50.051 Programming Language Concepts

W9-S3 Context Free Grammars (CFG)

Matthieu De Mari

Let us start with a problem

Consider the mathematical expressions below. Which ones are **valid**?

A. 2+7+9
B. (3+4)+7
C. ((6+9)+8)*2
D. 6+2++7
E. 8+3+
F. ((4+3)+8
G. (7+2))(+6

Let us start with a problem

Consider the mathematical expressions below. Which ones are **valid**?

A. **2+7+9**
B. **(3+4)+7**
C. **((6+9)+8)+2**
D. **6+2++7**
E. **8+3+**
F. **((4+3)+8**
G. **(7+2))(+6**

Let us start with a problem

Question: Let us consider mathematical expressions, consisting of

- single digit numbers,
- + operations,
- along with opening and closing parentheses.

Can you write a RegEx to check whether such a string containing a mathematical expression is valid or not?

Some examples of **valid** and **invalid** expressions (same expressions as before):

- **2+7+9**
- **(3+4)+7**
- **((6+9)+8)+2**
- **6+2++7**
- **8+3+**
- **((4+3)+8**
- **(7+2))(+6**

Let us start with a problem

Can you write a RegEx to check whether such a string containing a mathematical expression is valid or not?

→ No, unfortunately, RegEx is not powerful enough to do so.

This would require, among other things, to keep track of:

- How many parentheses have been opened after having read n characters of the given input string,
- How many have been closed after having read n characters of the given input string,
- And in which order opened parentheses have been closed.

Let us start with a problem

It is not possible to do so with a **finite state machine**.

After all, we could technically have **any number of parentheses in the expression, not just any finite number of them**.

So how could we keep track of this number with a RegEx that runs on **finite state machines**?

Important lesson: Syntax analysis tasks, e.g. checking parentheses, will typically require something more powerful than RegEx!

(But what then?)

I never expected I would be saying this in a CS course, but…

I never expected I would be saying this in a CS course, but…

Let us do some chemistry!

I never expected I would be saying this in a CS course, but…

Let us do some chemistry!

(Nani the hell is going on here?!)

A chemical problem

Consider the chemistry expressions below (no ions).
Which ones are **valid**?

A. CO

B. H_2O

C. H^2O

D. $C_{12}H_{22}O_{11}$

E. $C_2\,J_8$

A chemical problem

Consider the chemistry expressions below (no ions).
Which ones are **valid**?

A. CO *(carbon monoxyde)*

B. H_2O *(water)*

C. H^2O *(exponents are not allowed, except for ions)*

D. $C_{12}H_{22}O_{11}$ *(sugar)*

E. C_2J_8 *(no J element in periodic table)*

A chemical problem

Consider the chemistry expressions below (no ions).
Which ones are **valid**?

A. CO *(carbon monoxyde)*

B. H_2O *(water)*

C. H^2O *(exponents are not allowed, except for ions)*

D. $C_{12}H_{22}O_{11}$ *(sugar)*

E. $C_2\,J_8$ *(no J element in periodic table)*

Question: Is there a Regular Expression for checking if these formulas are valid?

A chemical problem

Consider the chemistry expressions below (no ions).
Which ones are **valid**?

A. CO *(carbon monoxyde)*

B. H_2O *(water)*

C. H^2O *(exponents are not allowed, except for ions)*

D. $C_{12}H_{22}O_{11}$ *(sugar)*

E. $C_2\,J_8$ *(no J element in periodic table)*

Question: Is there a Regular Expression for checking if these formulas are valid?

No, because parentheses can be used in chemistry formulas for compounds,
e.g. $Ca_3(PO_4)_2 = 3\text{Ca} + 4\text{P} + 8\text{O}$ → Same problem as before!

Introducing Context Free Grammars

Definition (Context-Free Grammars):

A **Context-Free Grammar (CFG)** is **a formal system used to generate and describe sets of strings based on a specific set of syntax rules**.

It is particularly useful for defining the syntax of programming languages and the structure of most natural languages.

The term "context-free" means that the **production rules** of the CFG are applied independently of the surrounding context.

A context free grammar is defined by **four elements**: a set of **terminals** and **non-terminals**, a **start symbol** and a set of **production rules**.

Elements of Context Free Grammars

Definition (Terminals and Non-Terminals):

A context-free grammar (CFG) revolves around (terminal and non-terminal) symbols.

Terminals are the **basic symbols in a language that cannot be further divided or transformed**.

In the case of our chemistry formulas, these would be elements symbols such as C, O, H, He, etc., as well as digits.

Non-terminals, on the other hand, **represent syntactic patterns or intermediate structures in the language.**

These can be **further decomposed and transformed into sequences of terminals and non-terminals**.

Elements of Context Free Grammars

Definition (Start Symbol):

The **Start Symbol** is a special **non-terminal** symbol **from which the derivation of strings begins**. It represents the main structure of the language, and all other rules ultimately derive from it.

Definition (Production Rules):

Production Rules define how **non-terminal** symbols can be replaced by sequences of **terminals** and **non-terminals**.

Written in the form $\boldsymbol{A} \rightarrow \boldsymbol{B}$, where $\boldsymbol{A}$ is a **non-terminal**, and $\boldsymbol{B}$ is a sequence of **terminals** and/or **non-terminals** that can replace $\boldsymbol{A}$.

Back to our Chemistry

Consider the CFG below.

$$Formula \rightarrow Molecule$$

$$Molecule \rightarrow Element$$
$$Molecule \rightarrow Element_{Count}$$
$$Molecule \rightarrow MoleculeElement$$
$$Molecule \rightarrow MoleculeElement_{Count}$$

$$Element \rightarrow C\ or\ O\ or\ H\ or\ He\ or\ \ldots$$
$$Count\ \rightarrow 1\ or\ 2\ or\ 3\ or\ \ldots$$

Back to our Chemistry

The symbol *Formula* is a **Non-Terminal**, which serves as a **Start Symbol**.

Consider the CFG below.

$$Formula \rightarrow Molecule$$

$$Molecule \rightarrow Element$$
$$Molecule \rightarrow Element_{Count}$$
$$Molecule \rightarrow MoleculeElement$$
$$Molecule \rightarrow MoleculeElement_{Count}$$

$$Element \rightarrow C\ or\ O\ or\ H\ or\ He\ or\ \ldots$$
$$Count\ \rightarrow 1\ or\ 2\ or\ 3\ or\ \ldots$$

Back to our Chemistry

Any symbol written as "*Word*" is a **non-terminal**.

Consider the CFG below.

$$Formula \rightarrow Molecule$$

$$Molecule \rightarrow Element$$
$$Molecule \rightarrow Element_{Count}$$
$$Molecule \rightarrow MoleculeElement$$
$$Molecule \rightarrow MoleculeElement_{Count}$$

$$Element \rightarrow C \ or \ O \ or \ H \ or \ He \ or \ \ldots$$
$$Count \rightarrow 1 \ or \ 2 \ or \ 3 \ or \ \ldots$$

Back to our Chemistry

Consider the CFG below.

All the **elements of the periodic table** and the **non-zero integer numbers** can be used as **terminals**.

$$Formula \rightarrow Molecule$$

$$Molecule \rightarrow Element$$
$$Molecule \rightarrow Element_{Count}$$
$$Molecule \rightarrow MoleculeElement$$
$$Molecule \rightarrow MoleculeElement_{Count}$$

$$Element \rightarrow C \ or \ O \ or \ H \ or \ He \ or \ \ldots$$
$$Count \rightarrow 1 \ or \ 2 \ or \ 3 \ or \ \ldots$$

Back to our Chemistry

Consider the CFG below.

We have defined 7 **production rules**.
Several production rules may appear and start with the same non-terminal.

$$Formula \rightarrow Molecule$$

$$Molecule \rightarrow Element$$
$$Molecule \rightarrow Element_{Count}$$
$$Molecule \rightarrow MoleculeElement$$
$$Molecule \rightarrow MoleculeElement_{Count}$$

$$Element \rightarrow C \text{ or } O \text{ or } H \text{ or } He \text{ or } \ldots$$
$$Count \rightarrow 1 \text{ or } 2 \text{ or } 3 \text{ or } \ldots$$

Back to our Chemistry

We may also use the keyword “or” for convenience

Consider the CFG below.

$$Formula \rightarrow Molecule$$

$$Molecule \rightarrow Element$$
$$Molecule \rightarrow Element_{Count}$$
$$Molecule \rightarrow MoleculeElement$$
$$Molecule \rightarrow MoleculeElement_{Count}$$

$$Element \rightarrow C \text{ or } O \text{ or } H \text{ or } He \text{ or } \ldots$$
$$Count \rightarrow 1 \text{ or } 2 \text{ or } 3 \text{ or } \ldots$$

Back to our Chemistry

Consider the CFG from earlier

$$Form \rightarrow Mol$$
$$Mol \rightarrow Elem$$
$$Mol \rightarrow Elem_{Count}$$
$$Mol \rightarrow MolElem$$
$$Mol \rightarrow MolElem_{Count}$$
$$Elem \rightarrow C\ or\ O\ or\ H\ or\ He\ or\ \ldots$$
$$Count\ \rightarrow 1\ or\ 2\ or\ 3\ or\ \ldots$$

Is the formula CO valid? Yes.

Because, it can be **derived** from the CFG production rules, starting from the start symbol *Formula*.

$Form \rightarrow Mol$ (rule 1)

$Mol \rightarrow MolElem$ (rule 4)

$MolElem \rightarrow ElemElem$ (using rule 2 on *Mol* symbol)

$ElemElem \rightarrow CElem$ (using rule 6 on first *Elem* symbol)

$CElem \rightarrow CO$ (using rule 6 on second *Elem* symbol)

Practice 1 and 2

Consider the CFG from earlier

$$Form \rightarrow Mol$$
$$Mol \rightarrow Elem$$
$$Mol \rightarrow Elem_{Count}$$
$$Mol \rightarrow MolElem$$
$$Mol \rightarrow MolElem_{Count}$$
$$Elem \rightarrow C\ or\ O\ or\ H\ or\ He\ or\ \ldots$$
$$Count\ \rightarrow 1\ or\ 2\ or\ 3\ or\ \ldots$$

Practice 1: Using the same logic, can you prove that

- $C_{12}H_{22}O_{11}$ is a valid expression?

Practice 2: Which additional production rule(s) would you add to cover for $Ca_3(PO_4)_2$?

Practice 1 and 2

Consider the CFG from earlier

$$Form \rightarrow Mol$$
$$Mol \rightarrow Elem$$
$$Mol \rightarrow Elem_{Count}$$
$$Mol \rightarrow MolElem$$
$$Mol \rightarrow MolElem_{Count}$$
$$Elem \rightarrow C\ or\ O\ or\ H\ or\ He\ or\ \ldots$$
$$Count\ \rightarrow 1\ or\ 2\ or\ 3\ or\ \ldots$$

Practice 1: Using the same logic, can you prove that

- $C_{12}H_{22}O_{11}$ is a valid expression?

Answer 1: To be shown on board.

Extra: Repeat to show that

- H_2O is a valid expression,
- H^2O is not a valid expression,
- And $C_2\ J_8$ is not a valid expression.

Practice 1 and 2

Consider the CFG from earlier

$$Form \rightarrow Mol$$
$$Mol \rightarrow Elem$$
$$Mol \rightarrow Elem_{Count}$$
$$Mol \rightarrow MolElem$$
$$Mol \rightarrow MolElem_{Count}$$
$$Elem \rightarrow C\ or\ O\ or\ H\ or\ He\ or\ \ldots$$
$$Count\ \rightarrow 1\ or\ 2\ or\ 3\ or\ \ldots$$

Practice 2: Which additional production rule(s) would you add to cover for $Ca_3(PO_4)_2$?

Practice 1 and 2

Consider the CFG from earlier

$$Form \rightarrow Mol$$
$$Mol \rightarrow Elem$$
$$Mol \rightarrow Elem_{Count}$$
$$Mol \rightarrow MolElem$$
$$Mol \rightarrow MolElem_{Count}$$
$$Elem \rightarrow C\ or\ O\ or\ H\ or\ He\ or\ \ldots$$
$$Count\ \rightarrow 1\ or\ 2\ or\ 3\ or\ \ldots$$

Practice 2: Which additional production rule(s) would you add to cover for $Ca_3(PO_4)_2$?

Answer 2: Many possible answers, but probably something like

$$Mol \rightarrow (Mol)_{Count}$$

Observation: It seems that CFGs are capable of checking parentheses!

Our CFG would reject $Ca_3PO_4)_2$, as there is no possible **derivation** for it.

Derivation and Syntax Validity

Definition (Derivation):

In CFG, a **derivation** is a **sequence of production rules** that

- starts from the start symbol,
- and rewrites non-terminal symbols using production rules,
- until only terminal symbols remain in the expression.

The resulting sequence of terminal symbols forms a string, called the **result of the derivation**.

Derivation and Syntax Validity

Theorem (Syntax Validity and CFGs):

A given string *x* of terminal symbols (e.g. $x = H_2O$) **has a valid syntax**, according to a given context-free grammar,

if and only if,

There exists a derivation for the given CFG, which produces the given string *x* as the result of the derivation.

How is that useful for compilers?

Important property:
Programming languages are ruled by syntax rules, which can be described as CFGs.

For instance, when declaring a variable of type integer (using no arithmetic operations on the right hand side of the equal sign, only literals) the stream of tokens should follow a specific syntax described by the CFG on the right.

Declar as start symbol of our CFG

$$Declar \rightarrow Type\ Identifier = Literal\ ;$$

Several possible Type keywords can be used, and that is described with a CFG rule

$$Type \rightarrow int\ or\ short\ or\ long\ or\ \ldots$$

Valid identifiers and literals are probably going to be represented as RegEx?
(As discussed in Practice activity 6 of W4S3).

How is that useful for compilers?

Use *Declar* as start symbol of our CFG

$$Declar \rightarrow Type\ Identifier = Literal\ ;$$

Several possible Type keywords can be used, and that is described with a CFG production rule (int, short, long will then be terminals).

$$Type \rightarrow int\ or\ short\ or\ long\ or\ \ldots$$

→ Can be used to check if a sequence of tokens makes sense or not! (Will catch the "identifier token followed by literal token" problem)

Practice 3

Question: Let us consider mathematical expressions, consisting of

- single digit numbers,
- + operations,
- along with opening and closing parentheses.

Can you write a CFG to check whether such a given mathematical expression has a valid syntax or not?

Some examples of **valid** and **invalid** expressions (same expressions as before):

- **2+7+9**
- **(3+4)+7**
- **((6+9)+8)+2**
- **6+2++7**
- **8+3+**
- **((4+3)+8**
- **(7+2))(+6**

Practice 3

Question: Let us consider mathematical expressions, consisting of

- single digit numbers,
- + operations,
- along with opening and closing parentheses.

Can you write a CFG to check whether such a given mathematical expression has a valid syntax or not?

Answer: Probably something along the lines of

$$Start \rightarrow Expr$$
$$Expr \rightarrow Expr + Expr$$
$$Expr \rightarrow Num$$
$$Expr \rightarrow (Expr)$$
$$Num \rightarrow 0 \; or \; 1 \; or \; 2 \; or \; \dots or \; 9$$

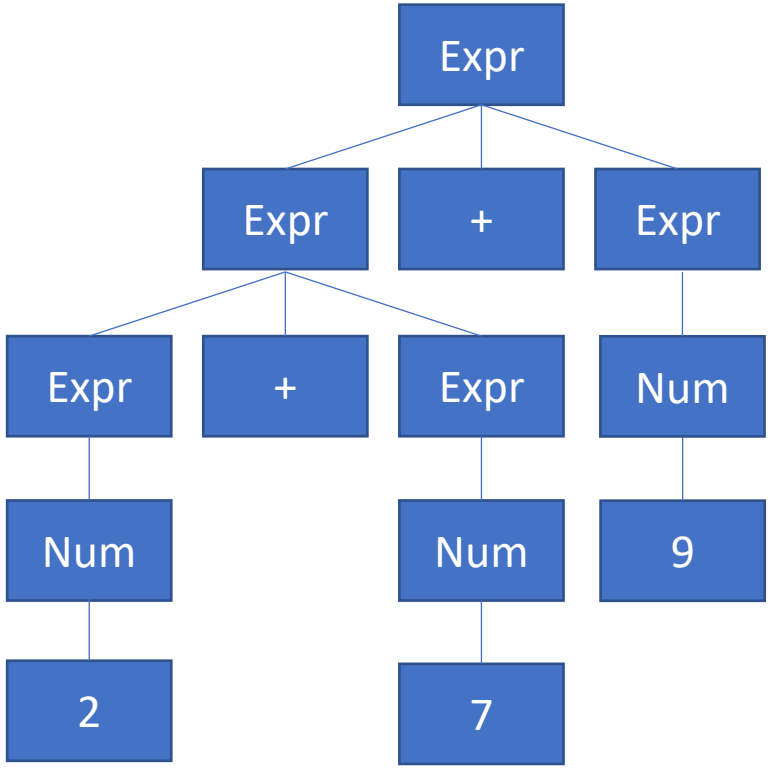
This is also something we could use to check if an arithmetic expression in our source code has a valid syntax!

Parse tree of a derivation

Derivation (parse tree of a CFG derivation):

For a given CFG derivation, we can build a parse tree,

- Whose root is the start symbol,
- Where every production rule, $X \to Y_1 \dots Y_N$ in the derivation sequence, adds children nodes $Y_1, \dots, Y_N$ to the node X.



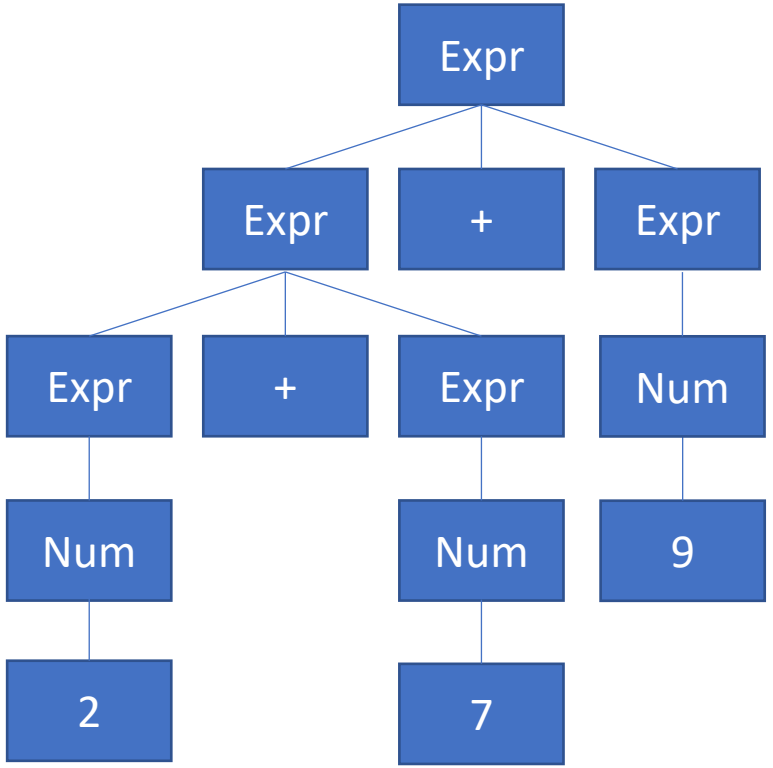
Parse tree of a derivation

Reusing the CFG rules below

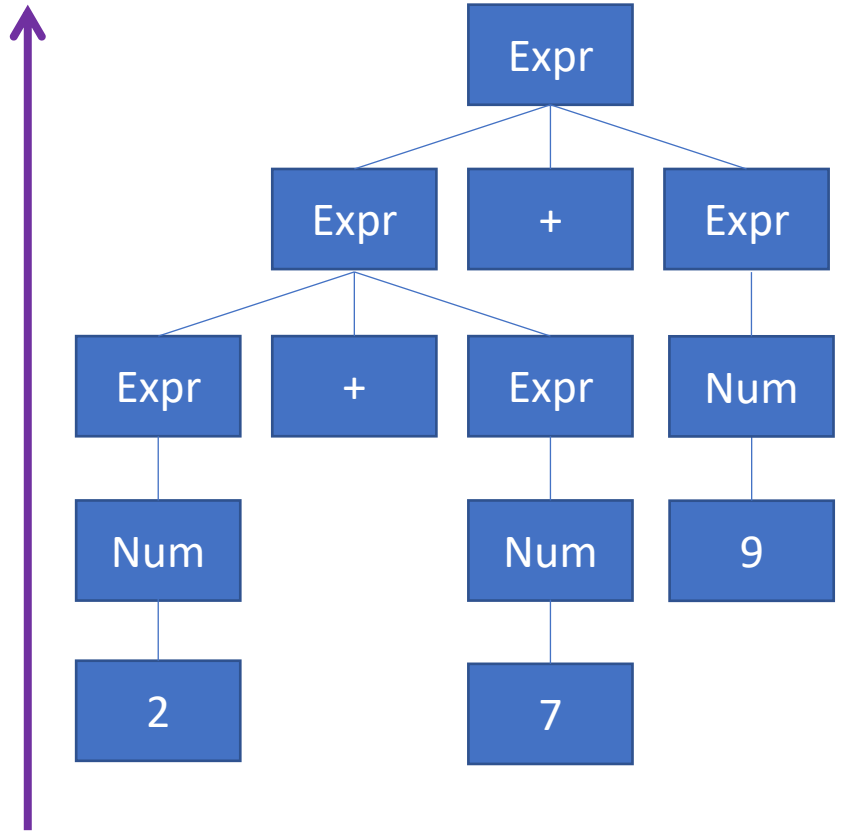
$$Start \rightarrow Expr$$
$$Expr \rightarrow Expr + Expr$$
$$Expr \rightarrow Num$$
$$Expr \rightarrow (Expr)$$
$$Num \rightarrow 0\ or\ 1\ or\ 2\ or\ \ldots or\ 9$$

We can define the parse tree for the derivation of 2+7+9, as shown on the right.

It describes the derivation used to check that 2+7+9 is a valid expression according to the CFG.



Parse tree of a derivation



Derivation (parse tree of a CFG derivation):

For a given CFG derivation, we can build a parse tree,

- Whose root is the start symbol,
- Where every production rule, $X \rightarrow Y_1 \ldots Y_N$ in the derivation sequence, adds children nodes $Y_1, \ldots, Y_N$ to the node X.

This parse tree is interesting because it shows the **order** in which we should compute the different operations, starting with 2 and 7, then 2+7, and finally (2+7)+9.

Quick question

Assuming that a given string *x* has a valid syntax for a given CFG and admits a valid derivation…

→ **Is the valid derivation unique?**

→ **Is there only one parse tree that could have been defined?**

Quick question

Assuming that a given string *x* has a valid syntax for a given CFG and admits a valid derivation…

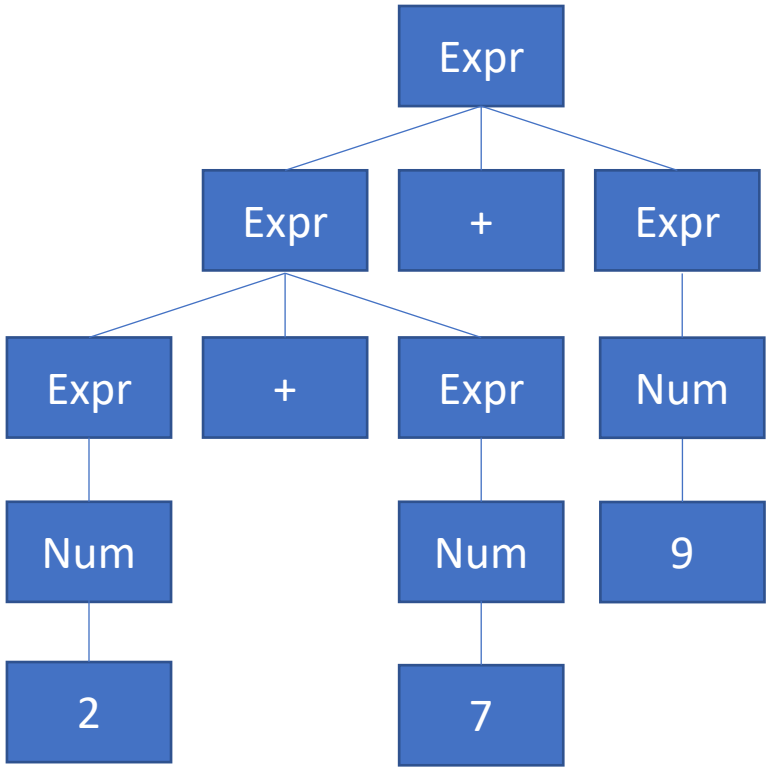
→ **Is the valid derivation unique?**

→ **Is there only one parse tree that could have been defined?**

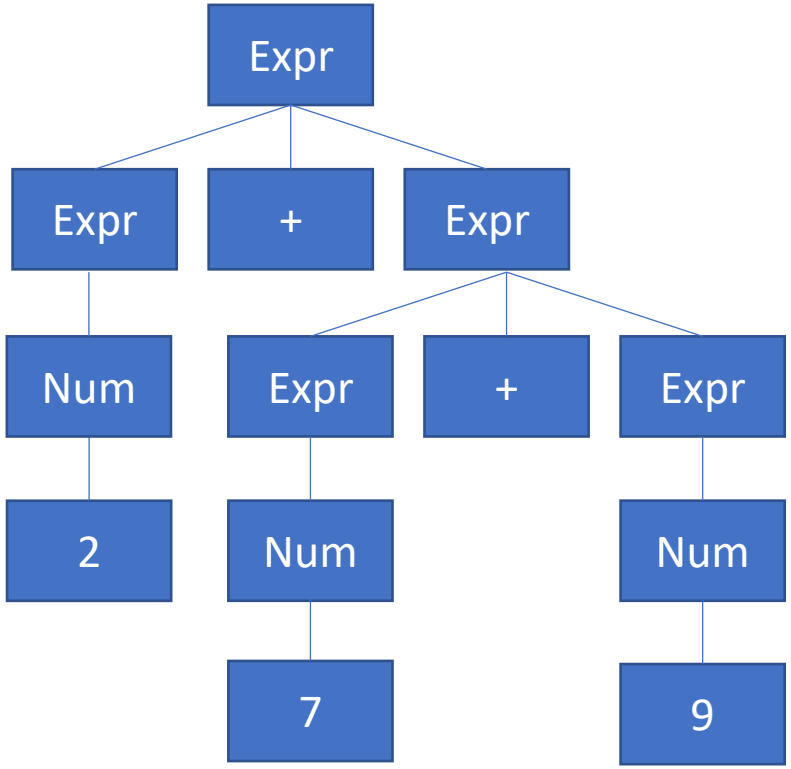
In general, no, multiple valid derivations might do the trick…

And that ambiguity might even be a problem in certain scenarios…

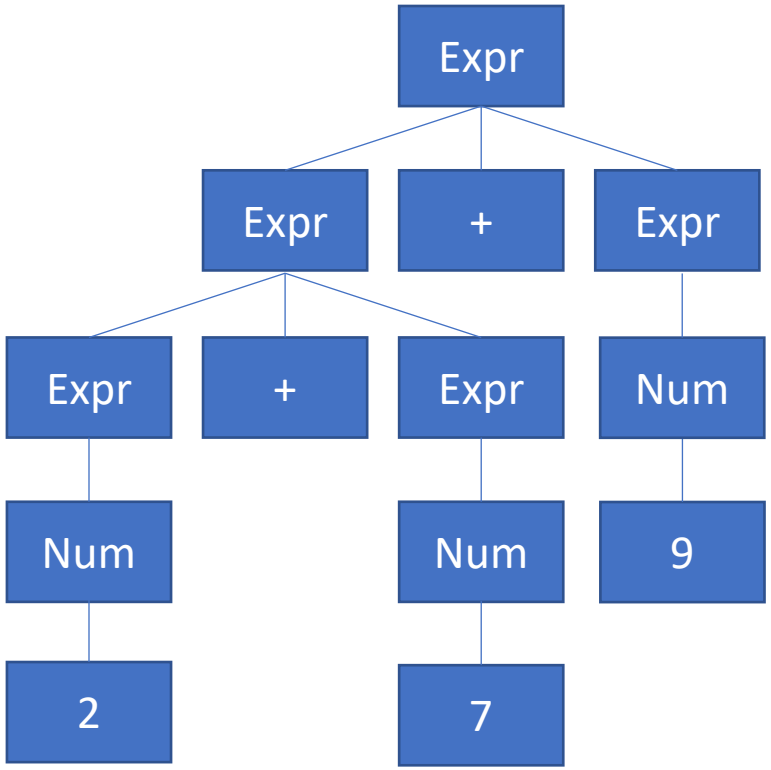
Two parse trees for 2+7+9



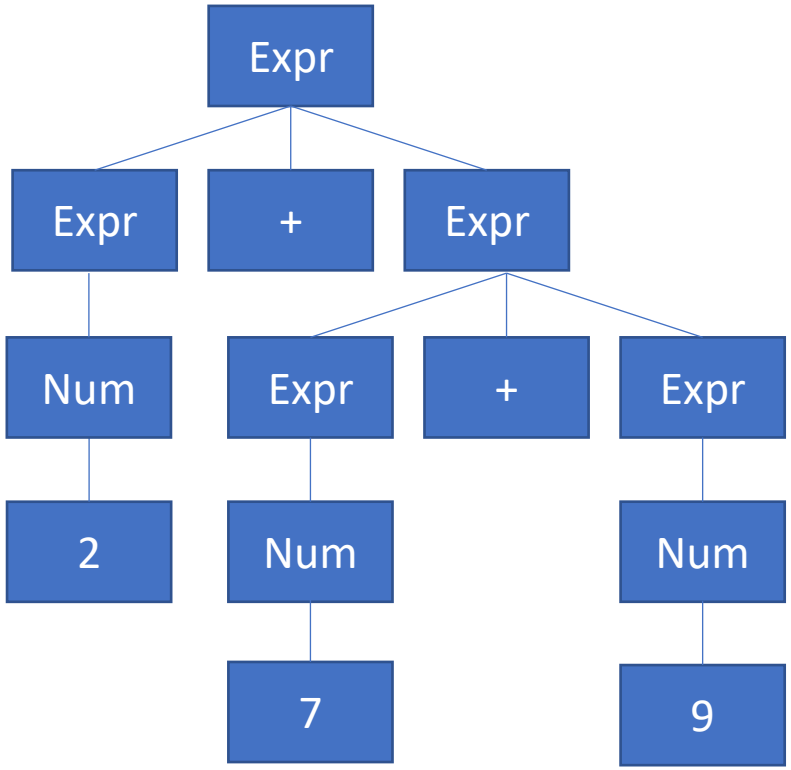
Question: Does that make a difference?



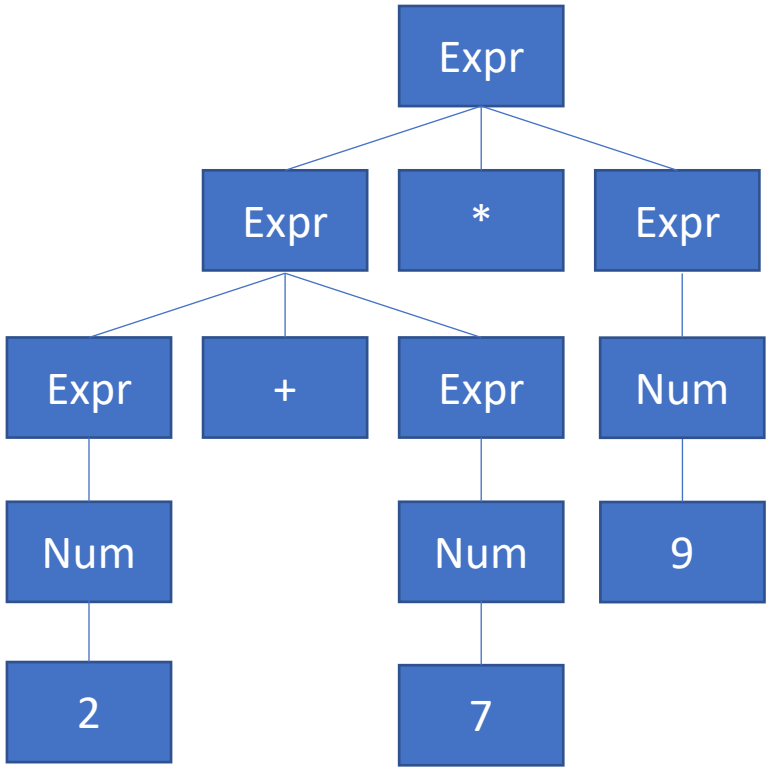
Two parse trees for 2+7+9



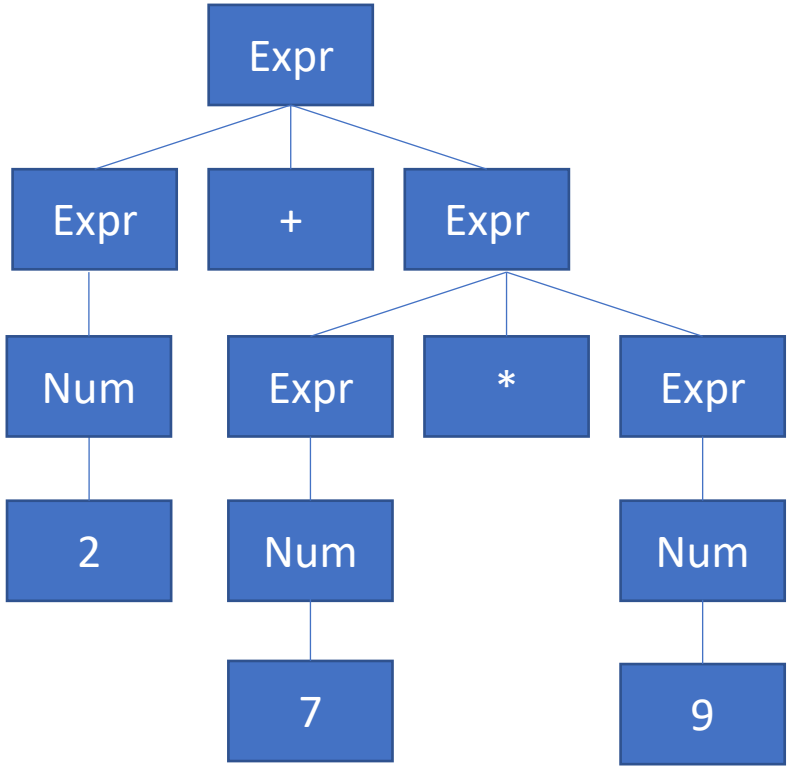
Question: Does that make a difference?
At the moment, no, because the precedence order and the sequence of operations does not matter here (at the moment, we have additions only).



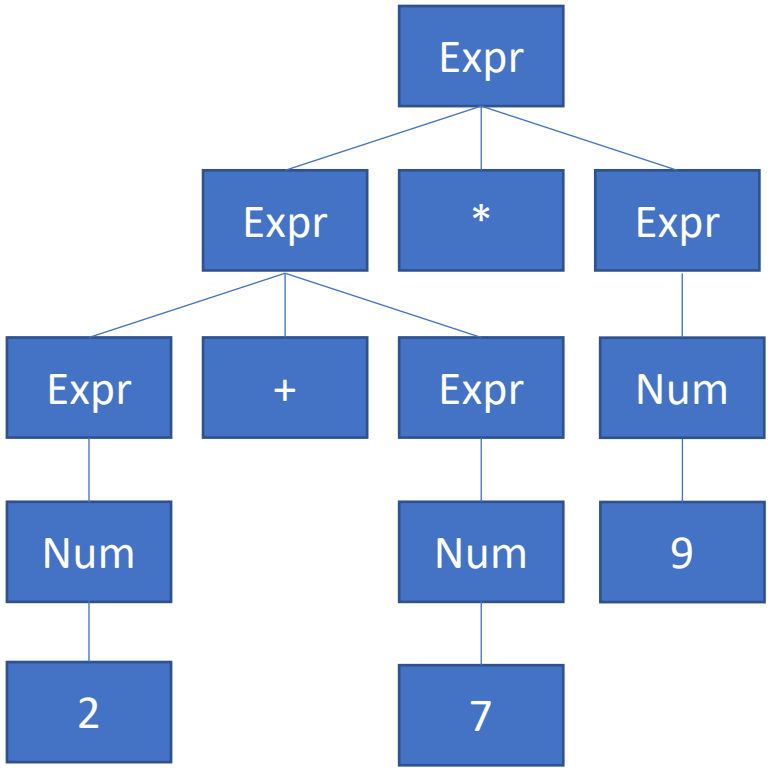
Two parse trees for 2+7*9



But what if we were building parse trees for 2+7*9 instead?



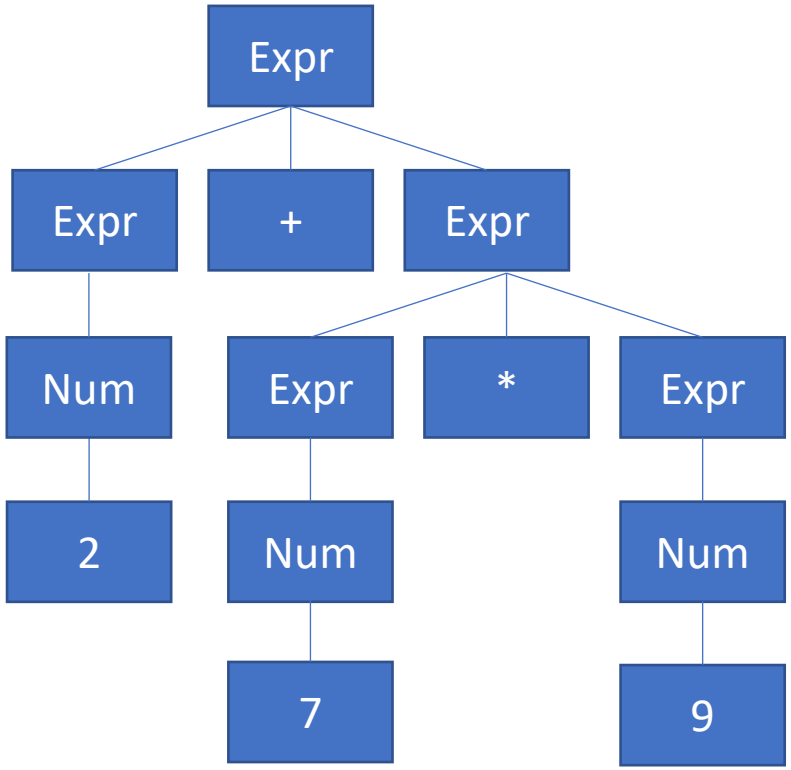
Two parse trees for 2+7*9



But what if we were building parse trees for 2+7*9 instead?

The right tree represents 2+(7*9). The left one represents (2+7)*9.

The order matters in that case!



Ambiguity

Definition (Ambiguity in a CFG derivation):

When using a CFG to check the syntax validity of an expression and building a parse tree, we say that a **CFG is ambiguous** if it can lead to two different derivations with two different parse trees.

In the case of arithmetic expressions and programming languages, ambiguity could prove problematic, because it means that

- Two different derivations might exist,
- Producing two different parse trees,
- And the result of both operations following the two parse trees might differ and lead to different outcomes for a given program (not good!).

Checking ambiguity algorithmically

Theorem (On checking the ambiguity of a CFG algorithmically):

For any given CFG… **There is no algorithm to check if a given CFG is ambiguous or not.**

This is known as the ambiguity problem for context-free grammars, and it is proven to be an **undecidable problem**.

Checking ambiguity algorithmically

Theorem (On checking the ambiguity of a CFG algorithmically):

For any given CFG… **There is no algorithm to check if a given CFG is ambiguous or not.**

This is known as the ambiguity problem for context-free grammars, and it is proven to be an **undecidable problem**.

*(**Note:** Similarly, there is no general algorithm that can determine whether a given program contains an infinite loop. This undecidable problem is known as the Halting problem. It means that you cannot define a compiler program that can check for the presence of infinite loops in the compiled source code. Sad!)*

Checking ambiguity algorithmically

Theorem (On checking the ambiguity of a CFG algorithmically):

For any given CFG… **There is no algorithm to check if a given CFG is ambiguous or not.**

This is known as the ambiguity problem for context-free grammars, and it is proven to be an **undecidable problem**.

There are however **manual methods** for designing CFGs that will be guaranteed to be non-ambiguous.

We will investigate them on an upcoming lecture.

Ambiguous grammars and compilers

Ambiguous grammars can cause significant problems in compilers.

- They make it challenging to derive the intended meaning or structure of the input program.
- Moreover, they can lead to incorrect parsing, which may result in incorrect intermediate code generation or even compiler crashes.
- Therefore, it is crucial to eliminate ambiguity in context-free grammars used in compiler design.

While checking that a CFG might be ambiguous is impossible algorithmically, there are however **manual methods** for designing CFGs that will be non-ambiguous.

Ambiguous grammars and compilers

Not all context-free grammars can be made non-ambiguous.

- Most CFGs used in programming, such as the CFG for checking balanced parentheses or the CFG for arithmetic expressions, can be represented using non-ambiguous grammars (that's a relief!).
- **But, ultimately, it is your responsibility as a programming language designer to define rules for the syntax of your programming language that can be translated into non-ambiguous CFGs!**
- Keep in mind that ambiguous grammars can often appear more concise or easier to define than their non-ambiguous counterparts. However, the problems they cause in compiler design often outweigh the benefits of simplicity or conciseness.

Ambiguous grammars and compilers

To ensure that there is only one correct parse tree for a given string, it is essential to eliminate ambiguity in the grammar.

Some techniques for resolving ambiguity in CFGs manually, will be discussed later during this lecture:

- **Rewriting the grammar entirely,**
- **Introducing precedence,**
- **Introducing delimiters,**
- **Introducing associativity rules.**

But for now…

Quiz time!

What is a Context-Free Grammar (CFG)?

A. A set of rules that can generate all strings in a language
B. A method for tokenizing source code
C. A formal system for describing the structure of a language
D. A technique for optimizing compiler performance

Quiz time!

What is a Context-Free Grammar (CFG)?

A. A set of rules that can generate all strings in a language

B. A method for tokenizing source code

C. A formal system for describing the structure of a language

D. A technique for optimizing compiler performance

Quiz time!

Which of the following best describes a production rule in a CFG?

A. A rule for scanning the input text
B. A rule for optimizing the generated code
C. A rule for reducing the number of steps in a computation
D. A rule for replacing a non-terminal symbol with a sequence of terminal and non-terminal symbols

Quiz time!

Which of the following best describes a production rule in a CFG?

A. A rule for scanning the input text

B. A rule for optimizing the generated code

C. A rule for reducing the number of steps in a computation

D. A rule for replacing a non-terminal symbol with a sequence of terminal and non-terminal symbols

Quiz time!

What is a derivation in the context of CFGs?

A. The process of breaking down a string into its constituent tokens
B. The process of generating code for a given input
C. The process of applying production rules to generate a string in the language
D. The process of defining and optimizing the structure of a parse tree

Quiz time!

What is a derivation in the context of CFGs?

A. The process of breaking down a string into its constituent tokens

B. The process of generating code for a given input

C. The process of applying production rules to generate a string in the language

D. The process of defining and optimizing the structure of a parse tree

Quiz time!

What is a parse tree?

A. A data structure for representing the structure of a language
B. A tree used for optimizing compiler performance
C. A tree representing the derivation of a string using a CFG
D. A tree used for breaking down a string into tokens

Quiz time!

What is a parse tree?

A. A data structure for representing the structure of a language

B. A tree used for optimizing compiler performance

C. A tree representing the derivation of a string using a CFG

D. A tree used for breaking down a string into tokens

Quiz time!

What is an ambiguous context-free grammar?

A. A grammar that can generate two different strings for the same derivation

B. A grammar whose production rules cannot produce a result string consisting of terminal symbols only

C. A grammar that generates only one parse tree for each string

D. A grammar that can generate more than one parse tree for the same string

Quiz time!

What is an ambiguous context-free grammar?

A. A grammar that can generate two different strings for the same derivation

B. A grammar whose production rules cannot produce a result string consisting of terminal symbols only

C. A grammar that generates only one parse tree for each string

D. A grammar that can generate more than one parse tree for the same string

Rewriting a CFG

All approaches to eliminate ambiguity will require to **rewrite the grammar**, in a way that eliminates the possibility of having multiple parse trees for a given input string.

The rewriting task involves following simple rules that will:

- **Restructure the production rules,**
- **And/or introduce new non-terminal symbols to more clearly define the intended structure of the language.**

By doing this, we ensure that there is only one possible derivation for each string, leading to a unique parse tree.

A first disambiguation example

As an example…

Consider the CFG for simplified arithmetic expressions using positive integers, additions and multiplications below.

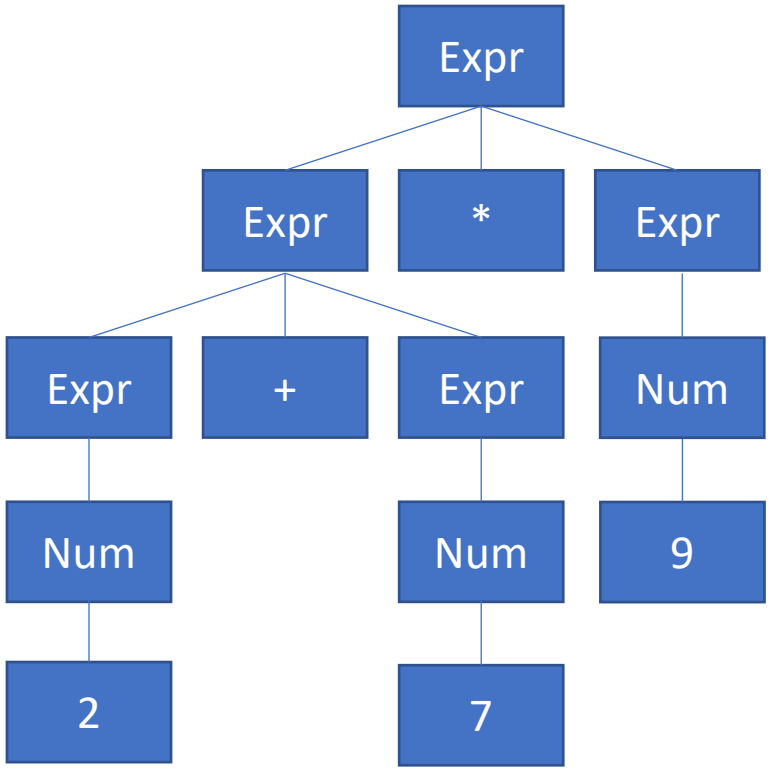
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$
$$E \rightarrow N$$
$$N \rightarrow 0 \ or \ 1 \ or \ 2 \ or \ \ldots$$

This grammar is ambiguous, because it does not specify the precedence order of + and *.

As we have seen in the previous lesson, it might lead to two different parse trees for the string 2*3+7, with completely different behaviours corresponding to two operations:

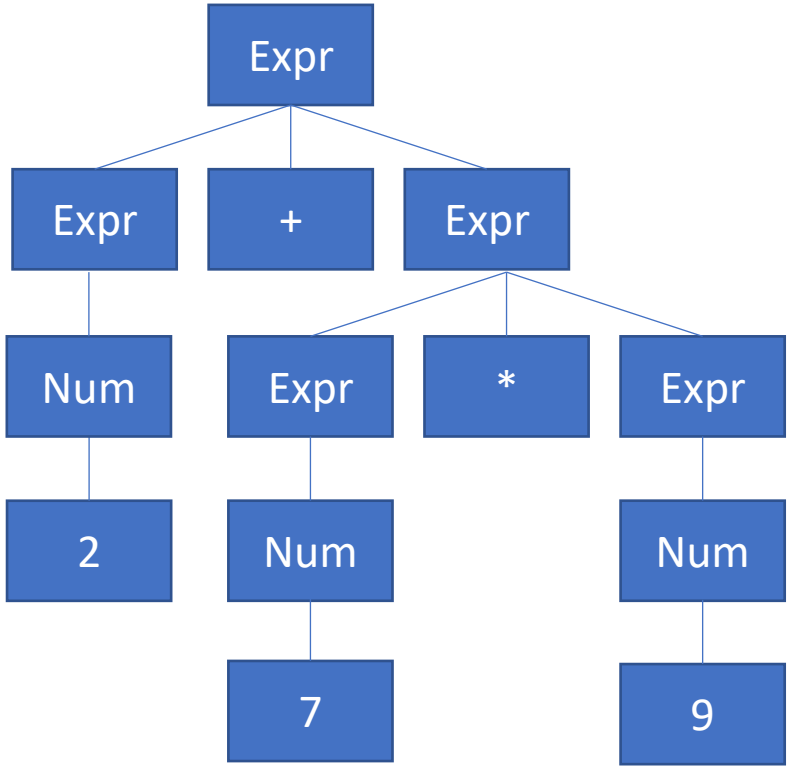
- (2*3)+7
- 2*(3+7)

Two parse trees for 2+7*9



Remember…

Ambiguity is not good for us!



A first disambiguation example

Now, consider this second CFG for simplified arithmetic expressions using positive integers, additions and multiplications below.

$$E \rightarrow E + T \; or \; T$$
$$T \rightarrow T * F \; or \; F$$
$$F \rightarrow (E) \; or \; N$$
$$N \rightarrow 0 \; or \; 1 \; or \; 2 \; or \; \ldots$$

Now, try using this second CFG and build a parse tree for the string 2*3+7.

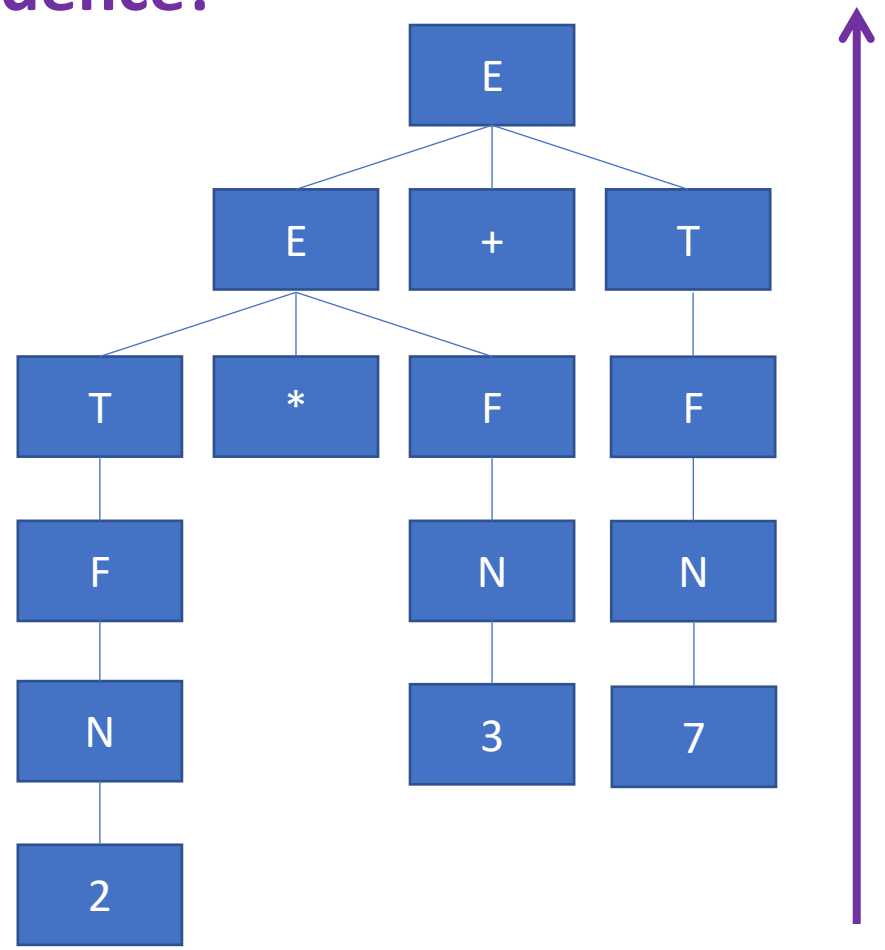
A first disambiguation example

Now, consider this second CFG for simplified arithmetic expressions using positive integers, additions and multiplications below.

$$E \rightarrow E + T \; or \; T$$
$$T \rightarrow T * F \; or \; F$$
$$F \rightarrow (E) \; or \; N$$
$$N \rightarrow 0 \; or \; 1 \; or \; 2 \; or \; \ldots$$

Now, try using this second CFG and build a parse tree for the string 2*3+7.

Only one possible tree and correct precedence!



A first disambiguation example

Now, consider this second CFG for simplified arithmetic expressions using positive integers, additions and multiplications below.

$$E \rightarrow E + T \; or \; T$$
$$T \rightarrow T * F \; or \; F$$
$$F \rightarrow (E) \; or \; N$$
$$N \rightarrow 0 \; or \; 1 \; or \; 2 \; or \; \ldots$$

Now, try using this second CFG and build a parse tree for the string 2*3+7.

How?! Why is this second CFG non-ambiguous now?

- New non-terminal symbols T and F to represent terms and factors.
- **Effectively established that multiplication has higher precedence than addition, as the * operator is now closer to the leaves of the parse tree.**
- Only one parse tree for each arithmetic expression!

A second disambiguation example

As a second example…

Consider the CFG below, which can be used for defining a (nested) if else statement.

$$S \rightarrow "if\ E\ then\ S"$$
$$or\ "if\ E\ then\ S\ else\ S"$$
$$E \rightarrow (an\ expression)$$
$$S \rightarrow (a\ statement)$$

This grammar is ambiguous and can lead to a problem called the **dangling else problem**.

Consider the string of tokens "if a then if b then s1 else s2".

Which if does the else refer to?

A second disambiguation example

As a second example…

Consider the CFG below, which can be used for defining a (nested) if else statement.

$$S \rightarrow "if\ E\ then\ S"$$
$$or\ "if\ E\ then\ S\ else\ S"$$
$$E \rightarrow (an\ expression)$$
$$S \rightarrow (a\ statement)$$

This grammar is ambiguous and can lead to a problem called the **dangling else problem**.

Consider the string of tokens "if a then if b then s1 else s2".

Which if does the else refer to?

```
if a then
  if b then
    statement1
  else
    statement2
```

```
if a then
  if b then
    statement1
else
  statement2
```

A second disambiguation example

Let us we rewrite the CFG as this now

$$S \rightarrow M \; or \; U$$

$$M \rightarrow "if \; E \; then \; M \; else \; M"$$
$$or \; "if \; E \; then \; U \; else \; M"$$
$$or \; "if \; E \; then \; M \; else \; U"$$

$$U \rightarrow "if \; E \; then \; S"$$

$$E \rightarrow (an \; expression)$$
$$S \rightarrow (a \; statement)$$

This CFG is no longer ambiguous, as it now establishes a precedence: **The first if that has not yet a matching else will now receive the else clause.**

```
if a then
  if b then
    statement1
  else
    statement2
```

```
if a then
  if b then
    statement1
else
  statement2
```

A second disambiguation example

Let us we rewrite the CFG as this now

$$S \rightarrow M \; or \; U$$

$$M \rightarrow "if \; E \; then \; M \; else \; M"$$
$$or \; "if \; E \; then \; U \; else \; M"$$
$$or \; "if \; E \; then \; M \; else \; U"$$

$$U \rightarrow "if \; E \; then \; S"$$

$$E \rightarrow (an \; expression)$$
$$S \rightarrow (a \; statement)$$

This CFG is no longer ambiguous, as it now establishes a precedence.

The first if that has not yet a matching else will now receive the else clause.

```
if a then
  if b then
    statement1
  else
    statement2
```

```
if a then
  if b then
    statement1
else
  statement2
```

A second disambiguation example

But wait, is that even correct to have such a precedence rule?!

This CFG is no longer ambiguous, as it now establishes a precedence.

The first if that has not yet a matching else will now receive the else clause.

```
if a then
  if b then
    statement1
  else
    statement2
```

```
if a then
  if b then
    statement1
else
  statement2
```

A second disambiguation example

But wait, is that even correct to have such a precedence rule?!

No, it is not.

In fact, that is the reason why programming languages will use **additional delimiters** for such statements.

- In C, { and }.
- In Python, indentation.

These delimiters should appear in the CFG production rules!

This CFG is no longer ambiguous, as it now establishes a precedence.

The first if that has not yet a matching else will now receive the else clause.

```
if a then
  if b then
    statement1
  else
    statement2
```

```
if a then
  if b then
    statement1
else
  statement2
```

A CFG for if-else with delimiters

A “simplified” (sigh…) CFG for if/else statements in C, with delimiters is:

$$S \rightarrow \ ifstmt$$
$$ifstmt \rightarrow \ \text{"if"}\ \text{"("}\ cond\ \text{")"}\ block$$
$$ifstmt \rightarrow \ \text{"if"}\ \text{"("}\ cond\ \text{")"}\ block\ \text{"else"}\ block$$
$$block \rightarrow \ \text{"\{"}\ stmtlist\ \text{"\}"}$$
$$block \rightarrow \ stmt$$
$$stmttlist \rightarrow \ stmt$$
$$stmtlist \rightarrow \ stmtlist\ stmt$$
$$stmt \ -> \ expr\ \text{";"}$$
$$stmt \ -> \ ifstmt$$
$$expr \ -> \ (any\ valid\ expression\ in\ C)$$
$$condition \rightarrow \ (any\ valid\ condition\ in\ C)$$

A CFG for if-else with delimiters

This “simplified” CFG:

- Lifts ambiguity by introducing delimiters (the { and } symbols).
- Reuses also the CFG for checking what a valid statement in C might be,
- Would reuse the CFG checking what a valid condition/Boolean could look like (which is what?)

Eventually, we have something similar for while, for, etc.

$$S \rightarrow ifstmt$$
$$ifstmt \rightarrow \text{"if" "("} cond \text{")"} block$$
$$ifstmt \rightarrow \text{"if" "("} cond \text{")"} block \text{"else"} block$$
$$block \rightarrow \text{"\{"} stmtlist \text{"\}"}$$
$$block \rightarrow stmt$$
$$stmttlist \rightarrow stmt$$
$$stmtlist \rightarrow stmtlist\ stmt$$
$$stmt \rightarrow expr \text{";"}$$
$$stmt \rightarrow ifstmt$$
$$expr \rightarrow (any\ valid\ expression\ in\ C)$$
$$condition \rightarrow (any\ valid\ condition\ in\ C)$$

Associativity rules

Definition (Associativity):

Left-associativity and **right-associativity** refer to the order in which operations of the same precedence level are evaluated in an expression.

These concepts are important when designing CFGs for programming languages (or other formal languages), as they determine how expressions involving multiple occurrences of the same operator should be parsed and evaluated.

This can be reflected in the way to write CFGs, so that **derivations will generate parse trees with the correct sequence of operations**.

Associativity rules

Definition (Associativity):

An operator is **left-associative** if operations are **evaluated from left to right**. When multiple occurrences of operators with same precedence appear in an expression, the leftmost operators are evaluated first.

This is the default associativity for most arithmetic operators in programming languages, such as addition.

For instance, in the expression 3 - 2 + 1, the subtraction operation would be evaluated first because of left-associativity, which would be equivalent to (3 - 2) + 1.

Associativity rules

Definition (Right-associativity):

An operator is **right-associative**, if operations are **evaluated from right to left**. When multiple occurrences of operators with same precedence appear in an expression, the rightmost operators are evaluated first.

This is typically the associativity used for exponentiation or assignment operators in programming languages.

For instance, in the expression 2**3**2 (in Python, because C does not have an exponentiation operator?!), the second exponentiation operation would be evaluated first because of right-associativity. This would be equivalent to 2**(3**2).

A left-associative example: addition

Addition should be left-associative (could also be right-associative, but would prefer if it was not).

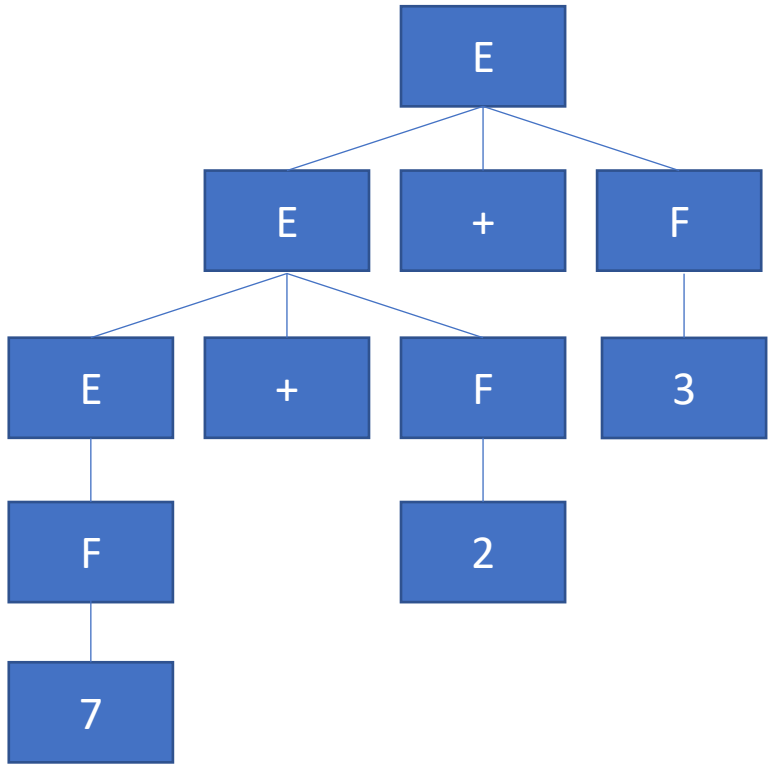
- 7+2+3 should be computed as (7+2)+3.
- Can be implemented with

$$E \rightarrow E + F$$

$$E \rightarrow F$$

$$F \rightarrow (a\ number)$$

Using E+F instead of F+E makes derivations and parsing trees implement left-associativity.



A left-associative example: addition

Exponentiation should be right-associative, not left-associative.

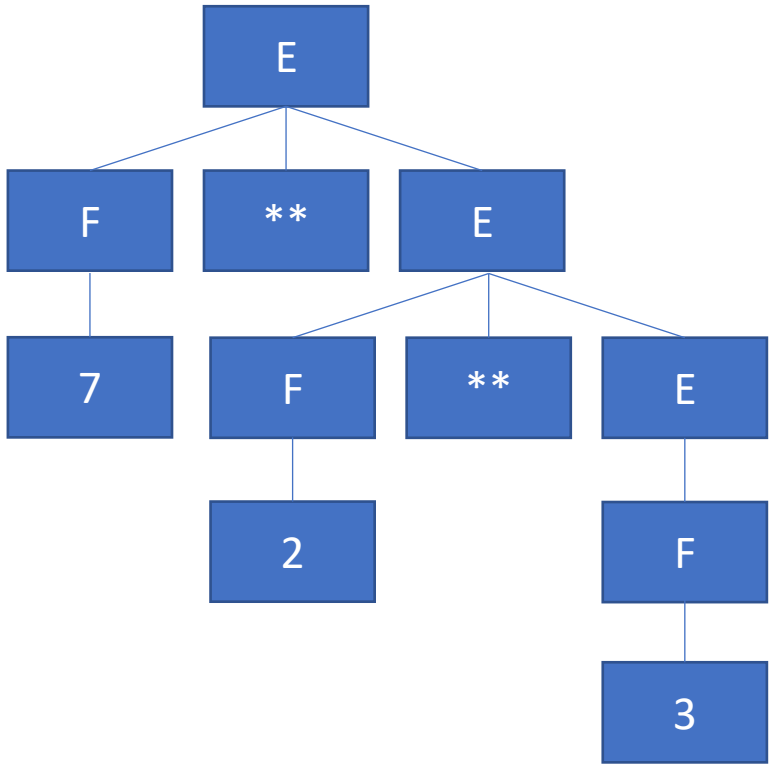
- 7**2**3 should be computed as 7**(2**3).
- Can be implemented with

E → F ** E

E → F

F → (a number)

Using F**E instead of E**F makes derivations and parsing trees implement right-associativity.



Quiz time!

Consider the operation below, in Python. Which one are Left-Associative? Right-Associative? Which ones could be Both? If both, which is preferable?

A. Subtraction, e.g. 7 – 2 – 3 .
B. Multiplication, e.g. 2 * 5 * 3.
C. Division, e.g. 8 / 4 / 2.
D. Assignment, e.g. x = y = z = 7.
E. Boolean or, e.g. x or y or z.

Quiz time!

Consider the operation below, in Python. Which one are Left-Associative? Right-Associative? Which ones could be Both? If both, which is preferable?

A. **Subtraction, e.g. 7 – 2 – 3 .**
B. Multiplication, e.g. 2 * 5 * 3.
C. Division, e.g. 8 / 4 / 2.
D. Assignment, e.g. x = y = z = 7.
E. Boolean or, e.g. x or y or z.

Quiz time!

Consider the operation below, in Python. Which one are Left-Associative? Right-Associative? Which ones could be Both? If both, which is preferable?

A. **Subtraction, e.g. 7 – 2 – 3 .**
B. **Multiplication, e.g. 2 * 5 * 3. (left)**
C. Division, e.g. 8 / 4 / 2.
D. Assignment, e.g. x = y = z = 7.
E. Boolean or, e.g. x or y or z.

Quiz time!

Consider the operation below, in Python. Which one are Left-Associative? Right-Associative? Which ones could be Both? If both, which is preferable?

A. **Subtraction, e.g. 7 – 2 – 3 .**
B. **Multiplication, e.g. 2 * 5 * 3. (left)**
C. **Division, e.g. 8 / 4 / 2.**
D. Assignment, e.g. x = y = z = 7.
E. Boolean or, e.g. x or y or z.

Quiz time!

Consider the operation below, in Python. Which one are Left-Associative? Right-Associative? Which ones could be Both? If both, which is preferable?

A. **Subtraction, e.g. 7 – 2 – 3 .**
B. **Multiplication, e.g. 2 * 5 * 3. (left)**
C. **Division, e.g. 8 / 4 / 2.**
D. **Assignment, e.g. x = y = z = 7.**
E. Boolean or, e.g. x or y or z.

Quiz time!

Consider the operation below, in Python. Which one are Left-Associative? Right-Associative? Which ones could be Both? If both, which is preferable?

A. **Subtraction, e.g. 7 – 2 – 3 .**
B. **Multiplication, e.g. 2 * 5 * 3. (left)**
C. **Division, e.g. 8 / 4 / 2.**
D. **Assignment, e.g. x = y = z = 7.**
E. **Boolean or, e.g. x or y or z. (left, for short circuit evaluation!)**