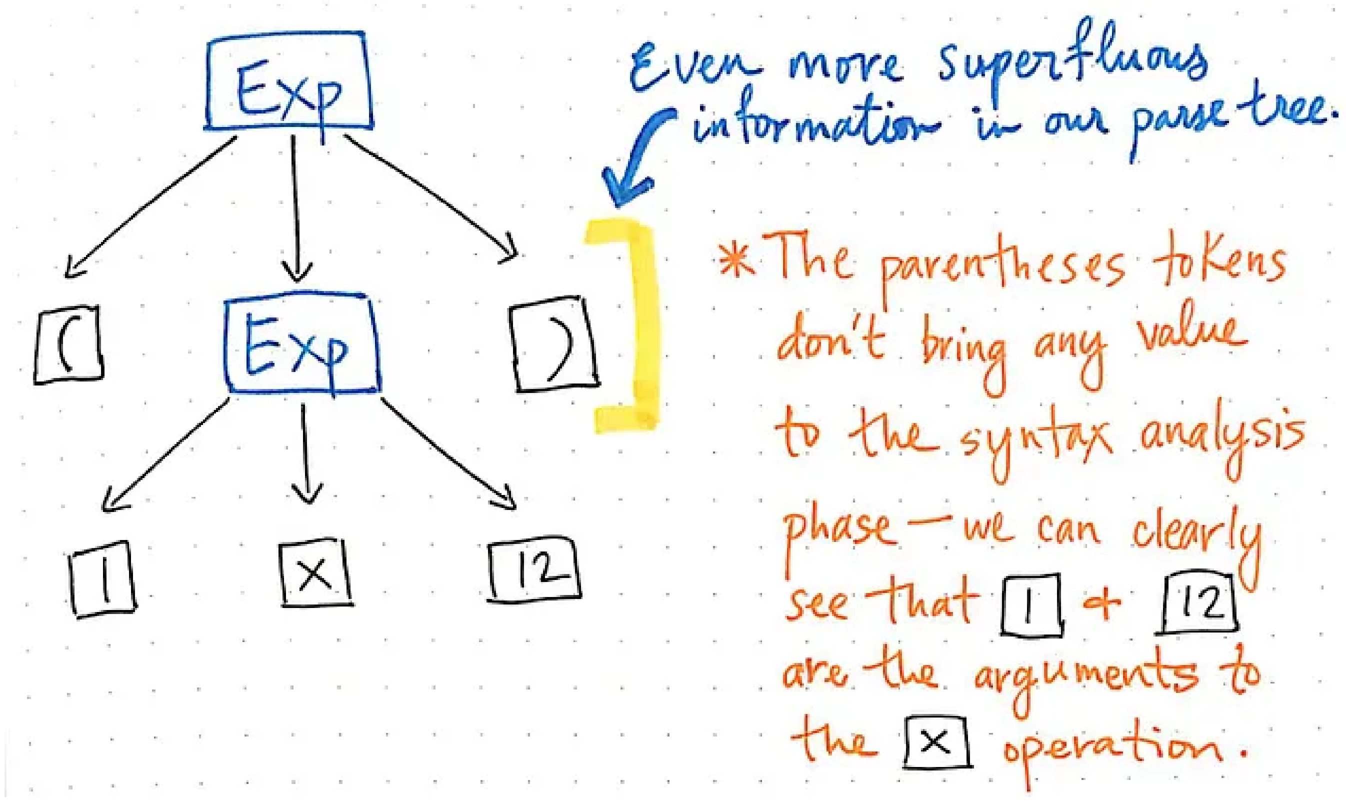
All of these single-successor nodes are pretty superfluous to us, and don’t help us at all. So, let’s get rid of them!


We can compress our parse tree and still maintain the correct nesting of nodes/tokens/terminals.
Exp
5
+
Exp
(
Exp
)
1
x
12
Compressing allows us to avoid redundancy

Compressing a parse tree allows us to avoid redundancy.

If we compress the single-successor nodes in our parse tree, we'll end up with a more compressed version of the same exact structure. Looking at the illustration above, we'll see that we're still maintaining the exact same nesting as before, and our nodes/tokens/terminals are still appearing in the correct place within the tree. But, we've managed to slim it down a bit.

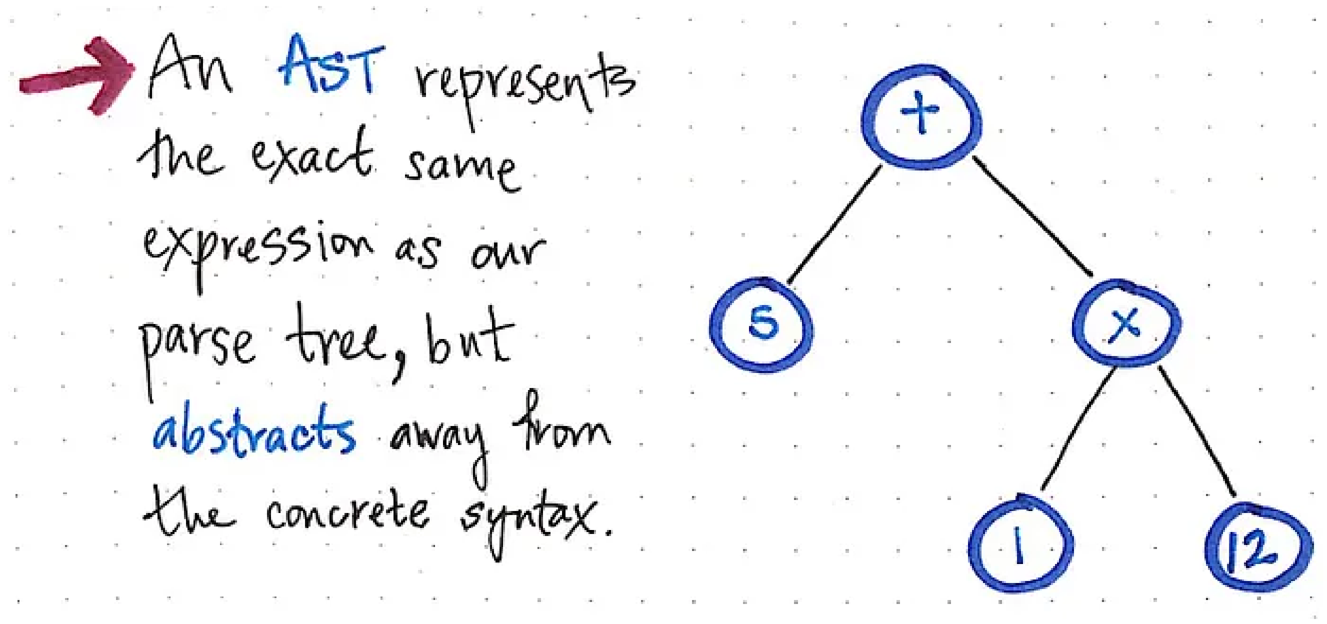
And we can trim some more of our tree, too. For example, if we look at our parse tree as it stands at the moment, we'll see that there's a mirror structure within it. The sub-expression of `(1 x 12)` is nested within parentheses `()`, which are tokens by their own right.


Even more superfluous information in our parse tree.
Exp
(
Exp
)
1
x
12
* The parentheses tokens don't bring any value to the syntax analysis phase — we can clearly see that 1 + 12 are the arguments to the x operation.

Superfluous information that is of no use to us can be removed from a parse tree.

However, these parentheses don’t really help us once we’ve got our tree in place. We already know that `1` and `12` are arguments that will be passed to the multiplication `x` operation, so the parentheses don’t tell us all that much at this point. In fact, we could compress our parse tree even further and get rid of these superfluous leaf nodes.

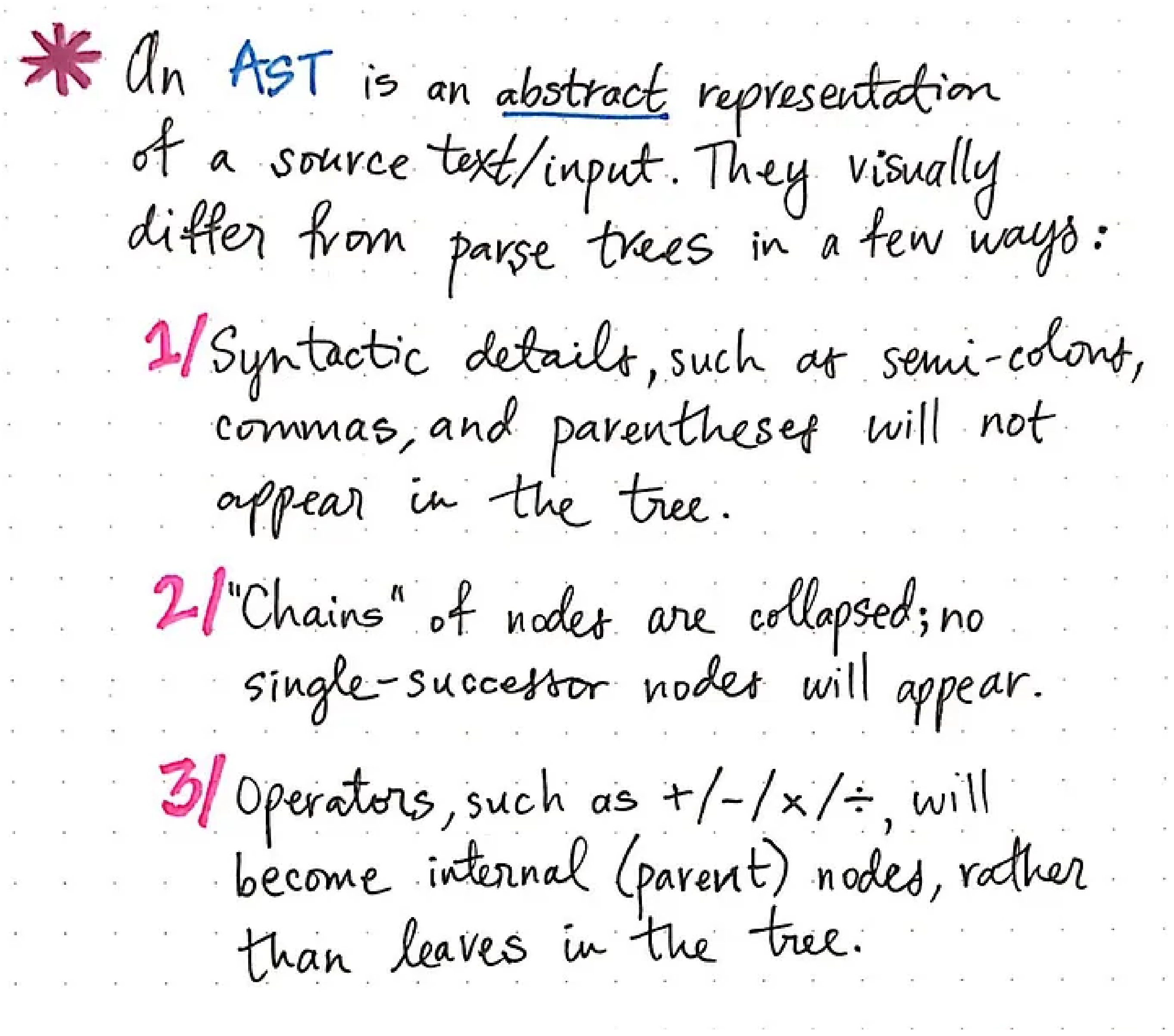
Once we compress and simplify our parse tree and get rid of the extraneous syntactic “dust”, we end up with a structure that looks visibly different at this point. That structure is, in fact, our new and much-anticipated friend: the abstract syntax tree.


An AST represents the exact same expression as our parse tree, but abstracts away from the concrete syntax.
+
5
x
1
12

An AST abstracts away from the concrete syntax.

The image above illustrates the exact same expression as our parse tree: `5 + (1 x 12)`. The difference is that it has abstracted away the expression from the concrete syntax. We don't see any more of the parentheses `()` in this tree, because they're not necessary. Similarly, we don't see non-terminals like `Exp`, since we've already figured out what the "expression" is, and we are able to pull out the *value* that really matters to us — for example, the number `5`.

This is exactly the distinguishing factor between an AST and a CST. We know that an abstract syntax tree ignores a significant amount of the syntactic information that a parse tree contains, and skips "extra content" that is used in parsing. But now we can see exactly *how* that happens!



✱ An AST is an abstract representation of a source text/input. They visually differ from parse trees in a few ways:

1/ Syntactic details, such as semi-colons, commas, and parentheses will not appear in the tree.

2/ "Chains" of nodes are collapsed; no single-successor nodes will appear.

3/ Operators, such as +/-/x/÷, will become internal (parent) nodes, rather than leaves in the tree.

An AST is an abstract representation of a source text.

Now that we've condensed a parse tree of our own, we'll be much better at picking up on some of the patterns that distinguish an AST from a CST.

There a few ways that an abstract syntax tree will visually-differ from a parse tree:

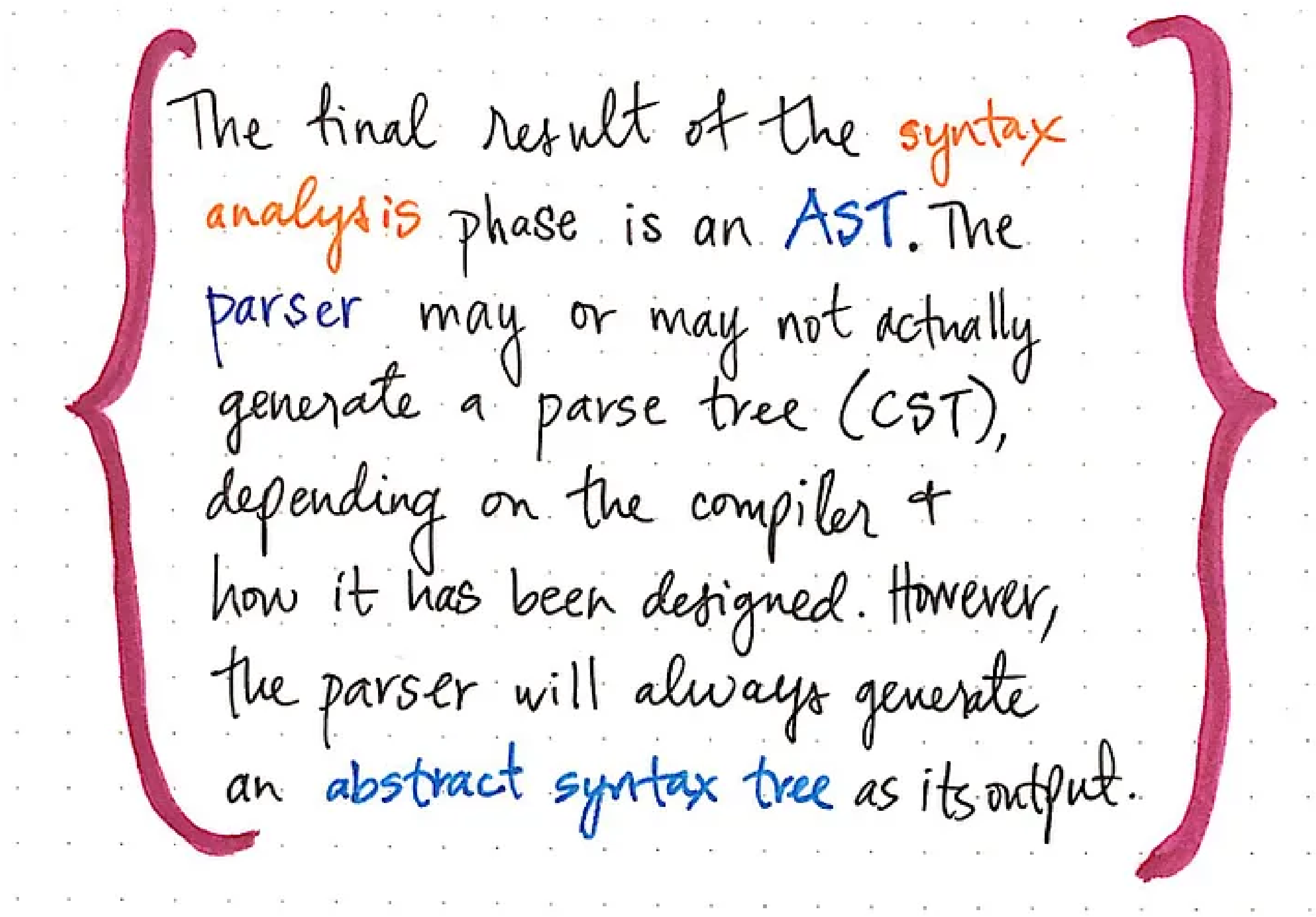
1. An AST will never contain syntactic details, such as commas, parentheses, and semicolons (depending, of course, upon the language).

2. An AST will have collapsed version of what would otherwise appear as single-successor nodes; it will never contain “chains” of nodes with a single child.
3. Finally, any operator tokens (such as `+`, `-`, `x`, and `/`) will become internal (parent) nodes in the tree, rather than the leaves which terminate in a parse tree.

Visually, an AST will always appear more compact than a parse tree, since it is, by definition, a compressed version of a parse tree, with less syntactic detail.

> It would stand to reason, then, that if an AST is a compacted version of a parse tree, we can only really create an abstract syntax tree if we have the things to build a parse tree to begin with!

This is, indeed, how the abstract syntax tree fits into the *larger* compilation process. An AST has a direct connection to the parse trees that we have already learned about, while simultaneously relying upon the lexer to do *its* job before an AST can ever be created.



The final result of the syntax analysis phase is an AST. The parser may or may not actually generate a parse tree (CST), depending on the compiler + how it has been designed. However, the parser will always generate an abstract syntax tree as its output.

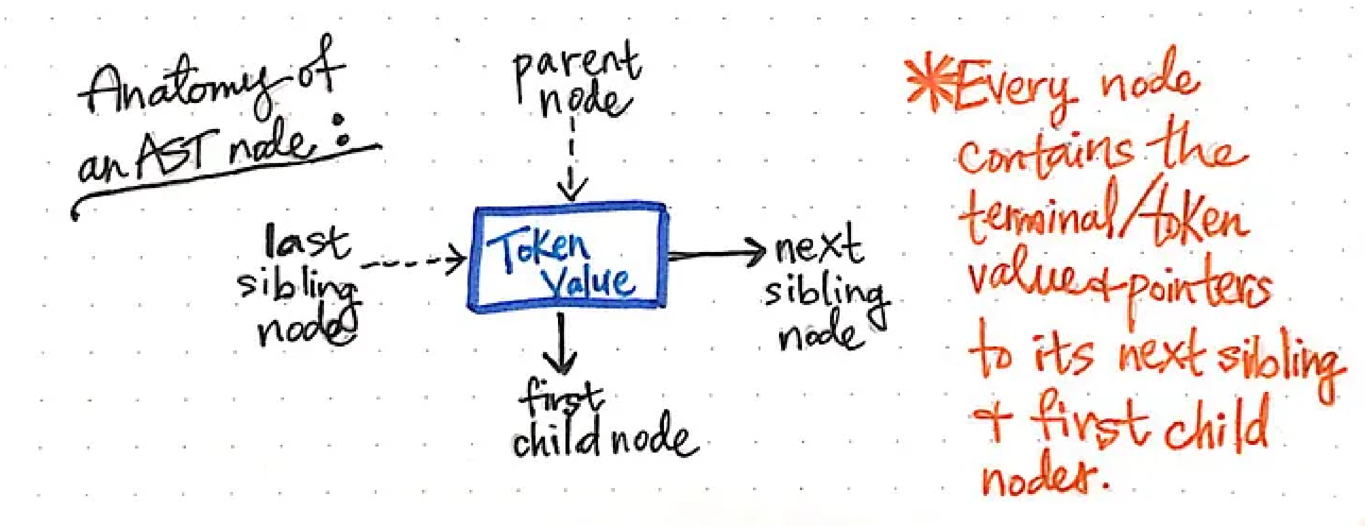
An AST is always the output of the parser.

The abstract syntax tree is created as the final result of the syntax analysis phase. The parser, which is front and center as the main “character” during syntax analysis, may or may not always generate a parse tree, or CST. Depending on the compiler itself, and how it was designed, the parser may directly go straight onto constructing a syntax tree, or AST. But the parser will always generate an AST as its output, no matter whether it creates a parse tree in between, or how many passes it might need to take in order to do so.

## Anatomy of an AST

Now that we know that the abstract syntax tree is important (but not necessarily intimidating!), we can start to dissect it a tiny bit more. An interesting aspect about how the AST is constructed has to do with the nodes of this tree.

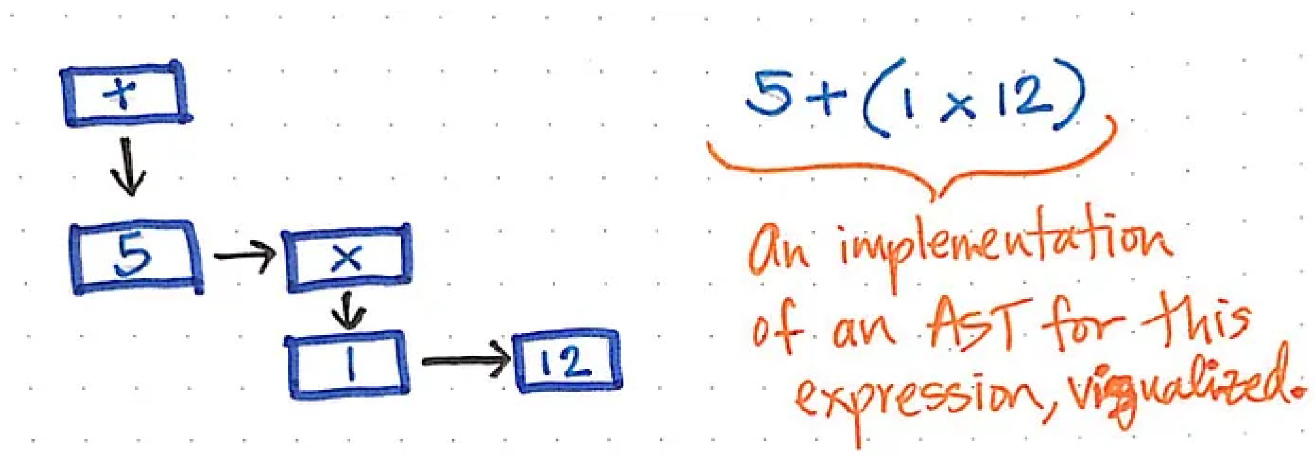
The image below exemplifies the anatomy of a single node within an abstract syntax tree.



The anatomy of an AST node.

We'll notice that this node is similar to others that we've seen before in that it contains some data (a `token` and its `value`). However, it also contains some very specific pointers. Every node in an AST contains references to its *next* sibling node, as well as its *first* child node.

For example, our simple expression of `5 + (1 x 12)` could be constructed into a visualized illustration of an AST, like the one below.


+
5
x
1
12
5+(1x12)
An implementation of an AST for this expression, visualized.

A simplified visualization of our AST expression.

We can imagine that reading, traversing, or “interpreting” this AST might start from the bottom levels of the tree, working its way back up to build up a value or a return `result` by the end.

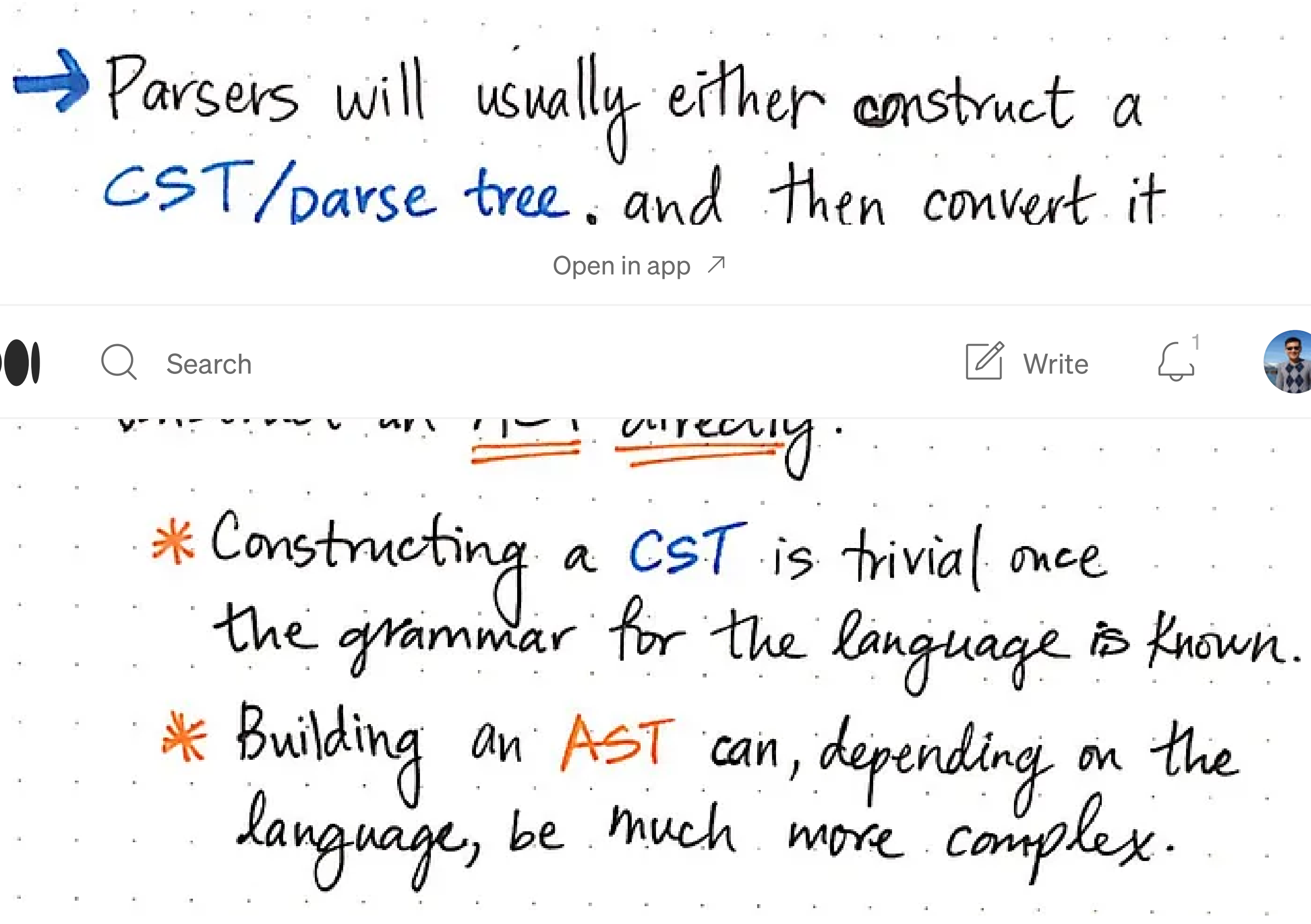
It can also help to see a coded version of the output of a parser to help supplement our visualizations. We can lean on various tools and use preexisting parsers to see a quick example of what our expression might look like when run through a parser. Below is an example of our source text, `5 + (1 * 12)`, run through Esprima, an ECMAScript parser, and its resulting abstract syntax tree, followed by a list of its distinct tokens.

```
// Using Esprima, a ECMAScript parser written in ECMAScript
// var esprima = require('esprima');
// var program = 'const answer = 42';

// Syntax
{
  "type": "Program",
  "body": [
    {
      "type": "VariableDeclaration",
      "declarations": [
          {
              "type": "VariableDeclarator",
              "id": {
```

```
         "type": "Identifier",
         "name": "result"
       },
       "init": {
         "type": "BinaryExpression",
         "operator": "+",
         "left": {
           "type": "Literal",
           "value": 5,
           "raw": "5"
         },
         "right": {
           "type": "BinaryExpression",
           "operator": "*",
           "left": {
             "type": "Literal",
             "value": 1,
             "raw": "1"
           },
           "right": {
             "type": "Literal",
             "value": 12,
             "raw": "12"
           }
         }
       }
     }
   ],
   "kind": "var"
  }
 ],
 "sourceType": "script"
}


// Token List
[
 {
  "type": "Keyword",
  "value": "var"
 },
 {
  "type": "Identifier",
  "value": "result"
 },
  {
    "type": "Punctuator",
    "value": "="
  },
  {
    "type": "Numeric",
    "value": "5"
  },
  {
    "type": "Punctuator",
    "value": "+"
  },
  {
    "type": "Punctuator",
    "value": "("
  },
  {
    "type": "Numeric",
    "value": "1"
  },
  {
    "type": "Punctuator",
    "value": "*"
  },
  {
    "type": "Numeric",
    "value": "12"
  },
  {
    "type": "Punctuator",
    "value": ")"
  },
  {
    "type": "Punctuator",
    "value": ";"
  }
]
```

A code visualization of our AST expression, using JavaScript.

In this format, we can see the nesting of the tree if we look at the nested objects. We'll notice that the values containing `1` and `12` are the `left` and `right` children, respectively, of a parent operation, `*`. We'll also see that the multiplication operation (`*`) makes up the *right subtree* of the entire expression itself, which is why it is nested within the larger `BinaryExpression` object, under the key `"right"`. Similarly, the value of `5` is the single `"left"` child of the larger `BinaryExpression` object.



→ Parsers will usually either construct a CST/parse tree, and then convert it

an AST directly.

* Constructing a CST is trivial once the grammar for the language is known.
* Building an AST can, depending on the language, be much more complex.

Building an AST can be complex sometimes.

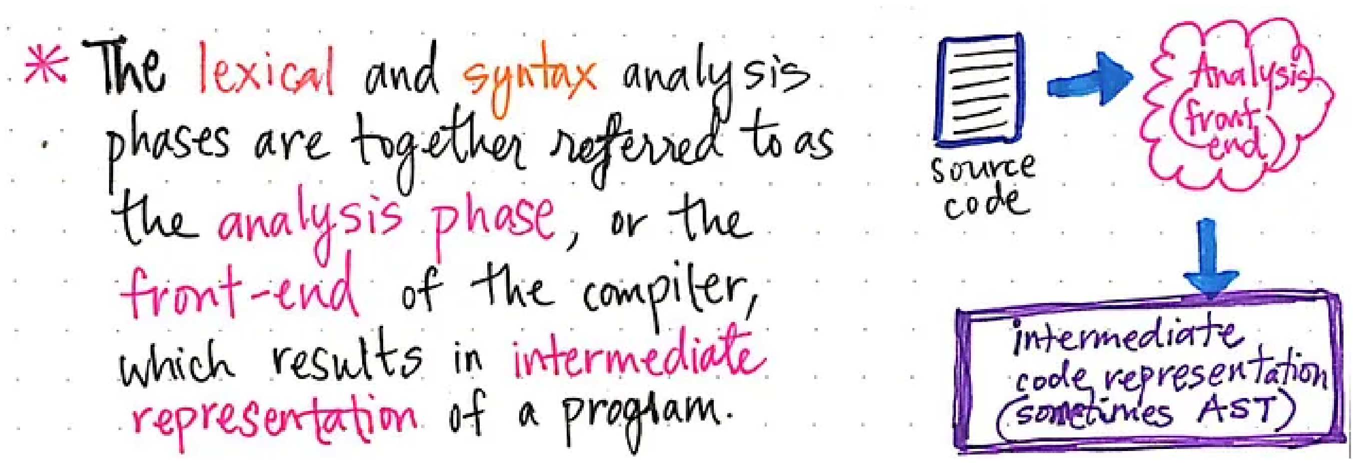
The most *intriguing* aspect of the abstract syntax tree is, however, the fact that even though they are so compact and clean, they aren't always an easy data structure to try and construct. In fact, building an AST can be pretty complex, depending on the language that the parser is dealing with!

Most parsers will usually either construct a parse tree (CST) and then convert it to an AST format, because that can sometimes be easier — even though it means more steps, and generally speaking, more passes through the source text. Building a CST is actually pretty easy once the parser knows the grammar of the language it's trying to parse. It doesn't need to do any complicated work of figuring out if a token is "significant" or not; instead, it just takes exactly what its sees, in the specific order that it sees it, and spits it all out into a tree.

On the other hand, there are some parsers that will try to do all of this as a single-step process, jumping straight to constructing an abstract syntax tree.

> Building an AST directly can be tricky, since the parser has to not only find the tokens and represent them correctly, but also decide which tokens matter to us, and which ones don't.

In compiler design, the AST ends up being super important for more than one reason. Yes, it can be tricky to construct (and probably easy to mess up), but also, it's the last and final result of the lexical and syntax analysis phases combined! The lexical and syntax analysis phases are often jointly called the ***analysis phase***, or ***the front-end*** of the compiler.


* The lexical and syntax analysis phases are together referred to as the analysis phase, or the front-end of the compiler, which results in intermediate representation of a program.
source code
Analysis (front end)
intermediate code representation (sometimes AST)

The inner workings of the front-end of our compiler.

We can think about the abstract syntax tree as the "final project" of the front-end of the compiler. It's the most important part, because its the last thing that the front-end has to show for itself. The technical term for this is called the ***intermediate code representation*** or the ***IR***, because it becomes the data structure that is ultimately used by the compiler to represent a source text.

An abstract syntax tree is the most common form of IR, but also, sometimes the most misunderstood. But now that we understand it a little bit better, we can start to change our perception of this scary structure! Hopefully, it's a little bit less intimidating to us now.

## Resources

There are a whole lot of resources out there on ASTs, in a variety of languages. Knowing where to start can be tricky, especially if you're looking to learn more. Below are a few beginner-friendly resources that dive into a whole lot more detail without being *too* overhwelming. Happy asbtracting!

1. The AST vs the Parse Tree, Professor Charles N. Fischer
2. What's the difference between parse trees and abstract syntax trees?, StackOverflow
3. Abstract vs. Concrete Syntax Trees, Eli Bendersky
4. Abstract Syntax Trees, Professor Stephen A. Edwards
5. Abstract Syntax Trees & Top-Down Parsing, Professor Konstantinos (Kostis) Sagonas
6. Lexical and Syntax Analysis of Programming Languages, Chris Northwood