

Formal Grammar

Principles of Programming Languages

Lecture 13

Outline

Discuss briefly the **interpretation pipeline**, and how it will look in the context of this course

Introduce **formal grammars** as a mathematical framework for thinking about syntax and parsing

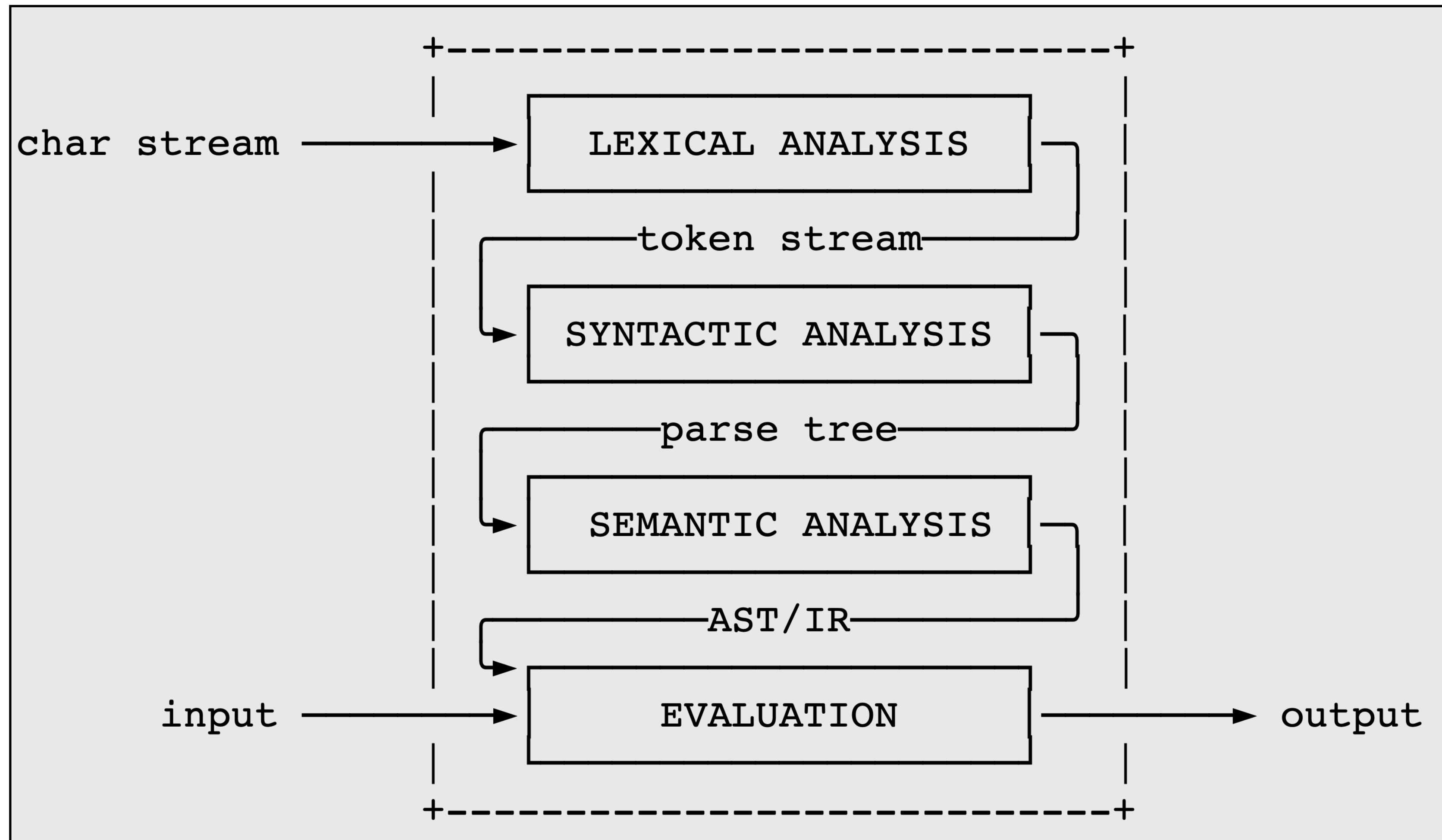
Look at what causes **ambiguity** in grammar

Learning Objectives

- Define sentential forms, production rules, BNF grammars, etc.
- Determine if the sentence S is recognized by the grammar \mathcal{G}
- Build a parse tree for S in the grammar \mathcal{G}
- Write a (leftmost) derivation for S in \mathcal{G}
- Define fixity, precedence, associativity, ambiguity, etc.
- Determine if the grammar \mathcal{G} is ambiguous
- Find a sentence in an ambiguous grammar with multiple parse trees/leftmost derivations
- Based on this grammar, determine if this operator left associative or right associative
- Based on this grammar, determine if which operator has higher precedence

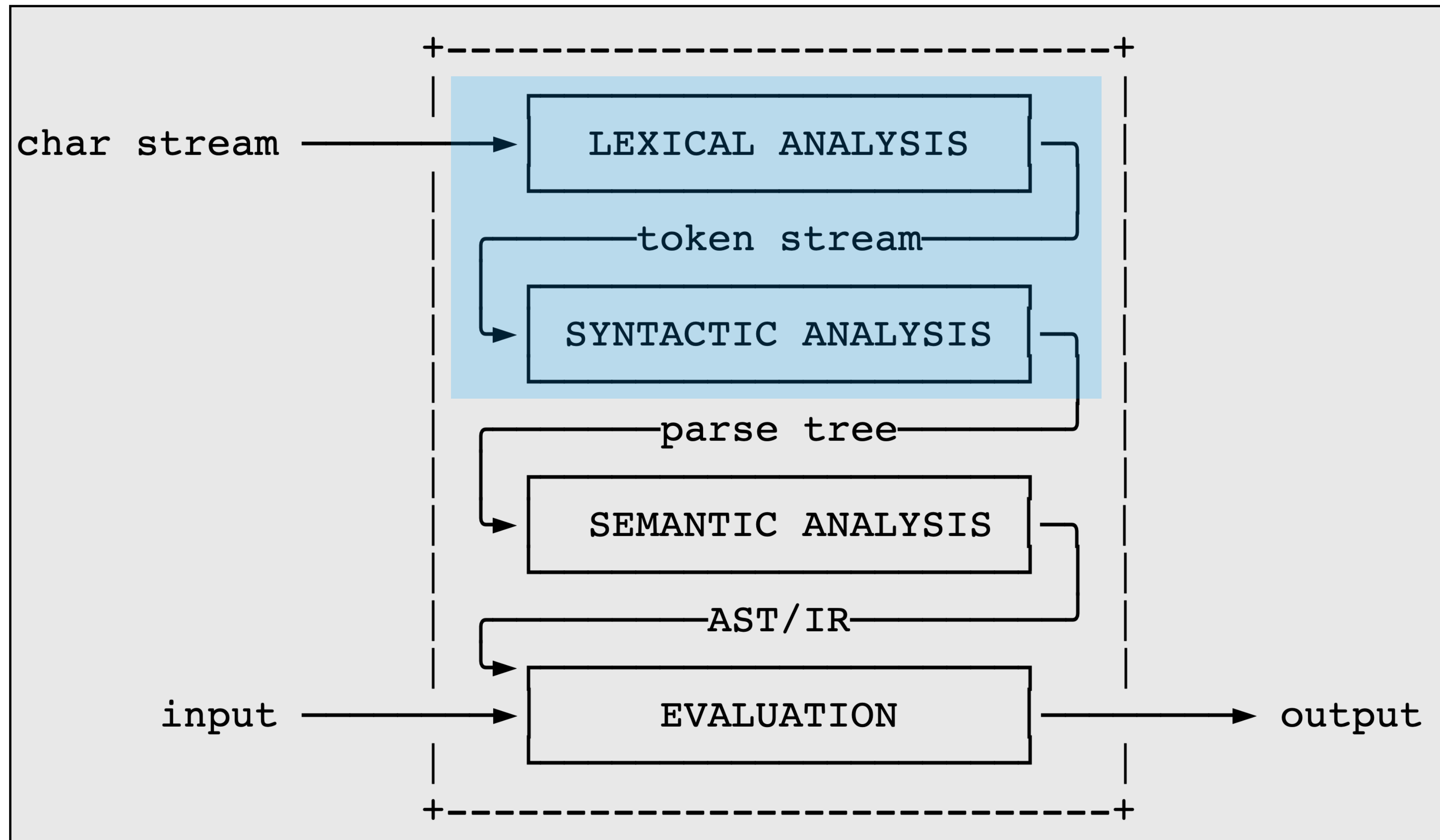
The Interpretation Pipeline

The Picture



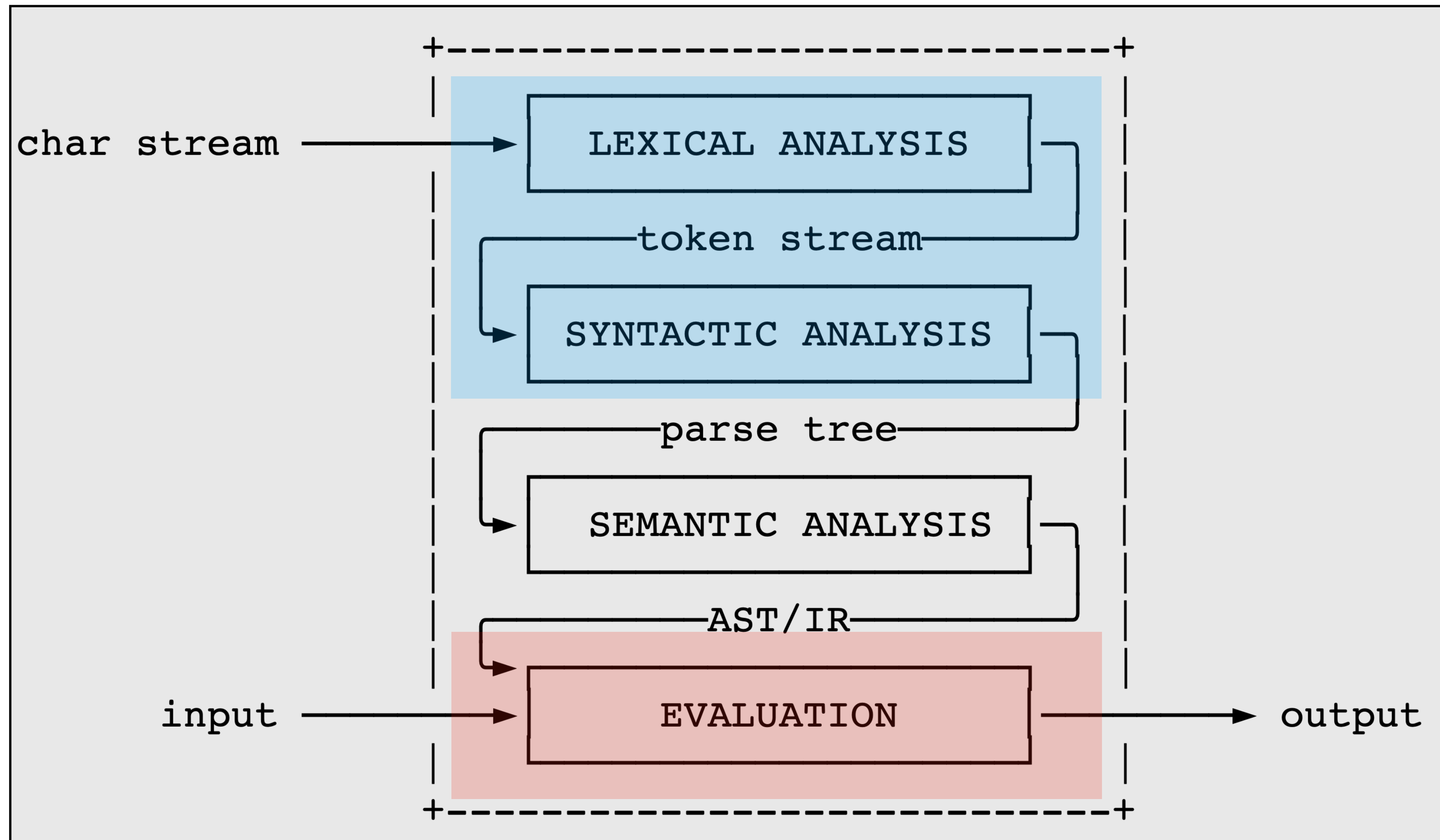
The Picture

parsing (this week)



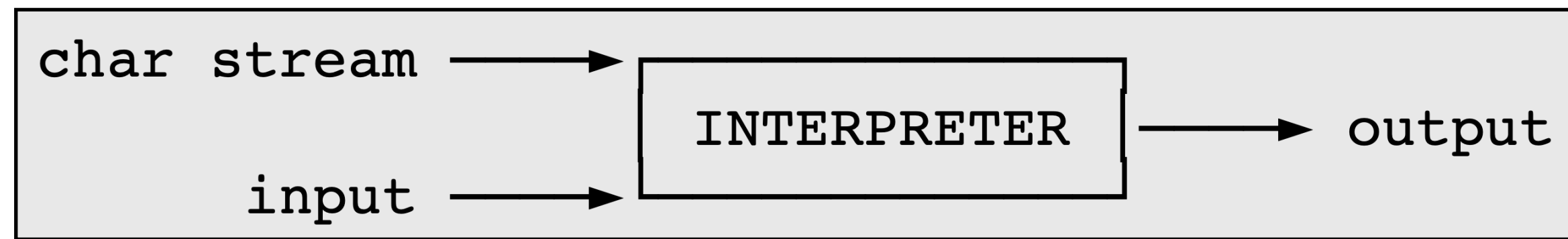
The Picture

parsing (this week)

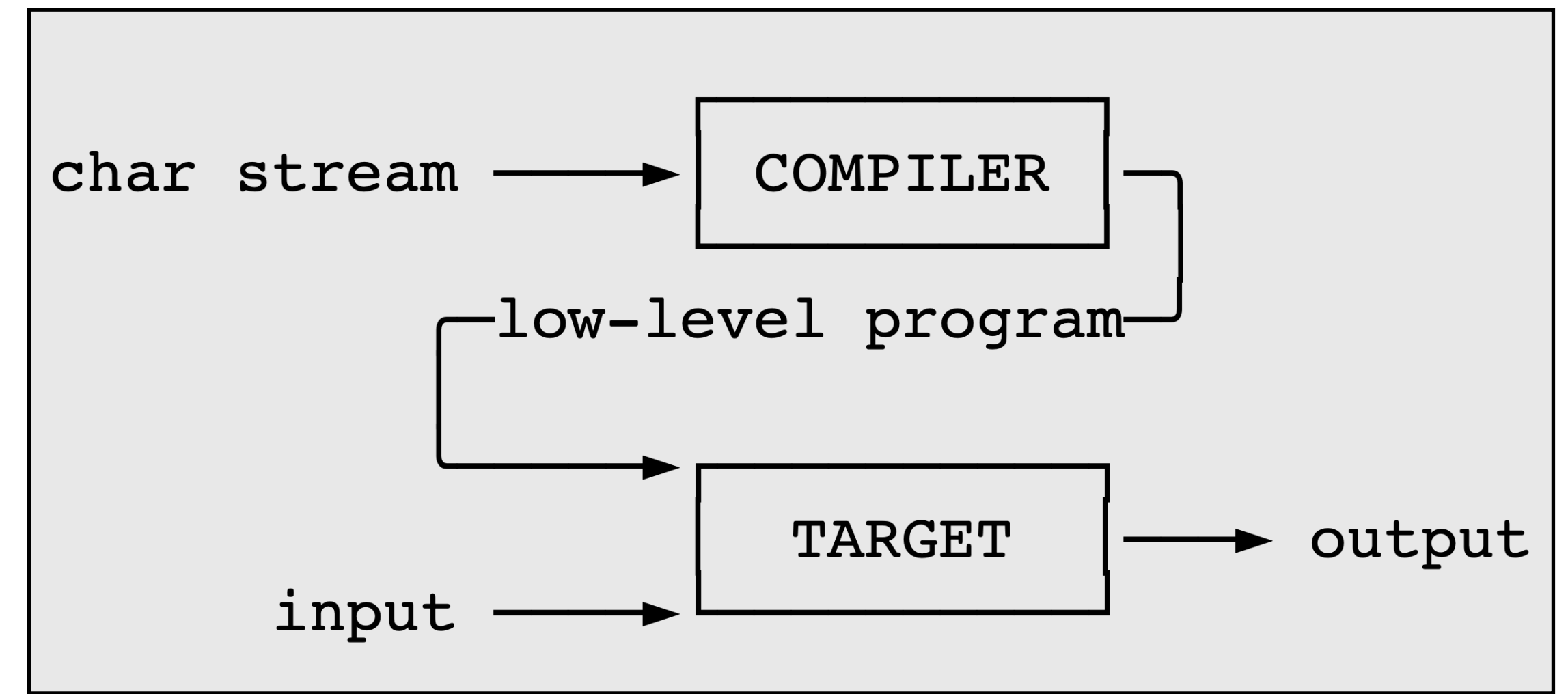


semantics (next week)

A Note on Compilation

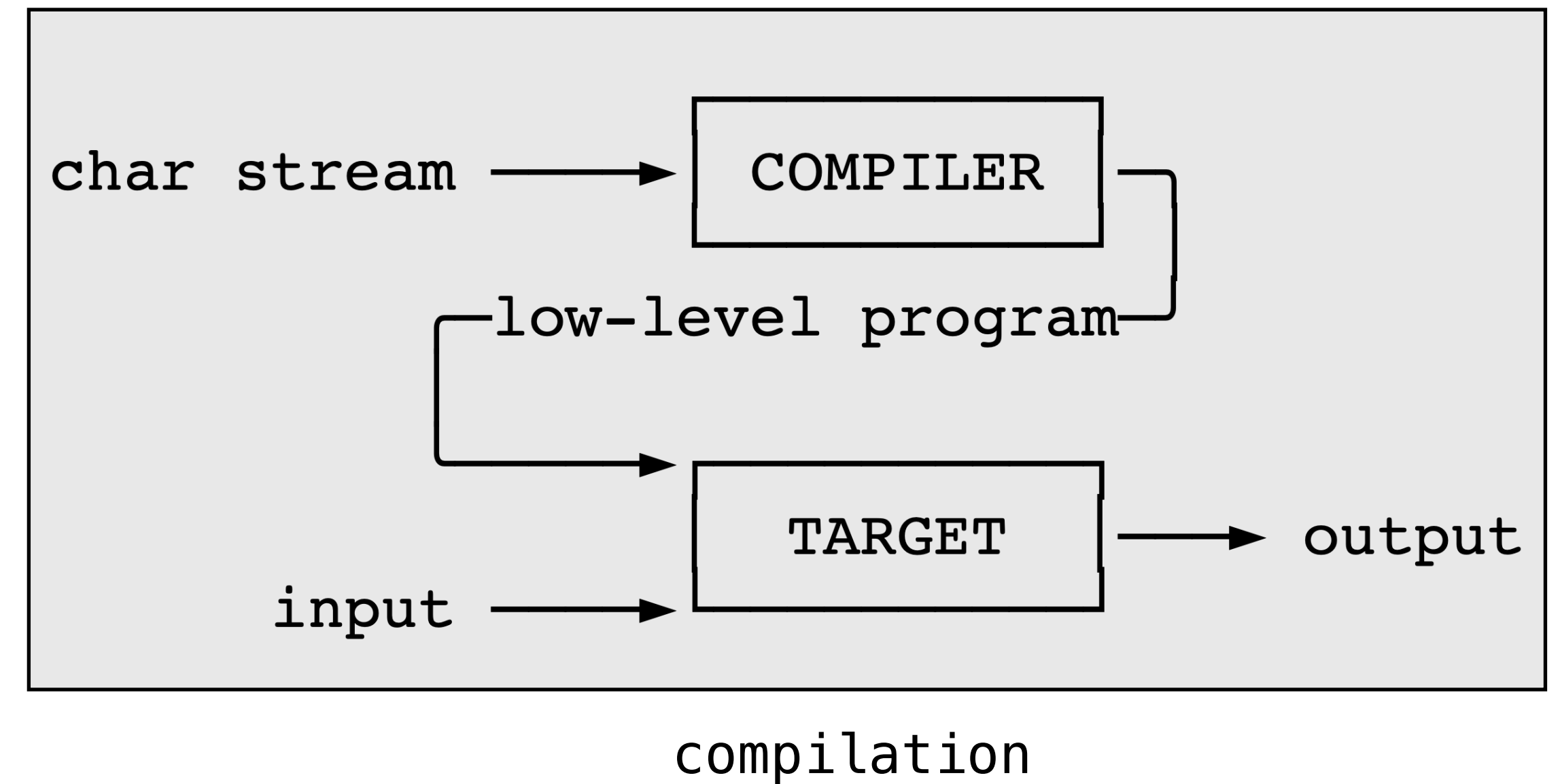
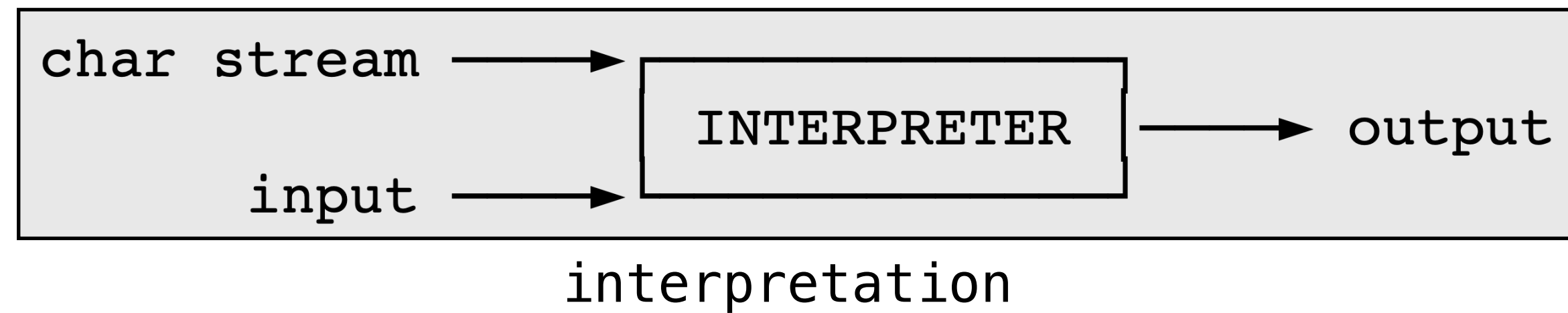


interpretation



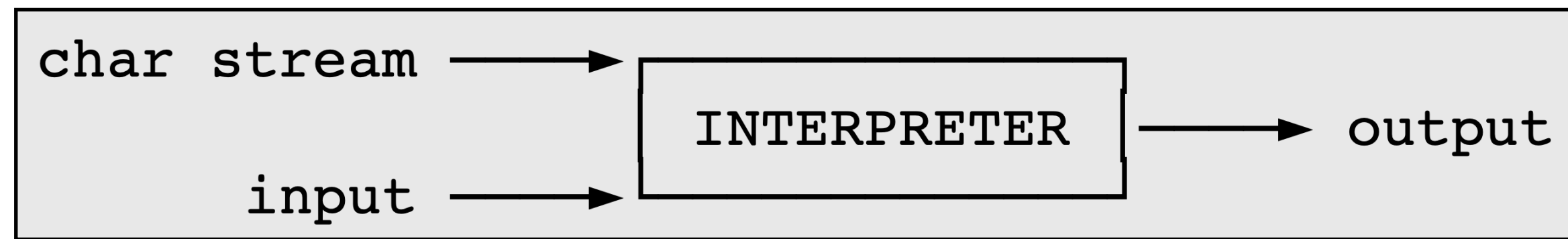
compilation

A Note on Compilation

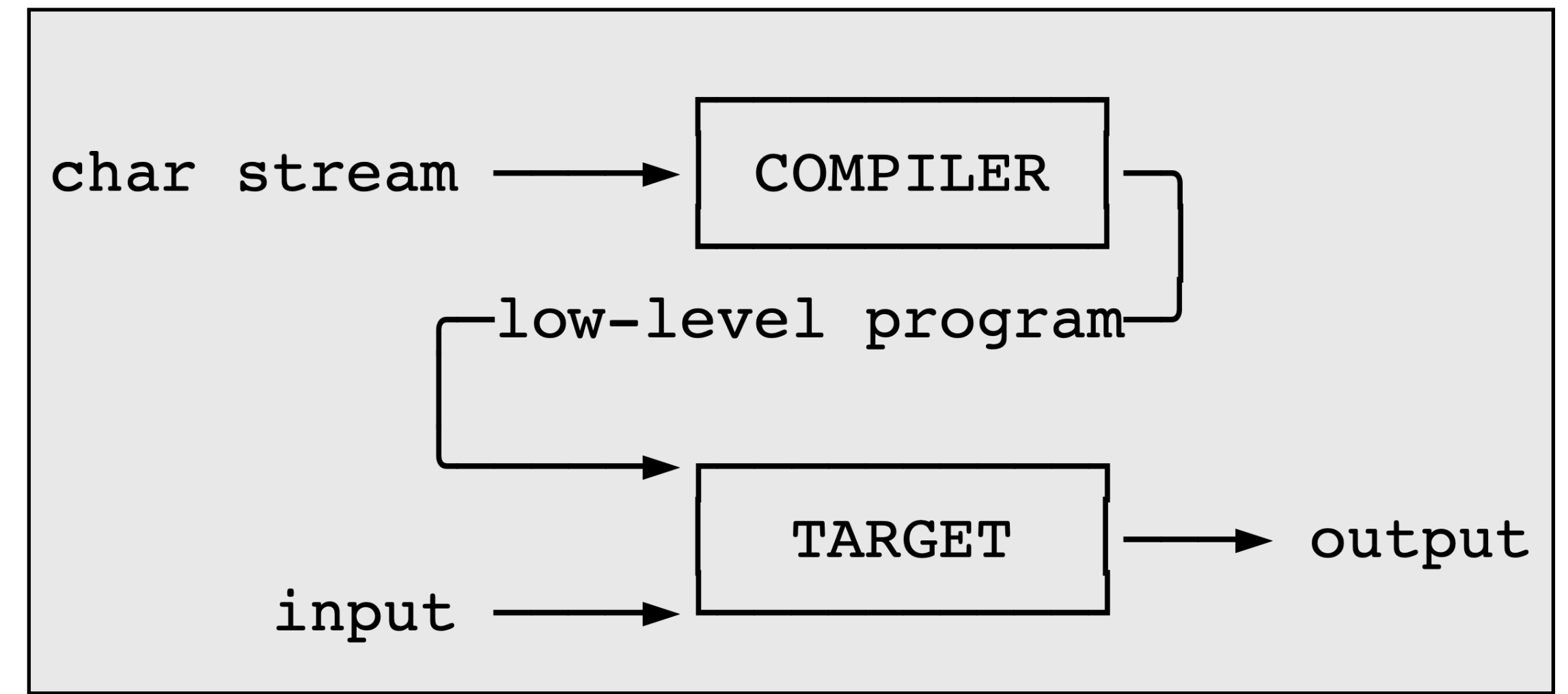


We will be building programs that directly read and evaluate programs (**interpreters**)

A Note on Compilation



interpretation



compilation

We will be building programs that directly read and evaluate programs (**interpreters**)

In a different course you may write a program which *translates* programs into another language which can then be evaluated elsewhere (**compilers**, we'll cover this briefly)

The Mini-Projects

There will be **three** mini-projects, each 2 weeks long.

For each project, you will build an interpreter.

You'll be given:

- » the syntax
- » the type rules (not in project 1)
- » the semantics

Today

We need a formal language for describing the syntax of programming languages

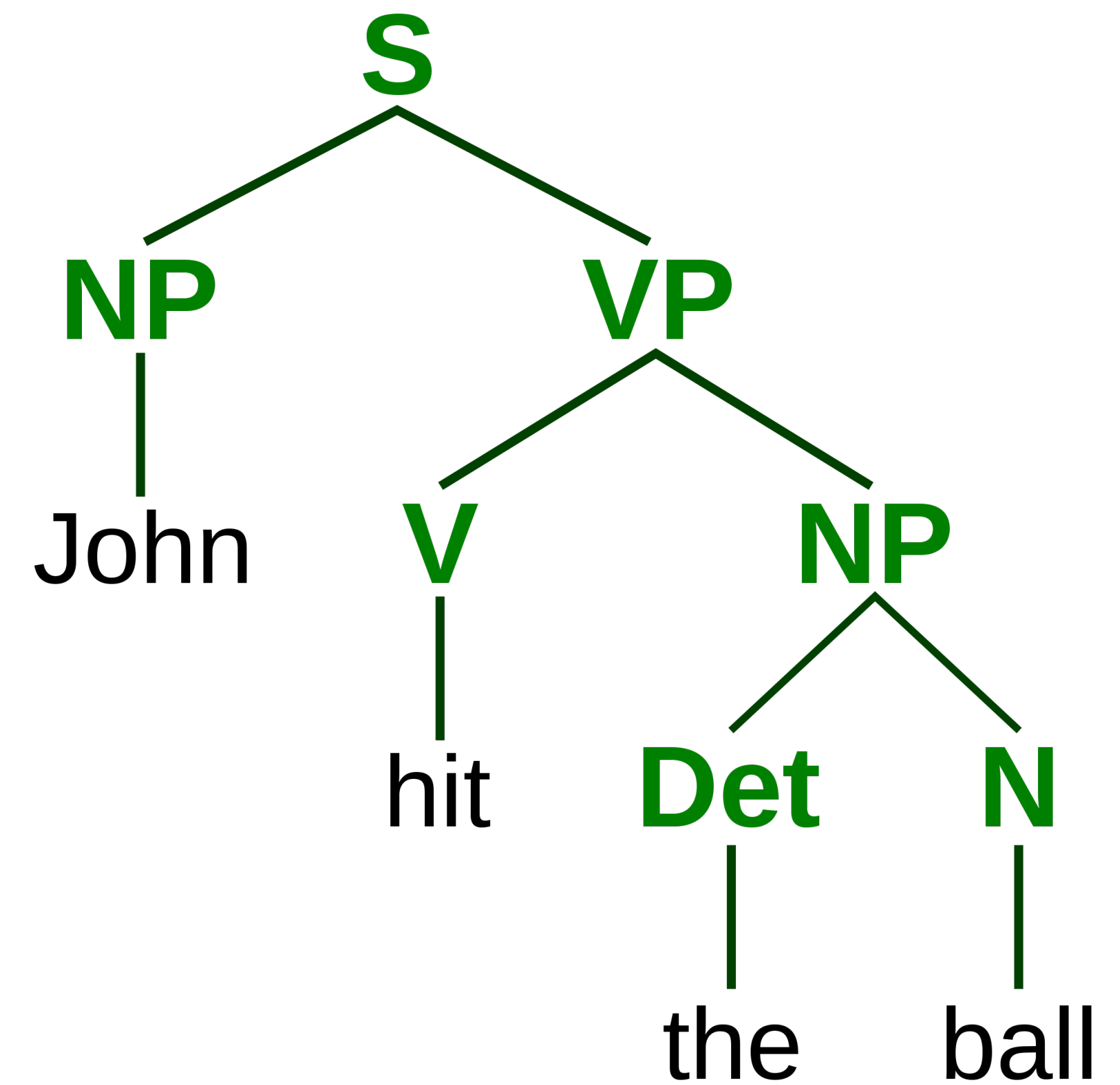
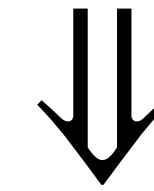
This is part of the study of **formal language theory**

Nearly every PL out there (including OCaml) is described using **Backus–Naur Form (BNF) Grammars**.

Formal Grammar

What is Grammar?

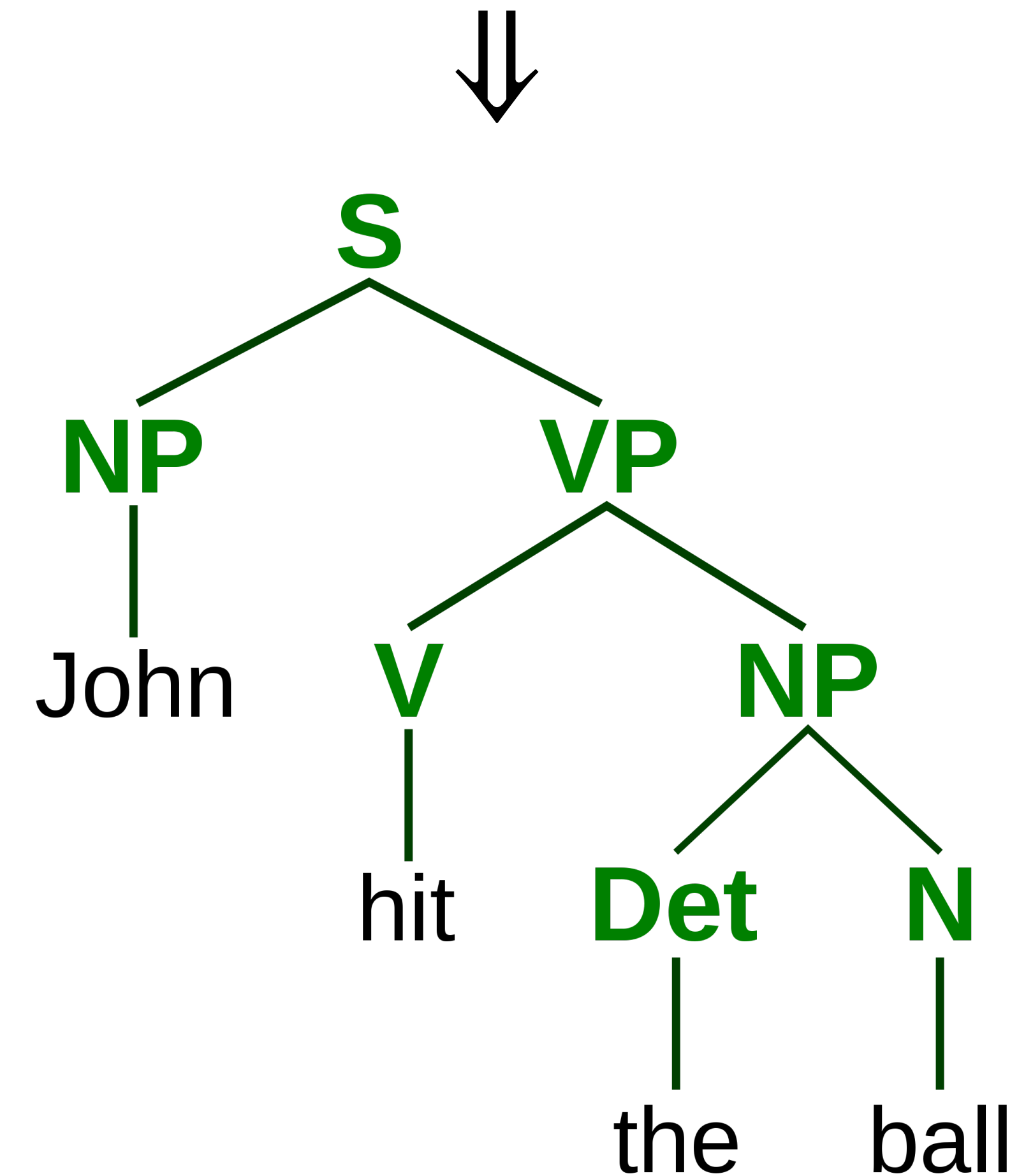
John hit the ball.



What is Grammar?

Grammar refers to the rules which govern what statements are well-formed.

John hit the ball.

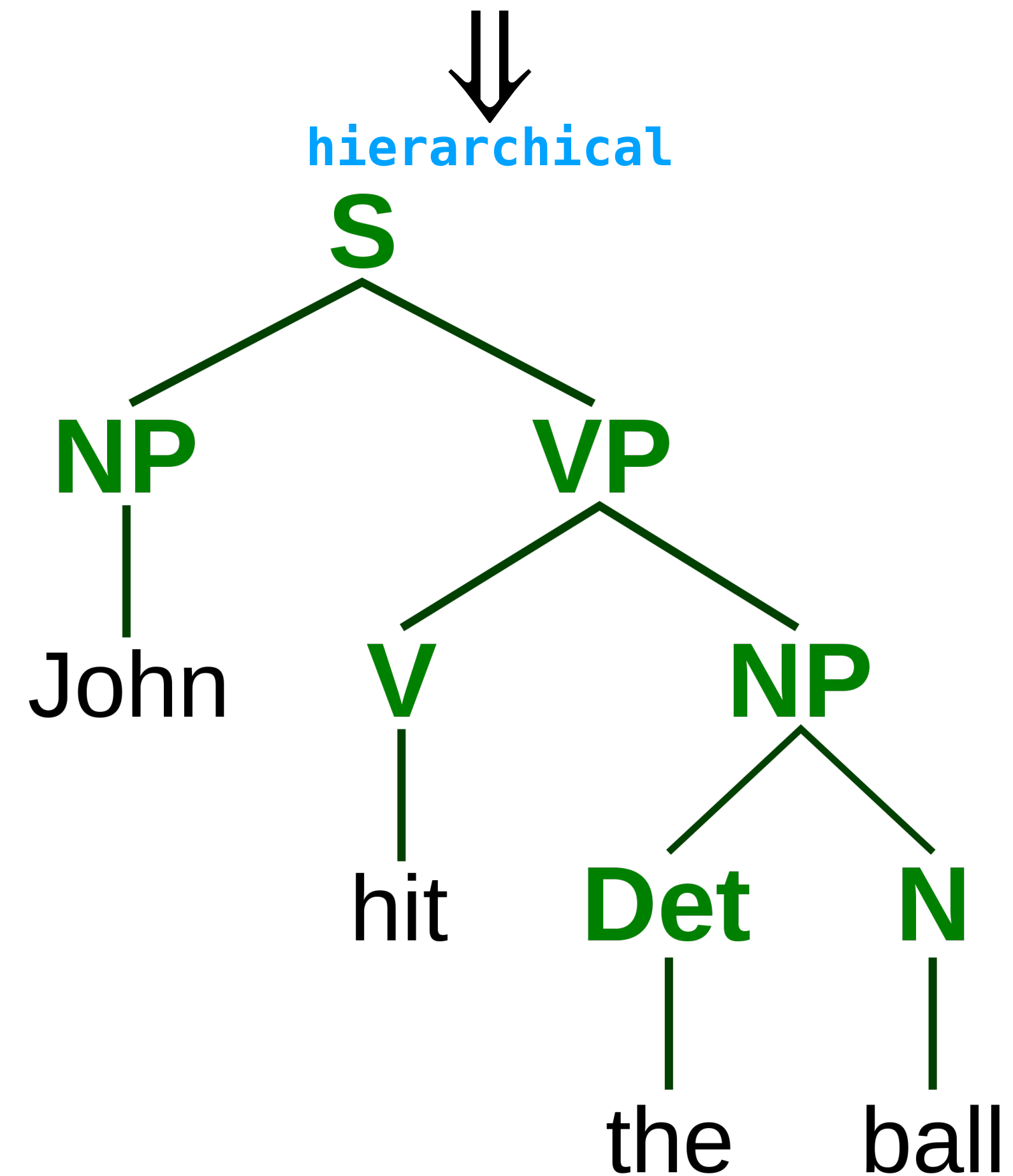


What is Grammar?

Grammar refers to the rules which govern what statements are well-formed.

Grammar gives **linear** statements (in natural language or code) their **hierarchical** structure.

John hit the ball.



Grammar vs. Semantics

I taught my car in the refrigerator. ✓

VS.

My the car taught I refrigerator. ✗

Grammar vs. Semantics

I taught my car in the refrigerator. ✓

VS.

My the car taught I refrigerator. ✗

Grammar is not (typically) interested in
meaning, just *structure*.

Grammar vs. Semantics

I taught my car in the refrigerator. ✓

VS.

My the car taught I refrigerator. ✗

Grammar is not (typically) interested in
meaning, just **structure**.

(As we will see, it is useful to separate these two concerns)

Grammars for Programming Languages

Grammars for Programming Languages

Formal grammars for PL
tell us which **programs** are
well-formed.

Grammars for Programming Languages

Formal grammars for PL
tell us which **programs** are
well-formed.

```
# let f x = x + 1;;  
val f : int -> int = <fun>
```

Grammars for Programming Languages

Formal grammars for PL
tell us which **programs** are
well-formed.

```
# let f x = x + 1;;  
val f : int -> int = <fun>
```

Well-formed programs don't
need to be meaningful.

Grammars for Programming Languages

Formal grammars for PL
tell us which **programs** are
well-formed.

Well-formed programs don't
need to be meaningful.

```
# let f x = x + 1;;  
val f : int -> int = <fun>  
# let rec x = x x x x ;;  
Line 1, characters 14-15:  
1 | let rec x = x x x x ;;  
                        ^
```

**Error: This expression has type ...
but an expression was ex ...
The type variable 'a occ ...**

Grammars for Programming Languages

Formal grammars for PL
tell us which **programs** are
well-formed.

Well-formed programs don't
need to be meaningful.

```
# let f x = x + 1;;  
val f : int -> int = <fun>  
# let rec x = x x x x ;;  
Line 1, characters 14-15:  
1 | let rec x = x x x x ;;  
                        ^
```

**Error: This expression has type ...
but an expression was ex ...
The type variable 'a occ ...**

```
# let rec f x = f x + 1 - 1;;  
val f : 'a -> int = <fun>
```

Grammars for Programming Languages

Formal grammars for PL
tell us which **programs** are
well-formed.

Well-formed programs don't
need to be meaningful.

```
# let f x = x + 1;;  
val f : int -> int = <fun>  
# let rec x = x x x x ;;  
Line 1, characters 14-15:  
1 | let rec x = x x x x ;;  
                        ^
```

**Error: This expression has type ...
but an expression was ex ...
The type variable 'a occ ...**

```
# let rec f x = f x + 1 - 1;;  
val f : 'a -> int = <fun>  
# let x = List.hd [];;  
Exception: Failure "hd".
```

Grammars for Programming Languages

Formal grammars for PL
tell us which **programs** are
well-formed.

Well-formed programs don't
need to be meaningful.

*(In OCaml, well-formed programs are
the ones we can type-check.)*

```
# let f x = x + 1;;  
val f : int -> int = <fun>  
# let rec x = x x x x ;;  
Line 1, characters 14-15:  
1 | let rec x = x x x x ;;  
                        ^
```

**Error: This expression has type ...
but an expression was ex ...
The type variable 'a occ ...**

```
# let rec f x = f x + 1 - 1;;  
val f : 'a -> int = <fun>  
# let x = List.hd [];;  
Exception: Failure "hd".
```

Grammars for Programming Languages

Formal grammars for PL
tell us which **programs** are
well-formed.

Well-formed programs don't
need to be meaningful.

*(In OCaml, well-formed programs are
the ones we can type-check.)*

```
# let f x = x + 1;;  
val f : int -> int = <fun>  
# let rec x = x x x x ;;  
Line 1, characters 14-15:  
1 | let rec x = x x x x ;;  
                        ^
```

**Error: This expression has type ...
but an expression was ex ...
The type variable 'a occ ...**

```
# let rec f x = f x + 1 - 1;;  
val f : 'a -> int = <fun>  
# let x = List.hd [];;  
Exception: Failure "hd".  
# let x = ;;  
Line 1, characters 8-10:  
1 | let x = ;;  
            ^^
```

Error: Syntax error

Fundamental Concern

*How do we formally represent
well-formed sentences?*

An Example

the cow jumped over the moon

An Example

the cow jumped over the moon

How do we know this a well-formed sentence?

An Example

`<article>` cow jumped over the moon

An Example

<article> <noun> jumped over the moon

An Example

<noun-phrase> jumped over the moon

An Example

<noun-phrase> jumped over <article> moon

An Example

<noun-phrase> jumped over <article> <noun>

An Example

<noun-phrase> jumped over <noun-phrase>

An Example

<noun-phrase> jumped over <noun-phrase>

a thing jumped over a thing

An Example

<noun-phrase> jumped <prep> <noun-phrase>

An Example

<noun-phrase> jumped <prep-phrase>

An Example

<noun-phrase> <verb> <prep-phrase>

An Example

<noun-phrase> <verb-phrase>

An Example

<noun-phrase> <verb-phrase>

a thing did a thing

An Example

<sentence>

An Example

<sentence>

*We know it's a sentence because it has the right
kind of hierarchical structure*

A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <article> moon
<noun-phrase> jumped over the moon
<article> <noun> jumped over the moon
<article> cow jumped over the moon
the cow jumped over the moon

A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <article> moon
<noun-phrase> jumped over the moon
<article> <noun> jumped over the moon
<article> cow jumped over the moon

the cow jumped over the moon

A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <article> moon
<noun-phrase> jumped over the moon
<article> <noun> jumped over the moon
<article> cow jumped over the moon

the cow jumped over the moon

A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <article> moon
<noun-phrase> jumped over the moon
<article> <noun> jumped over the moon

<article>

the cow jumped over the moon

A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <article> moon
<noun-phrase> jumped over the moon
<article> <noun> jumped over the moon

<article>

the cow jumped over the moon

A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <article> moon
<noun-phrase> jumped over the moon

<article> <noun>
the cow jumped over the moon

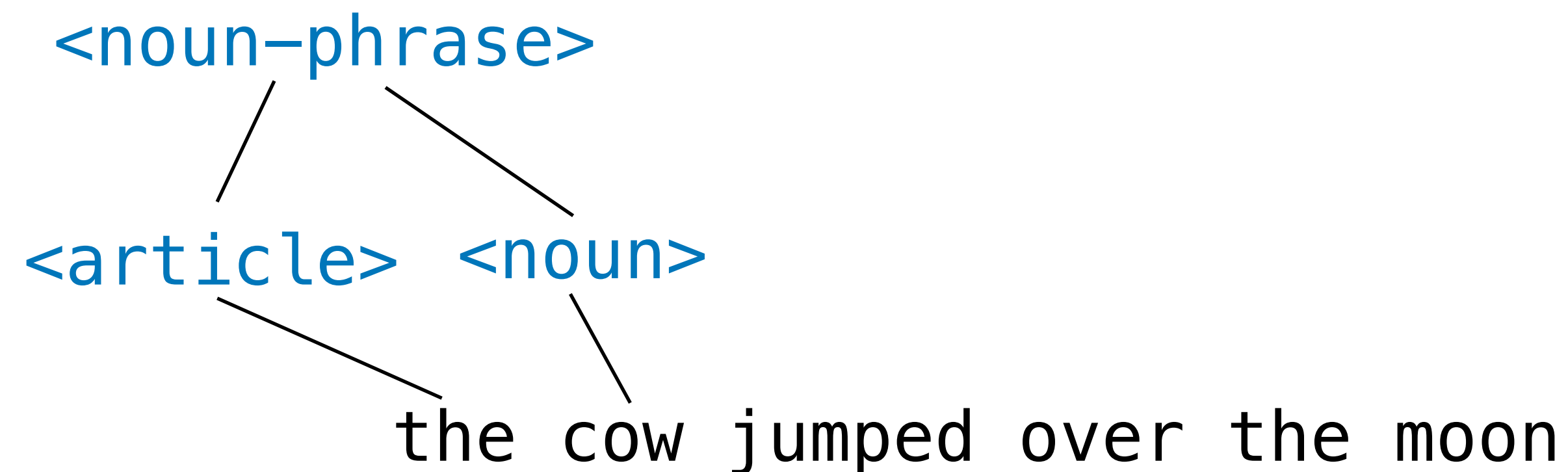
A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <article> moon
<noun-phrase> jumped over the moon

<article> <noun>
the cow jumped over the moon

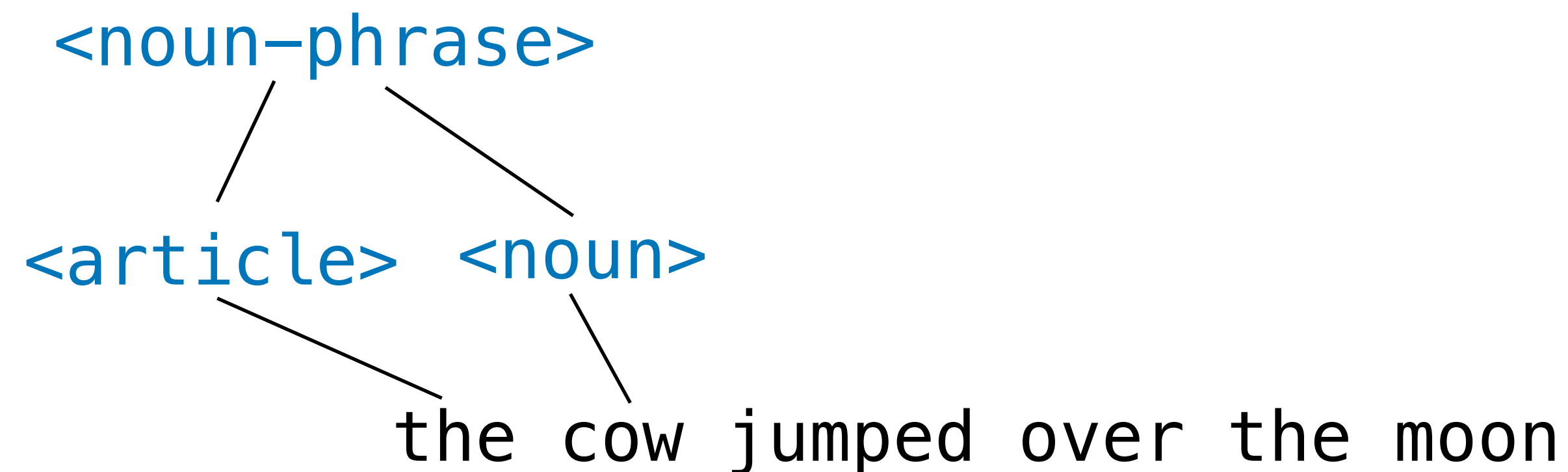
A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <article> moon



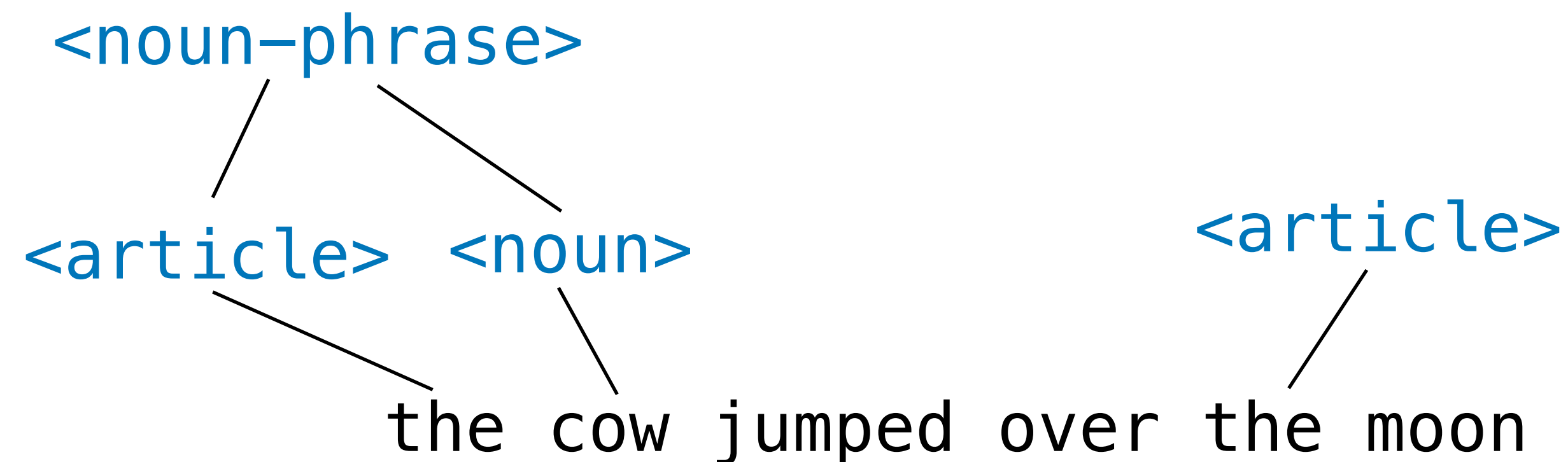
A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <article> moon



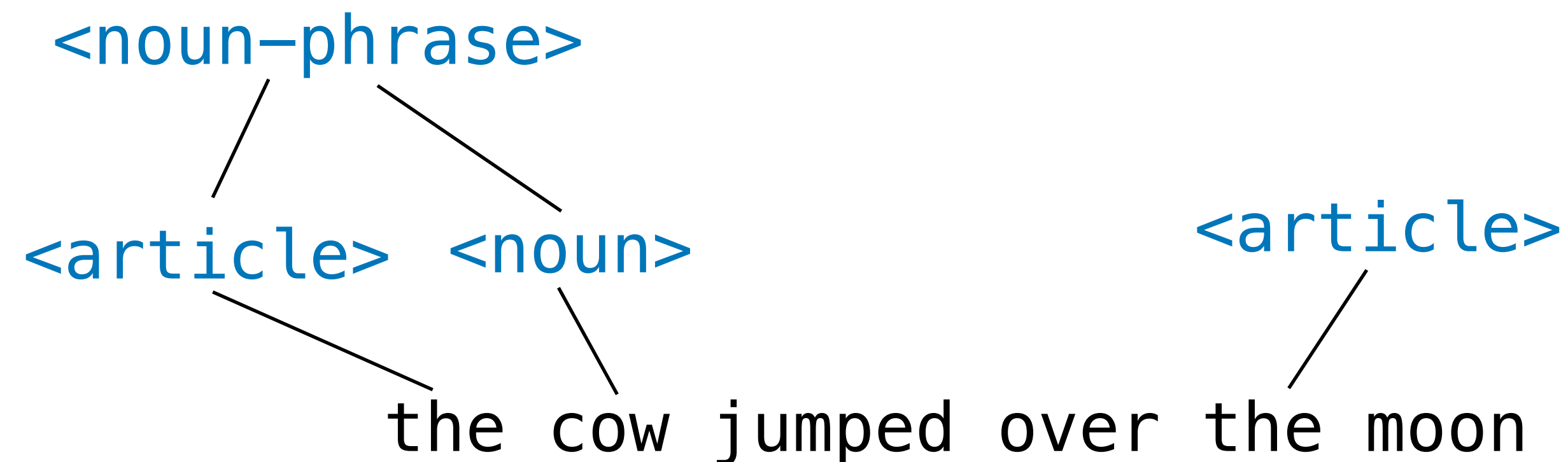
A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>



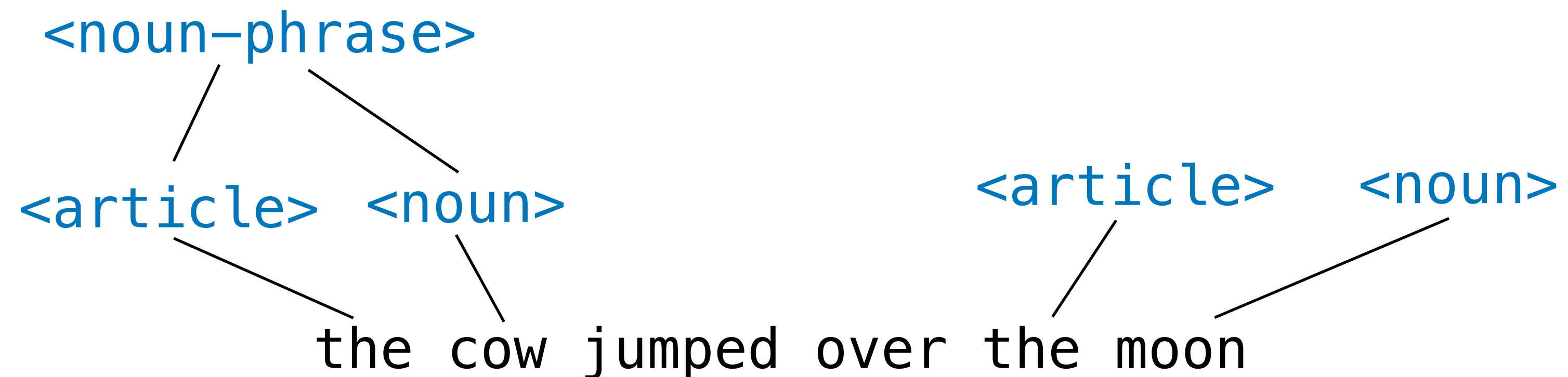
A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>



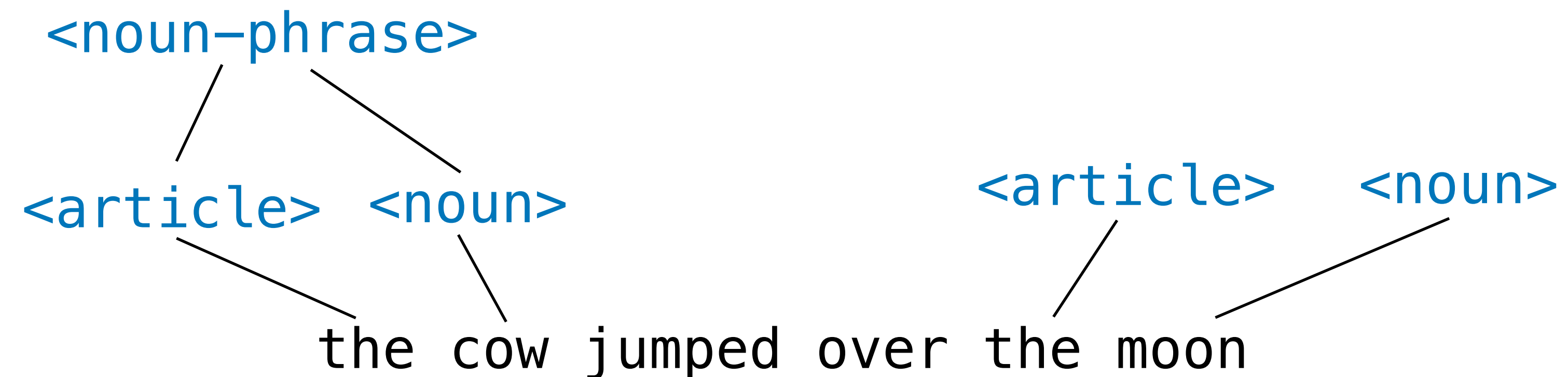
A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>



A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>



A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>



A Derivation

<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>



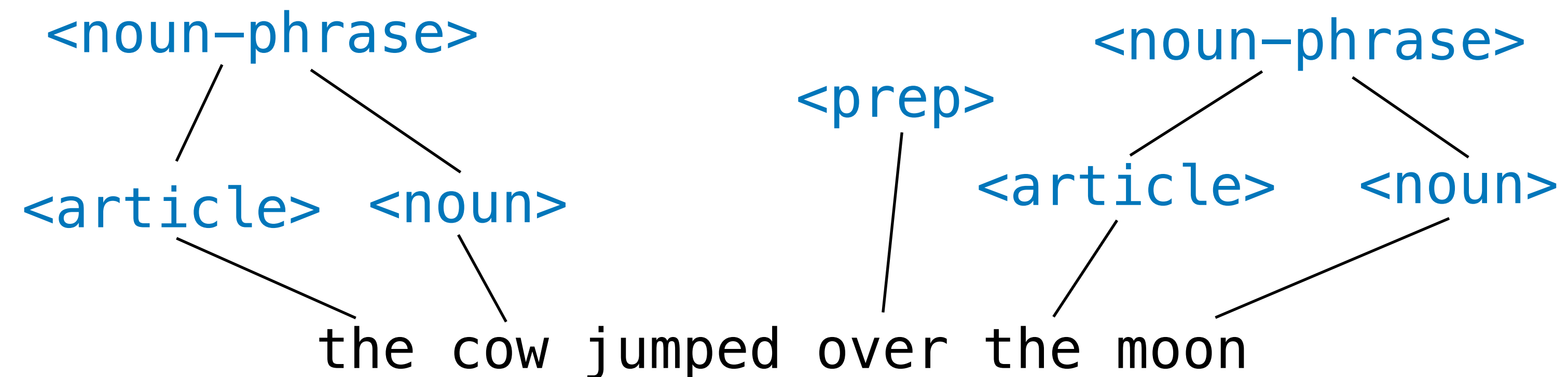
A Derivation

<sentence>

<noun-phrase> <verb-phrase>

<noun-phrase> <verb> <prep-phrase>

<noun-phrase> jumped <prep-phrase>



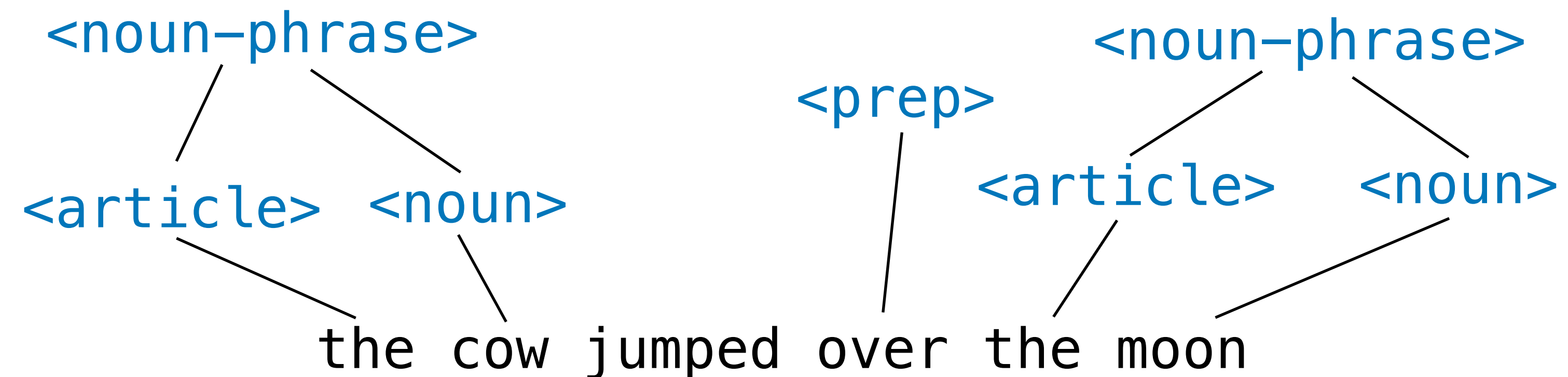
A Derivation

<sentence>

<noun-phrase> <verb-phrase>

<noun-phrase> <verb> <prep-phrase>

<noun-phrase> jumped <prep-phrase>

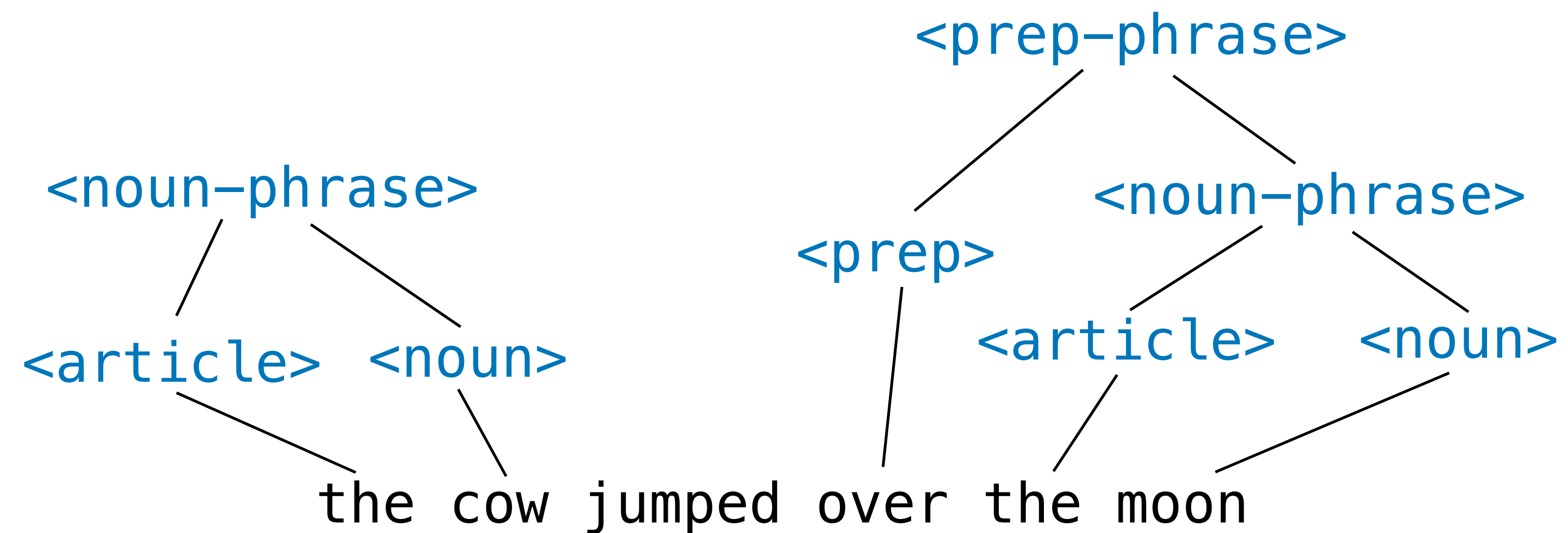


A Derivation

<sentence>

<noun-phrase> <verb-phrase>

<noun-phrase> <verb> <prep-phrase>

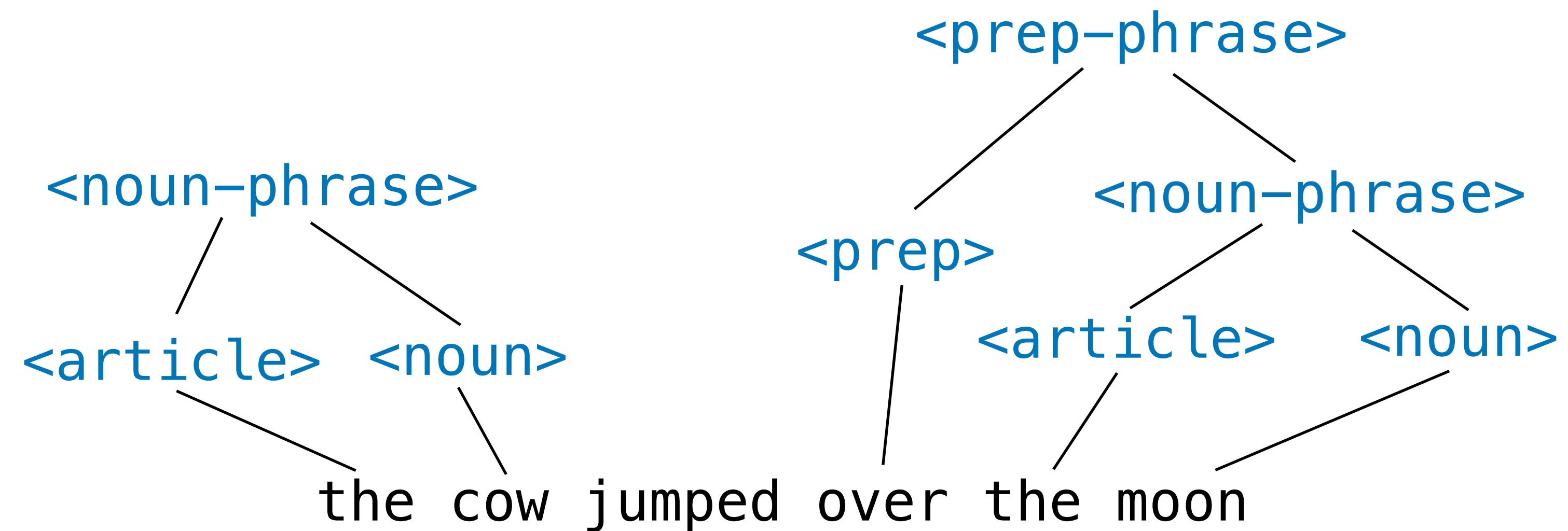


A Derivation

<sentence>

<noun-phrase> <verb-phrase>

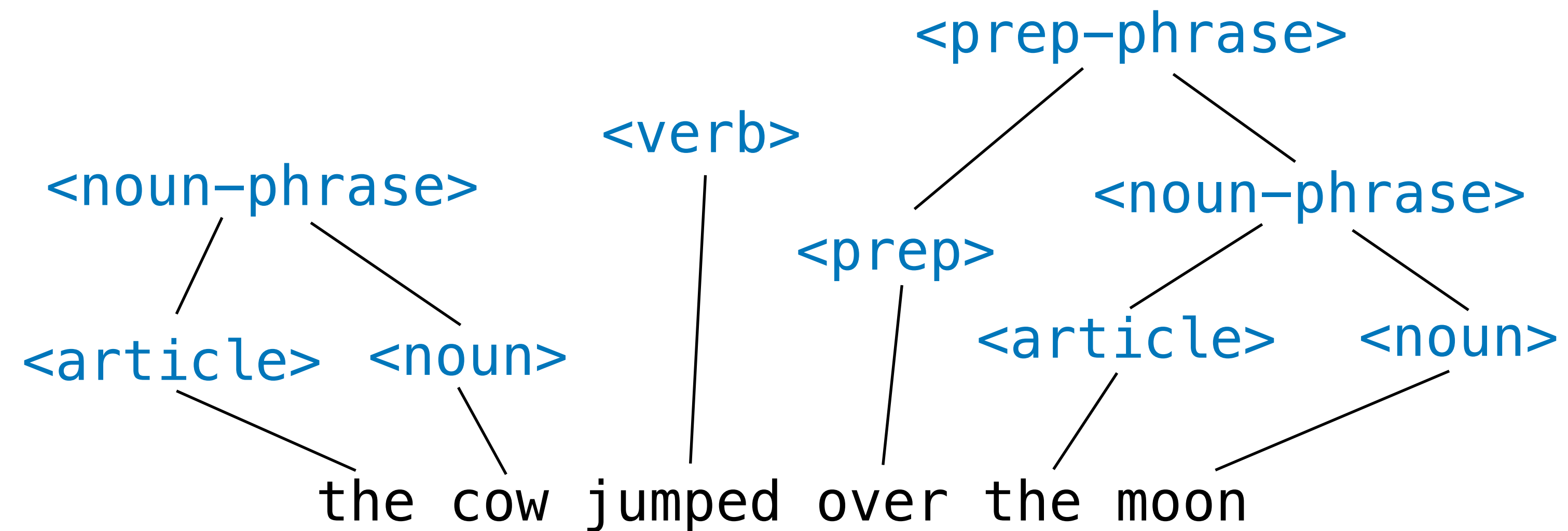
<noun-phrase> <verb> <prep-phrase>



A Derivation

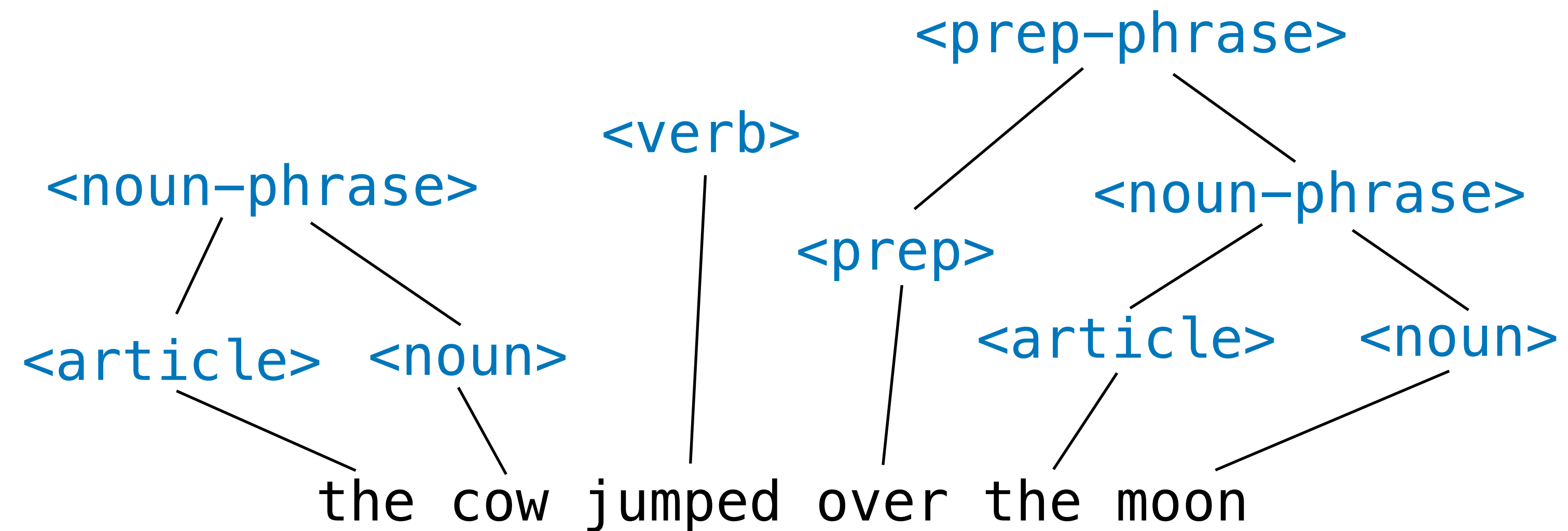
<sentence>

<noun-phrase> <verb-phrase>



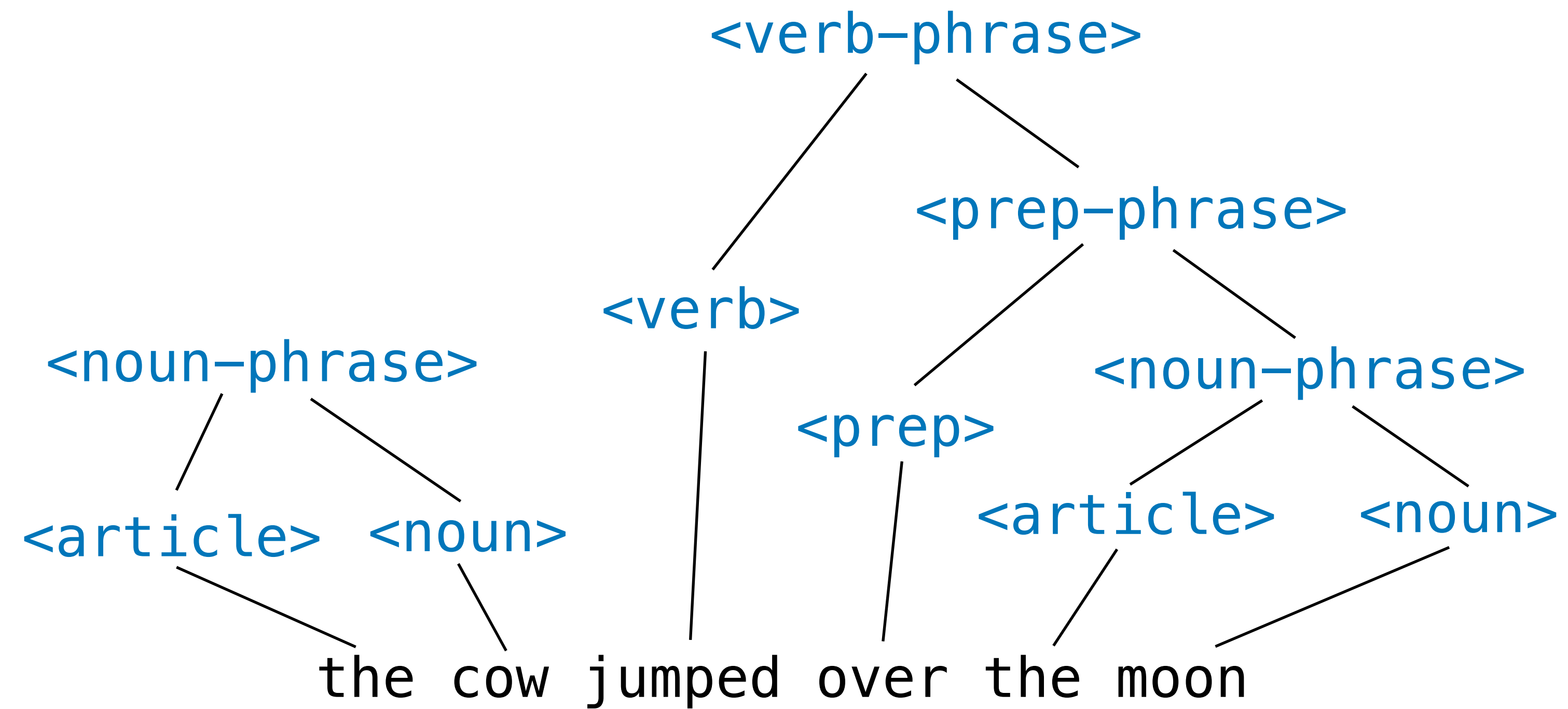
A Derivation

<sentence>
<noun-phrase> <verb-phrase>



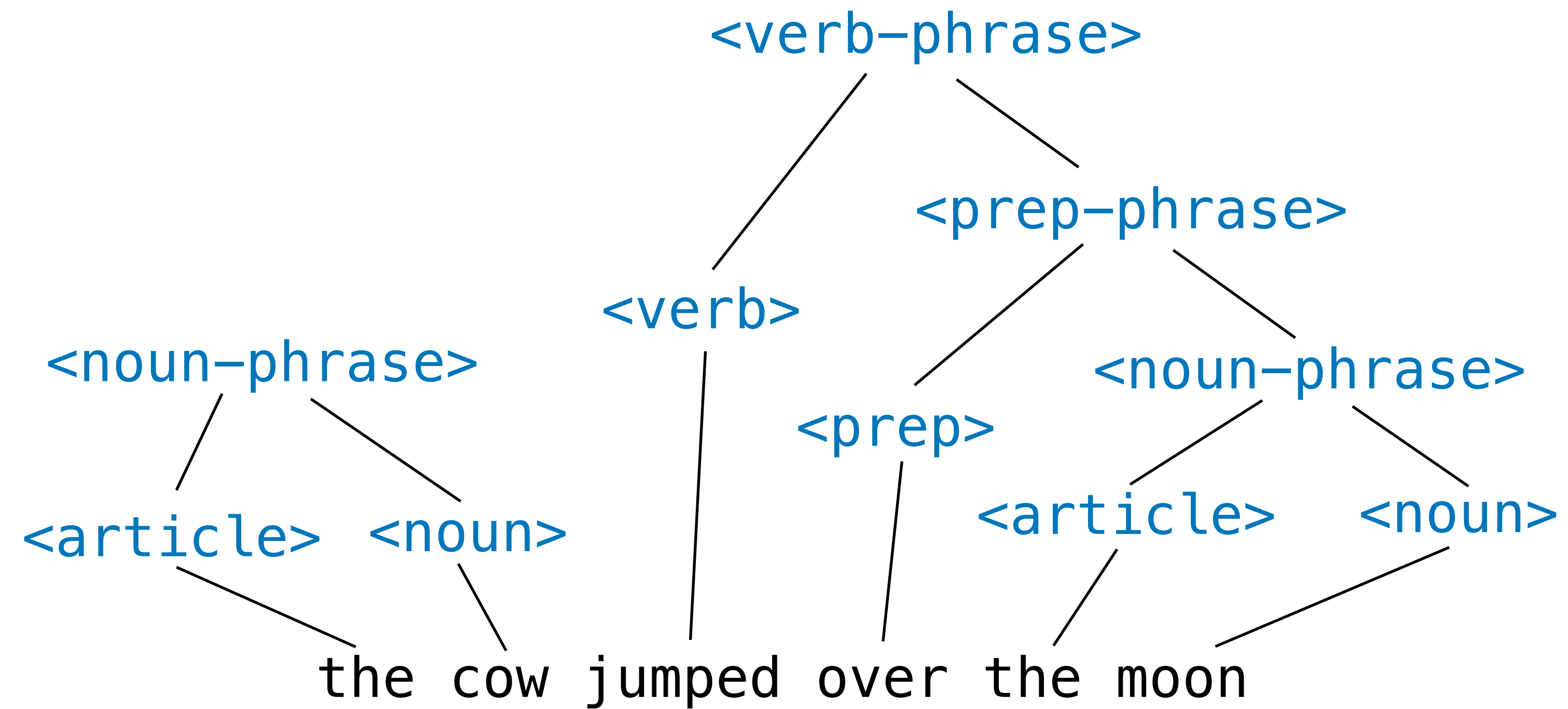
A Derivation

<sentence>

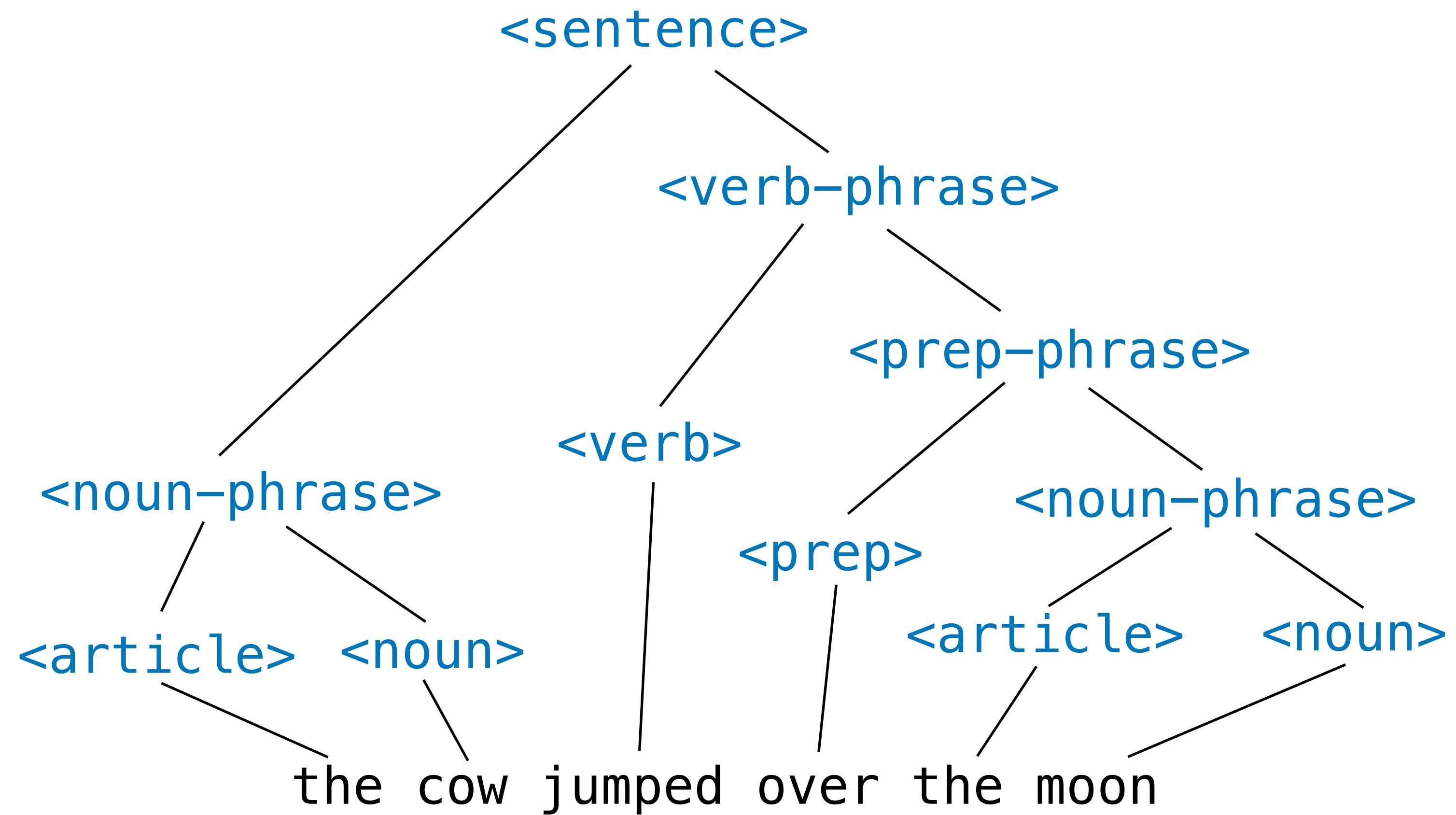


A Derivation

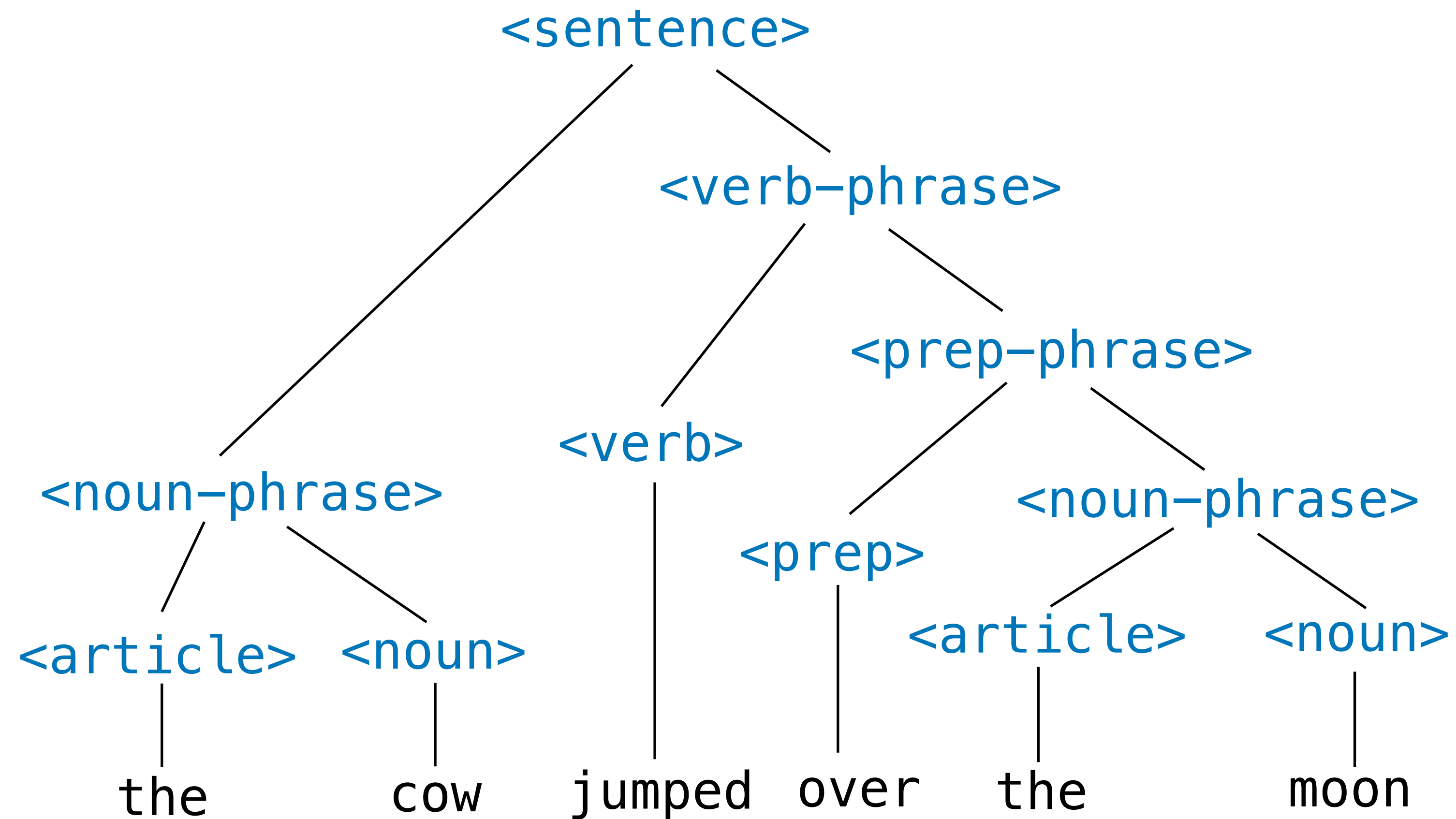
<sentence>



A Derivation



A Parse Tree



A derivation encodes hierarchical structure

Definitions (Symbols and Sentences)

Definitions (Symbols and Sentences)

A grammar is define in terms of a collection of symbols.

Definitions (Symbols and Sentences)

A grammar is define in terms of a collection of symbols.

Nonterminal symbols are symbols we will be allowed to "expand", like **<article>** in the previous example

Definitions (Symbols and Sentences)

A grammar is define in terms of a collection of symbols.

Nonterminal symbols are symbols we will be allowed to "expand", like `<article>` in the previous example

Terminal symbols are symbols cannot be further expanded, like *moon* in the previous example

Definitions (Symbols and Sentences)

A grammar is define in terms of a collection of symbols.

Nonterminal symbols are symbols we will be allowed to "expand", like `<article>` in the previous example

Terminal symbols are symbols cannot be further expanded, like *moon* in the previous example

A **sentential form** is a sequence of terminal or nonterminal symbols.

Definitions (Symbols and Sentences)

A grammar is define in terms of a collection of symbols.

Nonterminal symbols are symbols we will be allowed to "expand", like `<article>` in the previous example

Terminal symbols are symbols cannot be further expanded, like *moon* in the previous example

A **sentential form** is a sequence of terminal or nonterminal symbols.

A **sentence** is a sequence of *only terminal* symbols

Example

sent. form.

nonterminal

<noun-phrase>

jumped

over

<noun-phrase>

terminal

Production Rules

$\langle \text{non-term} \rangle ::= \textit{sent-form1} \mid \textit{sent-form2} \mid \dots$

Production Rules

`<non-term> ::= sent-form1 | sent-form2 | ...`

A **(BNF) production rule** describes what we can replace a non-terminal symbol with in a derivation.

Production Rules

`<non-term> ::= sent-form1 | sent-form2 | ...`

A **(BNF) production rule** describes what we can replace a non-terminal symbol with in a derivation.

The "|" means: we can replace it with one or the other sentential forms on either side of the "|".

Example

<sentence> ::= <noun-phrase> <verb-phrase>

<verb-phrase> ::= <verb> <prep-phrase>

<noun> ::= cow | moon

BNF Grammar

```
<sentence> ::= <noun-phrase> <verb-phrase>
<verb-phrase> ::= <verb> <prep-phrase>
                  | <verb>
<prep-phrase> ::= <prep> <noun-phrase>
<noun-phrase> ::= <article> <noun>
<article> ::= the
<noun> ::= cow
          | moon
<verb> ::= jumped
<prep> ::= over
```

BNF Grammar

A **BNF grammar** is defined by a collection of production rules and a **starting (nonterminal) symbol**

```
<sentence> ::= <noun-phrase> <verb-phrase>
<verb-phrase> ::= <verb> <prep-phrase>
                  | <verb>
<prep-phrase> ::= <prep> <noun-phrase>
<noun-phrase> ::= <article> <noun>
<article> ::= the
<noun> ::= cow
          | moon
<verb> ::= jumped
<prep> ::= over
```

BNF Grammar

A **BNF grammar** is defined by a collection of production rules and a **starting (nonterminal) symbol**

Note. We don't specify the symbols of a grammar, they are implicit in the rules

```
<sentence> ::= <noun-phrase> <verb-phrase>
<verb-phrase> ::= <verb> <prep-phrase>
                  | <verb>
<prep-phrase> ::= <prep> <noun-phrase>
<noun-phrase> ::= <article> <noun>
<article> ::= the
<noun> ::= cow
          | moon
<verb> ::= jumped
<prep> ::= over
```

BNF Grammar

A **BNF grammar** is defined by a collection of production rules and a **starting (nonterminal) symbol**

Note. We don't specify the symbols of a grammar, they are implicit in the rules

Note. We don't specify the start symbol, it's the left nonterminal symbol in the **first rule**

```
<sentence> ::= <noun-phrase> <verb-phrase>
<verb-phrase> ::= <verb> <prep-phrase>
                  | <verb>
<prep-phrase> ::= <prep> <noun-phrase>
<noun-phrase> ::= <article> <noun>
<article> ::= the
<noun> ::= cow
          | moon
<verb> ::= jumped
<prep> ::= over
```

Example

<expr>	::=	<op1>	<expr>	
		<op2>	<expr>	<expr>
		<var>		
<op1>	::=	not		
<op2>	::=	and		or
<var>	::=	x		y z

Example

<expr> ::= <op1> <expr>
 | <op2> <expr> <expr>
 | <var>

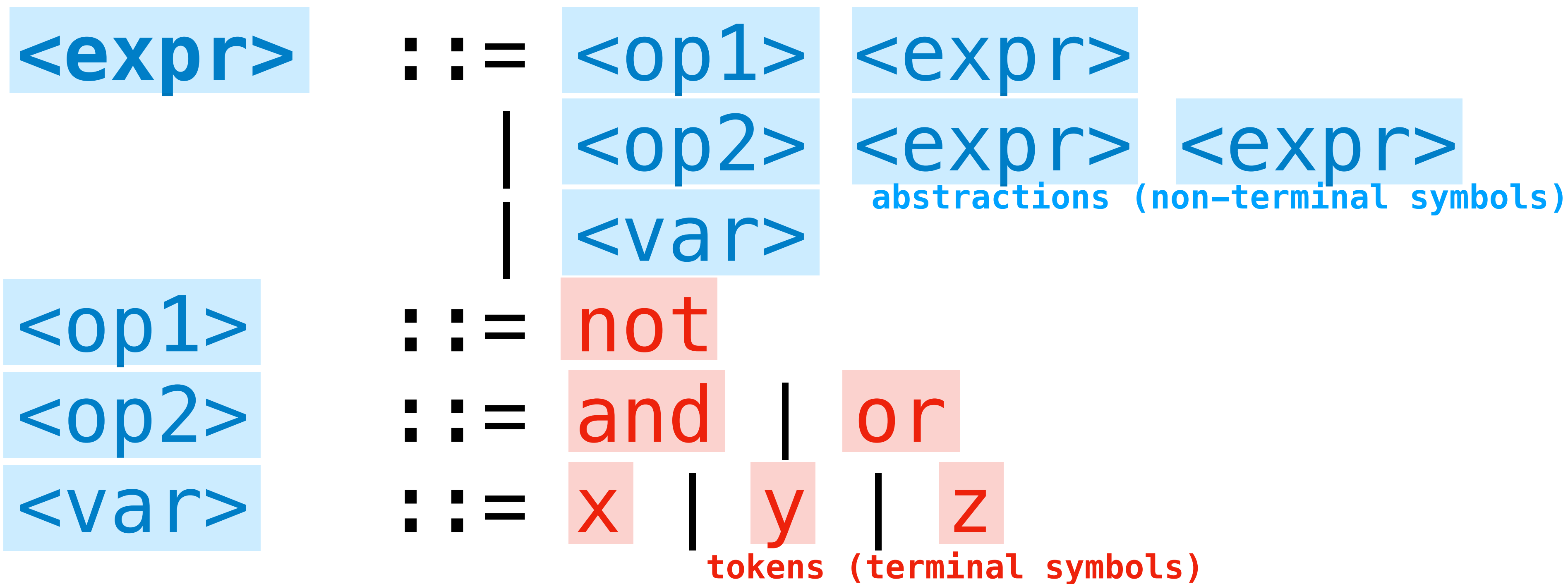
<op1> ::= not

<op2> ::= and | or

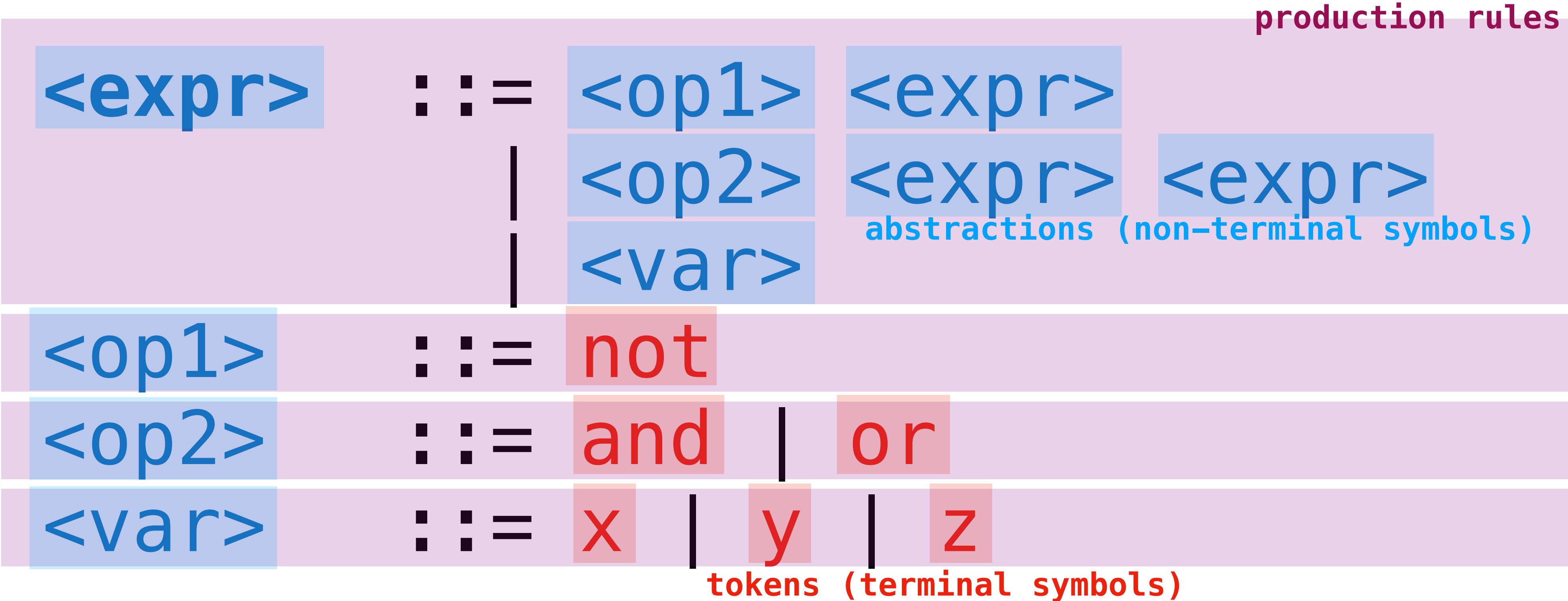
<var> ::= x | y | z

tokens (terminal symbols)

Example



Example



Practice Problem

`<sentence>` ::= `<noun-phrase>` `<verb-phrase>`
`<verb-phrase>` ::= `<verb>` `<prep-phrase>` | `<verb>`
`<prep-phrase>` ::= `<prep>` `<noun-phrase>`
`<noun-phrase>` ::= `<article>` `<noun>`
`<article>` ::= `the`
`<noun>` ::= `cow` | `moon`
`<verb>` ::= `jumped`
`<prep>` ::= `over`

What are the nonterminal and terminal symbols of this grammar?

Answer

```
<sentence> ::= <noun-phrase> <verb-phrase>
<verb-phrase> ::= <verb> <prep-phrase> | <verb>
<prep-phrase> ::= <prep> <noun-phrase>
<noun-phrase> ::= <article> <noun>
<article> ::= the
<noun> ::= cow | moon
<verb> ::= jumped
<prep> ::= over
```

Derivations and Parse Trees

Derivations and Parse Trees

Definition. A **derivation** is a
sequence of sentential forms
(beginning at the start symbol)
in which each form is the result
of replacing a non-terminal
symbol in the previous form
according to a production rule

Derivations and Parse Trees

Definition. A **derivation** is a **sequence of sentential forms** (beginning at the start symbol) in which each form is the result of **replacing a non-terminal symbol in the previous form** according to a production rule

Definition. A **leftmost derivation** is a derivation in which the leftmost nonterminal symbol is replaced in each line

Derivations and Parse Trees

Definition. A **derivation** is a sequence of sentential forms (beginning at the start symbol) in which each form is the result of replacing a non-terminal symbol in the previous form according to a production rule

Definition. A **leftmost derivation** is a derivation in which the leftmost nonterminal symbol is replaced in each line

```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
and not x <var>
and not x y
```

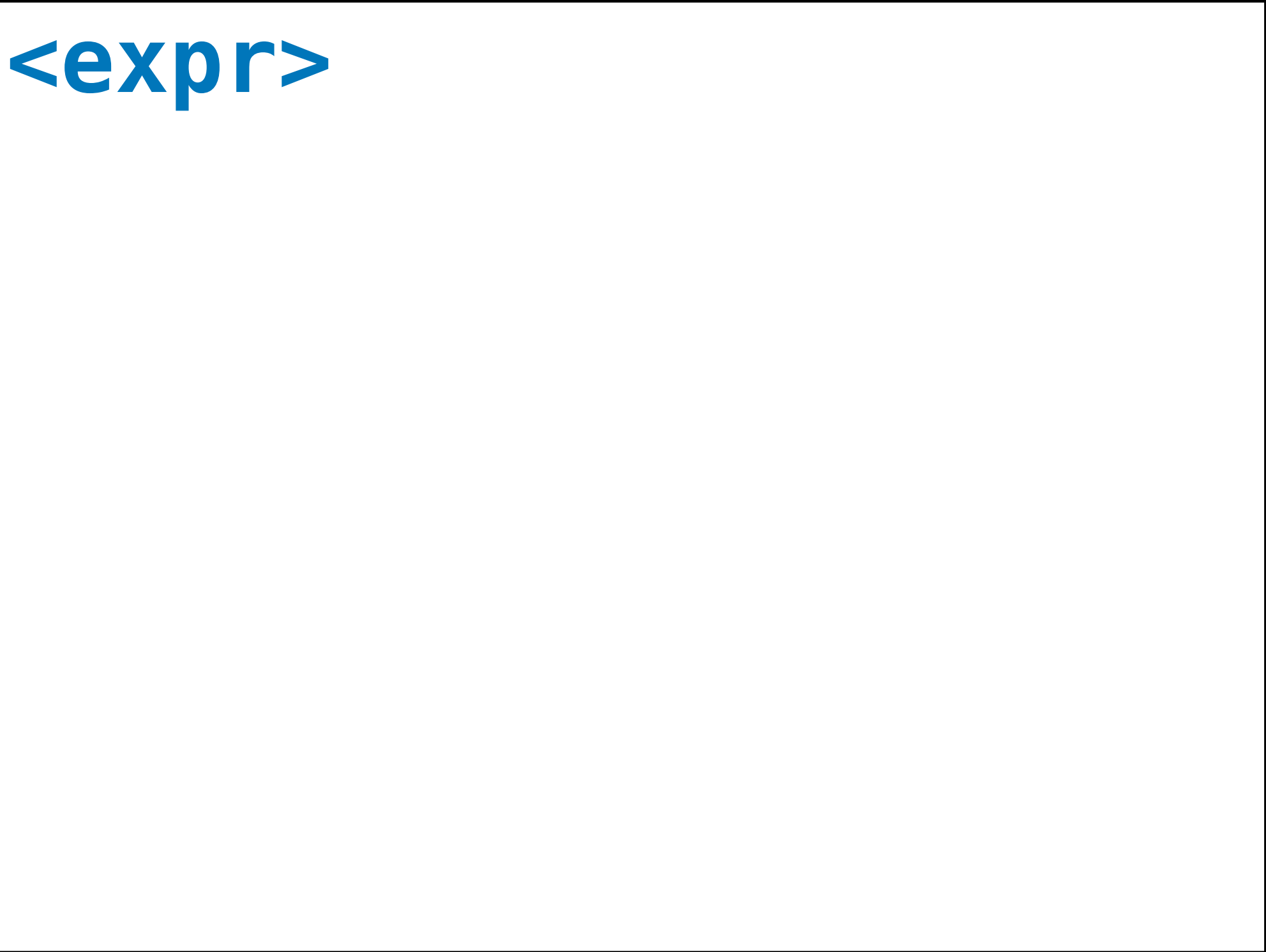
Derivations and Parse Trees

```
<expr> ::= <op1> <expr>
          | <op2> <expr> <expr>
          | <var>
<op1>   ::= not
<op2>   ::= and | or
<var>   ::= x  | y  | z
```


Derivations and Parse Trees

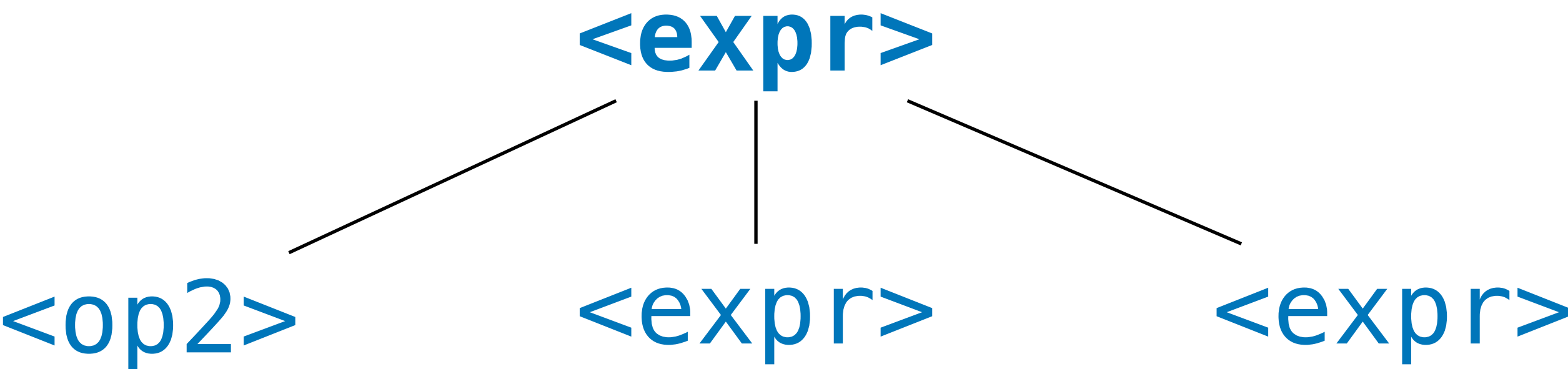
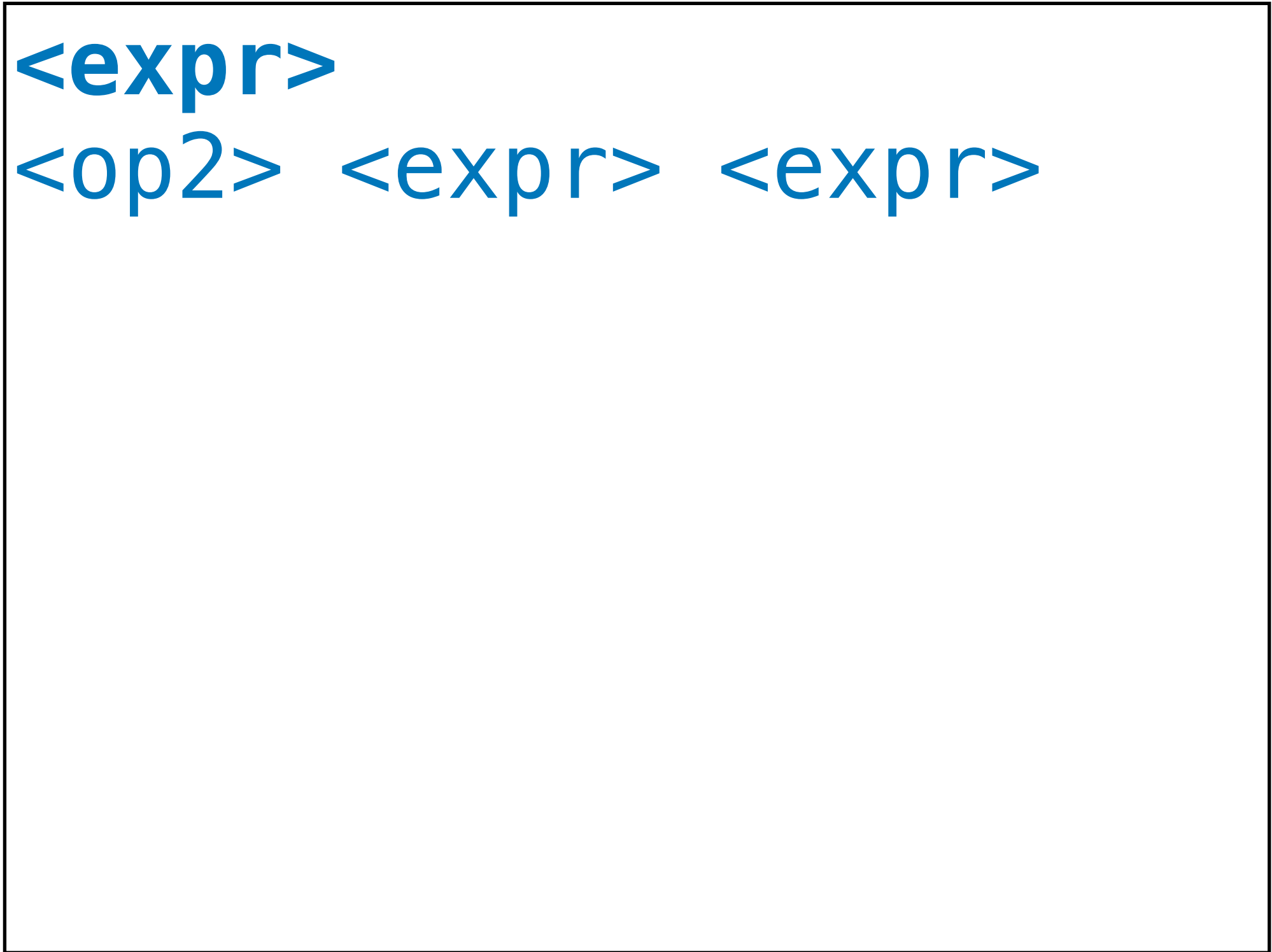
<expr>	::=	<op1>	<expr>	
			<op2>	<expr> <expr>
			<var>	
<op1>	::=	not		
<op2>	::=	and		or
<var>	::=	x		y z

<expr>



Derivations and Parse Trees

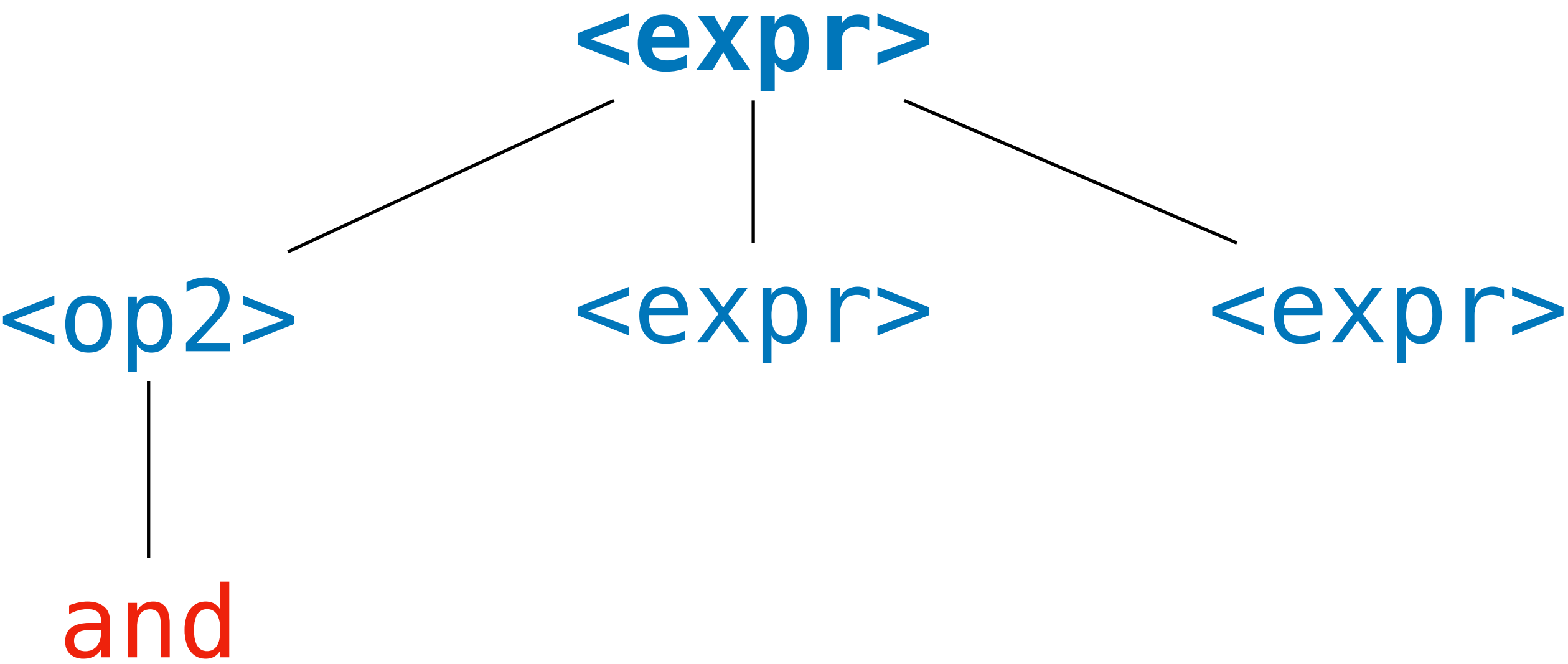
<expr>	::= <op1> <expr>			
	<op2> <expr> <expr>			
	<var>			
<op1>	::= not			
<op2>	::= and		or	
<var>	::= x		y	z



Derivations and Parse Trees

<expr>	::=	<op1>	<expr>	
		<op2>	<expr>	<expr>
		<var>		
<op1>	::=	not		
<op2>	::=	and	 	or
<var>	::=	x	 	y z

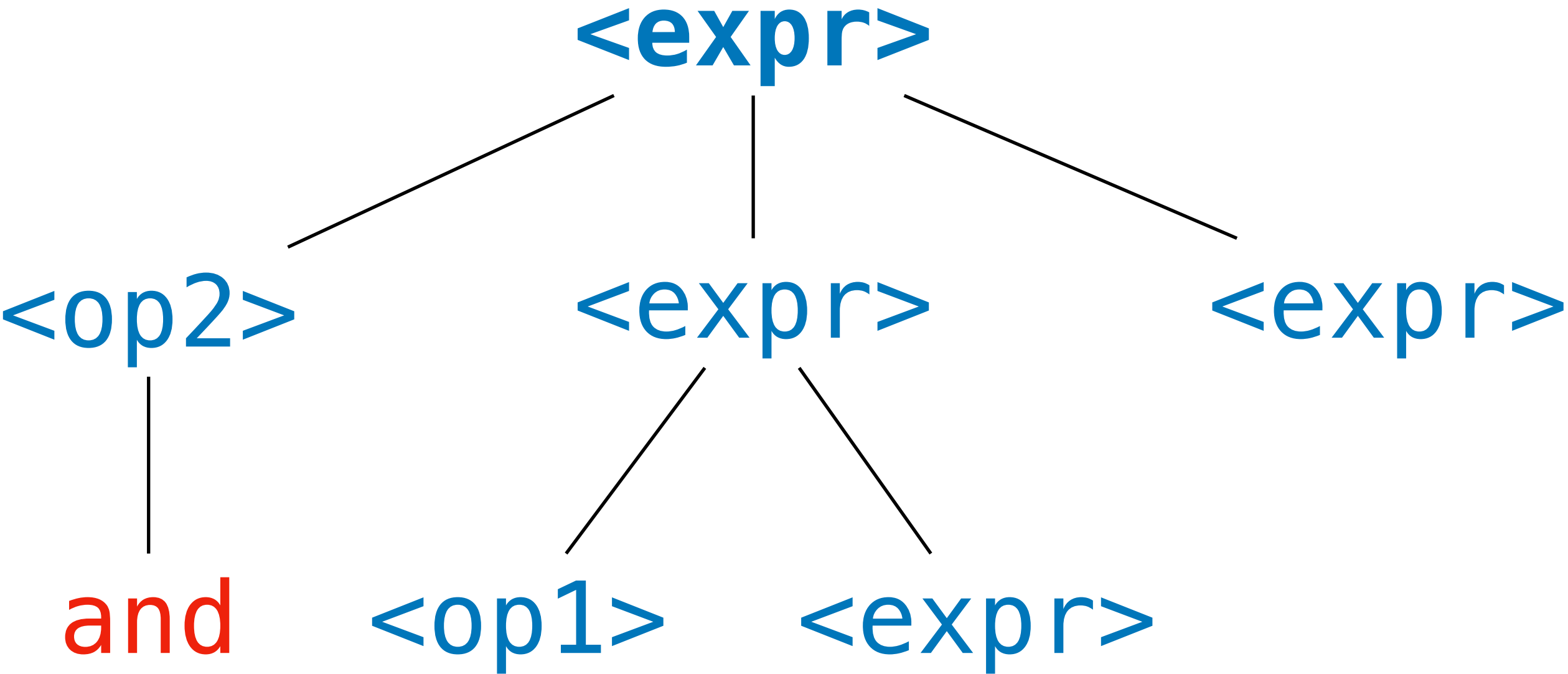
<expr>
<op2> **<expr>** **<expr>**
and **<expr>** **<expr>**



Derivations and Parse Trees

<expr>	::=	<op1>	<expr>	
		<op2>	<expr>	<expr>
		<var>		
<op1>	::=	not		
<op2>	::=	and		or
<var>	::=	x		y z

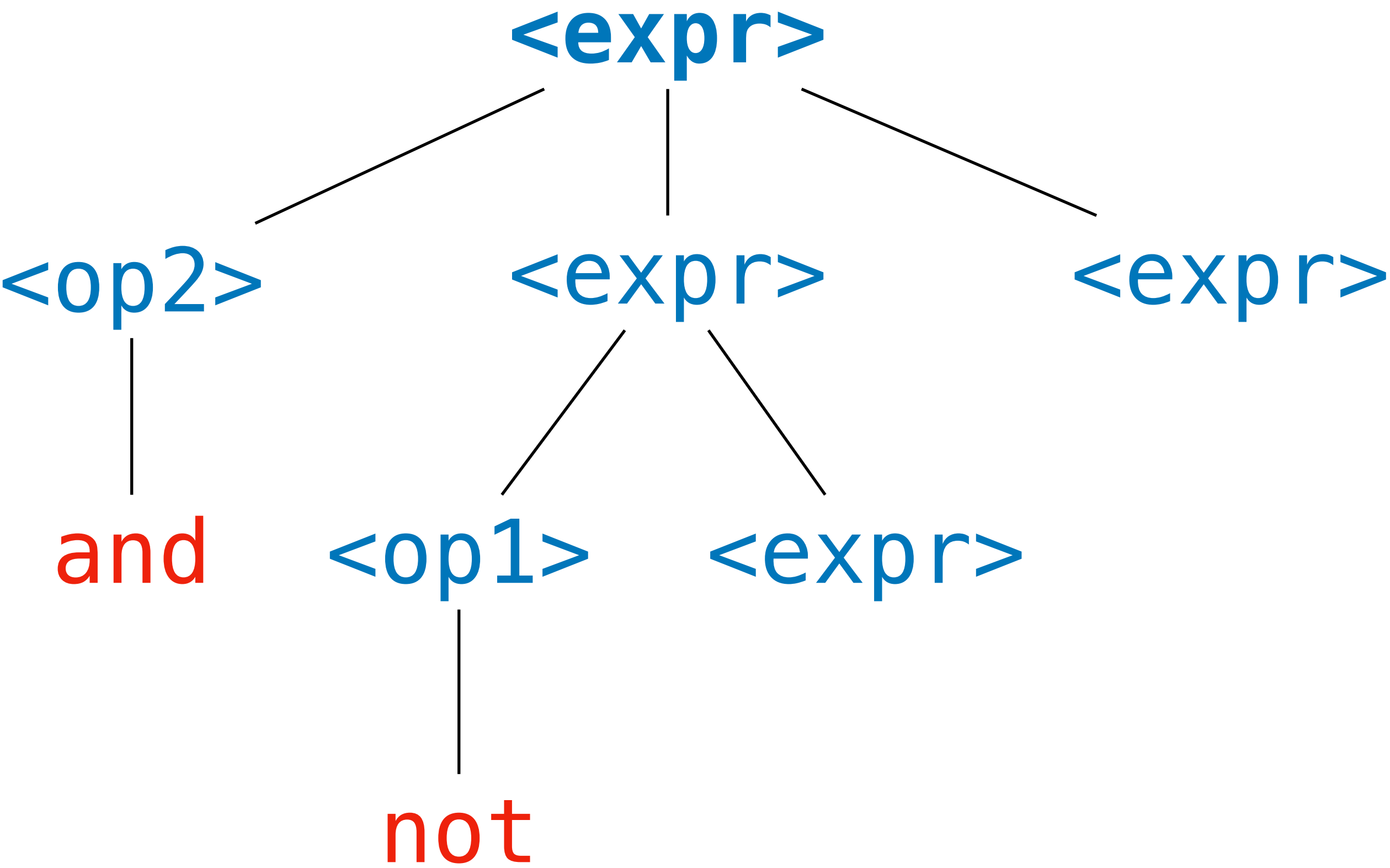
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>



Derivations and Parse Trees

<expr>	::=	<op1>	<expr>	
		<op2>	<expr>	<expr>
		<var>		
<op1>	::=	not		
<op2>	::=	and	 	or
<var>	::=	x	 	y z

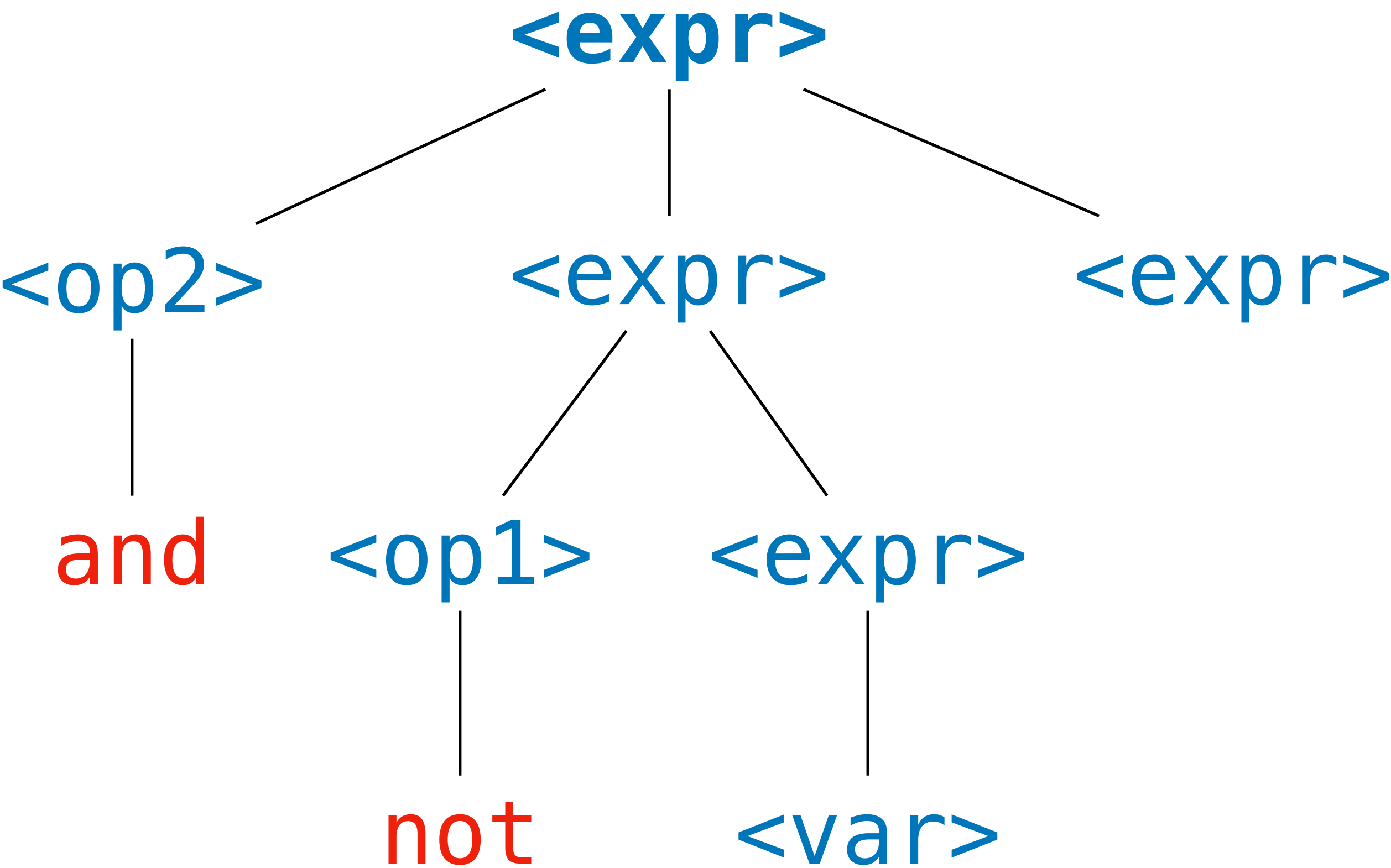
<expr>
<op2> **<expr>** **<expr>**
and **<expr>** **<expr>**
and **<op1>** **<expr>** **<expr>**
and **not** **<expr>** **<expr>**



Derivations and Parse Trees

<expr>	::=	<op1>	<expr>
		<op2>	<expr> <expr>
		<var>	
<op1>	::=	not	
<op2>	::=	and	or
<var>	::=	x	y z

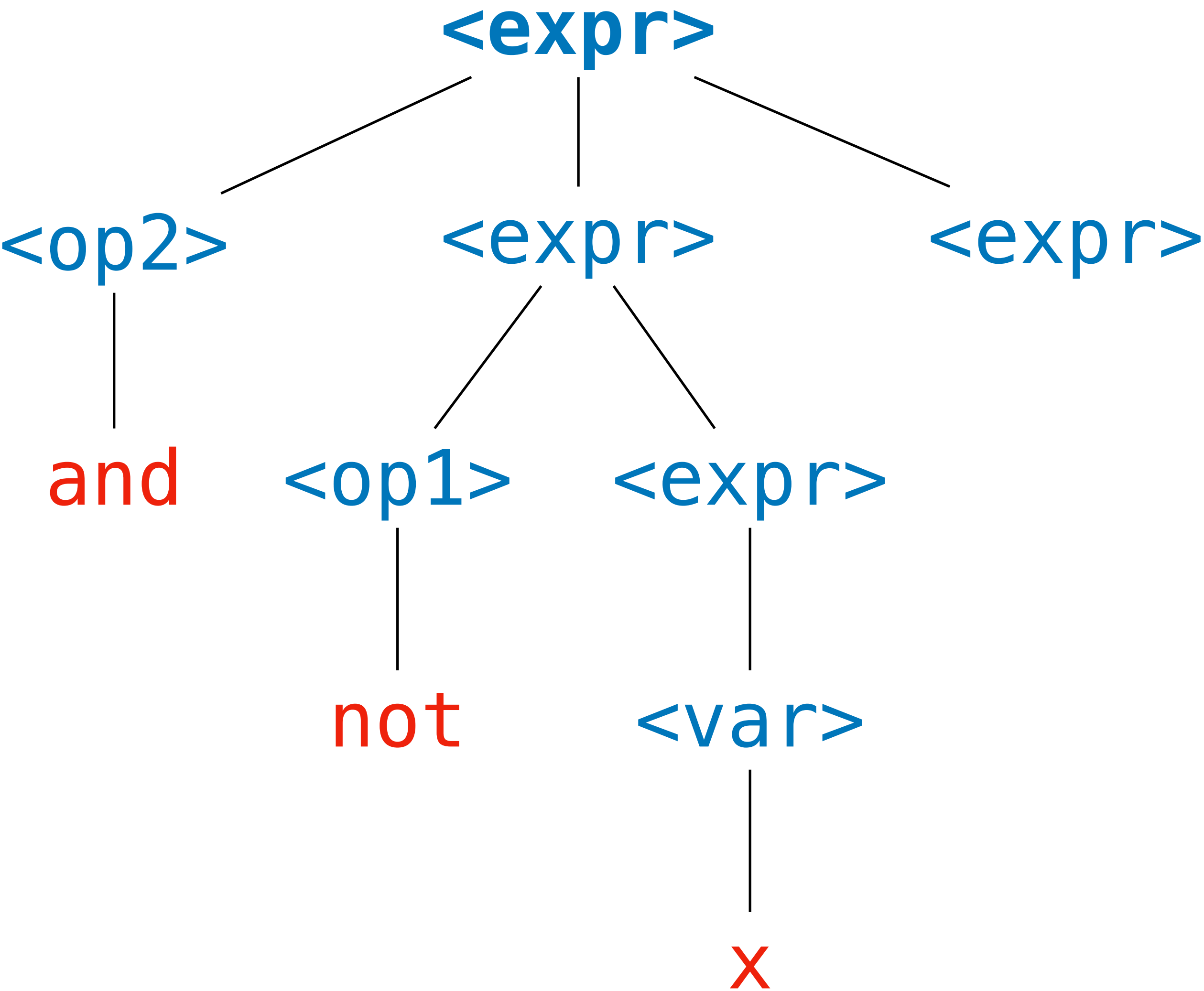
<expr>
<op2> **<expr>** **<expr>**
and **<expr>** **<expr>**
and **<op1>** **<expr>** **<expr>**
and not **<expr>** **<expr>**
and not **<var>** **<expr>**



Derivations and Parse Trees

<expr>	::=	<op1>	<expr>	
		<op2>	<expr>	<expr>
		<var>		
<op1>	::=	not		
<op2>	::=	and	 	or
<var>	::=	x	 	y z

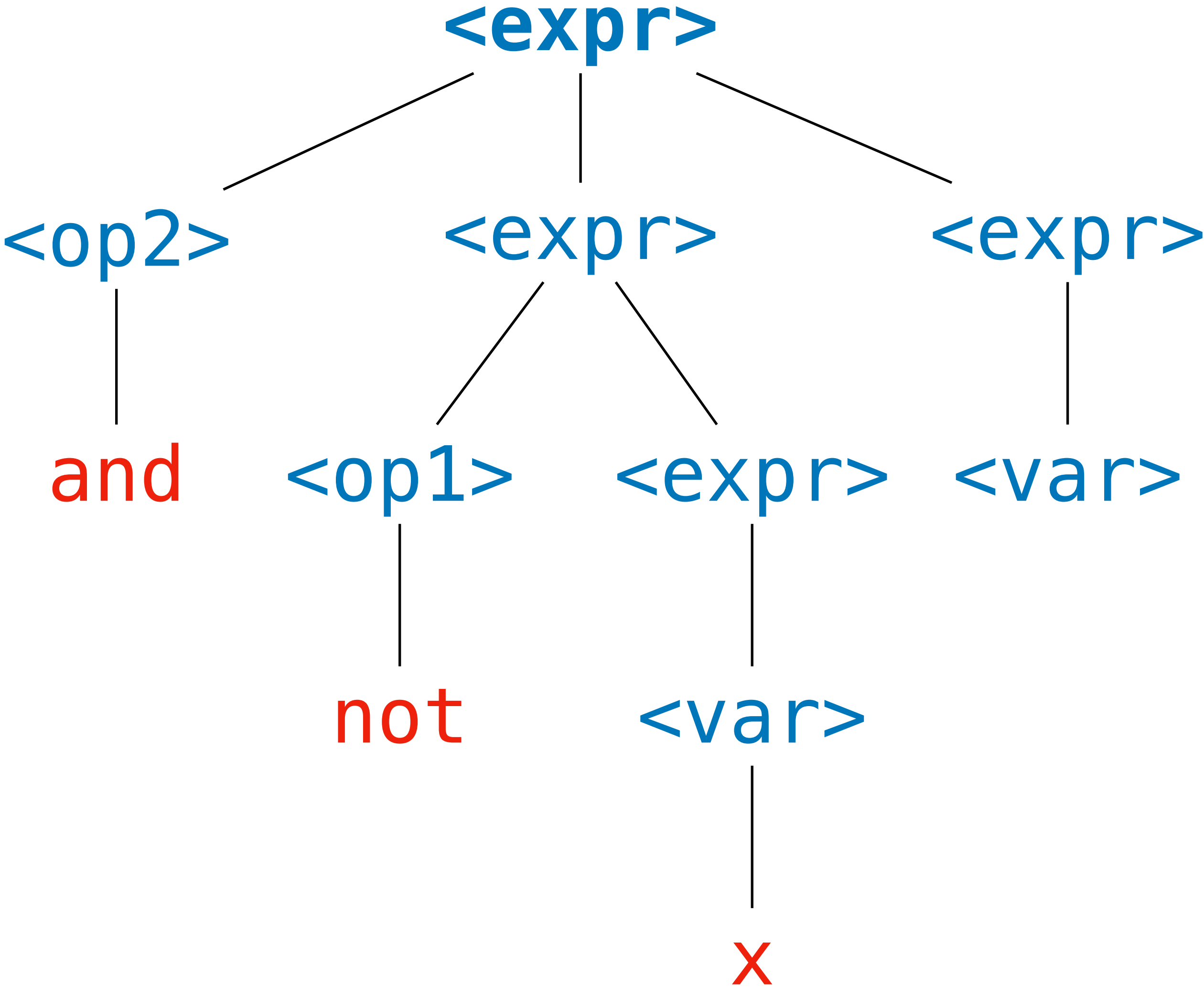
<expr>
<op2> **<expr>** **<expr>**
and **<expr>** **<expr>**
and **<op1>** **<expr>** **<expr>**
and not **<expr>** **<expr>**
and not **<var>** **<expr>**
and not x **<expr>**



Derivations and Parse Trees

<expr>	::=	<op1>	<expr>	
			<op2>	<expr> <expr>
			<var>	
<op1>	::=	not		
<op2>	::=	and		or
<var>	::=	x		y z

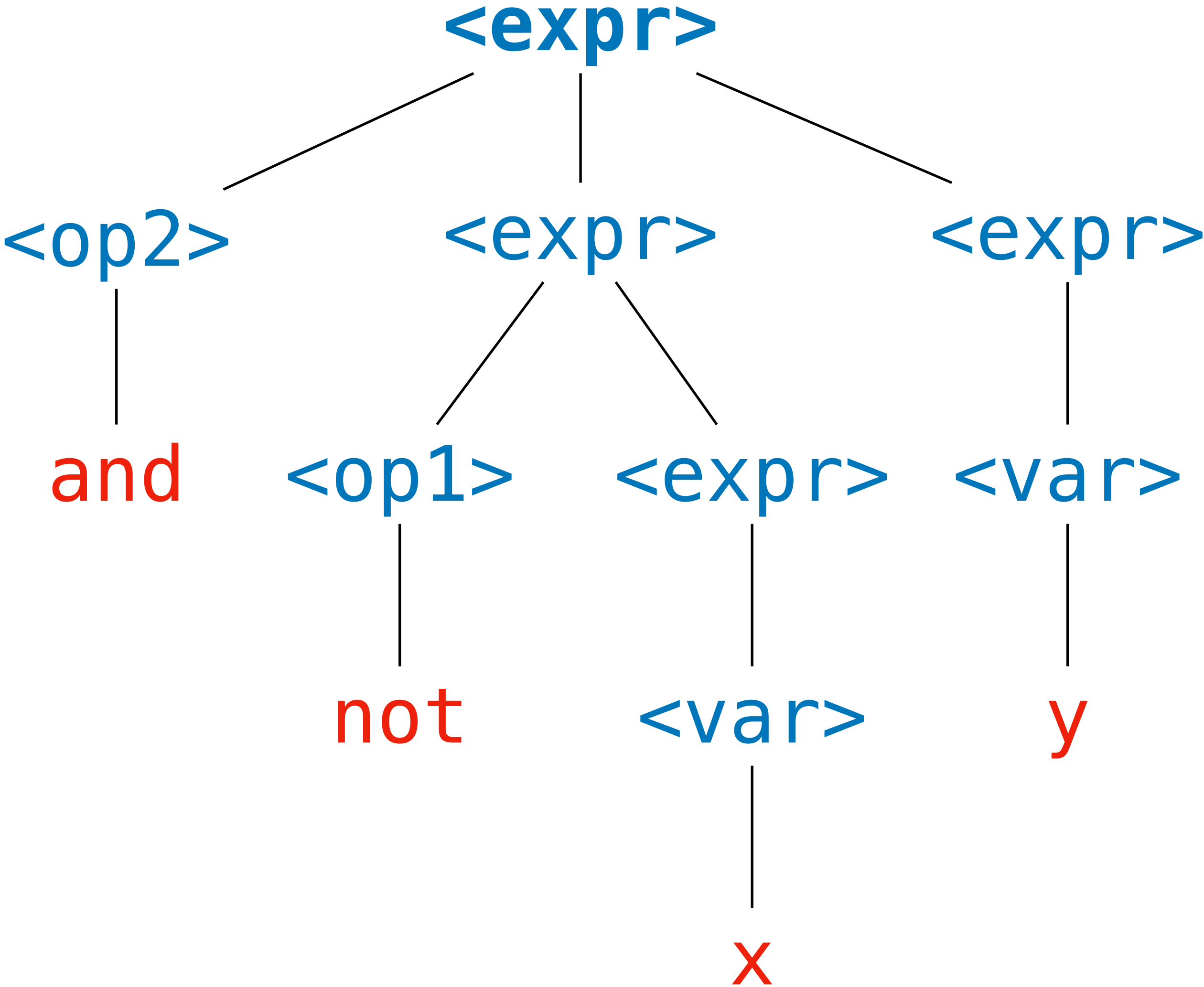
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
and not x <var>



Derivations and Parse Trees

<expr>	::=	<op1>	<expr>	
			<op2>	<expr> <expr>
			<var>	
<op1>	::=	not		
<op2>	::=	and		or
<var>	::=	x		y z

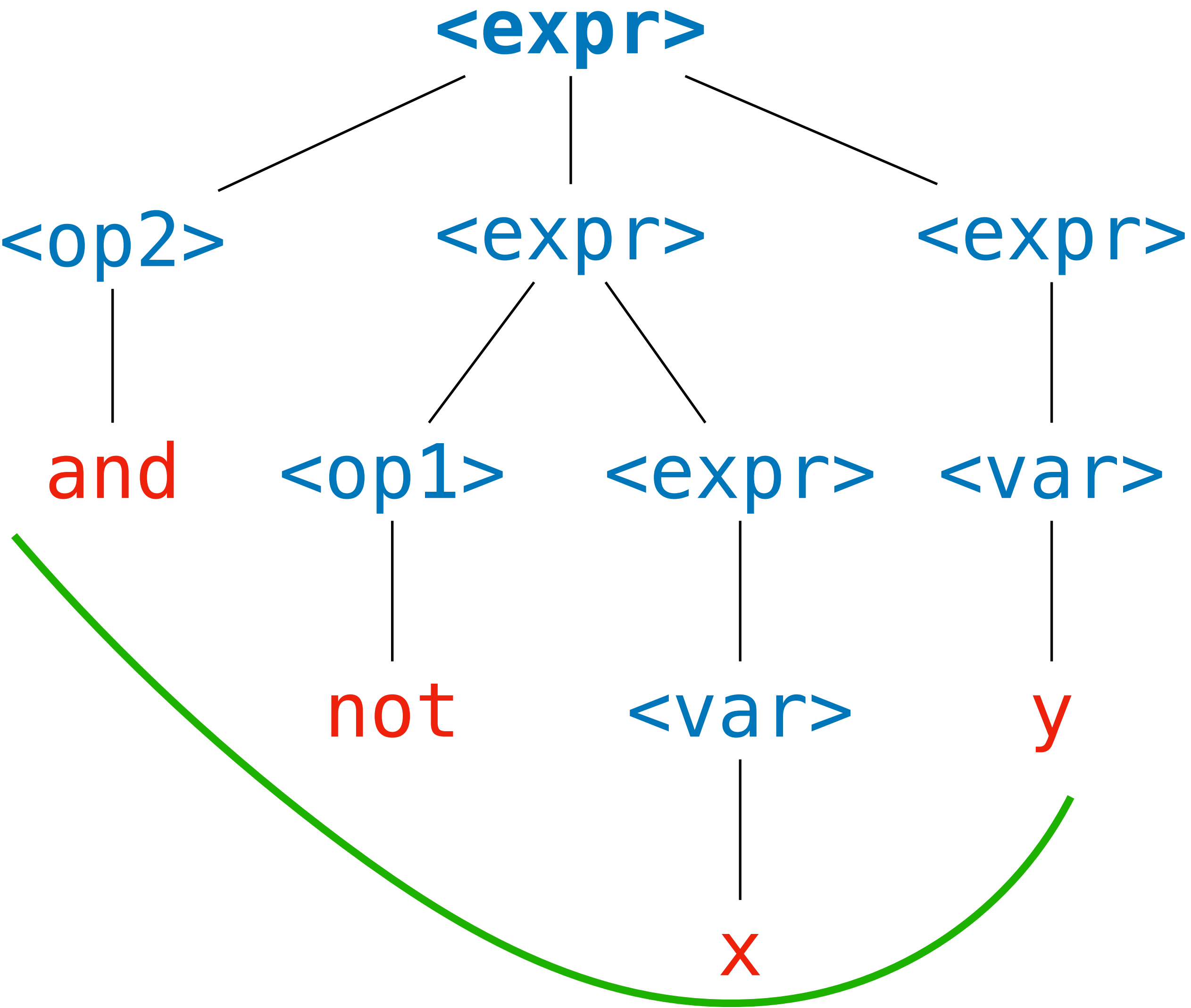
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
and not x <var>
and not x y



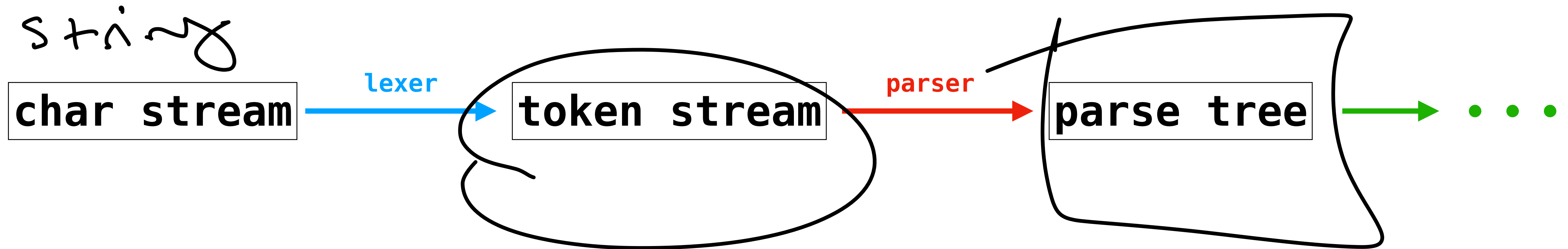
Derivations and Parse Trees

<expr>	::=	<op1>	<expr>
		<op2>	<expr> <expr>
		<var>	
<op1>	::=	not	
<op2>	::=	and	or
<var>	::=	x	y z

<expr>
<op2> **<expr>** **<expr>**
and **<expr>** **<expr>**
and **<op1>** **<expr>** **<expr>**
and not **<expr>** **<expr>**
and not **<var>** **<expr>**
and not **x** **<expr>**
and not **x** **<var>**
and not **x** **y**



Why do we care?



Why do we care?



Why do we care?



Why do we care?



We will parse **token streams** into **parse trees**.

Why do we care?



We will parse **token streams** into **parse trees**.

*It is much easier to **evaluate** something hierarchical than something which is linear.*

Practice Problem

<code><expr></code>	<code>::=</code>	<code><op1></code>	<code><expr></code>
			<code><expr></code>
			<code><op2></code>
			<code><expr></code>
			<code><var></code>
<code><op1></code>	<code>::=</code>	<code>not</code>	
<code><op2></code>	<code>::=</code>	<code>and</code>	<code>or</code>
<code><var></code>	<code>::=</code>	<code>x</code>	<code>y</code> <code>z</code>

*Give a derivation of **not x and y or z** in the above grammar, both as a sequence of sentential forms and as a parse tree.*

*(In Python, if **x** and **y** and **z** are **True**, what does this expression evaluate to?)*

Answer

$\langle \text{expr} \rangle$

$\langle \text{expr} \rangle \langle \text{op2} \rangle \langle \text{expr} \rangle$

$\langle \text{expr} \rangle \text{ and } \langle \text{expr} \rangle$

$\langle \text{op1} \rangle \langle \text{expr} \rangle \text{ and } \langle \text{expr} \rangle$

not $\langle \text{expr} \rangle$ and $\langle \text{expr} \rangle$

not $\langle \text{var} \rangle$ and $\langle \text{expr} \rangle$

not x and $\langle \text{expr} \rangle$

not x and $\langle \text{expr} \rangle \langle \text{op2} \rangle \langle \text{expr} \rangle$

not x and $\langle \text{expr} \rangle$ or $\langle \text{expr} \rangle$

← FILL IN

$\langle \text{expr} \rangle$	$::=$	$\langle \text{op1} \rangle \langle \text{expr} \rangle$
	$ $	$\langle \text{expr} \rangle \langle \text{op2} \rangle \langle \text{expr} \rangle$
	$ $	$\langle \text{var} \rangle$
$\langle \text{op1} \rangle$	$::=$	not
$\langle \text{op2} \rangle$	$::=$	and $ $ or
$\langle \text{var} \rangle$	$::=$	x $ $ y $ $ z

$((\text{not } x) \text{ and } y) \text{ or } z$

not x and $(y \text{ or } z)$

The Big Picture

When we specify a PL (e.g., in the projects)
you will be given a *BNF grammar*

You will need to know how to translate this
into a parser

So you will need *practice reading BNF
specifications*

Ambiguity

Ambiguity in Natural Language

The duck is ready for dinner.

John saw the man on the mountain with a telescope.

He said on Tuesday there would be an exam.

Ambiguity in Natural Language

The duck is ready for dinner.

John saw the man on the mountain with a telescope.

He said on Tuesday there would be an exam.

Natural language has ambiguities that can confuse the meaning of a sentence.

Ambiguity in Natural Language

The duck is ready for dinner.

John saw the man on the mountain with a telescope.

He said on Tuesday there would be an exam.

Natural language has ambiguities that can confuse the meaning of a sentence.

We have informal tactics for avoiding these pitfalls.

Ambiguity in Natural Language

*The duck is ready **to eat** dinner.*

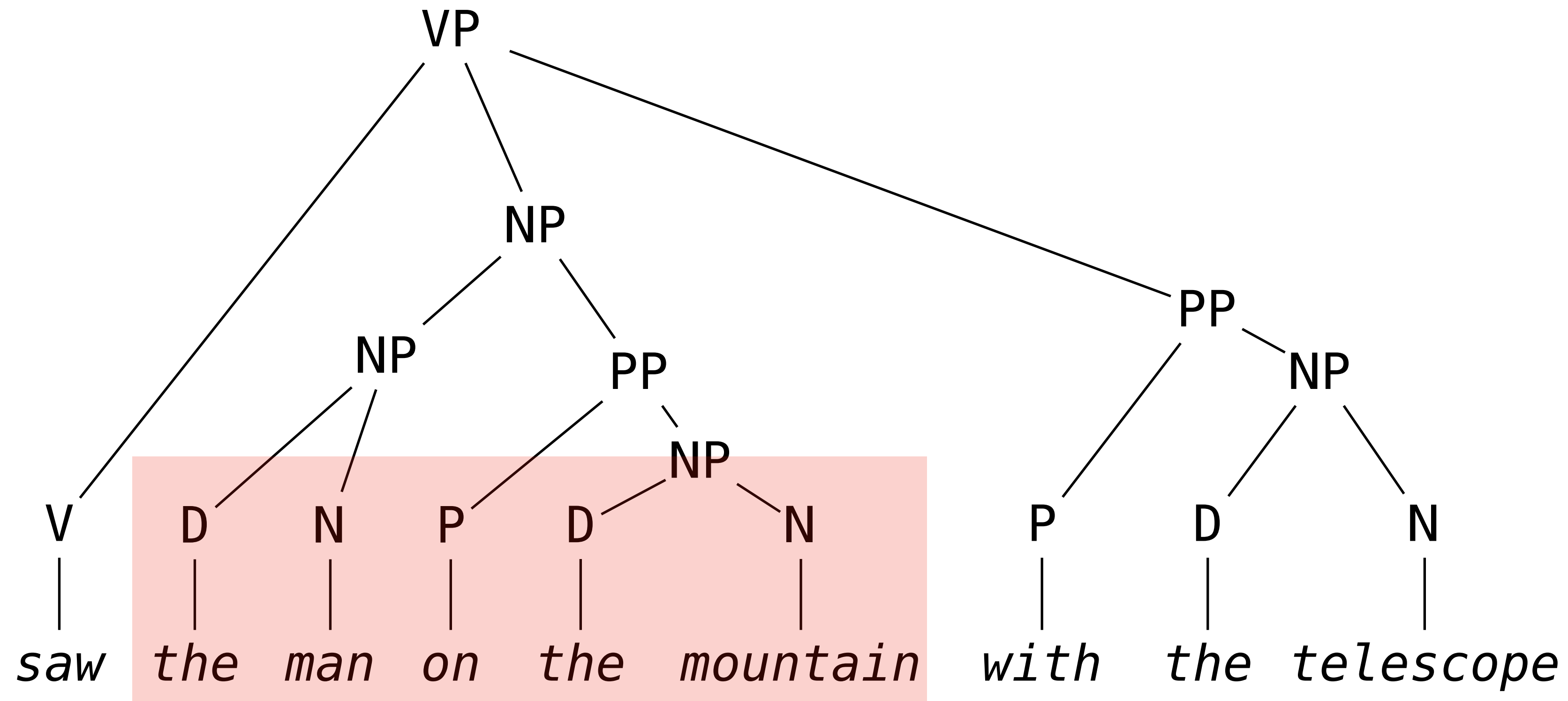
*John saw the man on the mountain **using** a telescope.*

*He said the exam would **be held** on Tuesday.*

Natural language has ambiguities that can **confuse the meaning** of a sentence.

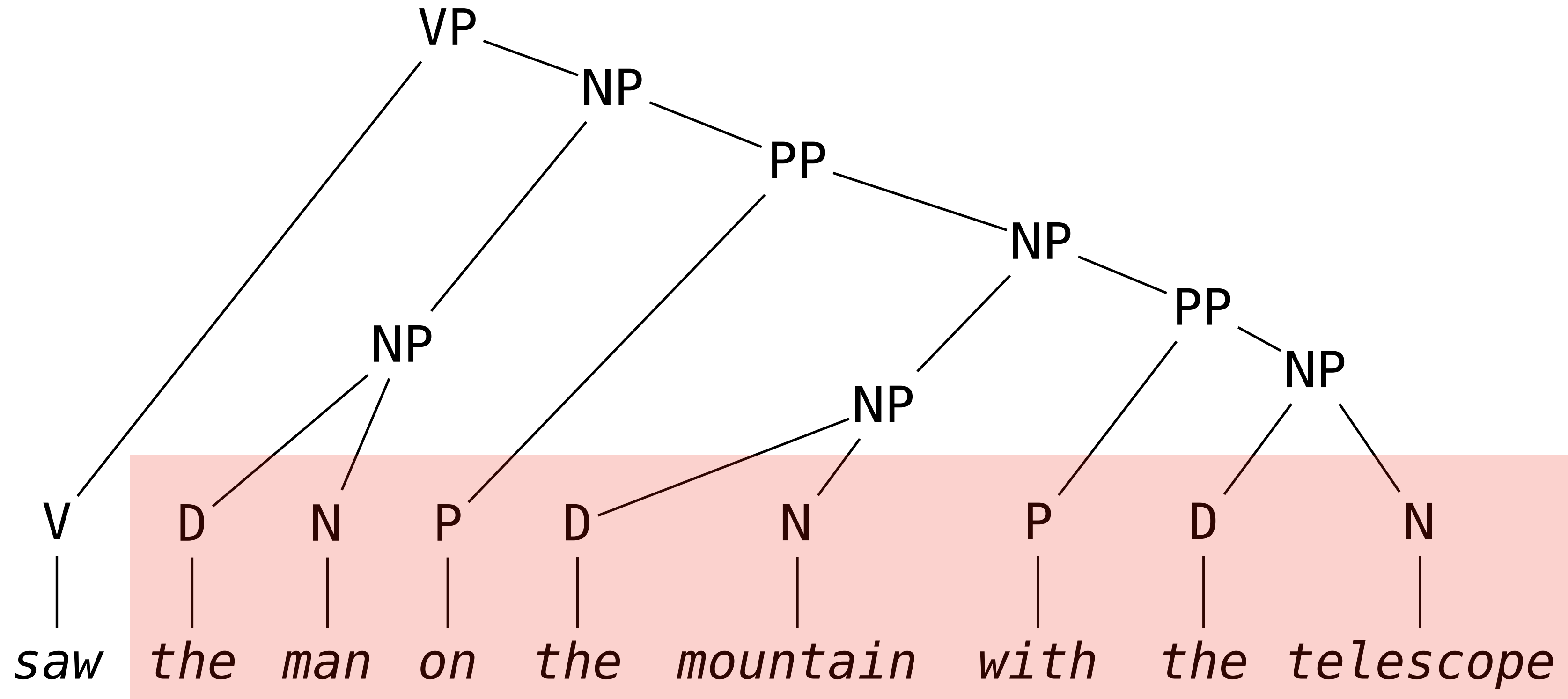
We have informal **tactics** for avoiding these pitfalls.

Aside: Ambiguity and Linearity



Ambiguity is caused by writing down
hierarchical structures in a **linear** fashion.

Aside: Ambiguity and Linearity



There is **no ambiguity** in the grammatical parse tree of this statement.

*The hierarchical structure changes
the meaning of the sentence*

Ambiguity in Formal Grammar

Ambiguity in Formal Grammar

Definition. A BNF grammar is **ambiguous** if there is a sentence with multiple parse trees/leftmost derivations.

Ambiguity in Formal Grammar

Definition. A BNF grammar is **ambiguous** if there is a sentence with multiple parse trees/leftmost derivations.

```
<expr> ::= <expr><op><expr>
          | <var>
<op>    ::= +
<var>   ::= x | y | z
```

Ambiguity in Formal Grammar

Definition. A BNF grammar is **ambiguous** if there is a sentence with multiple parse trees/leftmost derivations.

```
<expr> ::= <expr><op><expr>
          | <var>
<op>    ::= +
<var>   ::= x | y | z
```

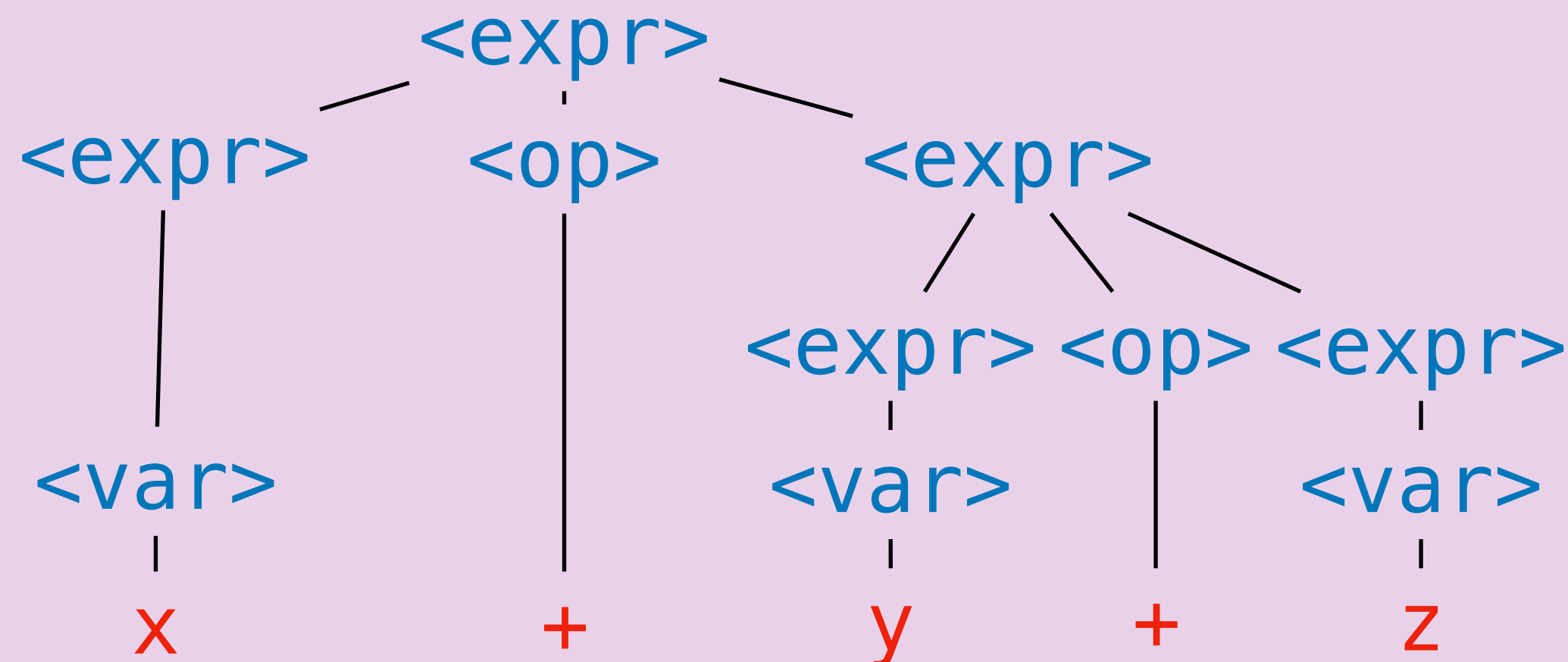
x + y + z can be derived as

Ambiguity in Formal Grammar

Definition. A BNF grammar is **ambiguous** if there is a sentence with multiple parse trees/leftmost derivations.

```
<expr> ::= <expr><op><expr>
          | <var>
<op>    ::= +
<var>   ::= x | y | z
```

x + y + z can be derived as

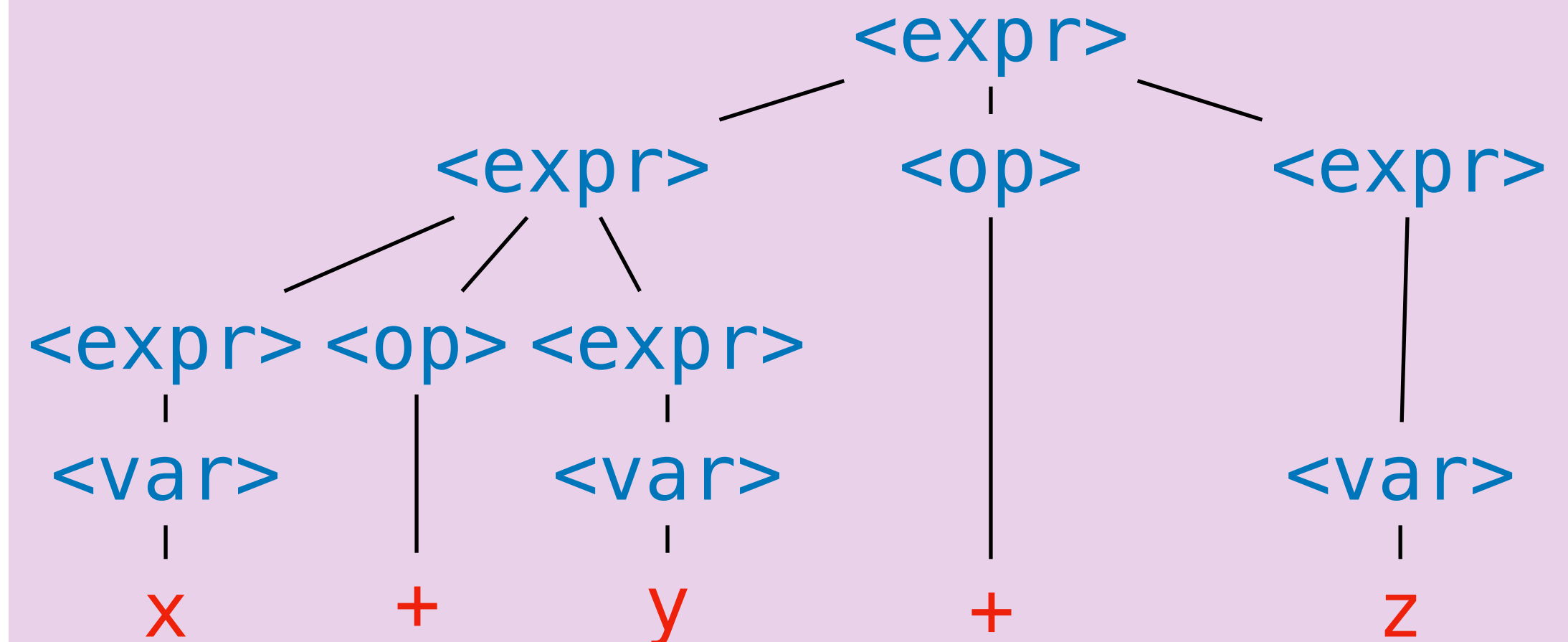
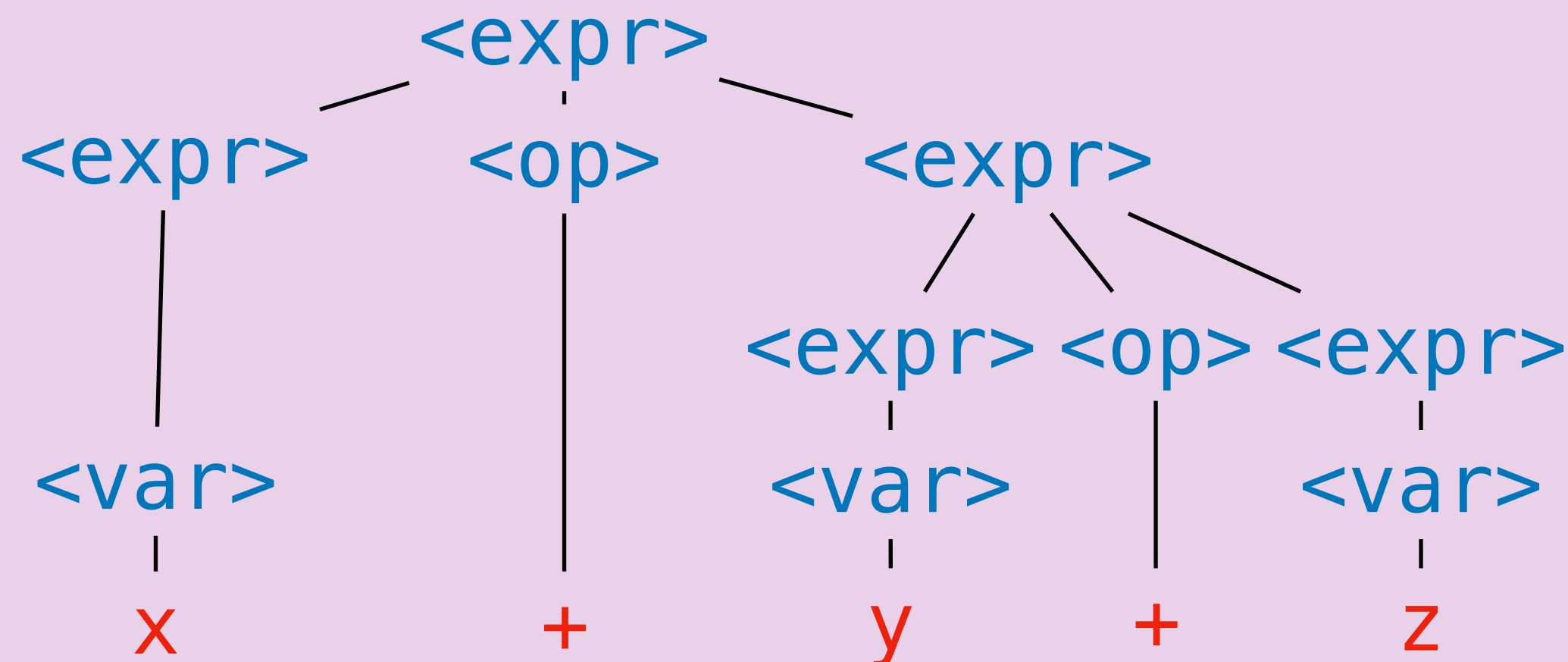


Ambiguity in Formal Grammar

Definition. A BNF grammar is **ambiguous** if there is a sentence with multiple parse trees/leftmost derivations.

```
<expr> ::= <expr><op><expr>
          | <var>
<op>    ::= +
<var>   ::= x | y | z
```

x + y + z can be derived as



Again, why do we care?

```
false && destory_everything ( ) || false
```

Again, why do we care?

```
false && destory_everything ( ) || false
```

Note that `1 + 1 + 1` is not ambiguous with respect to its *meaning* (it's value is `3` according to the standard definition of `+`)

Again, why do we care?



```
false && destory_everything () || false
```

Note that **1 + 1 + 1** is not ambiguous with respect to its *meaning* (it's value is **3** according to the standard definition of **+**)

But we make a **promise** to the user of a language that we won't make any **unspoken assumptions** about what they meant when they wrote down their program.

Practice Problem

```
<expr> ::= <bool>
          | <var>
          | if <expr> then <expr>
          | if <expr> then <expr> else <expr>
<bool>  ::= tru | fls
<var>   ::= x  | y  | z
```

Show that the above grammar is ambiguous.

Answer

```
<expr> ::= <bool>
          | <var>
          | if <expr> then <expr>
          | if <expr> then <expr> else <expr>
<bool> ::= tru | fls
<var>  ::= x | y | z
```

<expr>

if <expr> then <expr>

if <bool> then <e>

if tru then <e>

if true then if <e> then <e> else <e>

⋮

FILL IN

if true then if tru then x else x

Fundamental Concern

*What can we do about
ambiguity?*

Aside: Ambiguity and Computability

```
let is_ambiguous(g : grammar) : bool = ???
```

Aside: Ambiguity and Computability

```
let is_ambiguous(g : grammar) : bool = ???
```

It is *impossible* to write a program which determines if a grammar is ambiguous.

Aside: Ambiguity and Computability

```
let is_ambiguous(g : grammar) : bool = ???
```

It is *impossible* to write a program which determines if a grammar is ambiguous.

Not just hard, but **literally impossible**.

Aside: Ambiguity and Computability

```
let is_ambiguous(g : grammar) : bool = ???
```

It is *impossible* to write a program which determines if a grammar is ambiguous.

Not just hard, but **literally impossible**.

That's not to say we can't determine that *particular* grammars are ambiguous.

Fixity

Fixity

The fixity of an operator refers to where an operator is placed relative to its arguments:

Fixity

The fixity of an operator refers to where an operator is placed relative to its arguments:

prefix $f\ x\ ,\ (-\ x)$

Fixity

The fixity of an operator refers to where an operator is placed relative to its arguments:

prefix $f\ x\ ,\ (-\ x)$

postfix $a!\ (\text{get from ref})$

Fixity

The fixity of an operator refers to where an operator is placed relative to its arguments:

prefix $f\ x\ ,\ (-\ x)$

postfix $a!\ (\text{get from ref})$

infix $a\ *\ b,\ a\ +\ b,\ a\ \text{mod}\ b$

Fixity

The fixity of an operator refers to where an operator is placed relative to its arguments:

prefix $f\ x, (-\ x)$

postfix $a!\ (\text{get from ref})$

infix $a\ *\ b, a\ +\ b, a\ \text{mod}\ b$

mixfix $\text{if}\ b\ \text{then}\ x\ \text{else}\ y$

Polish Notation

$- \ / \ + \ 2 \ * \ 1 \ - \ 2 \ 3$

is equivalent to

$-(2 + (1 * (-2) / 3))$



To avoid ambiguity, we can make **all** operators **prefix** (or postfix) operators. *We don't even need parentheses.*

(This how early calculators worked.)

Example

```
<expr> ::= <bool>
          | <var>
          | ifthen <expr> <expr>
          | ifthenelse <expr> <expr> <expr>
<bool> ::= tru | fls
<var>  ::= x | y | z
```

No more ambiguity. But programs written like this are notoriously difficult to read...

Lots of Parentheses

<code><expr></code>	<code>::=</code>	<code>(<op1> <expr>)</code>
		<code> </code> <code>(<expr> <op2> <expr>)</code>
		<code> </code> <code><var></code>
<code><op1></code>	<code>::=</code>	<code>not</code>
<code><op2></code>	<code>::=</code>	<code>and or</code>
<code><var></code>	<code>::=</code>	<code>x y z</code>

Lots of Parentheses

<code><expr></code>	<code>::=</code>	<code>(<op1> <expr>)</code>
		<code> </code>
		<code>(<expr> <op2> <expr>)</code>
		<code> </code>
		<code><var></code>
<code><op1></code>	<code>::=</code>	<code>not</code>
<code><op2></code>	<code>::=</code>	<code>and or</code>
<code><var></code>	<code>::=</code>	<code>x y z</code>

If we want infix operators, we *could* add parentheses around all operators.

Lots of Parentheses

<code><expr></code>	<code>::=</code>	<code>(<op1> <expr>)</code> <code> </code> <code>(<expr> <op2> <expr>)</code> <code> </code> <code><var></code>	<code>((not x) and (not (not y)) or z)</code> <code>((((x or y) or z) or x) or y)</code>
<code><op1></code>	<code>::=</code>	<code>not</code>	<code>(not ((not x) and (not y)) or</code> <code> (x and z))</code>
<code><op2></code>	<code>::=</code>	<code>and or</code>	
<code><var></code>	<code>::=</code>	<code>x y z</code>	<code>(x and y)</code>

If we want infix operators, we *could* add parentheses around all operators.

But we run into a similar issue: *Too many parentheses are difficult to read.*

*Can we get away without
(or with fewer)
parentheses?*

Aside: The Cult of Parentheses

```
(defun fib (n)
  "Return the nth Fibonacci number."
  (if (< n 2)
      n
      (+ (fib (- n 1))
          (fib (- n 2))))))
```

Two Ingredients (or Flavors of Ambiguity)

Two Ingredients (or Flavors of Ambiguity)

Associativity:

How should arguments be grouped in an expression
like $1 + 2 + 3 + 4$?

Two Ingredients (or Flavors of Ambiguity)

Associativity:

How should arguments be grouped in an expression
like **1 + 2 + 3 + 4**?

Precedence:

How should arguments be grouped in an expression
like **1 + 2 * 3 + 4**?

Associativity

$$x \wedge (y \wedge z)$$

The associativity of an infix operator refers to how its arguments are grouped in the absence of parentheses:

left associative

$$1 + 2 + 3 \Rightarrow (1 + 2) + 3$$

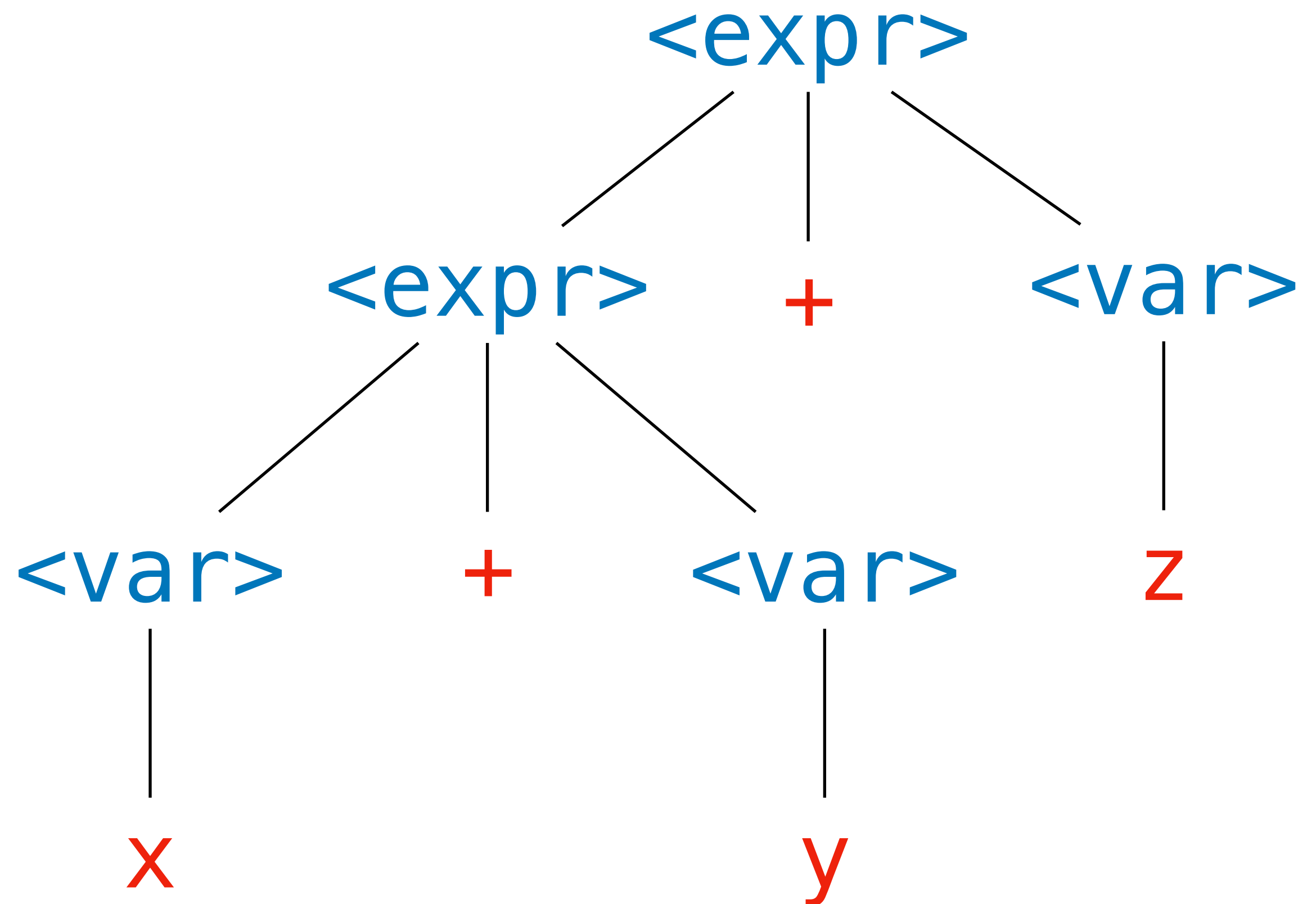
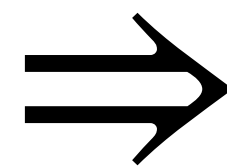
right associative

$$a \rightarrow b \rightarrow c \Rightarrow a \rightarrow (b \rightarrow c)$$

Associativity

<code><expr></code>	<code>::=</code>	<code><expr></code>	<code>+</code>	<code><expr></code>		
				<code><var></code>		
<code><var></code>	<code>::=</code>	<code>x</code>		<code>y</code>		<code>z</code>

`x + y + z`



"add the sum of x and y to z"

The Culprit

<code><expr></code>	<code>::=</code>	<code><expr></code>	<code>+</code>	<code><expr></code>		
		<code> </code>		<code><var></code>		
<code><var></code>	<code>::=</code>	<code>x</code>	<code> </code>	<code>y</code>	<code> </code>	<code>z</code>

The Culprit

<code><expr></code>	<code>::=</code>	<code><expr> + <expr></code>
	<code> </code>	<code><var></code>
<code><var></code>	<code>::=</code>	<code>x y z</code>

Any time we have a rule like this, we should be suspicious...

The Culprit

$\langle \text{expr} \rangle$	$::=$	$\langle \text{expr} \rangle + \langle \text{expr} \rangle$
	$ $	$\langle \text{var} \rangle$
$\langle \text{var} \rangle$	$::=$	$x \mid y \mid z$

Any time we have a rule like this, we should be suspicious...

$\langle \text{expr} \rangle + \langle \text{expr} \rangle \Rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle + \langle \text{expr} \rangle$

The Culprit

$\langle \text{expr} \rangle$	$::=$	$\langle \text{expr} \rangle + \langle \text{expr} \rangle$
	$ $	$\langle \text{var} \rangle$
$\langle \text{var} \rangle$	$::=$	$x \mid y \mid z$

Any time we have a rule like this, we should be suspicious...

$\langle \text{expr} \rangle + \langle \text{expr} \rangle \Rightarrow \langle \text{expr} \rangle + \langle \text{expr} \rangle + \langle \text{expr} \rangle$

Which $\langle \text{expr} \rangle$ did we replace?

Aside: Dealing with Associativity within the Grammar

$\langle \text{expr} \rangle$	$::=$	$\langle \text{expr} \rangle$	$+$	$\langle \text{var} \rangle$		
		$ $		$\langle \text{var} \rangle$		
$\langle \text{var} \rangle$	$::=$	x	$ $	y	$ $	z

$\langle \text{expr} \rangle$
 $\langle \text{expr} \rangle + \langle \text{var} \rangle$
 $\langle \text{expr} \rangle + z$
 $\langle \text{expr} \rangle + \langle \text{var} \rangle + z$
 $\langle \text{expr} \rangle + y + z$
 $\langle \text{var} \rangle + y + z$
 $x + y + z$

By enforcing that the second argument is a $\langle \text{var} \rangle$, we will get the left-associative parse tree.

Question. What about the right associative?

And Right Associativity

$\langle \text{type} \rangle$	$::=$	$\langle \text{base} \rangle \rightarrow \langle \text{type} \rangle$
		$\mid \langle \text{base} \rangle$
$\langle \text{base} \rangle$	$::=$	$()$

$\langle \text{type} \rangle$
 $\langle \text{base} \rangle \rightarrow \langle \text{type} \rangle$
 $() \rightarrow \langle \text{type} \rangle$
 $() \rightarrow \langle \text{base} \rangle \rightarrow \langle \text{type} \rangle$
 $() \rightarrow () \rightarrow \langle \text{type} \rangle$
 $() \rightarrow () \rightarrow \langle \text{base} \rangle$
 $() \rightarrow () \rightarrow ()$

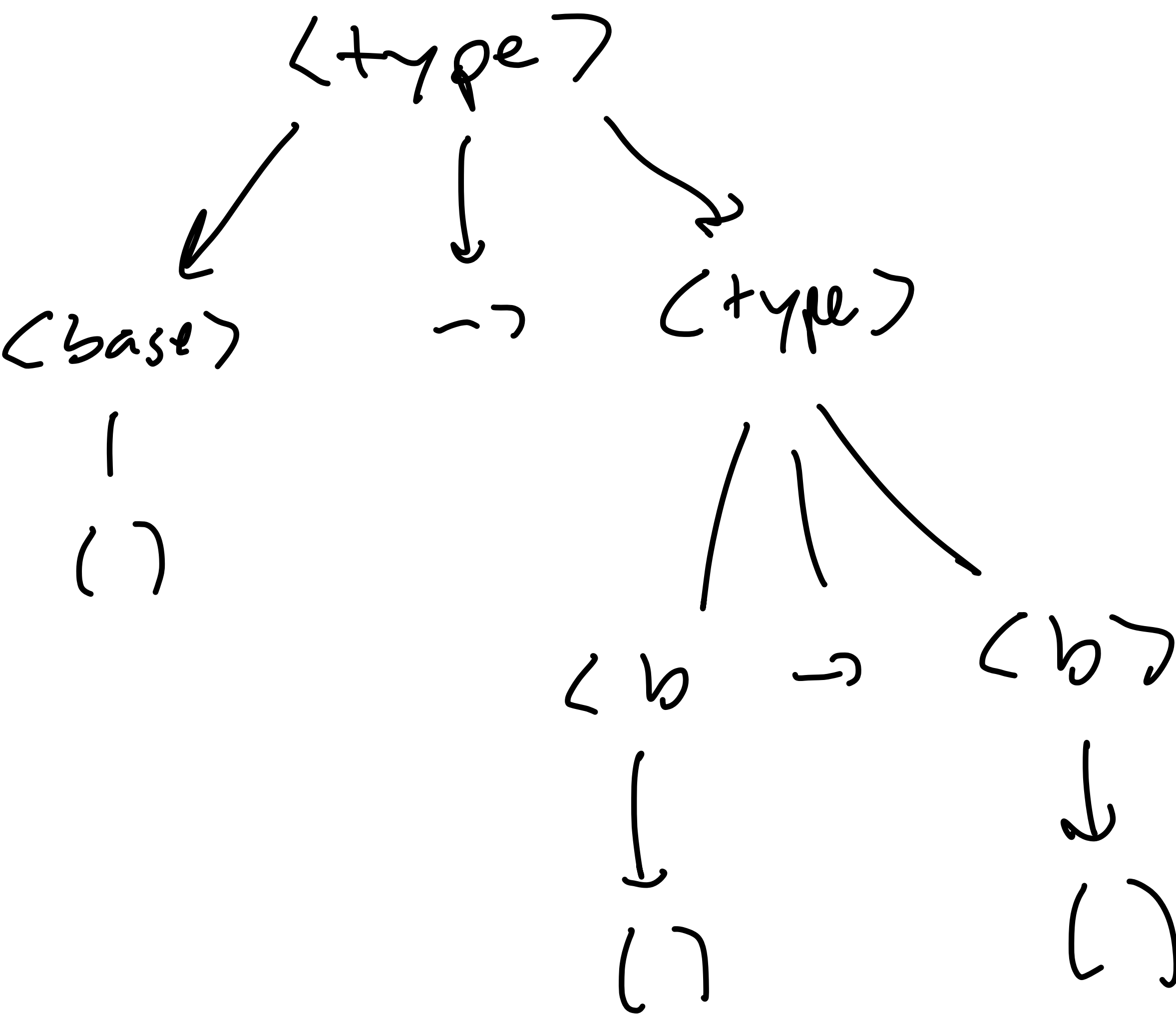
For right associativity, we break symmetry by "factoring out" the *left* argument.

Example Parse Tree

<type> ::= <base> -> <type>

| <base>

<base> ::= ()



<type>

<base> -> <type>

() -> <type>

() -> <base> -> <type>

() -> () -> <type>

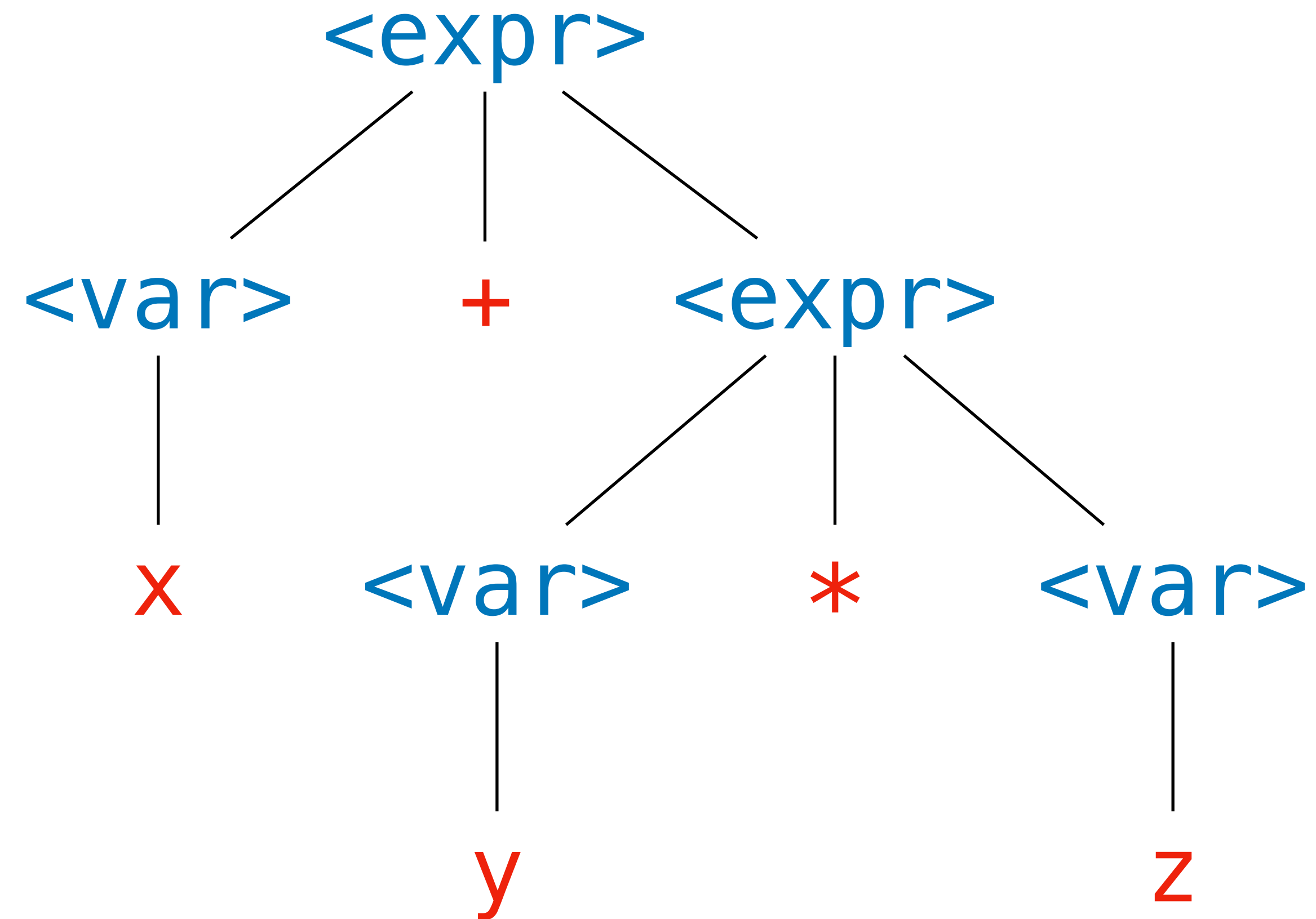
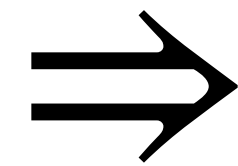
() -> () -> <base>

() -> () -> ()

Multiple Operators

<expr>	::=	<expr>	<op>	<var>
				<var>
<op>	::=	+		*
<var>	::=	x		y z

x + y * z



"add x to the product of y and z"

Question. What if we have multiple operators?
Which one should "bind tighter"?

Precedence

$$2 + 3 \times 6 = 2 + (3 \times 6) = 20$$

Precedence

$$2 + 3 \times 6 = 2 + (3 \times 6) = 20$$

The precedence of an operator refers to order in which an operator should be considered, relative to other operators.

Precedence

$$2 + 3 \times 6 = 2 + (3 \times 6) = 20$$

The precedence of an operator refers to order in which an operator should be considered, relative to other operators.

Example. **PEMDAS** (paren, exp, mul, div, add, sub)

Precedence

$$2 + 3 \times 6 = 2 + (3 \times 6) = 20$$

The precedence of an operator refers to order in which an operator should be considered, relative to other operators.

Example. **PEMDAS** (paren, exp, mul, div, add, sub)

Higher precedence means it "binds tighter".

Aside: Dealing with Precedence within the Grammar

<code><expr></code>	<code>::=</code>	<code><expr></code>	<code>+</code>	<code><term></code>		
		<code> </code>		<code><term></code>		
<code><term></code>	<code>::=</code>	<code><term></code>	<code>*</code>	<code><var></code>		
		<code> </code>		<code><var></code>		
<code><var></code>	<code>::=</code>	<code>x</code>	<code> </code>	<code>y</code>	<code> </code>	<code>z</code>

We factor out the `*` part of the `<expr>` rule.

Note that we handle *lower* precedence terms first, since terms *deeper* in the parse tree are evaluated first.

Understanding Check

<expr>	::=	<expr>	+	<term>
				<term>
<term>	::=	<term>	*	<var>
				<var>
<var>	::=	x		y z

*Write down the parse tree for **x + y * z**.*

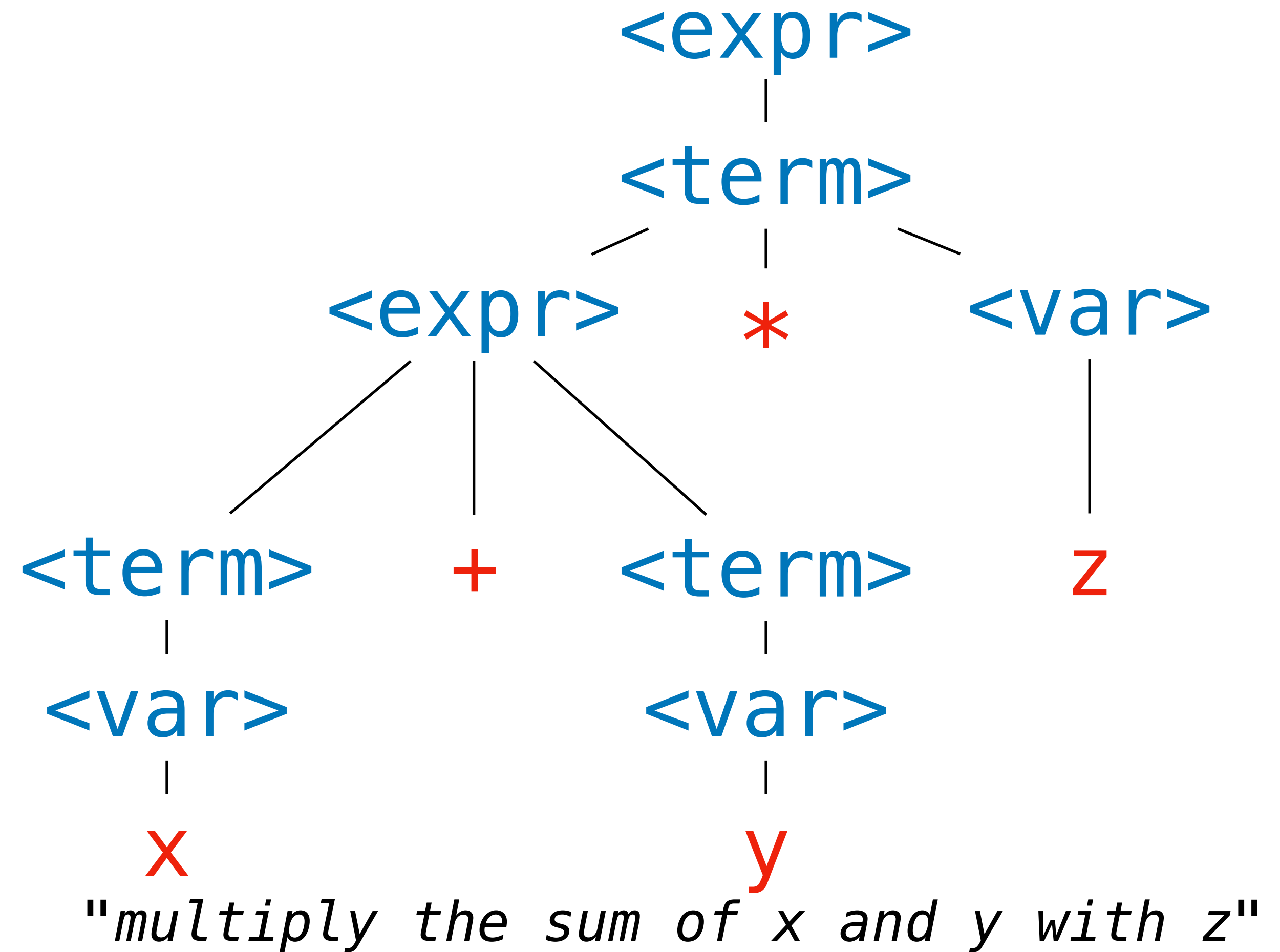
Answer

<expr>	::=	<expr>	+	<term>		
				<term>		
<term>	::=	<term>	*	<var>		
				<var>		
<var>	::=	x		y		z

x + y * z

The Issue of Parentheses Returns

<code><expr></code>	<code>::=</code>	<code><expr></code>	<code>+</code>	<code><term></code>		
		<code><term></code>				
<code><term></code>	<code>::=</code>	<code><term></code>	<code>*</code>	<code><var></code>		
		<code><var></code>				
<code><var></code>	<code>::=</code>	<code>x</code>	<code> </code>	<code>y</code>	<code> </code>	<code>z</code>

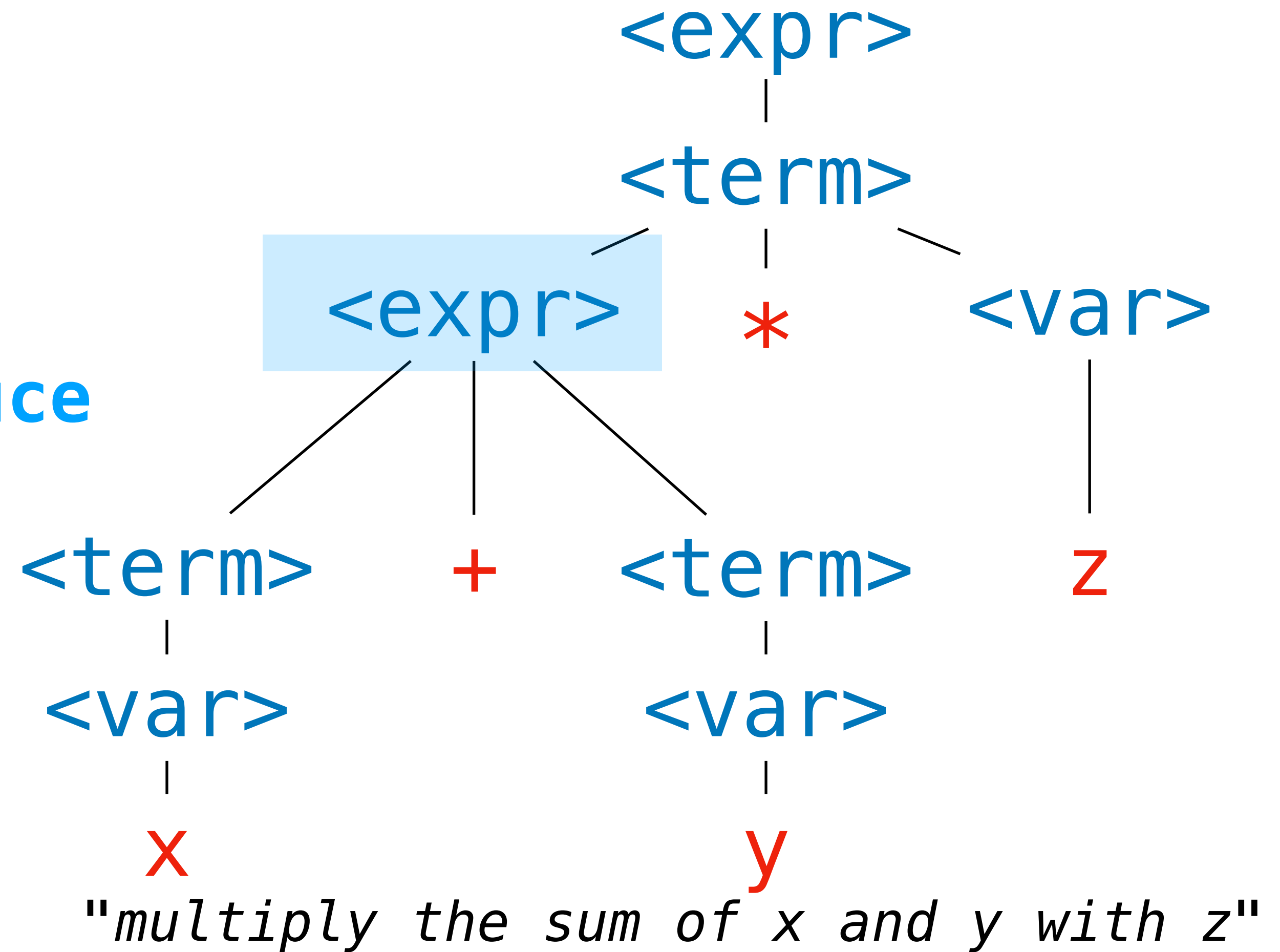


Question. Can we derive this parse tree?

The Issue of Parentheses Returns

<code><expr></code>	<code>::=</code>	<code><expr></code>	<code>+</code>	<code><term></code>		
		<code><term></code>				
<code><term></code>	<code>::=</code>	<code><term></code>	<code>*</code>	<code><var></code>		
		<code><var></code>				
<code><var></code>	<code>::=</code>	<code>x</code>	<code> </code>	<code>y</code>	<code> </code>	<code>z</code>

No, we need to introduce parentheses again.



Question. Can we derive this parse tree?

Dealing with Parentheses

<code><expr></code>	<code>::=</code>	<code><expr></code>	<code>+</code>	<code><term></code>
		<code> </code>		<code><term></code>
<code><term></code>	<code>::=</code>	<code><term></code>	<code>*</code>	<code><var></code>
		<code> </code>		<code><pars></code>
<code><pars></code>	<code>::=</code>	<code><var></code>	<code> </code>	<code>(<expr>)</code>
<code><var></code>	<code>::=</code>	<code>x</code>	<code> </code>	<code>y</code> <code> </code> <code>z</code>

We further factor out the part of the rule for parentheses. Note that **any expression** can appear in the parentheses.

(This is a circular, or mutually recursive, definition.)

Example

<expr>	::=	<expr>	+	<term>		
		 		<term>		
<term>	::=	<term>	*	<var>		
		 		<pars>		
<pars>	::=	<var>	 	(<expr>)		
<var>	::=	x	 	y	 	z

(x + y) * z

Other Considerations

There's a lot left to make a working grammar:

- » actual values (e.g., $(1 + 23) * 4$)
- » variable names (e.g., $valid_var + 33$)
- » multiple operations with the same precedence
(e.g. $1 + 3 - 2$)
- » multiple operations with different associativity (?)

Other Considerations

There's a lot left to make a working grammar:

- » actual values (e.g., $(1 + 23) * 4$)
- » variable names (e.g., $valid_var + 33$)
- » multiple operations with the same precedence (e.g. $1 + 3 - 2$)
- » multiple operations with different associativity (?)

This is what we will be doing when we build our interpreters.

Summary

To avoid ambiguity, we make choices beforehand about the **fixity**, **associativity** and **precedence**.

Determining ambiguity can be tricky, but usually possible for simple grammars.

We make the grammars of programming languages unambiguous so that we don't make **unspoken assumptions** about the users input.