Your opinion matters to us! 🌟 Help shape the future of tech education by filling out our quick survey! 📝 Share your thoughts on your experience in the Tech Community at BU 💻

Let's make a difference together!

Your feedback could win you a **Nintendo Switch or Airpods Max!**

Don't miss out! 🎉

Administrivia

Assignment 5 is due on Friday by 11:59PM.

Assignment 6 will be assigned on Friday and will be due after the break.

Midterm exam grades will be released on Wednesday.

Formal Grammar II: Ambiguity and Precedence

Principles of Programming Languages
Lecture 12

CAS CS 320

Objectives

Discuss ambiguity in grammar.

Look at ways of avoiding ambiguity.

Analyze the relationship between operator fixity, precedence, and ambiguity.

Keywords

BNF grammar

production rules

derivations

parse trees

ambiguity

Polish notation

associativity

fixity

precedence

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?)*

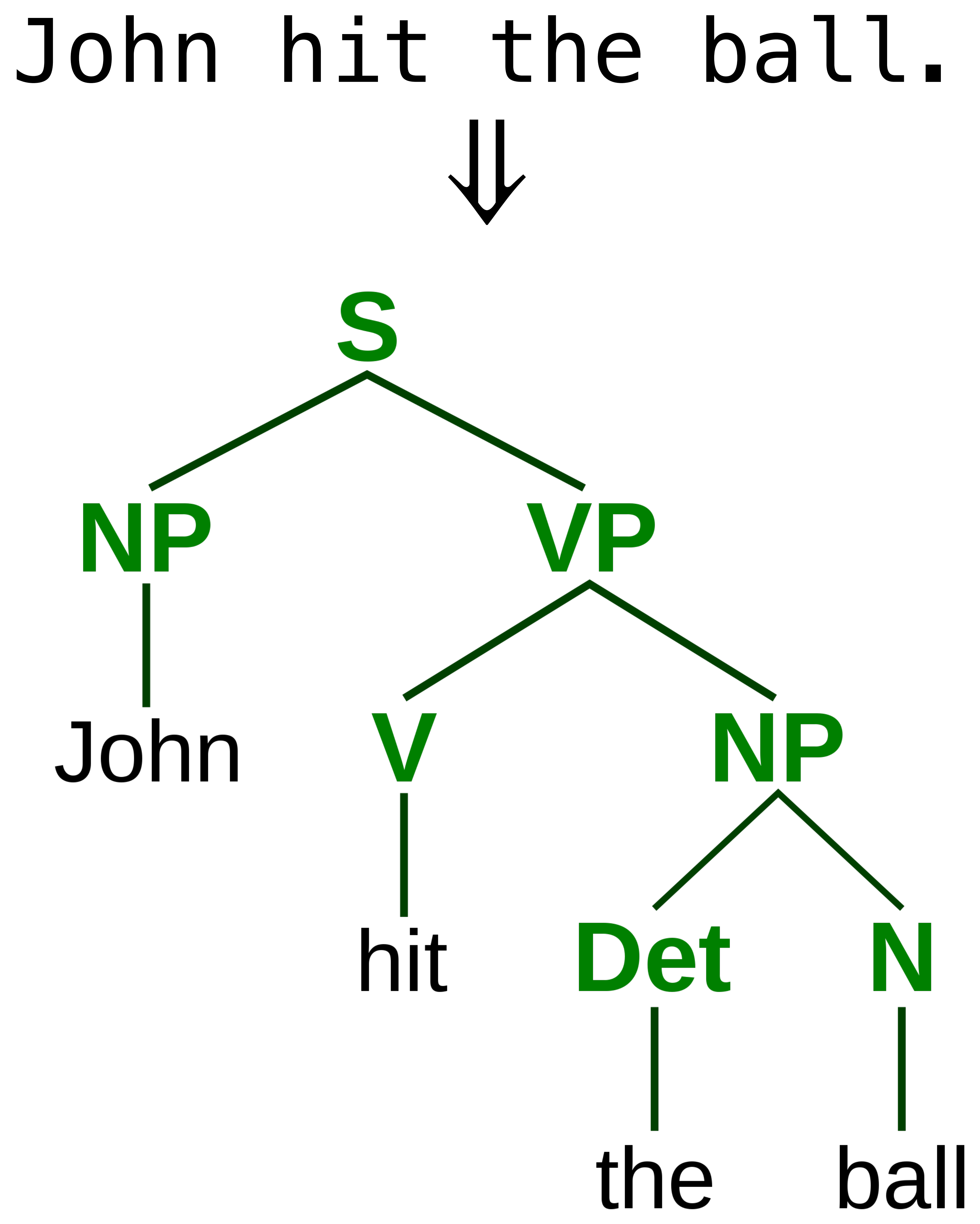
Answer

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

not x and y or z

Recap

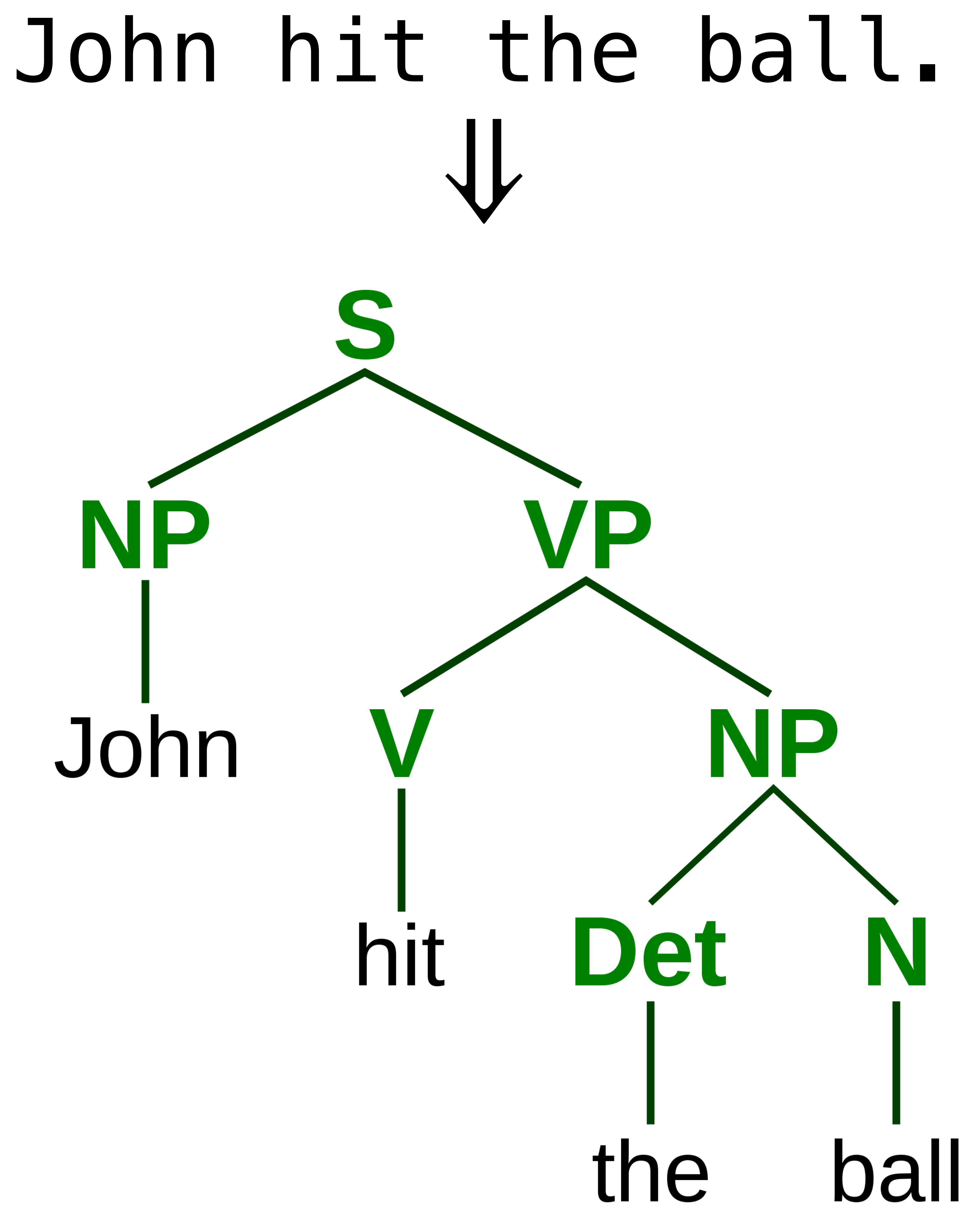
What is Grammar?



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

What is Grammar?

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

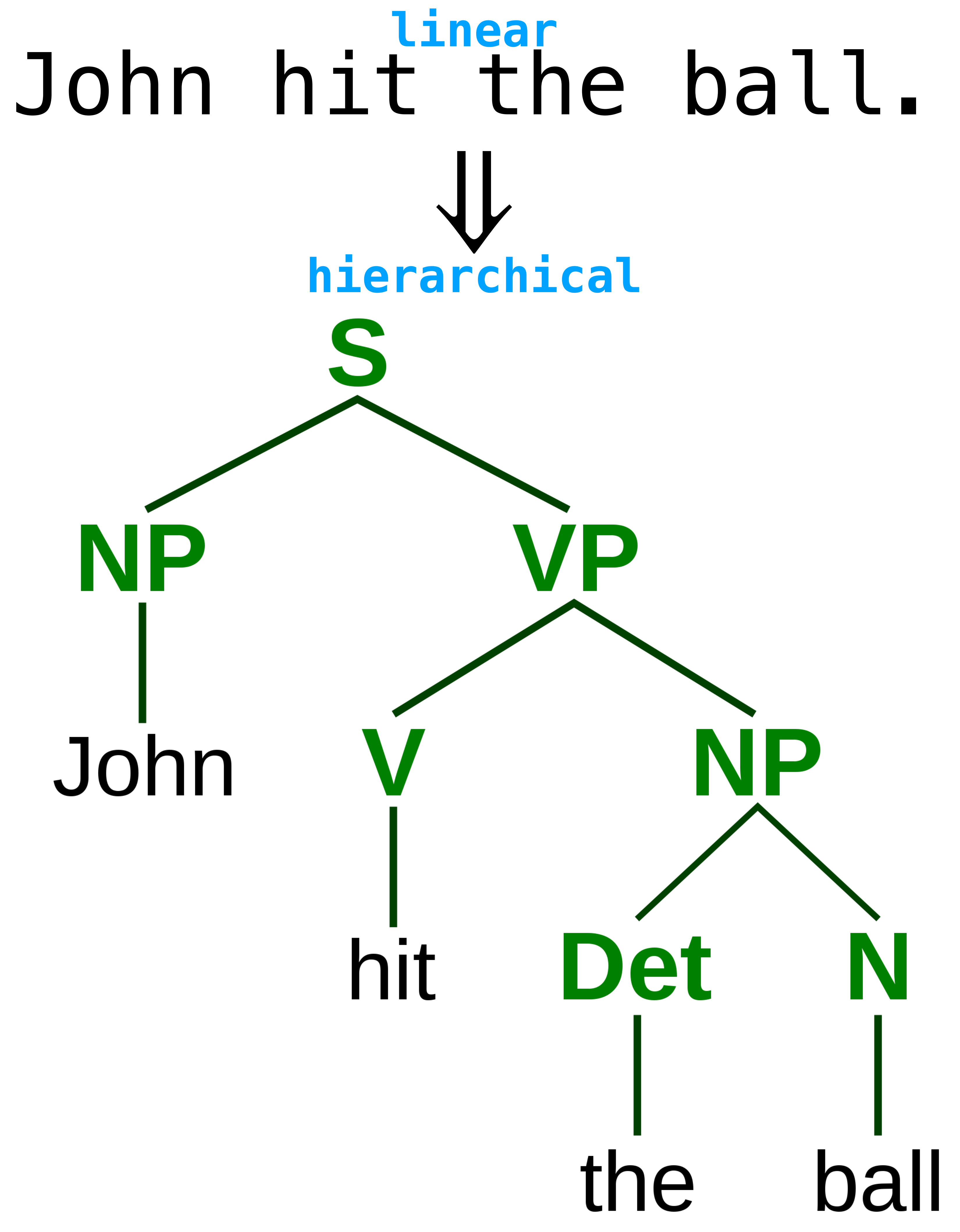


https://commons.wikimedia.org/wiki/File:ParseTree.svg

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

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.

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

```
# 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.

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

<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".
```

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
```

Recall: BNF Grammars

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

Recall: BNF Grammars

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

tokens (terminal symbols)

Recall: BNF Grammars

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

abstractions (non-terminal symbols)

tokens (terminal symbols)

Recall: BNF Grammars

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)

Production Rules

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

Production Rules

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

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

Production Rules

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

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

A production rule is describes what we can replace a non-terminal symbol with in a derivation.

Production Rules

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

A <u>sentential form</u> is a sequence of terminal or nonterminal symbols.

A <u>production rule</u> is 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 "|".

Derivations and Parse Trees

Derivations and Parse Trees

A grammar G is determined by a collection of production rules and a designated **starting non-terminal symbol.**

Derivations and Parse Trees

A grammar G is determined by a collection of production rules and a designated **starting non-terminal symbol.**

A derivation is a sequence of sentential forms (beginning at the designated 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

A grammar G is determined by a collection of production rules and a designated **starting non-terminal symbol.**

A derivation is a sequence of sentential forms (beginning at the designated 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

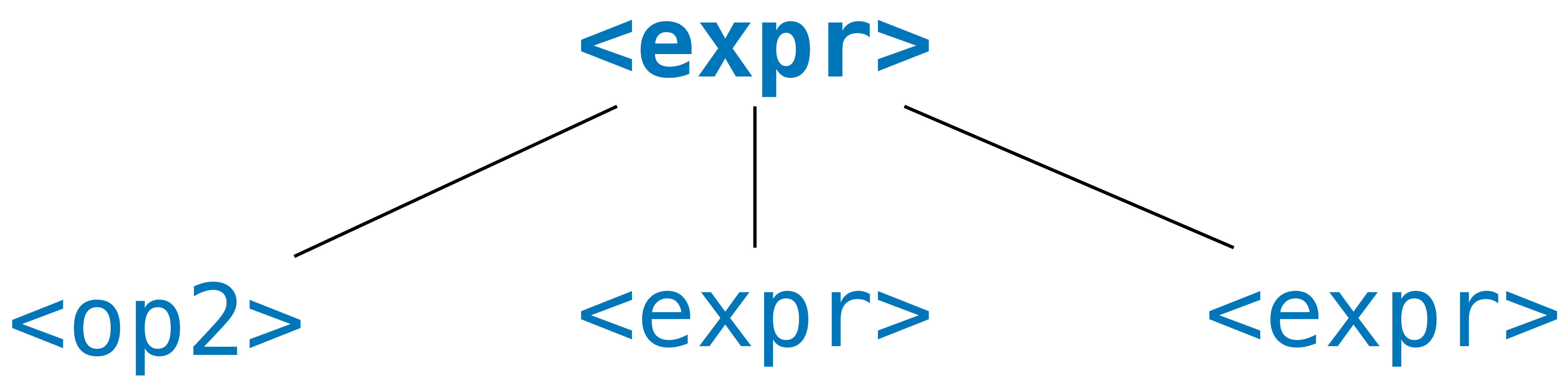
Derivations and Parse Trees

<expr>

<expr>

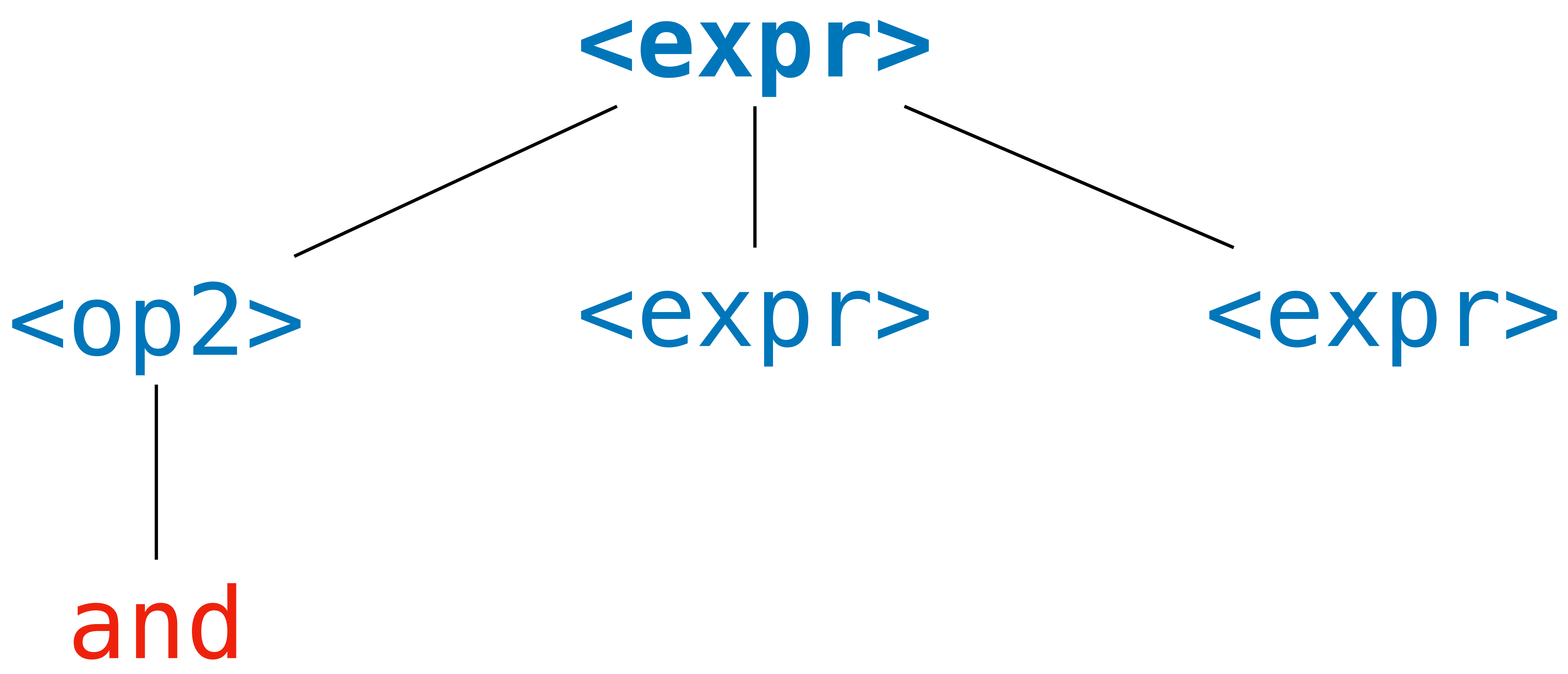
Derivations and Parse Trees

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



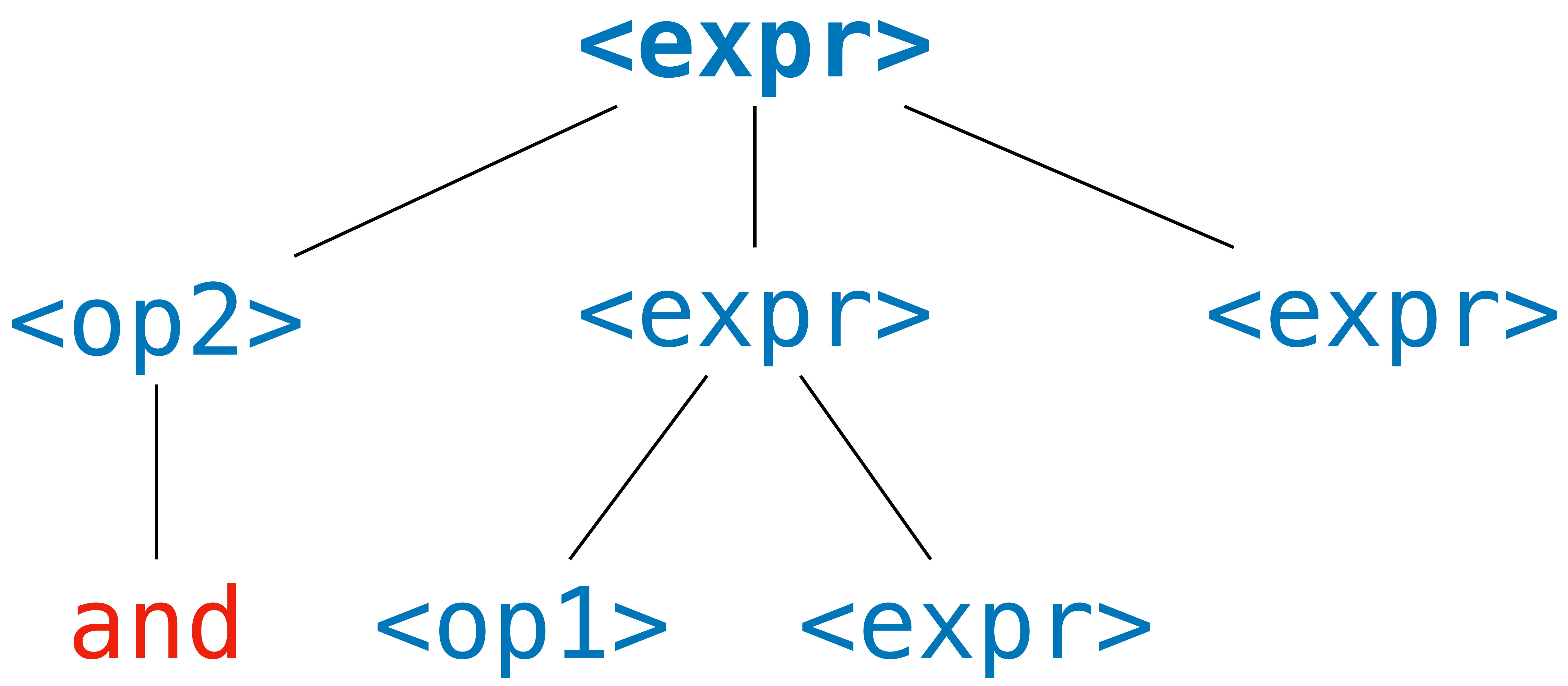
Derivations and Parse Trees

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



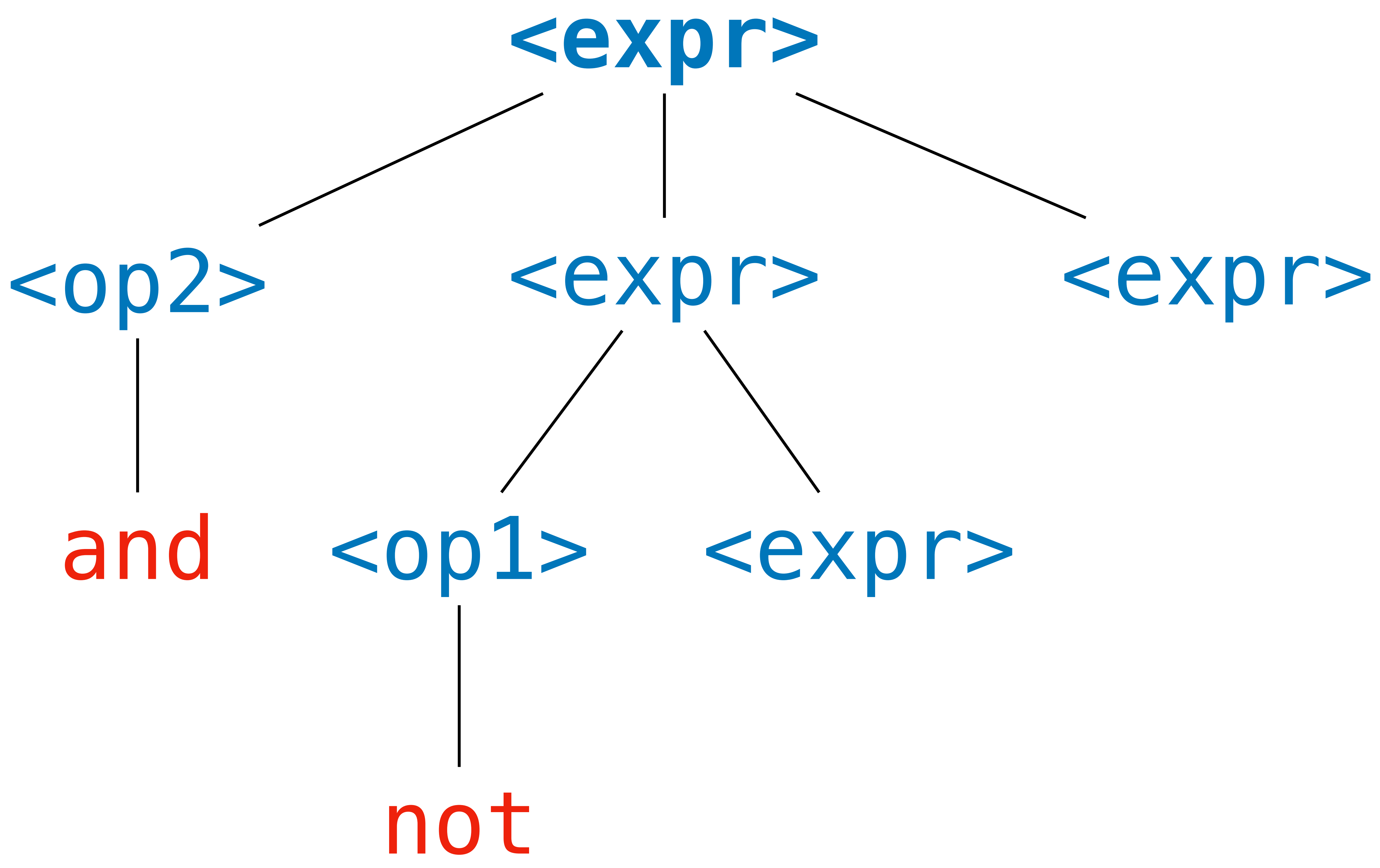
Derivations and Parse Trees

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



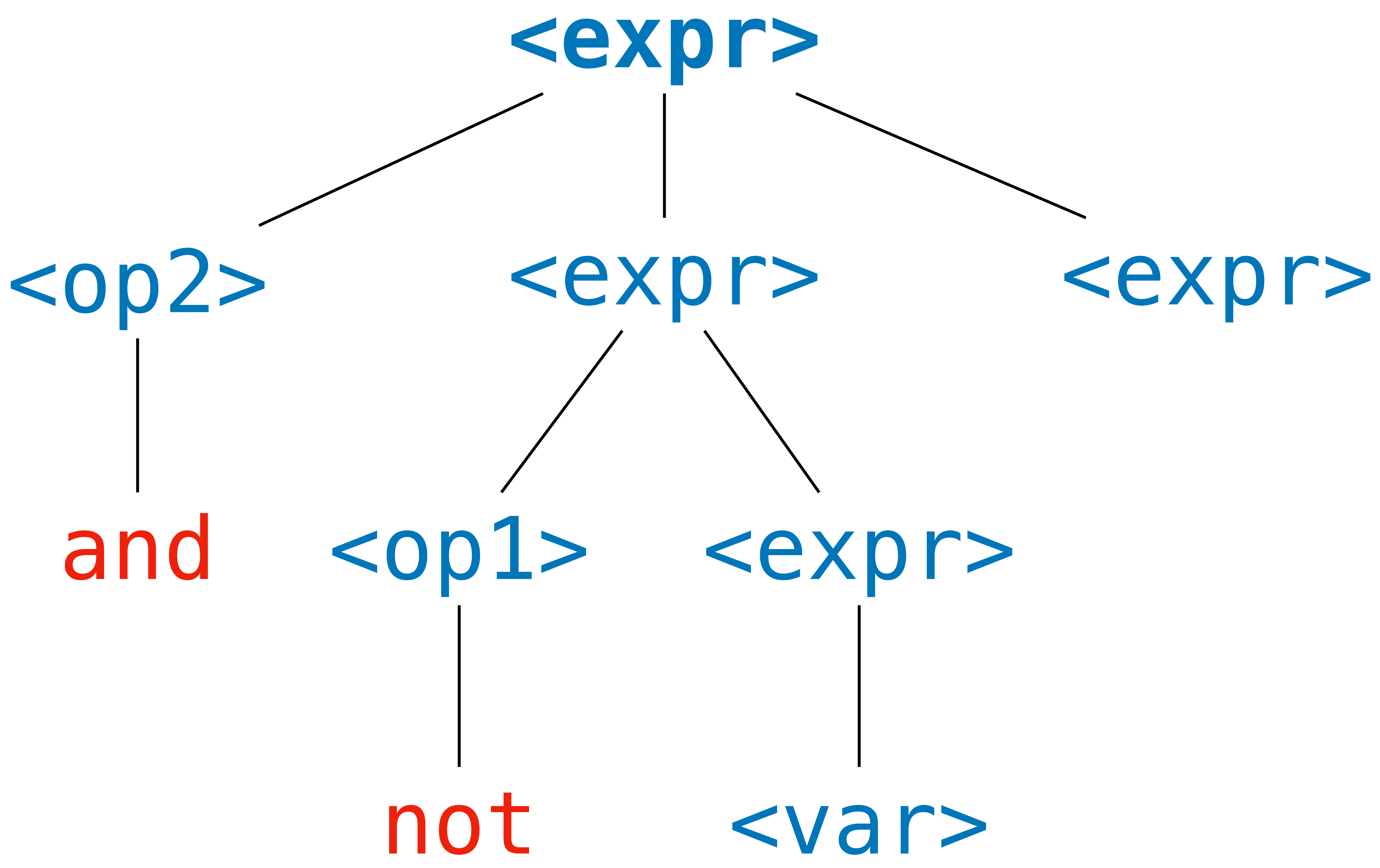
Derivations and Parse Trees

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



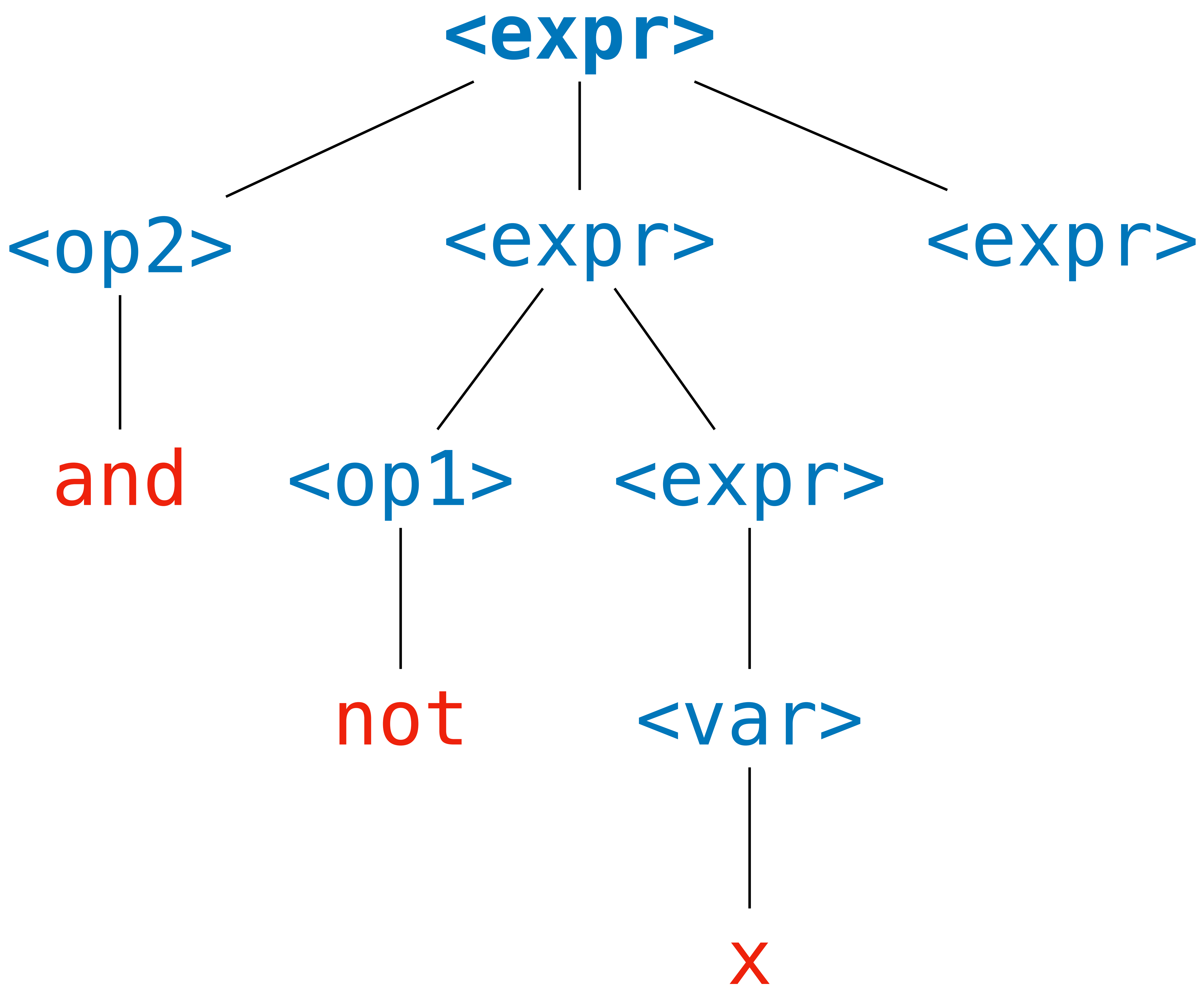
Derivations and Parse Trees

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



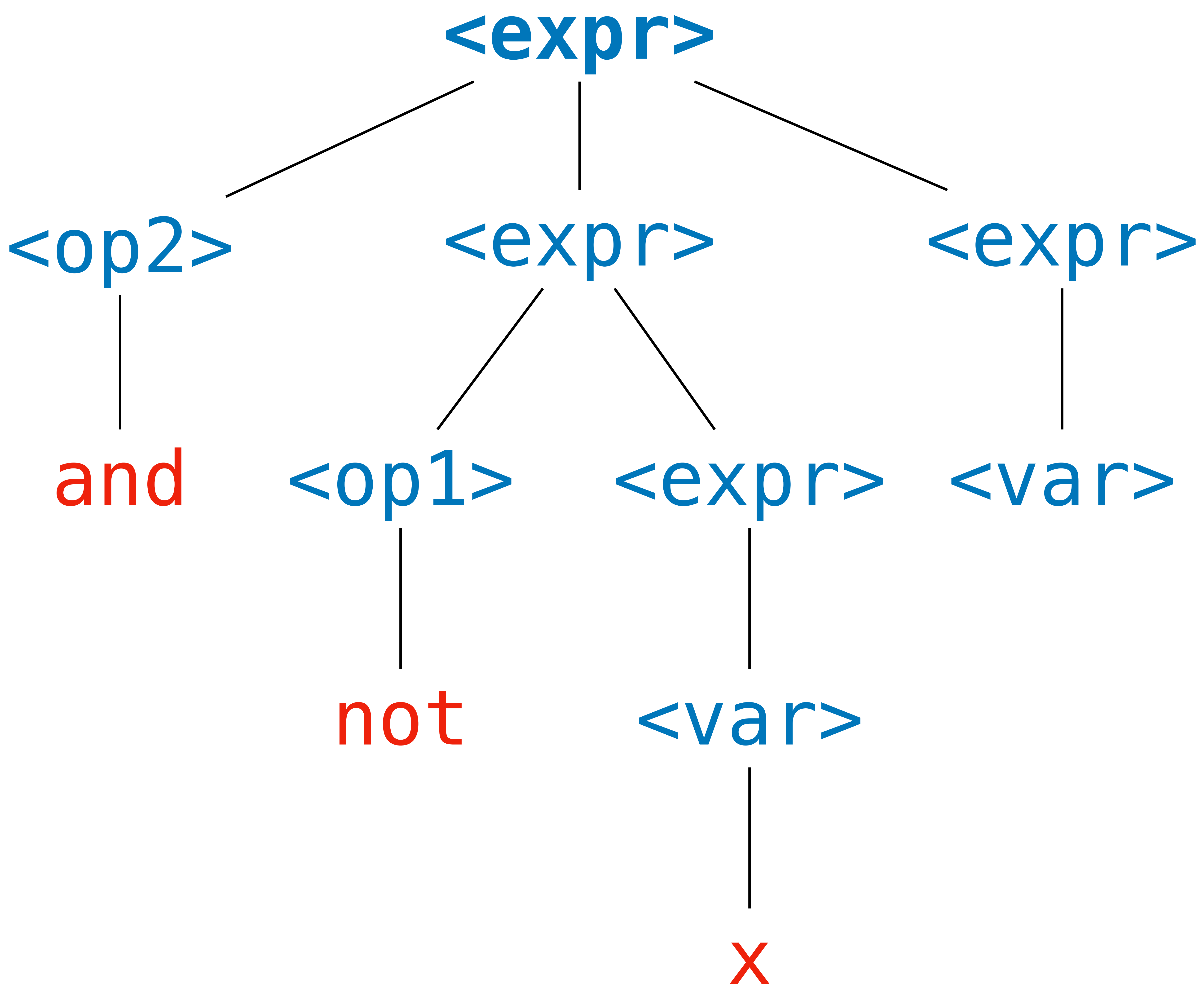
Derivations and Parse Trees

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



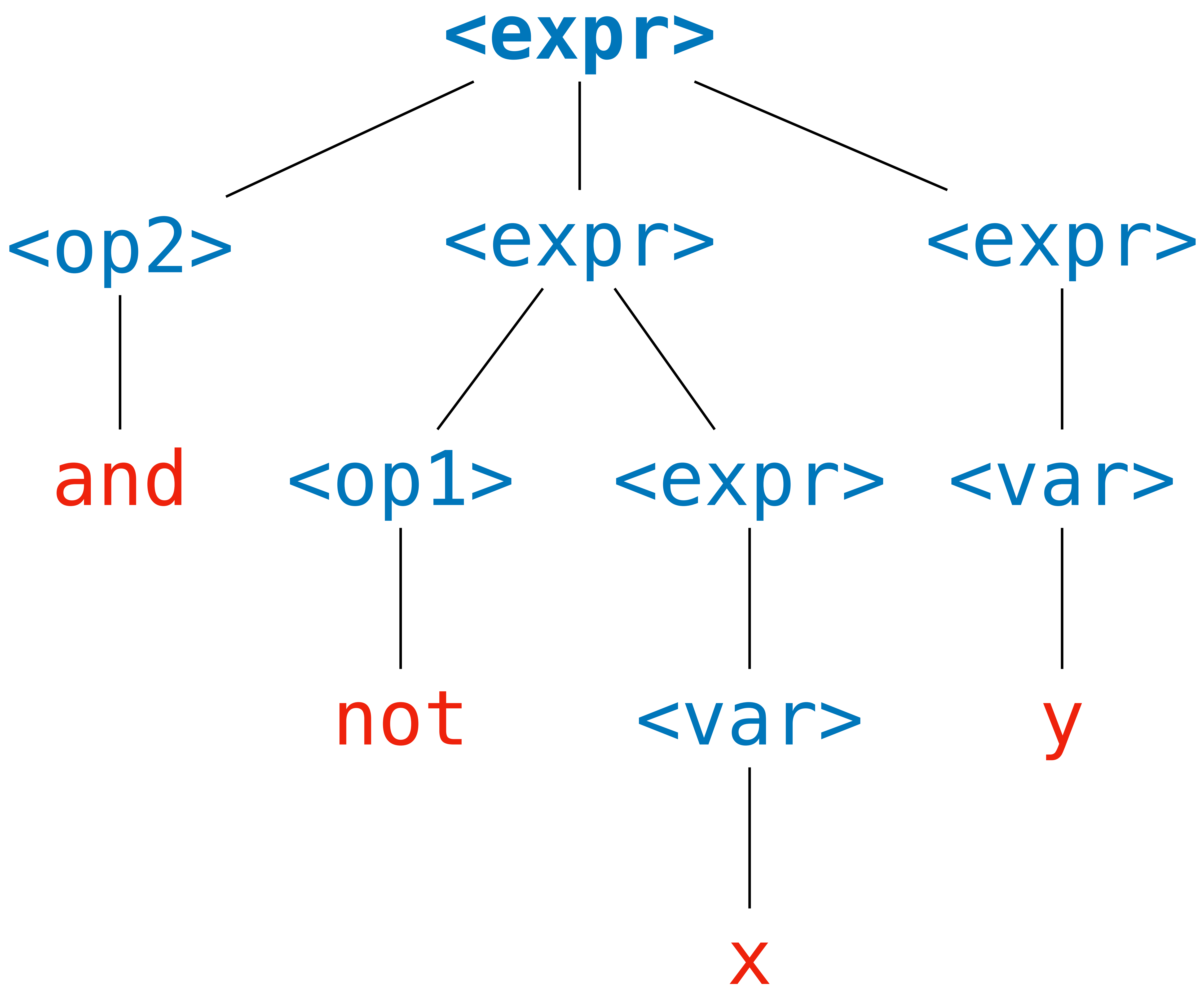
Derivations and Parse Trees

```
<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>
```



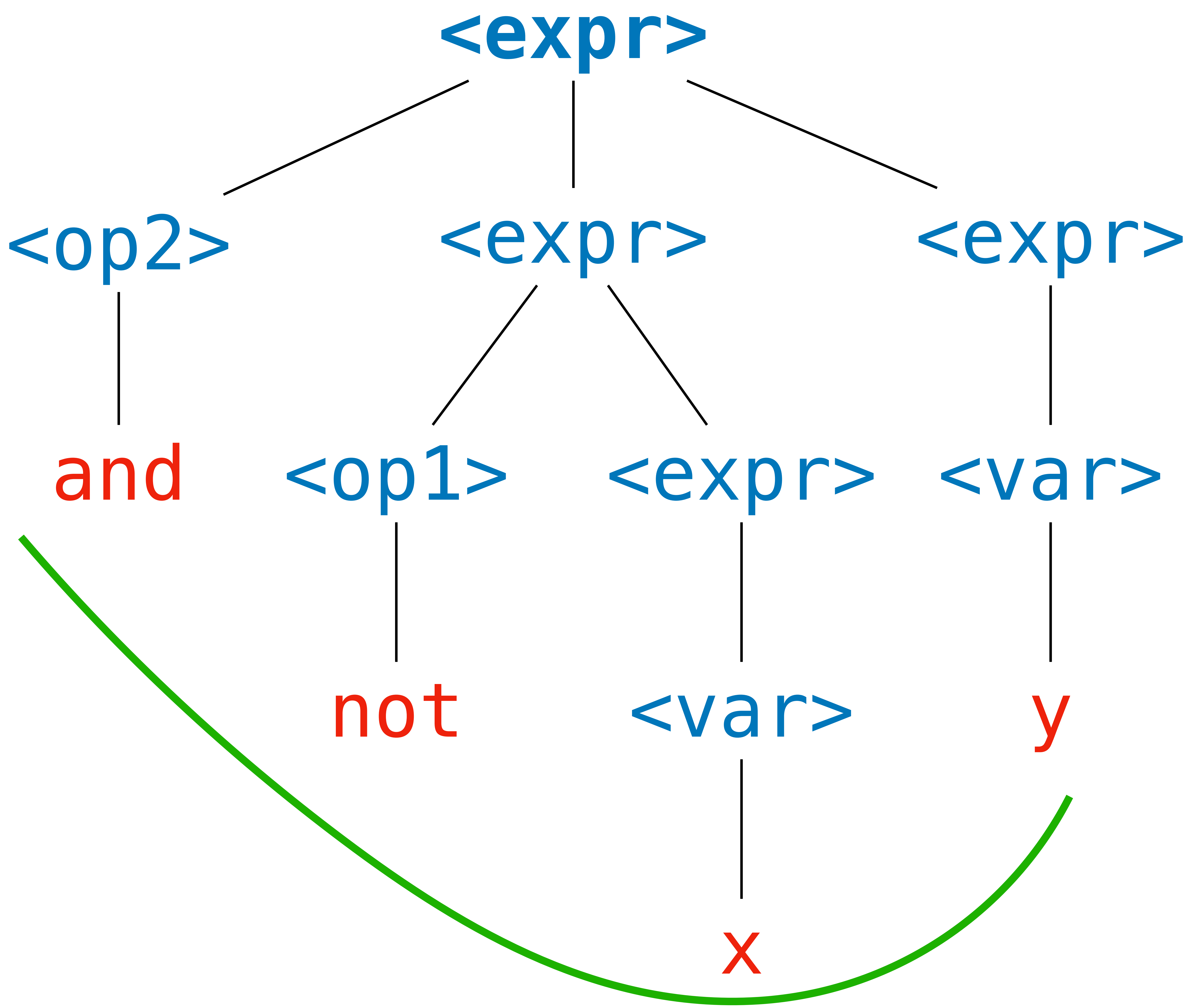
Derivations and Parse Trees

```
<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

```
<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?

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

Why do we care?

Why do we care?

Why do we care?

char stream $\xrightarrow{\text{scanner}}$ 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

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

Why do we care?

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

a.k.a. list of terminal symbols (token stream)

a.k.a. derivation in a grammar (parse tree)

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

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

BNF Grammars and ADTs

```
type op1 = NOT
type op2 = AND | OR
type var = X | Y | Z
type expr
  = RV of var
  | R1 of op1 * expr
  | R2 of op2 * expr * expr
```

```
(* ((NOT X) AND (Y OR Z)) *)
let ex : expr =
  R2 (AND,
      R1 (NOT, RV X),
      R2 (OR, RV Y, RV Z))
```

Parse Trees in a BNF grammar are easily represented as ADTs.

Functional languages are well-suited for building programming languages.

Understanding Check

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

Write the OCaml type for the parse trees of the above BNF Grammar.

Answer

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

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 **roasted** duck is ready for dinner.*

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

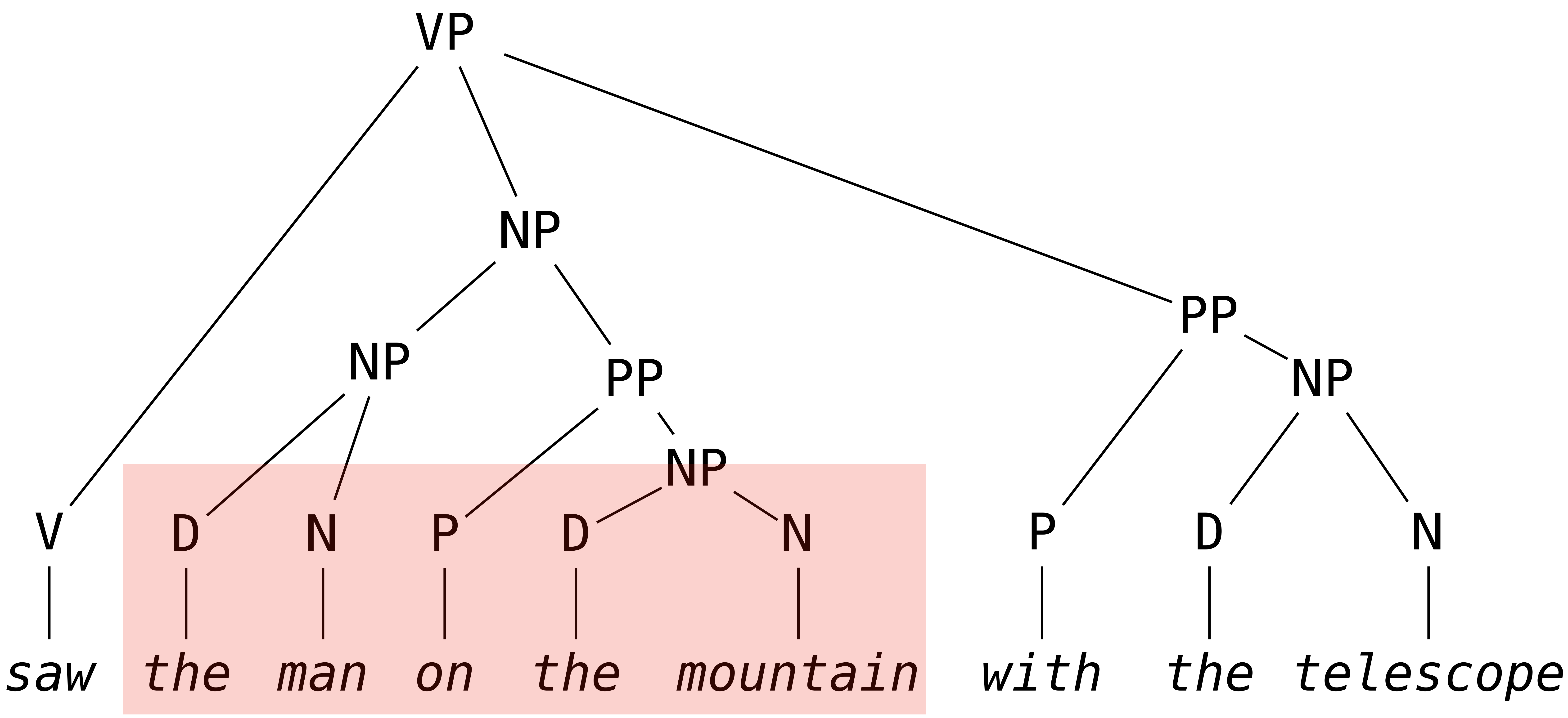
*He said the exam would **be held** on Tuesday.*

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

We have informal tactics for avoiding these pitfalls.

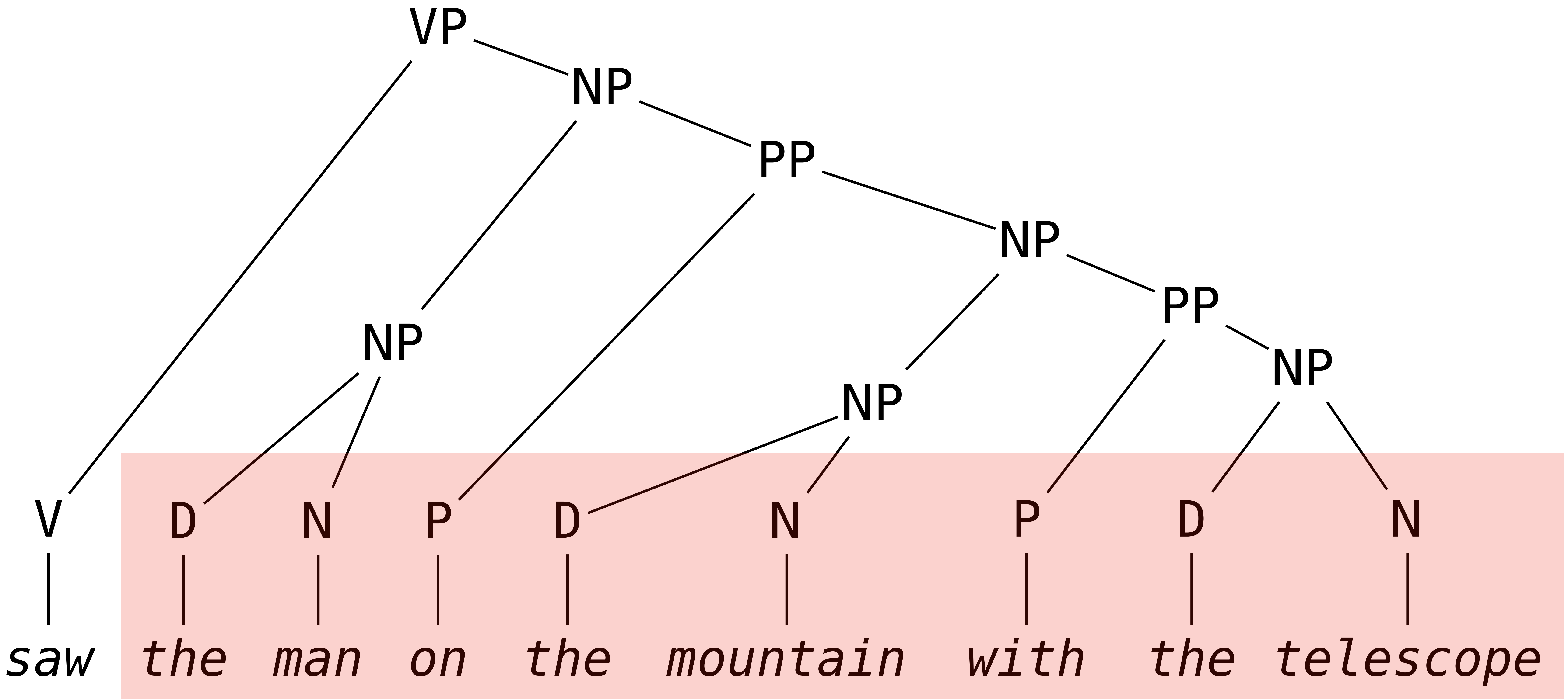
(examples taken from around the internet)

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.

Ambiguity in Formal Grammar

Ambiguity in Formal Grammar

Definition. A BNF grammar is **ambiguous** if there is a sentence with multiple derivations.

Ambiguity in Formal Grammar

Definition. A BNF grammar is **ambiguous** if there is a sentence with multiple 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 derivations.

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

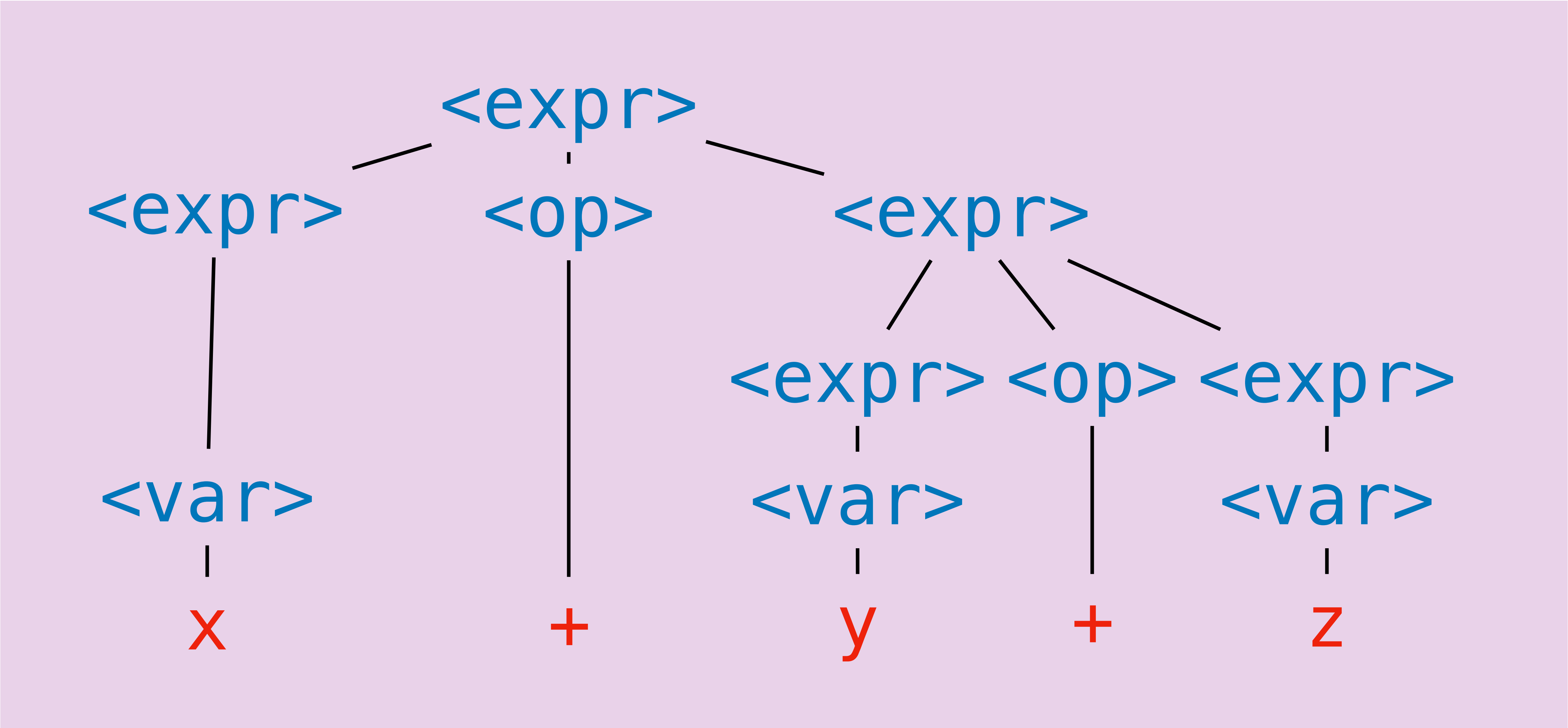
x + y + z can be derived as

Ambiguity in Formal Grammar

Definition. A BNF grammar is **ambiguous** if there is a sentence with multiple derivations.

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

x + y + z can be derived as

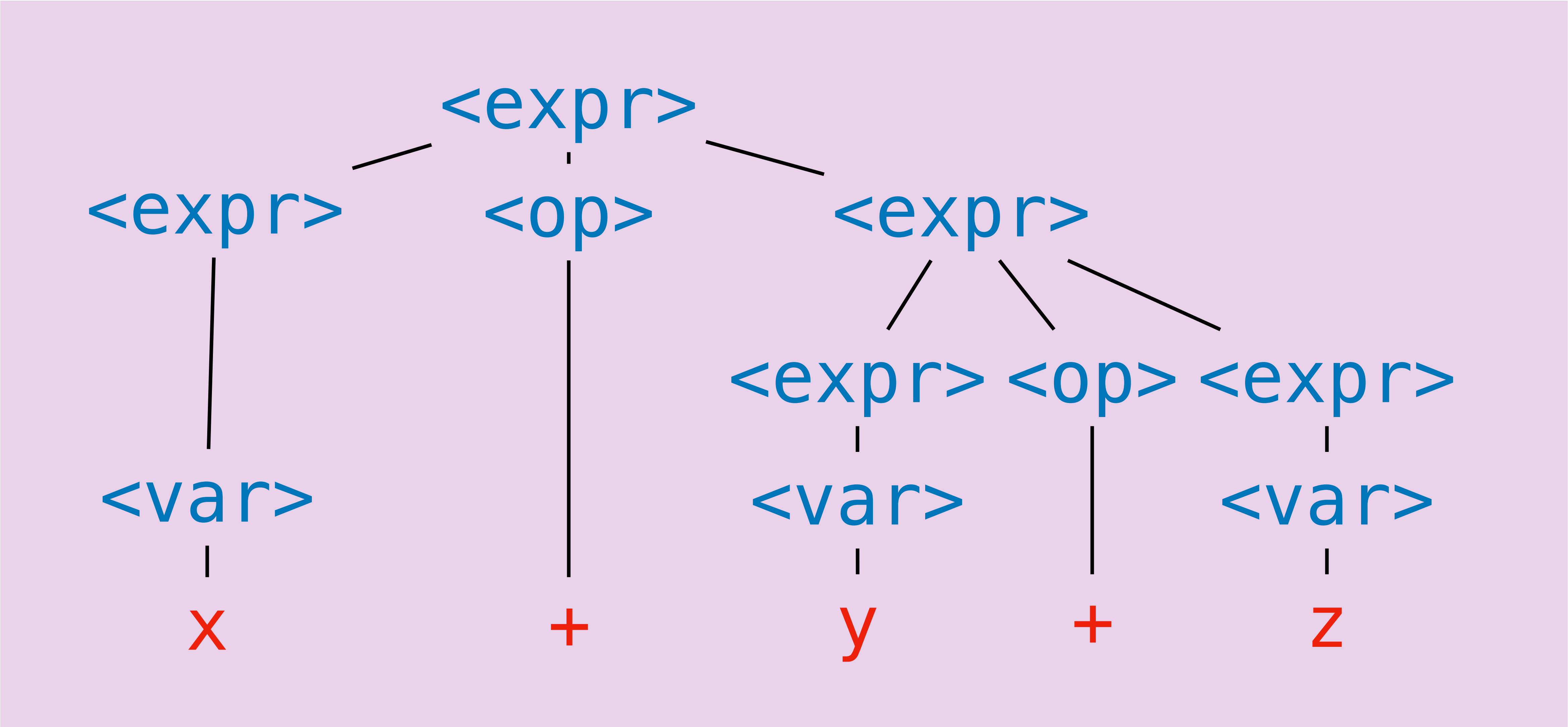


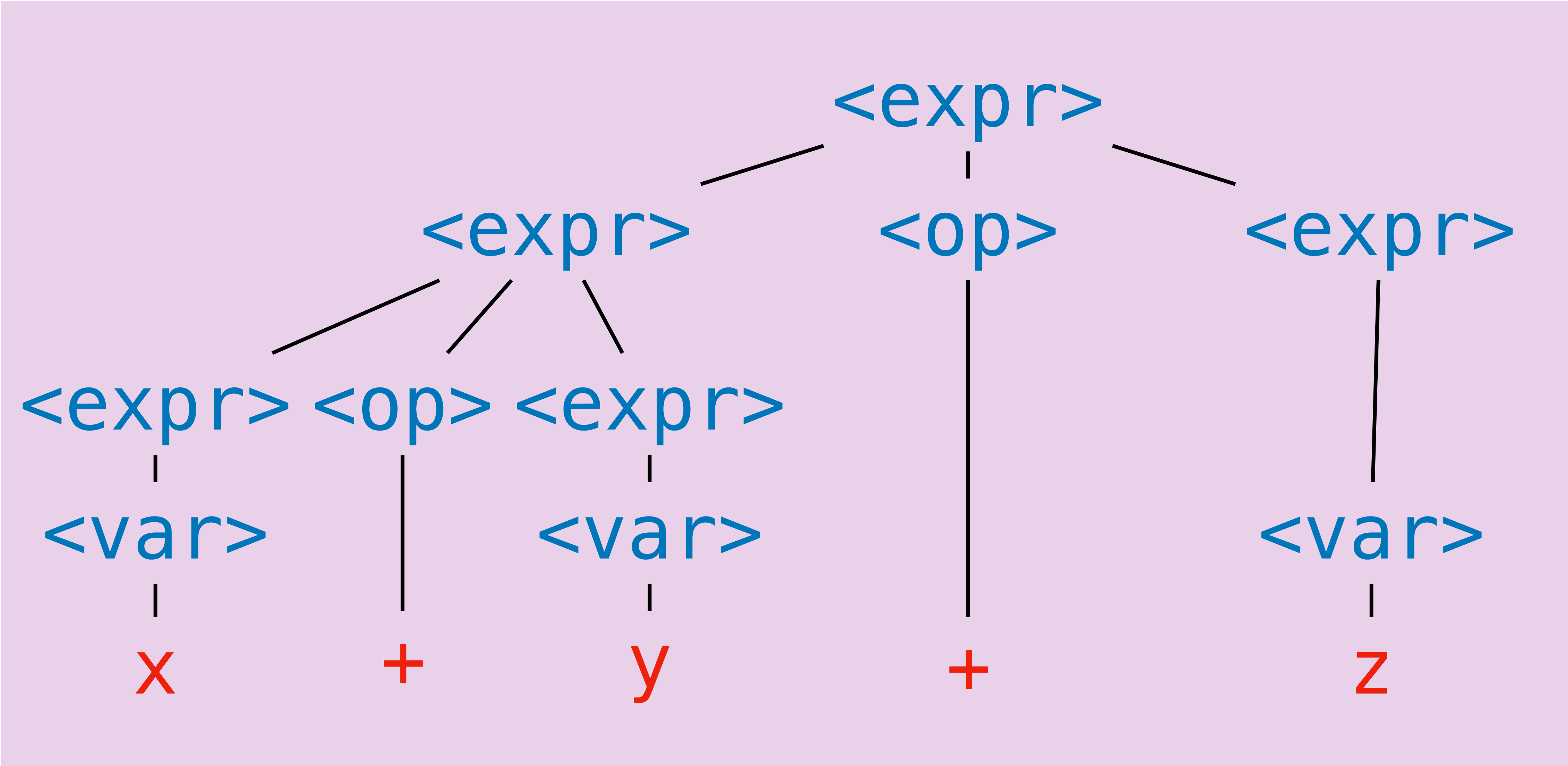
Ambiguity in Formal Grammar

Definition. A BNF grammar is **ambiguous** if there is a sentence with multiple 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.

Understanding Check

```
<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
```

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! (get from ref)

Fixity

The <u>fixity</u> 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)

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

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

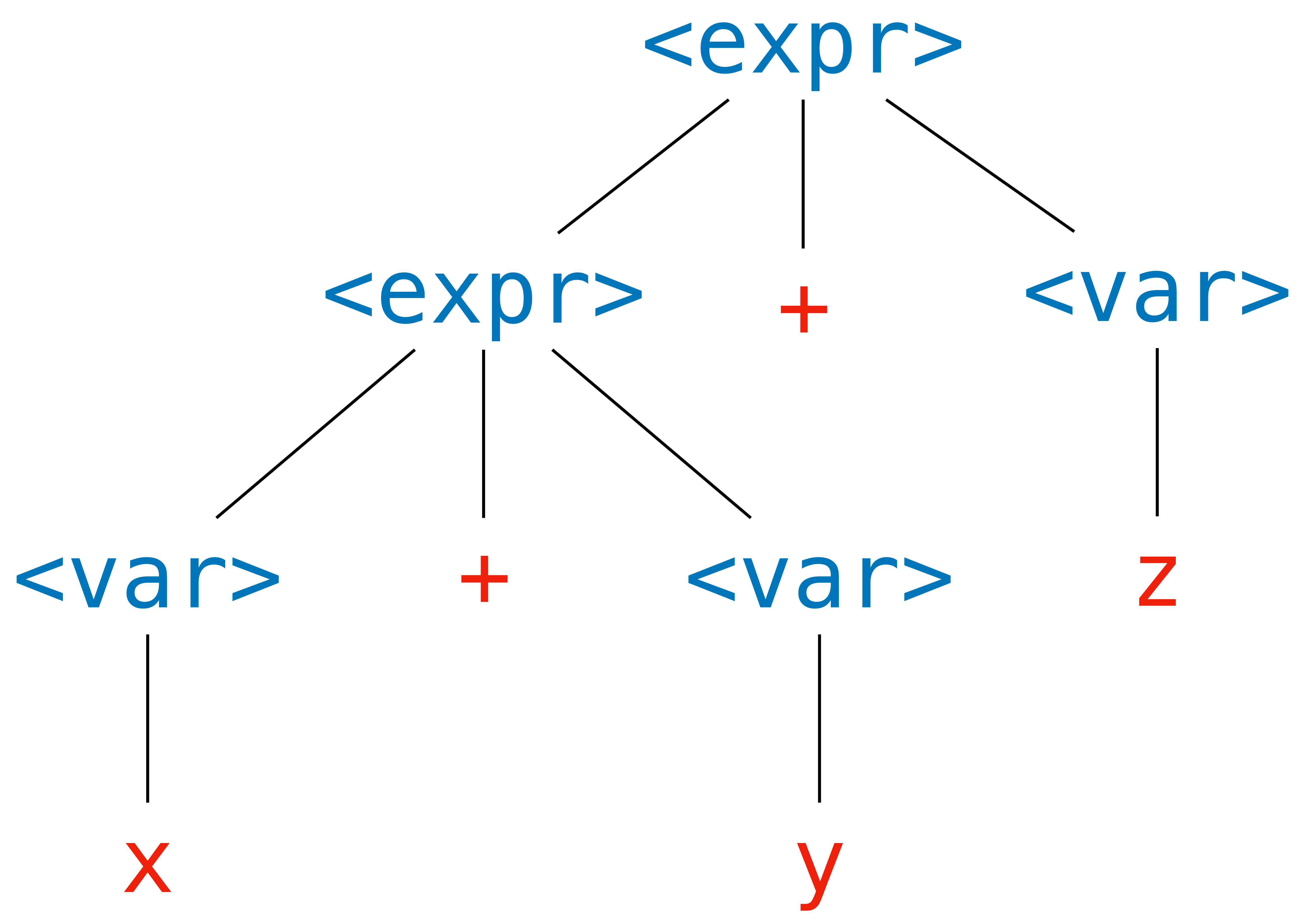
```
left associative       1 +  2 +  3 ⇒ (1 +  2) +  3

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

Associativity

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

x + y + z $\Rightarrow$



"add the sum of x and y to z"

Question. *Can we enforce that this expression goes to this parse tree?*

Breaking Symmetry

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

Breaking Symmetry

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

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

Breaking Symmetry

```
<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>
```

Breaking Symmetry

```
<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?

Dealing with Left Associativity

```
<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
```

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

Question. *What about the right associative?*

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

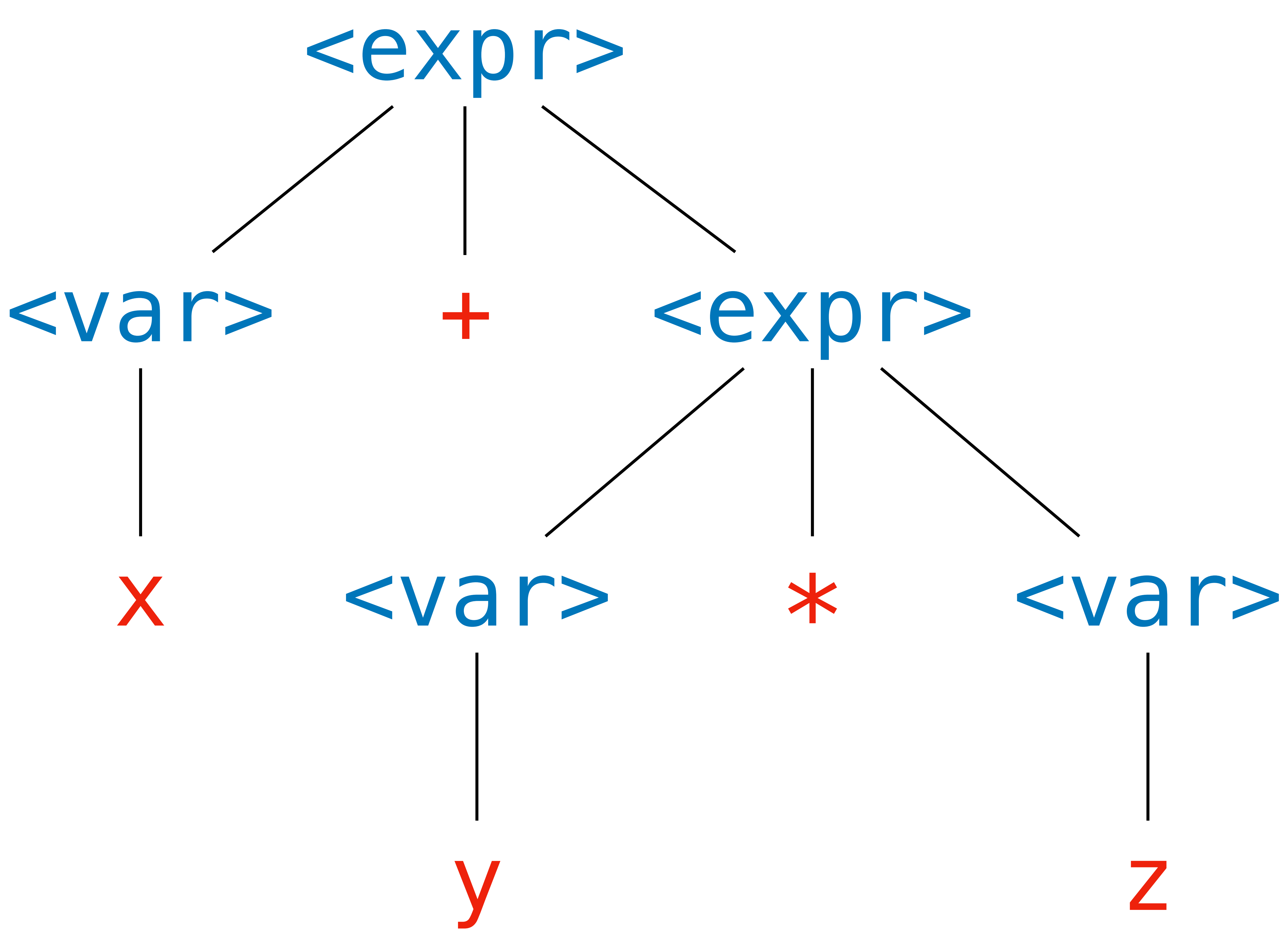
```
<type> ::= <base> -> <type>
         | <base>
<base> ::= ()
```

```
<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 <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)**

Higher precedence means it "binds tighter".

Dealing with Precedence

```
<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.`

Understanding Check

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

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

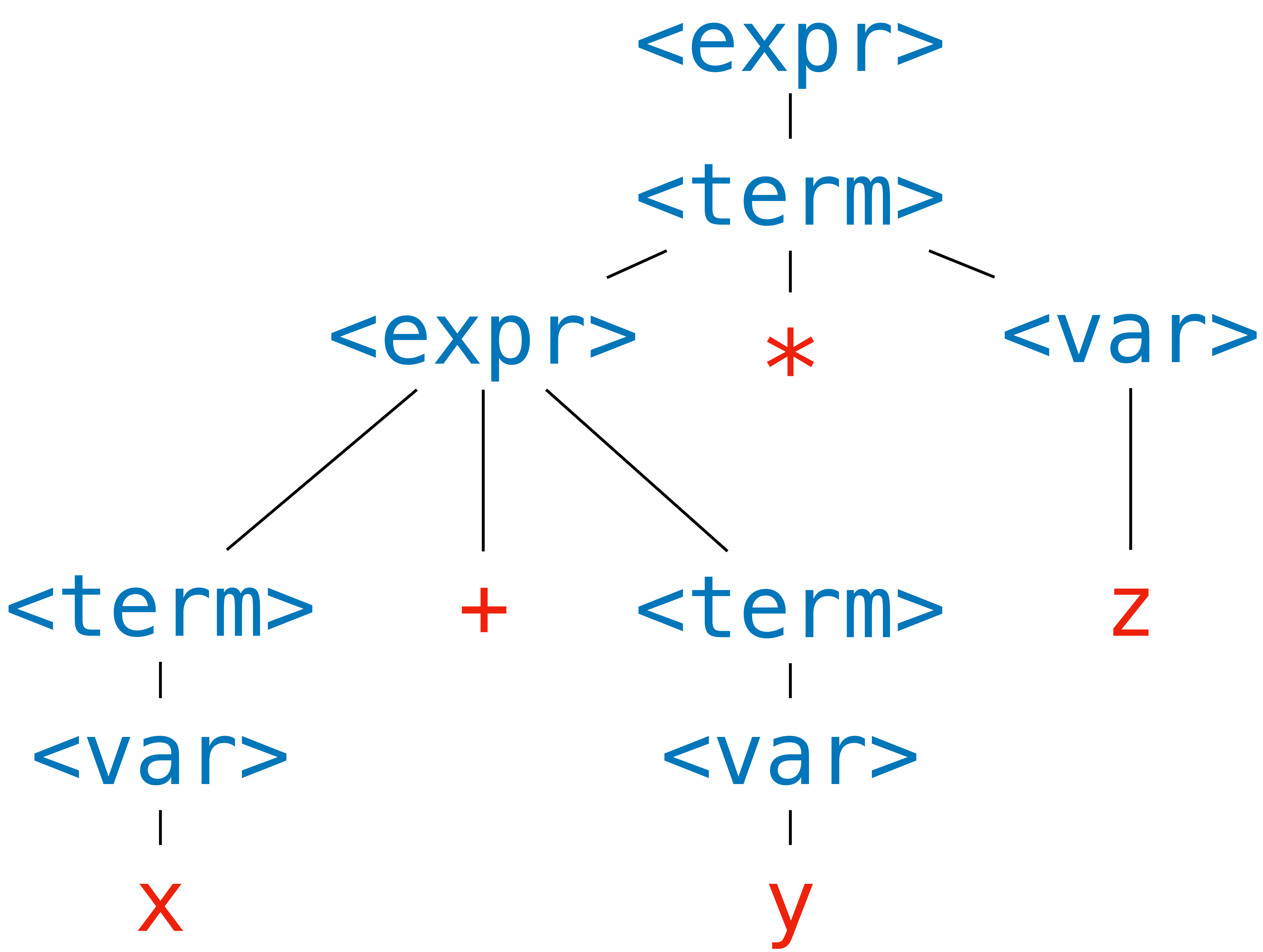
Answer

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

x + y * z

The Issue of Parentheses Returns

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



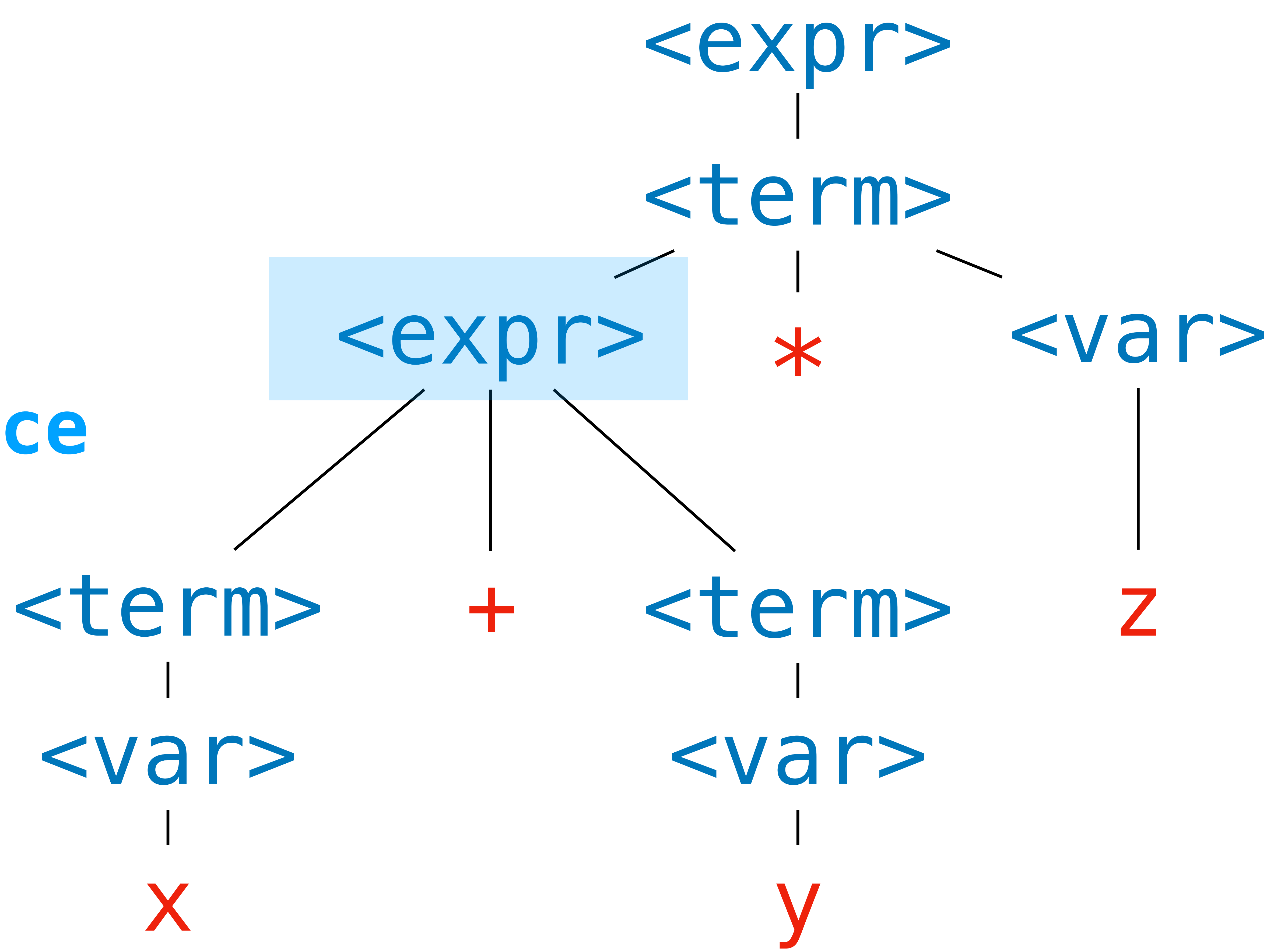
"multiply the sum of x and y with z"

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, 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.