Formal Grammar

Concepts of Programming Languages
Lecture 11

CAS CS 320

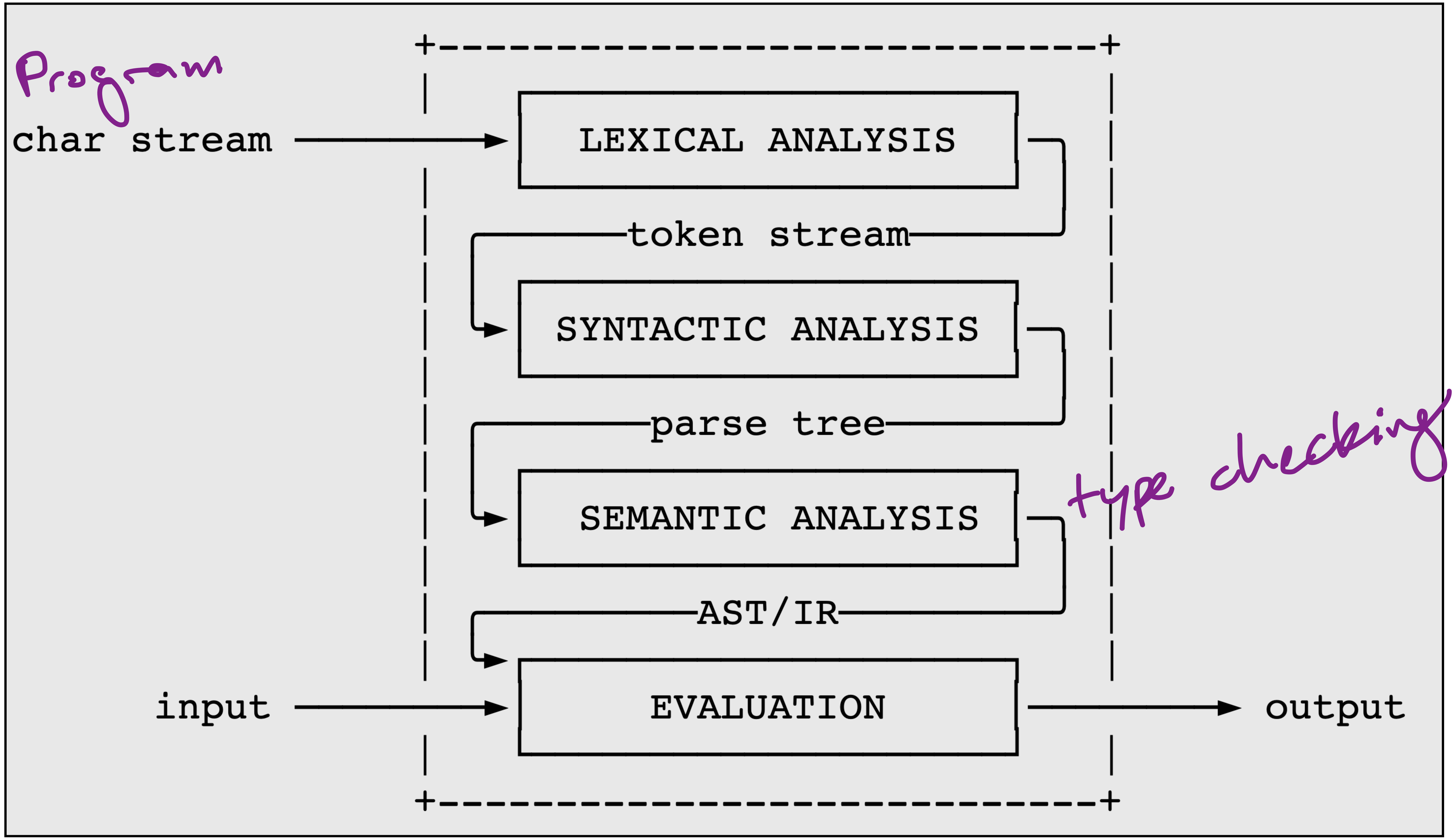
Outline

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

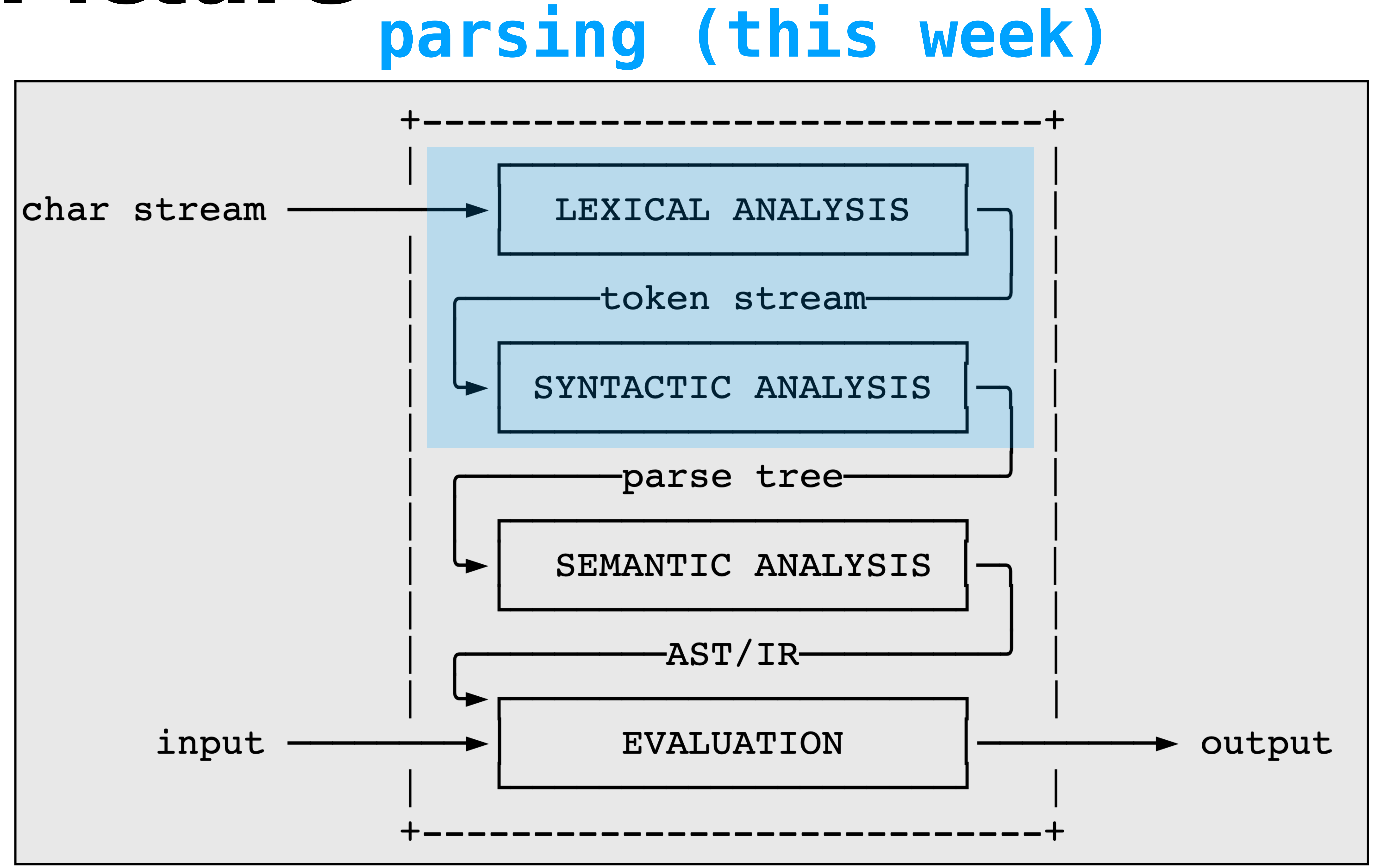
Introduce **formal grammars**, a mathematical framework for thinking about syntax and parsing

The Interpretation Pipeline

The Picture

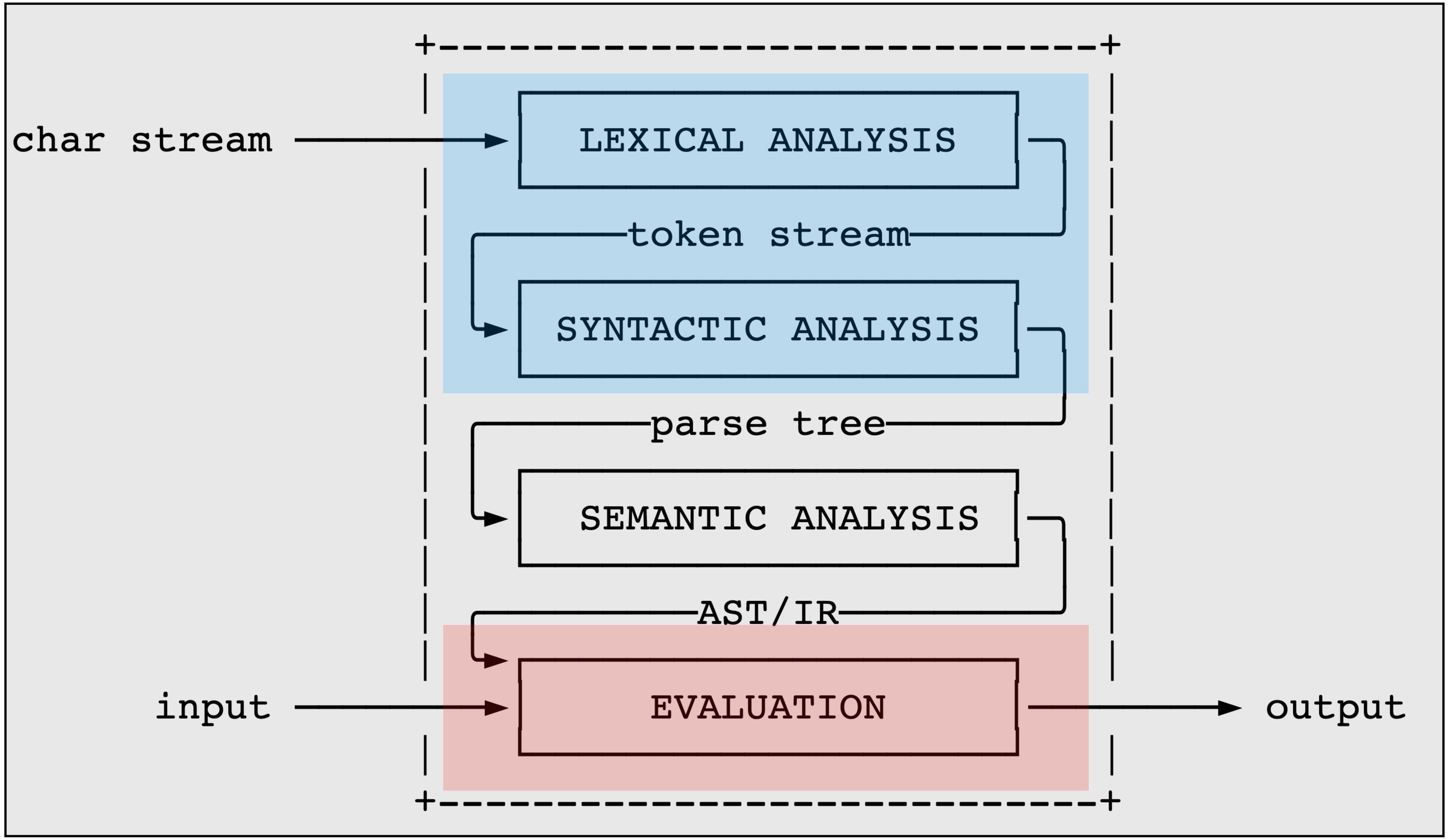

Program
char stream
LEXICAL ANALYSIS
token stream
SYNTACTIC ANALYSIS
parse tree
SEMANTIC ANALYSIS
type checking
AST/IR
input
EVALUATION
output

The Picture


parsing (this week)
char stream
LEXICAL ANALYSIS
token stream
SYNTACTIC ANALYSIS
parse tree
SEMANTIC ANALYSIS
AST/IR
input
EVALUATION
output

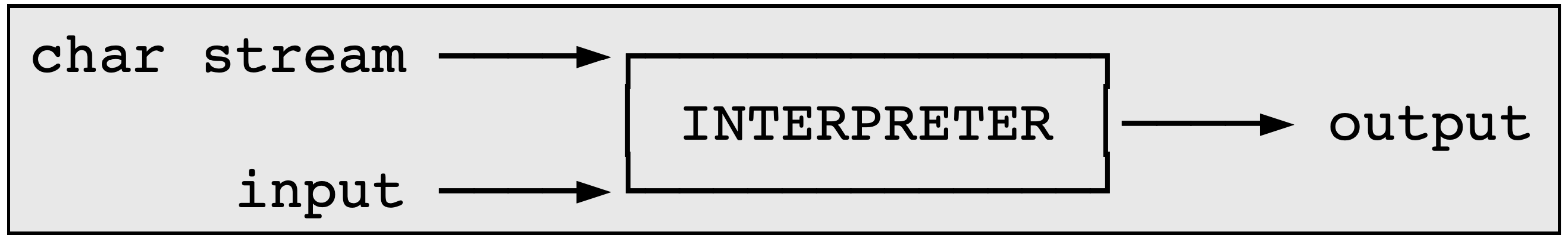
The Picture

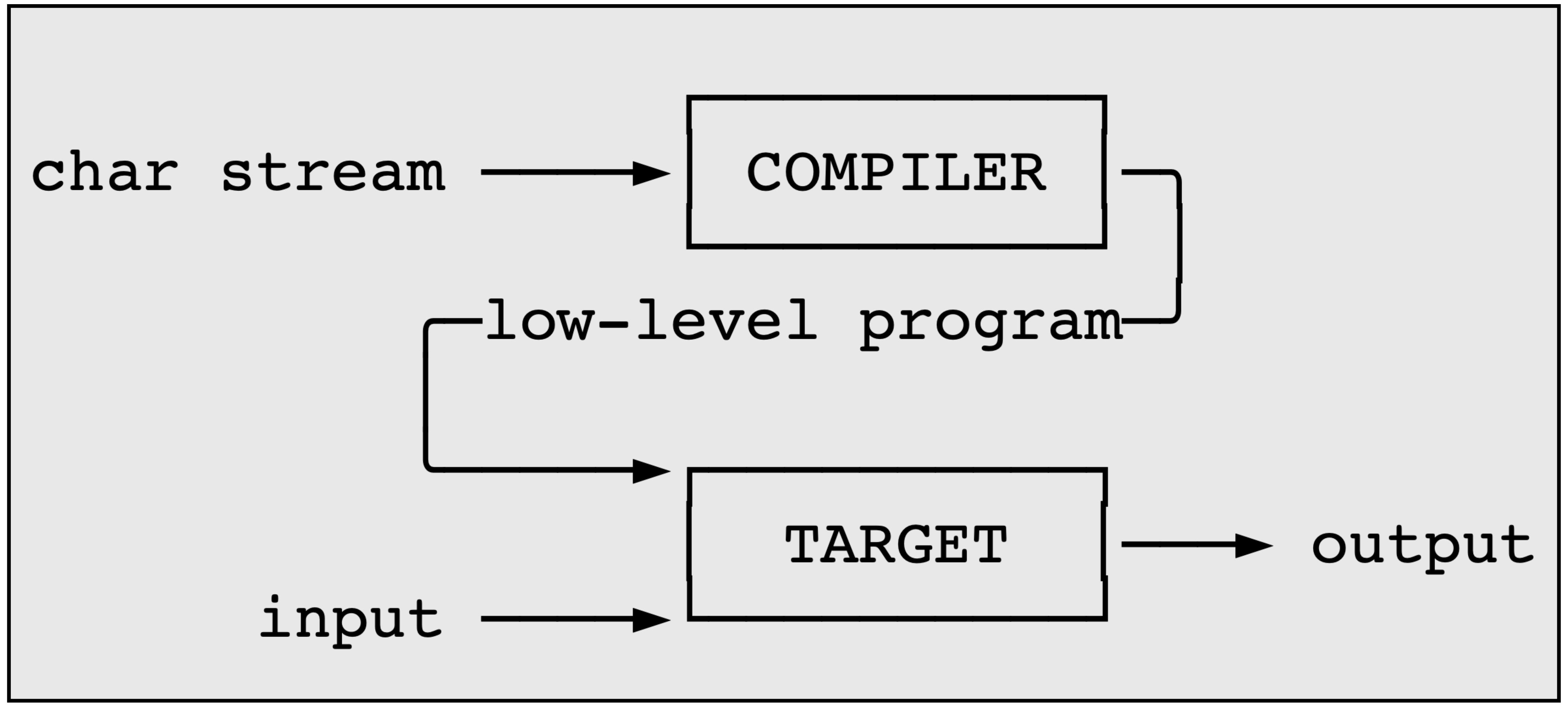
parsing (this week)



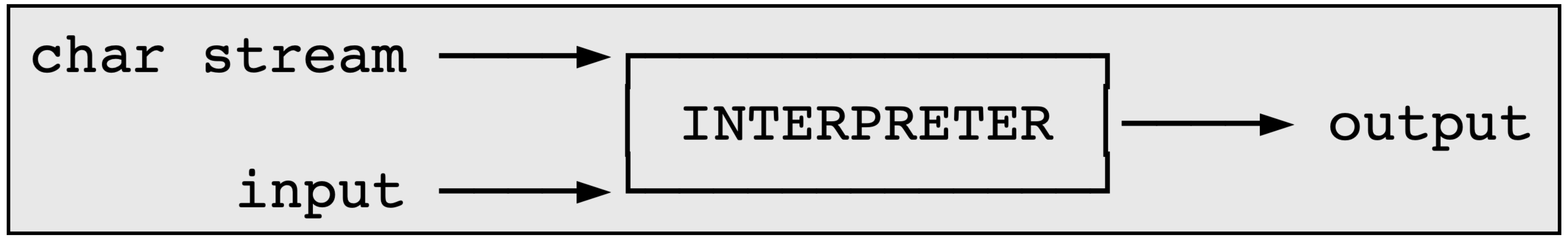
semantics (after the break)

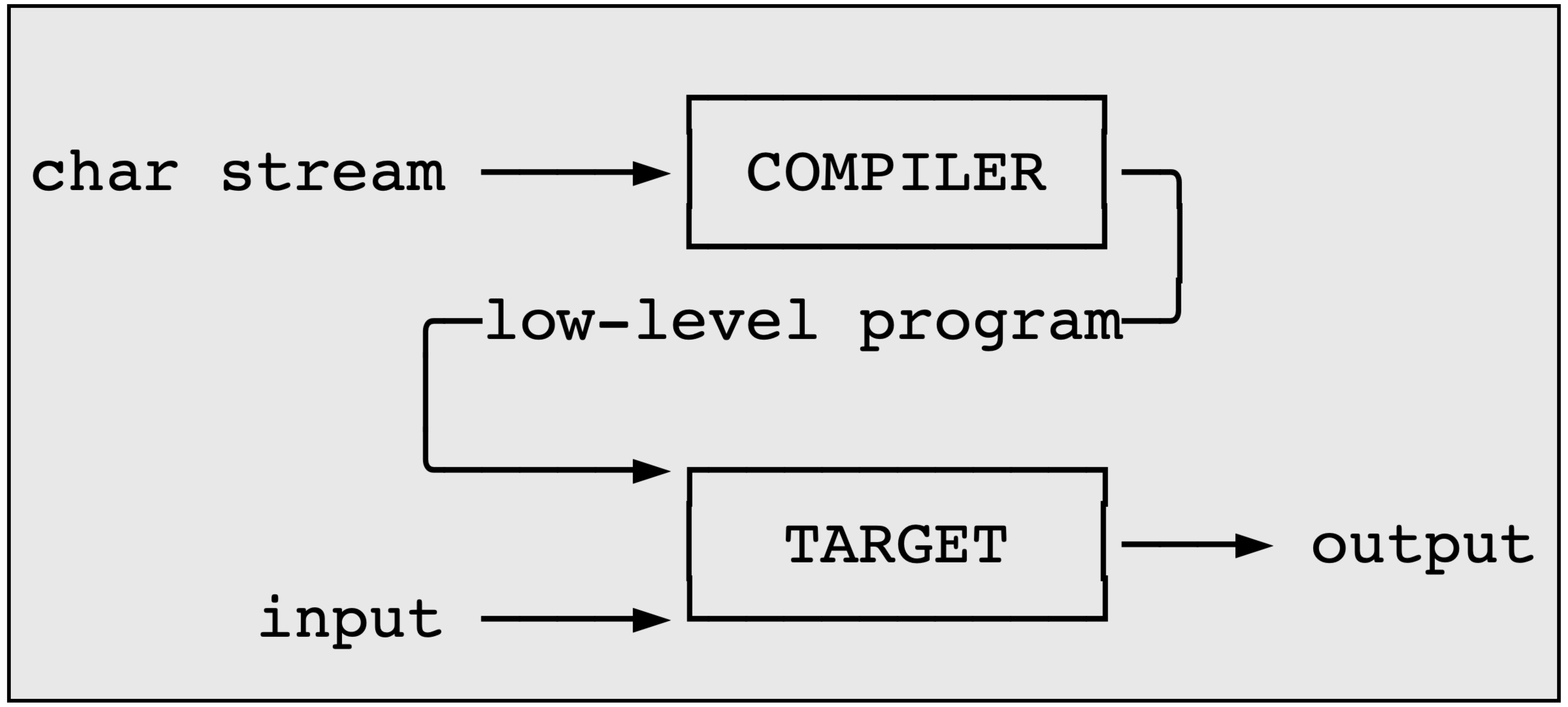
A Note on Compilation





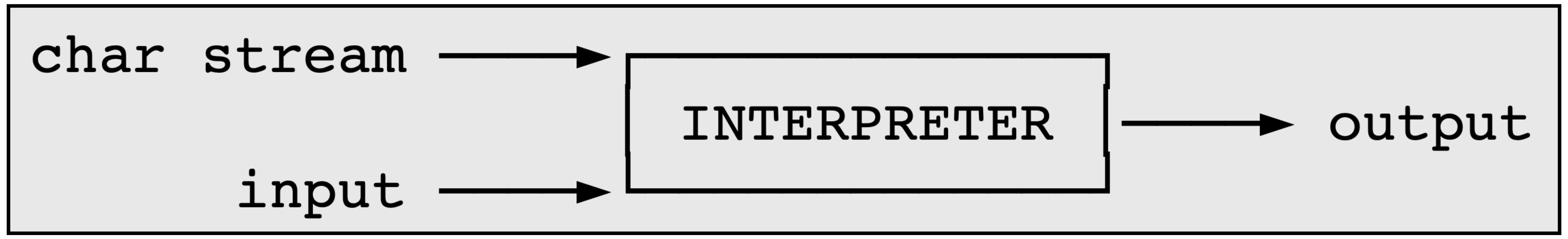
A Note on Compilation

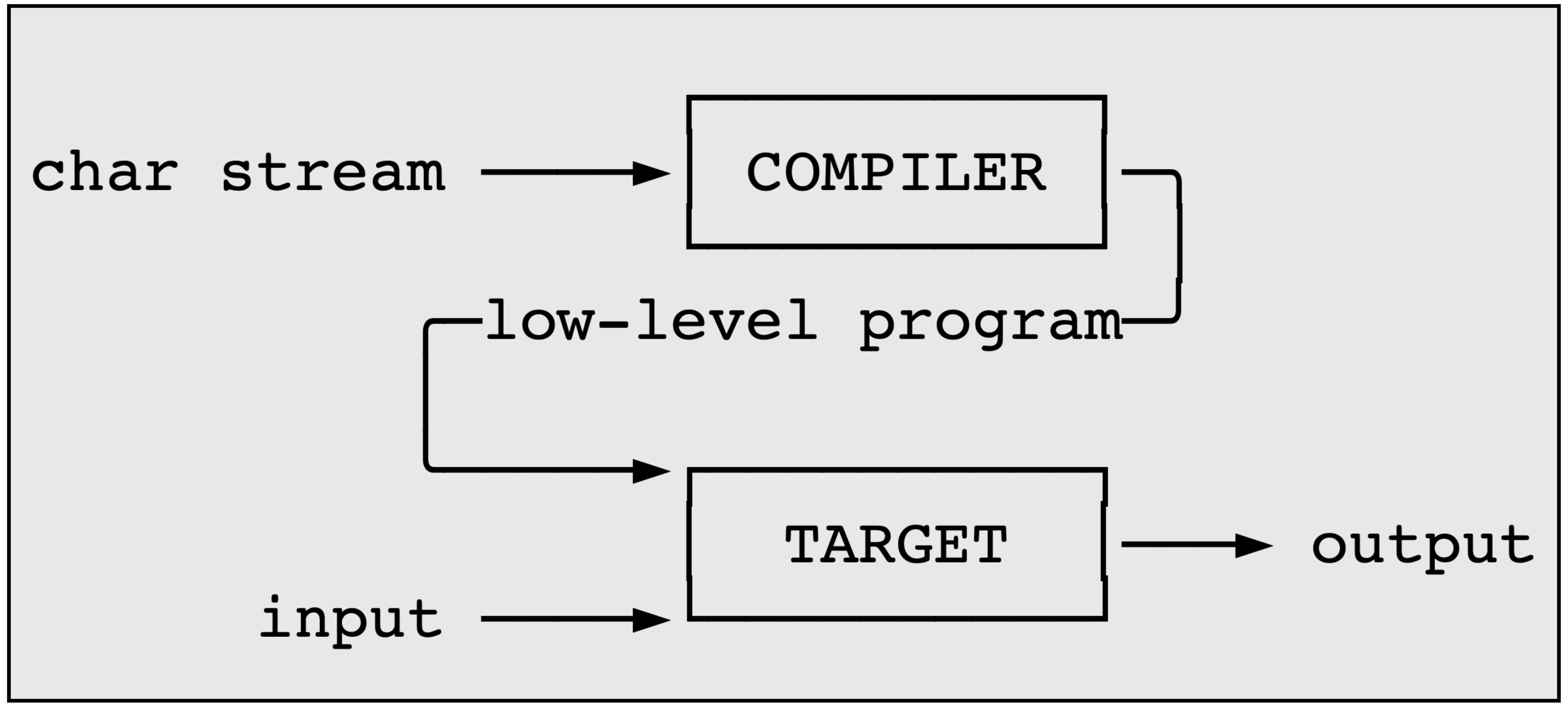




We will be building programs that directly read an evaluate programs (**interpreters**)

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

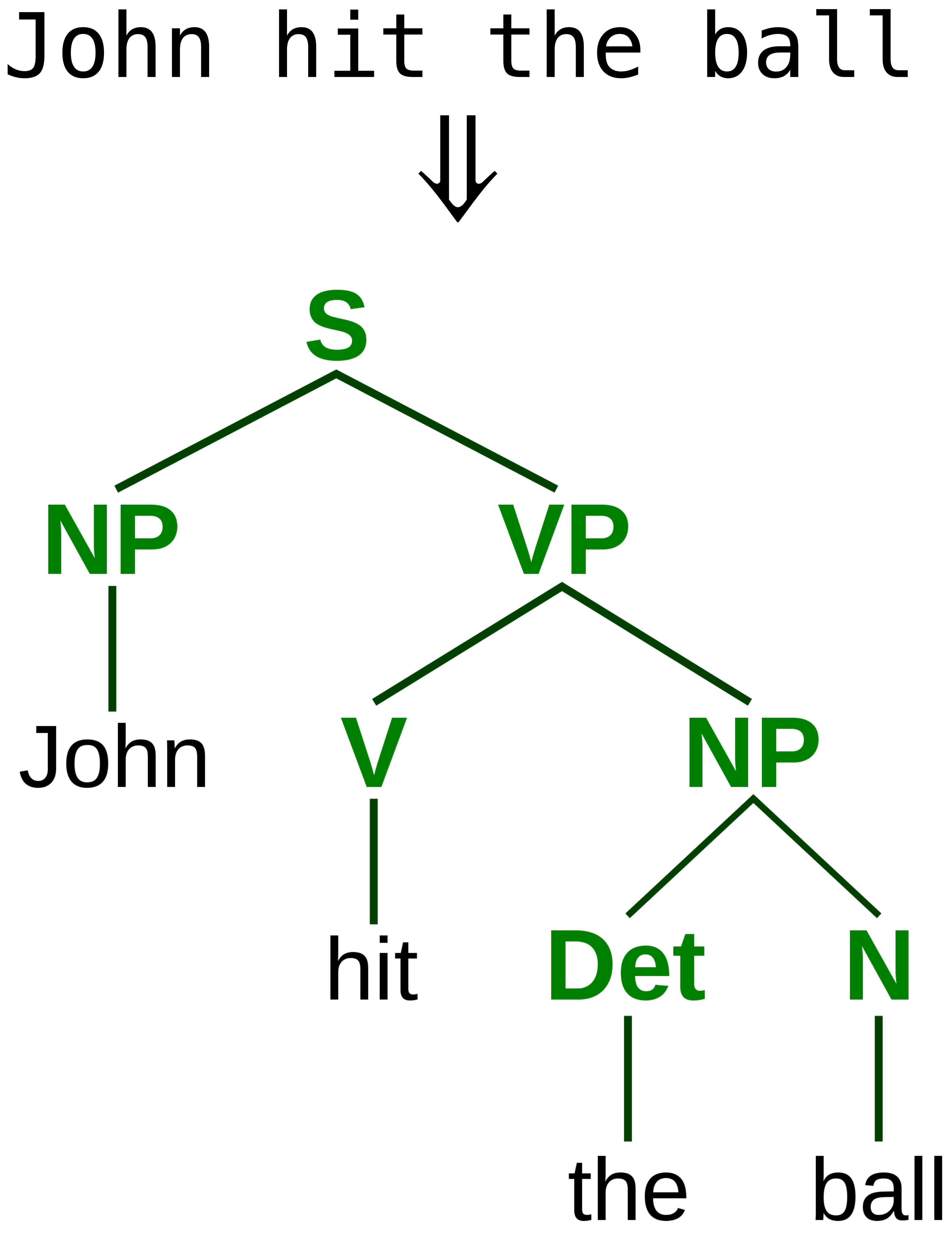
We need a formal language for describing the syntax of programming languages

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

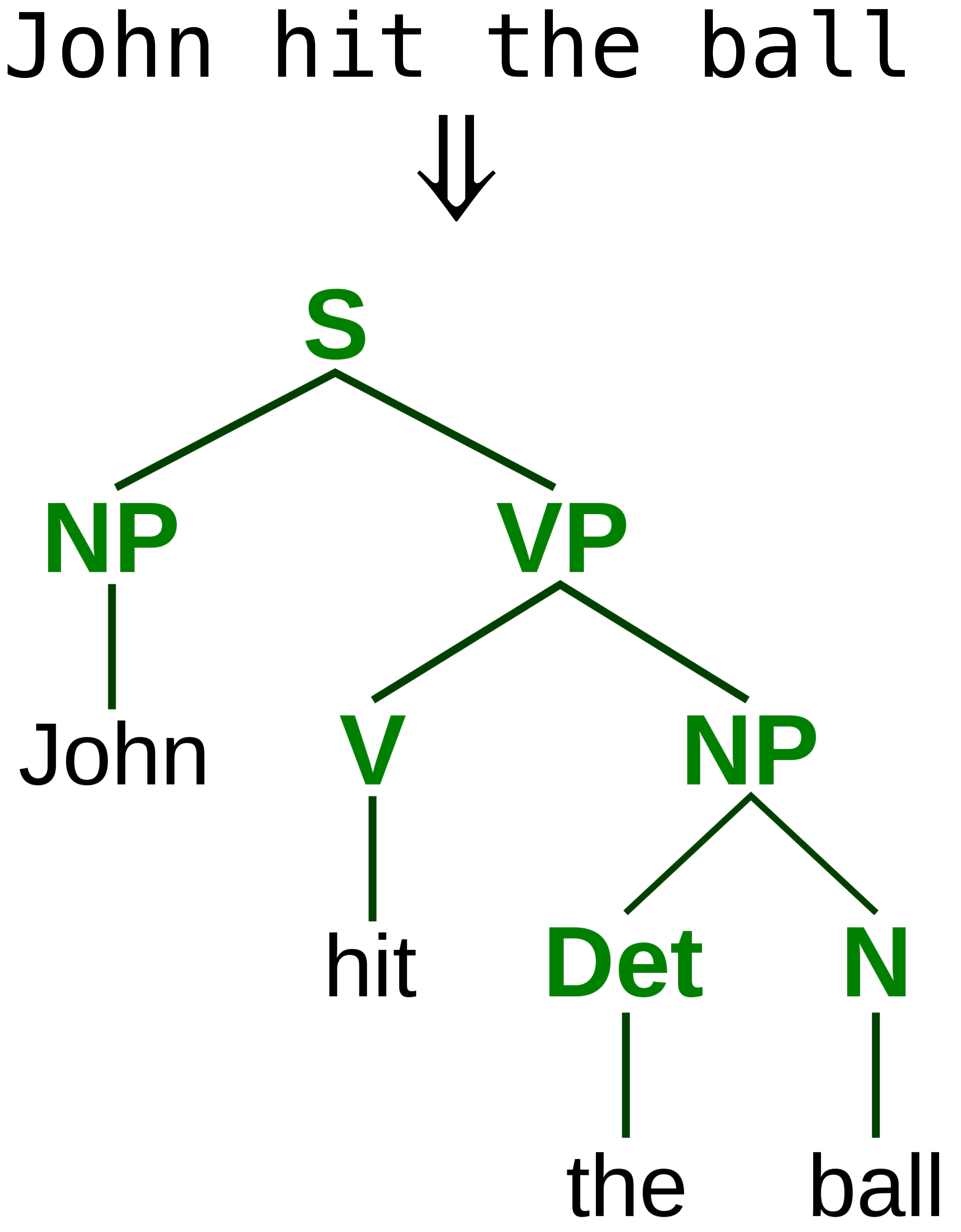
What is Grammar?



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

What is Grammar?

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

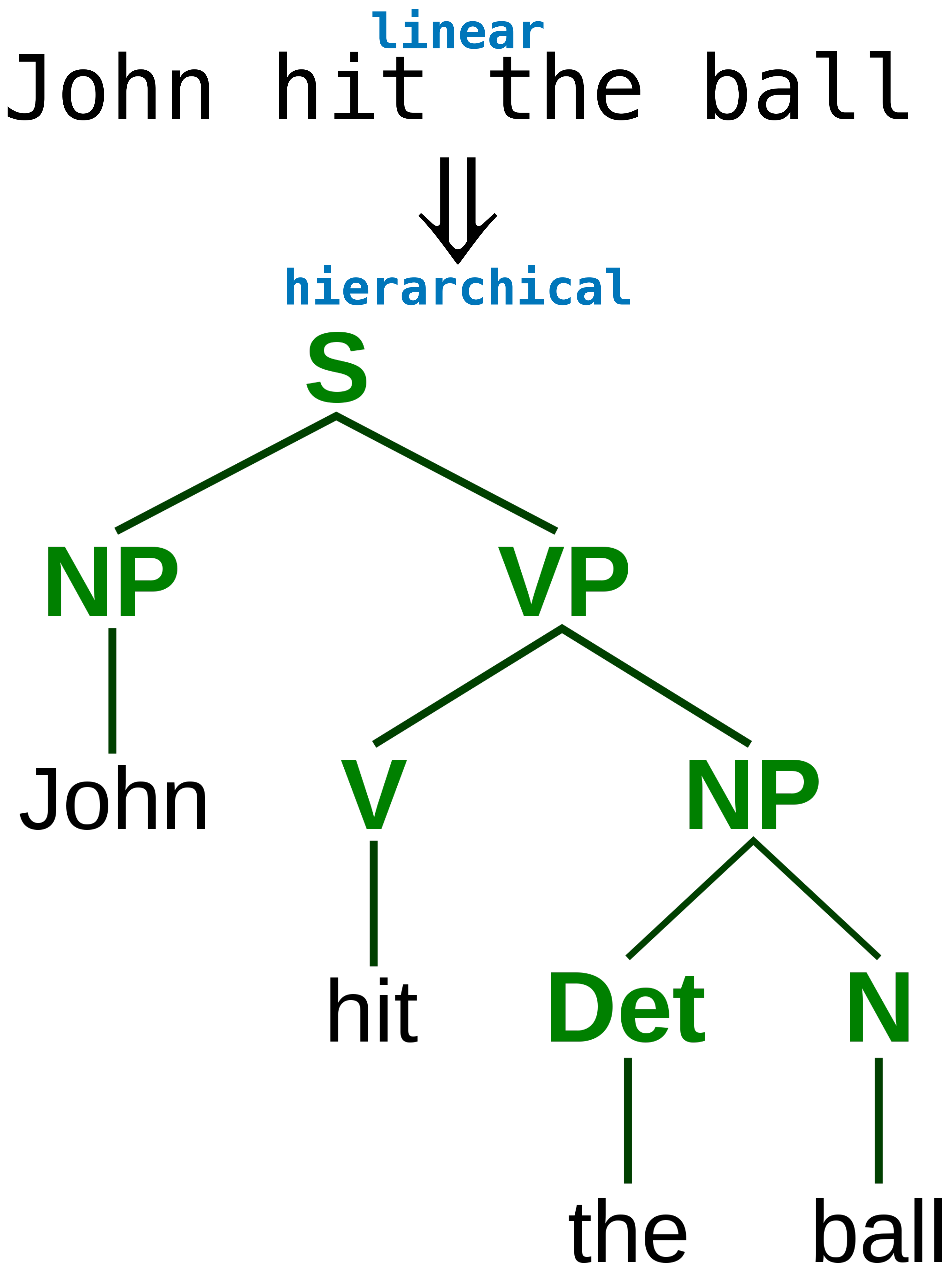


https://commons.wikimedia.org/wiki/File:ParseTree.svg

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

Formal grammars 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

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

<u>Formal grammars</u> 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

<u>Formal grammars</u> 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

<u>Formal grammars</u> 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

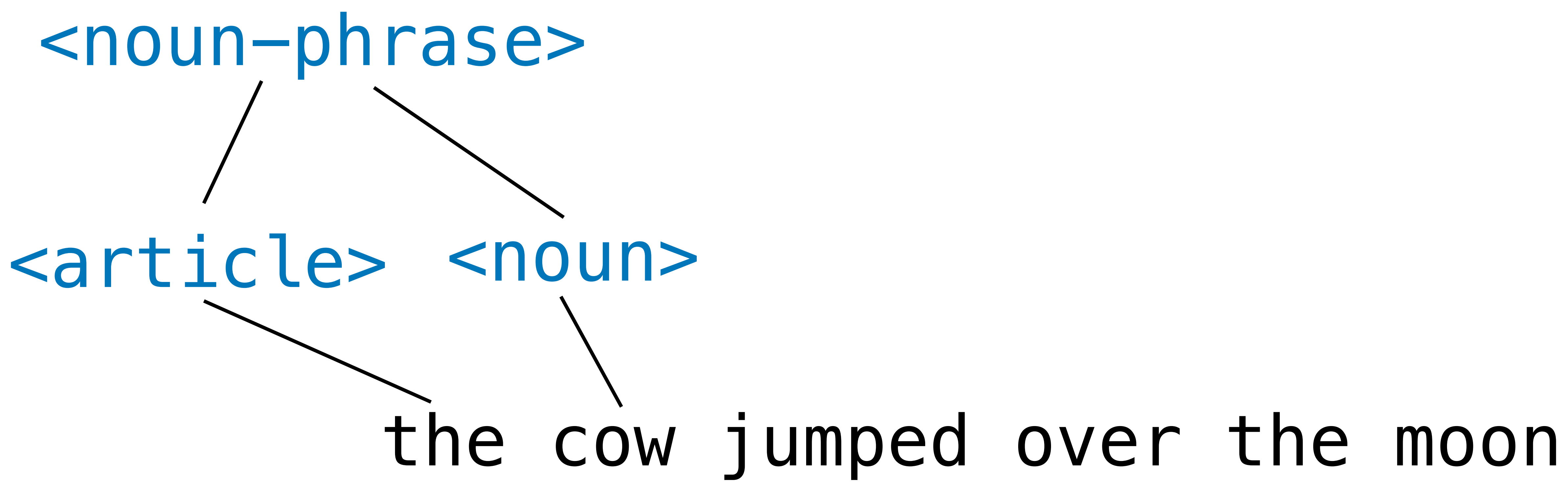
<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>
<noun-phrase> jumped <prep-phrase>
<noun-phrase> jumped <prep> <noun-phrase>
<noun-phrase> jumped over <noun-phrase>
<noun-phrase> jumped over <article> <noun>
<noun-phrase> jumped over <article> moon
<noun-phrase> jumped over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

<article> <noun>

the cow jumped over the moon

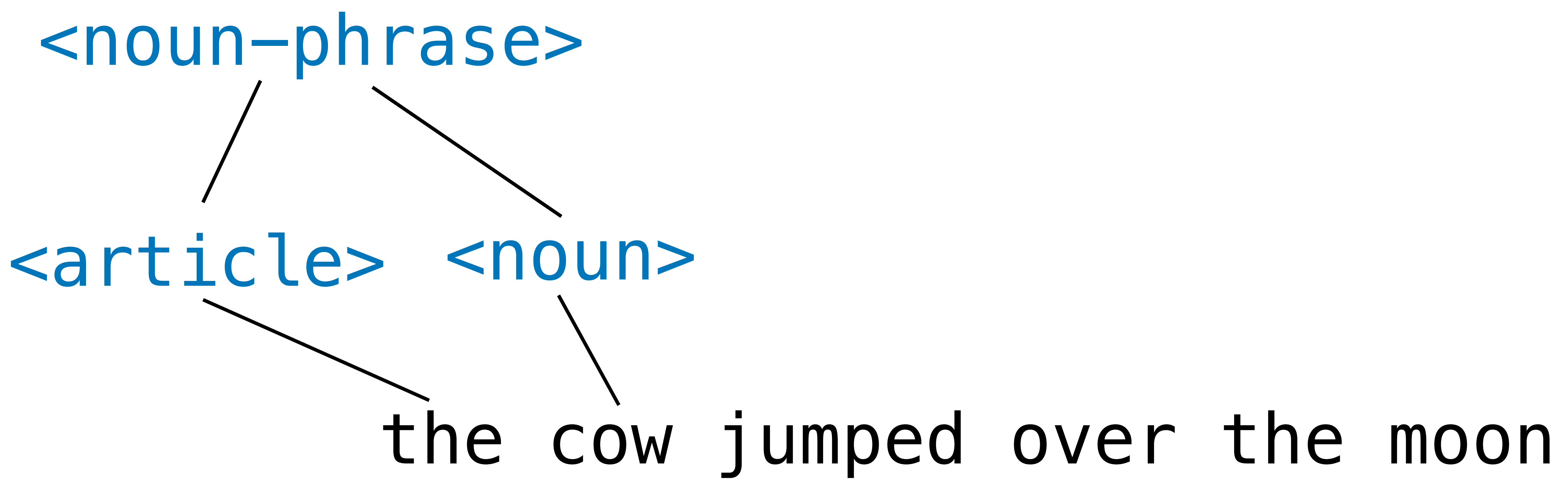
A Derivation

<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



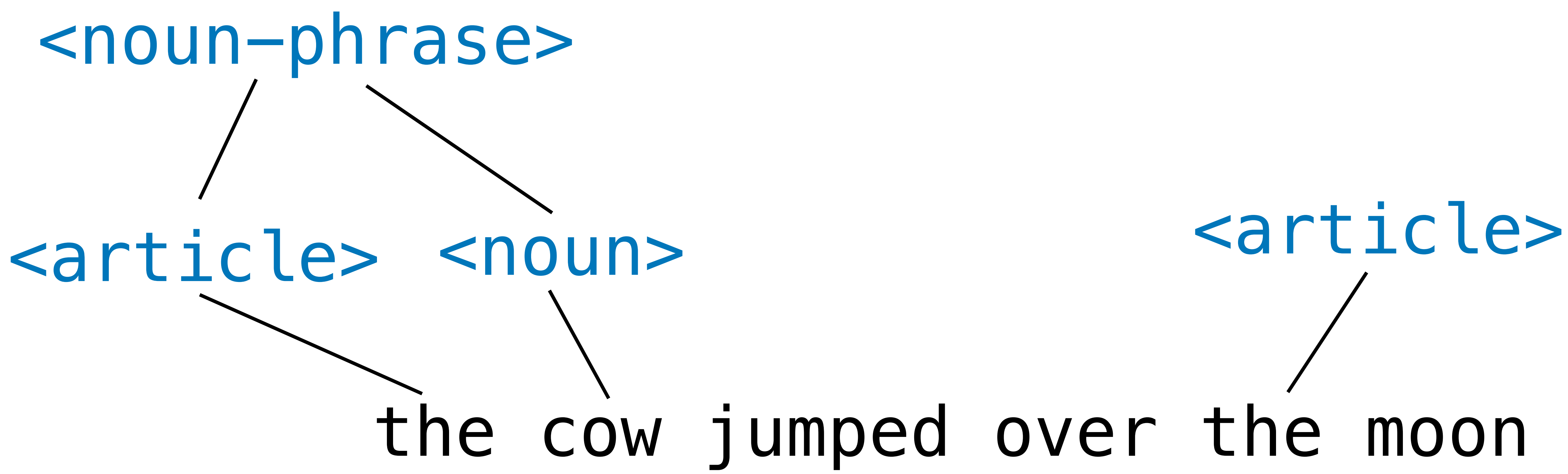
A Derivation

<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



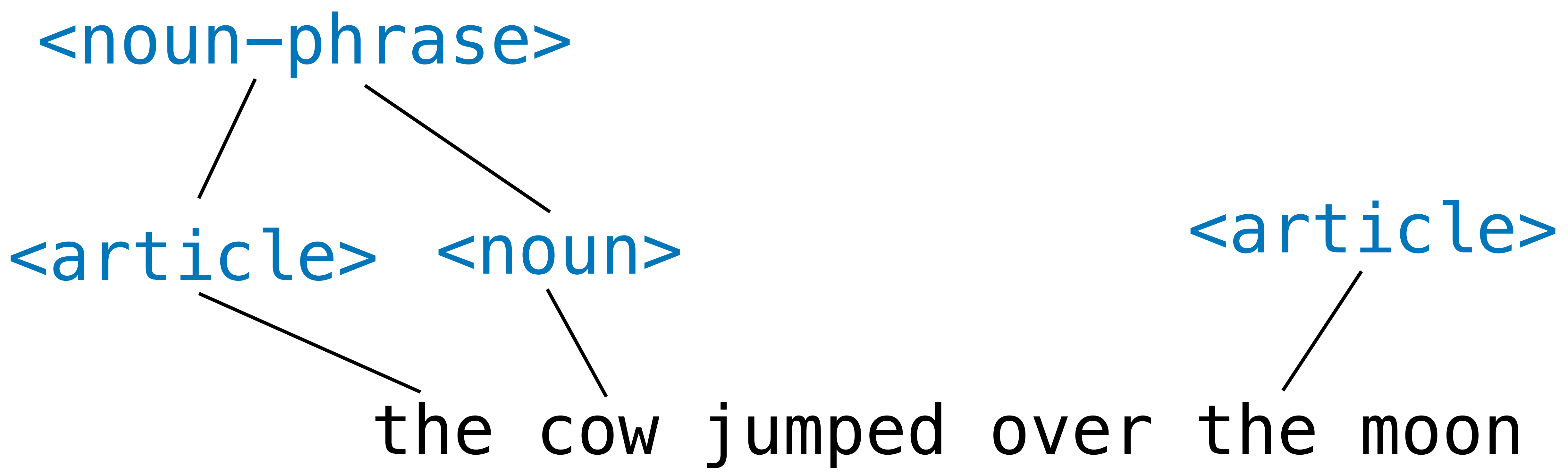
A Derivation

<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>



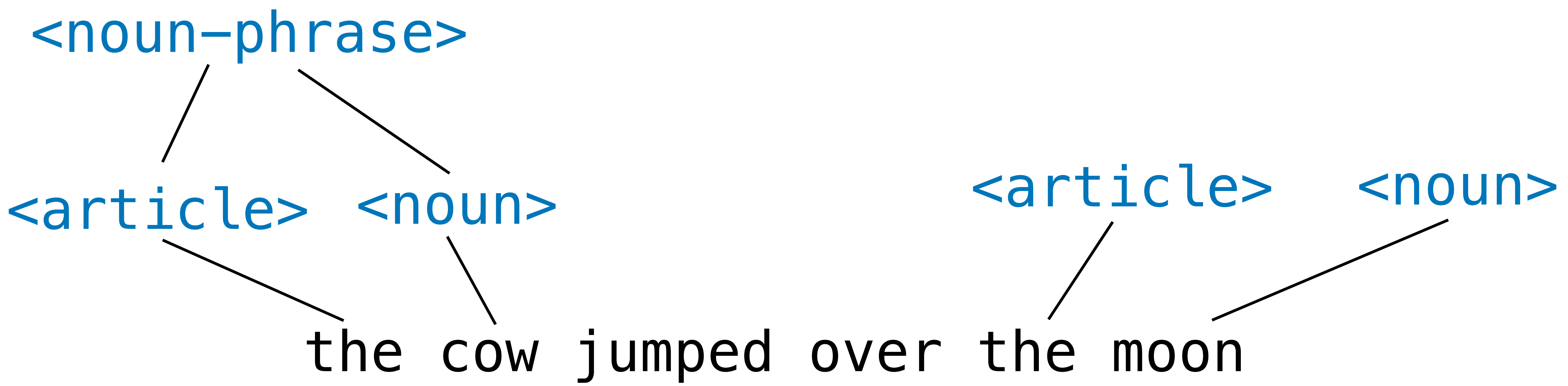
A Derivation

<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>

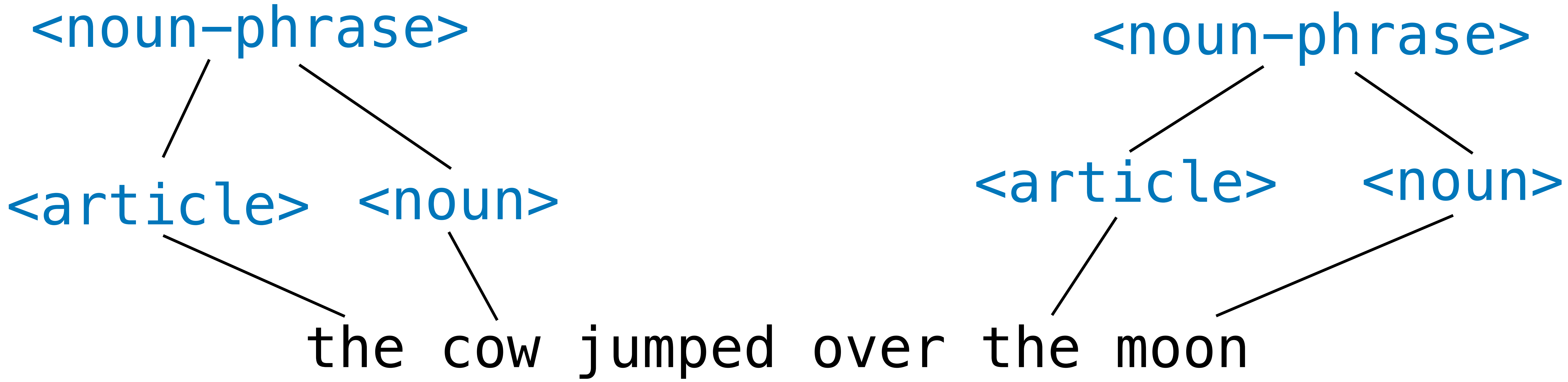


A Derivation

<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>

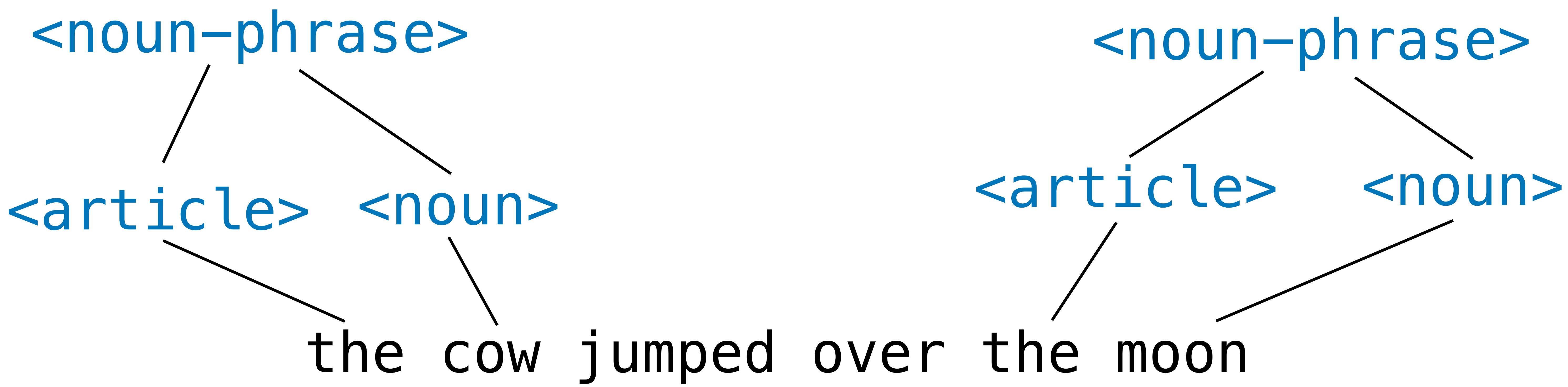
A Derivation

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



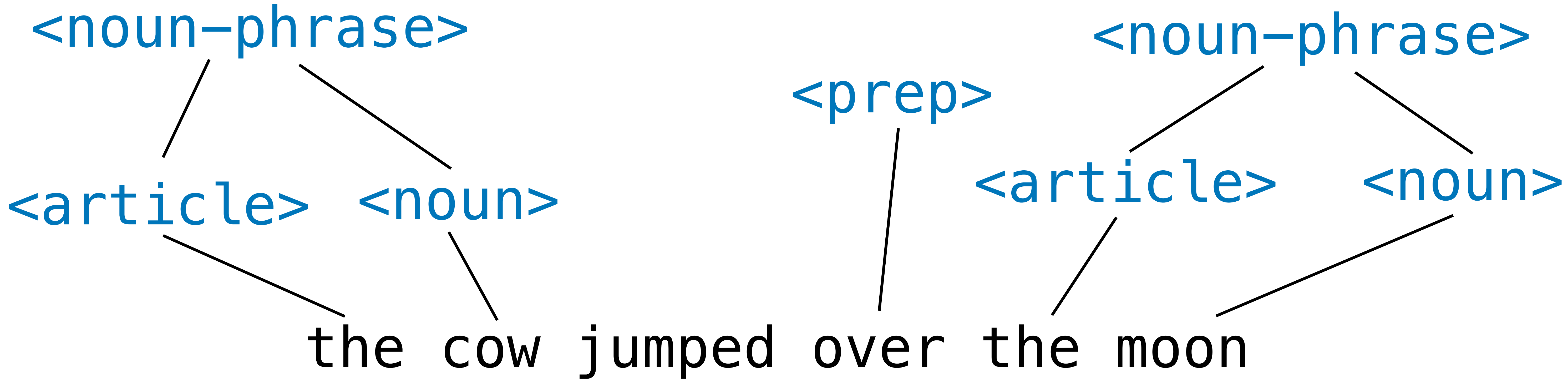
A Derivation

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



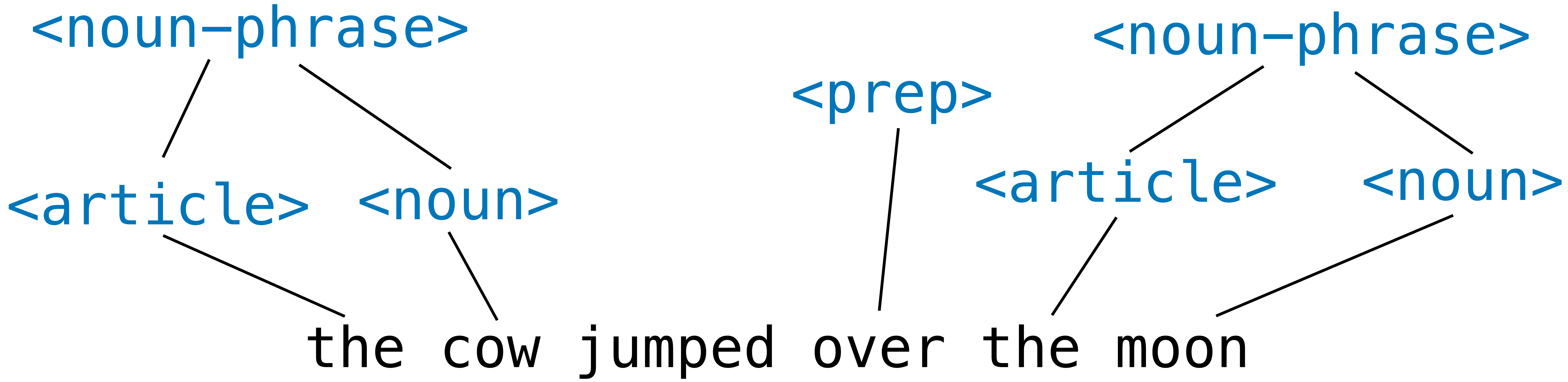
A Derivation

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



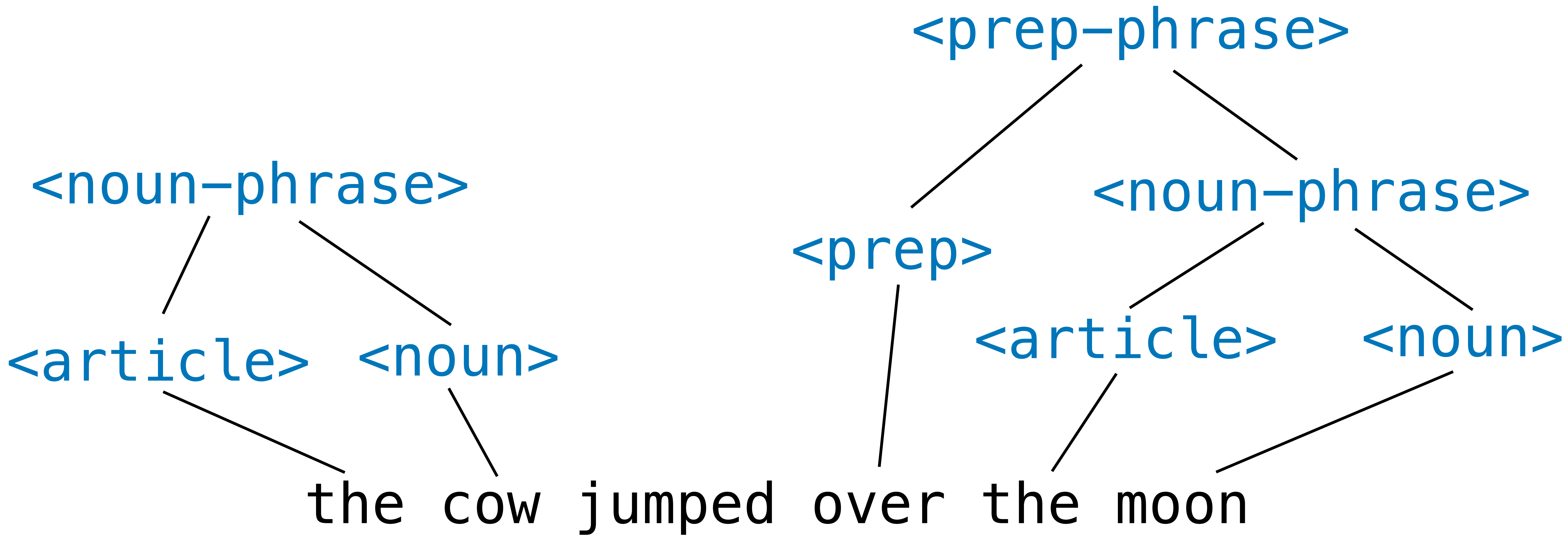
A Derivation

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



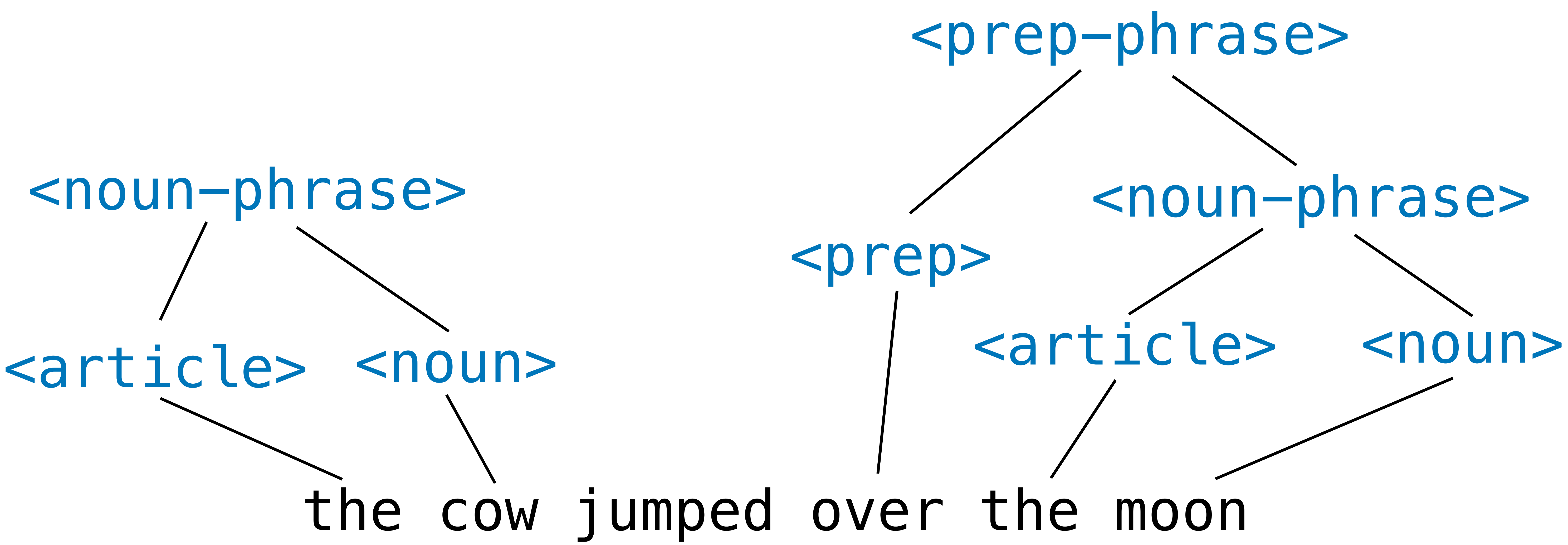
A Derivation

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



A Derivation

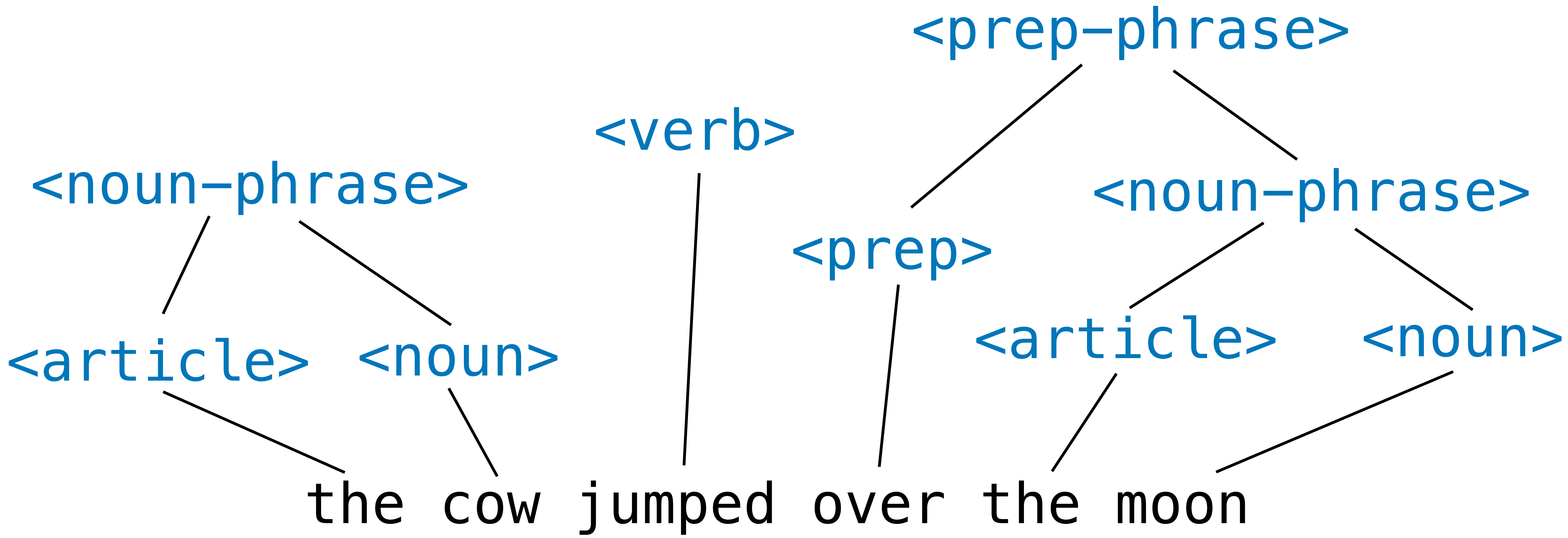
<sentence>
<noun-phrase> <verb-phrase>
<noun-phrase> <verb> <prep-phrase>



A Derivation

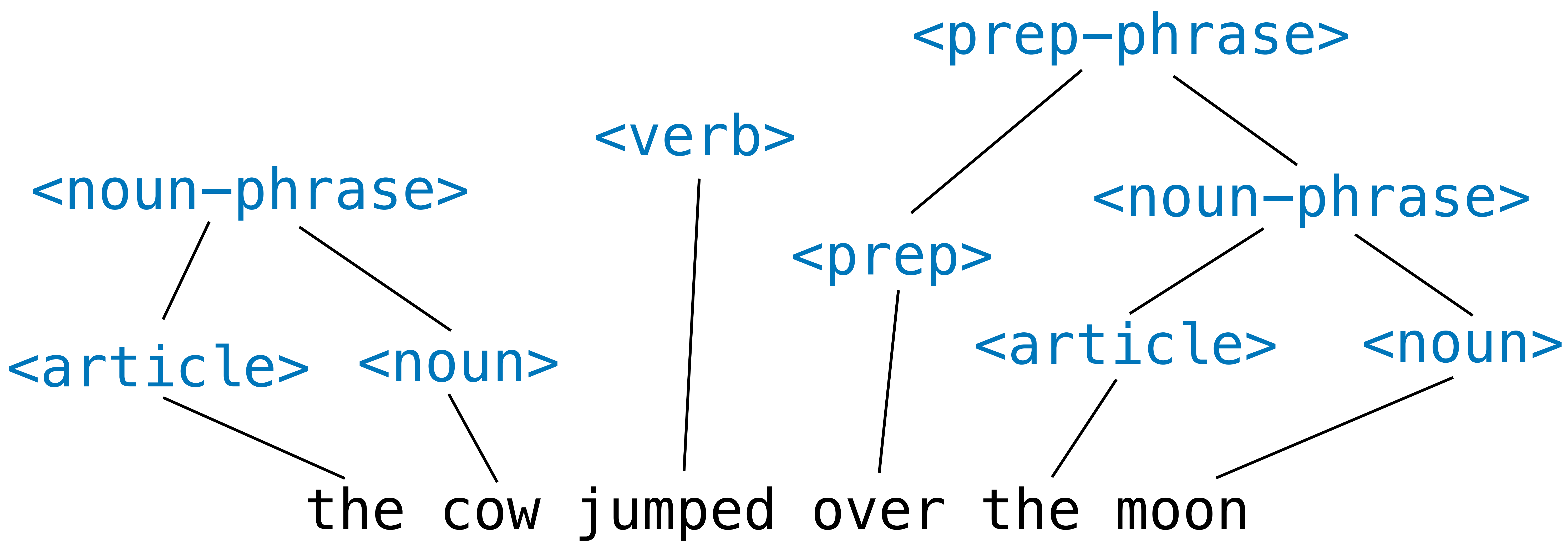
<sentence>

<noun-phrase> <verb-phrase>

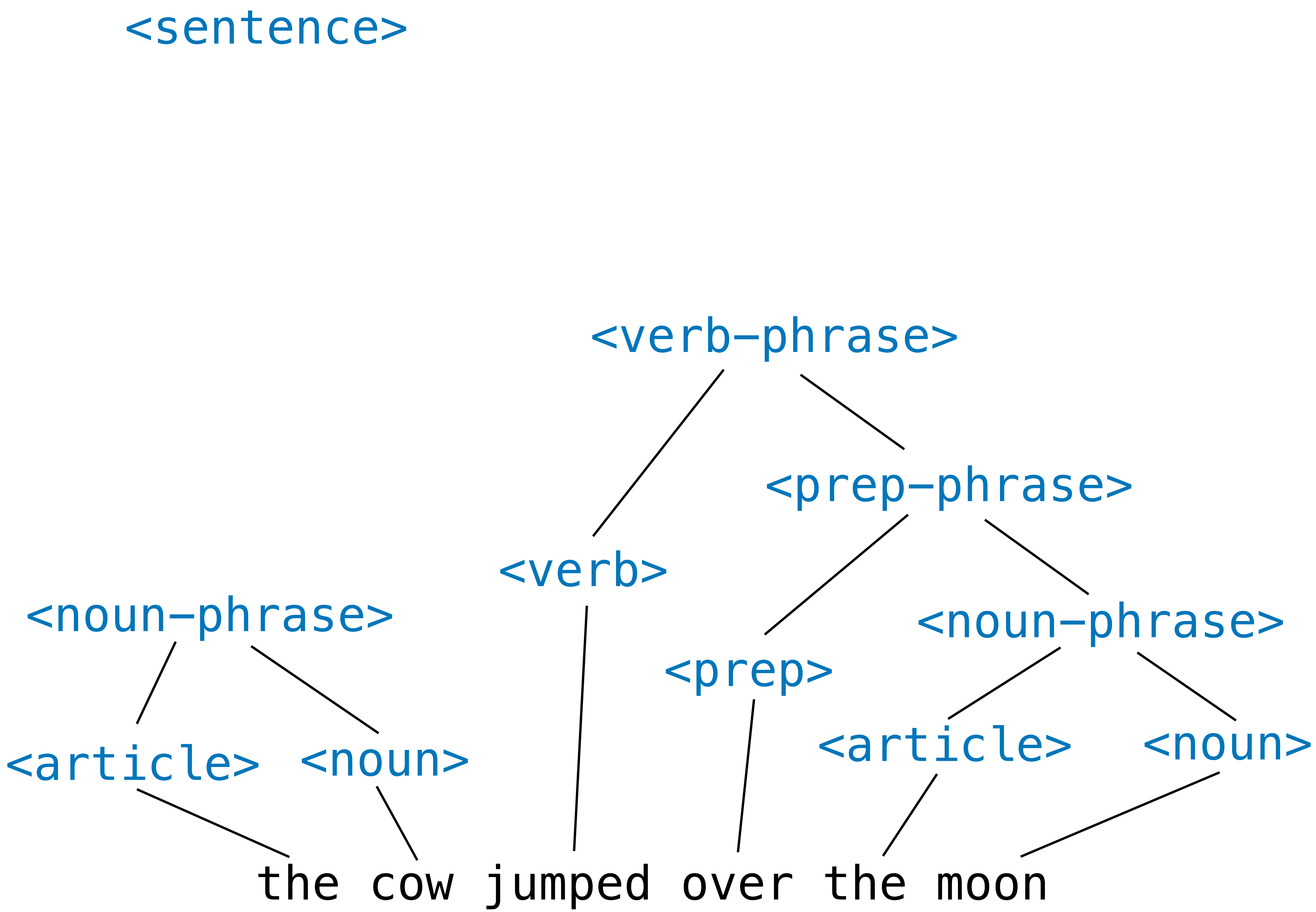


A Derivation

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

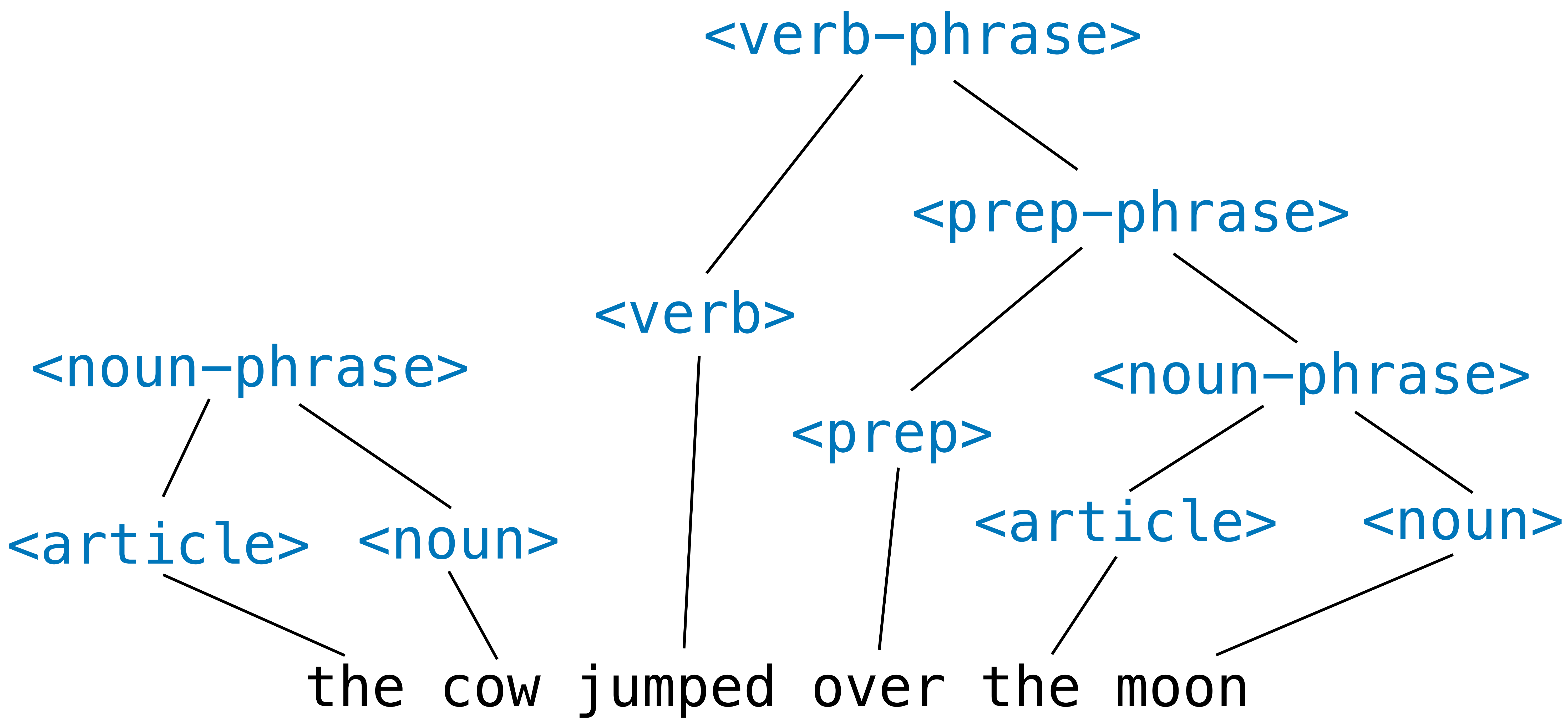


A Derivation

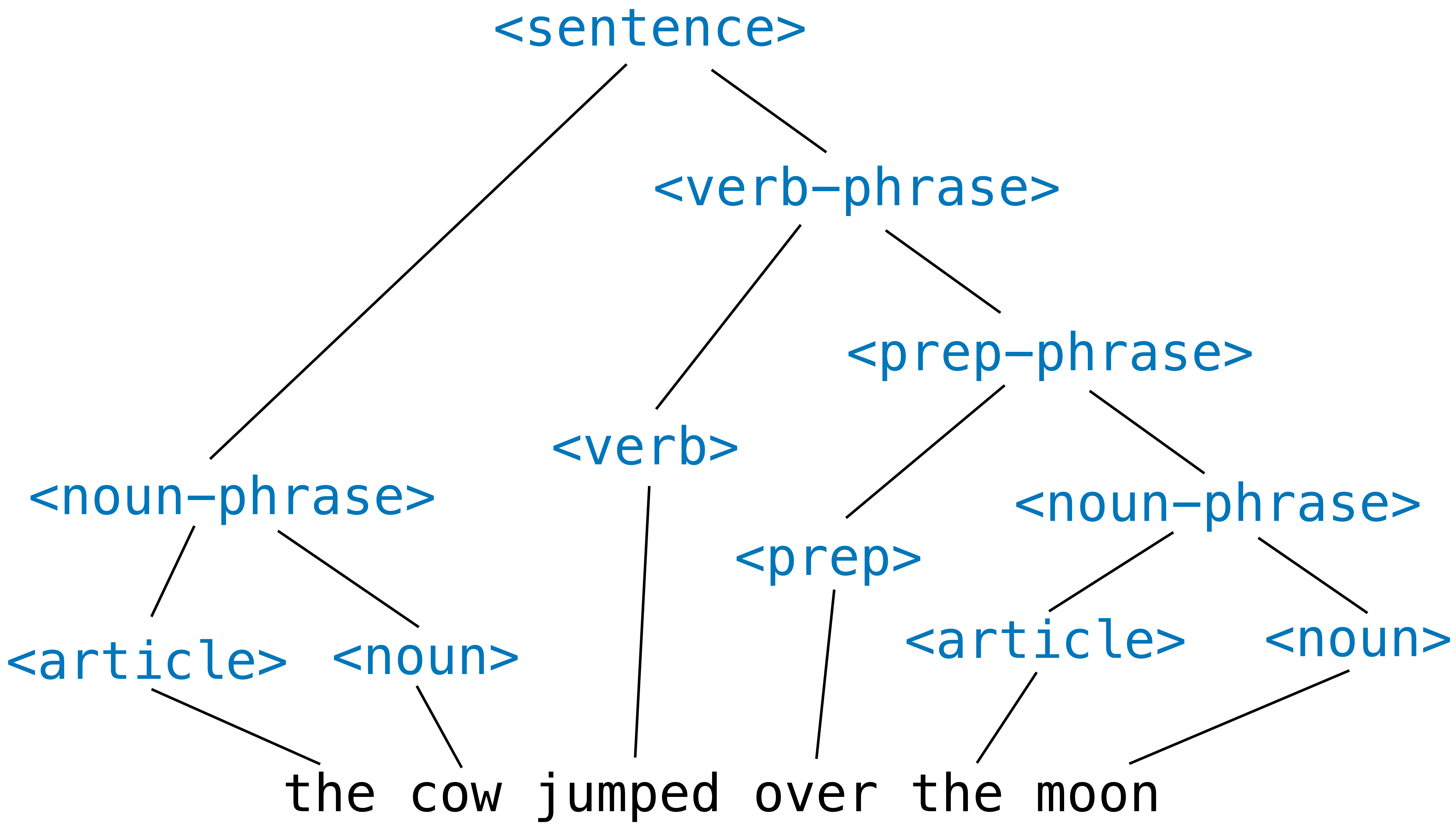


A Derivation

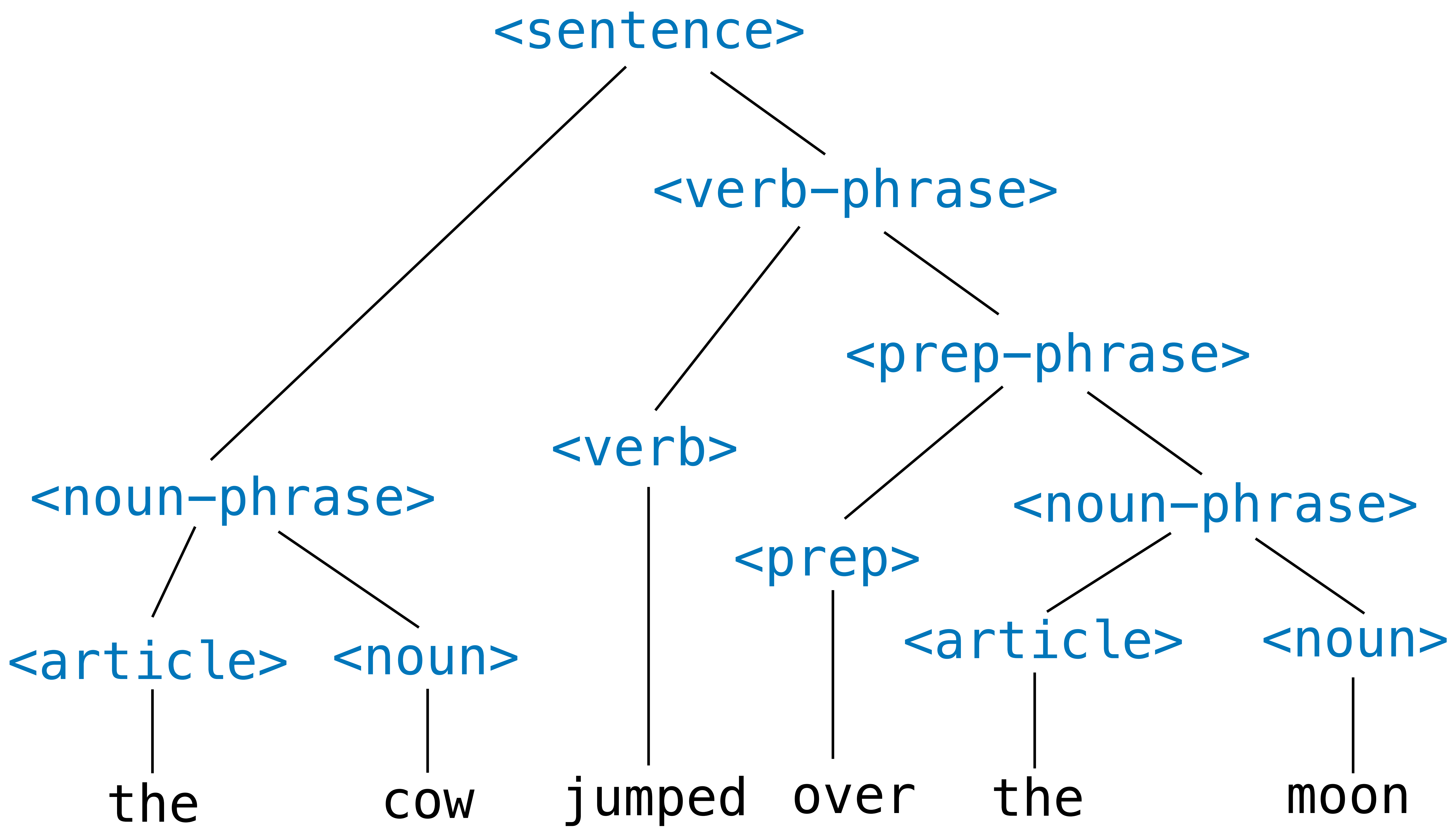
<sentence>



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)

NT symbols

<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)

terminal symbols

<noun-phrase> $\boxed{\text{jumped}}$ $\boxed{\text{over}}$ <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)

sent. form

<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

"or"

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

"can be replaced with"

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

alternative

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

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

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

not $\underbrace{\text{and} \quad x \quad x}_{\text{expr}}$ $\underbrace{\text{not} \quad y}_{\text{expr}}$

```
<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

non-terminal

```
<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
```

terminals

What are the nonterminal and terminal symbols of this grammar?

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

<expr> ::=
<op2><expr><expr>

<var> ::= x

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
```

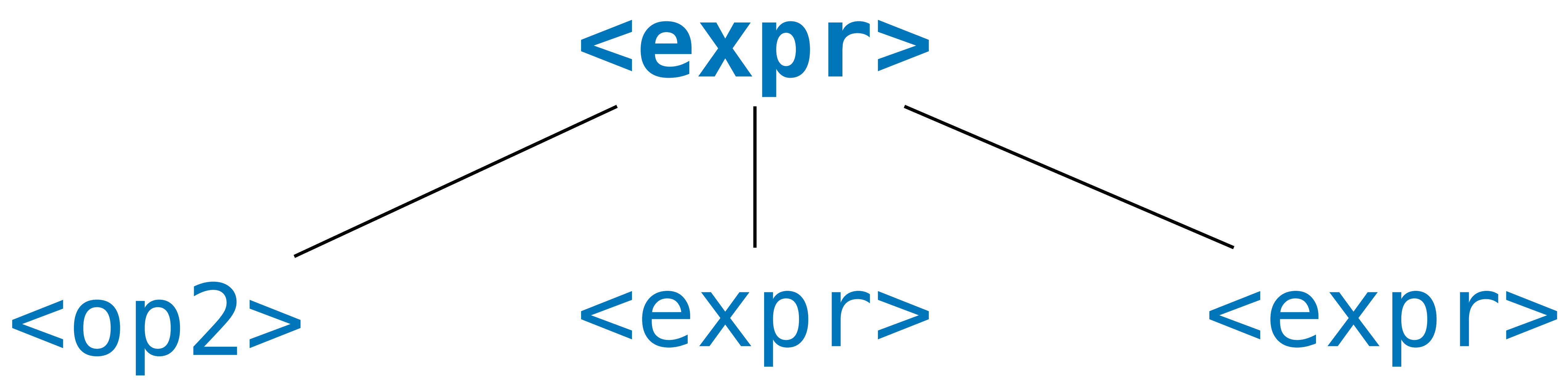
<expr>

<expr>

Derivations and Parse Trees

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

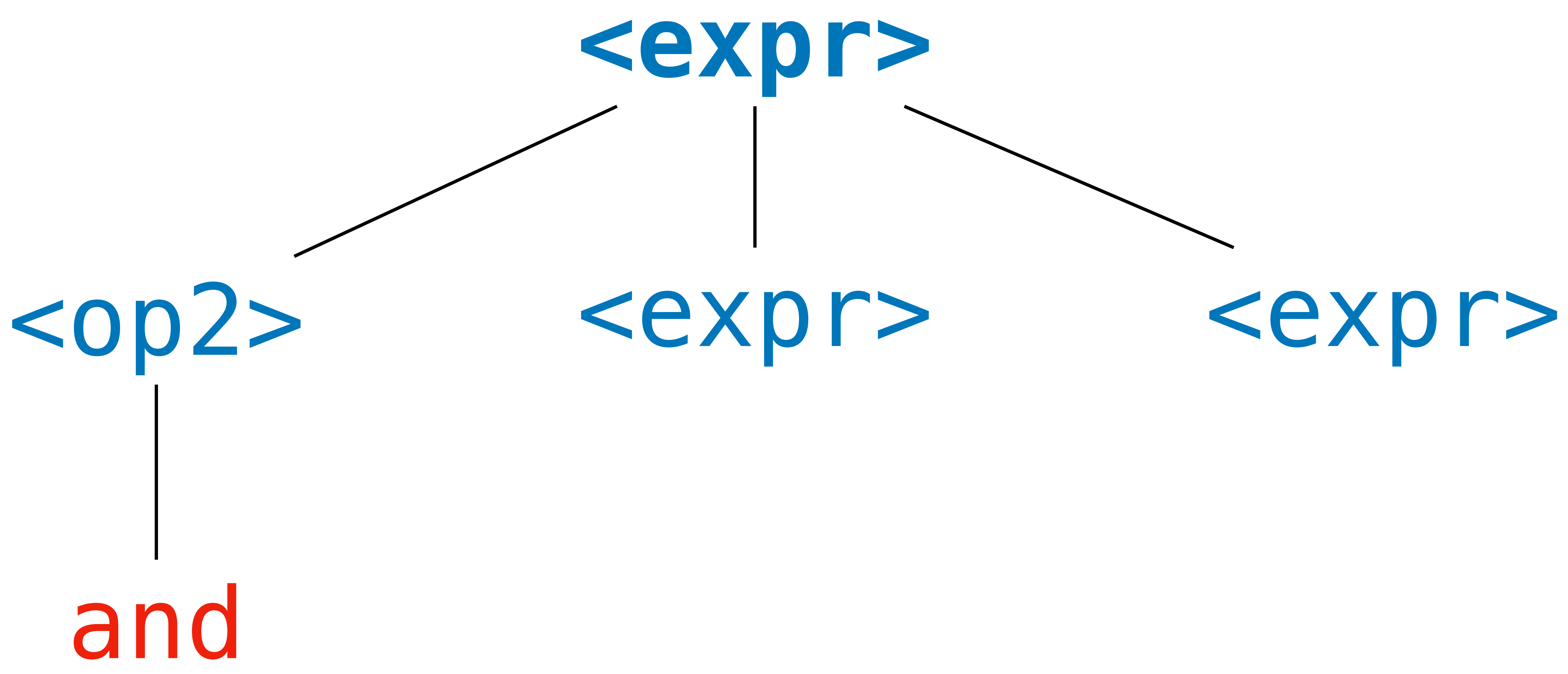
<expr>
<op2> <expr> <expr>



Derivations and Parse Trees

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

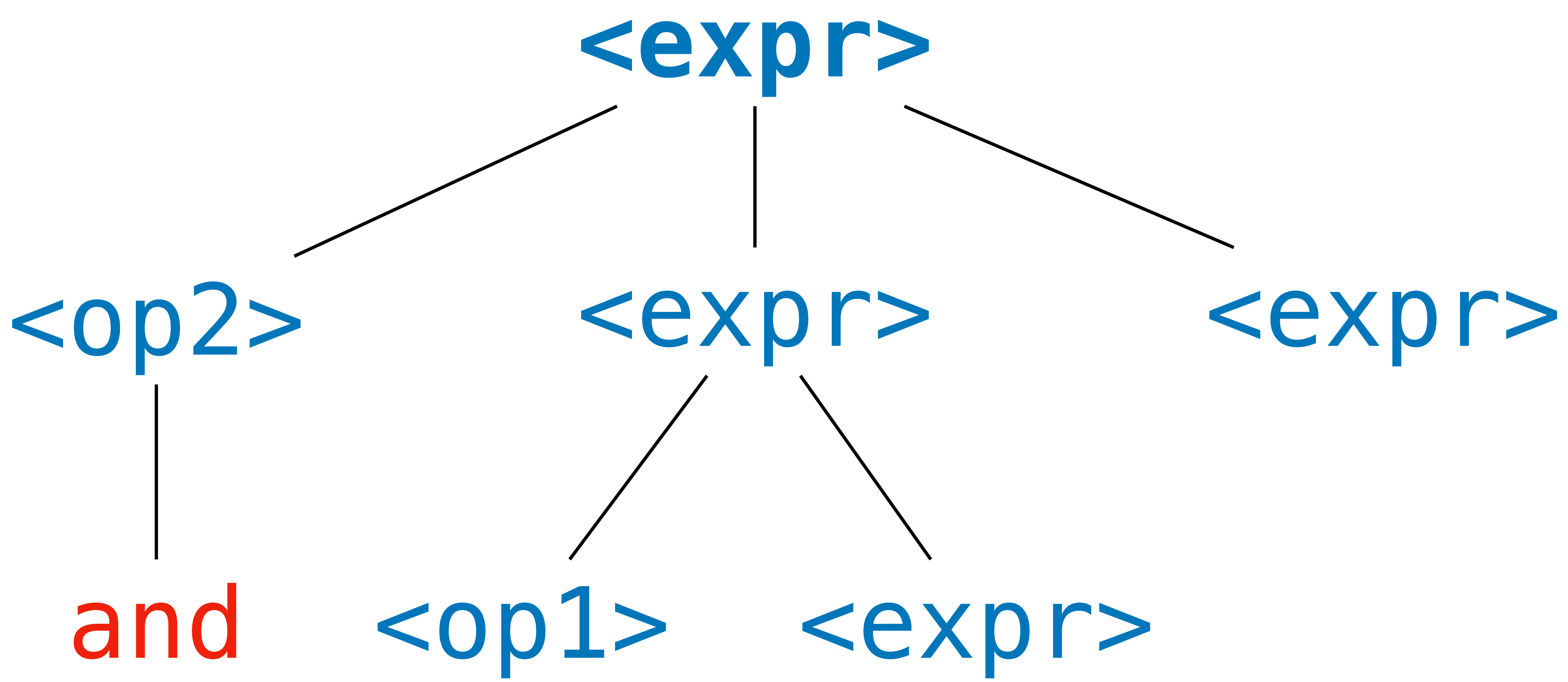
<expr>
<op2> <expr> <expr>
and <expr> <expr>



Derivations and Parse Trees

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

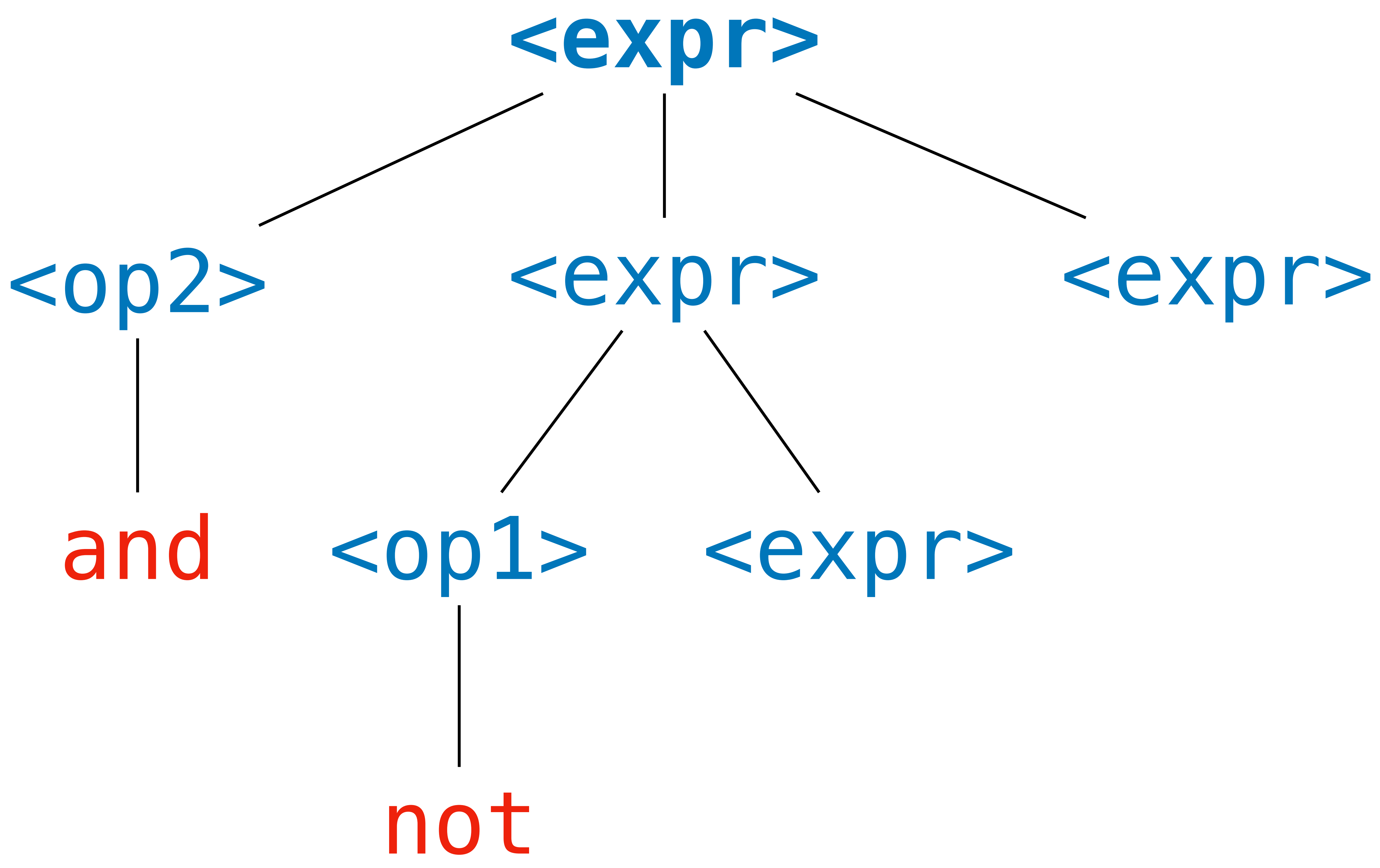
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>



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

Derivations and Parse Trees

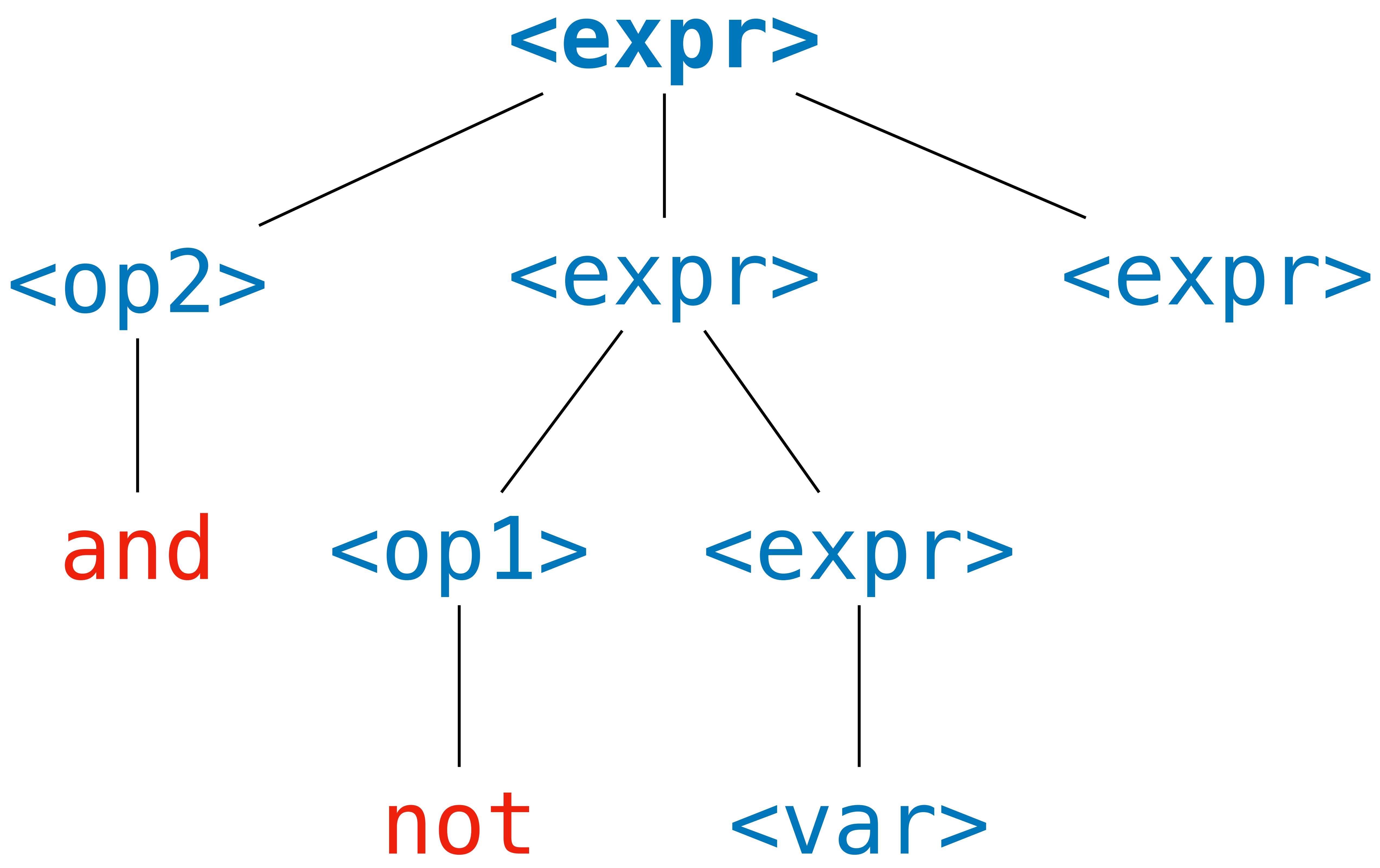
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
```



Derivations and Parse Trees

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

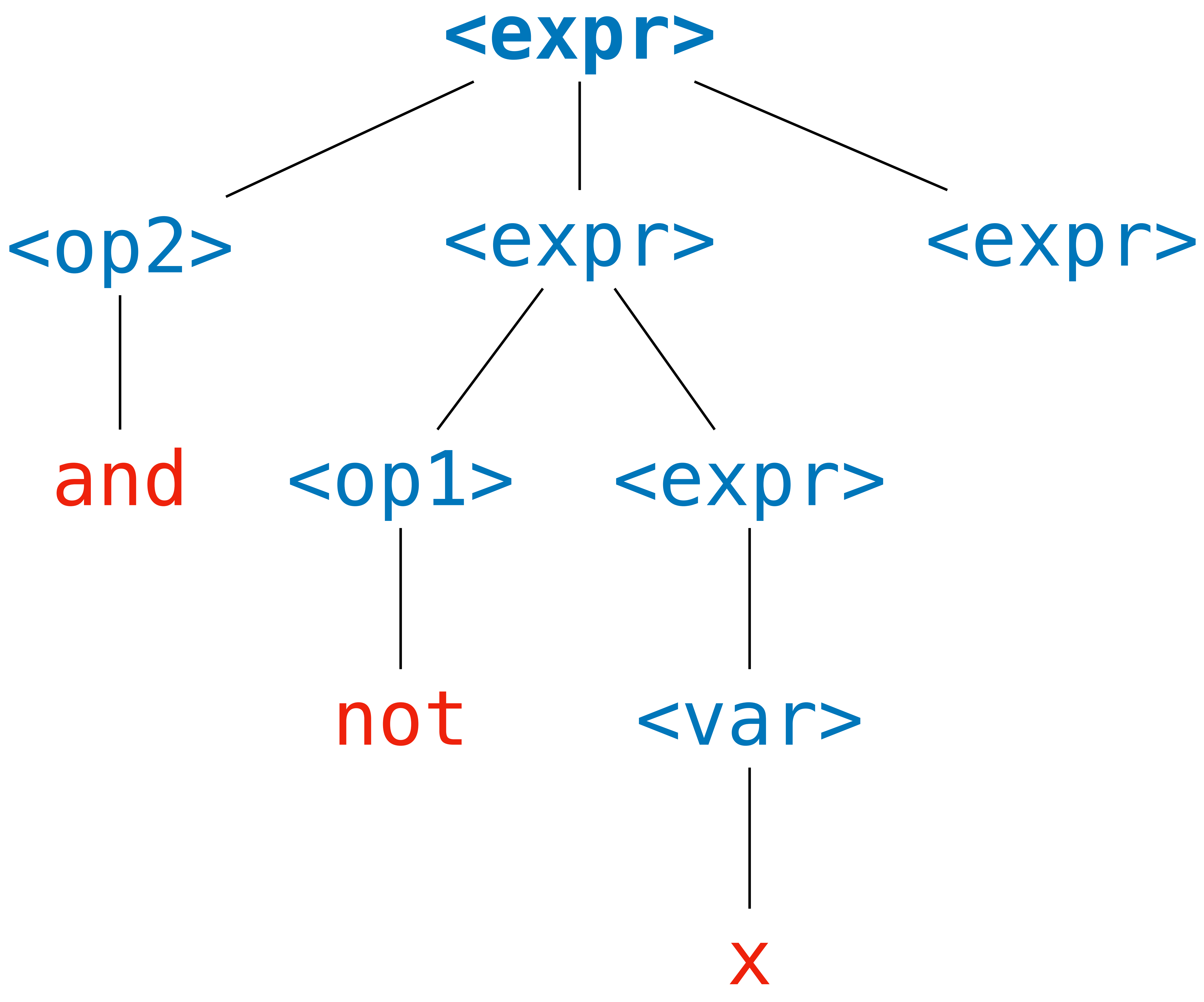
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
```



Derivations and Parse Trees

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

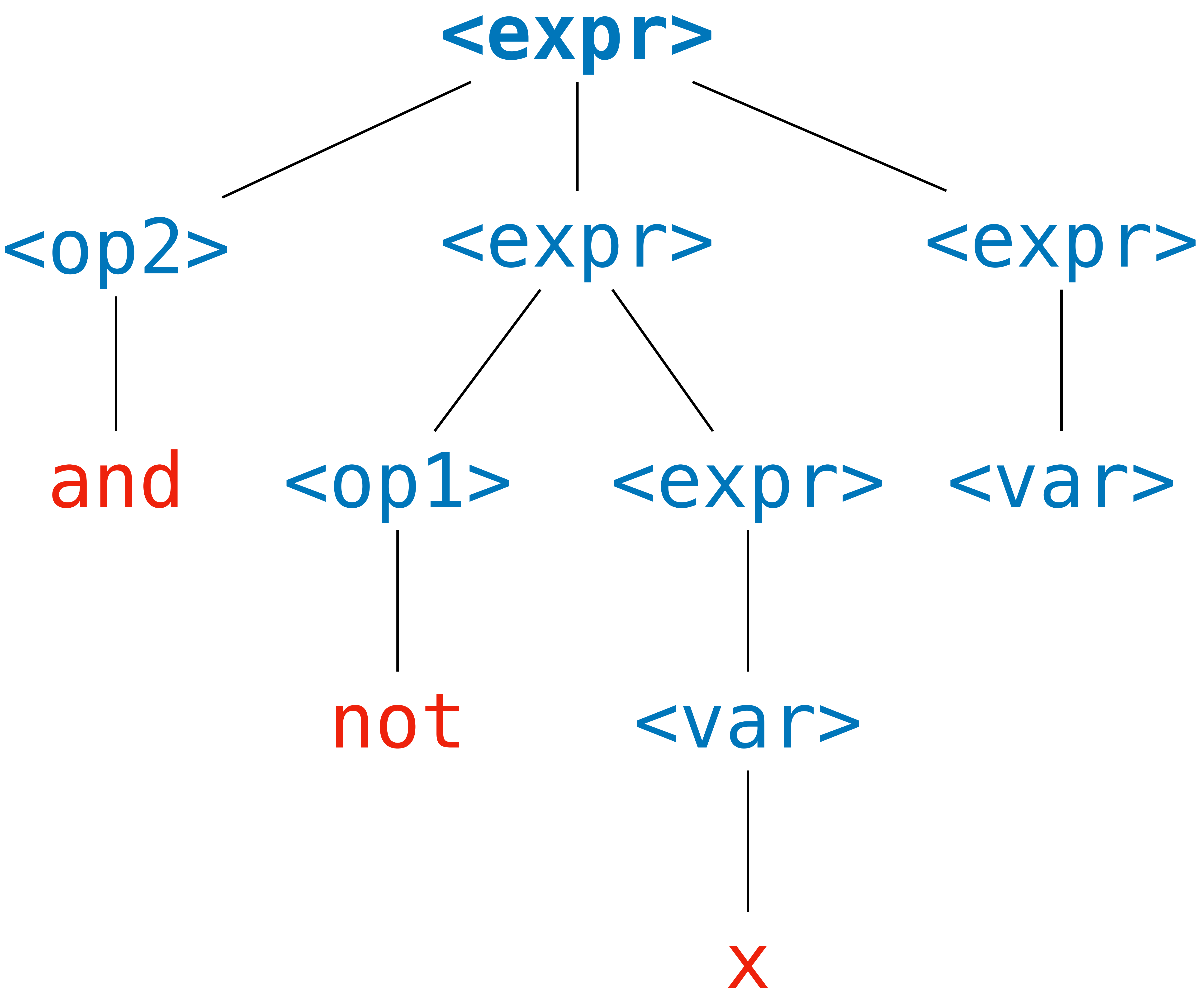
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
```



Derivations and Parse Trees

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

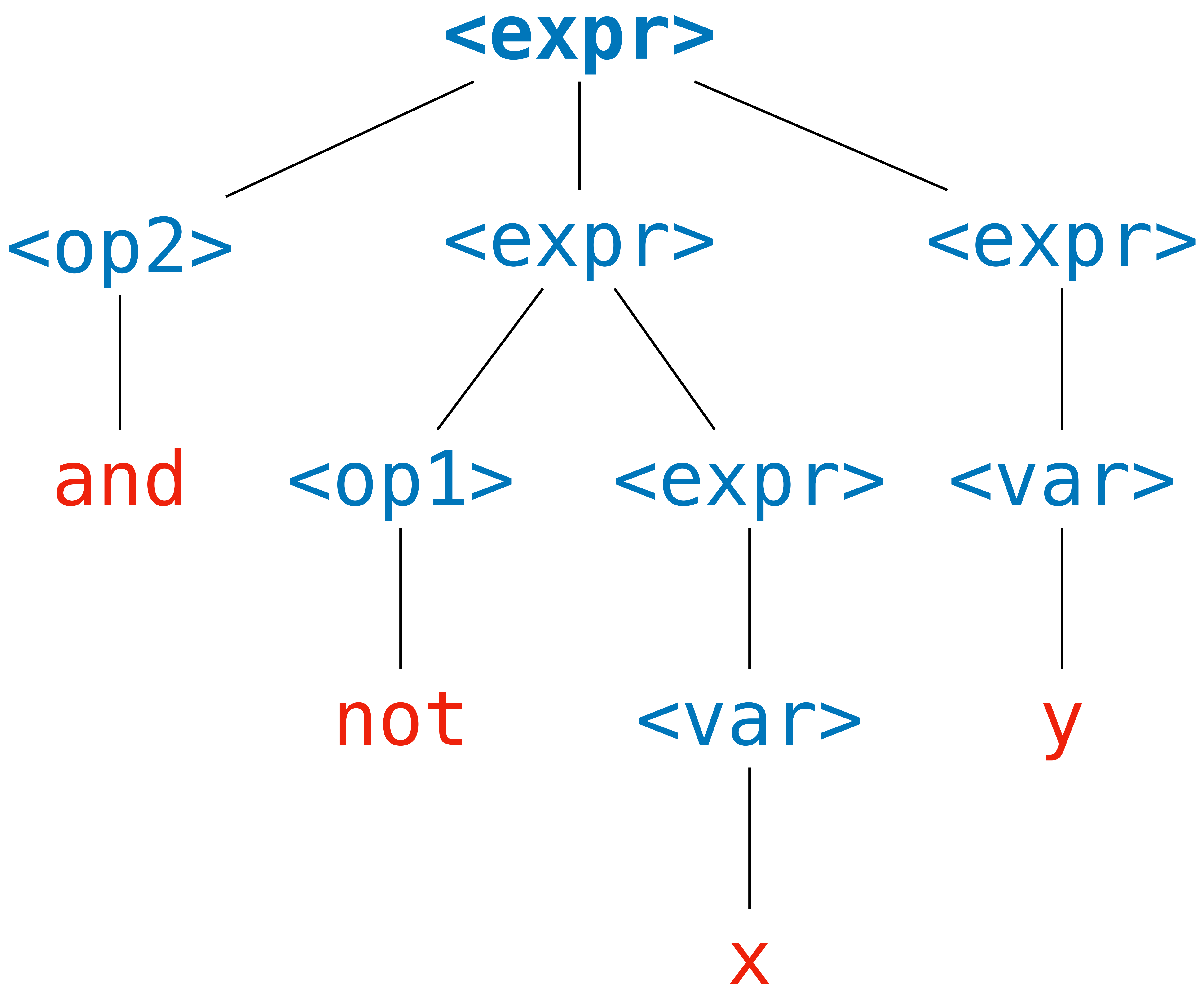
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
and not x <var>
```



Derivations and Parse Trees

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

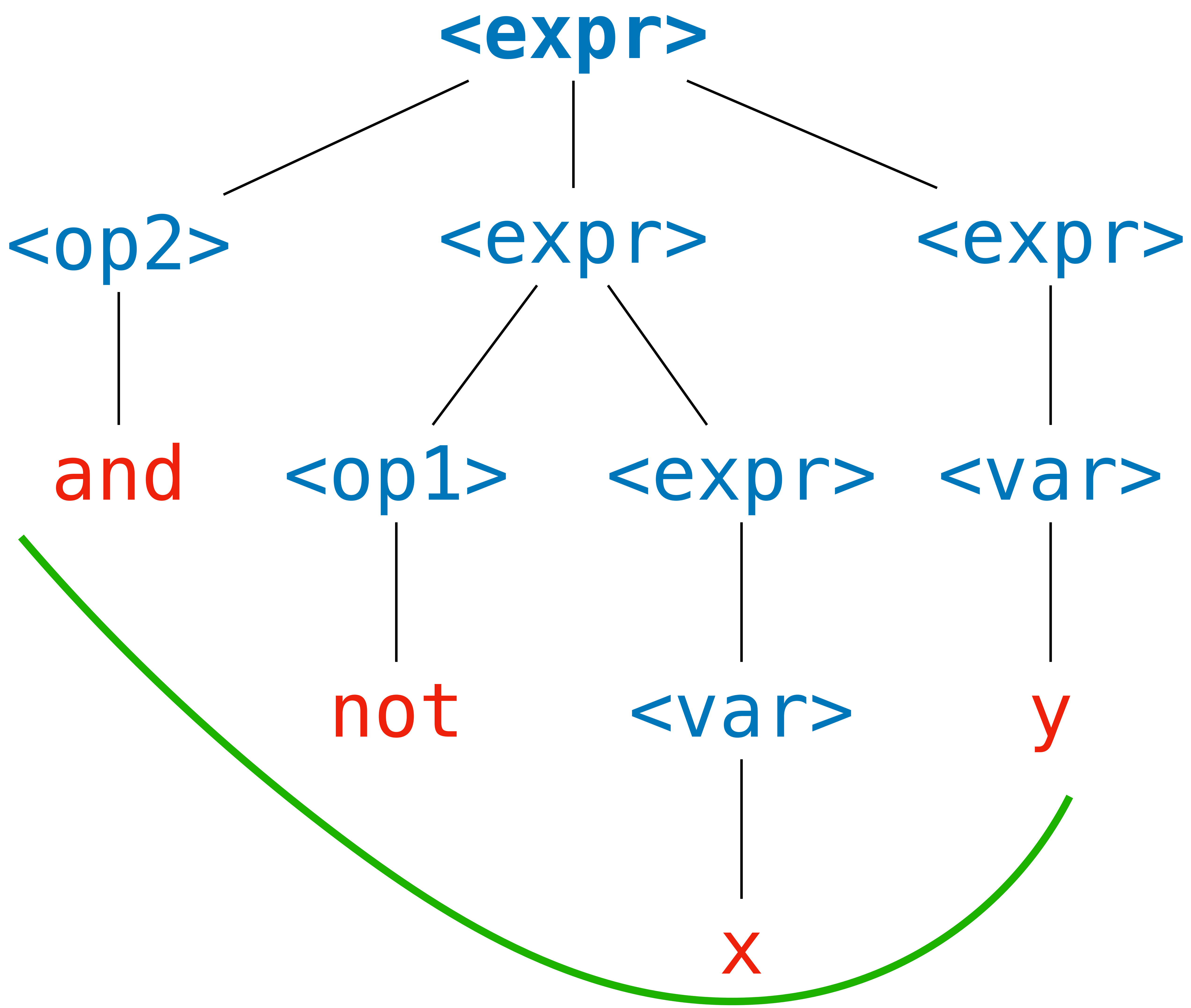
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
and not x <var>
and not x y
```



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

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

Why do we care?

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

(not x and y) or z

not x and (y or z)

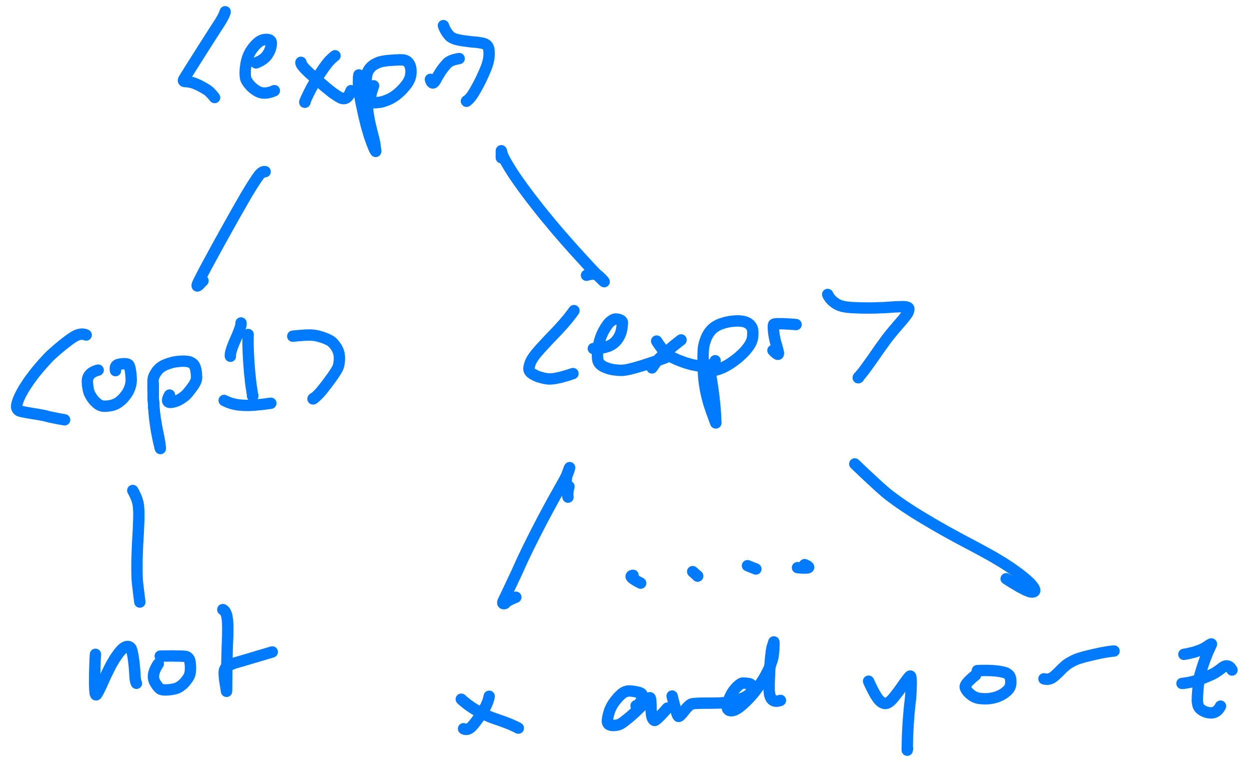
```
<expr>  ::= <op1> <expr>
          | <expr> <op2> <expr>
          | <var>
<op1>   ::= not
<op2>   ::= and | or
<var>   ::= x | y | z
```

True

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

derivation

<expr>

<expr> <op2> <expr>

<expr> and <expr>

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

<expr> and <expr> or <expr>

<op1> <expr> and <expr> or <expr>

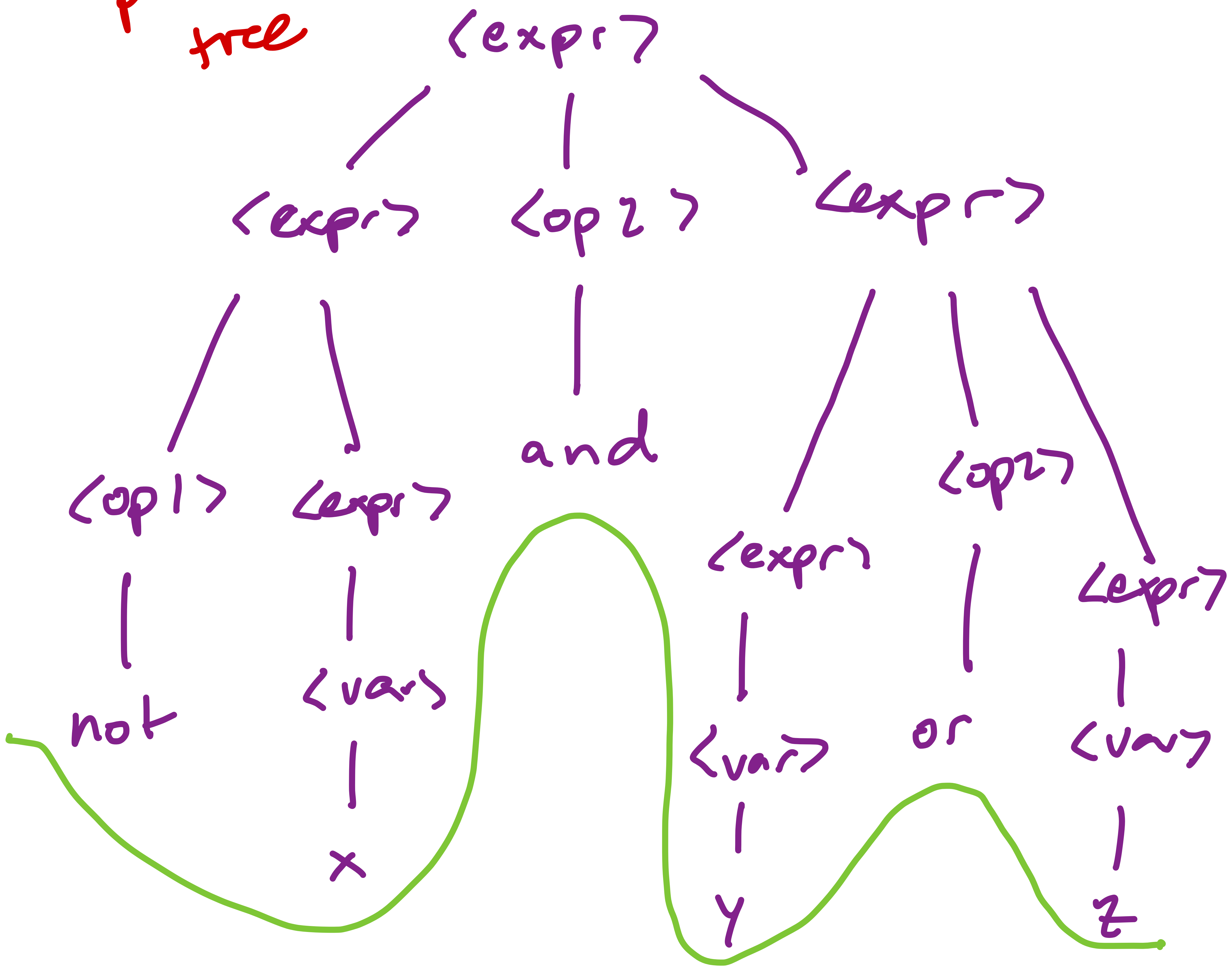
not <expr> and <expr> or <expr>

not <var> and <expr> or <expr>

not x and <expr> or <expr>

⋮ (finish)

parse tree



An Example from 320Caml

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

An Example from 320Caml

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

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

An Example from 320Caml

```
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

let x = e_1 in e_2

is a well-formed expression

Recall: If-Expressions (Syntax Rule)

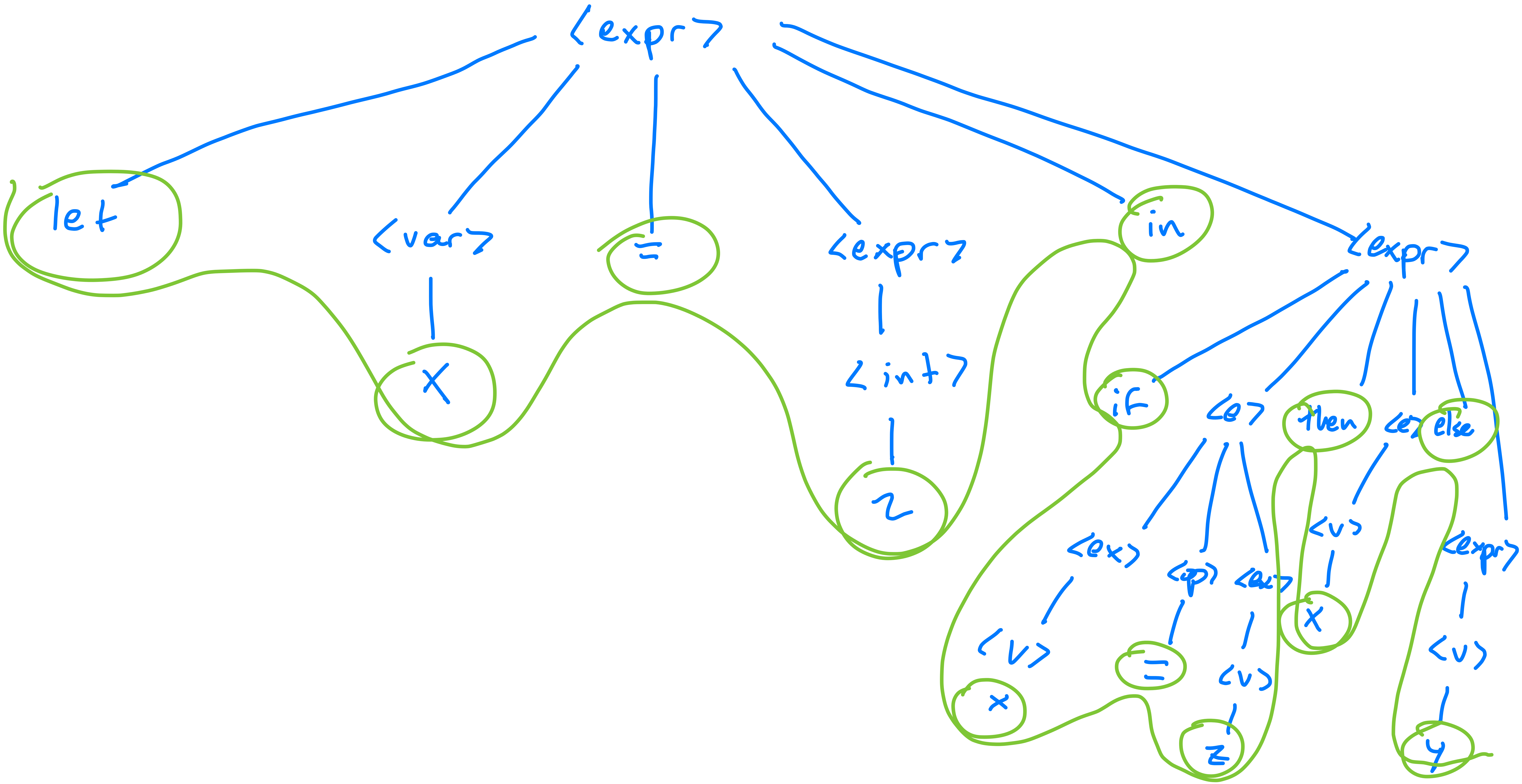
```
<expr> ::= if <expr> then <expr> else <expr>
```

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



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*