Formal Grammar

Concepts of Programming Languages
Lecture 12

CAS CS 320

Outline

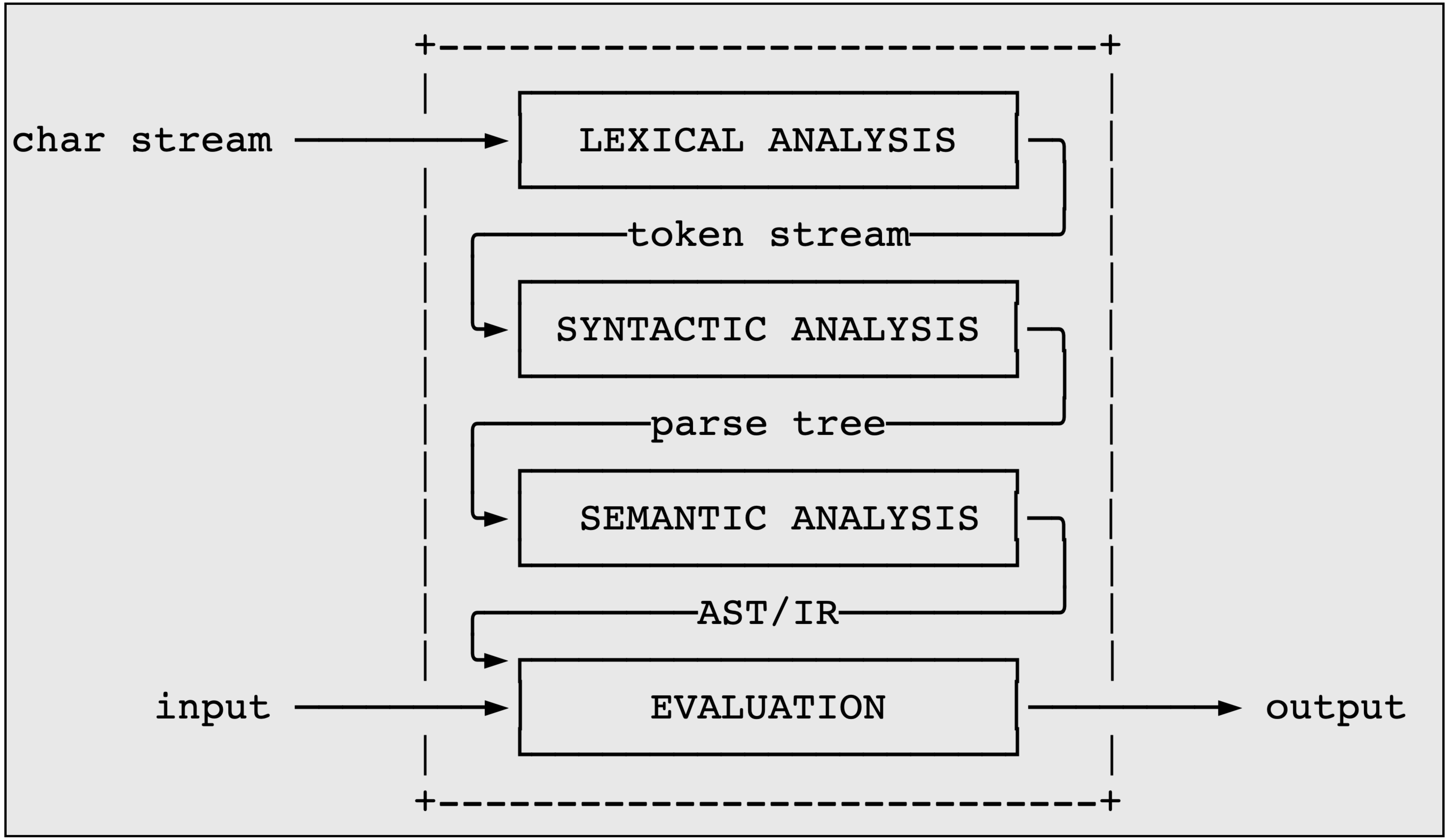
- » Discuss briefly the **interpretation pipeline**, and how it will look in the context of this course

- » Introduce **formal grammars**, a mathematical framework for thinking about syntax and parsing

- » Discuss **ambiguity** in grammar, and look at ways of avoiding it

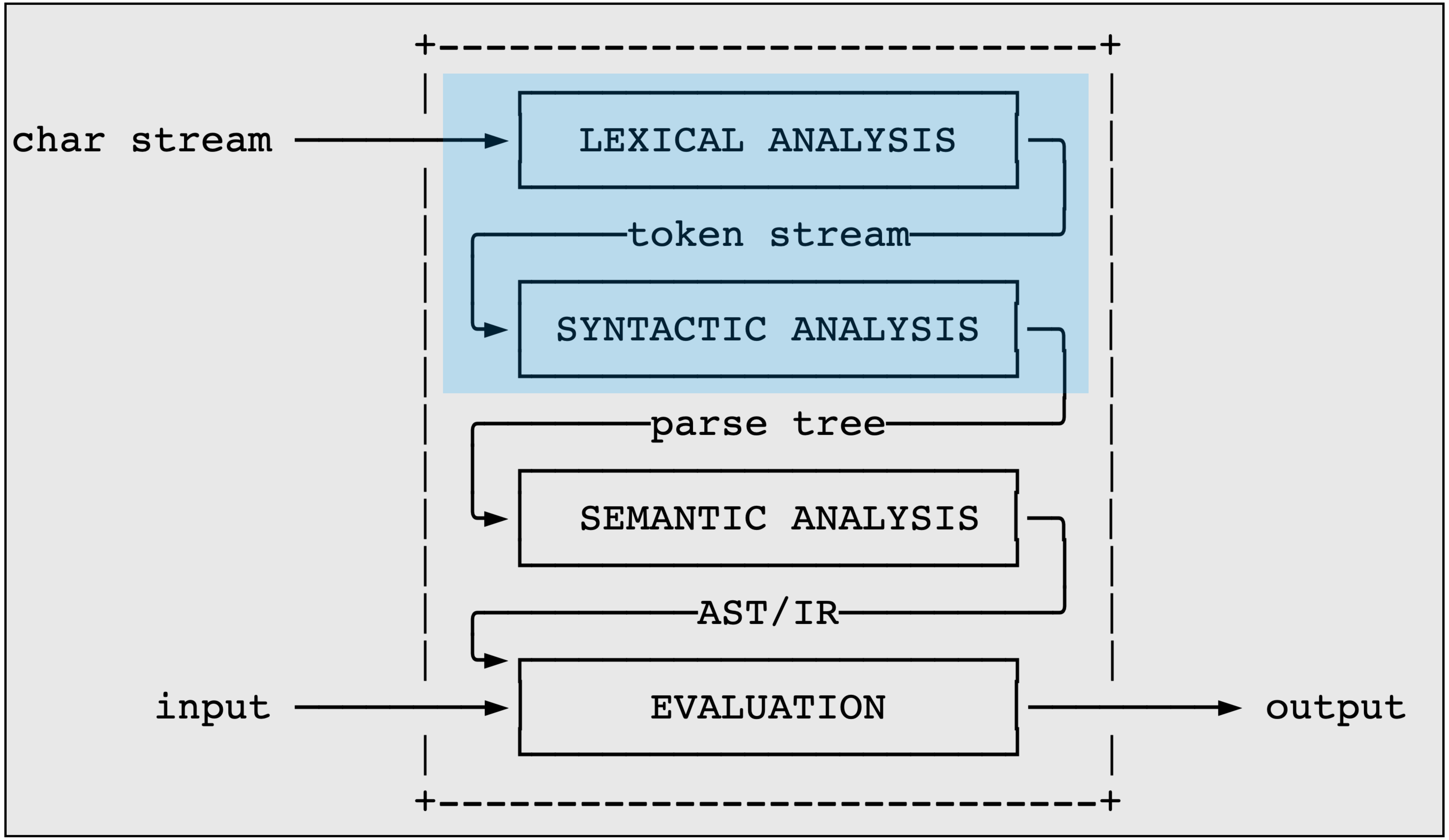
The Interpretation Pipeline

The Picture


char stream
LEXICAL ANALYSIS
token stream
SYNTACTIC ANALYSIS
parse tree
SEMANTIC ANALYSIS
AST/IR
input
EVALUATION
output

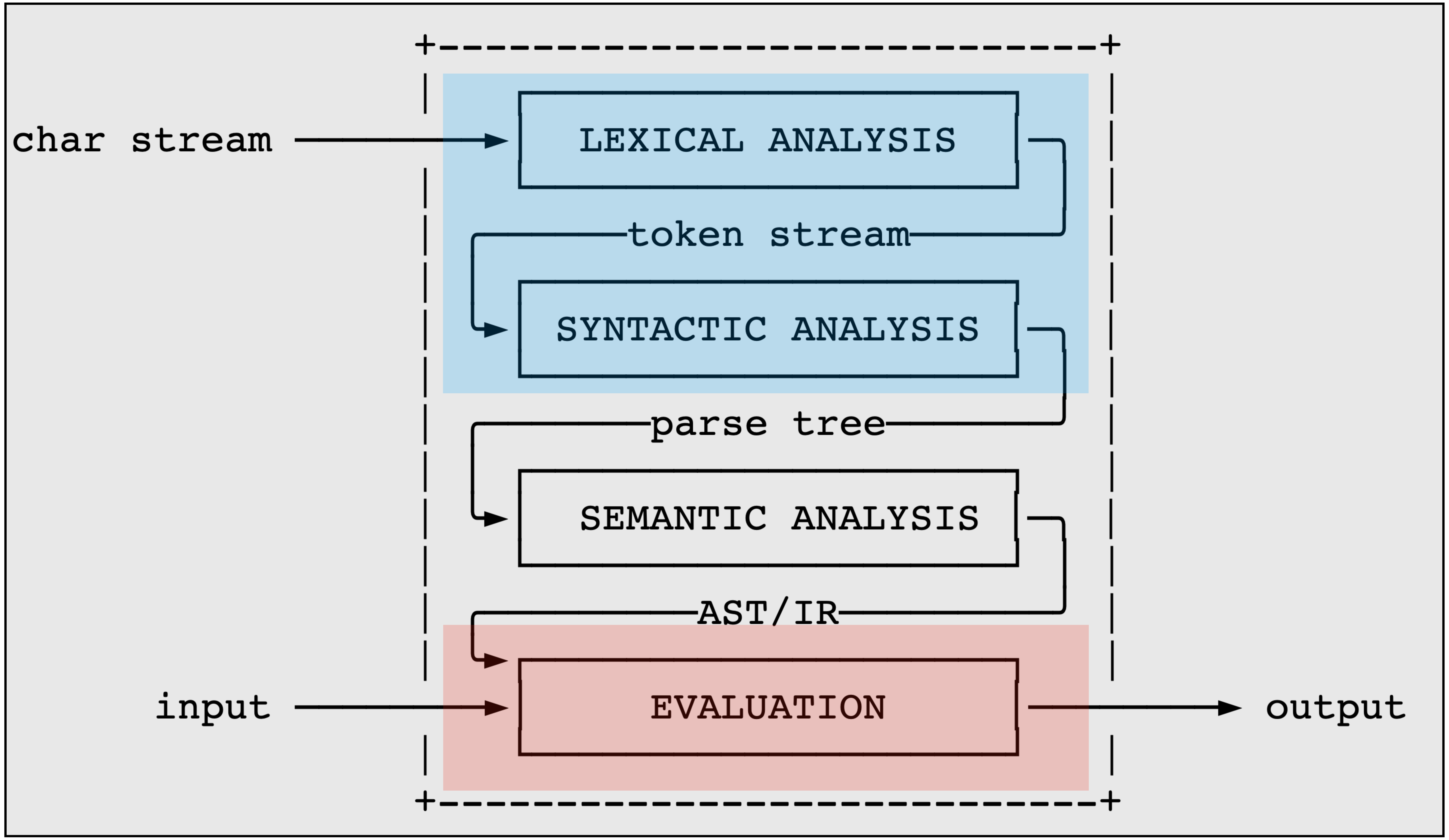
The Picture

parsing (this week)



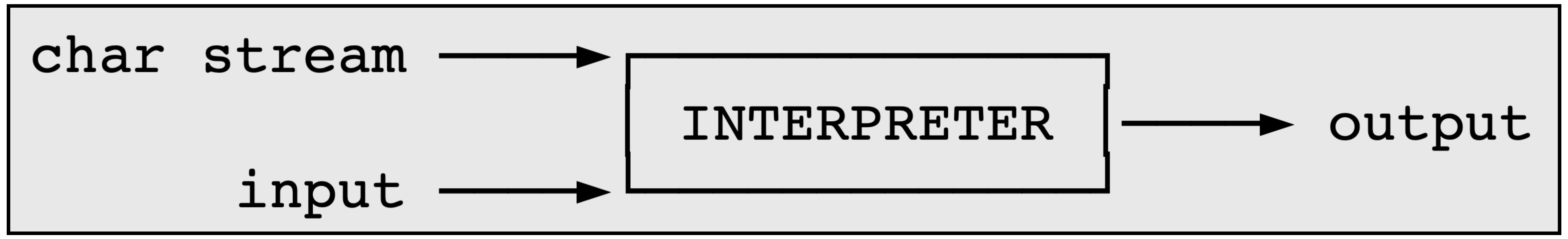
The Picture

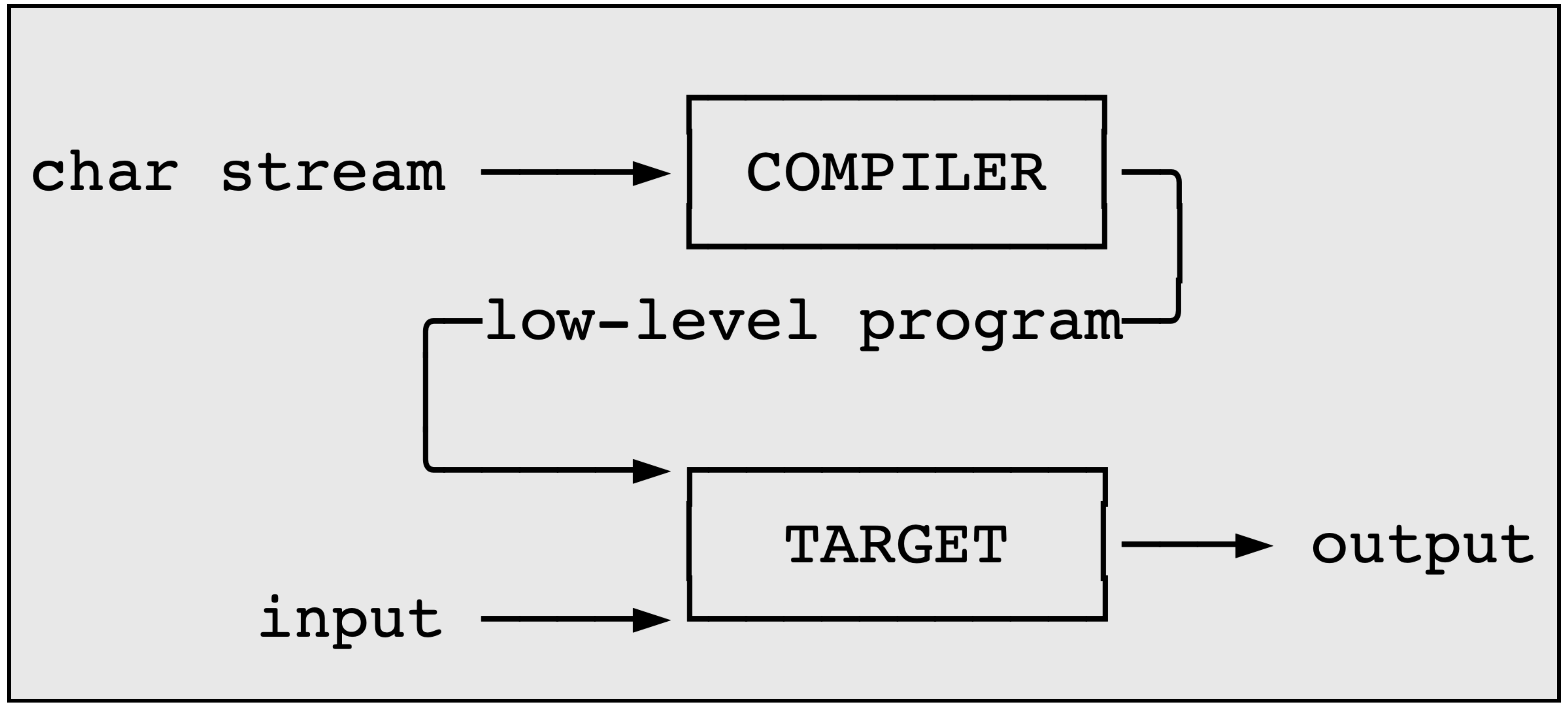
parsing (this week)



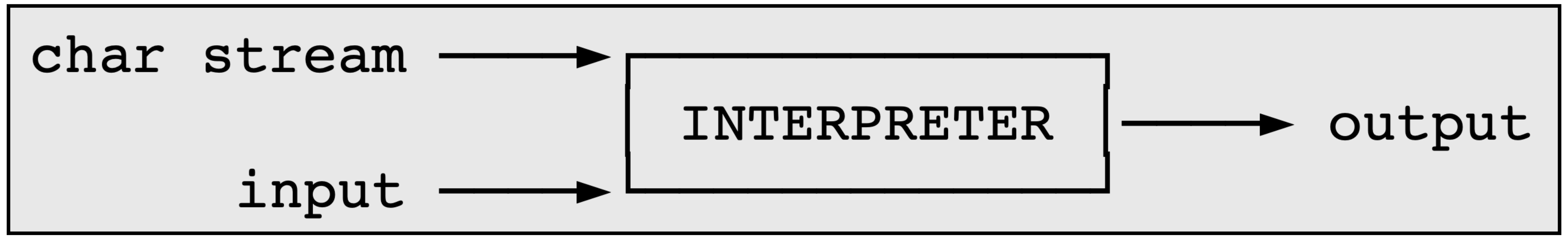
semantics (next week)

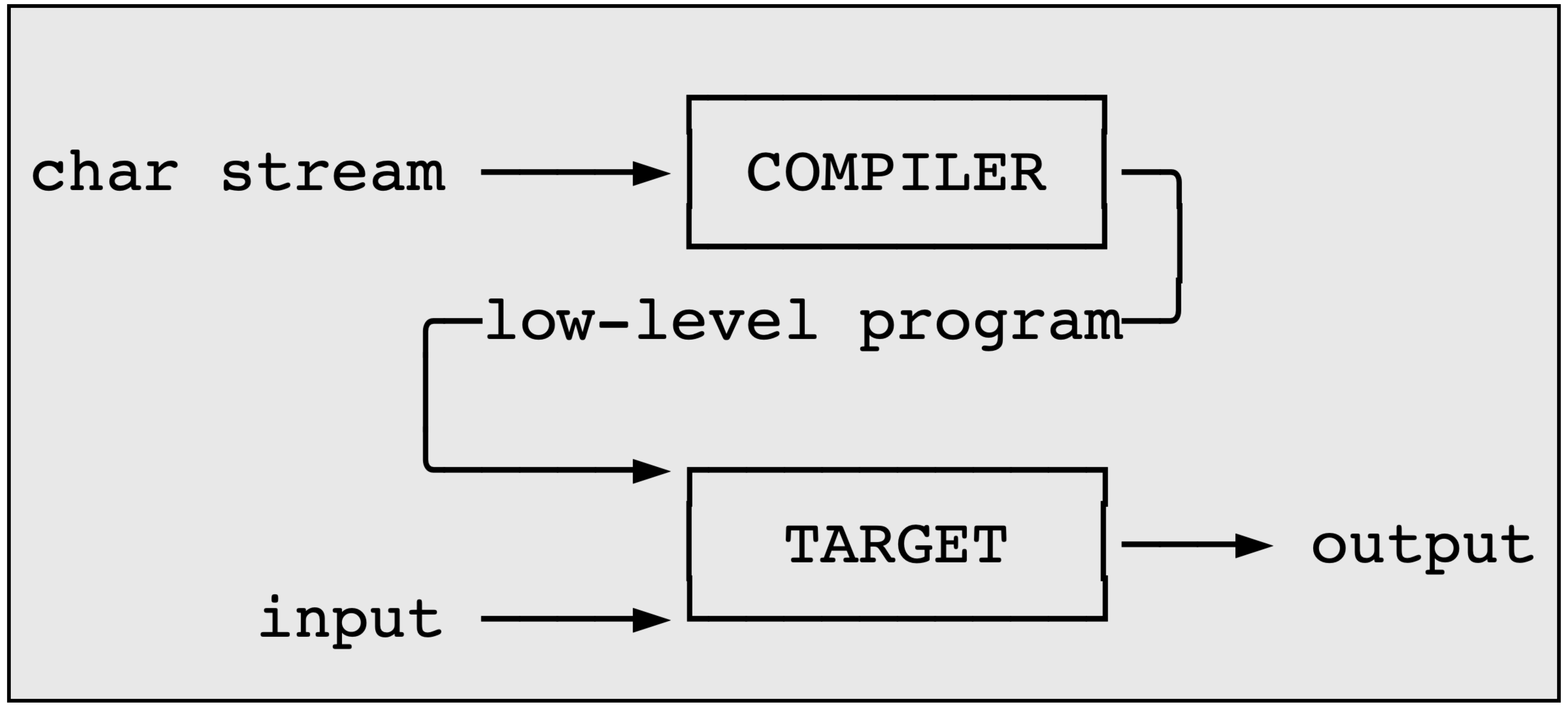
A Note on Compilation





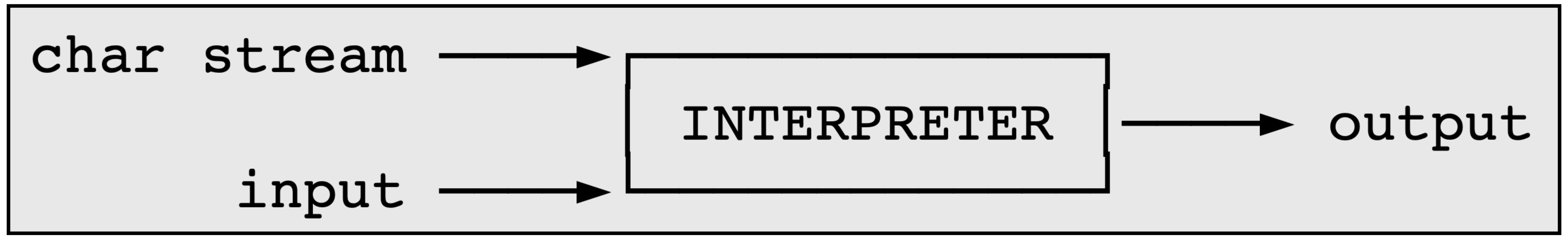
A Note on Compilation

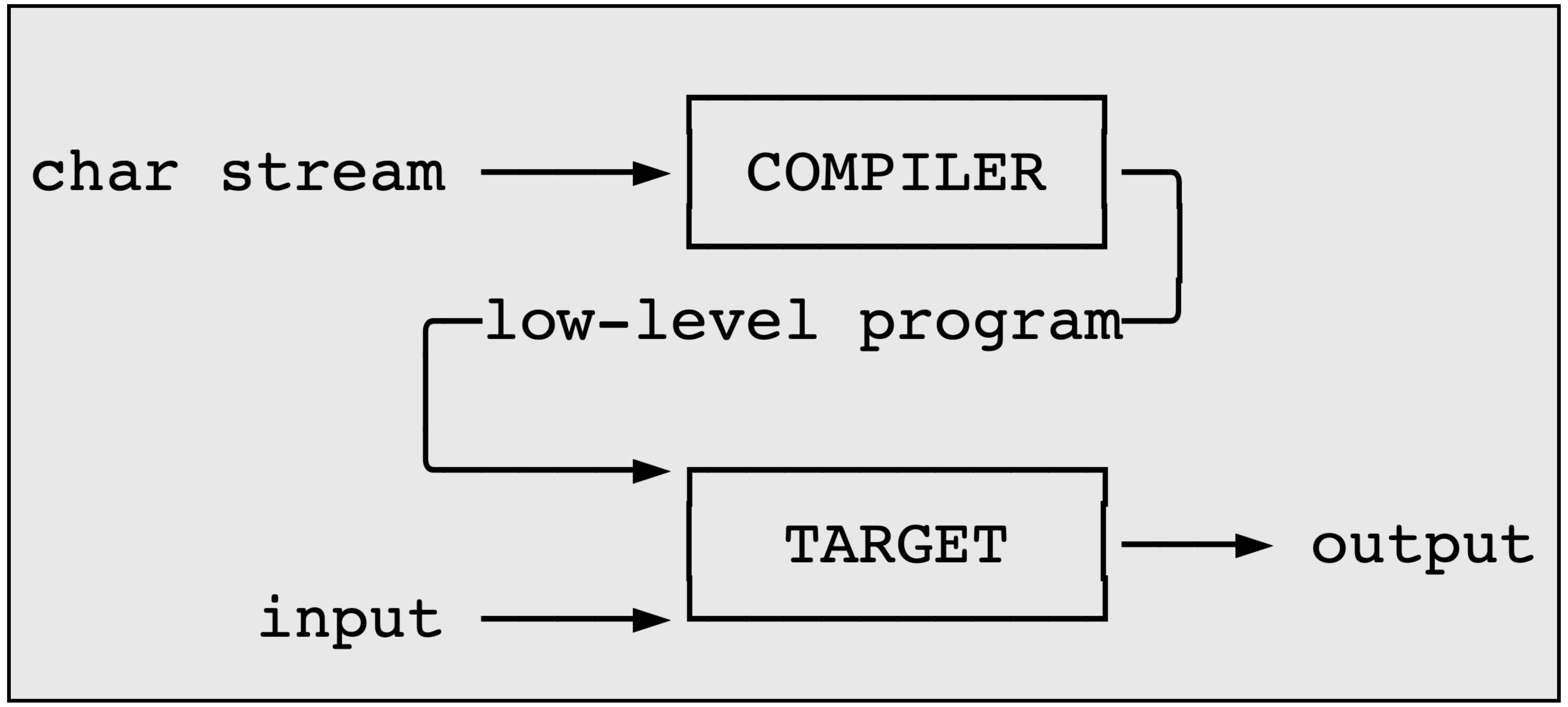




We will be building programs that directly read an evaluate programs (**interpreters**)

A Note on Compilation





We will be building programs that directly read an 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.

<u>You'll be given:</u>

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

Today

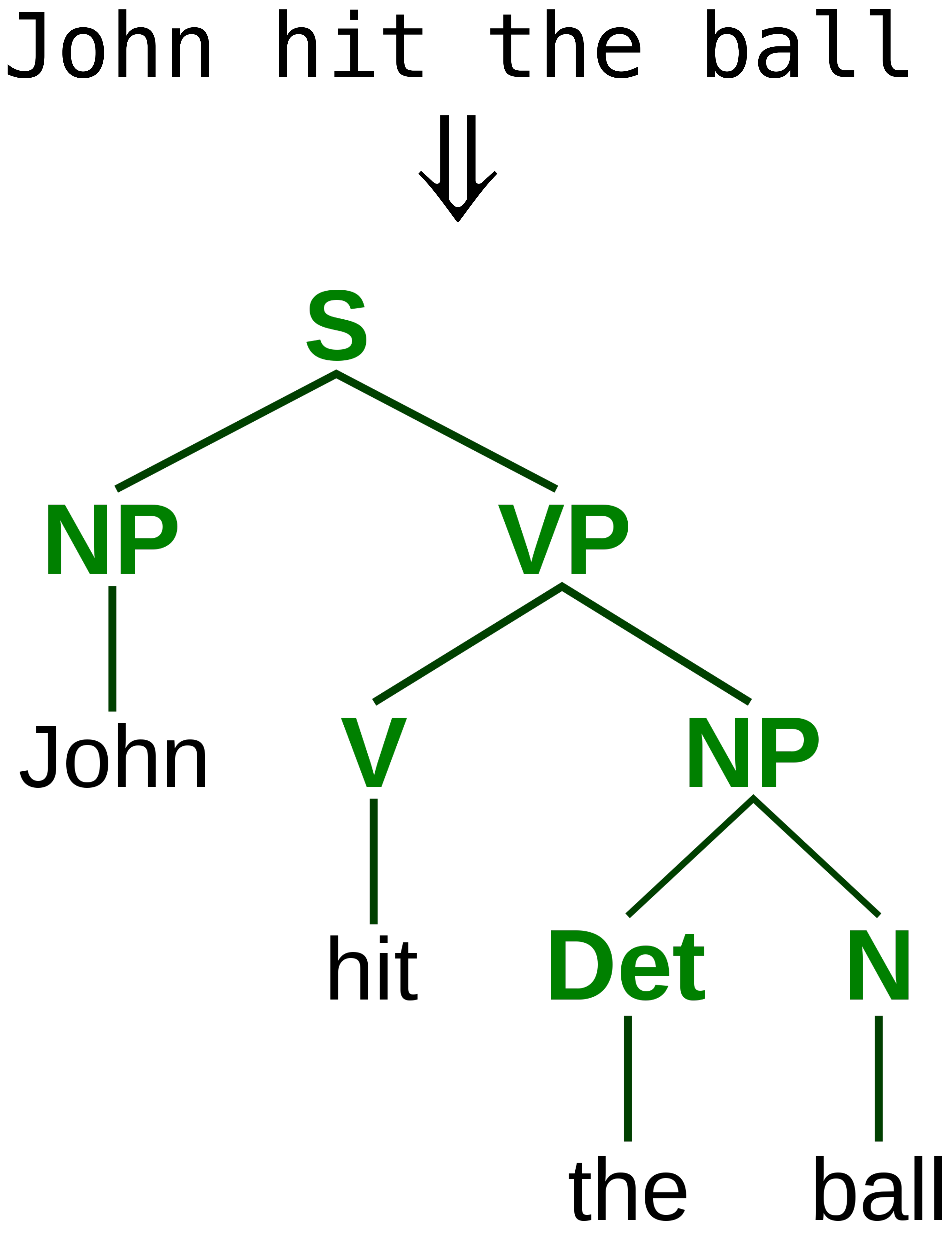
We need a formal language for describing the syntax of programming languages

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

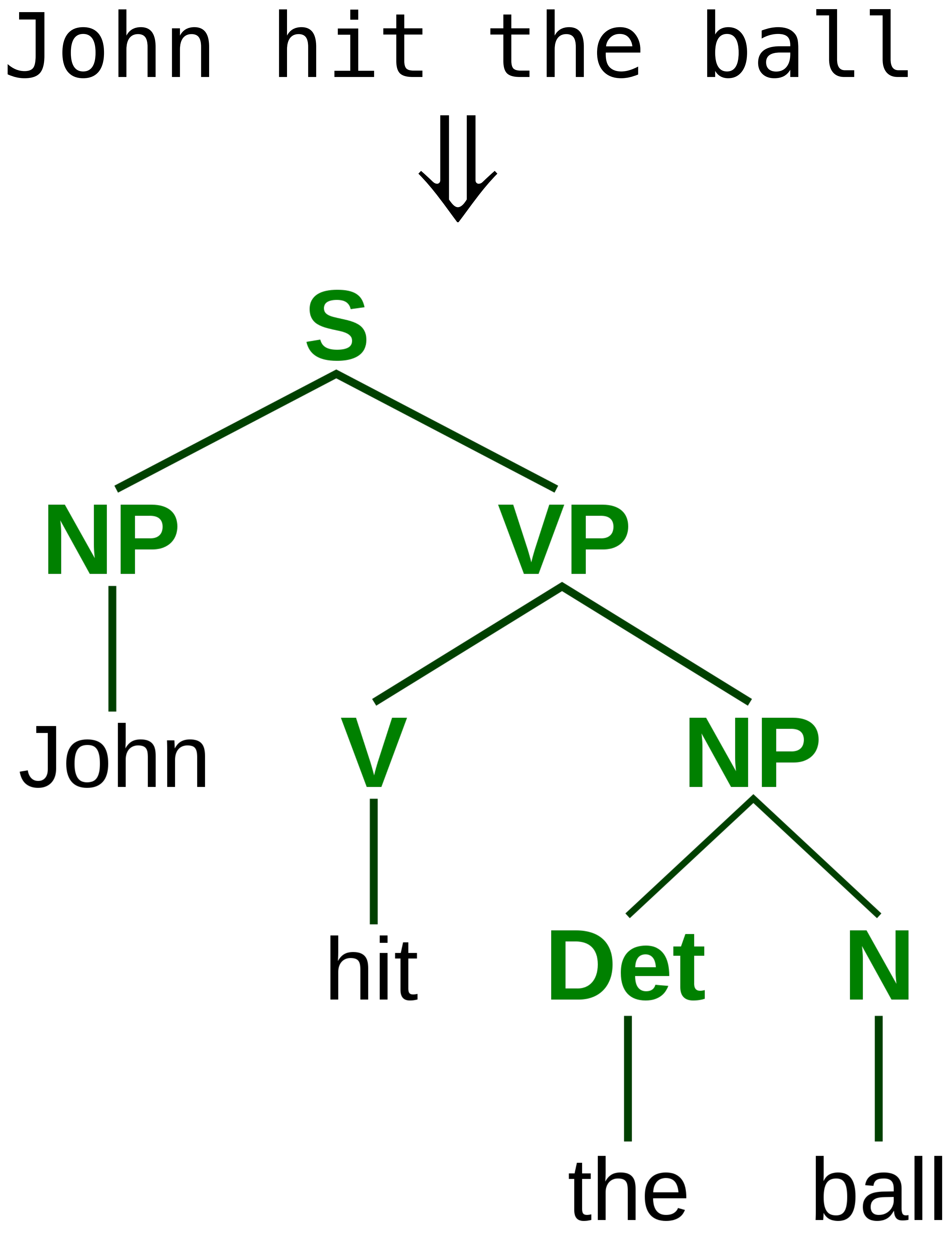
What is Grammar?



https://commons.wikimedia.org/wiki/File:ParseTree.svg

What is Grammar?

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

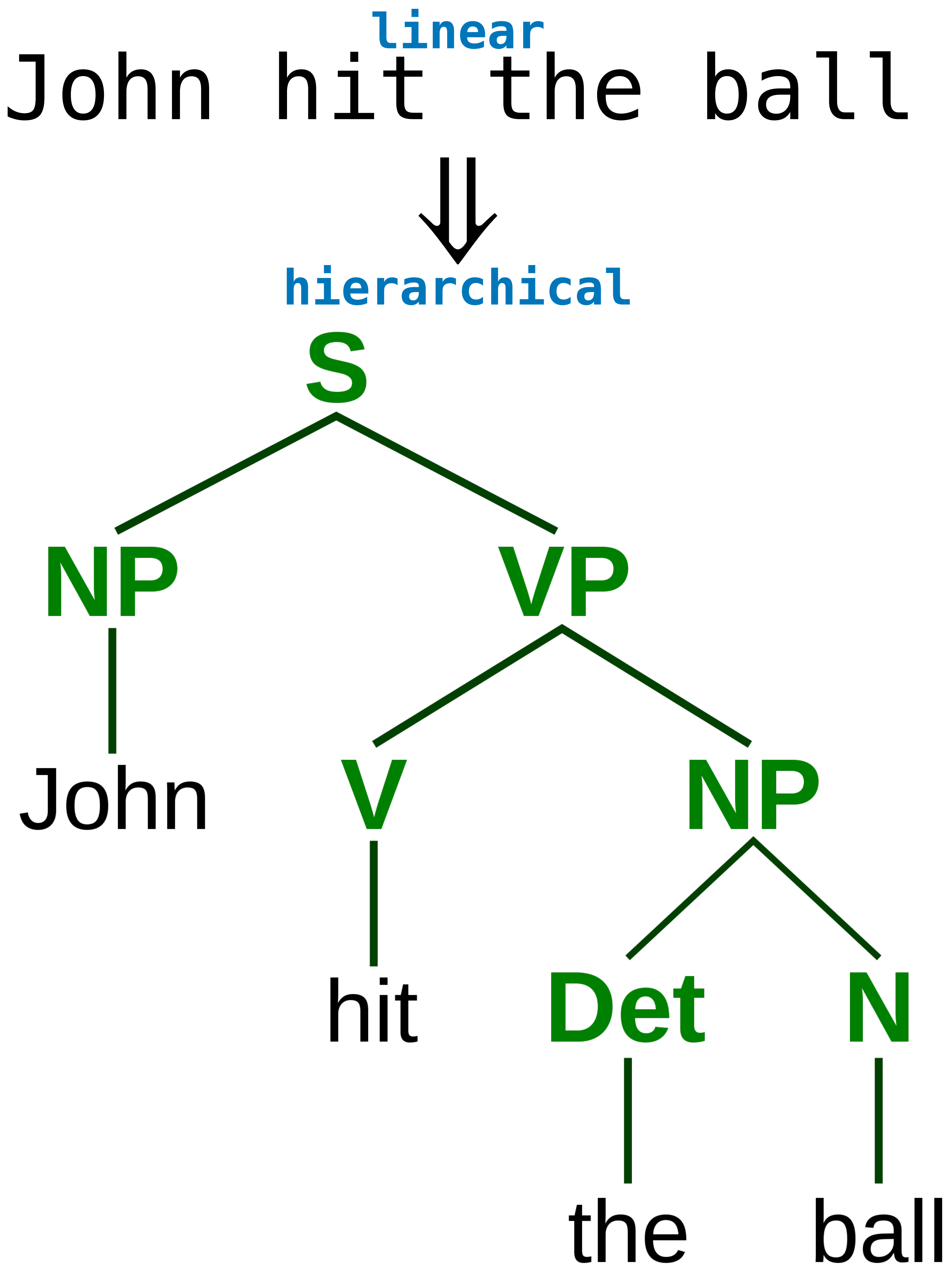


https://commons.wikimedia.org/wiki/File:ParseTree.svg

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



https://commons.wikimedia.org/wiki/File:ParseTree.svg

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

<u>Formal grammars</u> for PL
tell us which programs
are well-formed

Grammars for Programming Languages

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

<u>Formal grammars</u> for PL tell us which programs are well-formed

Grammars for Programming Languages

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

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

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

<u>Formal grammars</u> 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
```

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

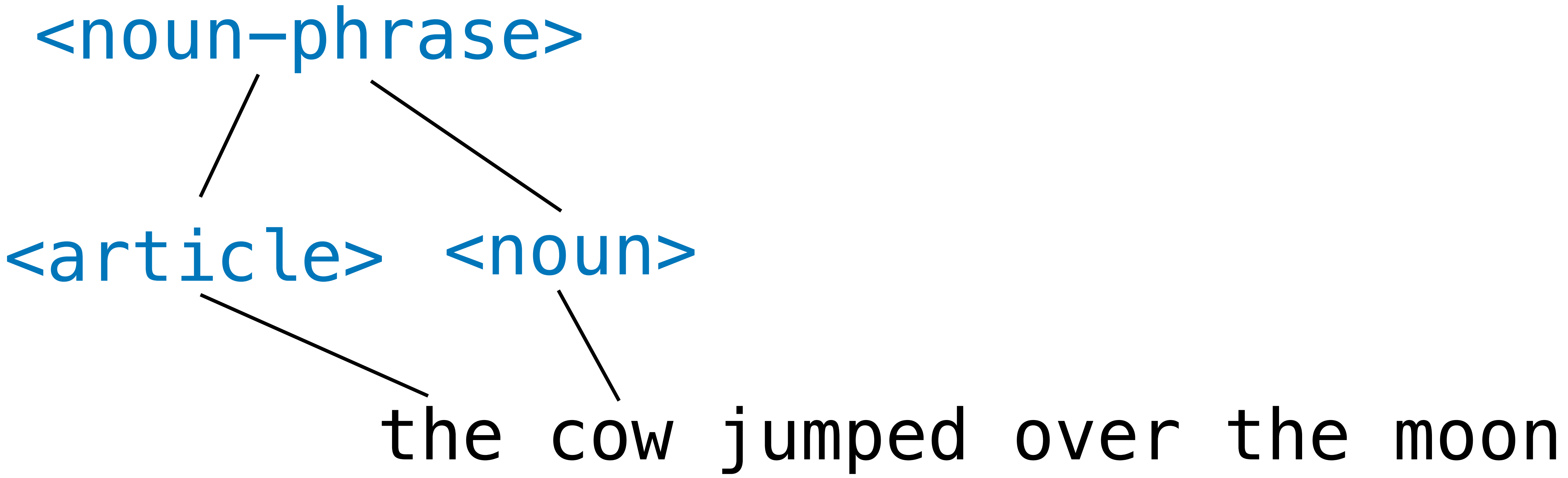
<noun-phrase>

<article> <noun>

the cow jumped over the moon

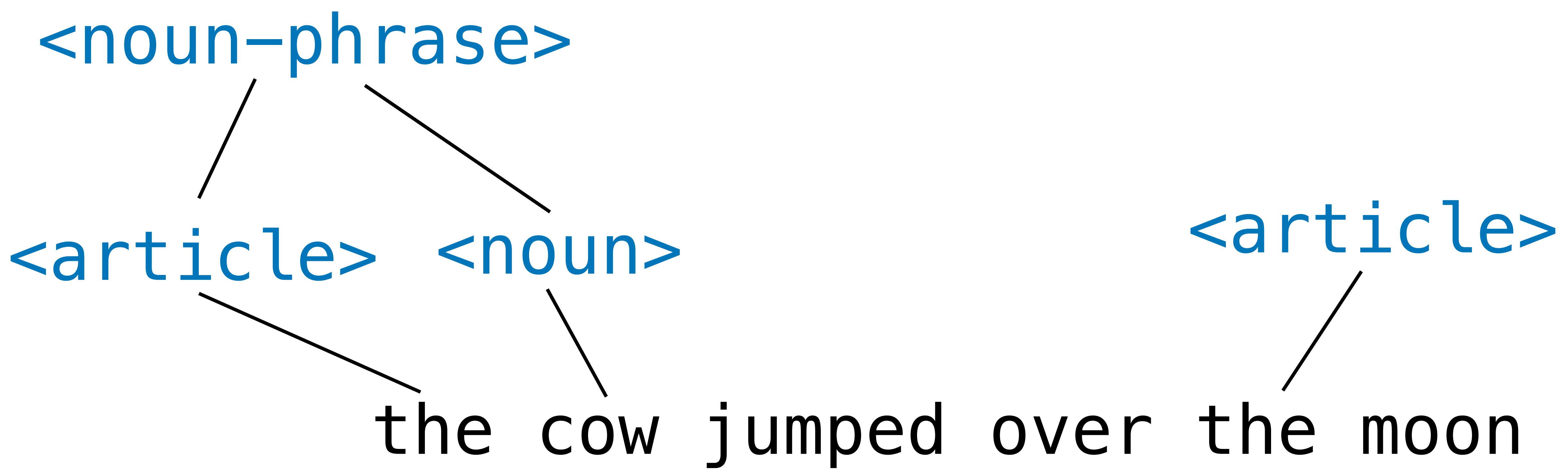
A Derivation

<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



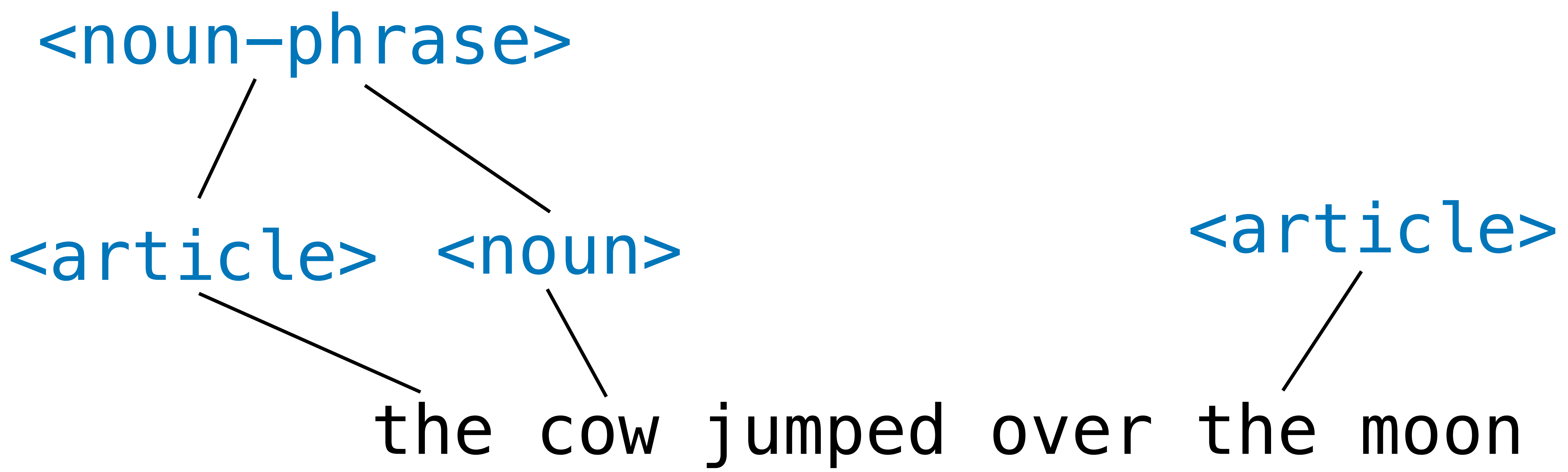
A Derivation

<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>



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>

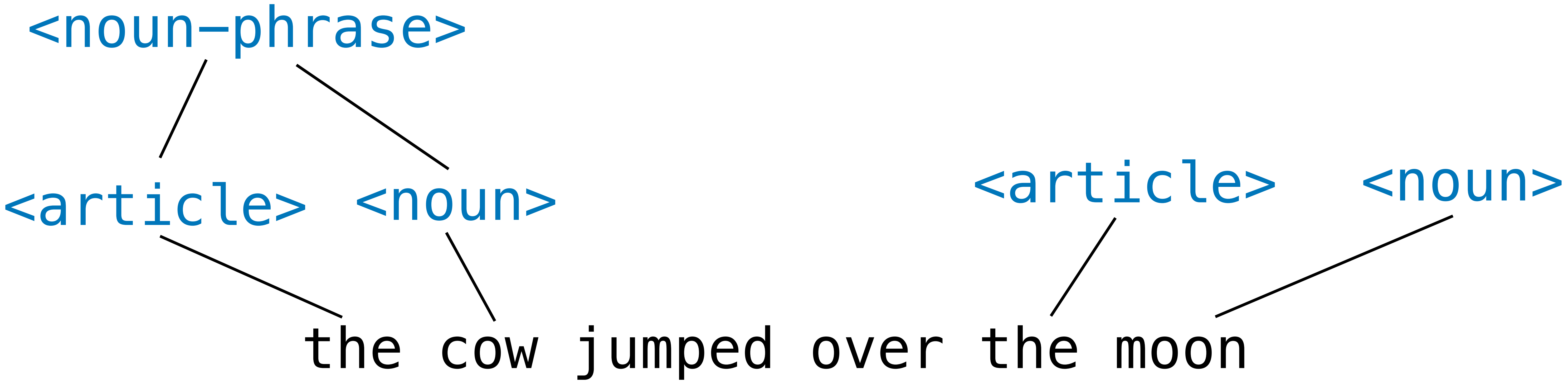


A Derivation

<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>

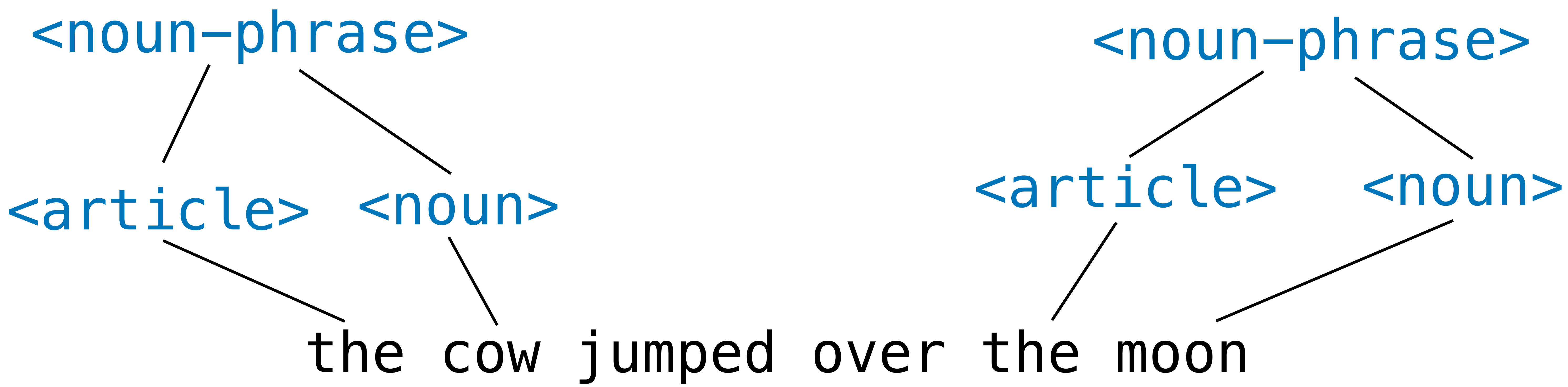
A Derivation

<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>

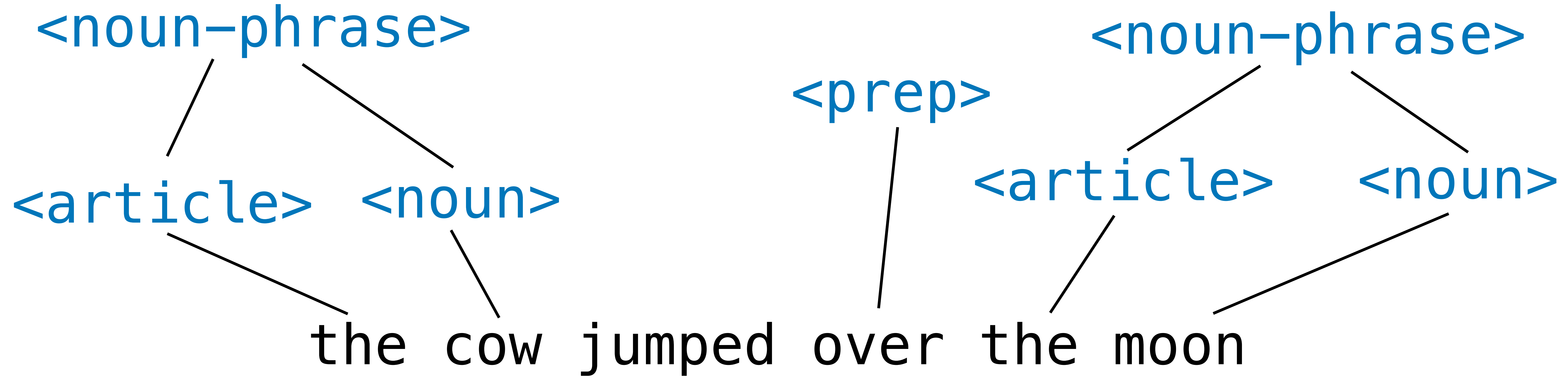


A Derivation

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

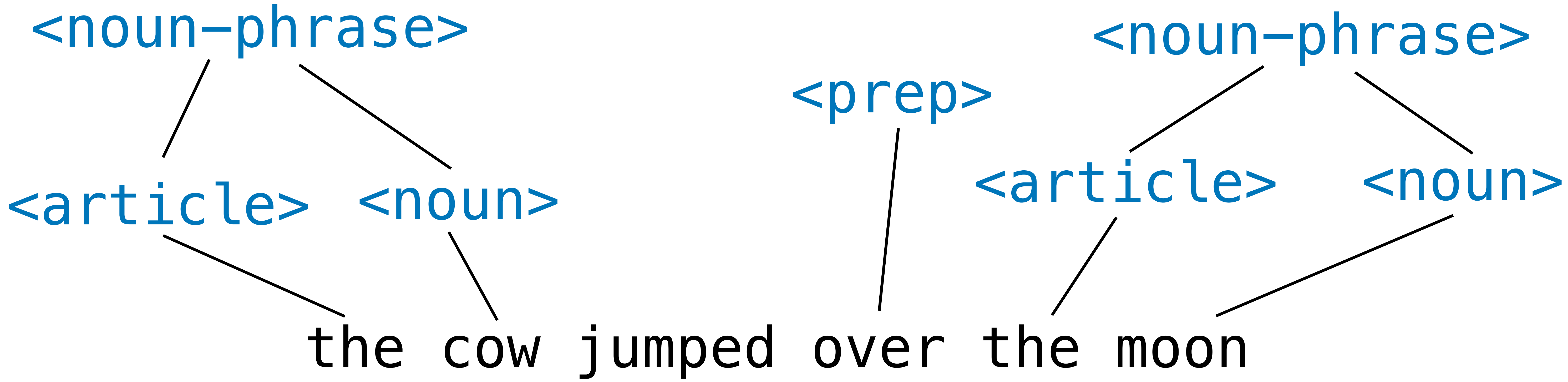
A Derivation

<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>

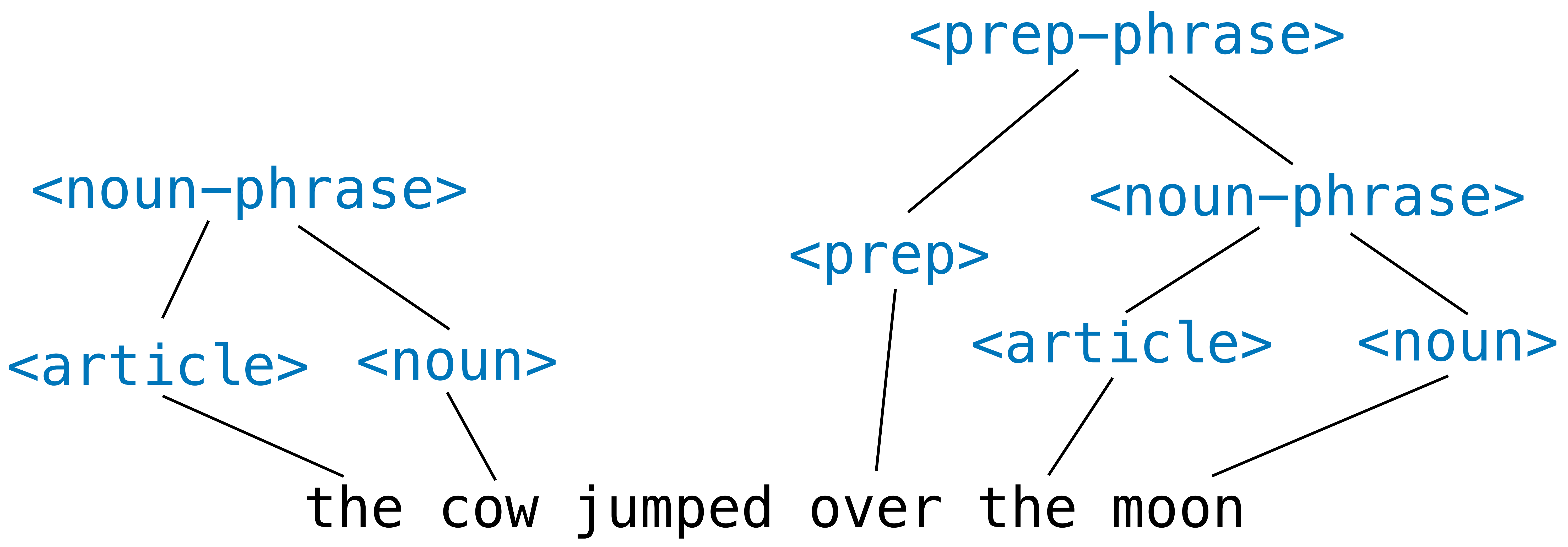


A Derivation

<sentence>

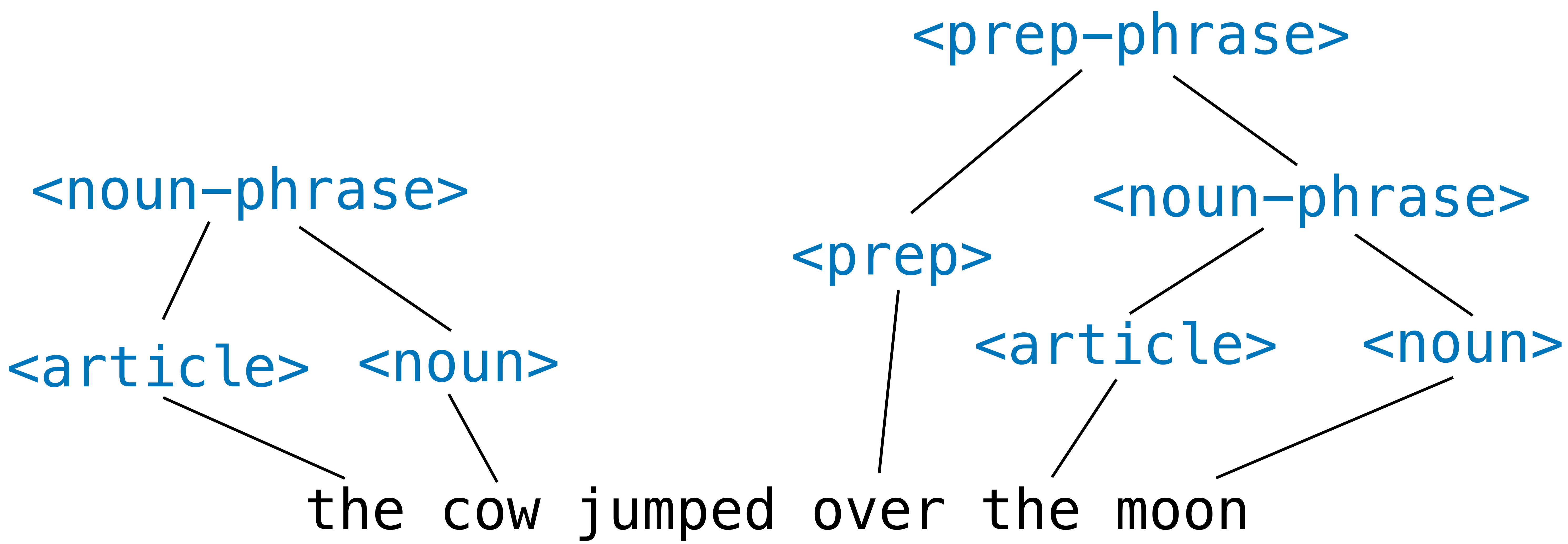
<noun-phrase> <verb-phrase>

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



A Derivation

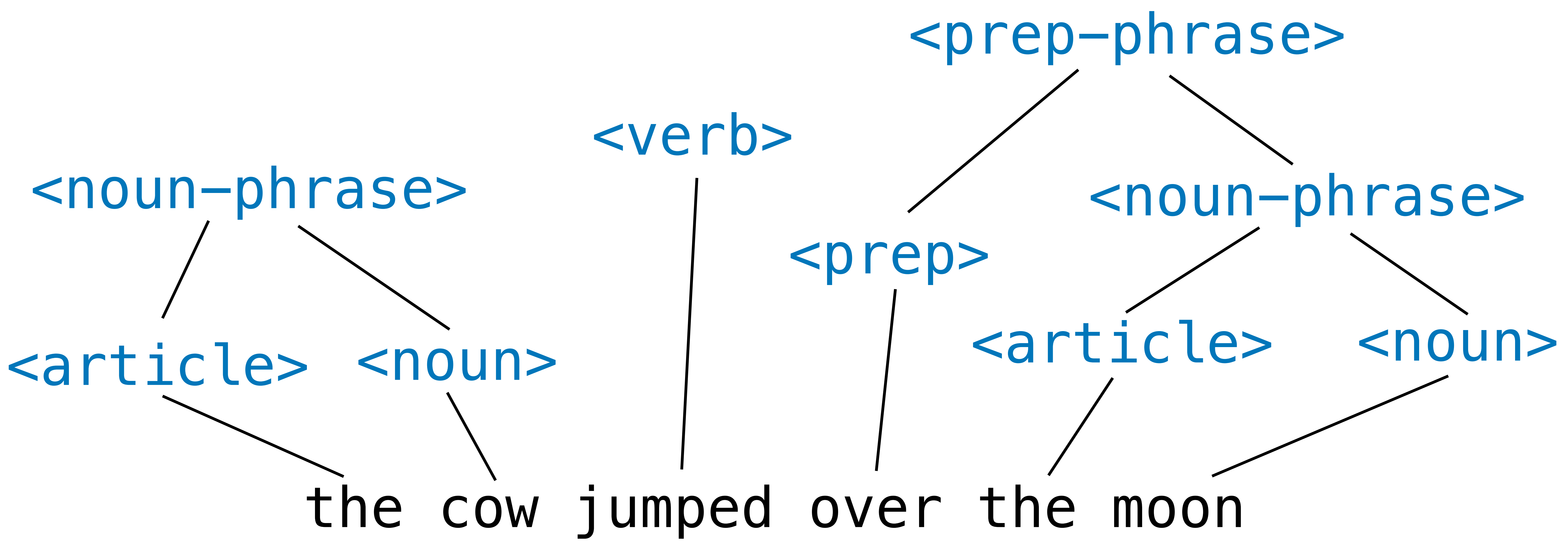
<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>



A Derivation

<sentence>

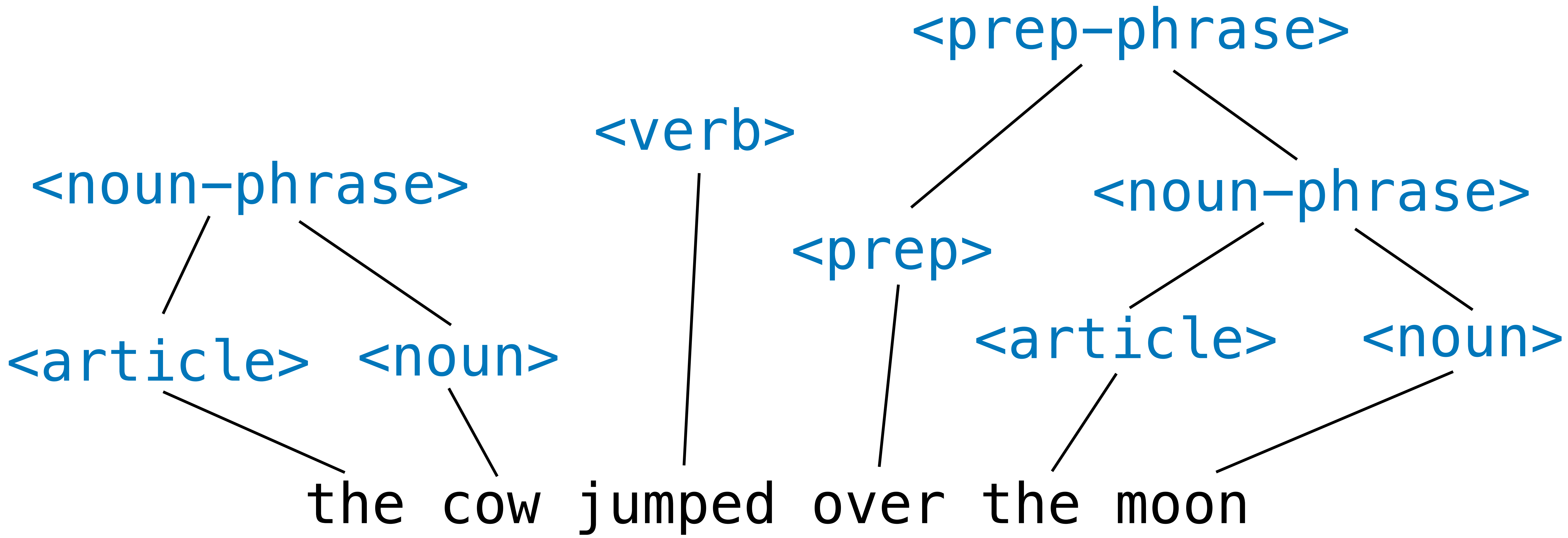
<noun-phrase> <verb-phrase>



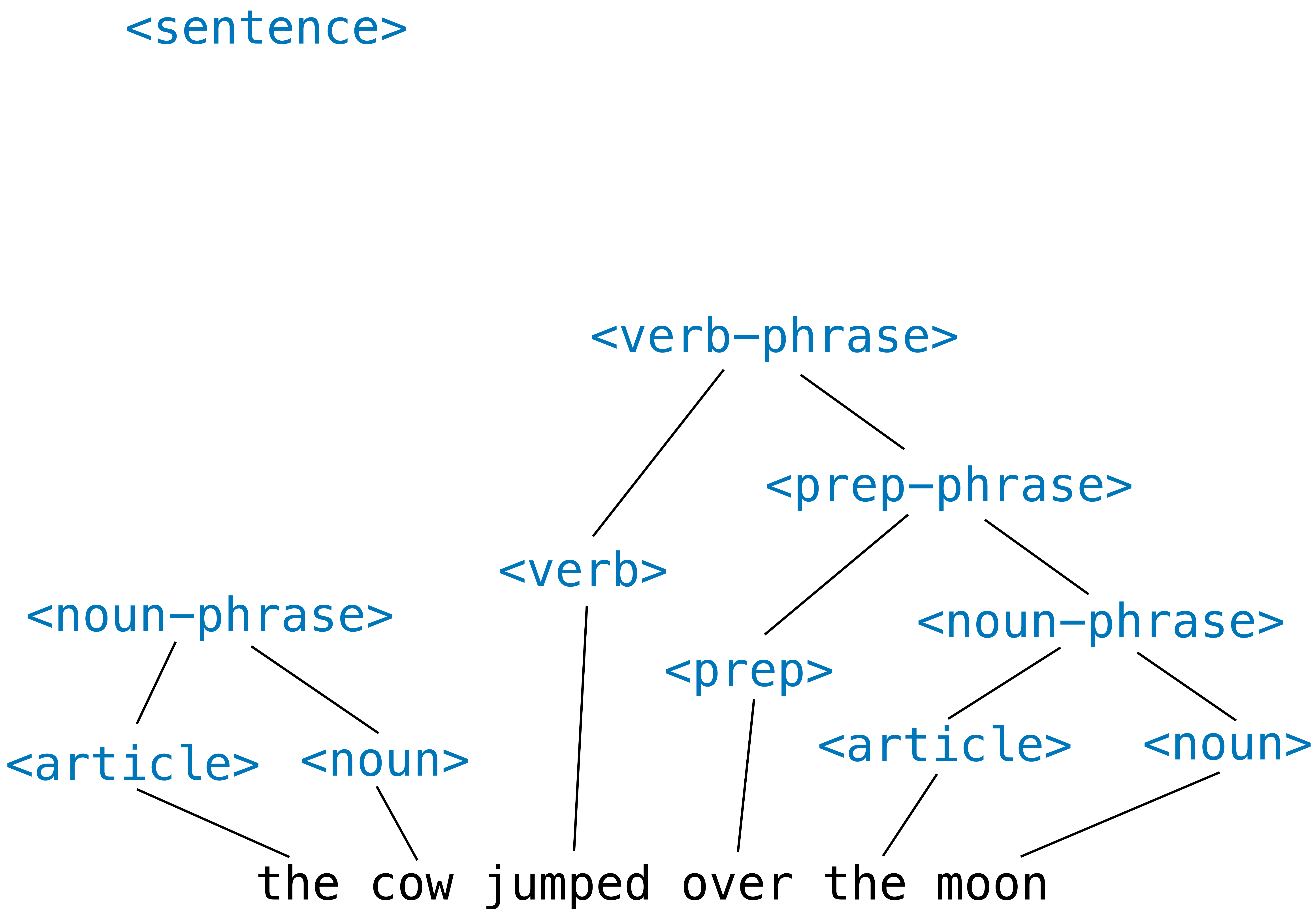
A Derivation

<sentence>

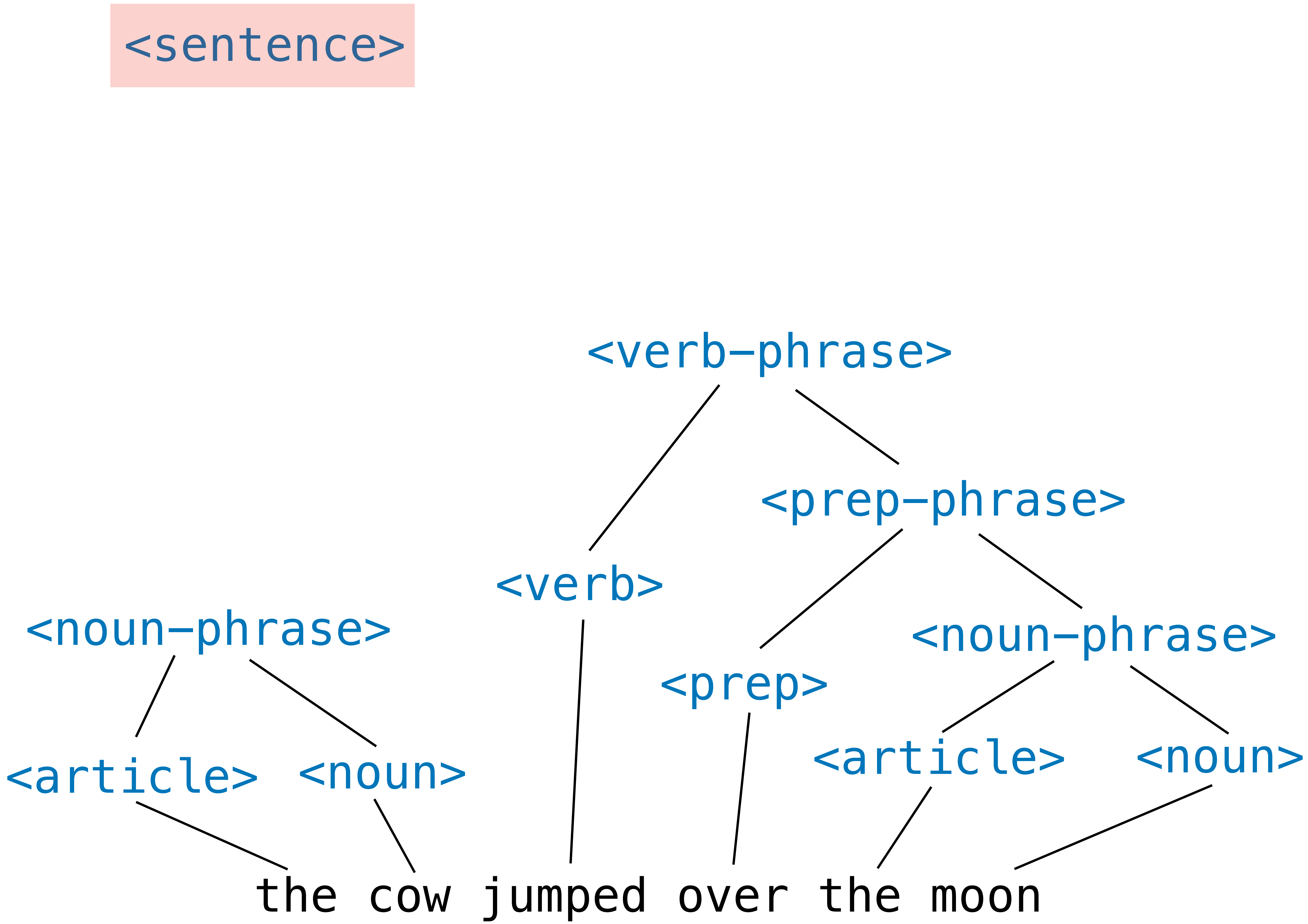
<noun-phrase> <verb-phrase>



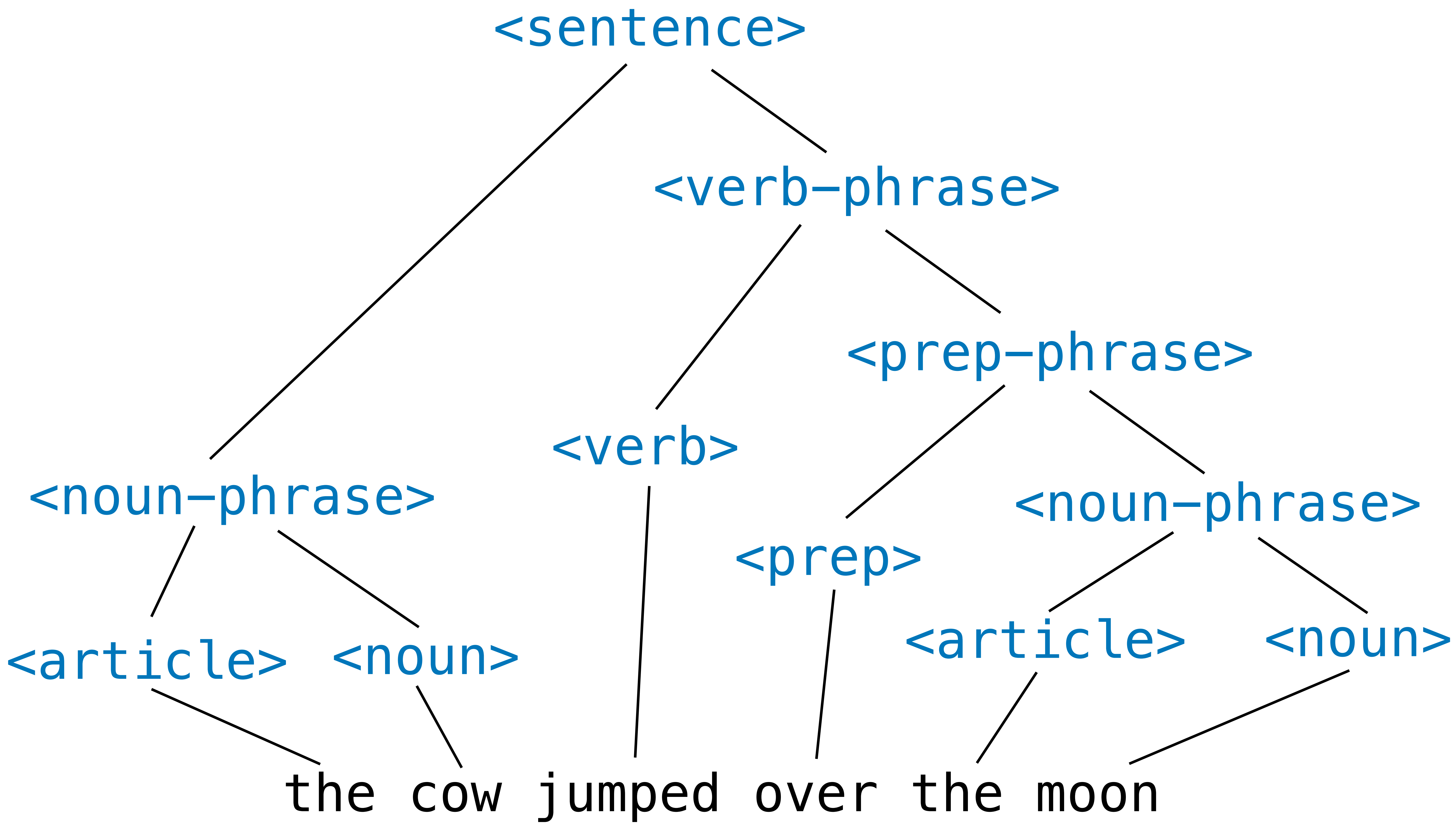
A Derivation



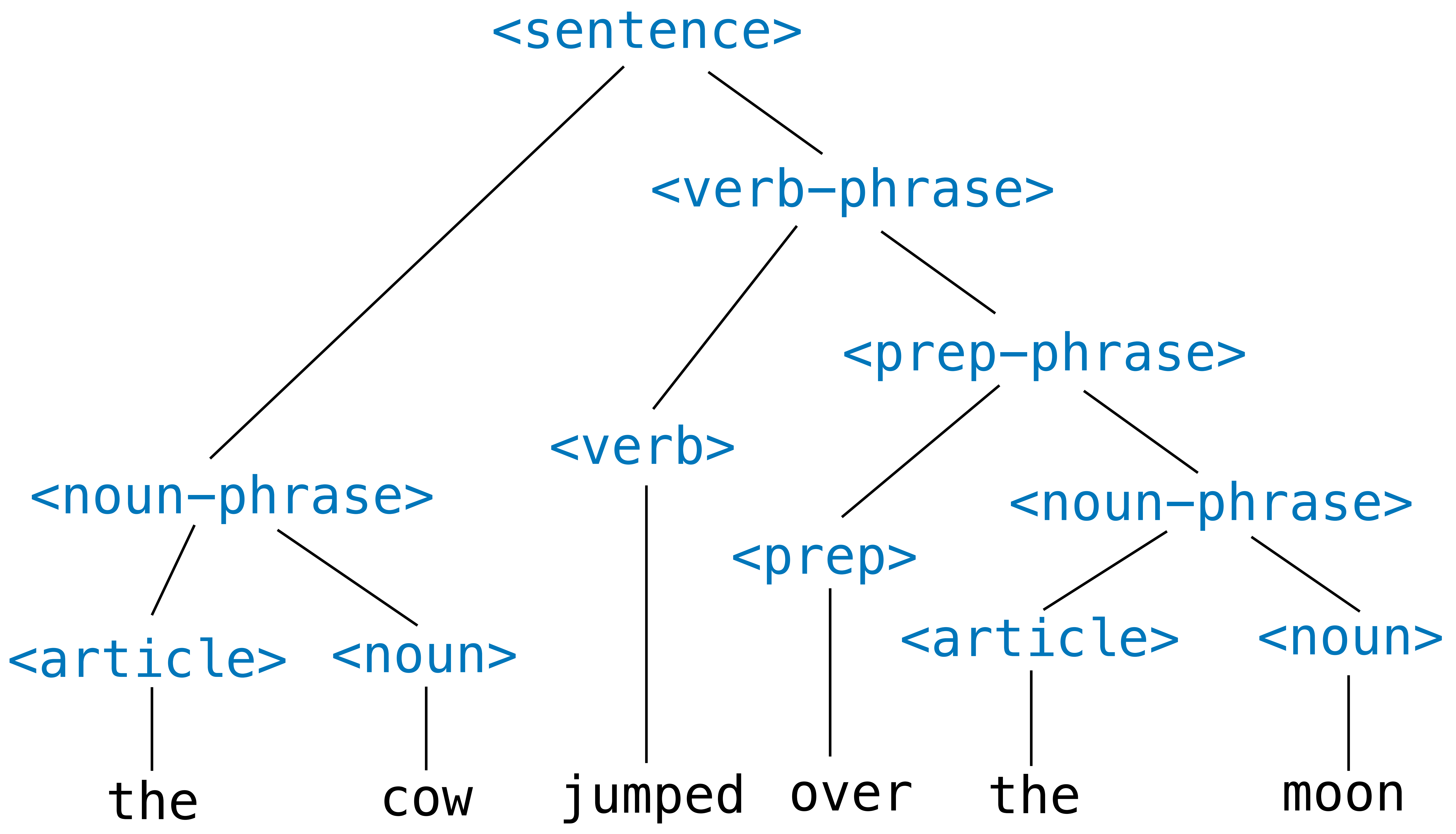
A Derivation



A Derivation



A Parse Tree



A derivation encodes hierarchical structure

Definitions (Symbols and Sentences)

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

Definitions (Symbols and Sentences)

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

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

Definitions (Symbols and Sentences)

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

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

Nonterminal symbols are symbols we will be allowed to "expand" (e.g., **<article>**)

Definitions (Symbols and Sentences)

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

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

Nonterminal symbols are symbols we will be allowed to "expand" (e.g., **<article>**)

Terminal symbols are symbols cannot be further expanded (e.g. *moon*)

Definitions (Symbols and Sentences)

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

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

Nonterminal symbols are symbols we will be allowed to "expand" (e.g., **<article>**)

Terminal symbols are symbols cannot be further expanded (e.g. *moon*)

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

Definitions (Symbols and Sentences)

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

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

Nonterminal symbols are symbols we will be allowed to "expand" (e.g., **<article>**)

Terminal symbols are symbols cannot be further expanded (e.g. *moon*)

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

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

Production Rules

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

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 "|"

Recall: Let-Expressions (Syntax Rule)

```
<expr> ::= let <var> = <expr> in <expr>
```

If x is a valid variable name, and e_1 is a well-formed expression and e_2 is a well-formed expression then

$$\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

is a well-formed expression

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

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

abstractions (non-terminal symbols)

tokens (terminal symbols)

Example

production rules

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

abstractions (non-terminal symbols)

tokens (terminal symbols)

Example

```
<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?

Derivations and Parse Trees

```
<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

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

<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

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
```

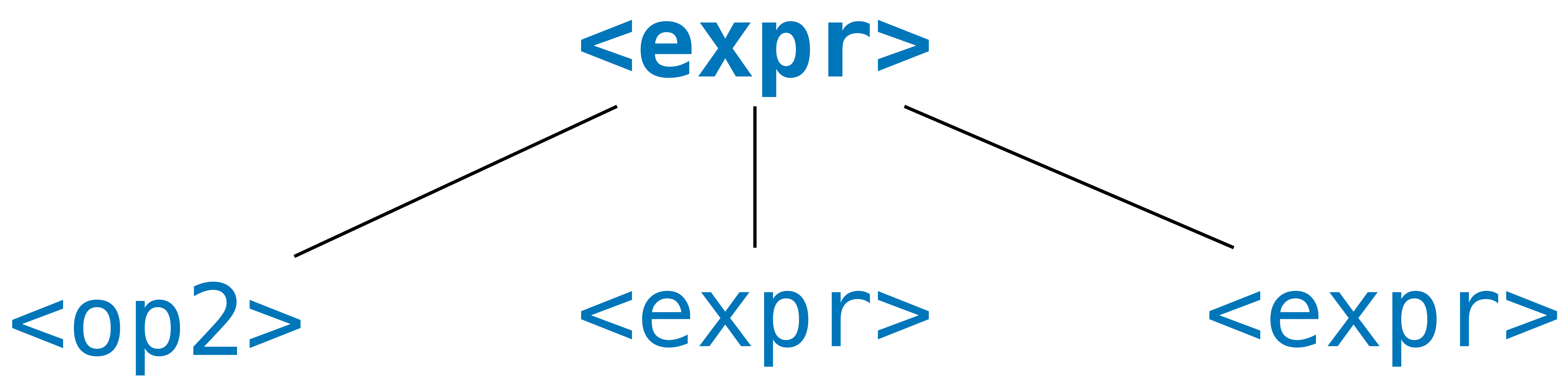
<expr>

<expr>

Derivations and Parse Trees

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

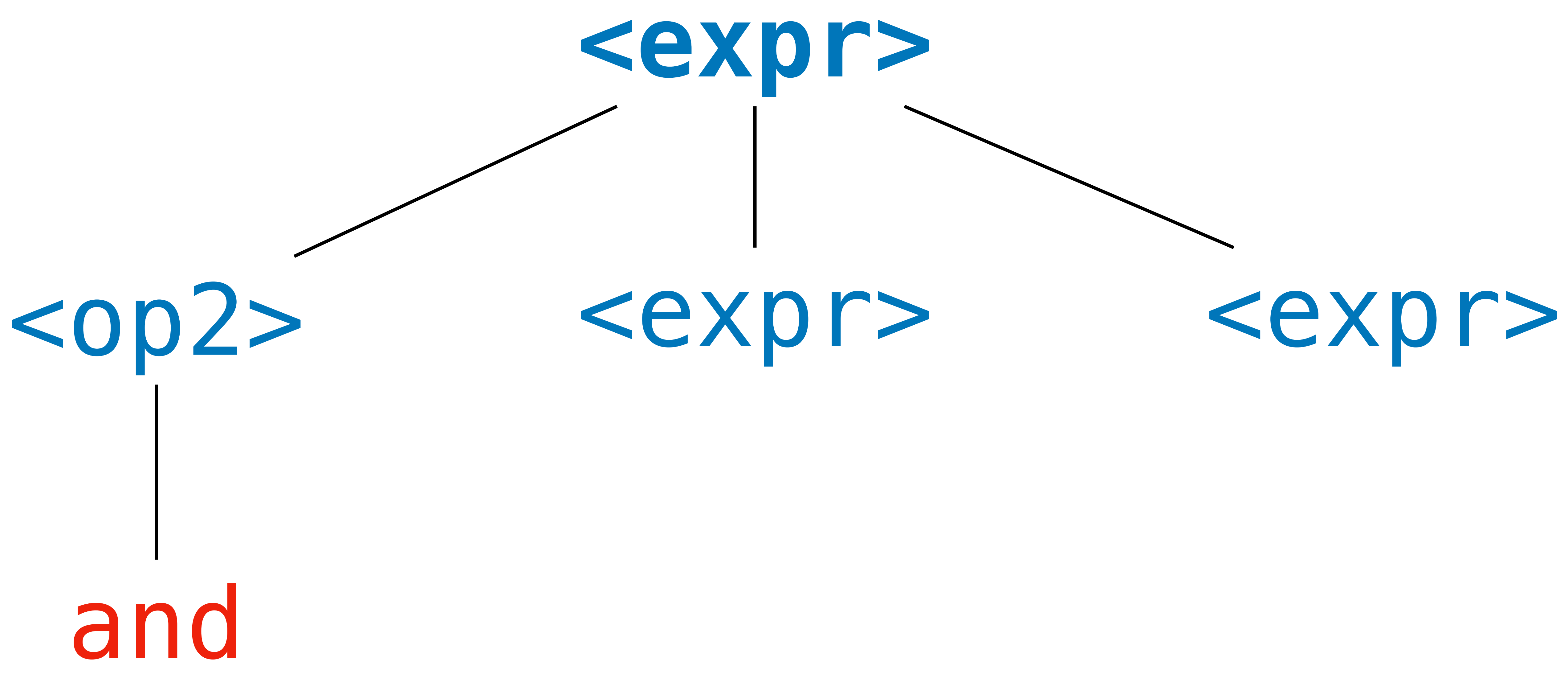
<expr>
<op2> <expr> <expr>



Derivations and Parse Trees

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

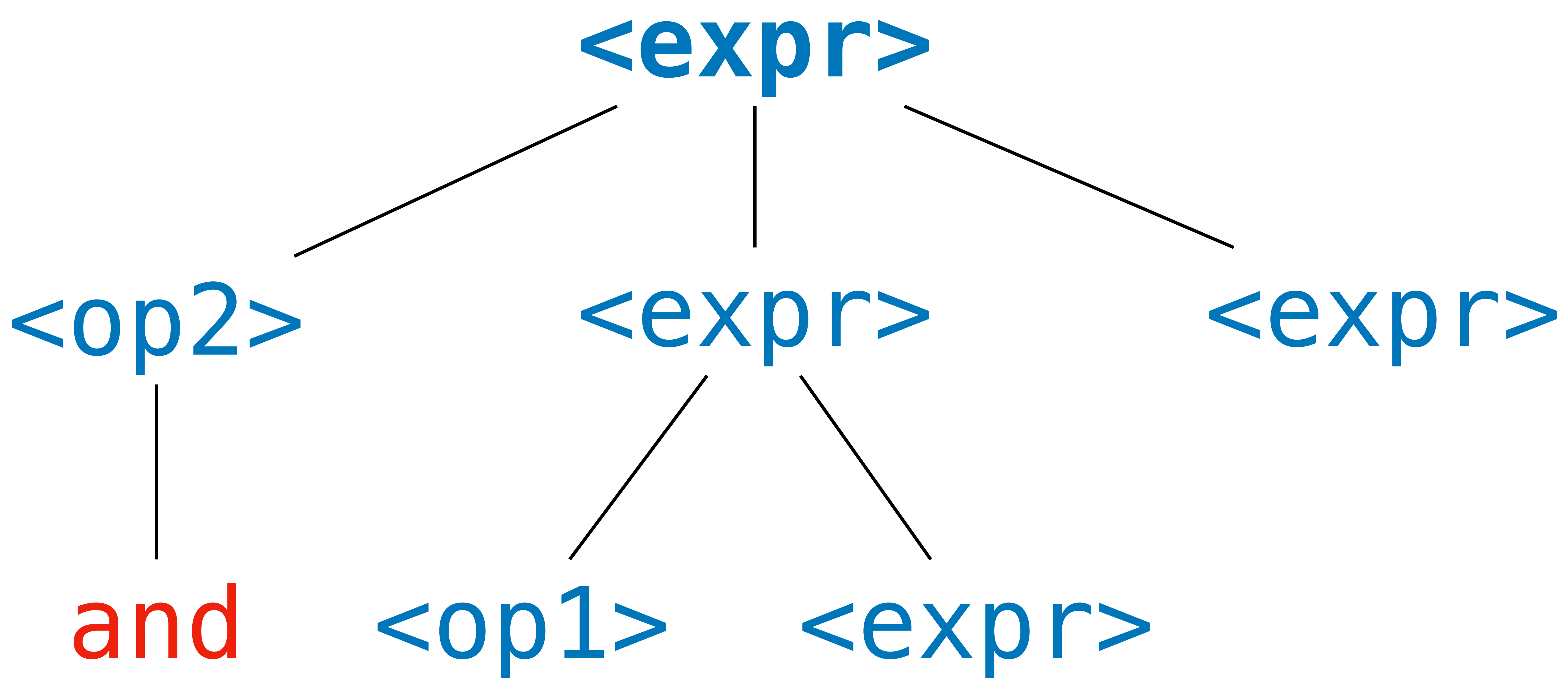
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
```



Derivations and Parse Trees

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

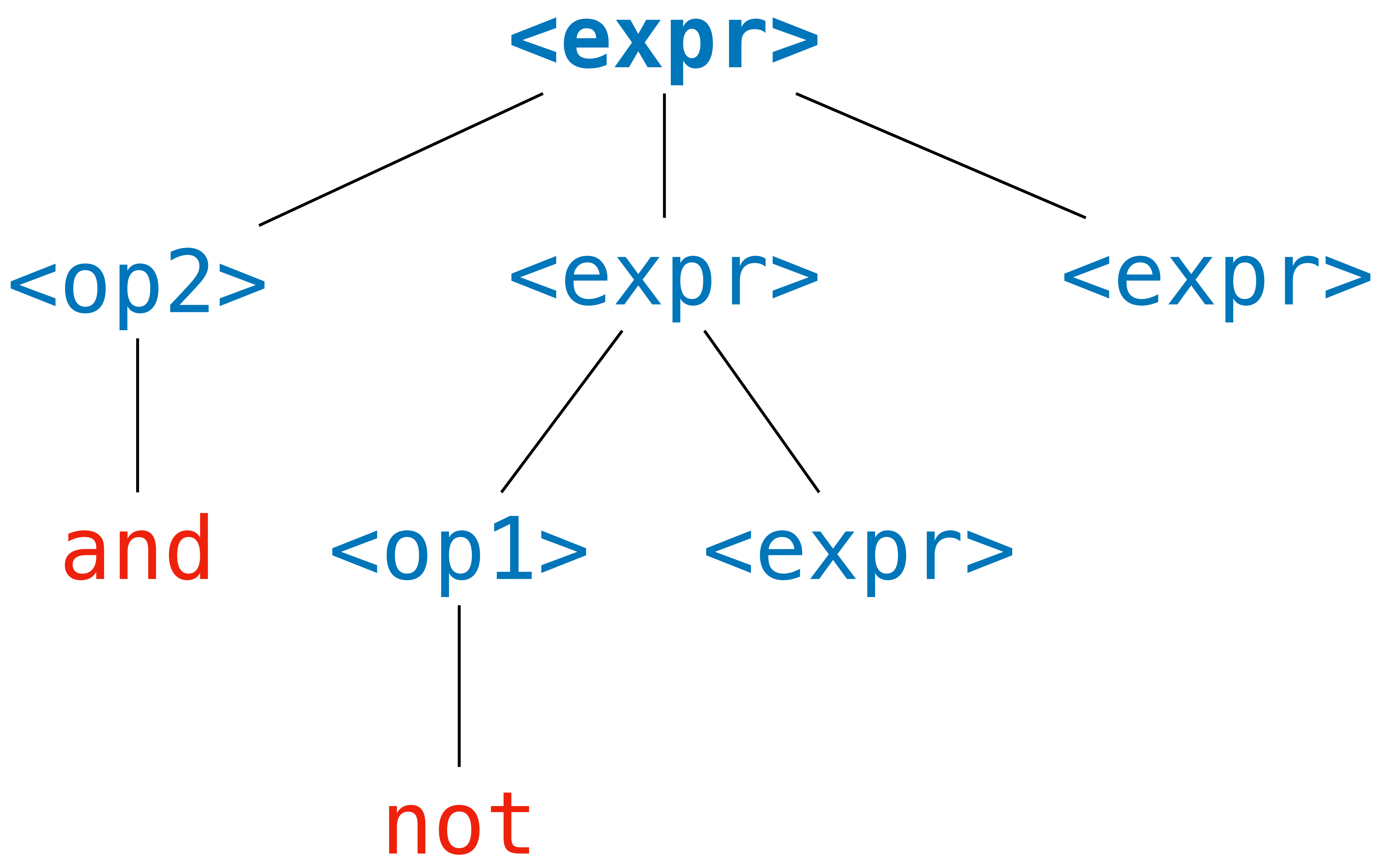
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
```



Derivations and Parse Trees

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

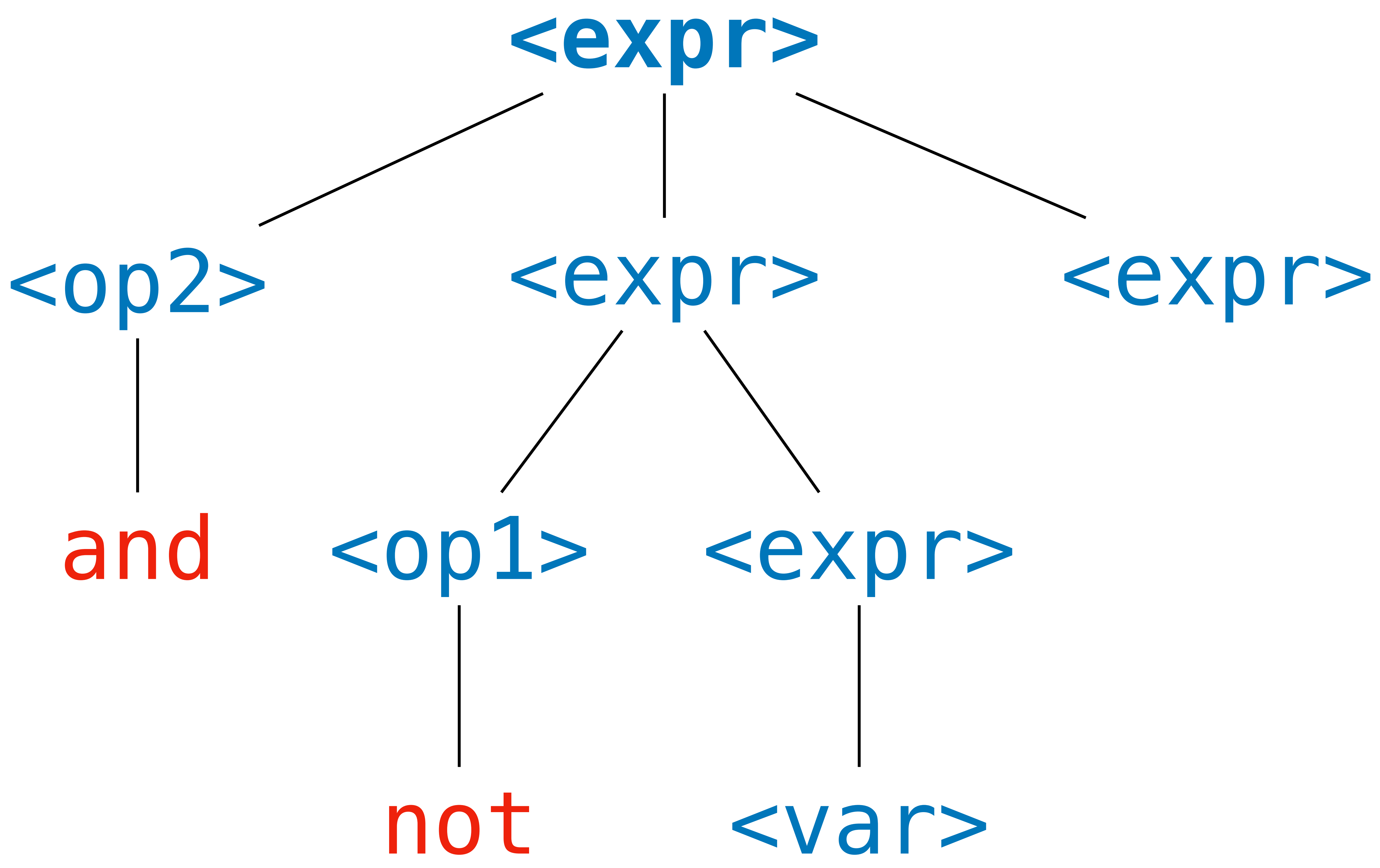
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
```



Derivations and Parse Trees

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

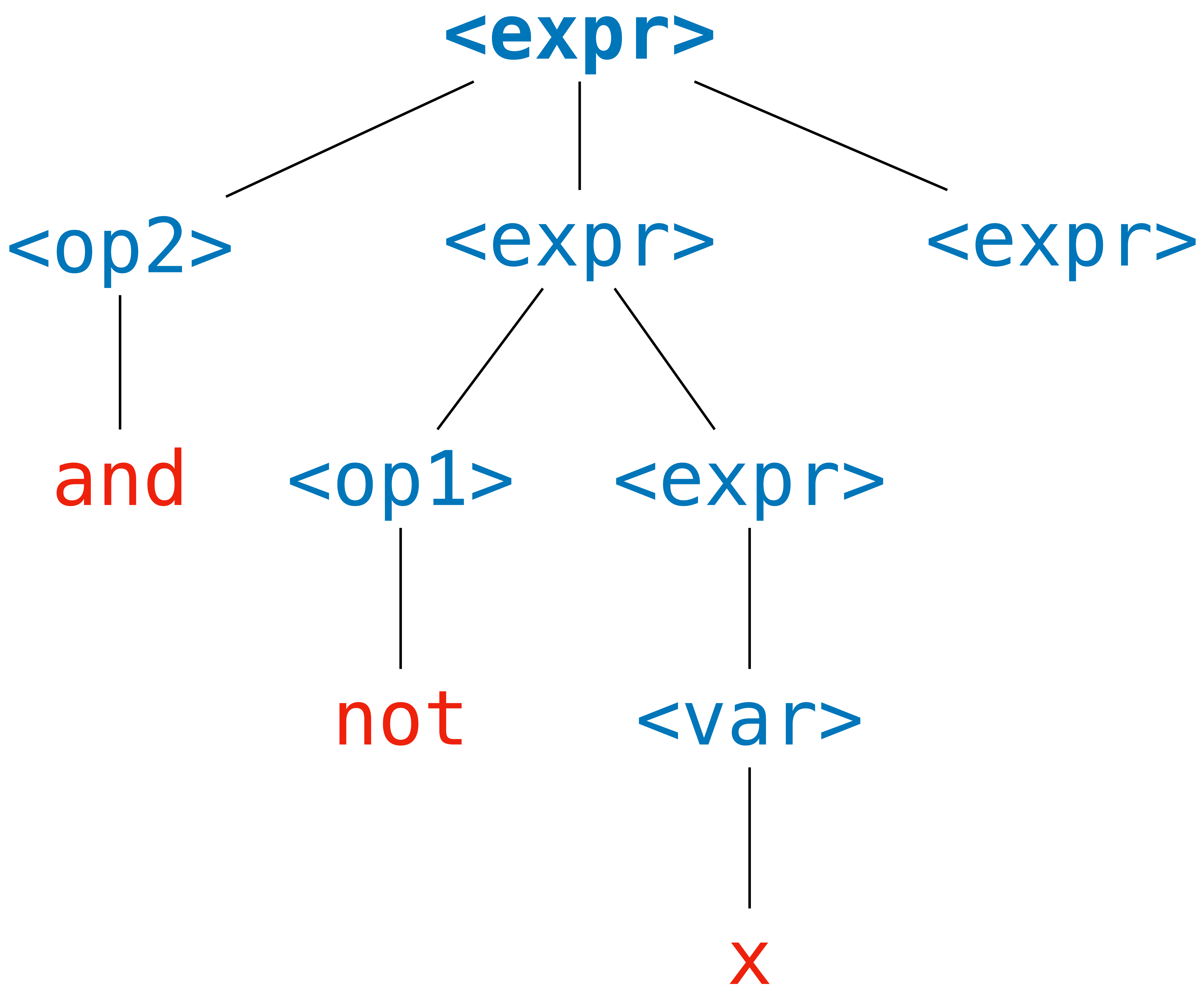
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
```



Derivations and Parse Trees

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

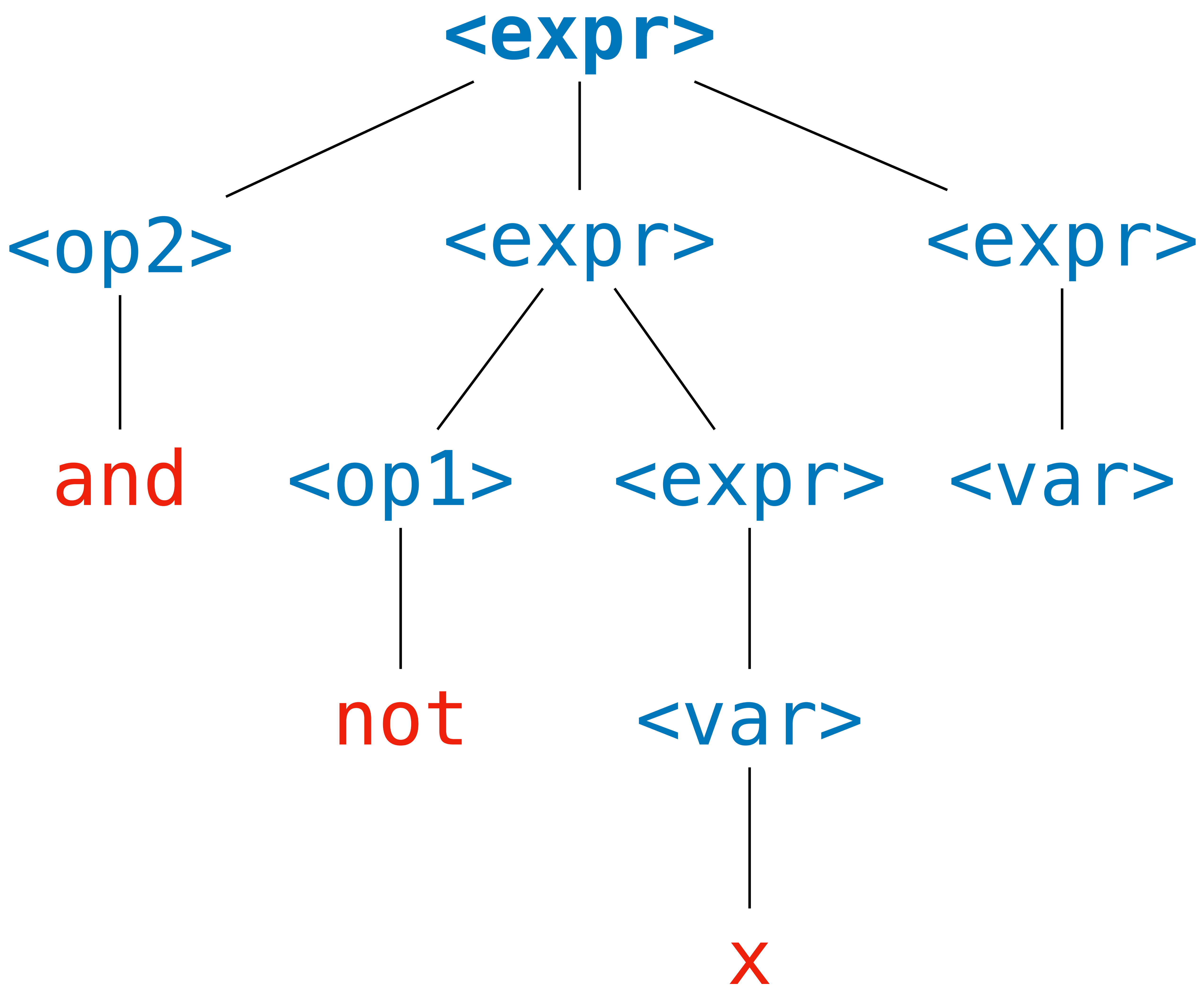
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
```



Derivations and Parse Trees

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

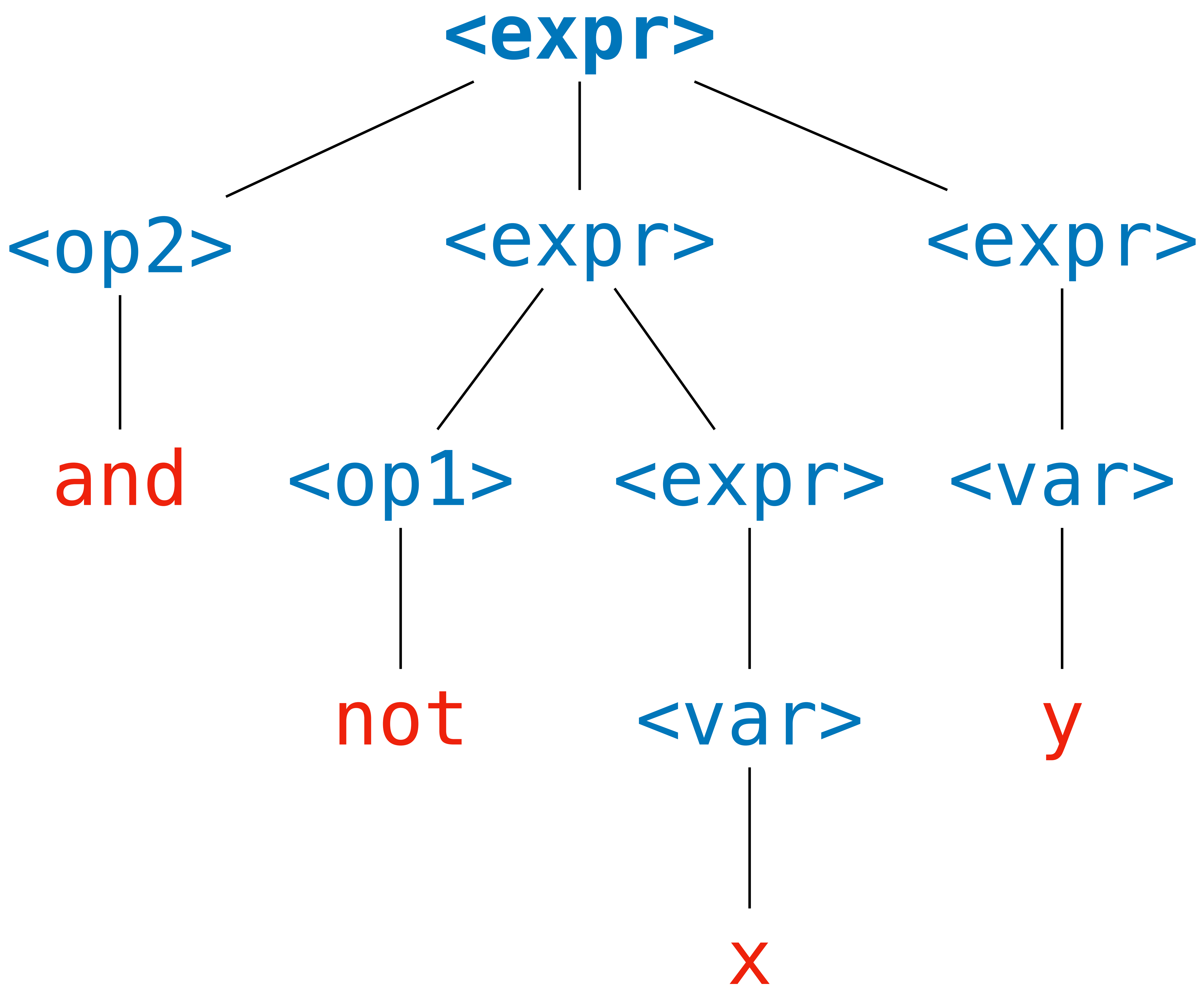
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
and not x <var>
```



Derivations and Parse Trees

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

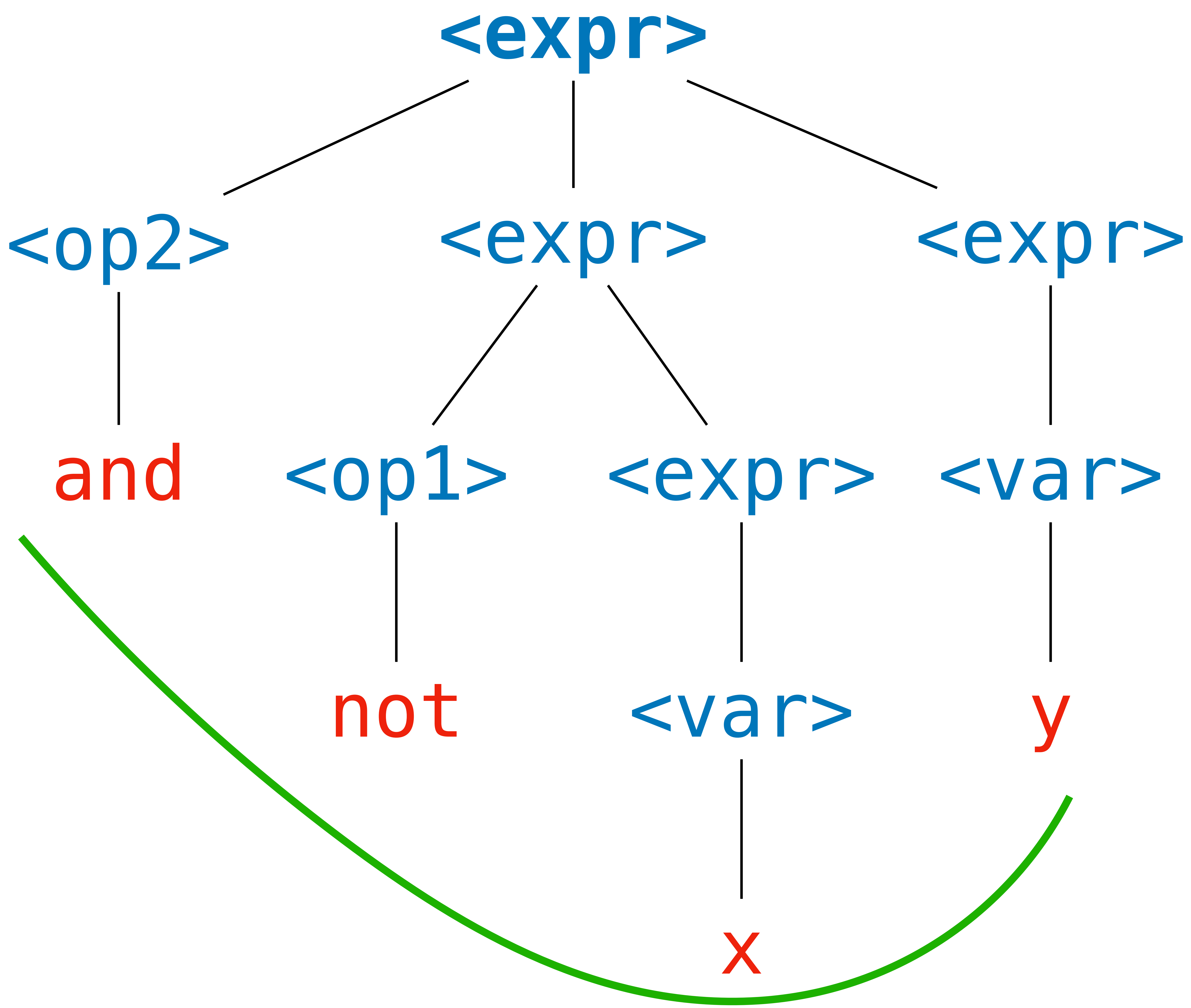
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
and not x <var>
and not x y
```



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
```



The point: parse trees and derivations represent the same hierarchical structure

Why do we care?

$\boxed{\texttt{char stream}} \xrightarrow{\texttt{lexer}} \boxed{\texttt{token stream}} \xrightarrow{\texttt{parser}} \boxed{\texttt{parse tree}} \longrightarrow \cdots$

Why do we care?

`char stream` $\xrightarrow{\text{lexer}}$ `token stream` $\xrightarrow{\text{parser}}$ `parse tree` $\longrightarrow$ . . .

`a.k.a. list of terminal symbols`

Why do we care?

`char stream` $\xrightarrow{\text{lexer}}$ `token stream` $\xrightarrow{\text{parser}}$ `parse tree` $\longrightarrow$ . . .

`token stream`: a.k.a. list of terminal symbols

`parse tree`: a.k.a. derivation in a grammar

Why do we care?

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

Why do we care?

`char stream` —lexer→ `token stream` —parser→ `parse tree` → . . .

token stream: a.k.a. list of terminal symbols

parse tree: a.k.a. derivation in a grammar

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

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

Practice Problem

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

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

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

not x and y or z

Example

```
let x = 2 in if x = z then x else y
```

Example

```
let x = 2 in if x = z then x else y
```

How can we demonstrate that this is a well-formed expression?

Example

```
let x = 2 in if x = z then x else y
```

How can we demonstrate that this is a well-formed expression?

Answer: Well build a derivation/parse tree for it with the root **<expr>**!

Recall: Let-Expressions (Syntax Rule)

```
<expr> ::= let <var> = <expr> in <expr>
```

If x is a valid variable name, and e_1 is a well-formed expression and e_2 is a well-formed expression then

$$\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

is a well-formed expression

Recall: If-Expressions (Syntax Rule)

```
<expr> ::= if <expr> then <expr> else <expr>
```

If e_1 is a well-formed expression and e_2 is a well-formed expression and e_3 is a well-formed expression, then

if e_1 then e_2 else e_3

is a well-formed expression

```
let x = 2 in if x = z then x else y
```

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.

(examples taken from around the internet)

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

(examples taken from around the internet)

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

(examples taken from around the internet)

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.*

(examples taken from around the internet)

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

(examples taken from around the internet)

Ambiguity in Natural Language

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

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

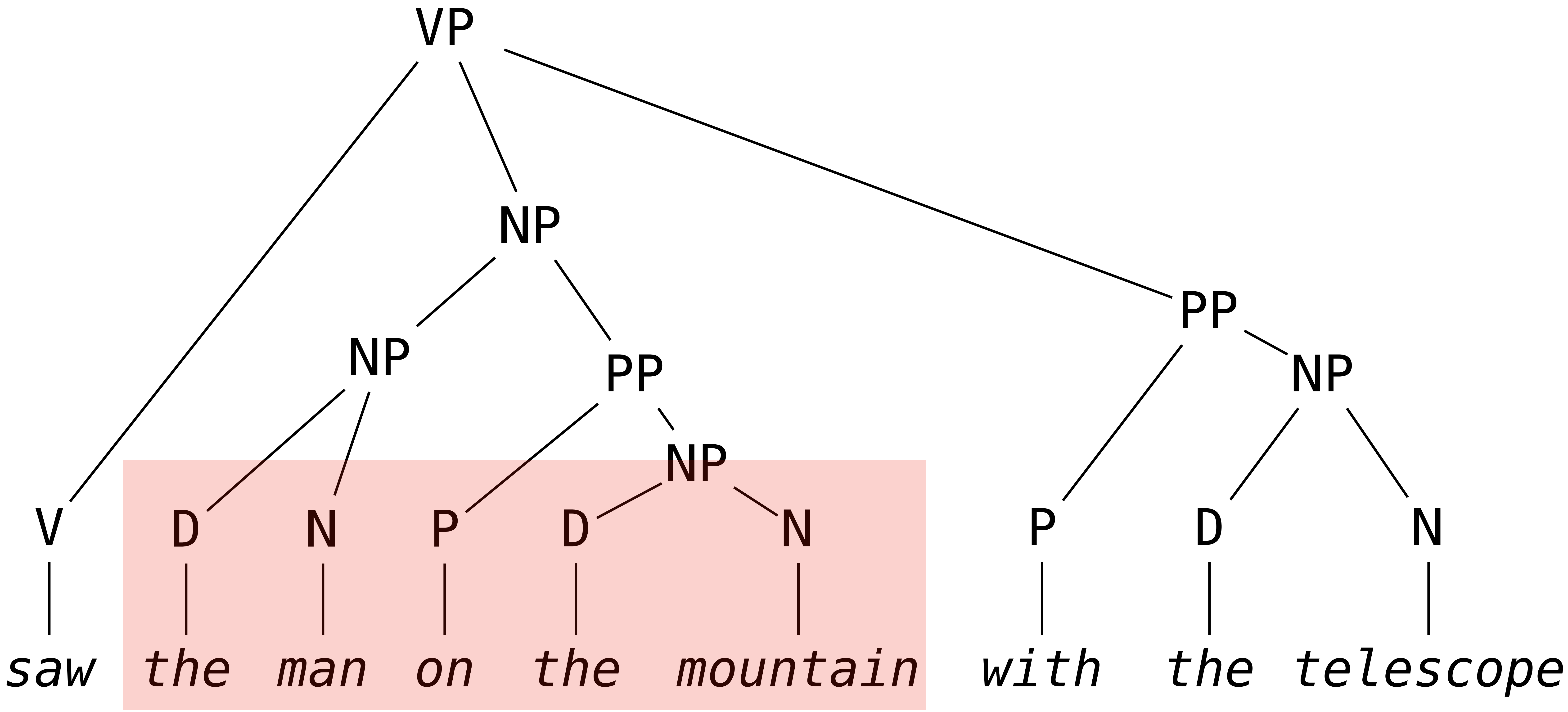
*He said the exam would **be held** on Tuesday.*

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

We have informal tactics for avoiding these pitfalls

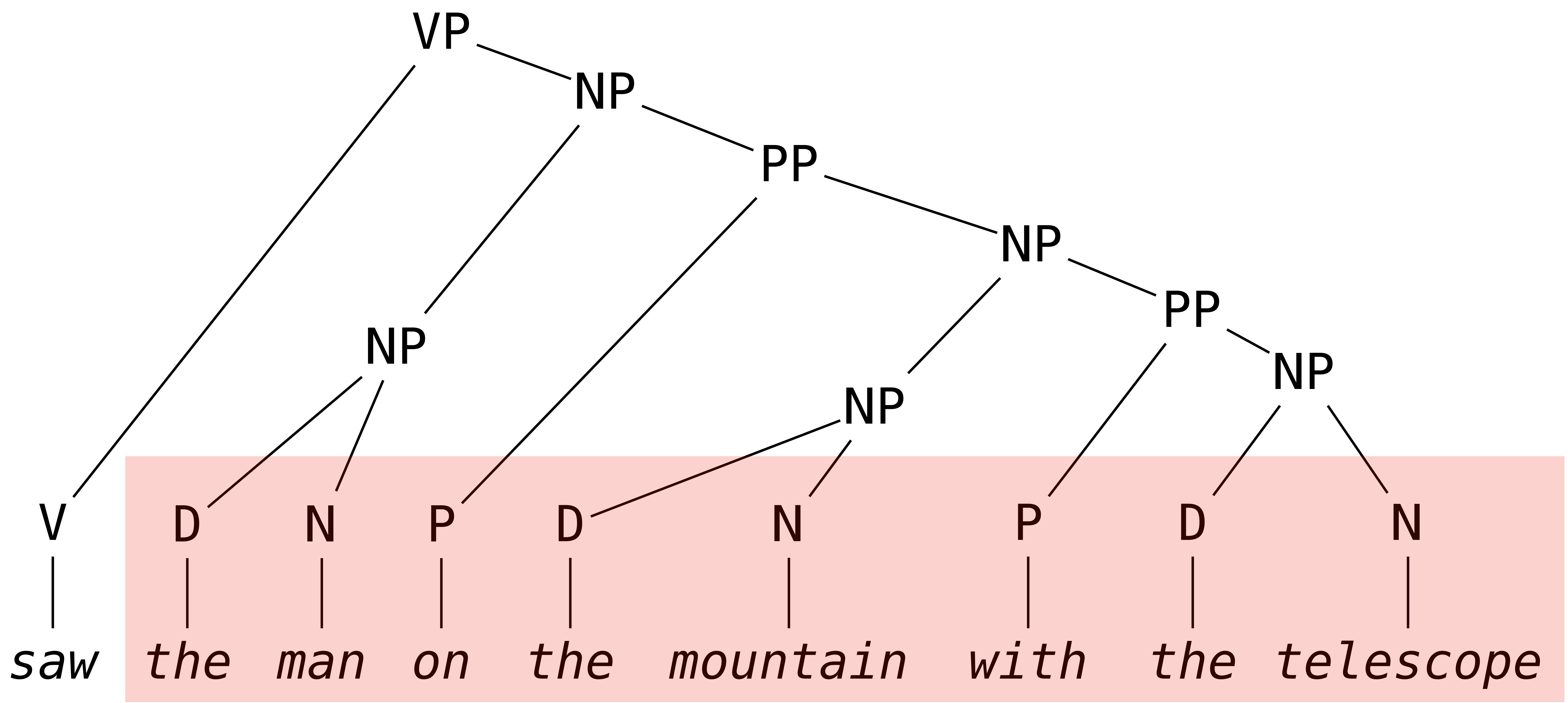
(examples taken from around the internet)

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
```

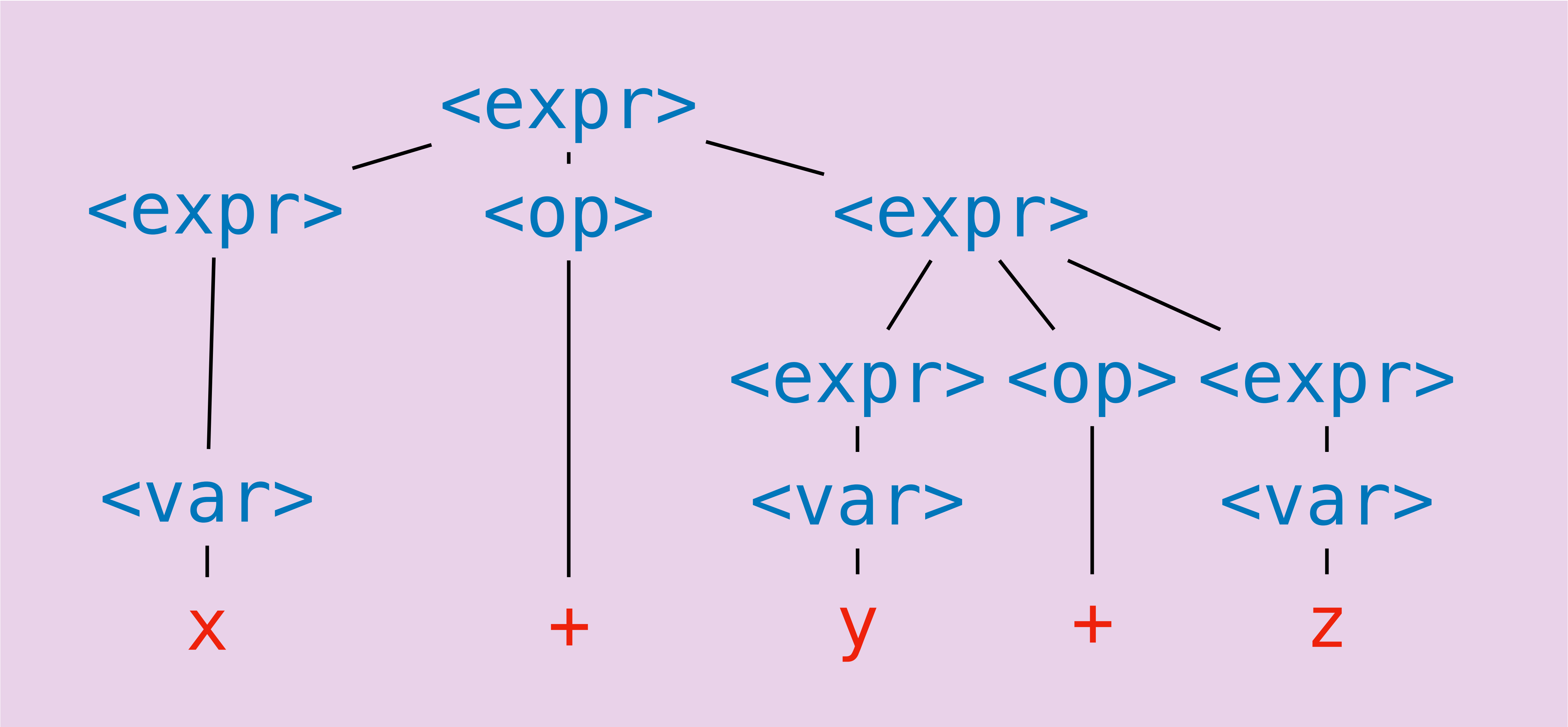
x + y + z can be derived as

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

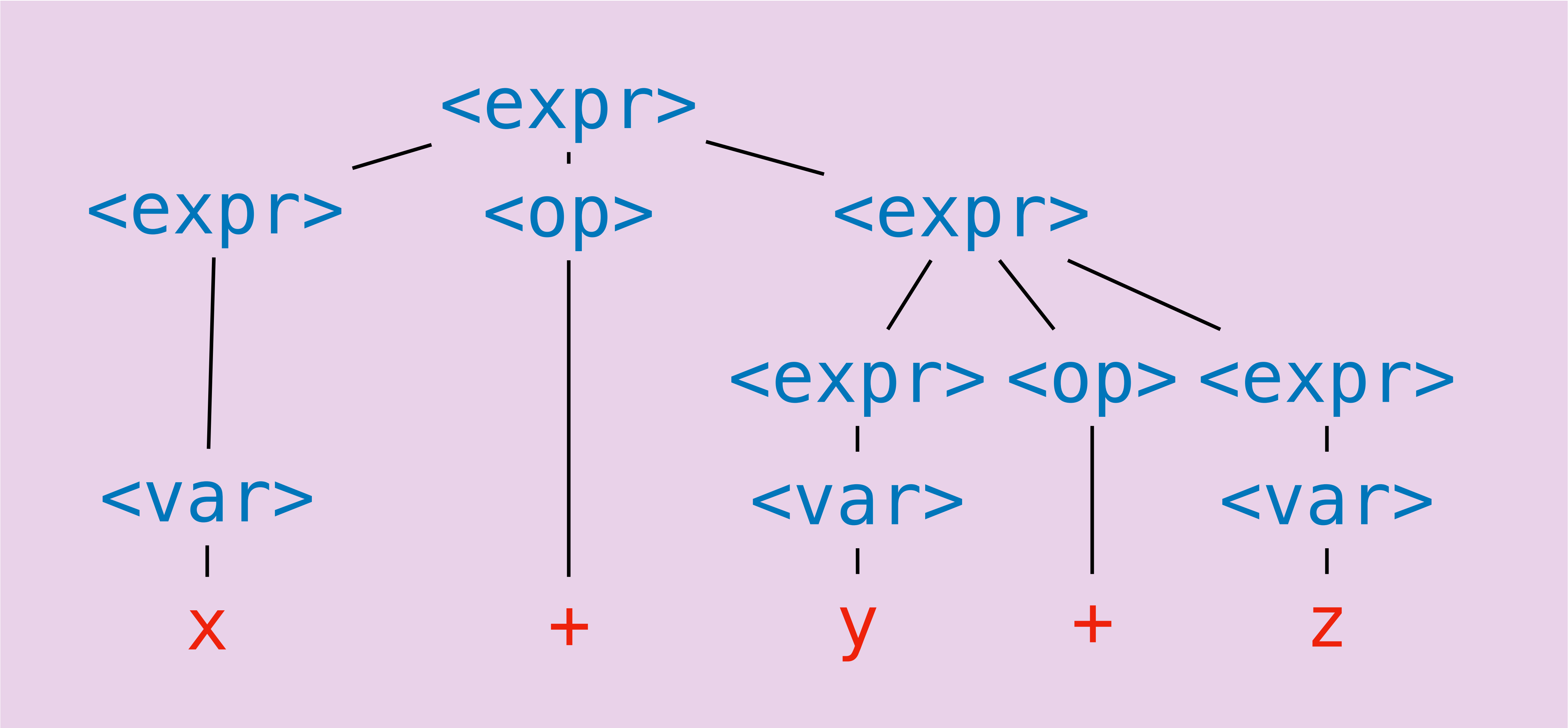


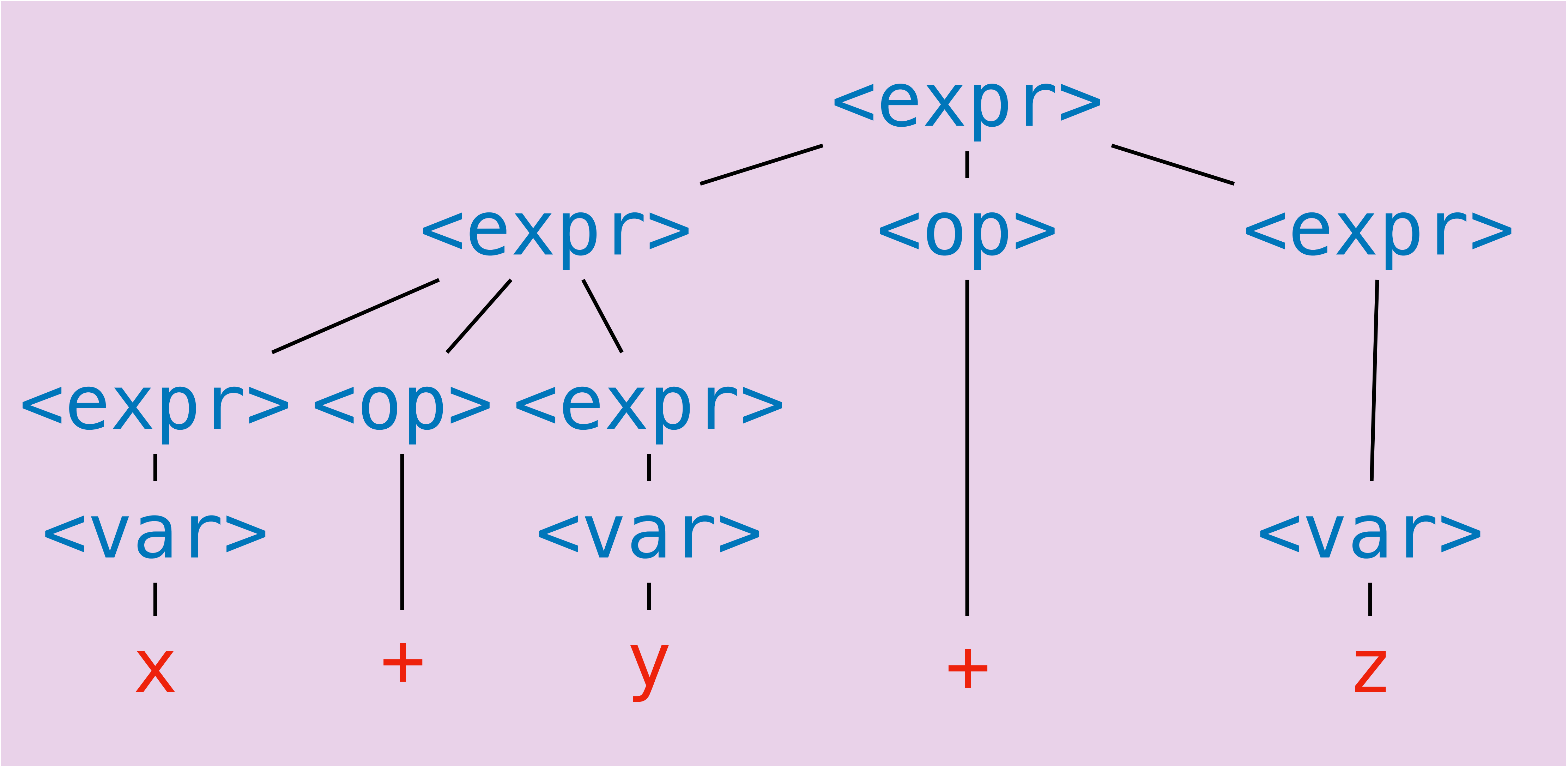
Ambiguity in Formal Grammar

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





Example

```
<expr> ::= x
         | if <expr> then <expr>
         | if <expr> then <expr> else <expr>
```

What can we do about ambiguity?

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! (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! (get from ref)
infix	a * b, a + b, a 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! (get from ref)
infix	a * b, a + b, a mod b
mixfix	if b then x 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

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

Lots of Parentheses

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

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

Lots of Parentheses

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

```
(((not x) and (not (not y)) or z)

((((x or y) or z) or x) or y)

(not ((not x) and (not y)) or
     (x and z))

(x and y)
```

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?

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)

<u>Associativity:</u>

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

<u>Precedence:</u>

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

Associativity

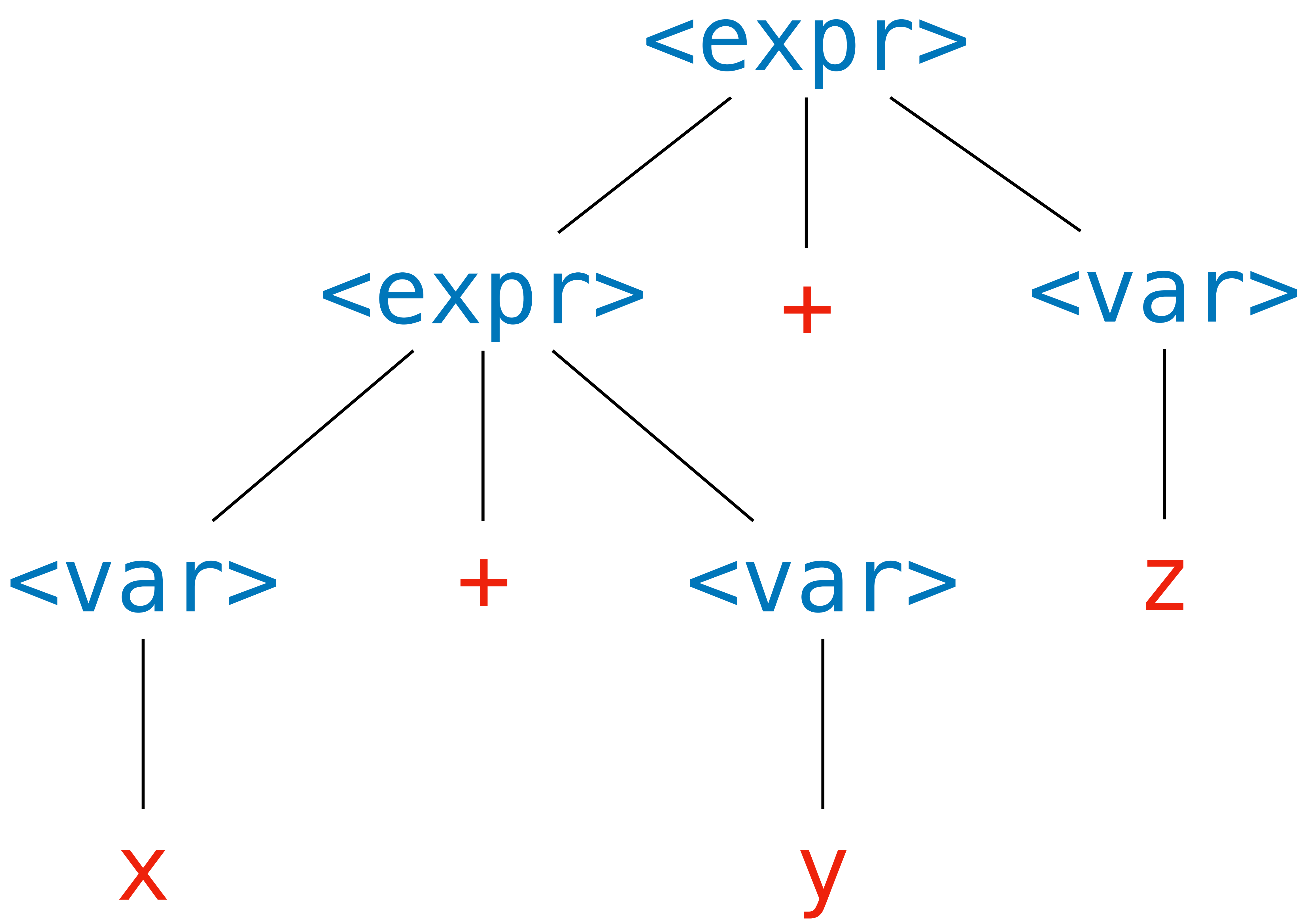
The associativity of an infix operator refers to how its arguments are grouped in the absence of parentheses:

```
left associative      1 +  2 +  3 ⇒ (1 +  2) +  3

right associative     a -> b -> c ⇒  a -> (b -> c)
```

Associativity

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

x + y + z ⇒



"*add the sum of x and y to z*"

How do we enforce that we get a tree of this shape?

The Culprit

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

The Culprit

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

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

The Culprit

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

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

```
<expr> + <expr> ⇒ <expr> + <expr> + <expr>
```

The Culprit

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

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

<expr> + <expr> ⇒ <expr> + <expr> + <expr>

Which <expr> did we replace?

The Solution: Breaking Symmetry

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

```
<expr>
<expr> + <var>
<expr> + z
<expr> + <var> + z
<expr> + y + z
<var> + y + z
x + y + z
```

We make sure that one of the arguments must be "simpler"

By enforcing that the second argument is a <var>, we will get the left-associative parse tree

Example Parse Tree

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

```
<expr>
<expr> + <var>
<expr> + z
<expr> + <var> + z
<expr> + y + z
<var> + y + z
x + y + z
```

And Right Associativity

```
<type> ::= <base> -> <type>
         | <base>
<base> ::= ()
```

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

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

Example Parse Tree

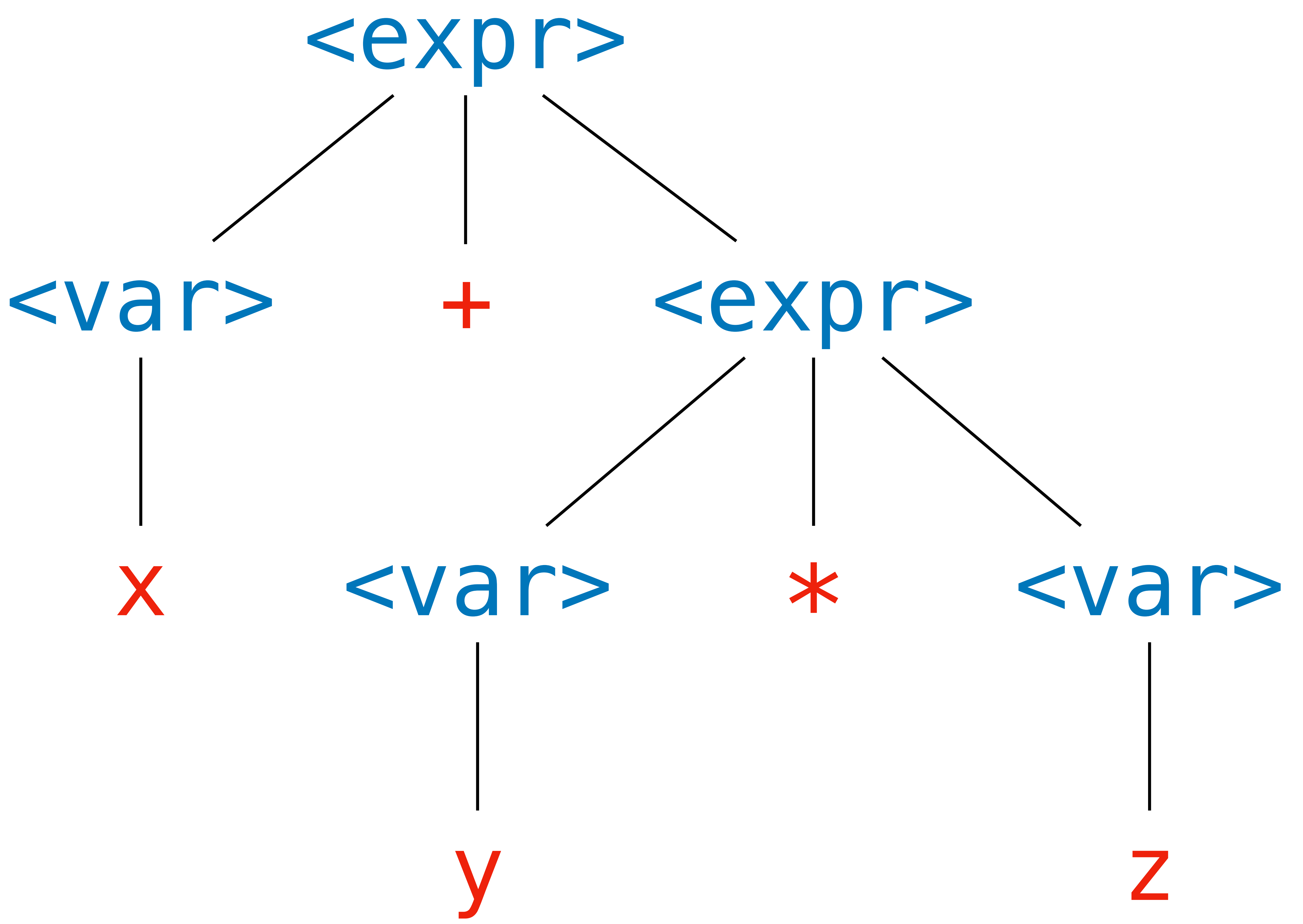
```
<type> ::= <base> -> <type>
         | <base>
<base> ::= ()
```

```
<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 <u>precedence</u> 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"

Dealing with Precedence within the Grammar

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

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

A Note on Associativity and Precedence

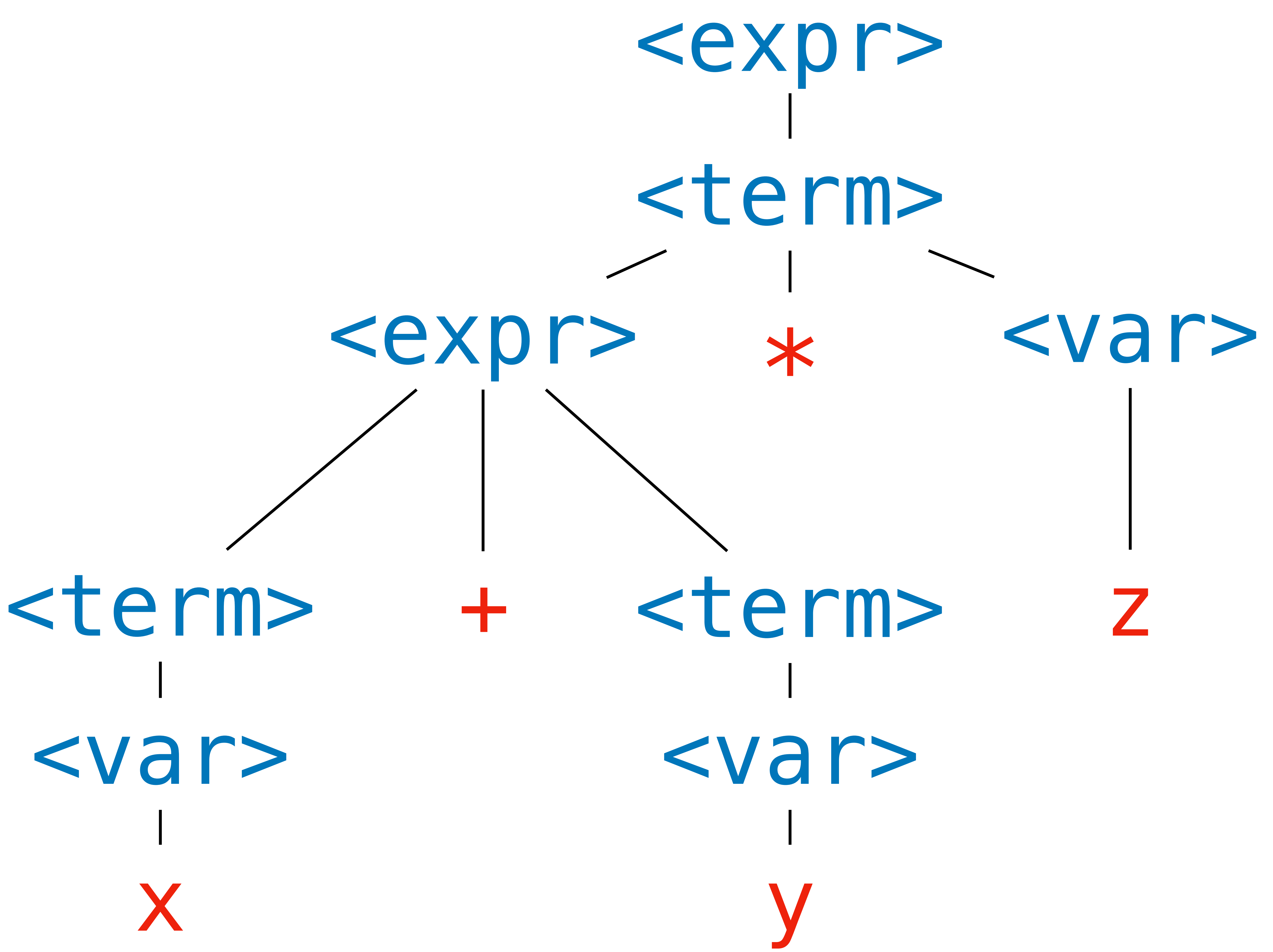
```
%token PLUS
%token MINUS
%token TIMES
%token DIVIDE

%left PLUS, MINUS
%left TIMES, DIVIDE
```

In most situations, we actually won't deal with associativity and precedence in this way

Using a parser generator we'll often be able to *specify* the precedence of an operator

The Issue of Parentheses Returns

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



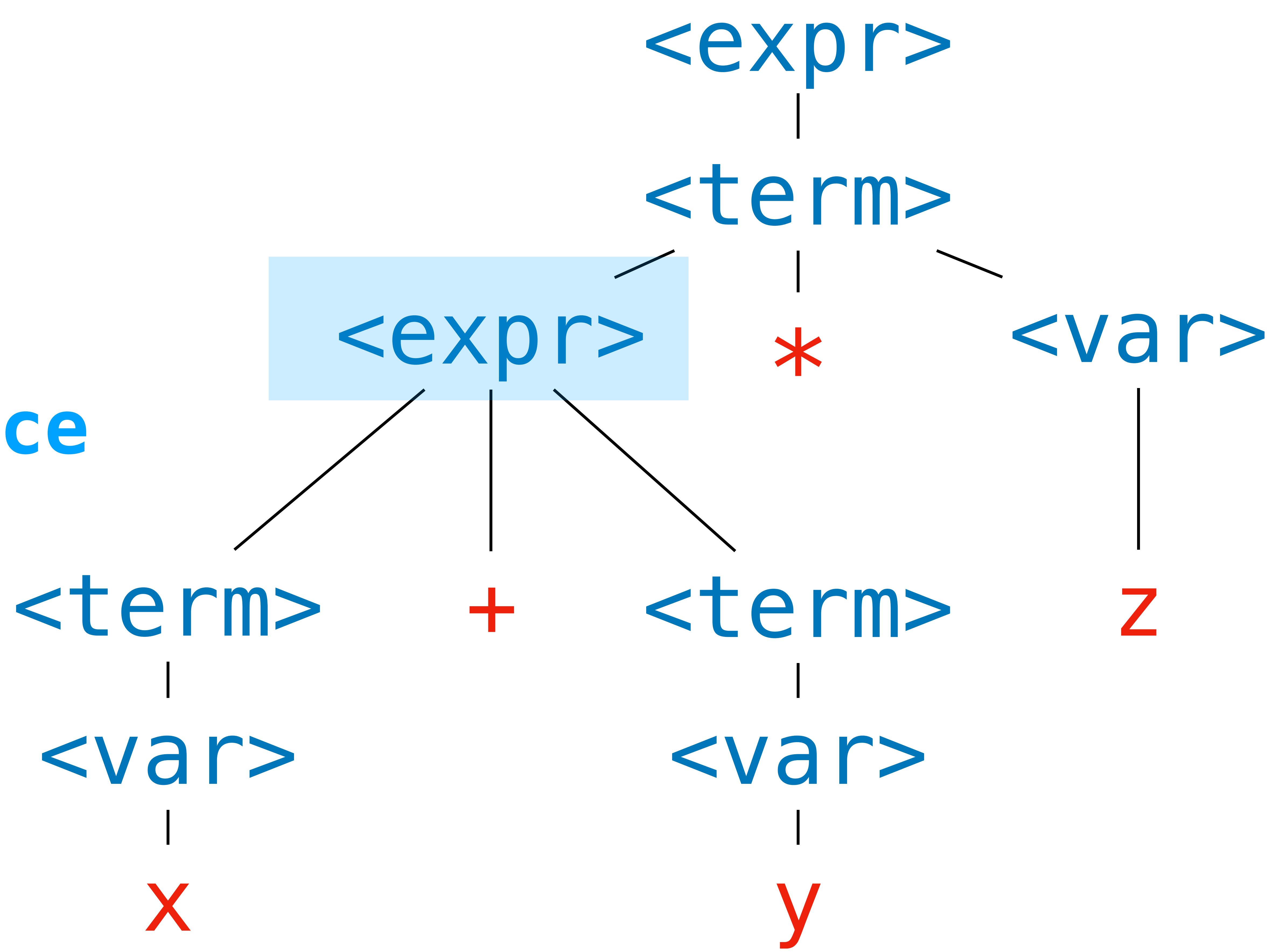
"multiply the sum of x and y with z"

Question. *Can we derive this parse tree?*

The Issue of Parentheses Returns

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

No, we need to introduce parentheses again.



"multiply the sum of x and y with z"

Question. *Can we derive this parse tree?*

Dealing with Parentheses

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

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, production rule)

Example

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

(x + y) * z

Summary

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*

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