Parser Generators

Concepts of Programming Languages
Lecture 13

CAS CS 320

Outline

- Extend our BNF syntax to be a bit more convenient
- Introduce **parser generators**
- Discuss **lexical analysis**
- Demo **Menhir**, the parser generator for this course

Recap

Recall: Example

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

Recall: Example

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

tokens (terminal symbols)

Recall: Example

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

abstractions (non-terminal symbols)

tokens (terminal symbols)

Recall: 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)

Recall: Parse Trees

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

Recall: Parse Trees

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

<expr>

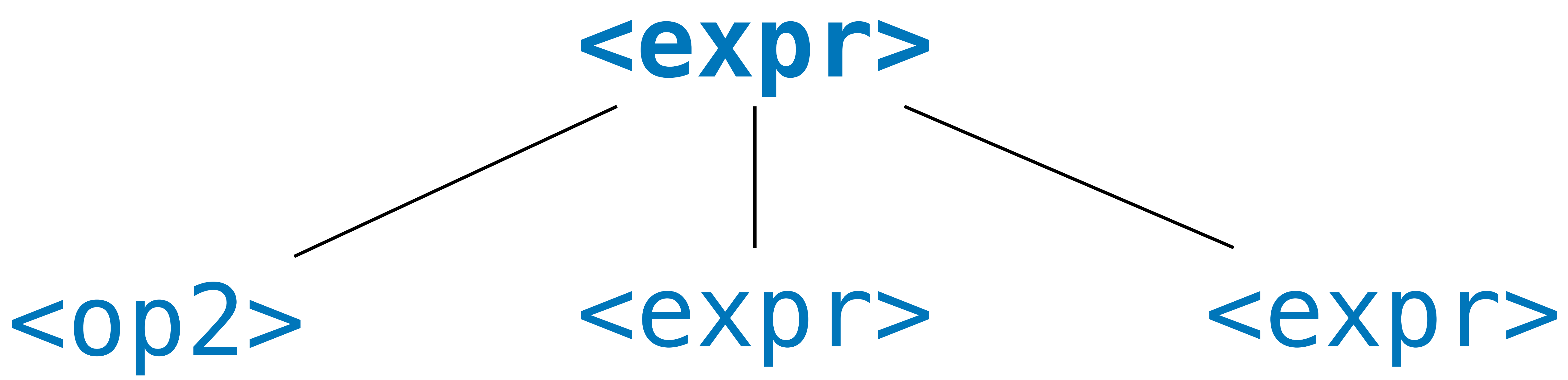
<expr>

Recall: Parse Trees

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

<expr>

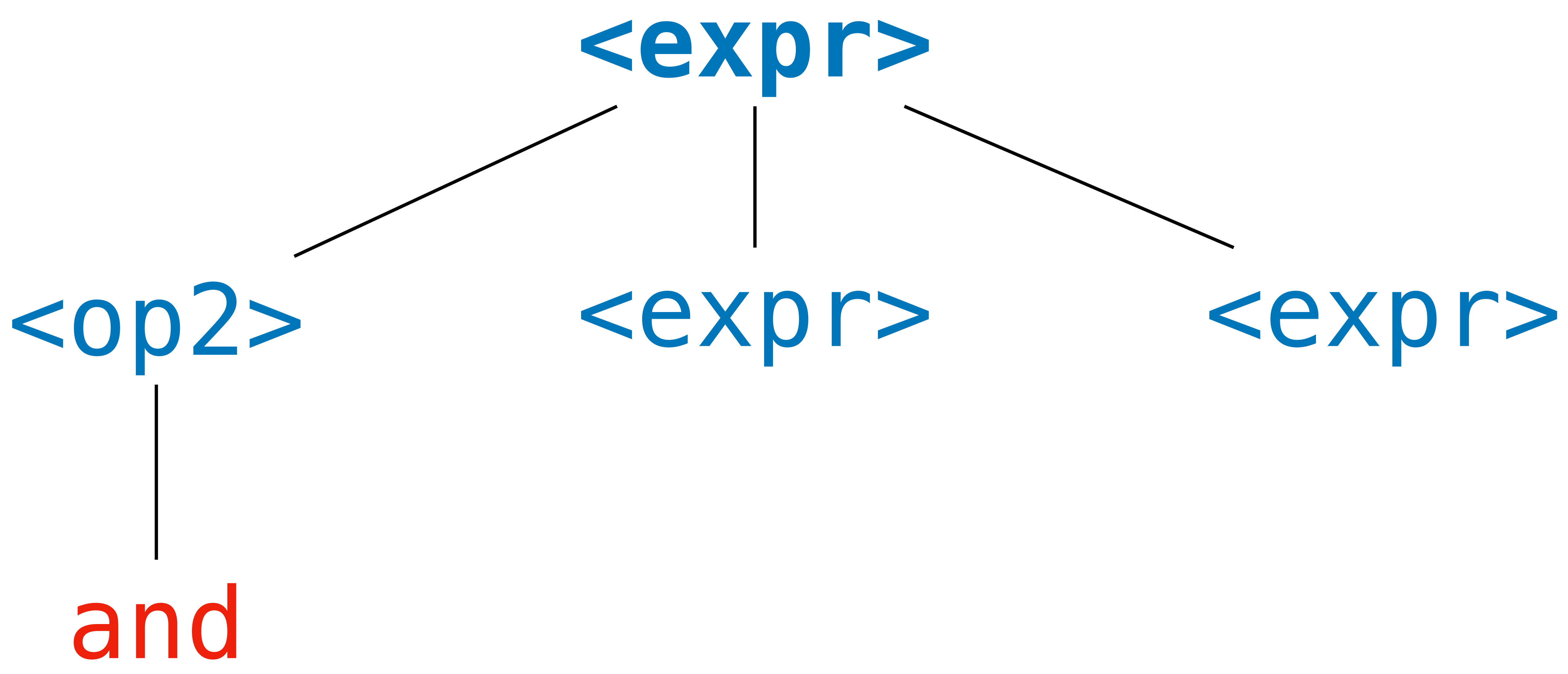
<op2> <expr> <expr>



Recall: Parse Trees

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

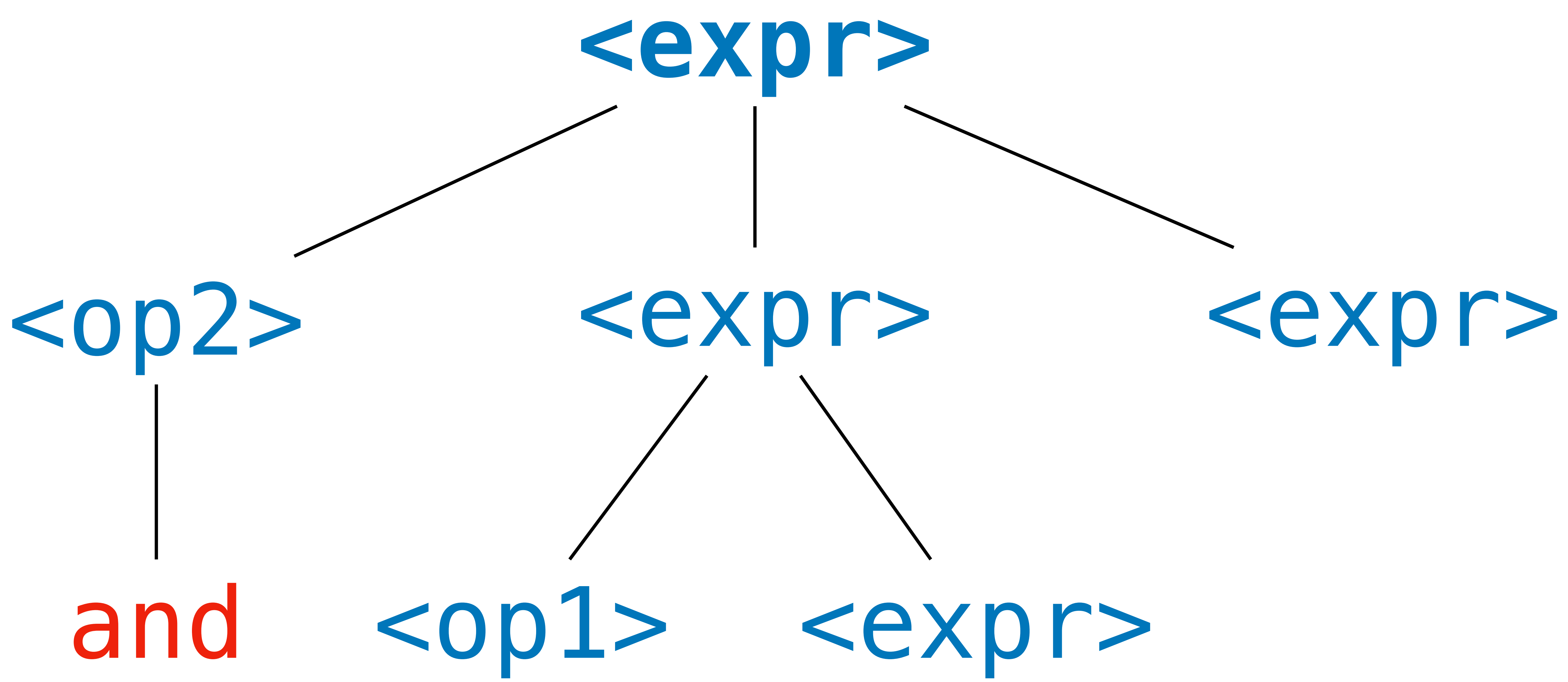
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
```



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

Recall: Parse Trees

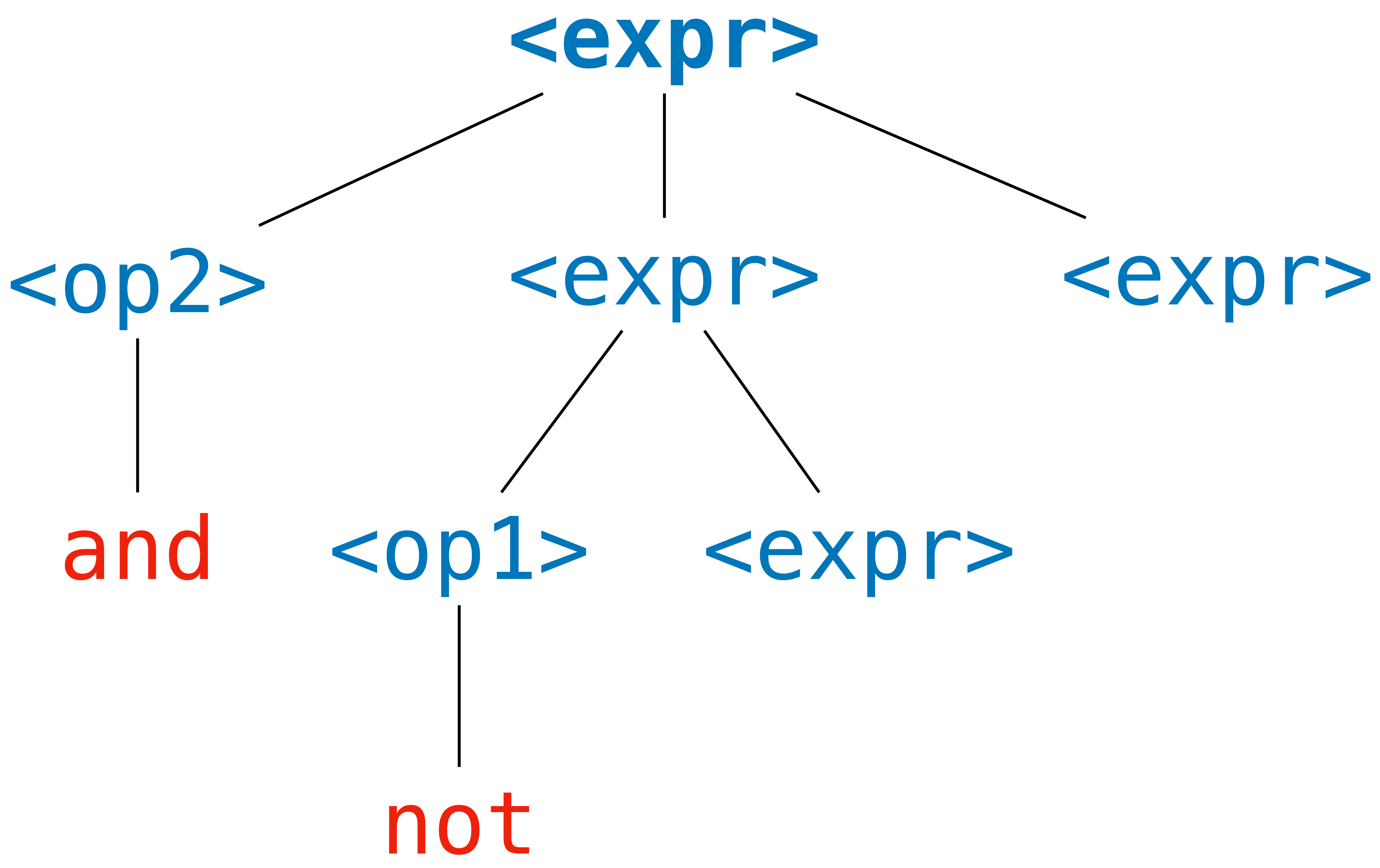
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
```



Recall: Parse Trees

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

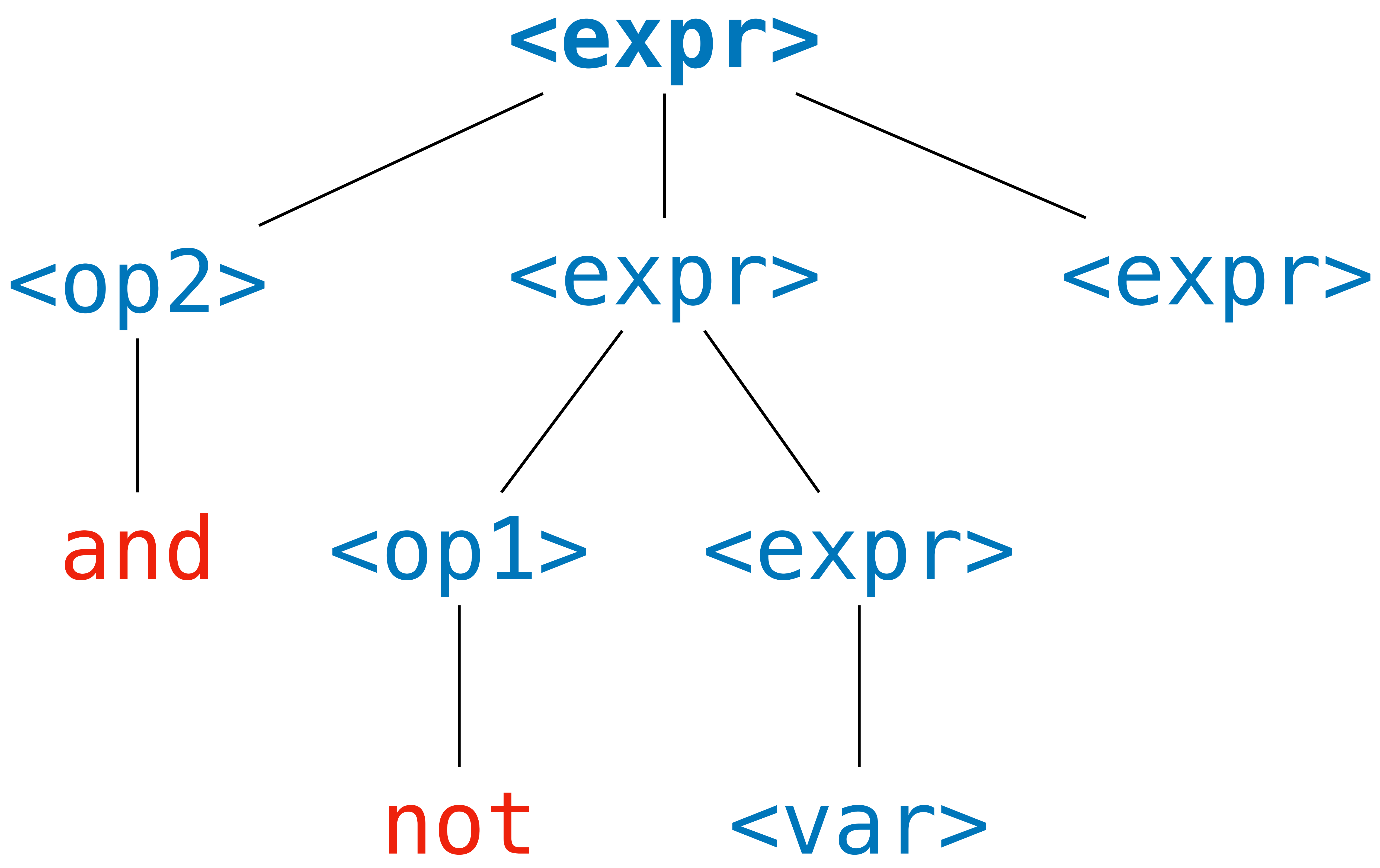
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
```



Recall: Parse Trees

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

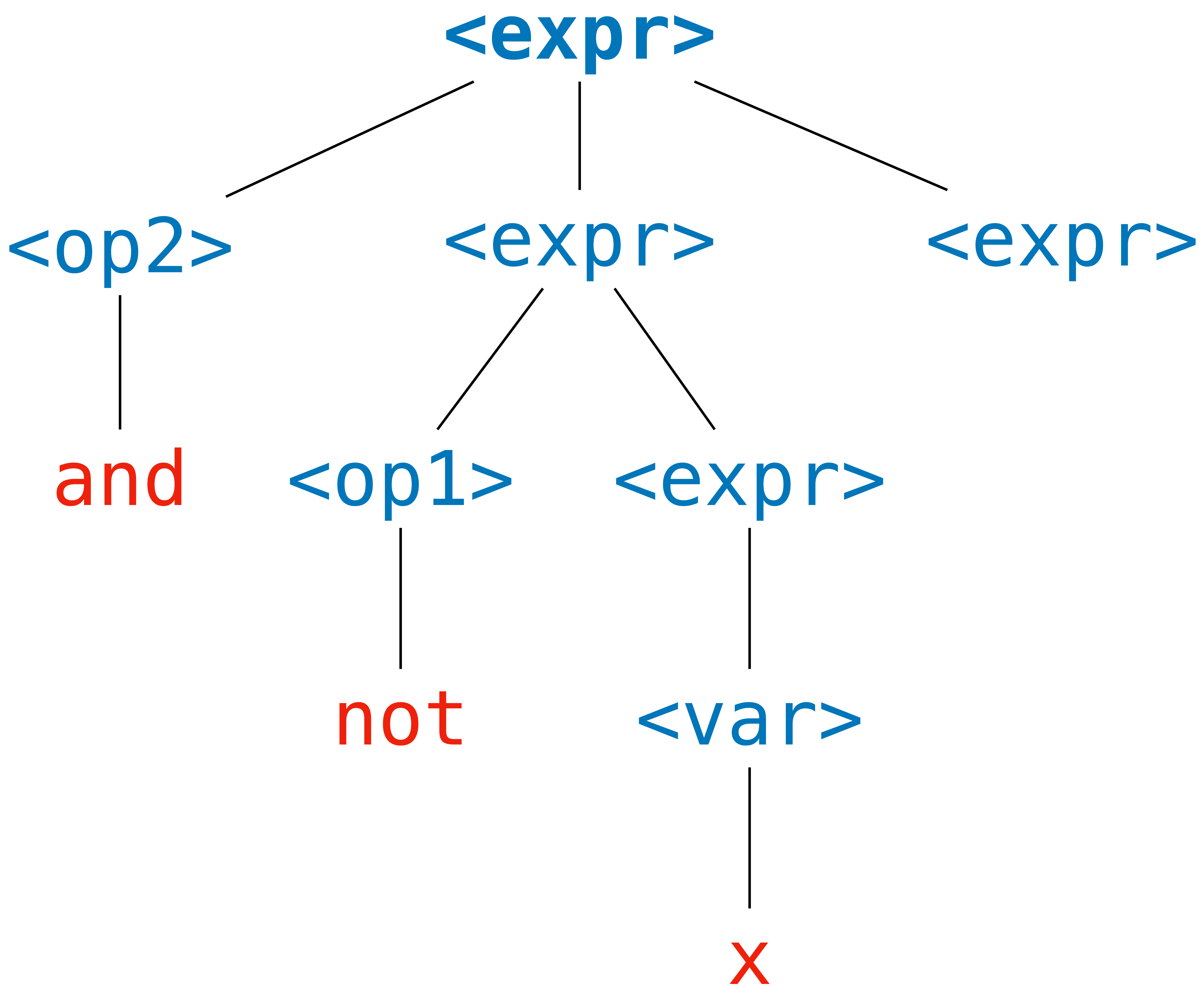
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
```



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

Recall: Parse Trees

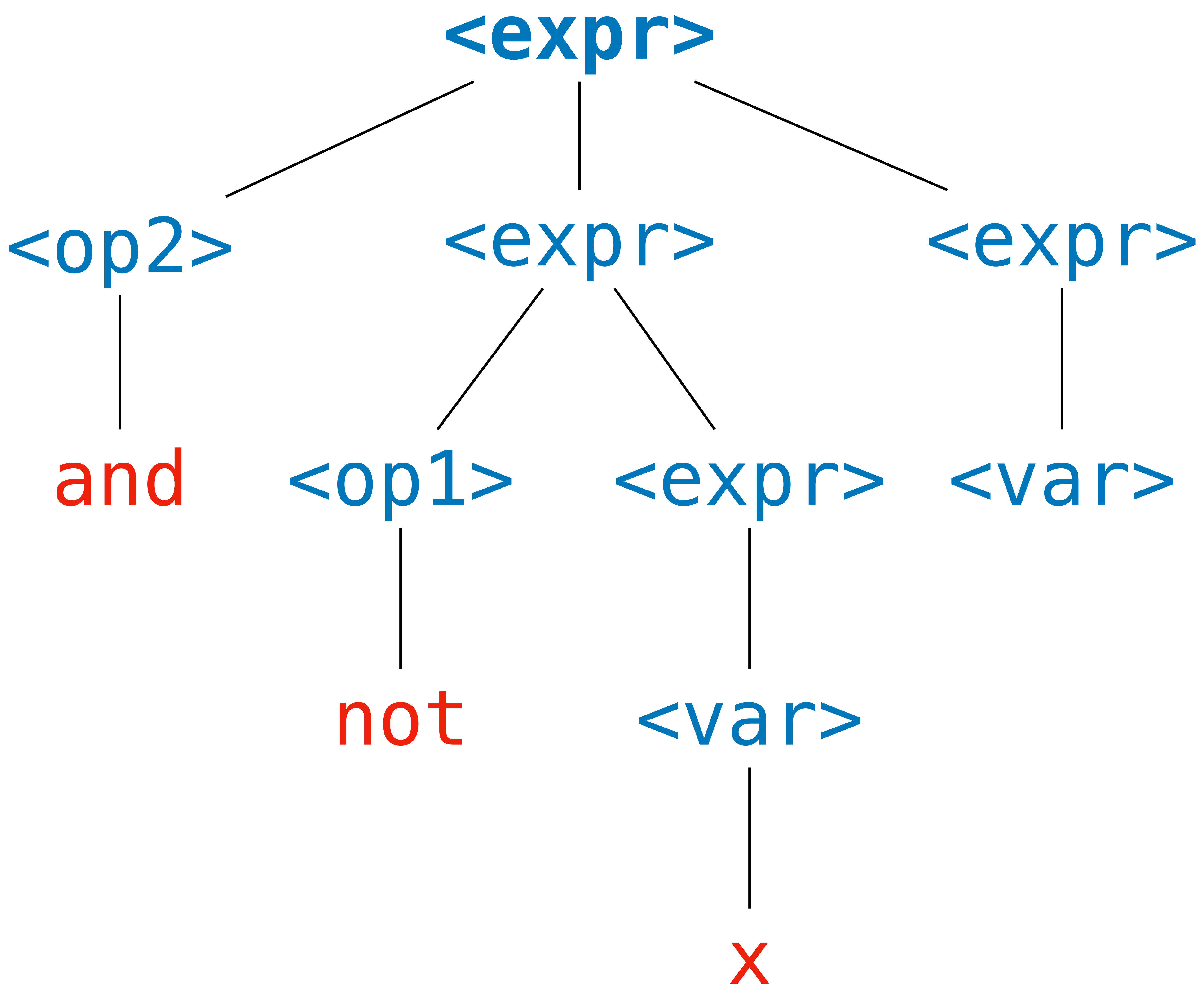
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
```



Recall: Parse Trees

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

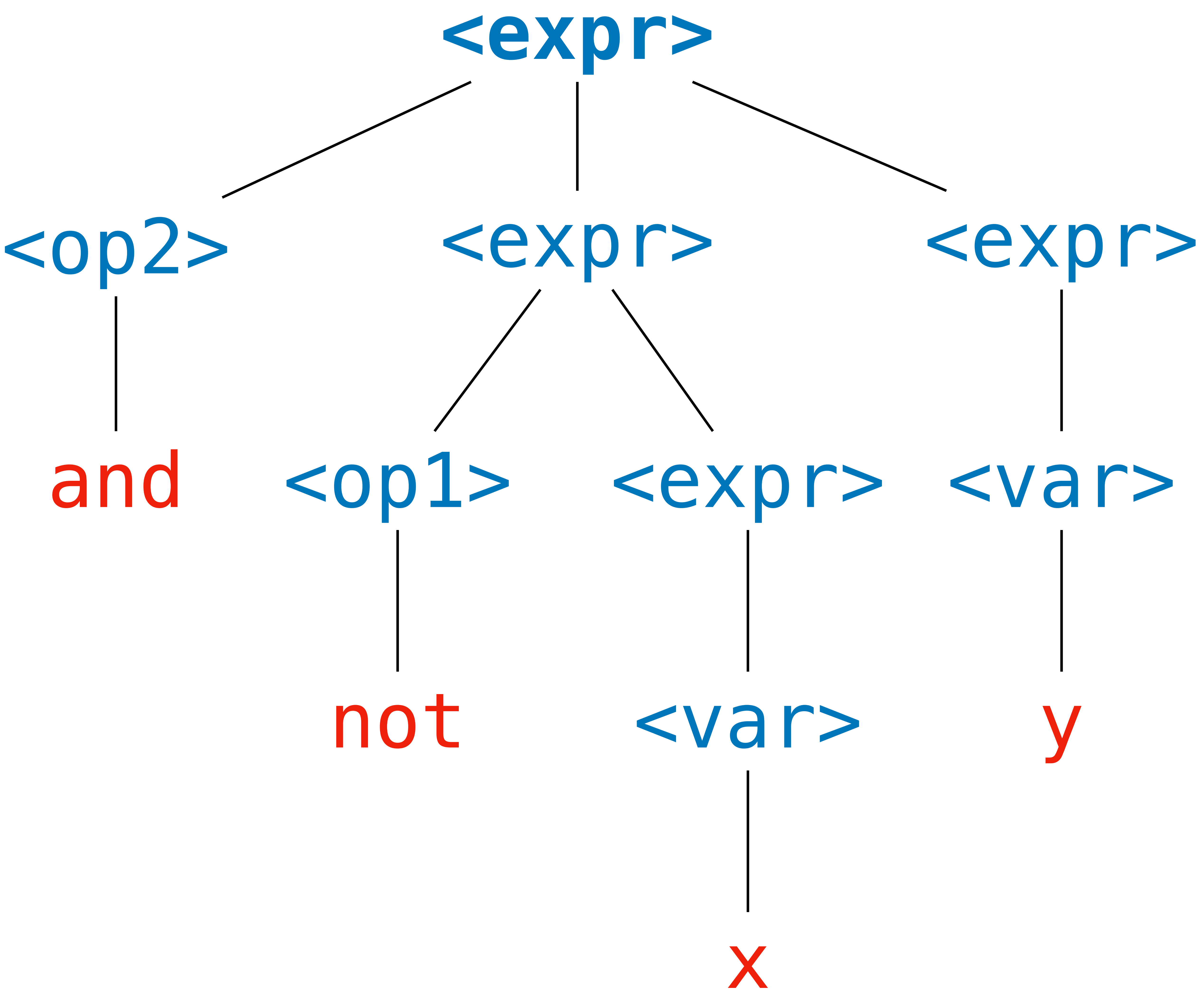
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
and not x <var>
```



Recall: Parse Trees

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

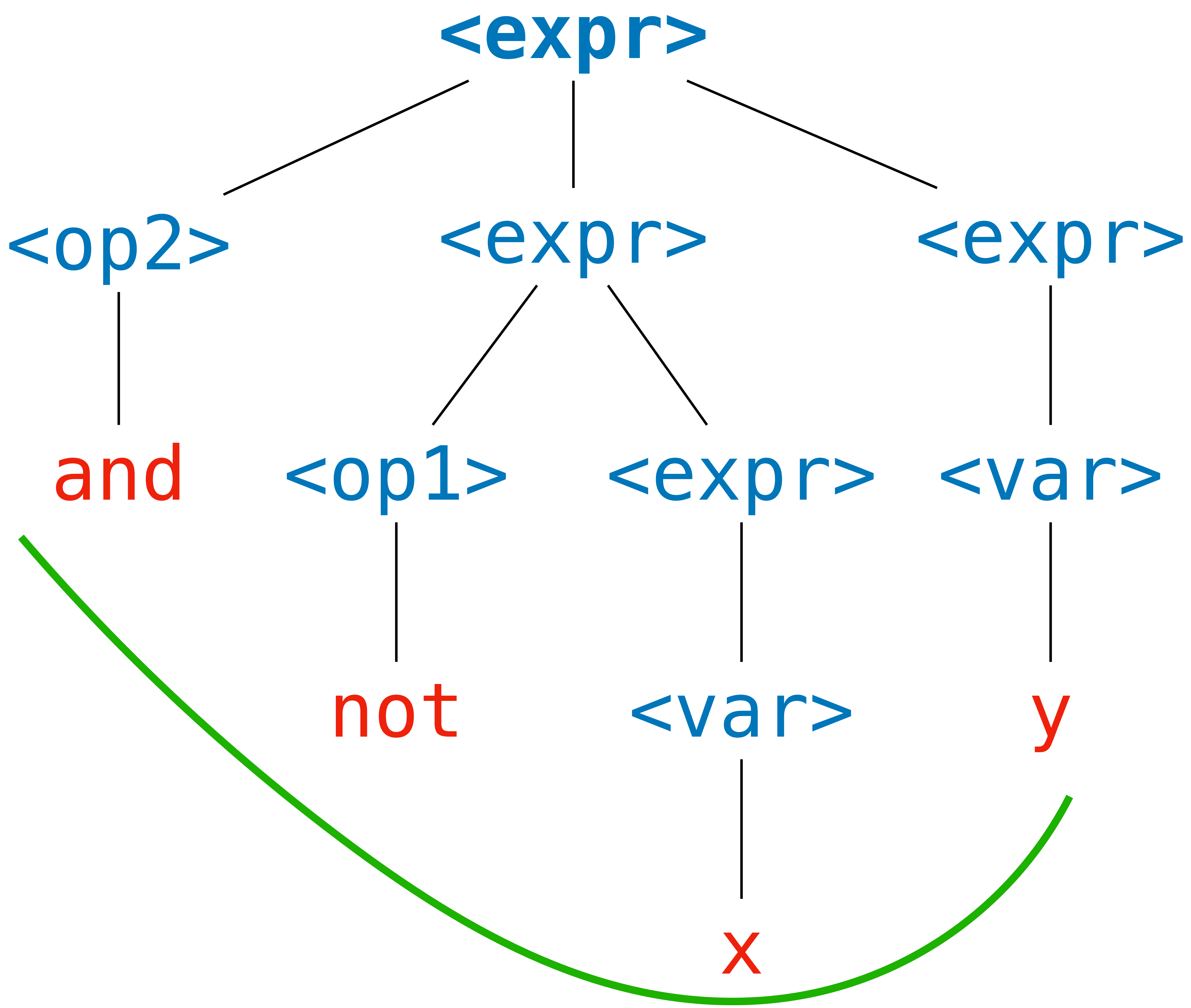
```
<expr>
<op2> <expr> <expr>
and <expr> <expr>
and <op1> <expr> <expr>
and not <expr> <expr>
and not <var> <expr>
and not x <expr>
and not x <var>
and not x y
```



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

Recall: 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
```



Recall: Ambiguity in Formal Grammar

Recall: Ambiguity in Formal Grammar

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

Recall: 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
```

Recall: 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
```

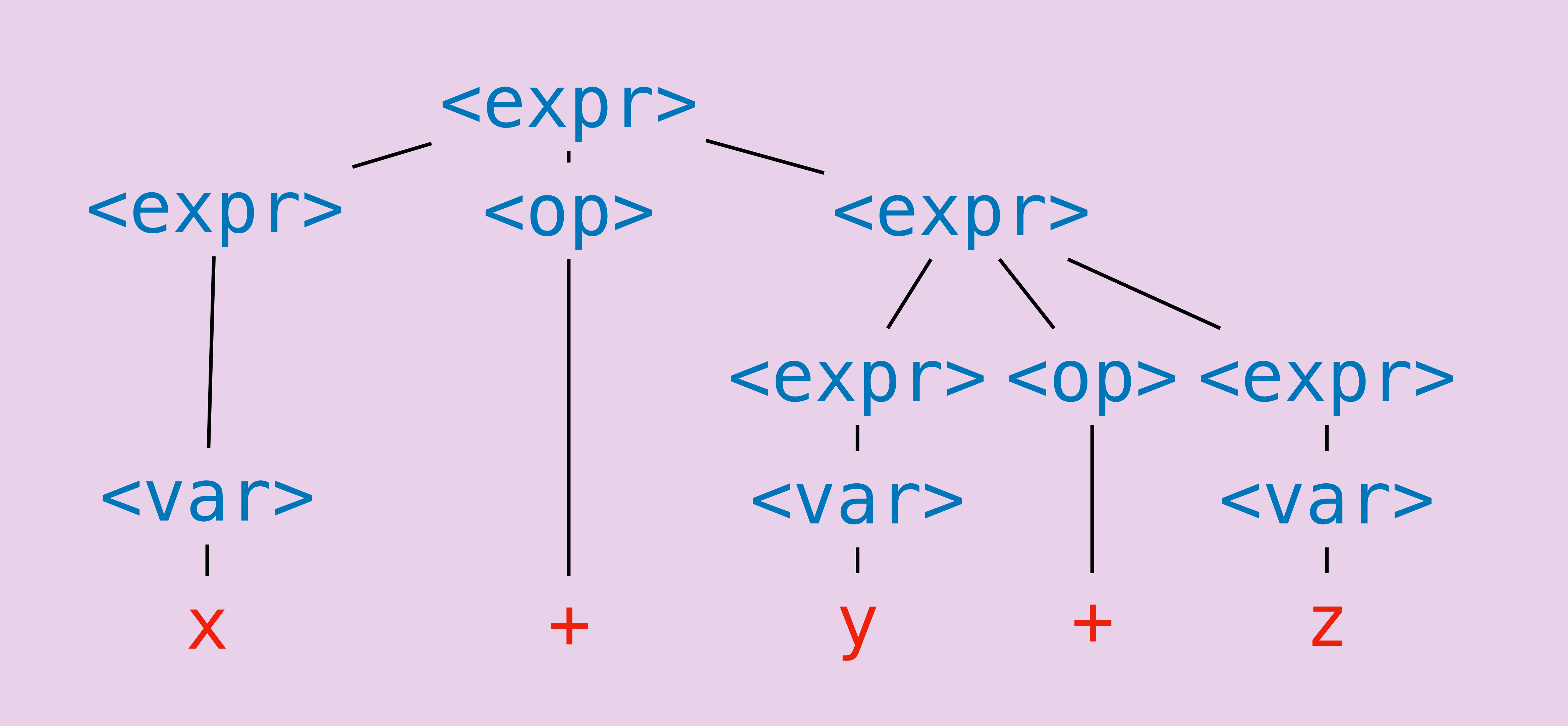
x + y + z can be derived as

Recall: 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

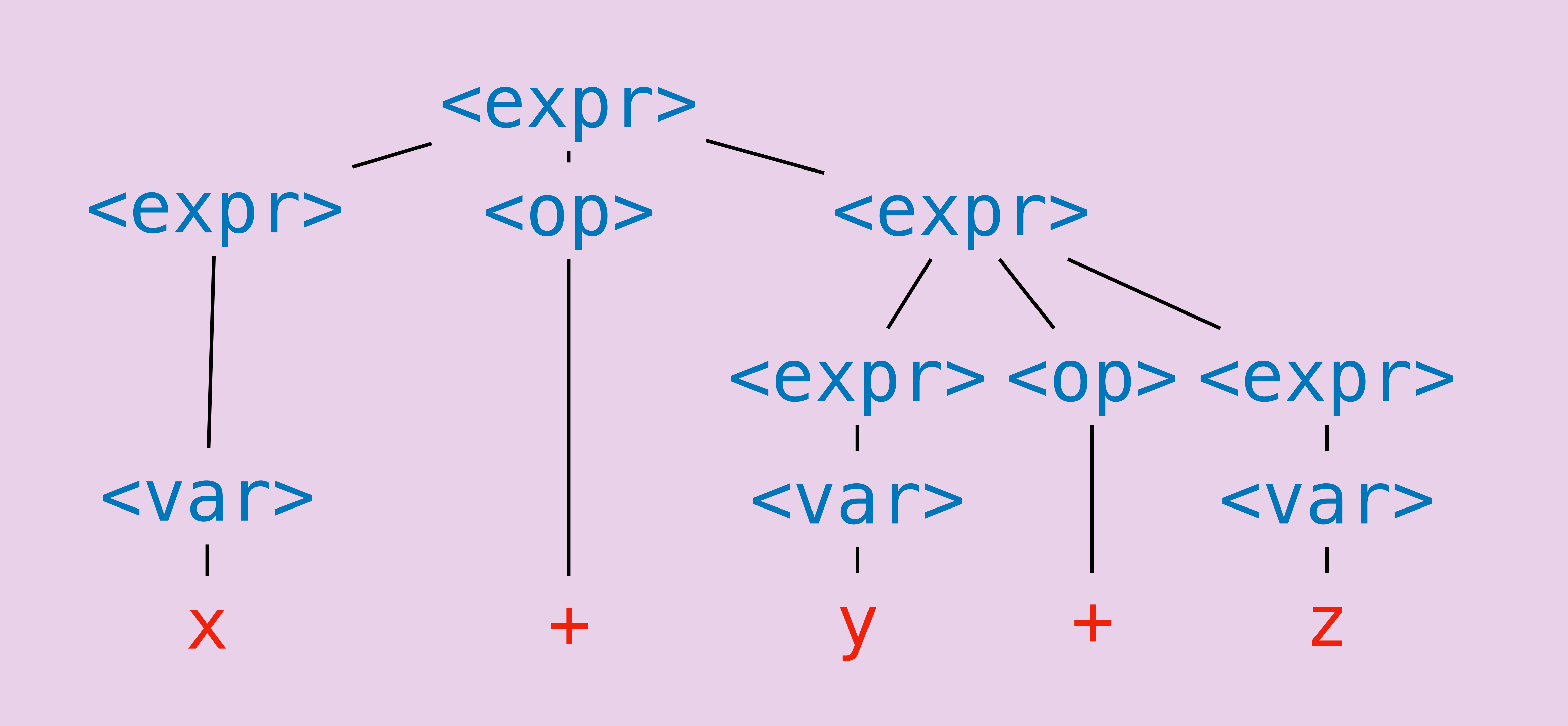


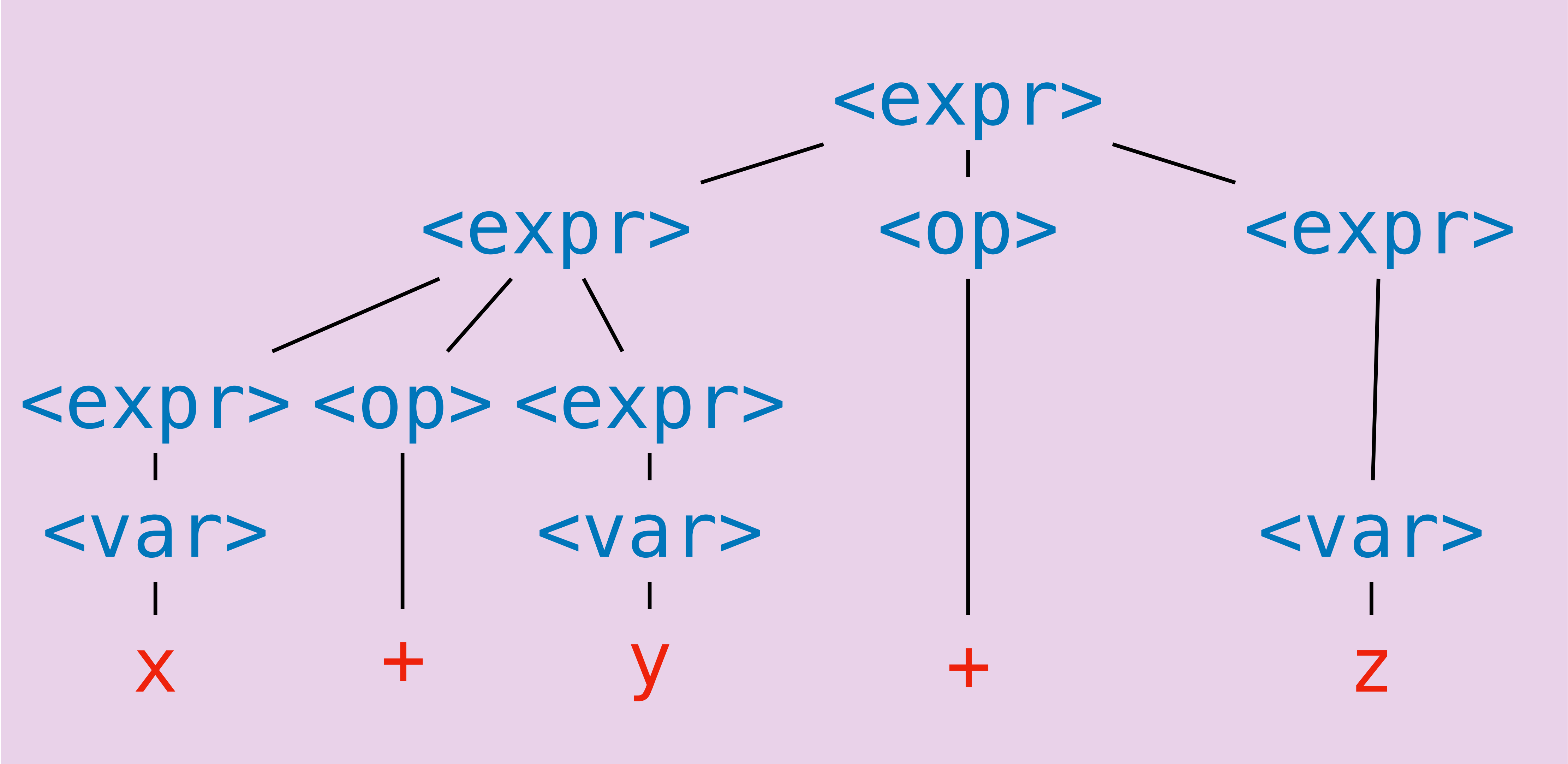
Recall: Ambiguity in Formal Grammar

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





Practice Problem

```
<expr> ::= fun <var> -> <expr>
         | <expr> <expr>
         | <var>
<var>  ::= x
```

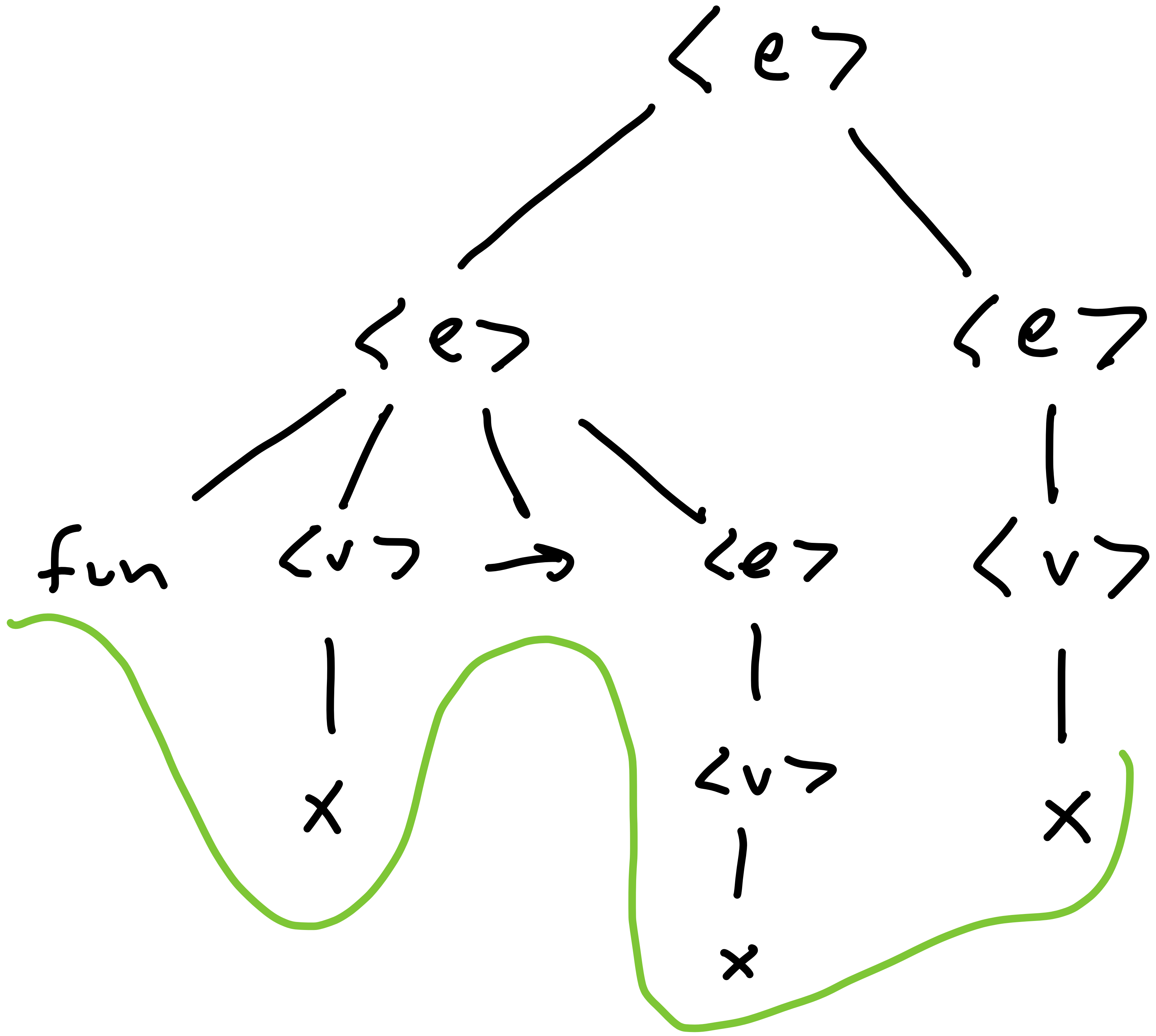
Demonstrate that the above grammar is ambiguous

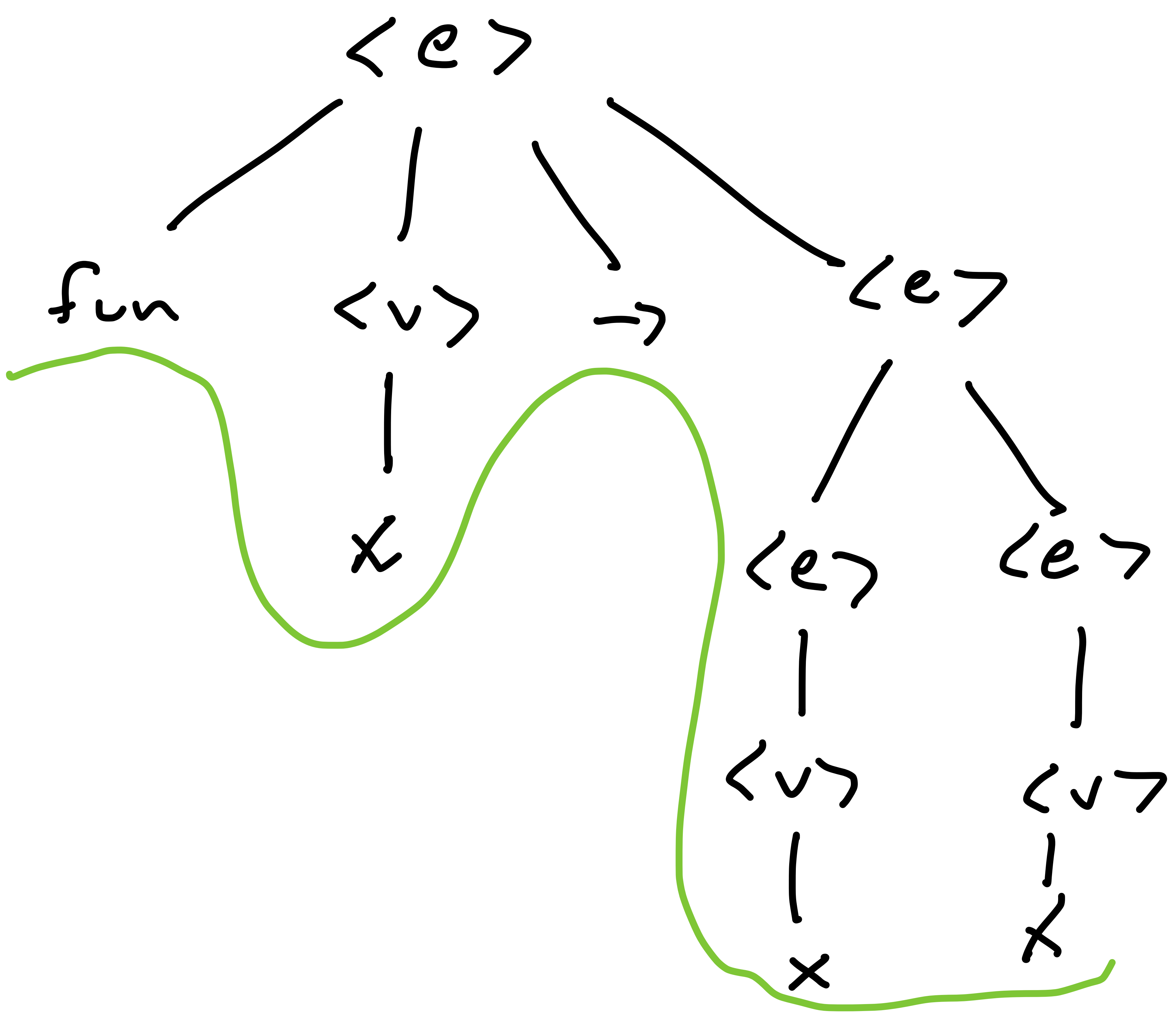
Solution

```
<expr> ::= fun <var> -> <expr>
         | <expr> <expr>
         | <var>
<var>  ::= x
```

fun x → x x

fun x → x x





x x x

x x x

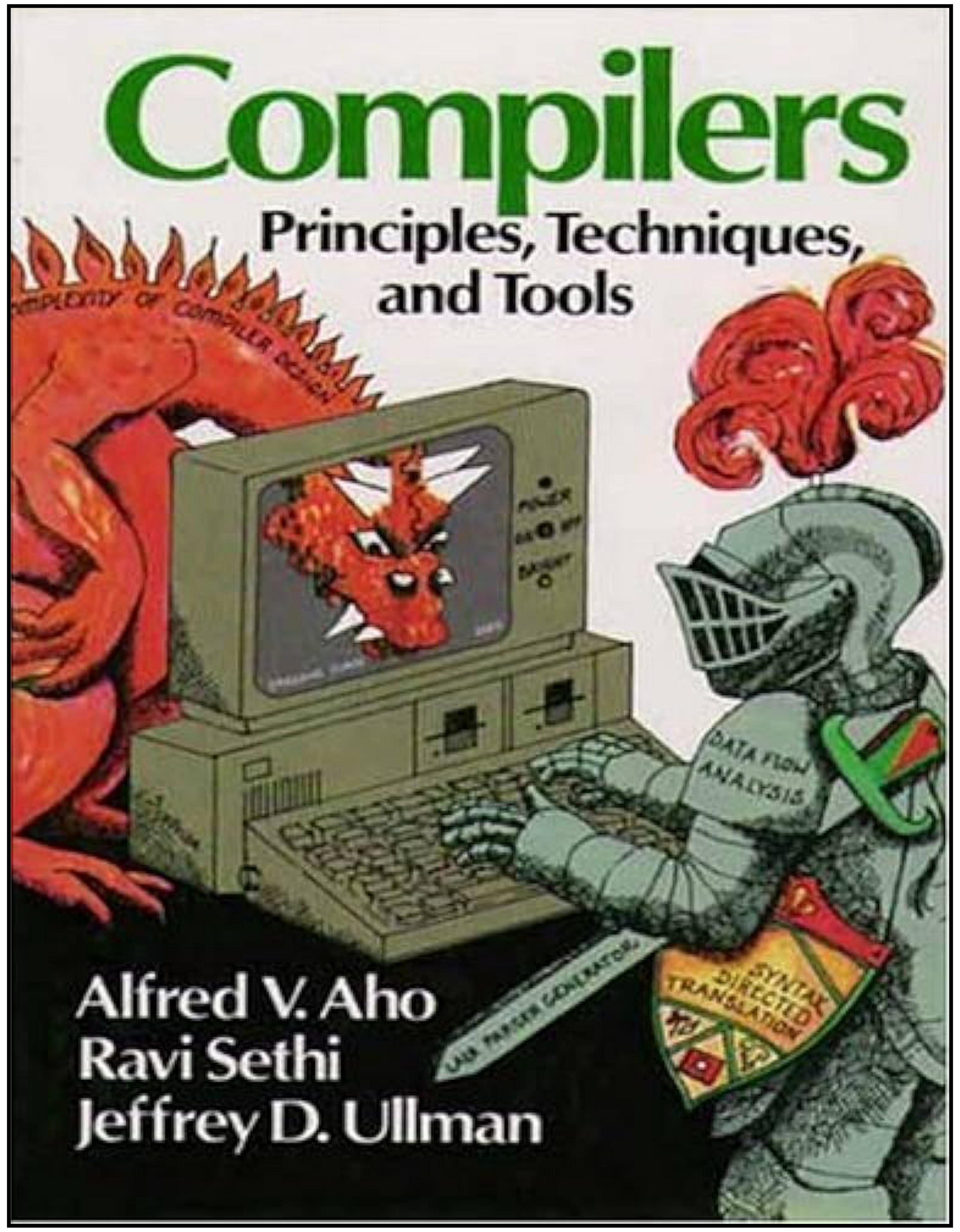
Motivation

A Note on "History"

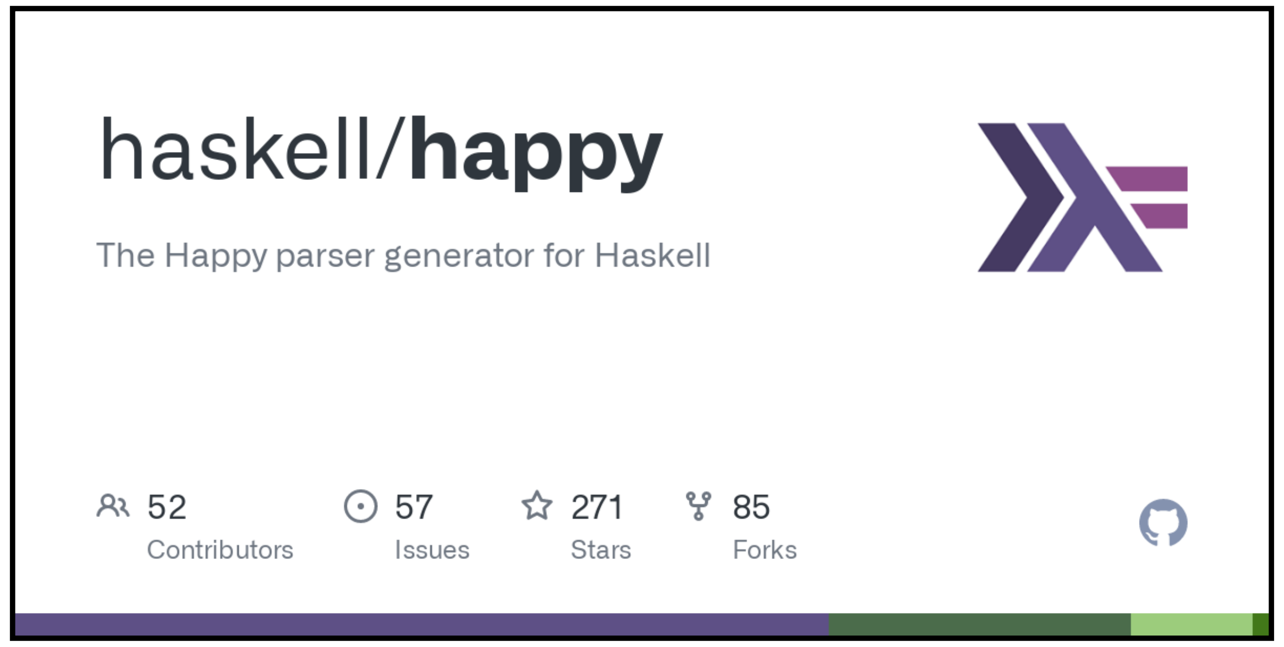
Lexical analysis and parsing are typically associated with ***compiler design***

Compiler design was once a fundamental requirement in CS programs. *This is not really the case anymore*

Also, we have ***parser generators***



Parser Generators



(Beaver)

Parser generators are programs which, given a representation of a language (e.g., as an ***EBNF grammar***), build a parser for you

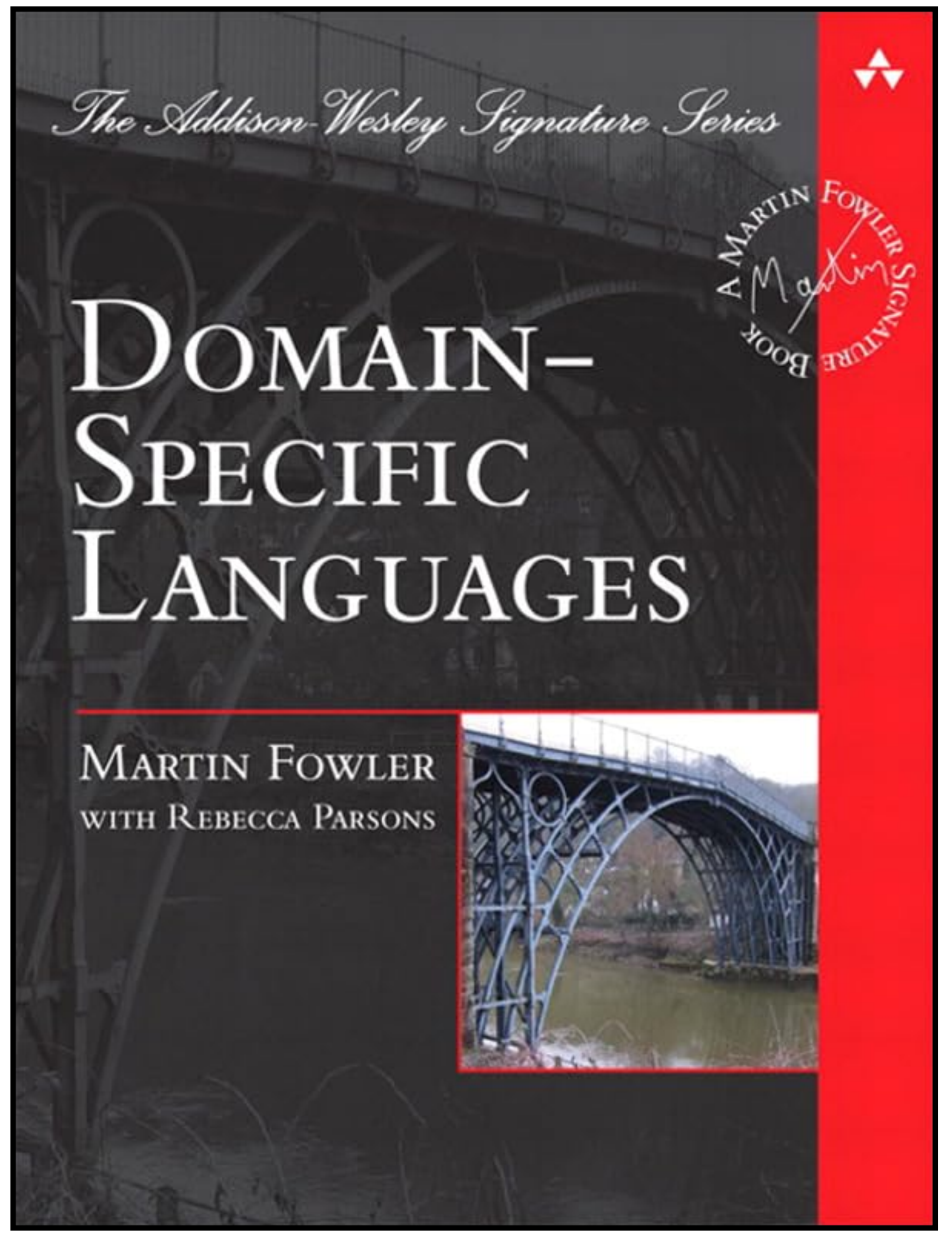
(So there was a point to learning (E)BNF for the "real-world")

Aside: Domain-Specific Languages

Domain-specific languages (DSLs) are simple programming languages for domain-specific tasks, e.g.

» Emacs Lisp
» SQL

*We need **parsers** for these languages if we want to use them...*



Extended BNF

Extended BNF

```
<expr>           ::= <only-mul-div> { (+ | -) <only-mul-div> }
<only-mul-div>   ::= <var-or-parens> { (* | /) <var-or-parens> }
<var-or-parens> ::= x | ( <expr> )
```

Extended BNF

```
<expr>          ::= <only-mul-div> { (+ | -) <only-mul-div> }
<only-mul-div>  ::= <var-or-parens> { (* | /) <var-or-parens> }
<var-or-parens> ::= x | ( <expr> )
```

Extended BNF is essentially syntactic sugar. It let's us express BNF grammars in more compact way

Extended BNF

```
<expr>            ::= <only-mul-div> { (+ | -) <only-mul-div> }
<only-mul-div>  ::= <var-or-parens> { (* | /) <var-or-parens> }
<var-or-parens> ::= x | ( <expr> )
```

Extended BNF is essentially syntactic sugar. It let's us express BNF grammars in more compact way

EBNF is not more expressive than BNF

Extended BNF

```
<expr>          ::= <only-mul-div> { (+ | -) <only-mul-div> }
<only-mul-div>  ::= <var-or-parens> { (* | /) <var-or-parens> }
<var-or-parens> ::= x | ( <expr> )
```

Extended BNF is essentially syntactic sugar. It let's us express BNF grammars in more compact way

EBNF is not more expressive than BNF

But it allows us to specify:

Extended BNF

```
<expr>            ::= <only-mul-div> { (+ | -) <only-mul-div> }
<only-mul-div>    ::= <var-or-parens> { (* | /) <var-or-parens> }
<var-or-parens>   ::= x | ( <expr> )
```

Extended BNF is essentially syntactic sugar. It let's us express BNF grammars in more compact way

EBNF is not more expressive than BNF

<u>But it allows us to specify:</u>

» Optional parts of production rule

Extended BNF

```
<expr>           ::= <only-mul-div> { (+ | -) <only-mul-div> }
<only-mul-div>   ::= <var-or-parens> { (* | /) <var-or-parens> }
<var-or-parens> ::= x | ( <expr> )
```

Extended BNF is essentially syntactic sugar. It let's us express BNF grammars in more compact way

EBNF is not more expressive than BNF

<u>But it allows us to specify:</u>

» Optional parts of production rule
» Repeated parts of a production rule

Extended BNF

```
<expr>          ::= <only-mul-div> { (+ | -) <only-mul-div> }
<only-mul-div>  ::= <var-or-parens> { (* | /) <var-or-parens> }
<var-or-parens> ::= x | ( <expr> )
```

Extended BNF is essentially syntactic sugar. It let's us express BNF grammars in more compact way

EBNF is not more expressive than BNF

<u>But it allows us to specify:</u>

» Optional parts of production rule
» Repeated parts of a production rule

Note: EBNF means different things to different people

Optional Syntax

BNF:

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

EBNF:

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

Menhir:

```
expr =
  | IF; e1 = expr; THEN; e2 = expr; e3_opt = else?
    { match e3_opt with
      | None -> It (e1, e2)
      | Some e3 -> Ite (e1, e2, e3)
    }
else =
  | ELSE; e = expr { e }
```

Repetition Syntax

BNF:

```
<word> ::= <letter> | <letter> <word>
```

EBNF:

```
<word> ::= <letter> { <letter> }
```

Menhir:

```
word =
  | l = letter; ls = letter*
    { String.of_list (l :: ls) }
```

Interlude: Regular Expressions

Regular Grammars

```
<nonterminal> ::= terminal
<nonterminal> ::= terminal <nonterminal>
<nonterminal> ::= ϵ (the empty string)
```

A **regular grammar** is a BNF grammar with the above kinds of rules

Regular grammars are easier to parse

Example

<s> ::= a <s>
<s> ::= b <a>
<a> ::= ϵ
<a> ::= c <a>

$a^n b c^m$

<s>
a <s>
a a <s>
a a a <s>
a a a a <s>
aaaa b <a>
aaaabc<a>
aaaa bcc <a>
aaaa bcc

Regular Expressions (Formally)

Regular Expressions (Formally)

Regular expressions (Regex) provide a compact way of describing regular grammars:

Regular Expressions (Formally)

Regular expressions (Regex) provide a compact way of describing regular grammars:

» A **terminal symbols** is a regex

Regular Expressions (Formally)

Regular expressions (Regex) provide a compact way of describing regular grammars:

» A **terminal symbols** is a regex

» **[t1 ... tk]** is a regex describing an any one of the symbols **t1, t2, ..., tk**

Regular Expressions (Formally)

Regular expressions (Regex) provide a compact way of describing regular grammars:

» A **terminal symbols** is a regex

» **[t1 ... tk]** is a regex describing an any one of the symbols **t1, t2, ..., tk**

» **(e1 | ... | ek)** is a regex describing any one of the expressions **e1, e2, ..., ek**

Regular Expressions (Formally)

Regular expressions (Regex) provide a compact way of describing regular grammars:

» A **terminal symbols** is a regex

» **[t1 ... tk]** is a regex describing an any one of the symbols **t1, t2, ..., tk**

» **(e1 | ... | ek)** is a regex describing any one of the expressions **e1, e2, ..., ek**

» **exp**$*$ is a regex describing zero or more occurrences of **exp**

Regular Expressions (Formally)

Regular expressions (Regex) provide a compact way of describing regular grammars:

» A **terminal symbols** is a regex

» **[t1 ... tk]** is a regex describing an any one of the symbols **t1, t2, ..., tk**

» **(e1 | ... | ek)** is a regex describing any one of the expressions **e1, e2, ..., ek**

» **exp*** is a regex describing zero or more occurrences of **exp**

» **exp+** is a regex describing one or more occurrences of **exp**

Regular Expressions (Formally)

0^*1^*

Regular expressions (Regex) provide a compact way of describing regular grammars:

√ » A **terminal symbols** is a regex

» **[t1 ... tk]** is a regex describing an any one of the symbols **t1, t2, ..., tk**

» **(e1 | ... | ek)** is a regex describing any one of the expressions **e1, e2, ..., ek**

√ » **exp**∗ is a regex describing zero or more occurrences of **exp**

» **exp**+ is a regex describing one or more occurrences of **exp**

» **exp**? is a regex describing zero or one occurrences of **exp**

e_1 ... e_k is a regex

Example

```
<s> ::= a <s>
<s> ::= b <a>
<a> ::= ϵ
<a> ::= c <a>
```

is equivalent to

0 or more

a*bc*

0 or more

or

'a'* 'b' 'c'*

in ocamllex syntax

Example: Numbers and Variables

optional

`-?[0-9]+`

numbers

digits

-13

144

99

-0000

lower case

`[a-z][a-z0-9A-Z_']*`

variables

alphanumeric

x

xyz

x_X_z

a___''' _

We'll leave it there, take CS332 if you want more, or read the Wikipedia page...

Lexical Analysis

The "Lexing" Problem

$$\texttt{"let"} \approx [\texttt{'l'},\ \texttt{'e'},\ \texttt{'t'}] \mapsto \textit{\textbf{LET}}$$

$$\texttt{"fun"} \approx [\texttt{'f'},\ \texttt{'u'},\ \texttt{'n'}] \mapsto \textit{\textbf{FUN}}$$

The "Lexing" Problem

$$\texttt{"let"} \approx \texttt{['l', 'e', 't']} \mapsto \textit{\textbf{LET}}$$

$$\texttt{"fun"} \approx \texttt{['f', 'u', 'n']} \mapsto \textit{\textbf{FUN}}$$

The Goal. *Convert a stream of characters into a stream of tokens*

The "Lexing" Problem

$$\texttt{"let"} \approx [\texttt{'l'}, \texttt{'e'}, \texttt{'t'}] \mapsto \textit{LET}$$

$$\texttt{"fun"} \approx [\texttt{'f'}, \texttt{'u'}, \texttt{'n'}] \mapsto \textit{FUN}$$

The Goal. *Convert a stream of characters into a stream of tokens*

» Characters are grouped so together so they correspond to the *smallest units* at the level of the language

The "Lexing" Problem

$$\texttt{"let"} \approx \texttt{['l', 'e', 't']} \mapsto \textit{LET}$$

$$\texttt{"fun"} \approx \texttt{['f', 'u', 'n']} \mapsto \textit{FUN}$$

The Goal. *Convert a stream of characters into a stream of tokens*

» Characters are grouped so together so they correspond to the *smallest units* at the level of the language

» Whitespace and comments are *ignored*

The "Lexing" Problem

$$\texttt{"let"} \approx \texttt{['l', 'e', 't']} \mapsto \textit{\textbf{LET}}$$

$$\texttt{"fun"} \approx \texttt{['f', 'u', 'n']} \mapsto \textit{\textbf{FUN}}$$

The Goal. *Convert a stream of characters into a stream of tokens*

» Characters are grouped so together so they correspond to the *smallest units* at the level of the language

» Whitespace and comments are *ignored*

» Syntax errors are caught, when possible

Lexing vs. Parsing

Lexing vs. Parsing

Lexical Analysis is about *small-scale* language constructs

Lexing vs. Parsing

Lexical Analysis is about *small-scale* language constructs

» keywords, names, literals

Lexing vs. Parsing

Lexical Analysis is about *small-scale* language constructs

- keywords, names, literals

Syntactic Analysis (Parsing) is about *large-scale* language constructs

Lexing vs. Parsing

Lexical Analysis is about *small-scale* language constructs

» keywords, names, literals

Syntactic Analysis (Parsing) is about *large-scale* language constructs

» expressions, statements, modules

Why separate them?

Why separate them?

*Good question...*for simple implementations, we don't

<u>But there are benefits for larger projects:</u>

Why separate them?

*Good question...*for simple implementations, we don't

But there are benefits for larger projects:

- **Simplicity.** It's *easier to think about* parsing if we don't need to worry about whitespace, characters, etc.

Why separate them?

*Good question...*for simple implementations, we don't

<u>But there are benefits for larger projects:</u>

» **Simplicity.** It's *easier to think about* parsing if we don't need to worry about whitespace, characters, etc.

» **Portability.** Files are finicky things, handled differently across different operating systems. *Abstracting this away* for parsing is just good software engineering

Lexemes and Tokens

input program:	fun	l	->	l	++	[	100	]
lexemes:	"fun"	"l"	"->"	"l"	"++"	"["	"100"	"]"
tokens:	FUN	(ID "l")	ARR	(ID "l")	(OP "++")	LBRAK	(INT 100)	RBRAK

Lexemes and Tokens

input program:	fun	l	->	l	++	[	100	]
lexemes:	"fun"	"l"	"->"	"l"	"++"	"["	"100"	"]"
tokens:	FUN	(ID "l")	ARR	(ID "l")	(OP "++")	LBRAK	(INT 100)	RBRAK

A **lexeme** is a sequence of characters associated a syntactic unit in a language

Lexemes and Tokens

input program:	fun	l	->	l	++	[	100	]
lexemes:	"fun"	"l"	"->"	"l"	"++"	"["	"100"	"]"
tokens:	FUN	(ID "l")	ARR	(ID "l")	(OP "++")	LBRAK	(INT 100)	RBRAK

A **lexeme** is a sequence of characters associated a syntactic unit in a language

A **token** is a lexeme together with information about what kind of unit it is

Lexemes and Tokens

```
input program:  fun      l   ->      l      ++  [         100  ]
     lexemes: "fun"     "l"  "->"    "l"    "++" "["      "100" "]"
      tokens:  FUN  (ID "l") ARR  (ID "l") (OP "++") LBRAK (INT 100)  RBRAK
```

A **lexeme** is a sequence of characters associated a syntactic unit in a language

A **token** is a lexeme together with information about what kind of unit it is

» "12" and "234" are both INT_LITS, whereas "let" is a KEYWORD.

Lexemes and Tokens

```
input program:  fun        l    ->        l          ++   [            100    ]

     lexemes: "fun"      "l"   "->"      "l"        "++" "["          "100" "]"

      tokens:  FUN   (ID "l") ARR   (ID "l") (OP "++") LBRAK (INT 100)   RBRAK
```

A **lexeme** is a sequence of characters associated a syntactic unit in a language

A **token** is a lexeme together with information about what kind of unit it is

- » "12" and "234" are both INT_LITS, whereas "let" is a KEYWORD.

We typically represent tokens as an ADT

Aside: One Token at a time

```
"  let@#_)($#@_J_@O#GKJ"  --next_token-->  (LET, "@#_)($#@_J_@O#GKJ")

"le x = 2"                --next_token-->  FAILURE
```

Aside: One Token at a time

```
"  let@#_)($#@_J_@0#GKJ"  --next_token-->  (LET, "@#_)($#@_J_@0#GKJ")

"le x = 2"  --next_token-->  FAILURE
```

The approach:

Aside: One Token at a time

```
"  let@#_)($#@_J_@O#GKJ" --next_token--> (LET, "@#_)($#@_J_@O#GKJ")

"le x = 2"               --next_token--> FAILURE
```

The approach:

» Given a stream of characters, determine if there is a valid lexeme at the *beginning*

Aside: One Token at a time

`"  let@#_)($#@_J_@O#GKJ"` $\xrightarrow{\texttt{next\_token}}$ `(LET, "@#_)($#@_J_@O#GKJ")`

`"le x = 2"` $\xrightarrow{\texttt{next\_token}}$ **FAILURE**

The approach:

» Given a stream of characters, determine if there is a valid lexeme at the *beginning*

» If there is, return its corresponding token and the *remainder of the stream*

Parsing with Menhir

General Parsing

General Parsing

In Theory. *Determine if a given sentence is recognized by a given grammar*

General Parsing

In Theory. *Determine if a given sentence is recognized by a given grammar*

In Practice. *Given a grammar, write a program which converts a string recognized by that grammar into an ADT*

Today

```
<prog>  ::= <expr>

<expr>  ::= let <var> = <expr> in <expr>
          | <expr1>

<bop>   ::= + | - | * | /

<expr1> ::= <expr1> <bop> <expr1>
          | <num>
          | <var>
          | ( <expr> )

<num>   ::= 0 ; DUMMY VALUE
<var>   ::= x ; DUMMY VALUE

; In lex.mll:
;
; let num = '-'? ['0'-'9']+
; let var = ['a'-'z' '_'] ['a'-'z' 'A'-'Z' '0'-'9' '_' '\'']*
```

Operators in order of increasing precedence:

Operator	**Associativity**
+, -	left
*, /	left

We'll be building a parser for the this grammar

A Rough Sketch

1. Specify the tokens (i.e., terminal symbols) of the grammar

2. Specify the rules of the grammar (using a BNF-like syntax)

3. Specify the rules of the lexer (i.e., which strings go to which tokens)