50.051 Programming Language Concepts

W10-S2 Context Free Grammars (CFG)

Matthieu De Mari

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

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 given 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 given mathematical expression is valid or not?

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

Checking that requires 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

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

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

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 with a RegEx that runs on **finite** state machines?

Let us start with a problem

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

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

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 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. $\mathrm{H_2O}$

C. $\mathrm{H^2O}$

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

E. $\mathrm{C_2\ J_8}$

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

A chemical problem

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

- **CO (carbon monoxyde)**
- **H_2O (water)**
- **H^2O (exponents are not allowed, except for ions)**
- **$C_{12}H_{22}O_{11}$ (sugar)**
- **$C_2\ J_8$ (no J element in periodic table)**

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

A chemical problem

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

- **CO (carbon monoxyde)**
- **H_2O (water)**
- **H^2O (exponents are not allowed, except for ions)**
- **$C_{12}H_{22}O_{11}$ (sugar)**
- **$C_2\ J_8$ (no J element in periodic table)**

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

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

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 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):

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

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

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

These can be **further decomposed 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.

$$Formula \to Molecule$$

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

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

Back to our Chemistry

Consider the CFG below.

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

$$Formula \rightarrow Molecule$$

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

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

Back to our Chemistry

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

Consider the CFG below.

$$Formula \rightarrow Molecule$$

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

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

Back to our Chemistry

Consider the CFG below.

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

$$Formula \rightarrow Molecule$$

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

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

Back to our Chemistry

Consider the CFG 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

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

Is the formula CO valid? Yes.

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

$Form \rightarrow Mol$ (rule 1)

$Mol \rightarrow MolElem$ (rule 4)

$MolEle \rightarrow ElemElem$ (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

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

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

- $\mathrm{H_2O}$ is a valid expression,
- $\mathrm{C_{12}H_{22}O_{11}}$ is a valid expression,
- $\mathrm{H^2O}$ is not a valid expression,
- And $\mathrm{C_2\ J_8}$ is not 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

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

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

- H_2O is a valid expression,
- $C_{12}H_{22}O_{11}$ is a valid expression,
- H^2O is not a valid expression,
- And $C_2\ J_8$ is not a valid expression?

Answer 1: To be shown on board.

Practice 1 and 2

Consider the CFG from earlier

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

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

Restricted

Practice 1 and 2

Consider the CFG from earlier

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

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

Answer 2: Probably something like

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

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

CFG would reject $Ca_3PO_4)_2$.

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.

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

Derivation and Syntax Validity

Theorem (Syntax Validity):

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

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?

Let us assume we have used our tokenizer on a given source code and we have obtained a tokens stream of some sort.

Which of the two tokens streams below shows that the code has a syntax problem of some sort?

A. Token(KEYWORD_INT, “int”), Token(IDENTIFIER, “x”), Token(EQSIGN, “=”), Token(INT_LITERAL, “1023”), Token(SEMICOL, “;”).

B. Token(KEYWORD_INT, “int”), Token(INT_LITERAL, “1023”), Token(EQSIGN, “=”), Token(IDENTIFIER, “x”), Token(SEMICOL, “;”).

How is that useful for compilers?

Let us assume we have used our tokenizer on a given source code and we have obtained a tokens stream of some sort.

Which of the two tokens streams below shows that the code has a syntax problem of some sort?

A. Token(KEYWORD_INT, “int”), Token(IDENTIFIER, “x”), Token(EQSIGN, “=”), Token(INT_LITERAL, “1023”), Token(SEMICOL, “;”).

B. Token(KEYWORD_INT, “int”), Token(INT_LITERAL, “1023”), Token(EQSIGN, “=”), Token(IDENTIFIER, “x”), Token(SEMICOL, “;”).

Equivalent question: Can you write "int 1023 = x;" in C?

How is that useful for compilers?

Let us assume we have used our tokenizer on a given source code and we have obtained a tokens stream of some sort.

Which of the two tokens streams below shows that the code has a syntax problem of some sort?

A. Token(KEYWORD_INT, “int”), Token(IDENTIFIER, “x”), Token(EQSIGN, “=”), Token(INT_LITERAL, “1023”), Token(SEMICOL, “;”).

B. Token(KEYWORD_INT, “int”), Token(INT_LITERAL, “1023”), Token(EQSIGN, “=”), Token(IDENTIFIER, “x”), Token(SEMICOL, “;”).

Equivalent question: Can you write "int 1023 = x;" in C?
No, this statement is incorrect because it does not follow the proper syntax for declaring and initializing a variable in C, which is "int x = 1023;" .

How is that useful for compilers?

Let us assume we have used our tokenizer on a given source code and we have obtained a tokens stream of some sort.

Which of the two tokens streams below shows that the code has a syntax problem of some sort?

A. **Token(KEYWORD_INT, “int”), Token(IDENTIFIER, “x”), Token(EQSIGN, “=”), Token(INT_LITERAL, “1023”), Token(SEMICOL, “;”).**

This is fine.

B. **Token(KEYWORD_INT, “int”), Token(INT_LITERAL, “1023”), Token(EQSIGN, “=”), Token(IDENTIFIER, “x”), Token(SEMICOL, “;”).**

This one has a syntax problem (identifier appears on right hand side of the equal sign and literal value on the left hand side).

How is that useful for compilers?

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

$$Declar \rightarrow Type\ TOKENID\ TOKENEQ\ Literal\ TOKENSEMICOL$$

Several possible keywords for integer variables, to decide on number of bits

$$Type \rightarrow TOKENINT\ or\ TOKENSHORT\ or\ TOKENLONG$$

Could technically have decimal and exponential notations for int literals

$$Literal \rightarrow TOKENINTLITERALDEC\ or\ TOKENINTLITERALEXP$$

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

$$Expr \rightarrow Term$$
$$Expr \rightarrow Expr + Term$$
$$Term \rightarrow Num$$
$$Term \rightarrow (Expr)$$
$$Num \rightarrow 0 \text{ } or \text{ } 1 \text{ } or \text{ } 2 \text{ } or \dots or \text{ } 9$$

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,

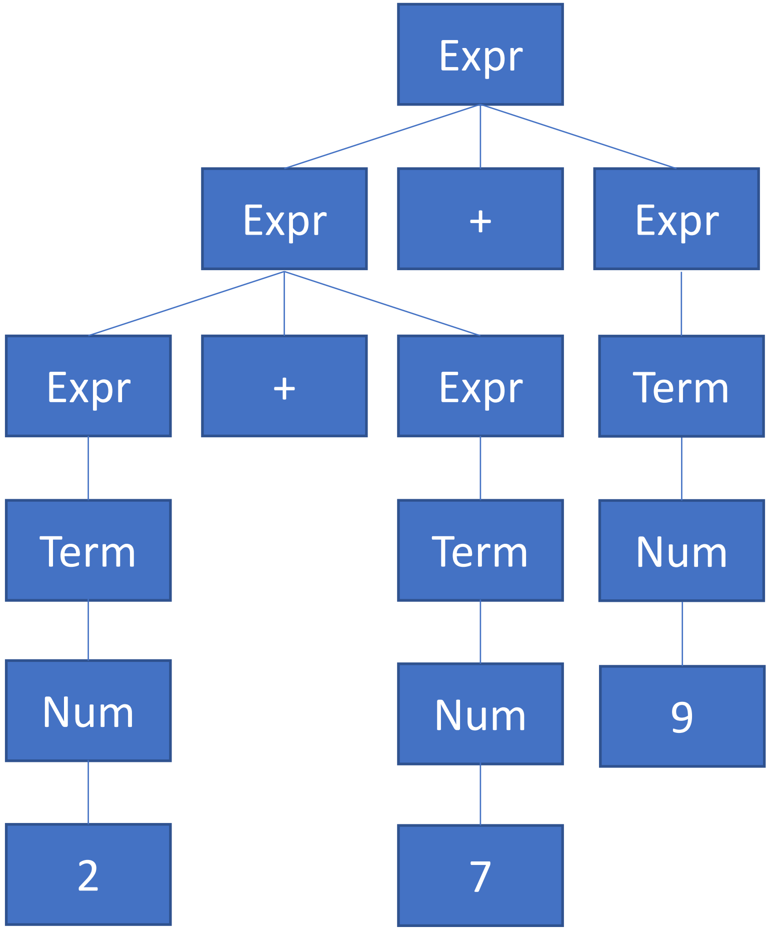
- Whose root is the start symbol,
- Where every production rule, $X \rightarrow Y_1 \dots Y_N$ in the derivation sequence, adds children nodes $Y_1, \dots, Y_N$ to the node X.

Parse tree of a derivation

Reusing the production rules below

$$Expr \to Term$$
$$Expr \to Expr + Expr$$
$$Term \to Num$$
$$Term \to (Expr)$$
$$Num \to 0 \; or \; 1 \; or \; 2 \; or \; \ldots \; or \; 9$$

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

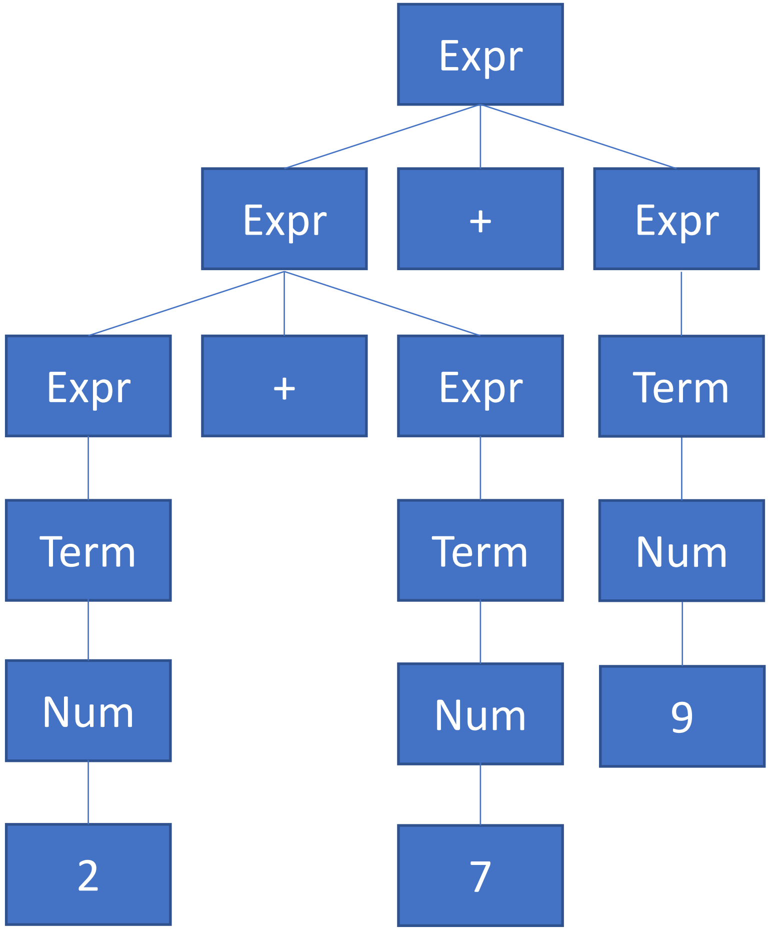


Parse tree of a derivation

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, $X \rightarrow Y_1 \dots Y_N$ in the derivation sequence, adds children nodes $Y_1, \dots, Y_N$ 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?**

Restricted

Quick question

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

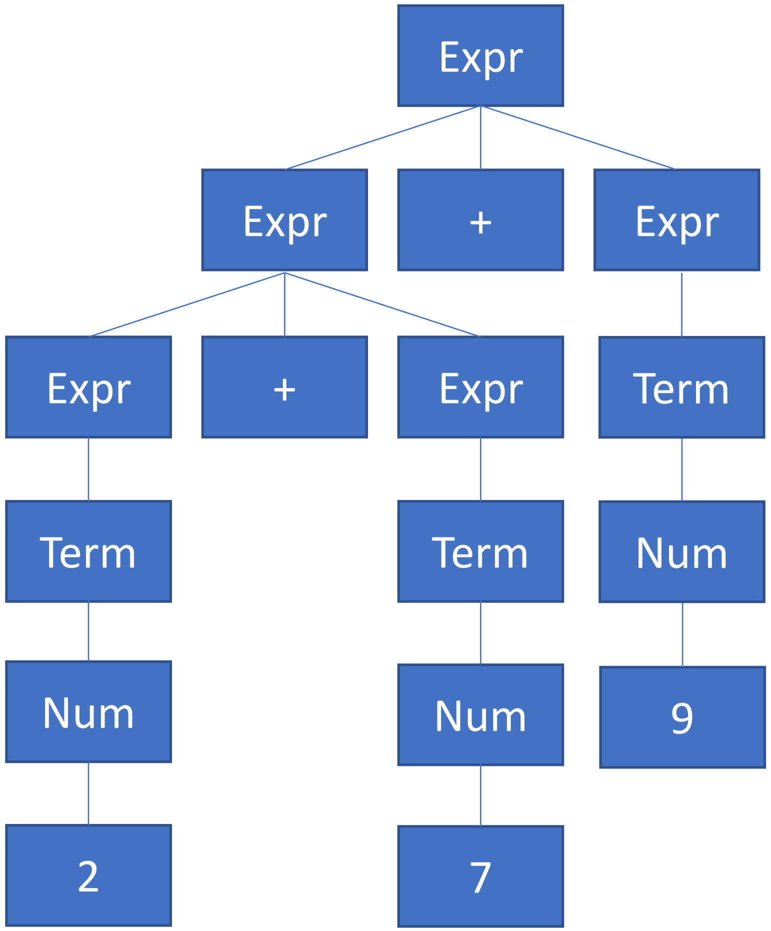
→ **Is the valid derivation unique?**

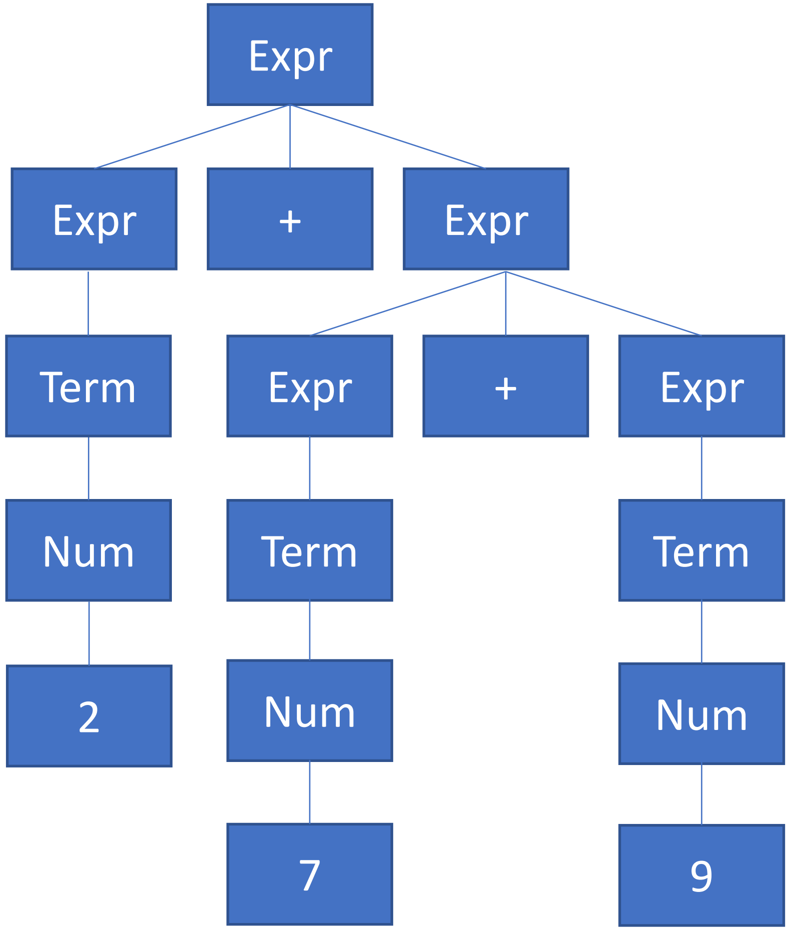
→ **Is there only one parse tree that could have been defined?**

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

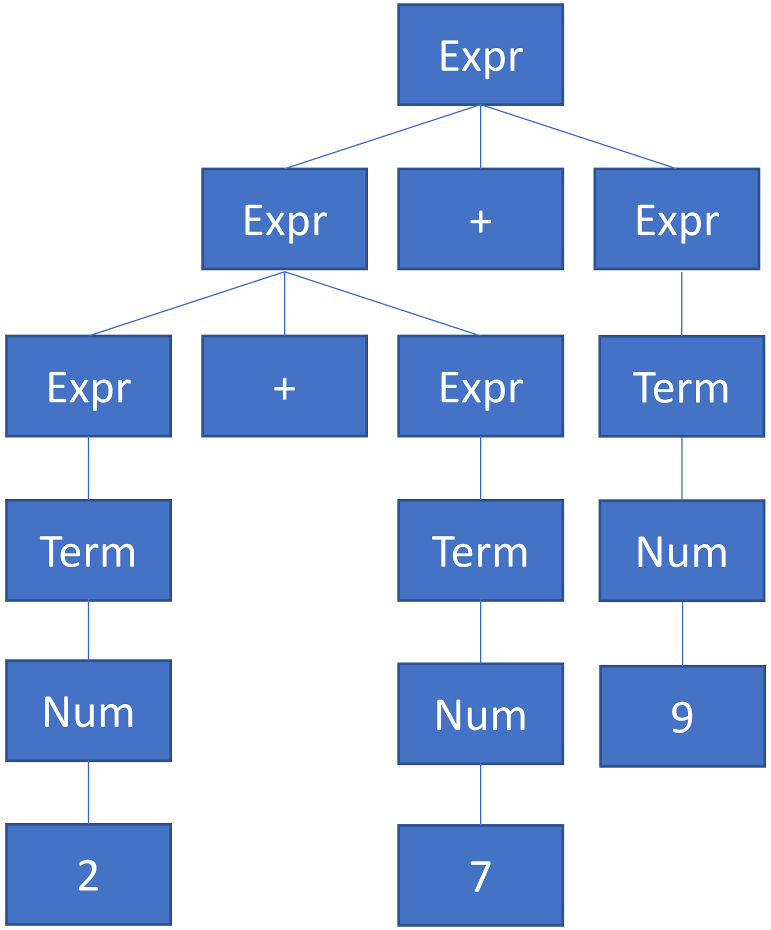
And that ambiguity might even be a problem in certain scenarios…

Two parse trees for 2+7+9

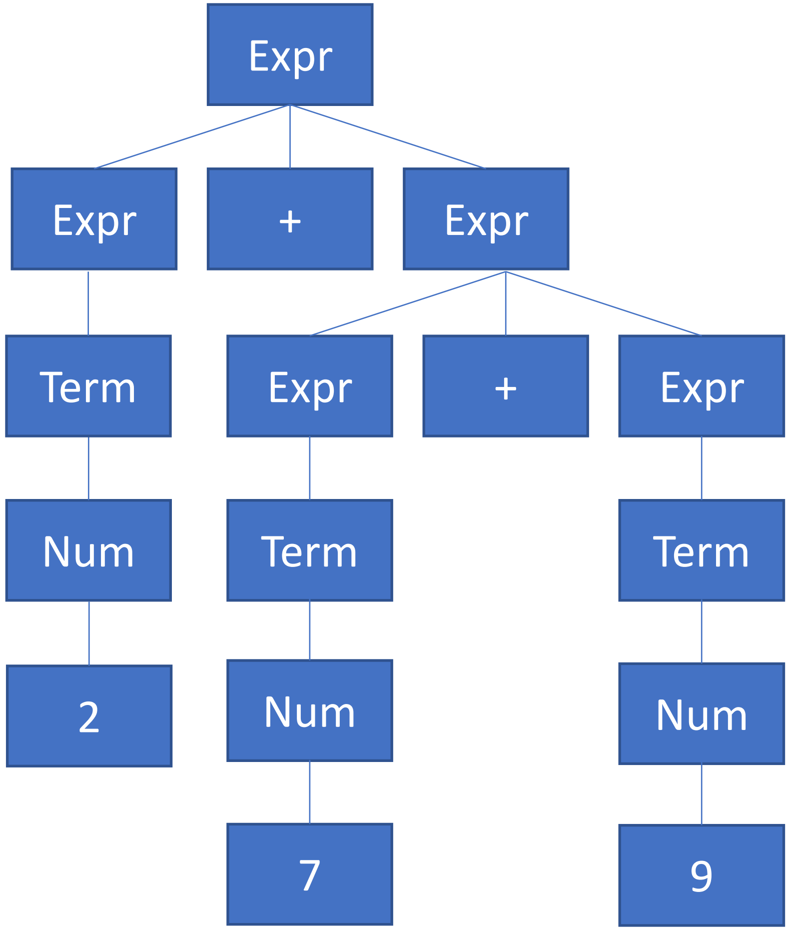




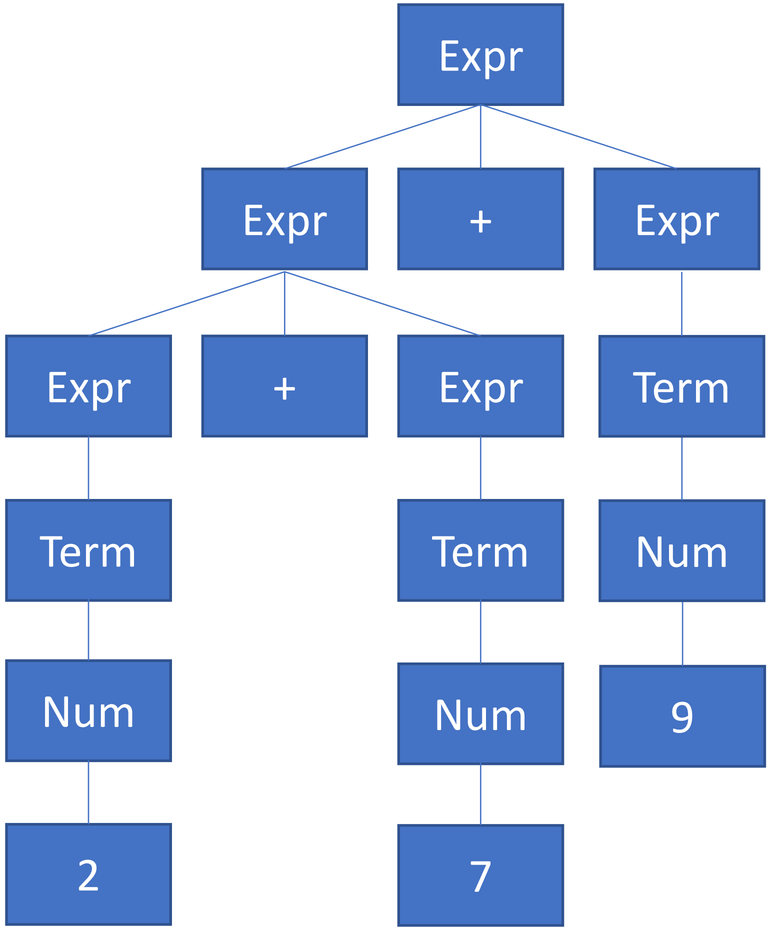
Two parse trees for 2+7+9



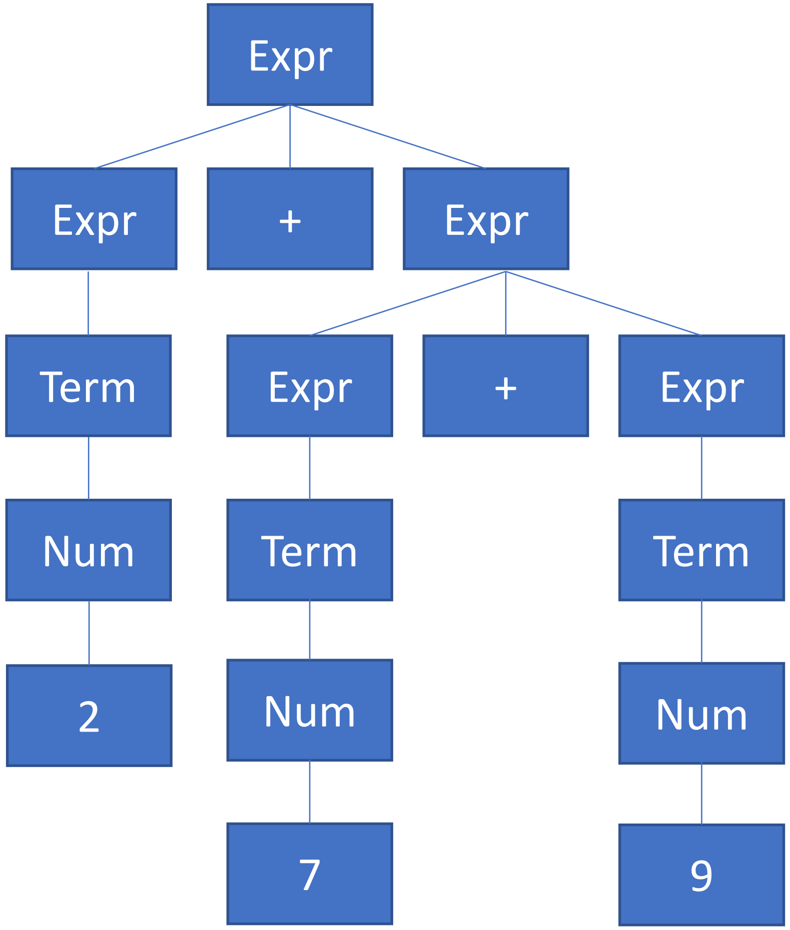
Question: Does that make a difference?



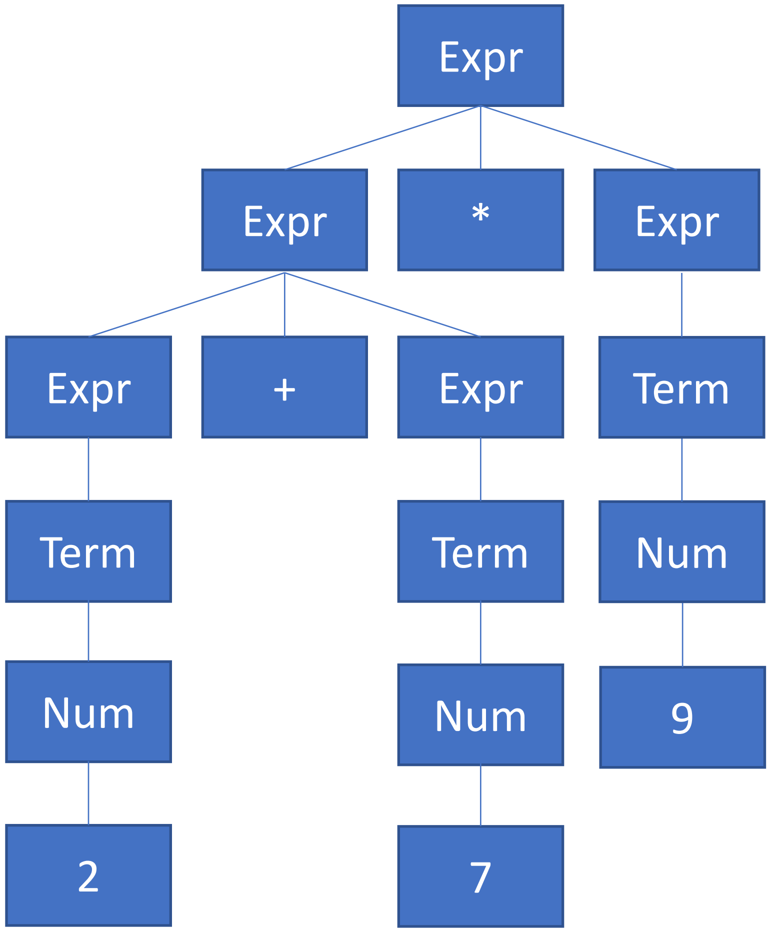
Two parse trees for 2+7+9



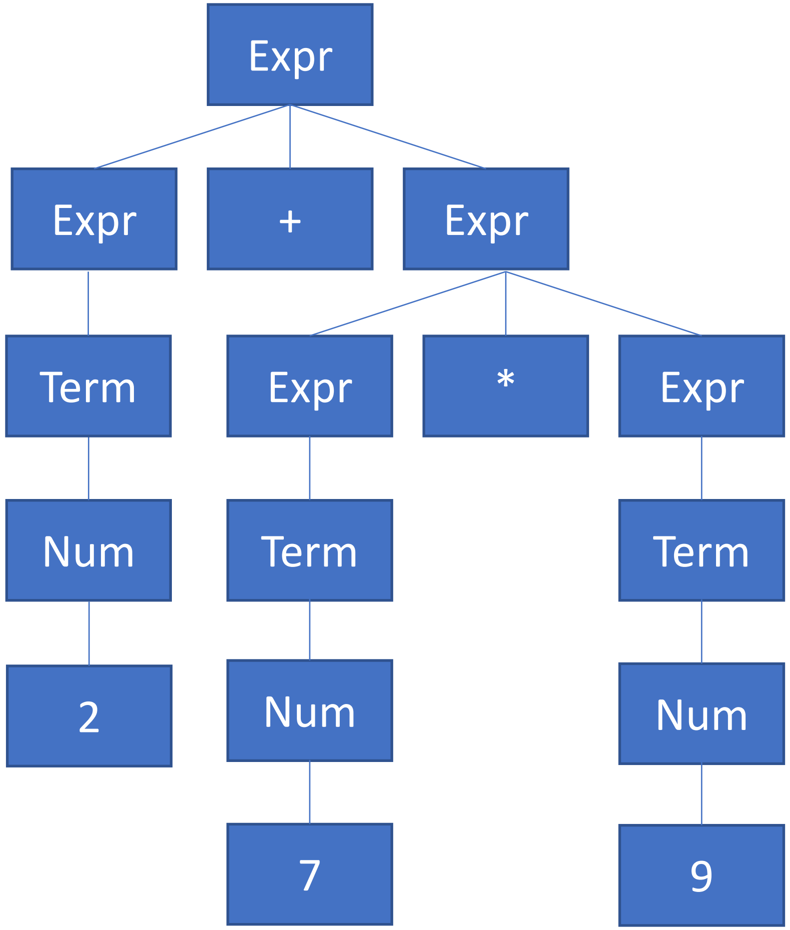
Question: Does that make a difference?
At the moment, no, because the order does not matter.



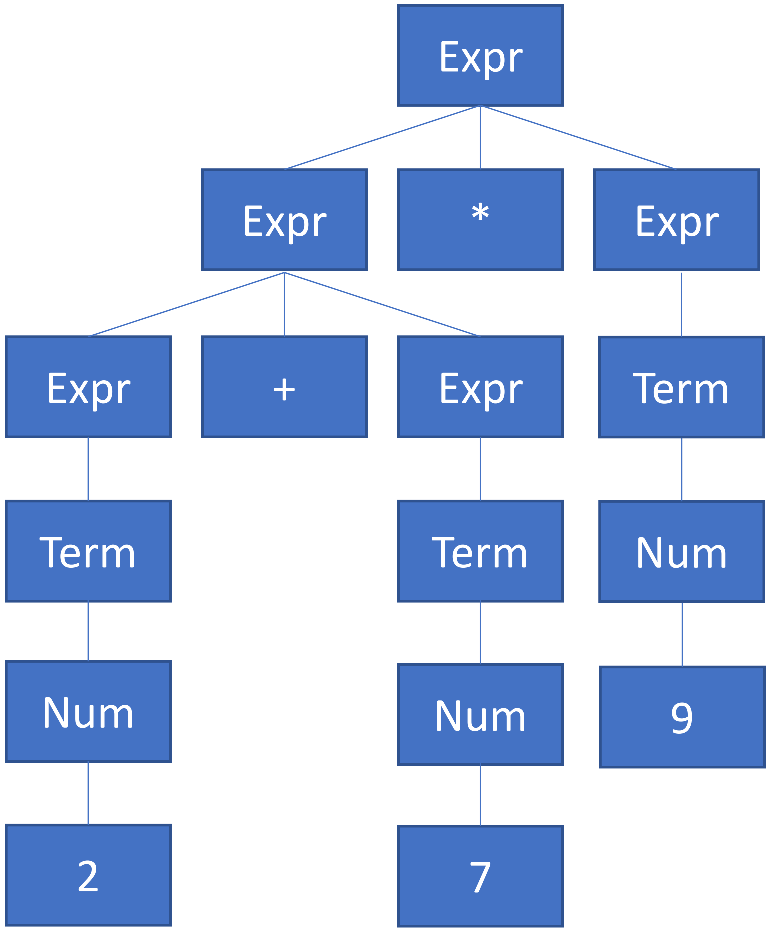
Two parse trees for 2+7+9



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

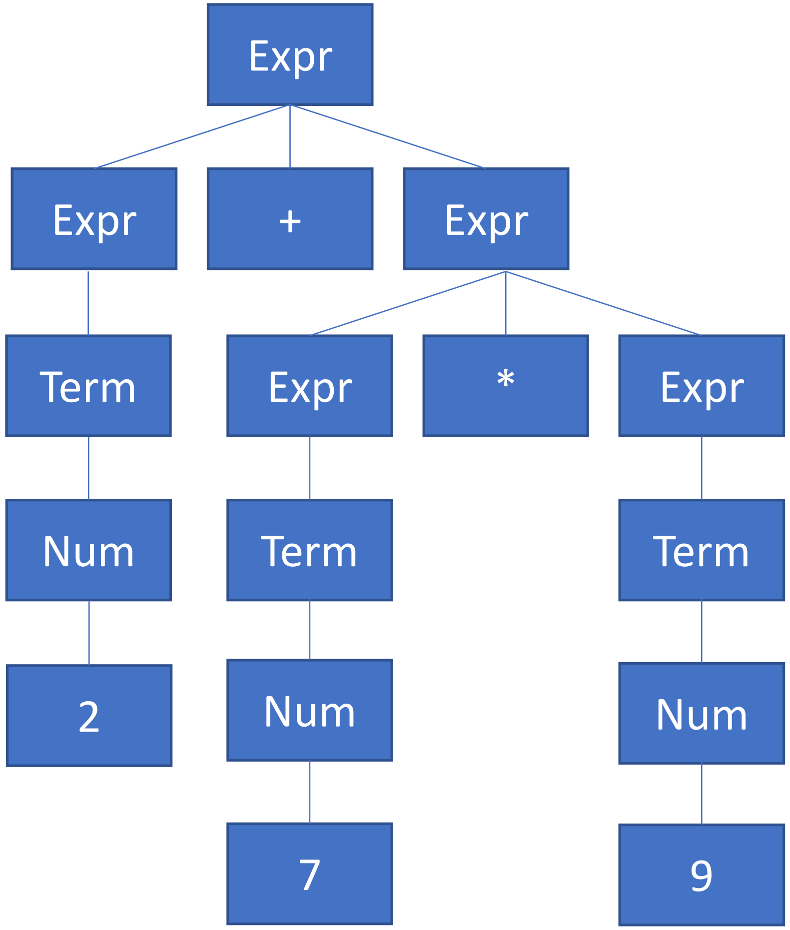


Two parse trees for 2+7+9



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 parse trees with different results.

In the case of arithmetic expressions and programming languages, this 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):

Let us consider a given CFG.

<u>There is no algorithm to check if a given CFG is ambiguous or not.</u>

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):

Let us consider a 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 is known as the Halting problem, and means that you cannot define a compiler program that can check for the presence of infinite loops in the compiled source code.)*

Checking ambiguity algorithmically

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

Let us consider a given CFG.

<u>There is no algorithm to check if a given CFG is ambiguous or not.</u>

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

There are however manual method for designing CFGs that will be non-ambiguous.

We will investigate them on the next lecture.

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