Practice Problem

```
<s> ::= A <a> | A <b>
<a> ::= A B
<b> ::= B <b> | B <s>
```

Is the following sentence recognized by the above grammar?

A B B A A B

Answer

```
<s> ::= A <a> | A <b>
<a> ::= A B
<b> ::= B <b> | B <s>
```

A B B A A B

<s>

A <b>

A B <b>

A B B <s>

A B B A <a>

A B B A A B

Parser Generators

Principles of Programming Languages
Lecture 14

CAS CS 320

Outline

Extend our BNF syntax to be a bit more convenient

Introduce **parser generators**

Discuss **lexical analysis**

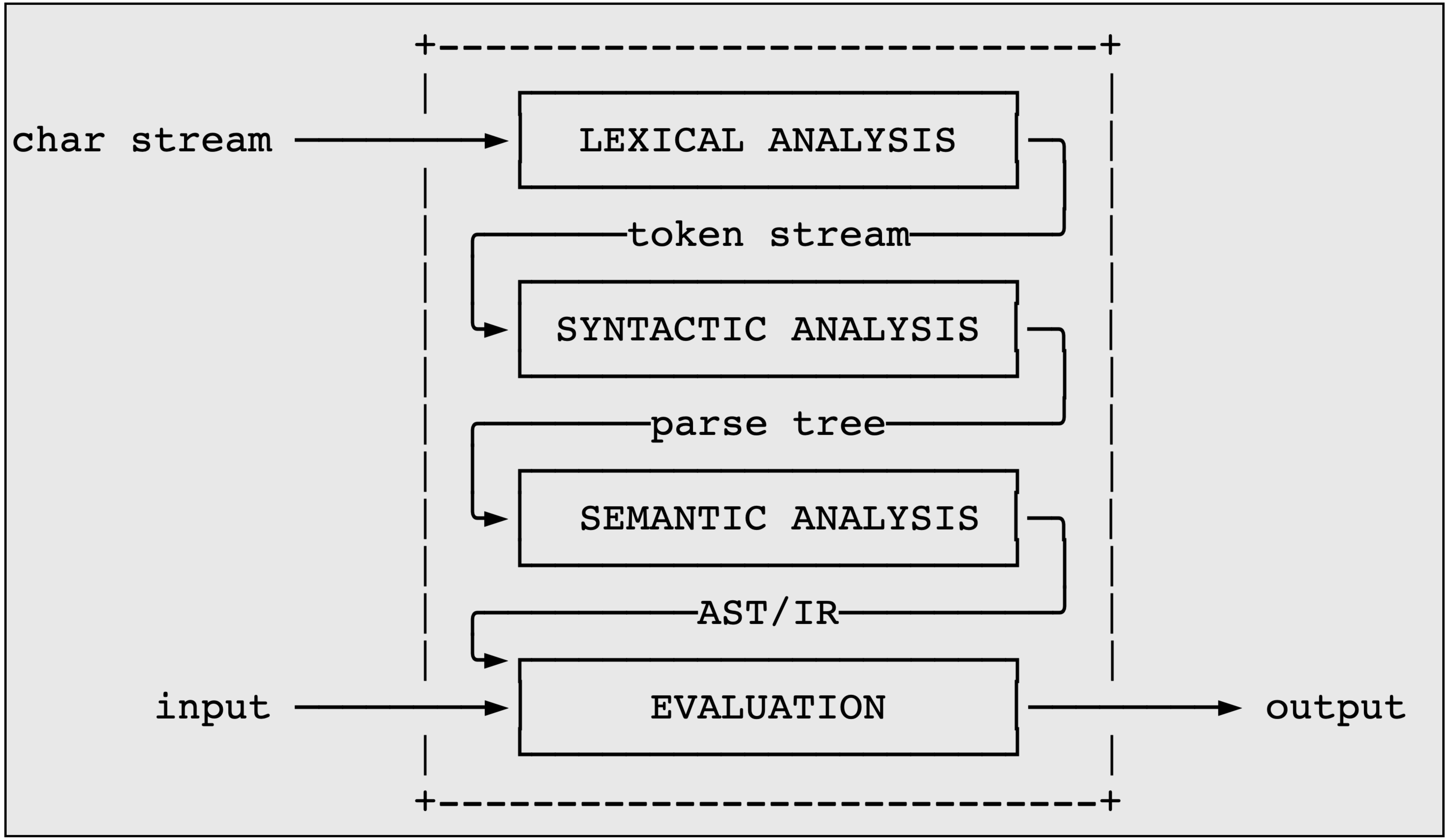
Demo **Menhir**, the parser generator for this course

Learning Objectives

- All the same questions as last time, but for extended BNF
- Describe the difference between lexing and parsing
- Read a regular expression and understand generally what it does
- Build a parser for a grammar using Menhir

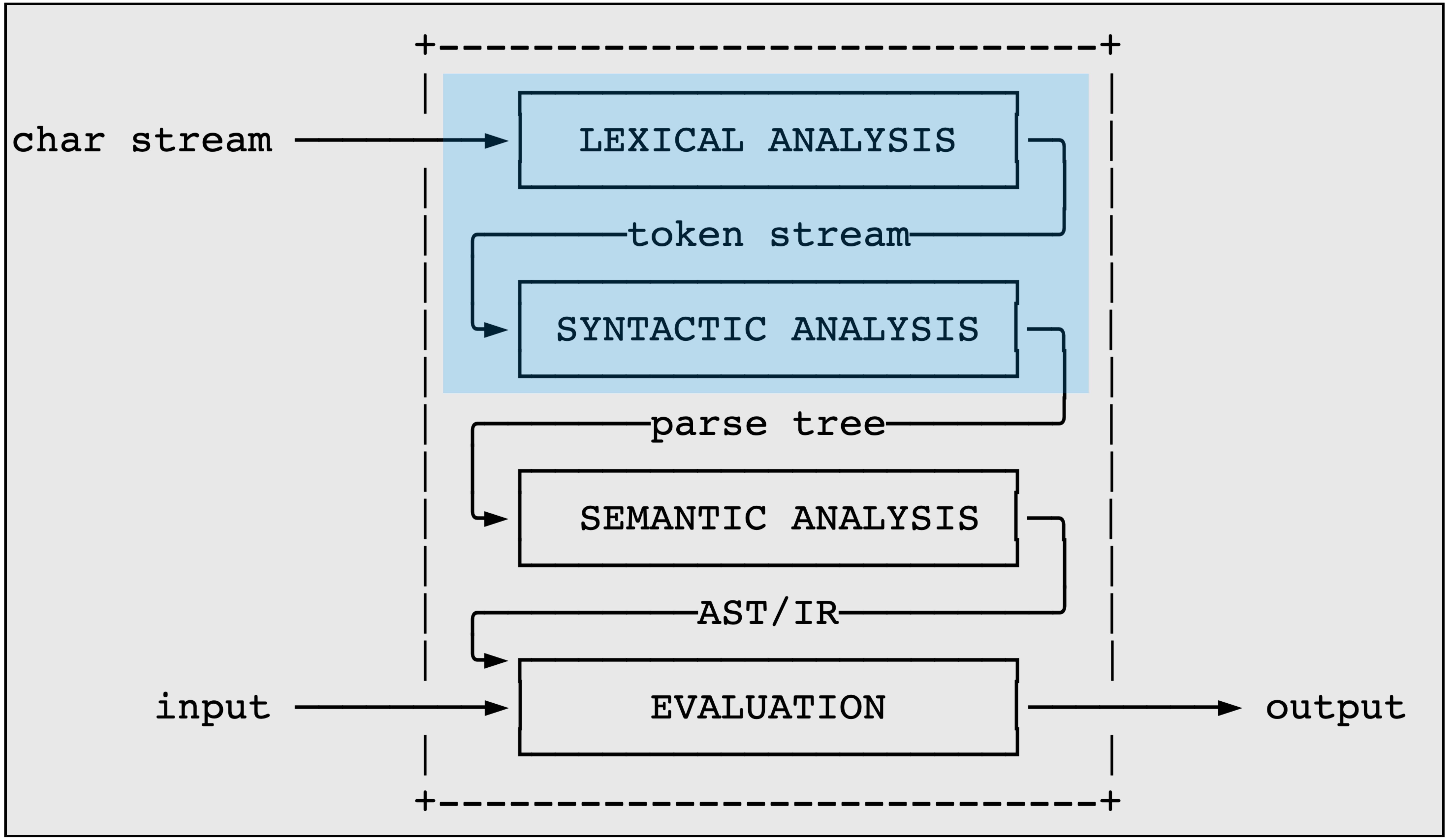
Recap + Motivation

Recall: The Picture



Recall: The Picture

parsing



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)

Recall: Derivations and Parse Trees

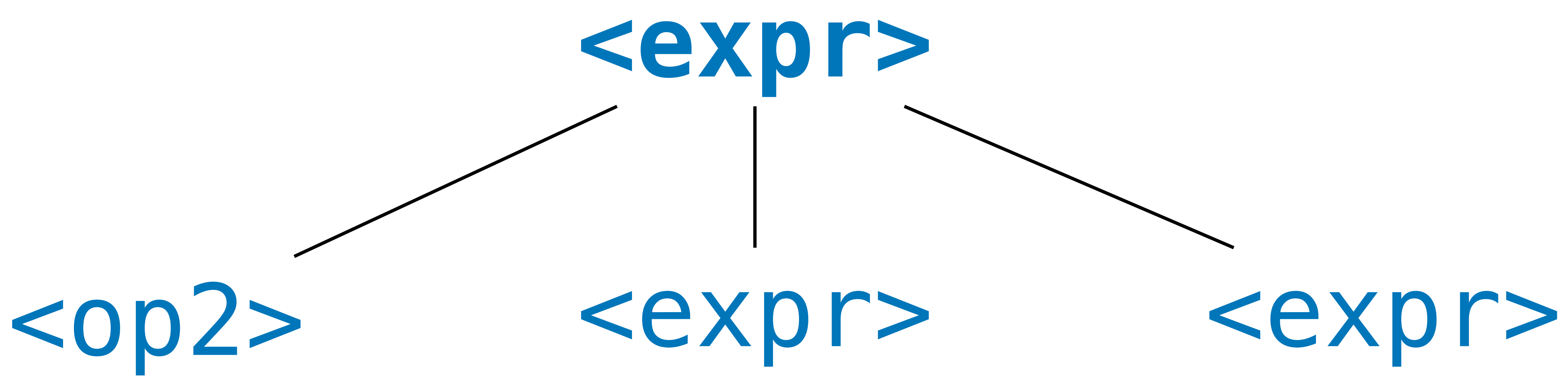
Recall: Derivations and Parse Trees

<expr>

<expr>

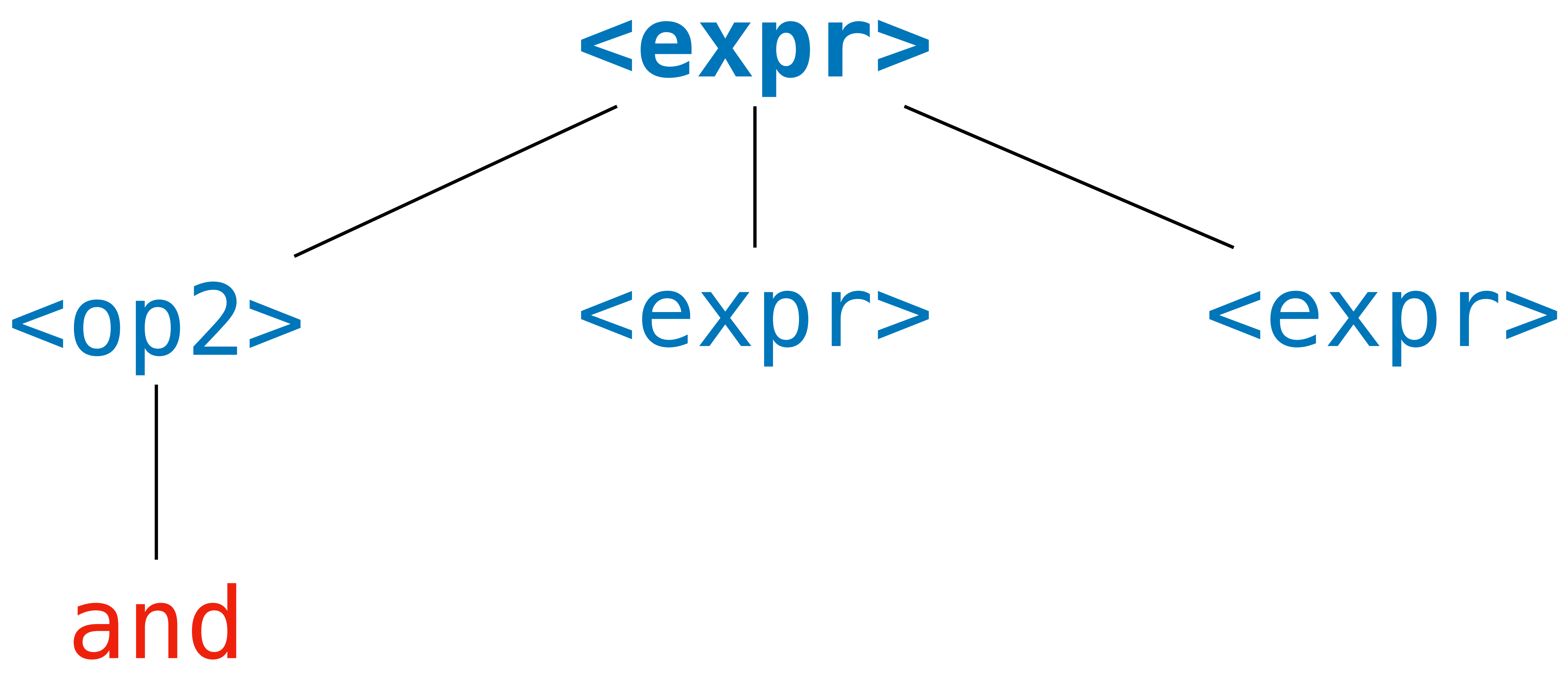
Recall: Derivations and Parse Trees

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



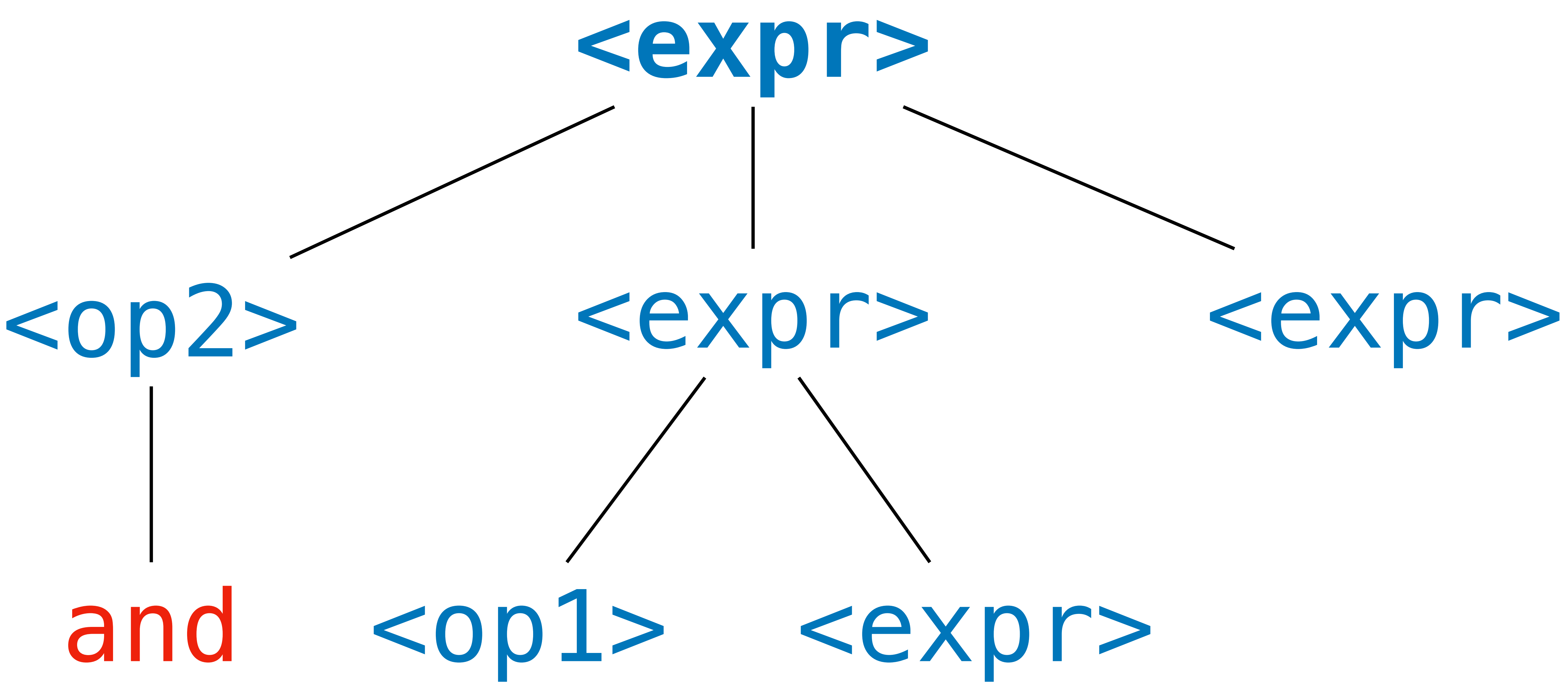
Recall: Derivations and Parse Trees

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



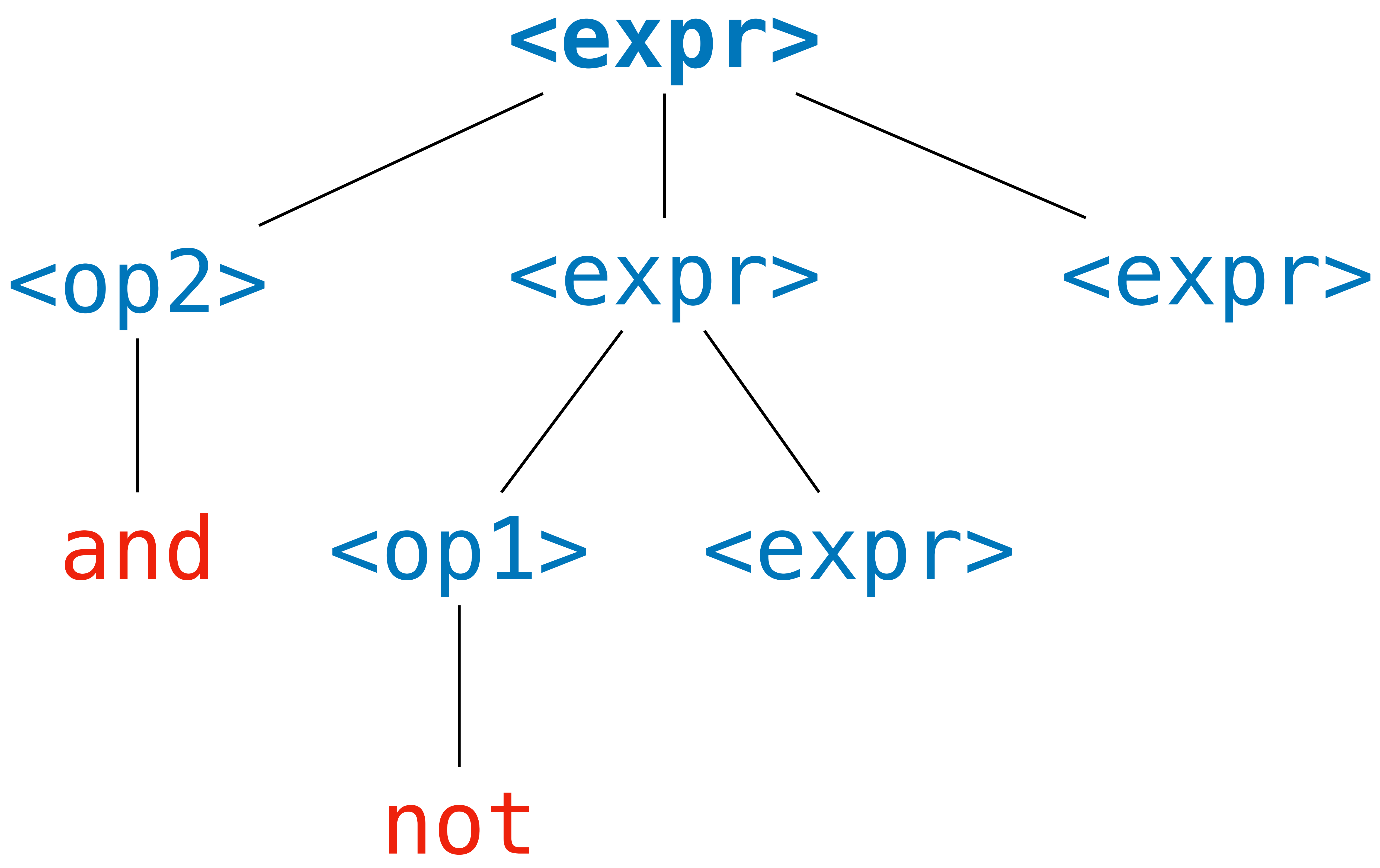
Recall: Derivations and Parse Trees

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



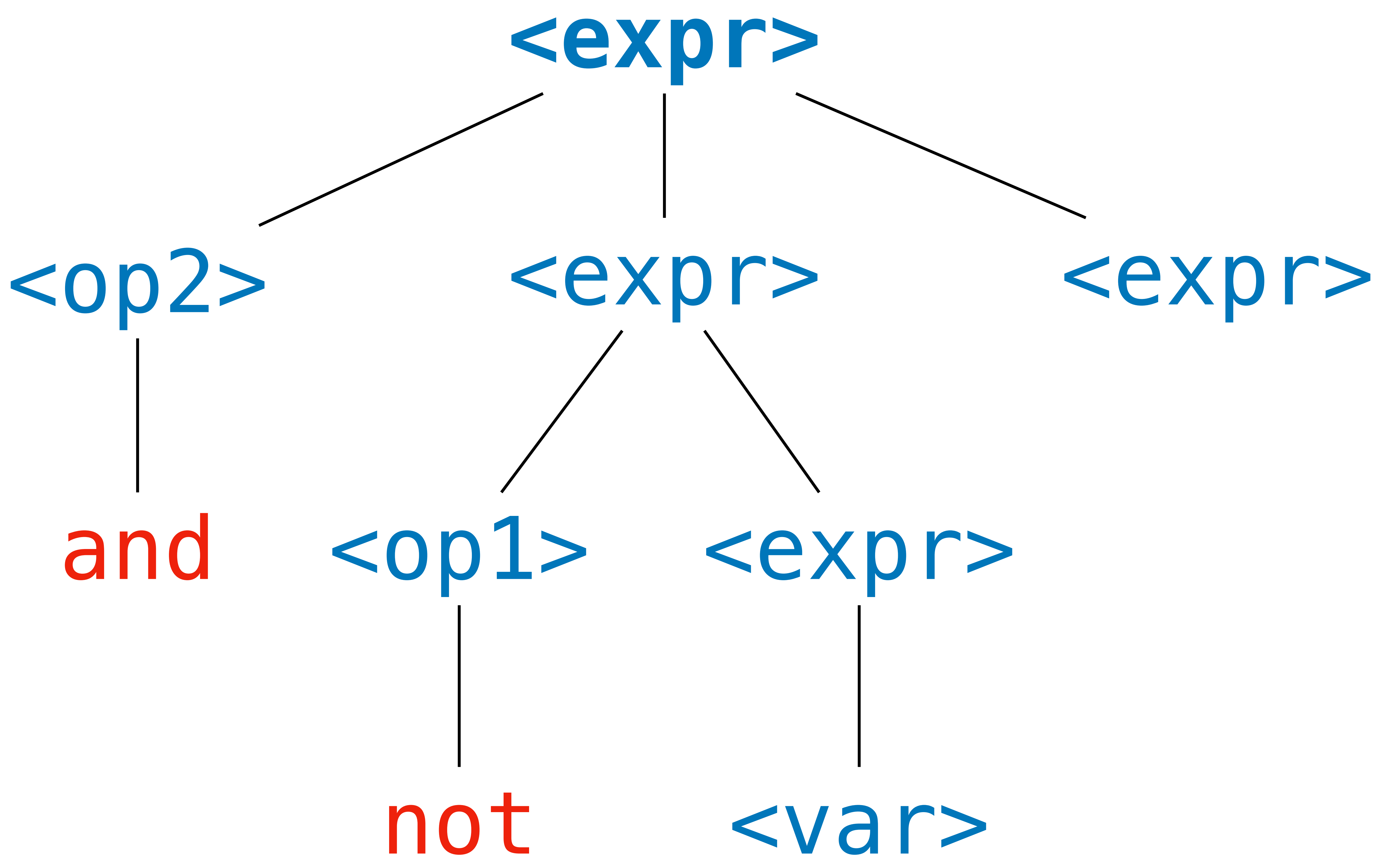
Recall: Derivations and Parse Trees

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



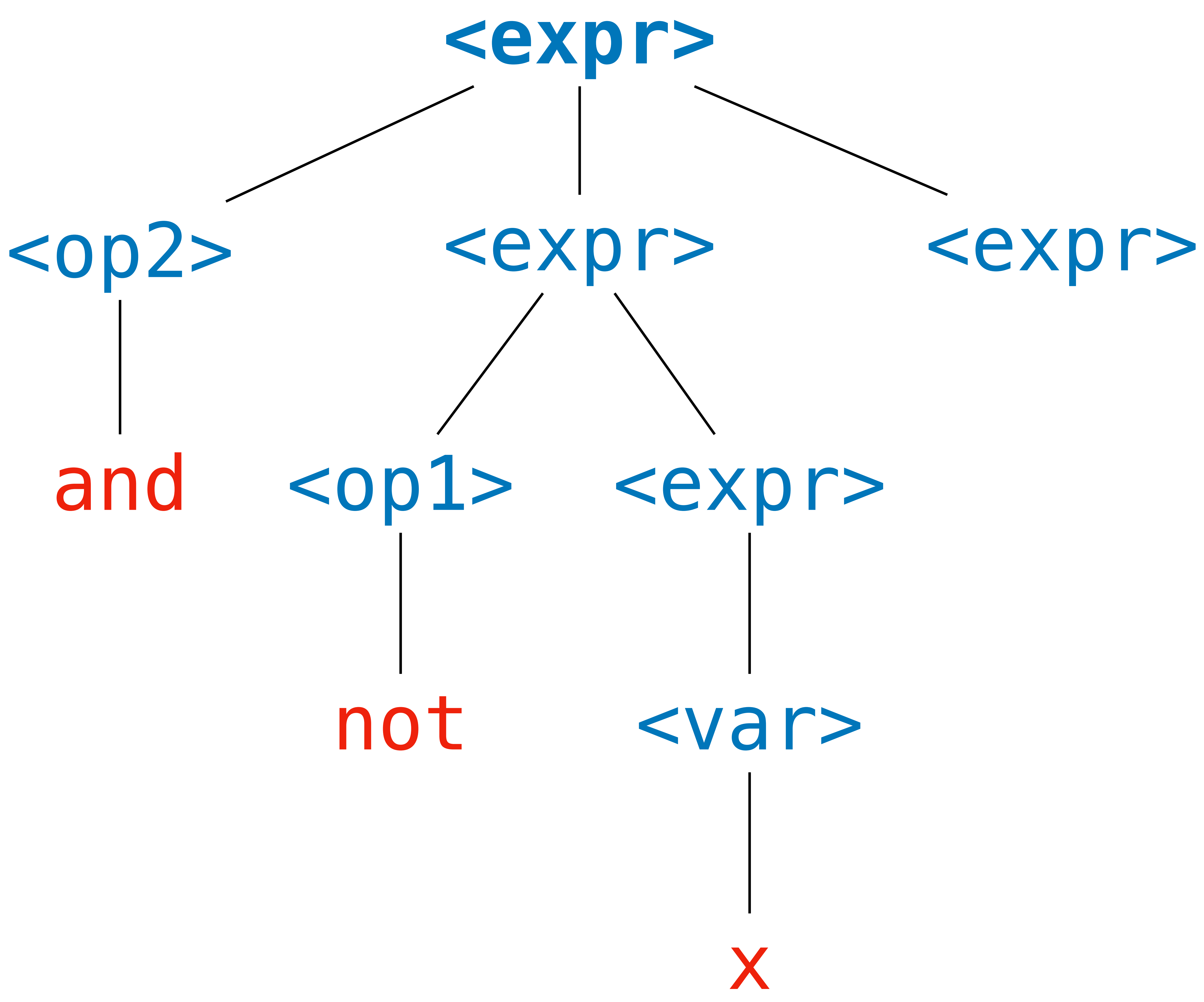
Recall: Derivations and Parse Trees

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



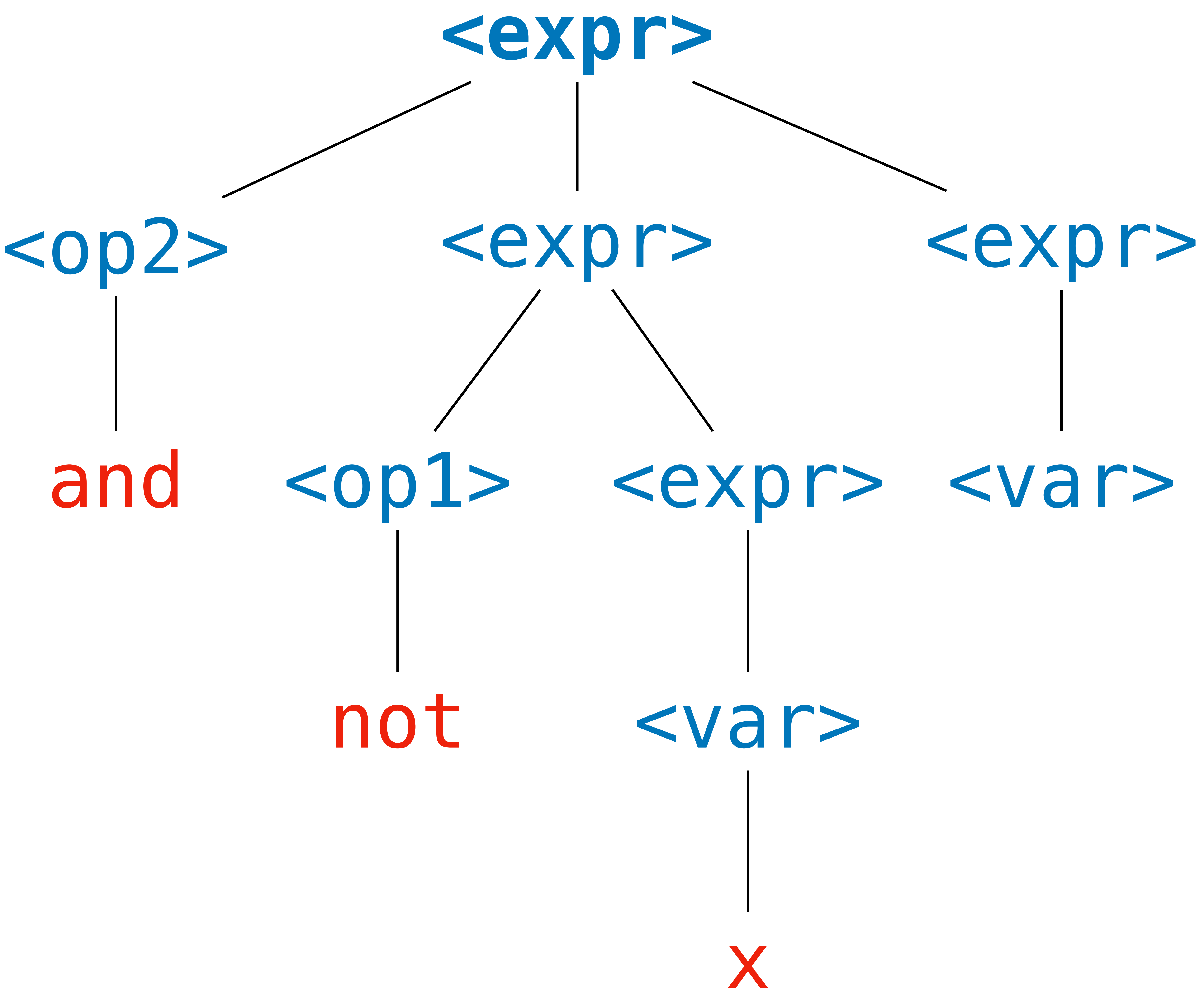
Recall: Derivations and Parse Trees

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



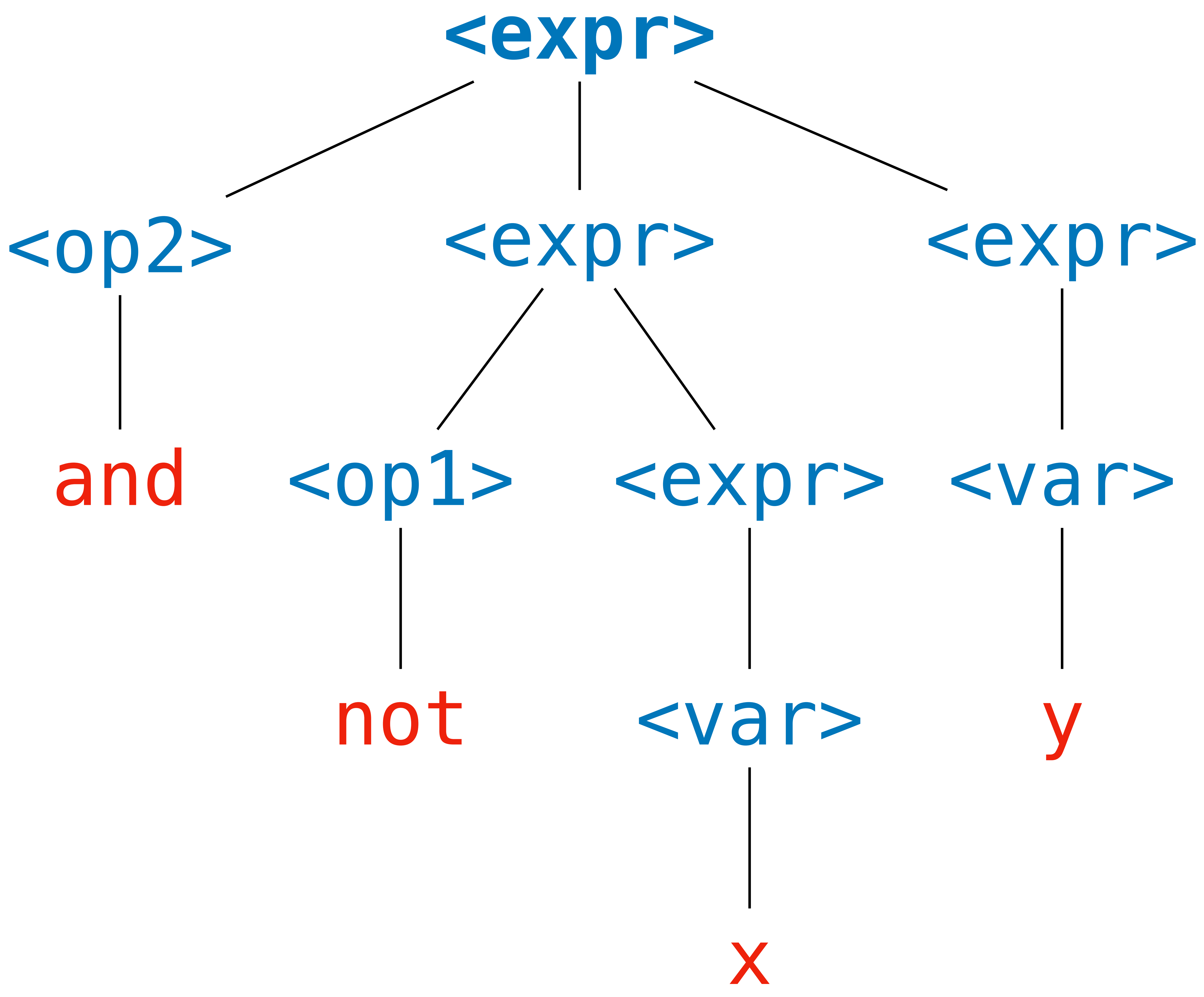
Recall: Derivations and Parse Trees

```
<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>
```



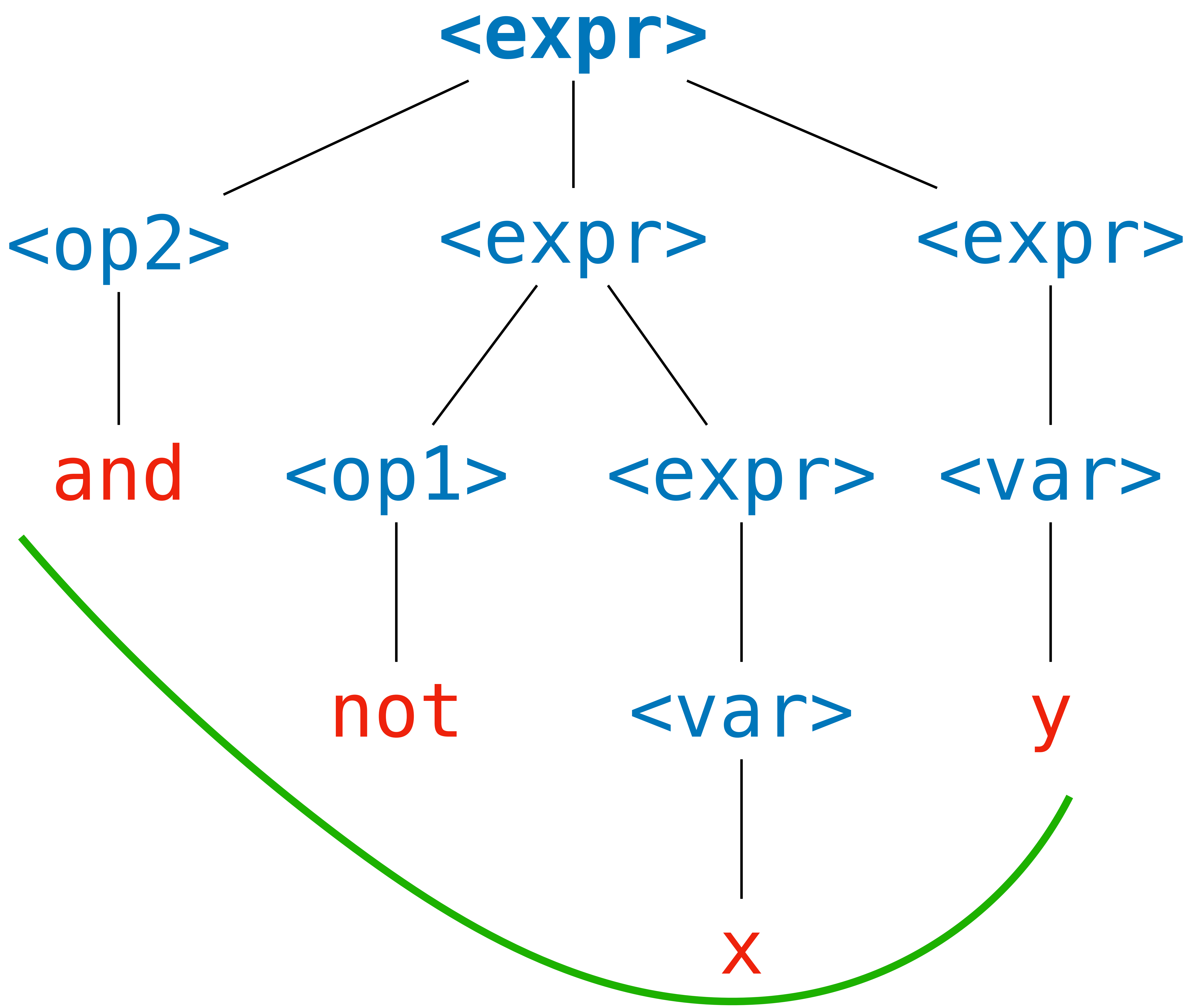
Recall: Derivations and Parse Trees

```
<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
```



Recall: 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
```

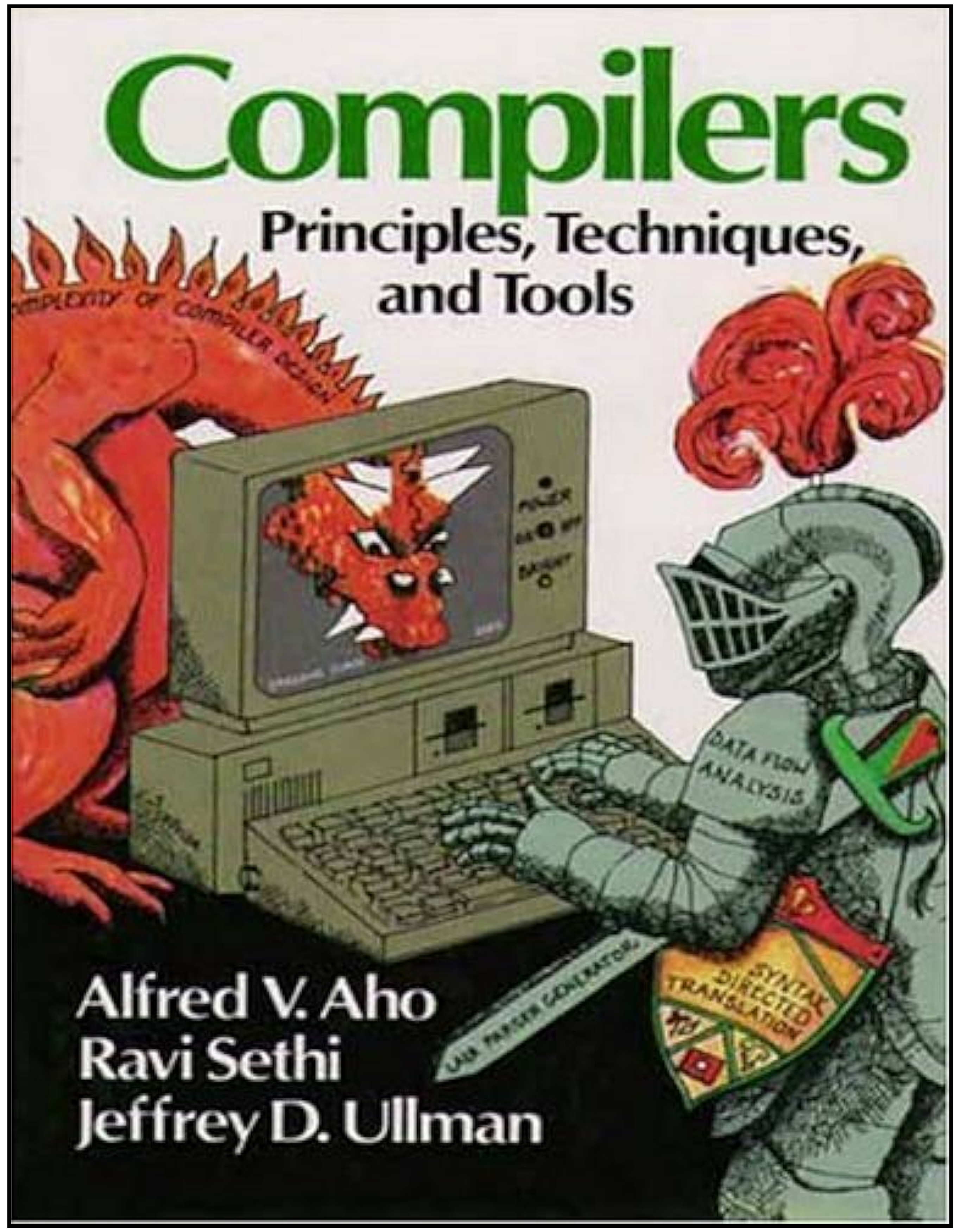


A Note on "History"

Lexical analysis and parsing are typically associated with ***Compiler Design***.

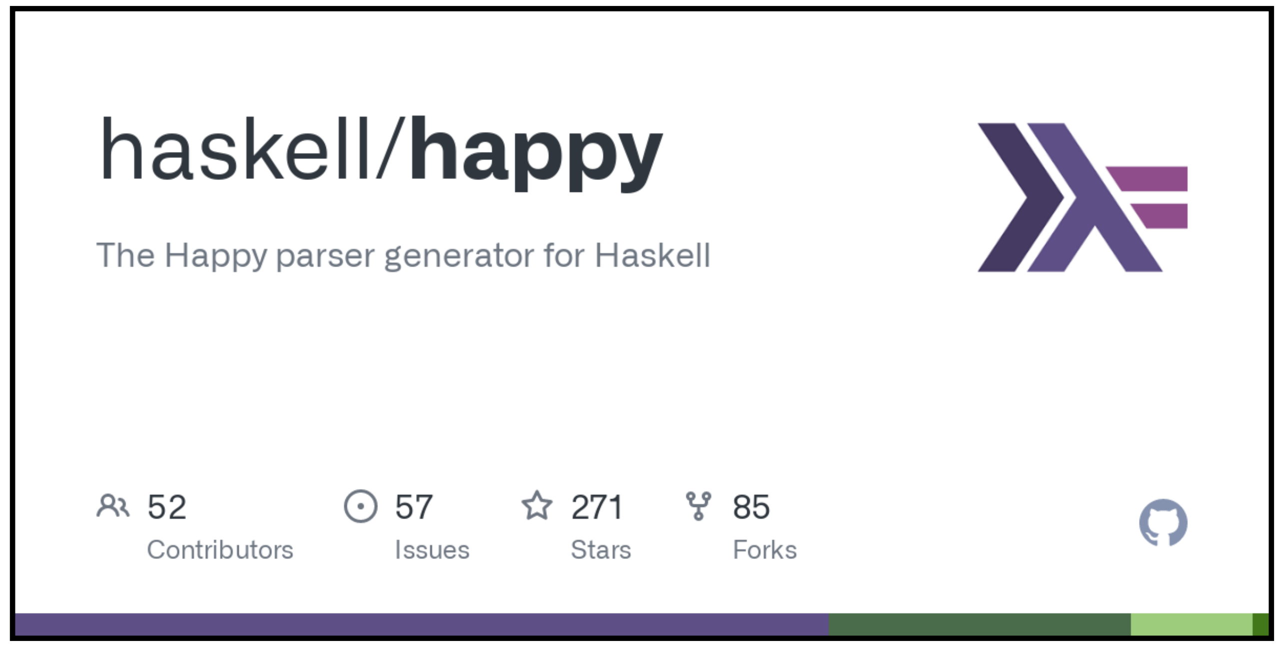
Compiler design was once a fundamental requirement in CS programs. *This is not really the case anymore.*

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



Parser Generators

ANTLR



(Beaver)

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

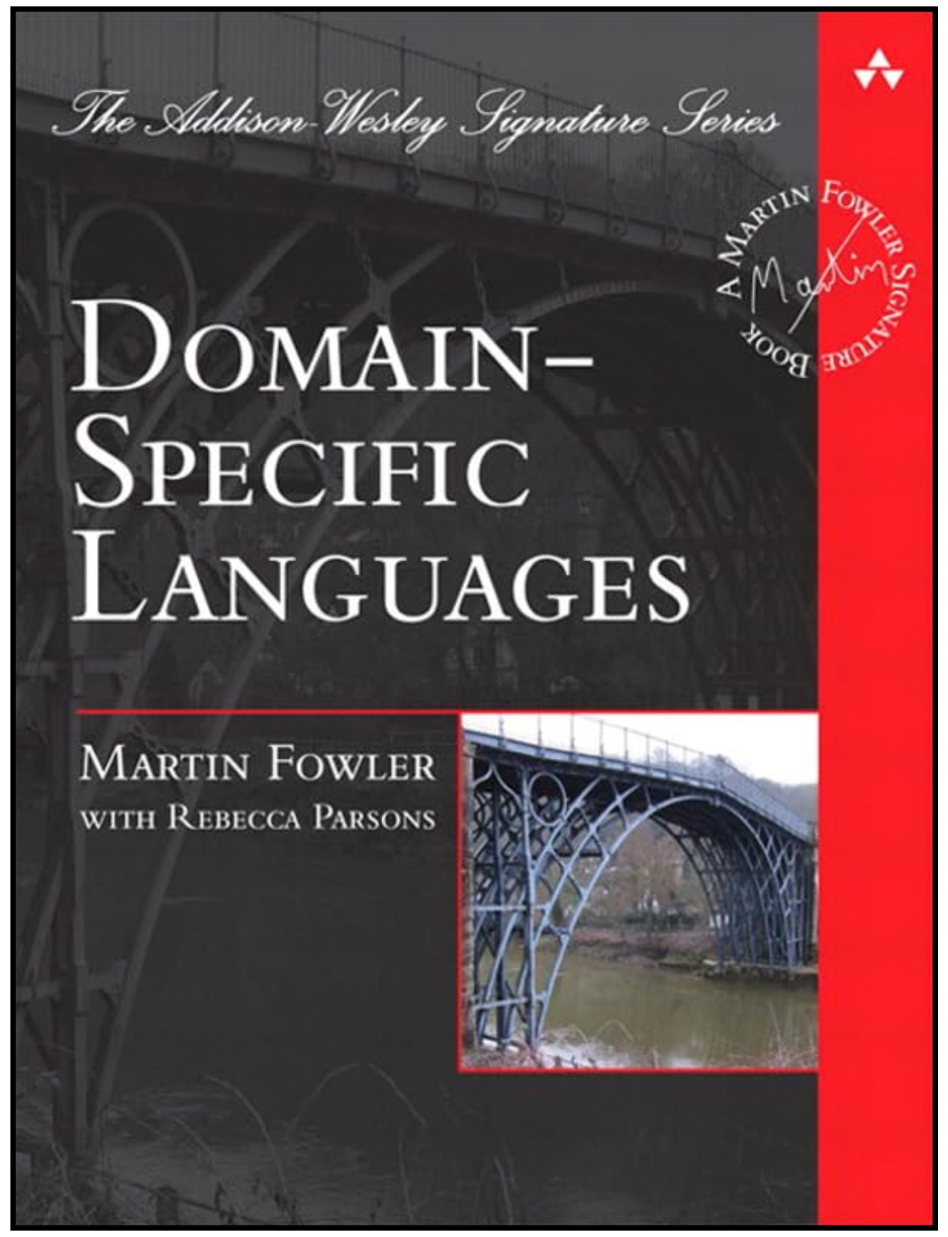
(So there was a point to learning (E)BNF for the "real-world")

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

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:

» 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

<w>
<l> <w>
c <w>
c <l><w>
c a <w>
⋮

BNF:

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

EBNF:

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

Menhir:

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

A Note on EBNF and Derivations/Parse Trees

EBNF syntax is *meta-syntax* it should not appear in a derivation. Any ENBF syntax should be immediately expanded in a derivation or parse tree.

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

```
<expr>
if <expr> then <expr>
⋮
```

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

```
<word>
<letter> <letter> <letter> <letter>
⋮
```

Interlude: Regular Expressions

Regular Grammars

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

```
<nonterminal> ::= terminal

<nonterminal> ::= terminal <nonterminal>

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

Example

<s>
a <s>
a a <s>
a a a <s>
a a a a b <a>
a a a a b c c c

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

Regular Expressions

Regular Expressions

Regular expressions provide a compact way of describing regular grammars

Regular Expressions

<u>Regular expressions</u> provide a compact way of describing regular grammars

- A **terminal symbols** is a regular expression

Regular Expressions

Regular expressions provide a compact way of describing regular grammars

- A **terminal symbols** is a regular expression
- **[t1 t2 ... tk]** is a regular expression describing an any one of the terminal symbols **t1, t2, ..., tk**

Regular Expressions

Regular expressions provide a compact way of describing regular grammars

- A **terminal symbols** is a regular expression
- **[t1 t2 ... tk]** is a regular expression describing an any one of the terminal symbols **t1, t2, ..., tk**
- **(e1 | e2 | ... | ek)** is a regular expression describing any one of the expressions **e1, e2, ..., ek**

Regular Expressions

Regular expressions provide a compact way of describing regular grammars

- A **terminal symbols** is a regular expression
- **[t1 t2 ... tk]** is a regular expression describing an any one of the terminal symbols **t1, t2, ..., tk**
- **(e1 | e2 | ... | ek)** is a regular expression describing any one of the expressions **e1, e2, ..., ek**
- **exp*** is a regular expression describing zero or more occurrences of **exp**

Regular Expressions

Regular expressions provide a compact way of describing regular grammars

- A **terminal symbols** is a regular expression
- **[t1 t2 ... tk]** is a regular expression describing an any one of the terminal symbols **t1, t2, ..., tk**
- **(e1 | e2 | ... | ek)** is a regular expression describing any one of the expressions **e1, e2, ..., ek**
- **exp**∗ is a regular expression describing zero or more occurrences of **exp**
- **exp**+ is a regular expression describing one or more occurrences of **exp**

Regular Expressions

Regular expressions provide a compact way of describing regular grammars

- A **terminal symbols** is a regular expression
- **[t1 t2 ... tk]** is a regular expression describing an any one of the terminal symbols **t1, t2, ..., tk**
- **(e1 | e2 | ... | ek)** is a regular expression describing any one of the expressions **e1, e2, ..., ek**
- **exp**∗ is a regular expression describing zero or more occurrences of **exp**
- **exp**+ is a regular expression describing one or more occurrences of **exp**
- **exp**? is a regular expression describing zero or one occurrences of **exp**

Example

<s> ::= a <s> | b <a>

~~<s> ::= b <a>~~

<a> ::= ϵ | c <a>

~~<a> ::= c <a>~~

is equivalent to

0 or more a

0 or more c's

a*bc*

b

or

```
'a'* 'b' 'c'*
```

in ocamllex syntax

Example: Numbers and Variables

optional negation

lower letter

```
-?[0-9]+            [a-z][a-z0-9A-Z_']+
```

numbers

1 or more digits

variables

1 or more alphanum

1

23

-23

-00100

x

x''

one_two

'''

a___ __

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

Lexical Analysis

The "Lexing" Problem

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

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

The "Lexing" Problem

```
"let" ≈ ['l', 'e', 't'] ↦ LET
"fun" ≈ ['f', 'u', 'n'] ↦ 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

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

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

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

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

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

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.

» 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.

<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.

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

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

Aside: One Token at a time

```
"  let@#_)($#@_J_@O#GKJ"  --next_token-->  (LET, "@#_)($#@_J_@O#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_@0#GKJ"` $\xrightarrow{\texttt{next\_token}}$ `(LET, "@#_)($#@_J_@0#GKJ")`

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

(VAR "le", "x = 2")

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

We'll be building a parser for the this grammar

```
<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' '_' '\'']*
```

let x=2 in x

let x=2 in
let y=3 in
x+y

Operators in order of increasing precedence:

Operator	Associativity
+, -	left
*, /	left

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)