CMPU 240 · Theory of Computation

Context-free Grammars

3 March 2026

“Many of us have an innate predisposition to be curious. I believe that after a traditional education, or working in an environment with many people, curiosity is a decision requiring intent and discipline.”

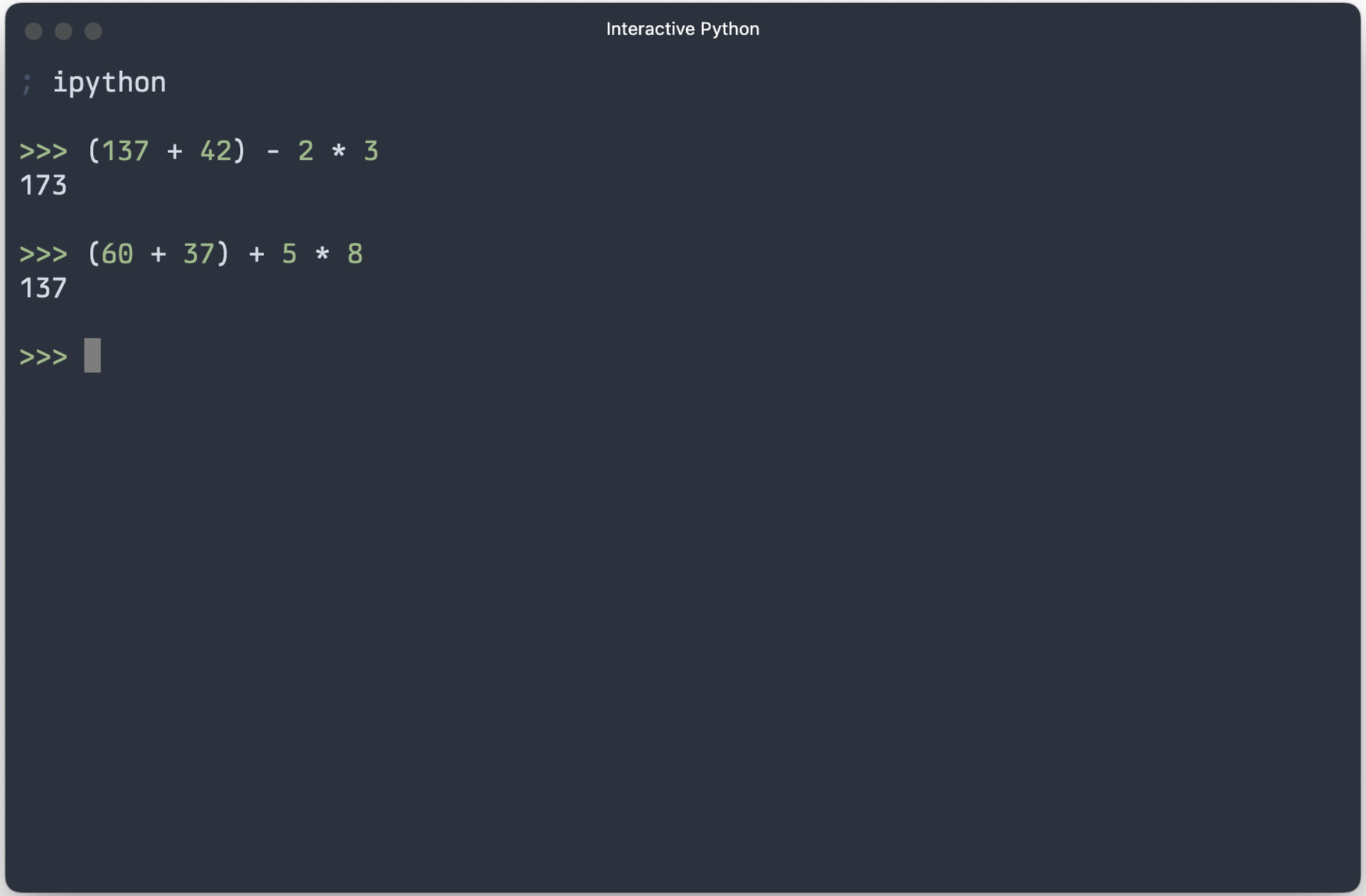
Jony Ive, *The Wall Street Journal Magazine*, 4 October 2021


Interactive Python
; ipython
>>>


Interactive Python
; ipython
>>> (137 + 42) - 2 * 3


Interactive Python
; ipython
>>> (137 + 42) - 2 * 3
173
>>>


Interactive Python
; ipython
>>> (137 + 42) - 2 * 3
173
>>> (60 + 37) + 5 * 8


Interactive Python
; ipython
>>> (137 + 42) - 2 * 3
173
>>> (60 + 37) + 5 * 8
137
>>>

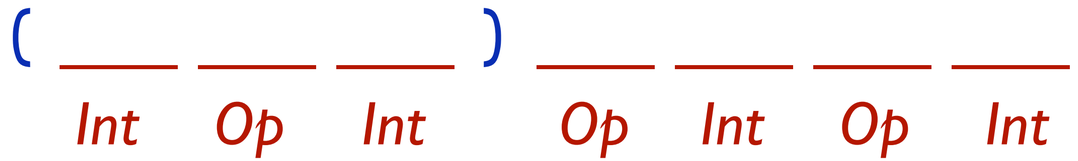

Interactive Python
; ipython
>>> (137 + 42) - 2 * 3
173
>>> (60 + 37) + 5 * 8
137
>>> (200 / 2) + 6 / 2


Interactive Python
; ipython
>>> (137 + 42) - 2 * 3
173
>>> (60 + 37) + 5 * 8
137
>>> (200 / 2) + 6 / 2
103.0
>>>

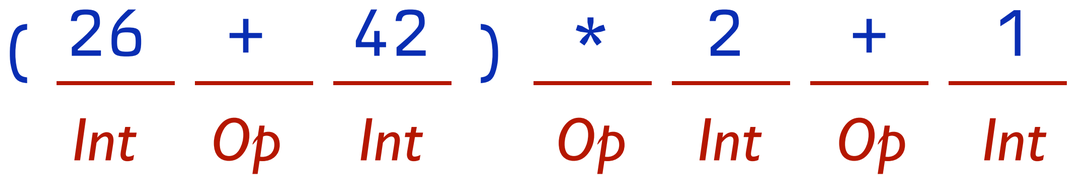
Thousands of TREES (PLURAL NOUN) ago, there were calendars that enabled the ancient COMPUTERS (PLURAL NOUN) to divide a year into twelve COOKIES (PLURAL NOUN), each month into 25 (NUMBER) weeks, and each week into seven TOMATOES (PLURAL NOUN). At first, people told time by a sun clock, sometimes known as the CHOCOLATE (NOUN) dial. Ultimately, they invented the great timekeeping devices of today, such as the grandfather PUMPKIN (NOUN), the pocket CHEESE (NOUN), the alarm MILK (NOUN), and, of course, the EYE (PART OF BODY) watch. Children learn about clocks and time almost before they learn their AB M (LETTER OF THE ALPHABET)'s. They are taught that a day consists of 24 KIDS (PLURAL NOUN), an hour has 60 BATS (PLURAL NOUN), and a minute has 60 WITCHES (PLURAL NOUN). By the time they are in kindergarten, they know if the big TOE (PART OF THE BODY) is at twelve and the little NOSE (PART OF THE BODY) is at three, that it is 13 (NUMBER) o'clock. I wish we could continue this

In Mad Libs, you have blanks that you fill with an indicated type of word.

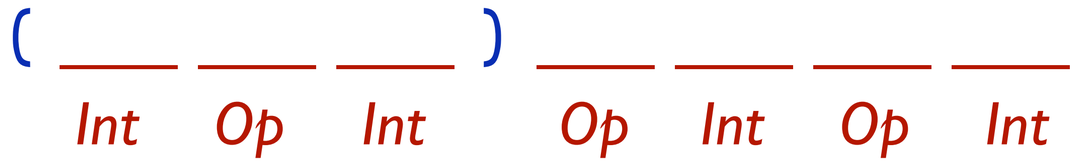
We can describe languages in a similar way:



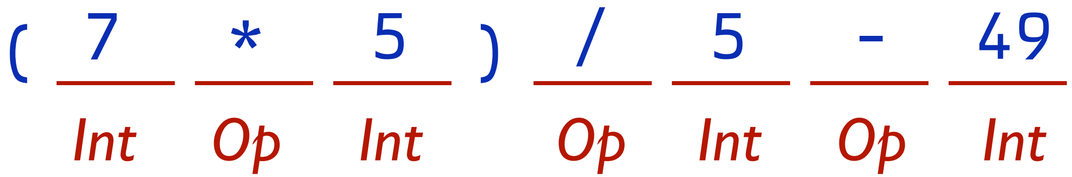
We can describe languages in a similar way:



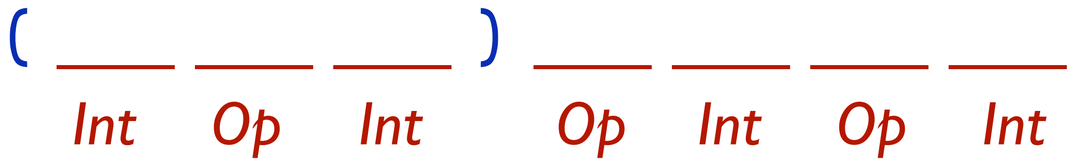
We can describe languages in a similar way:



We can describe languages in a similar way:

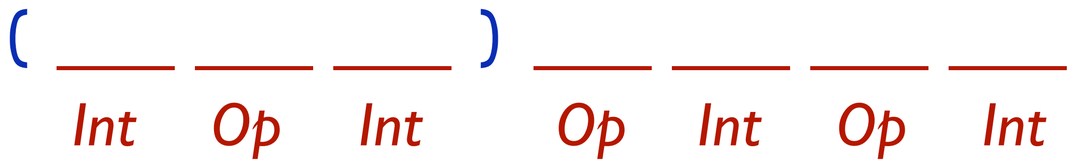


If you had a computer pre-programmed with a template like this, then you could enter a string and check whether it's valid.



You can also understand what individual pieces of the string mean based on which part of the template they're filling in!

But this only lets us make one form of arithmetic expression. What about others?



Recursive Mad Libs

Expr

Recursive Mad Libs

Expr

What can an arithmetic expression be?

Recursive Mad Libs

`int`

Expr

What can an arithmetic expression be?

`int`	A single number

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

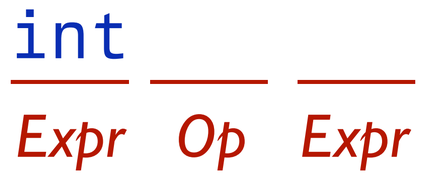
Recursive Mad Libs

Expr *Op* *Expr*

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

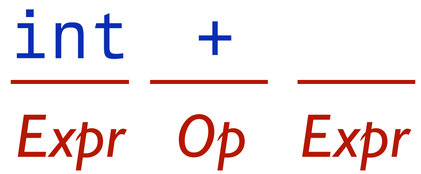
Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs

Expr *Op* *Expr*

What can an arithmetic expression be?

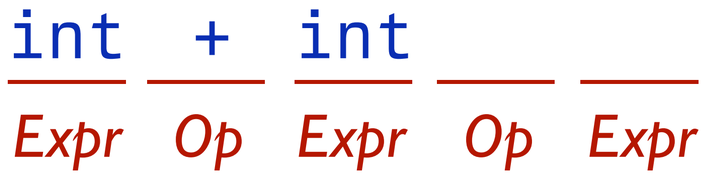
`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

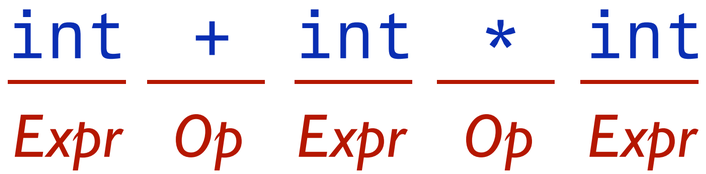
Recursive Mad Libs

`int`	`+`	`int`	`*`	___
Expr	*Op*	*Expr*	*Op*	*Expr*

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

(____)

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

Expr *Op* *Expr*

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

(____ ____ ____)

Expr *Op* *Expr*

What can an arithmetic expression be?

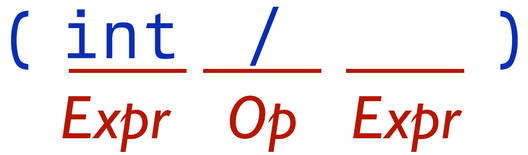
`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

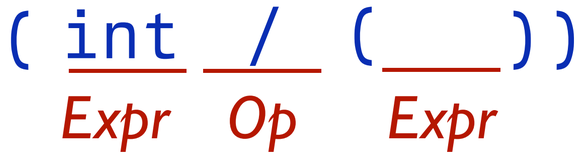
(____)

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

Expr

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

Expr *Op* *Expr*

What can an arithmetic expression be?

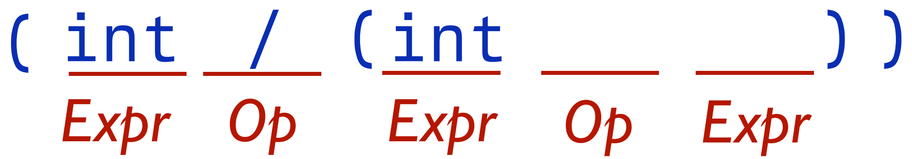
`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

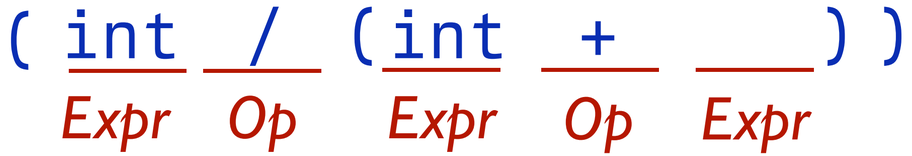
Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs



What can an arithmetic expression be?

`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Recursive Mad Libs

```
( int  /  (int  +  int) )
  Expr Op   Expr Op  Expr
```

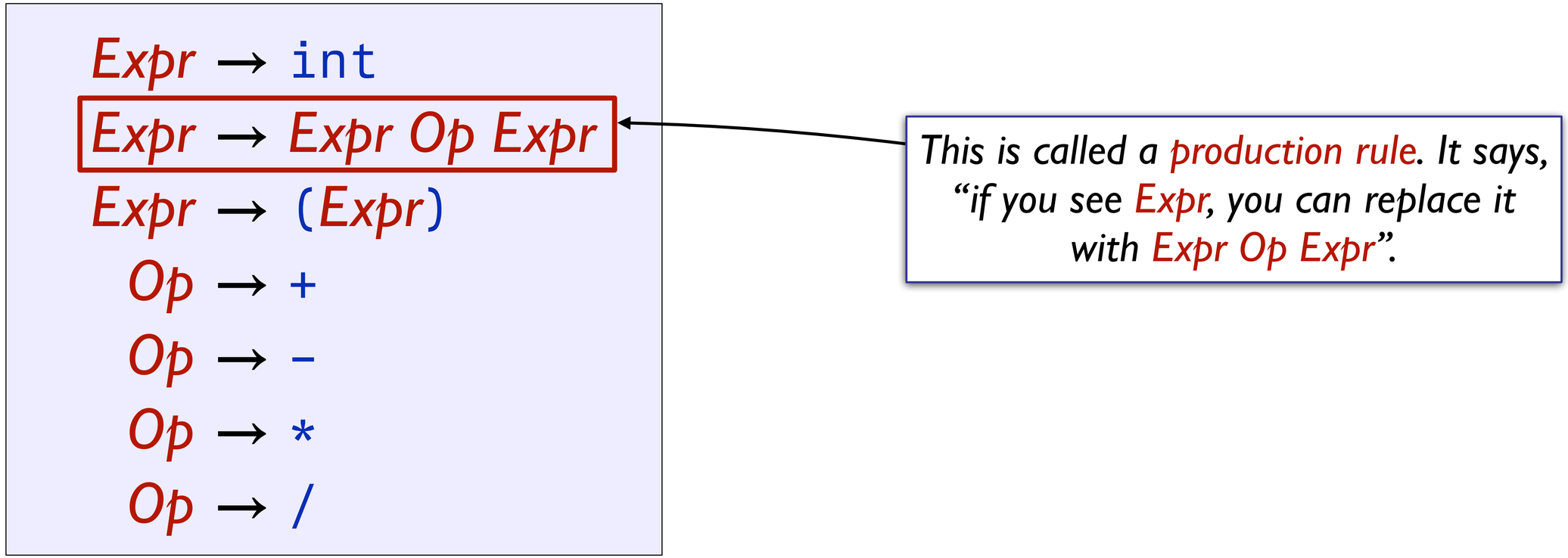
What can an arithmetic expression be?

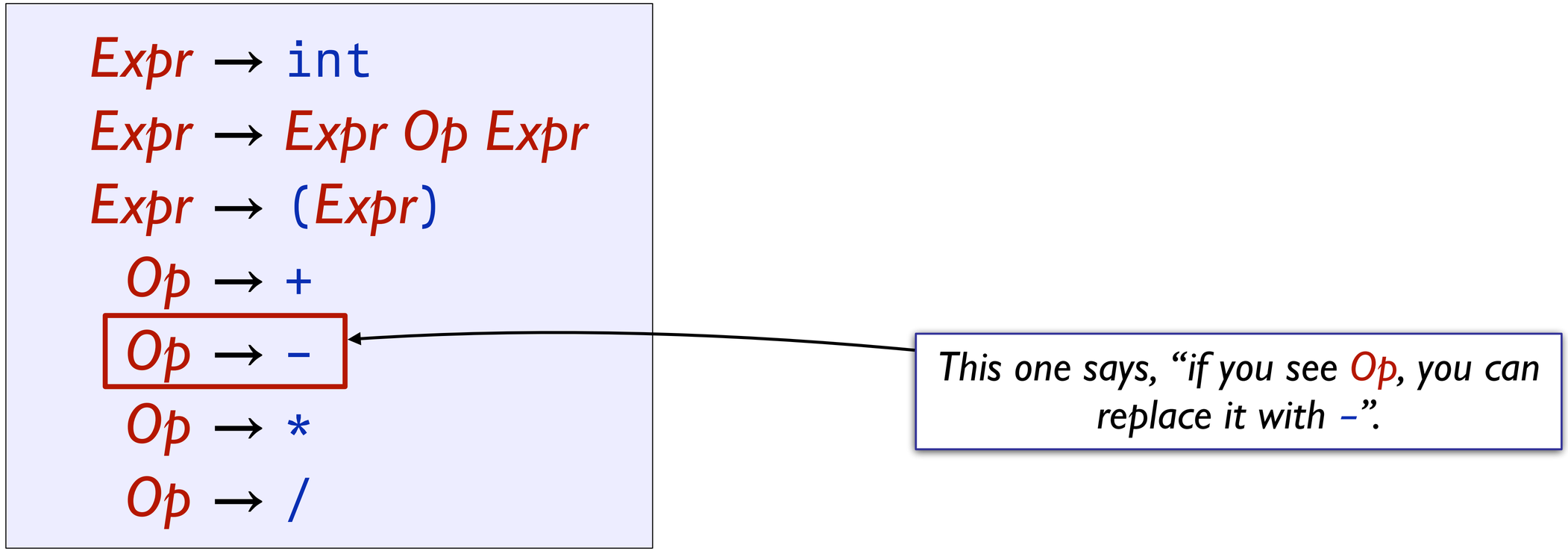
`int`	A single number
Expr Op Expr	Two expressions joined by an operator
(*Expr*)	A parenthesized expression

Context-free grammars

A *context-free grammar* (CFG) is a way of recursively describing how to form the strings in a language.

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \\
\mathit{Expr} &\rightarrow \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\mathit{Expr} &\rightarrow \texttt{(}\mathit{Expr}\texttt{)} \\
\mathit{Op} &\rightarrow \texttt{+} \\
\mathit{Op} &\rightarrow \texttt{-} \\
\mathit{Op} &\rightarrow \texttt{*} \\
\mathit{Op} &\rightarrow \texttt{/}
\end{aligned}
$$


Expr → int
Expr → Expr Op Expr
Expr → (Expr)
Op → +
Op → -
Op → *
Op → /
This is called a production rule. It says, "if you see Expr, you can replace it with Expr Op Expr".


Expr → int
Expr → Expr Op Expr
Expr → (Expr)
Op → +
Op → -
Op → *
Op → /
This one says, "if you see Op, you can replace it with -".

Expr → `int`
Expr → *Expr Op Expr*
Expr → `(`*Expr*`)`
Op → `+`
Op → `-`
Op → `*`
Op → `/`

Expr

These red symbols are variables. They're placeholders that get expanded.

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \\
\mathit{Expr} &\rightarrow \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\mathit{Expr} &\rightarrow \texttt{(}\mathit{Expr}\texttt{)} \\
\mathit{Op} &\rightarrow \texttt{+} \\
\mathit{Op} &\rightarrow \texttt{-} \\
\mathit{Op} &\rightarrow \texttt{*} \\
\mathit{Op} &\rightarrow \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Expr} \\
\Rightarrow\ &\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \\
\mathit{Expr} &\rightarrow \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\mathit{Expr} &\rightarrow \texttt{(}\mathit{Expr}\texttt{)} \\
\mathit{Op} &\rightarrow \texttt{+} \\
\mathit{Op} &\rightarrow \texttt{-} \\
\mathit{Op} &\rightarrow \texttt{*} \\
\mathit{Op} &\rightarrow \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Expr} \\
\Rightarrow\ &\mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\Rightarrow\ &\mathit{Expr}\ \mathit{Op}\ \texttt{(}\mathit{Expr}\texttt{)}
\end{aligned}
$$

The symbols in blue monospace are terminals. They're the final characters used in the string and never get replaced.

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \\
\mathit{Expr} &\rightarrow \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\mathit{Expr} &\rightarrow \texttt{(}\mathit{Expr}\texttt{)} \\
\mathit{Op} &\rightarrow \texttt{+} \\
\mathit{Op} &\rightarrow \texttt{-} \\
\mathit{Op} &\rightarrow \texttt{*} \\
\mathit{Op} &\rightarrow \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ \texttt{int} \\
\Rightarrow\ & \texttt{int}\ \mathit{Op}\ \texttt{int} \\
\Rightarrow\ & \texttt{int / int}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \\
\mathit{Expr} &\rightarrow \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\mathit{Expr} &\rightarrow \texttt{(}\mathit{Expr}\texttt{)} \\
\mathit{Op} &\rightarrow \texttt{+} \\
\mathit{Op} &\rightarrow \texttt{-} \\
\mathit{Op} &\rightarrow \texttt{*} \\
\mathit{Op} &\rightarrow \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ \texttt{(}\mathit{Expr}\texttt{)} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ \texttt{(}\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}\texttt{)} \\
\Rightarrow\ & \mathit{Expr}\ \texttt{*}\ \texttt{(}\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}\texttt{)} \\
\Rightarrow\ & \texttt{int}\ \texttt{*}\ \texttt{(}\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}\texttt{)} \\
\Rightarrow\ & \texttt{int}\ \texttt{*}\ \texttt{(int}\ \mathit{Op}\ \mathit{Expr}\texttt{)} \\
\Rightarrow\ & \texttt{int}\ \texttt{*}\ \texttt{(int}\ \mathit{Op}\ \texttt{int)} \\
\Rightarrow\ & \texttt{int}\ \texttt{*}\ \texttt{(int + int)}
\end{aligned}
$$

Formally, a context-free grammar G is a tuple of four items, (V, Σ, P, S):

V is a set of *variables*,

Σ is a set of *terminal symbols*,

P is a set of *production rules* of the form *head* $\rightarrow$ *body*, where

the left-hand side, *head*, is a variable and
the right-hand side, *body*, is a string of zero or more terminals and/or variables that can be substituted for the head

$S \in V$ is the *start symbol* (which must be a variable) that begins the derivation.

Conventionally, the start symbol is the head of the first production rule.

To keep things distinct, in the slides I'll write

Variables like this: A, B, C, D

Terminals like this: a, b, c, d

Arbitrary strings of terminals and variables: α, γ, ω

You don't need to color-code your assignments, but it helps to otherwise follow these conventions.

A notational shorthand

Expr → `int`
Expr → *Expr Op Expr*
Expr → `(`*Expr*`)`
Op → `+`
Op → `-`
Op → `*`
Op → `/`

Expr → `int` | *Expr Op Expr* | `(`*Expr*`)`
Op → `+` | `-` | `*` | `/`

Derivations and parse trees

A *derivation* is a sequence of zero or more steps where variables are replaced by the right-hand side of a production rule.

We write $\alpha \overset{*}{\Rightarrow} \beta$ if string α derives string β by zero or more replacements.

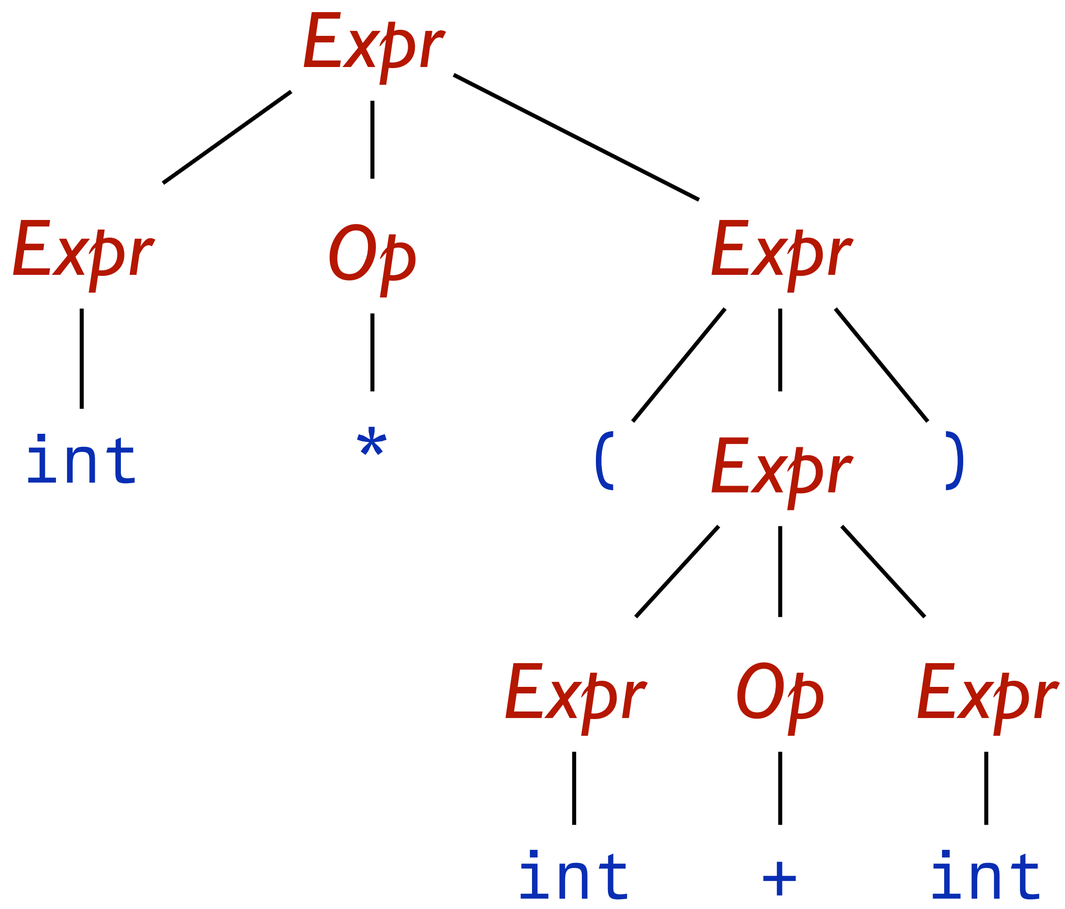
Remember: We use Greek letters to stand for strings of terminals and variables.

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \mid \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid (\mathit{Expr}) \\
\mathit{Op} &\rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ (\mathit{Expr}) \\
\Rightarrow\ & \mathit{Expr}\ \mathit{Op}\ (\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \mathit{Expr}\ \texttt{*}\ (\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \texttt{int}\ \texttt{*}\ (\mathit{Expr}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \texttt{int}\ \texttt{*}\ (\texttt{int}\ \mathit{Op}\ \mathit{Expr}) \\
\Rightarrow\ & \texttt{int}\ \texttt{*}\ (\texttt{int}\ \mathit{Op}\ \texttt{int}) \\
\Rightarrow\ & \texttt{int}\ \texttt{*}\ (\texttt{int}\ \texttt{+}\ \texttt{int})
\end{aligned}
$$

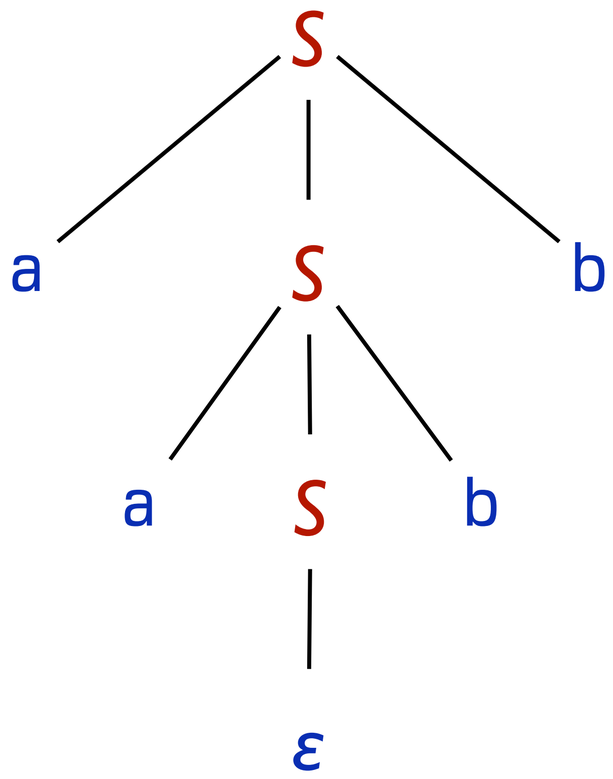
$$
\mathit{Expr} \stackrel{*}{\Rightarrow} \texttt{int}\ \texttt{*}\ (\texttt{int}\ \texttt{+}\ \texttt{int})
$$

An alternative way to write a derivation is as a *parse tree*.



$S \rightarrow aSb \mid \varepsilon$

$S \rightarrow aSb \mid \varepsilon$



Note that we draw a leaf node for ε, even though ε isn't in the set of terminals.

The language of a grammar

If G is a CFG with alphabet Σ and start symbol S, then the *language of G* is

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow}_G w\}.$$

That is, $L(G)$ is the set of strings of terminals derivable from the start symbol.

By $\overset{*}{\Rightarrow}_G$ we're making it explicit that the derivation is using the rules from grammar G.

If G is a CFG with terminals Σ and start symbol S, then the language of G is $L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow}_G w\}$.

Consider the following CFG G over $\Sigma = \{a, b, c, d\}$:

$S \to Sa \mid dT$

$T \to bTb \mid c$

Which of the following strings are in $L(G)$?

dca

dc

cad

bcb

d*T*aa

Given a context-free grammar, the problem of *parsing* a string is to find a parse tree for that string (if one exists).

A natural language example

Natural language processing

By building context-free grammars for human languages and applying statistical inference, it's possible for a computer to recover the likely meaning of a sentence.

In fact, CFGs were first called *phrase-structure grammars* and were introduced by Noam Chomsky in his book *Syntactic Structures* (1957).

Context-free grammars allow natural recursion among things like *nouns*, *verbs*, *prepositions*, and their phrases.

Noun phrases can appear inside verb phrases and prepositional phrases, and vice versa.

```
NP The cat
   PP in
      NP the hat
         PP with
            NP a feather ...
```

S → *NP VP*

NP → *Pronoun* | *ProperNoun* | *Det Nominal*

Nominal → *Nominal Noun* | *Noun*

VP → *Verb* | *Verb NP* | *Verb NP PP* | *Verb PP*

PP → *Preposition NP*

Pronoun → `I` | `you` | `he` | `she` | `they` | ...

ProperNoun → `Patti` | `Tina` | `Yoko` | `Aretha` | ...

Det → `the` | `a` | `an` | *ProperNoun* `'s` | ...

Noun → `cat` | `dog` | `computer` | ...

...

Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo

FUN FACT The English word *buffalo* has two meanings:

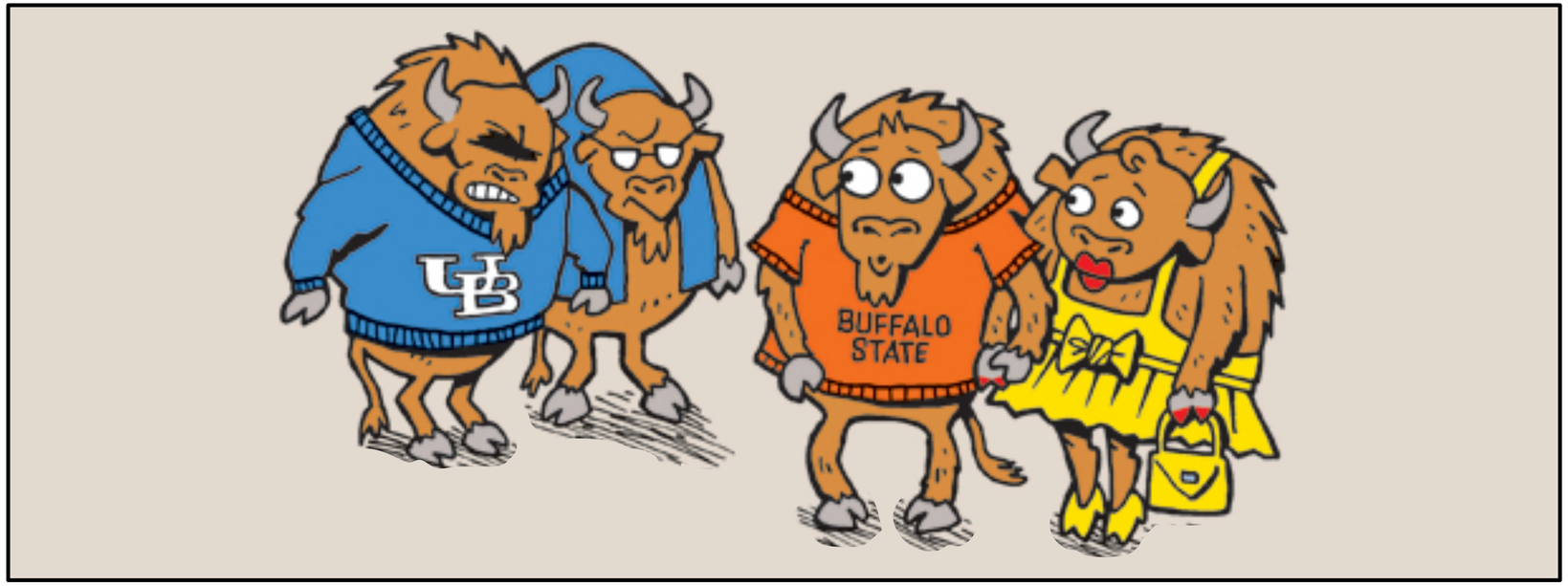
as a *noun*: the name of several species of oxen

as a *verb*: to overpower, overawe, or constrain by superior force or influence; to outwit, perplex

PLURAL FUN Also, the plural of the noun *buffalo* is *buffalo*.

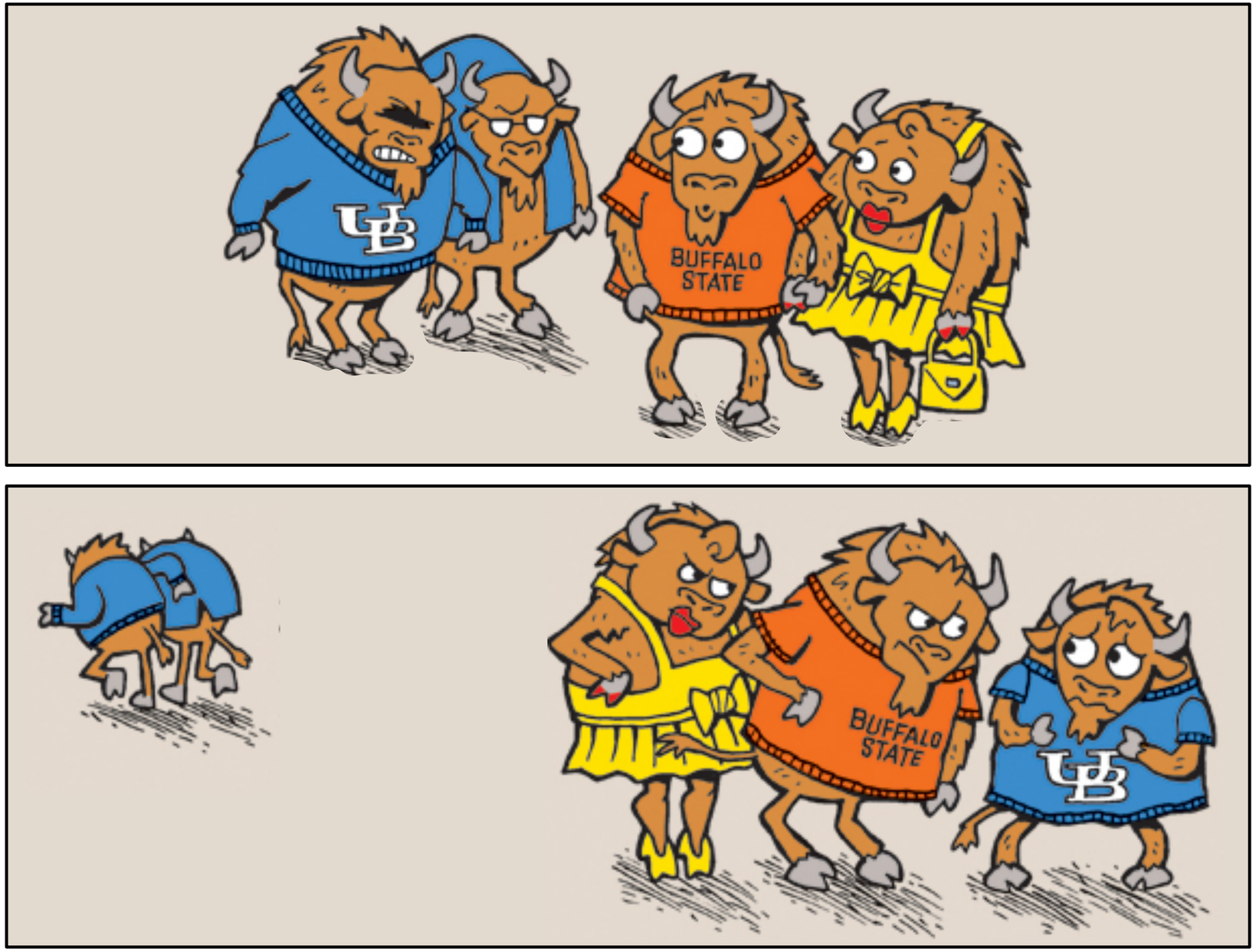
UPSTATE FUN There's also a city in New York named *Buffalo*.

Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo



Greg Williams

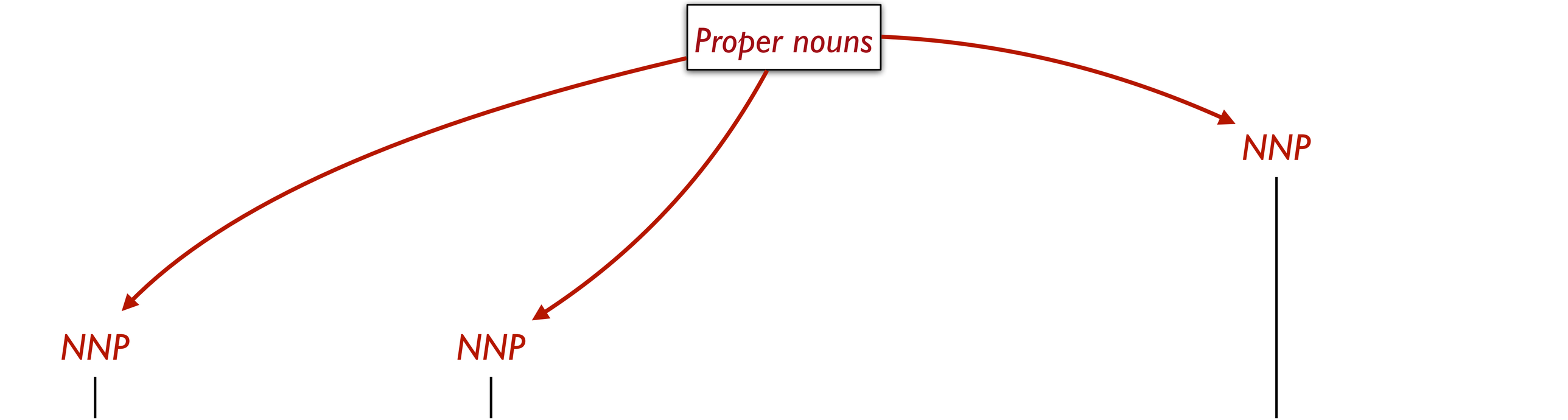
Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo

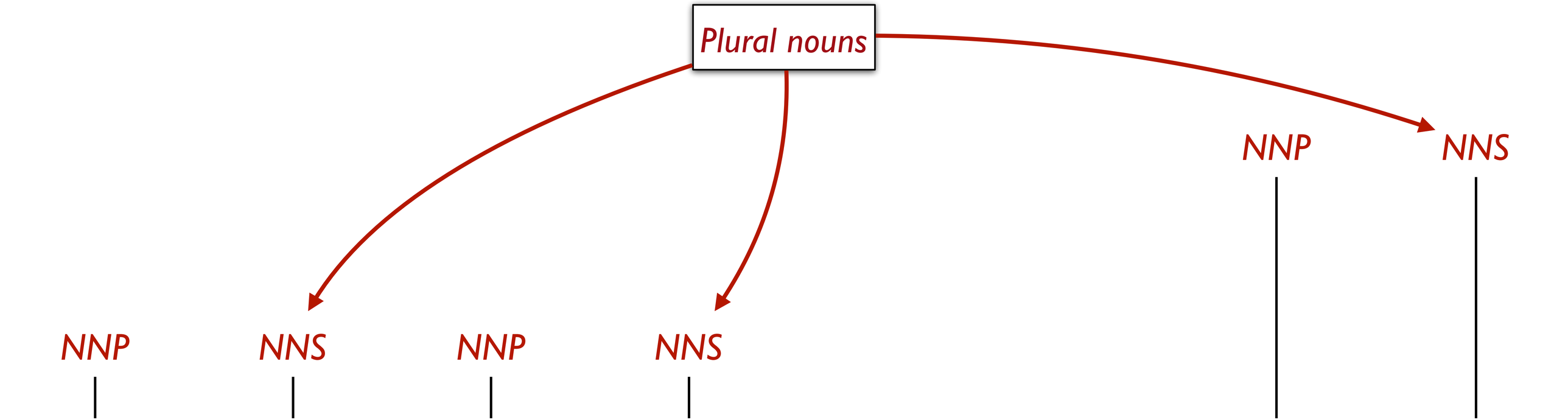


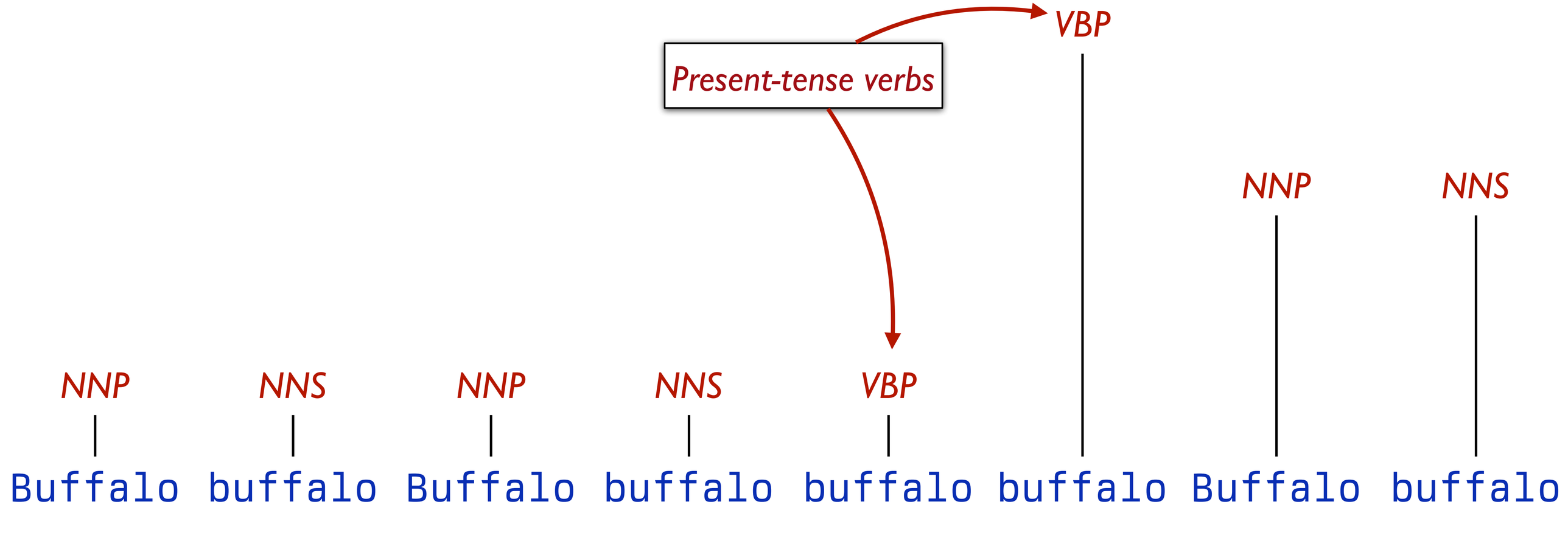
Greg Williams

(CC BY-SA)

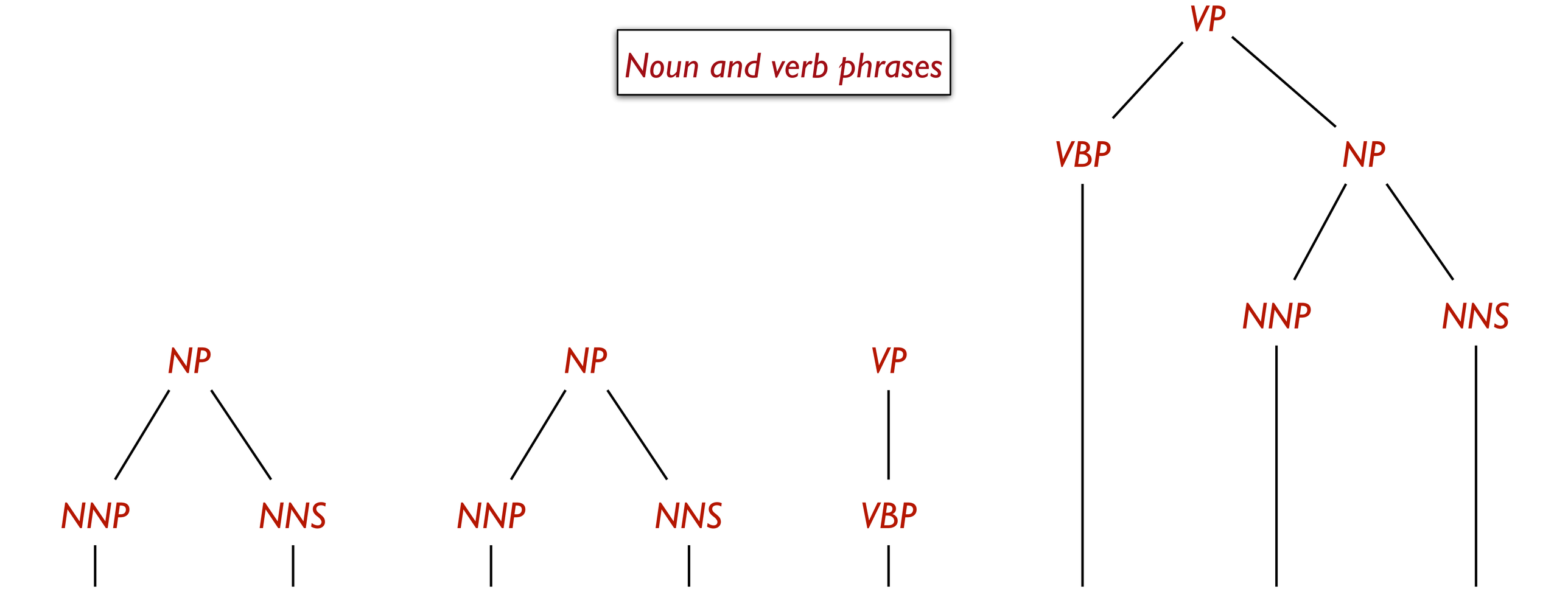
Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo


Proper nouns
NNP
NNP
NNP
Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo

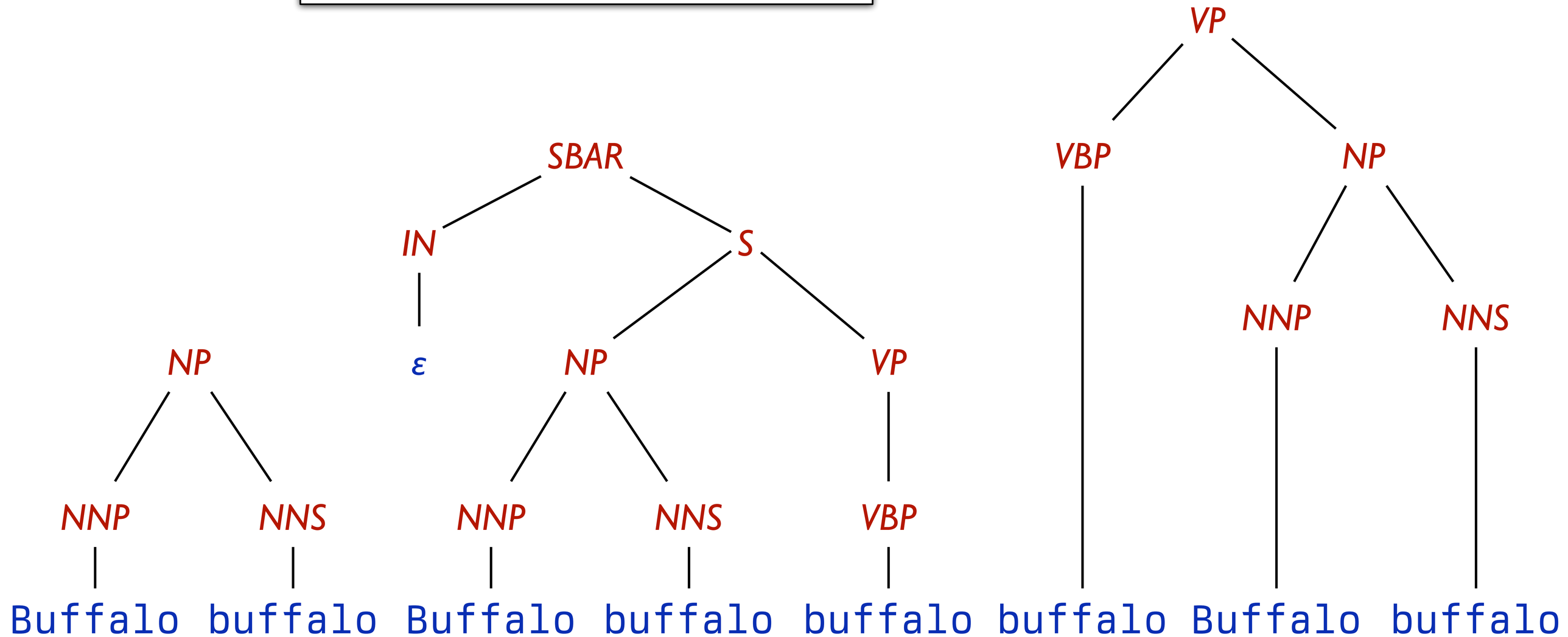

Plural nouns
NNP
NNS
NNP
NNS
NNP
NNS
Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo

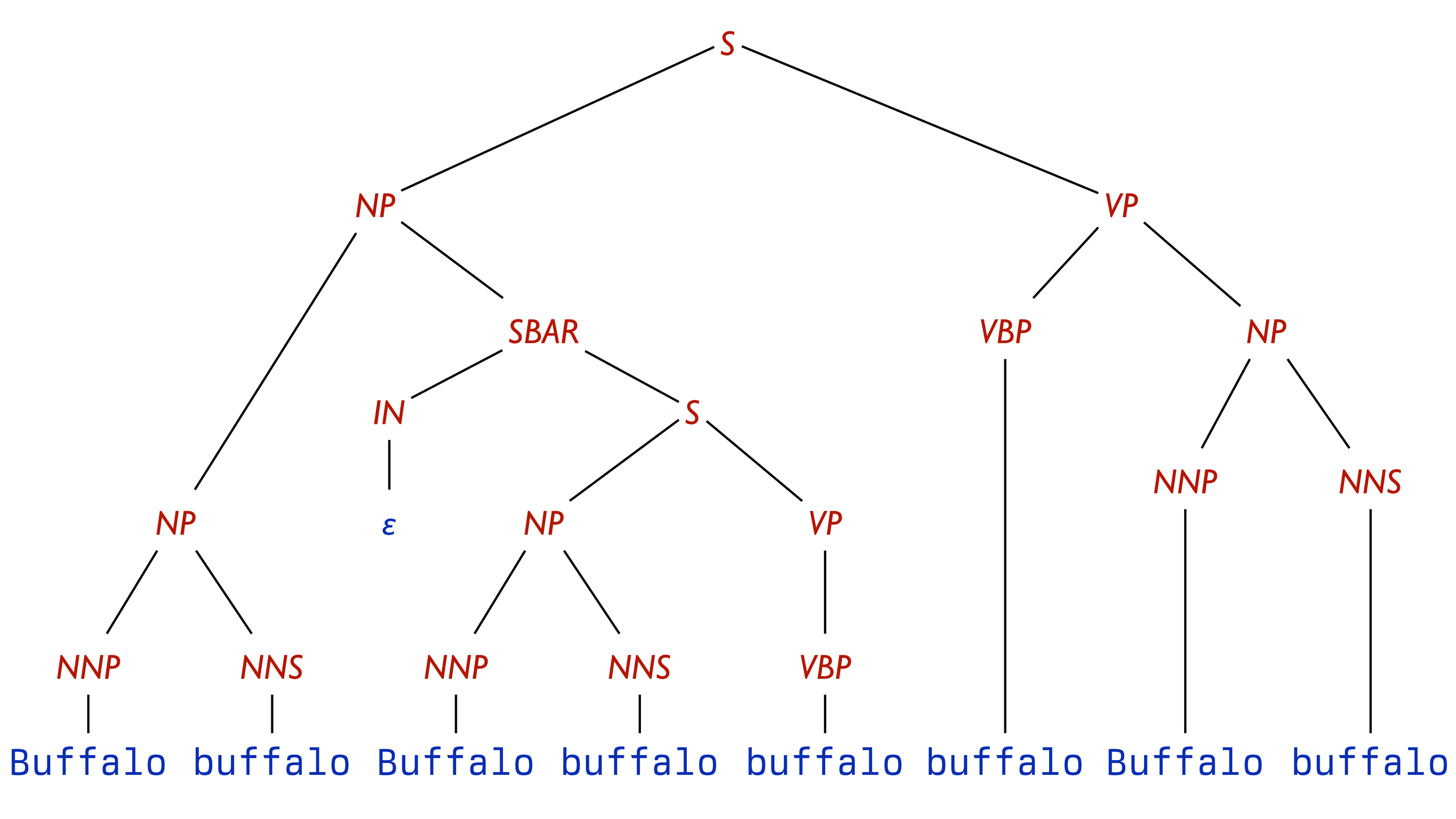

Present-tense verbs
VBP
NNP
NNS
NNP
NNS
VBP
NNP
NNS
Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo

Noun and verb phrases



A subordinate clause,
without a subordinating conjunction (IN)




S
NP
VP
SBAR
VBP
NP
IN
S
NNP
NNS
NP
ε
NP
VP
NNP
NNS
NNP
NNS
VBP
Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo

$$
\begin{aligned}
S &\rightarrow NP\ VP \\
NP &\rightarrow NNP\ NNS \mid NP\ SBAR \\
SBAR &\rightarrow IN\ S \\
VP &\rightarrow VBP \mid VBP\ NP \\
NNP &\rightarrow \texttt{Buffalo} \\
NNS &\rightarrow \texttt{buffalo} \\
VBP &\rightarrow \texttt{buffalo}
\end{aligned}
$$

CLAIM We could expand the grammar and keep going: $\{\texttt{buffalo}^n \mid n \in \mathbb{N}_0\}$ is a subset of English.

A programming language example

Given a CFG describing the structure of a programming language and an input program (a string), the compiler or interpreter needs to recover the parse tree.

The parse tree represents the structure of the program – what's declared where, how expressions nest, etc.

Annex A
(informative)

Language syntax summary

1 NOTE The notation is described in 6.1.

A.1 Lexical grammar

A.1.1 Lexical elements

(6.4) *token:*
keyword
identifier
constant
string-literal
punctuator

(6.4) *preprocessing-token:*
header-name
identifier
pp-number
character-constant
string-literal
punctuator
each non-white-space character that cannot be one of the above

A CFG (with slightly different notation) for the C programming language

Block → *Stmt*
| `{` *Stmts* `}`

Stmts → ε
| *Stmt Stmts*

Stmt → *Expr*`;`
| `if (`*Expr*`)` *Block*
| `while (`*Expr*`)` *Block*
| `do` *Block* `while (`*Expr*`);`
| *Block*
| …

Expr → `var`
| `const`
| *Expr* + *Expr*
| *Expr* - *Expr*
| *Expr* = *Expr*
| …

```
{
    var = var * var;
    if (var)
        var = const;
    while (var) {
        var = var + const;
    }
}
```

A big problem

Interactive Python

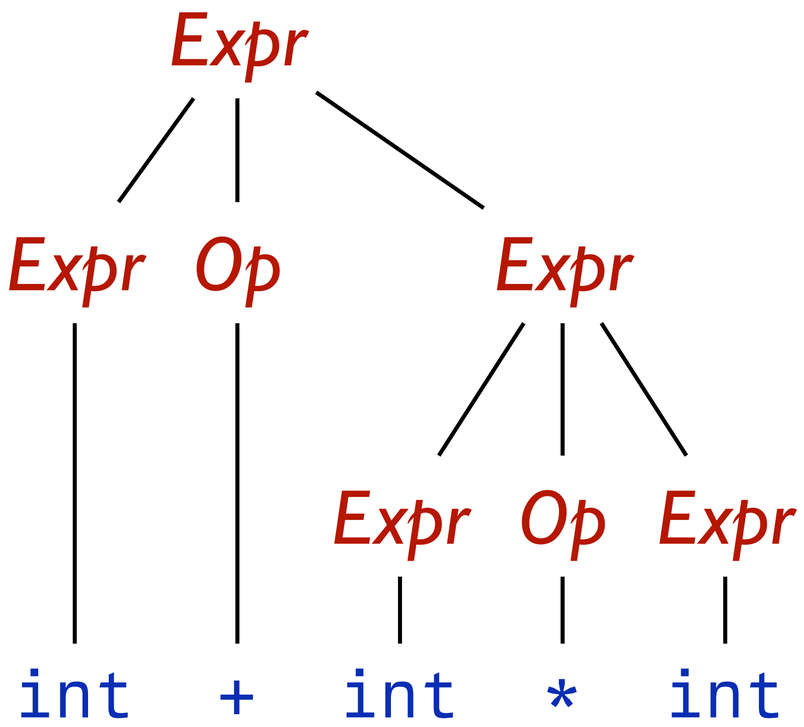
```
; ipython

>>> (137 + 42) - 2 * 3
173

>>> (60 + 37) + 5 * 8
137

>>> (200 / 2) + 6 / 2
103.0

>>>
```

Let's return to our grammar for arithmetic expressions

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \mid \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid (\mathit{Expr}) \\
\mathit{Op} &\rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

```
int + int * int
```

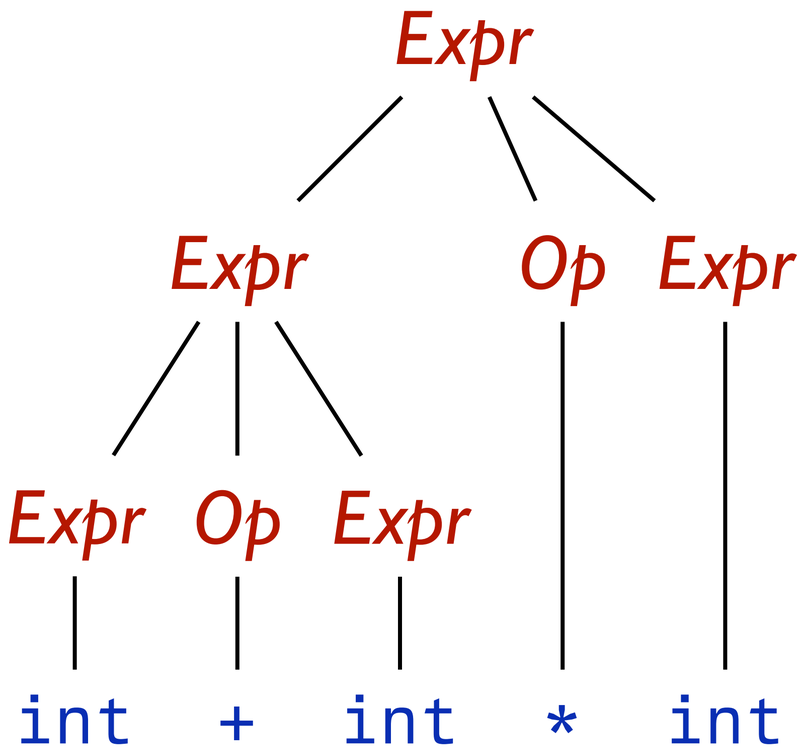
$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \mid \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid (\mathit{Expr}) \\
\mathit{Op} &\rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$



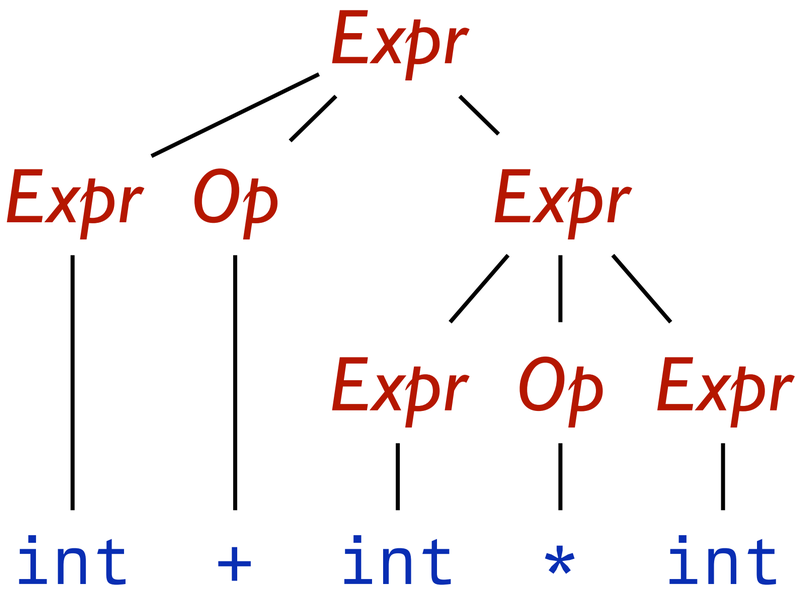
$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \mid \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid (\mathit{Expr}) \\
\mathit{Op} &\rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

```
int + int * int
```

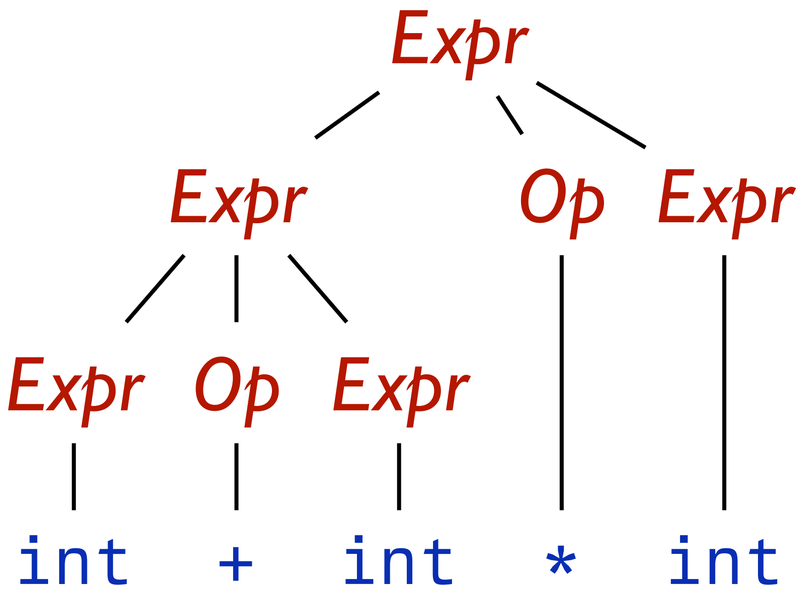
$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \mid \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid (\mathit{Expr}) \\
\mathit{Op} &\rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$



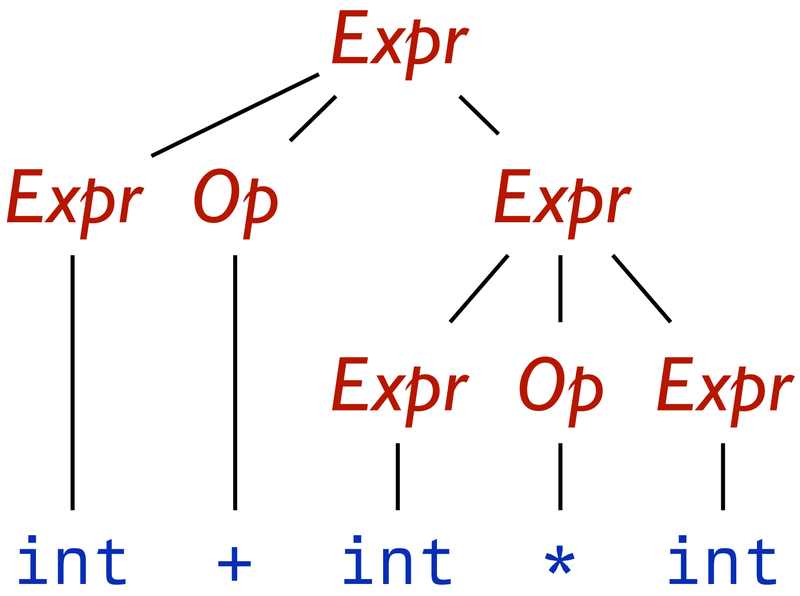
This parse tree means that we multiply before adding.

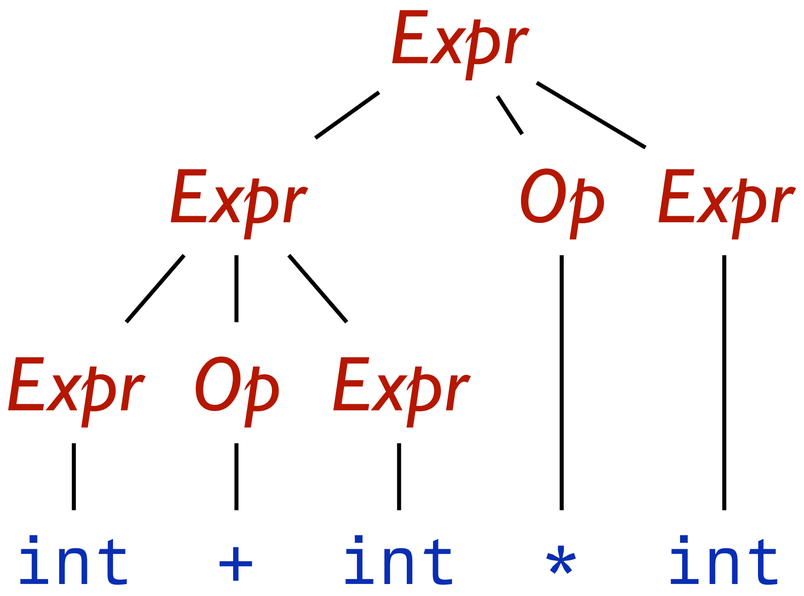


This parse tree means that we add before multiplying.



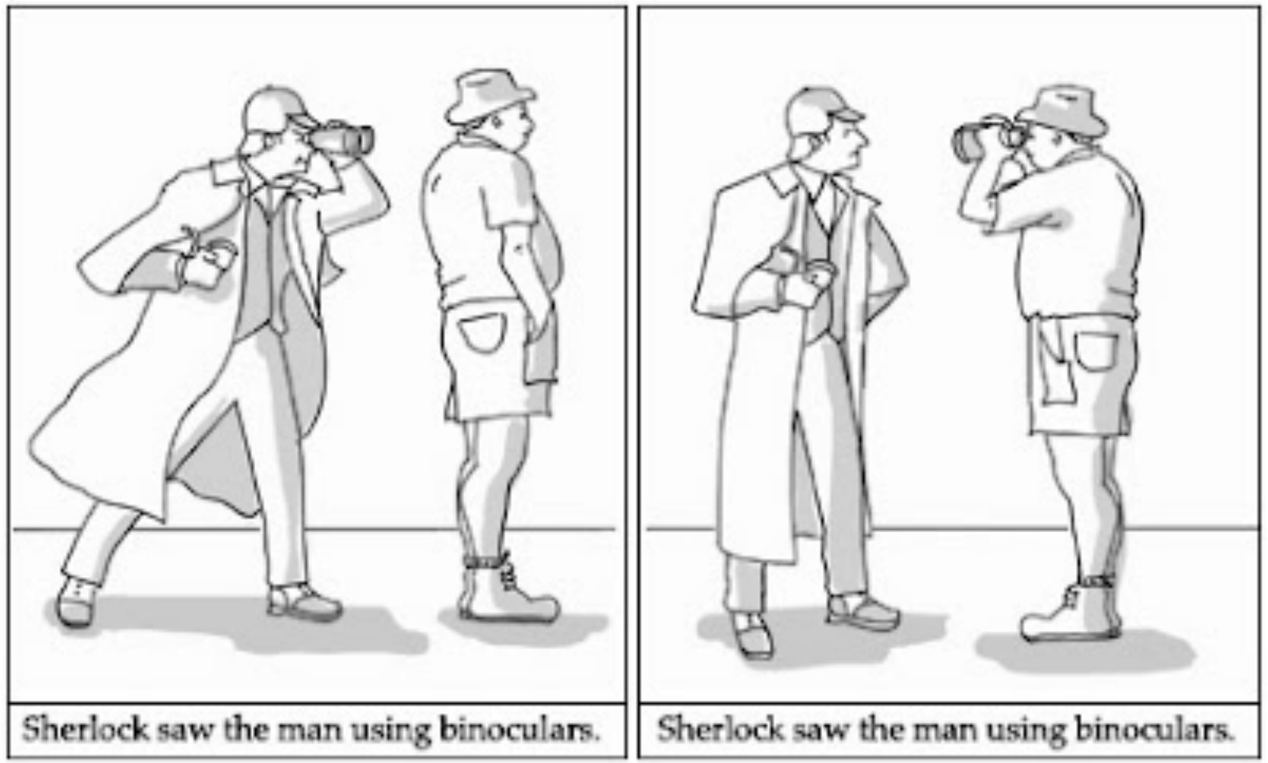
A grammar is *ambiguous* if there is more than one way to parse a string.





A grammar is ***ambiguous*** if there is more than one way to parse a string.

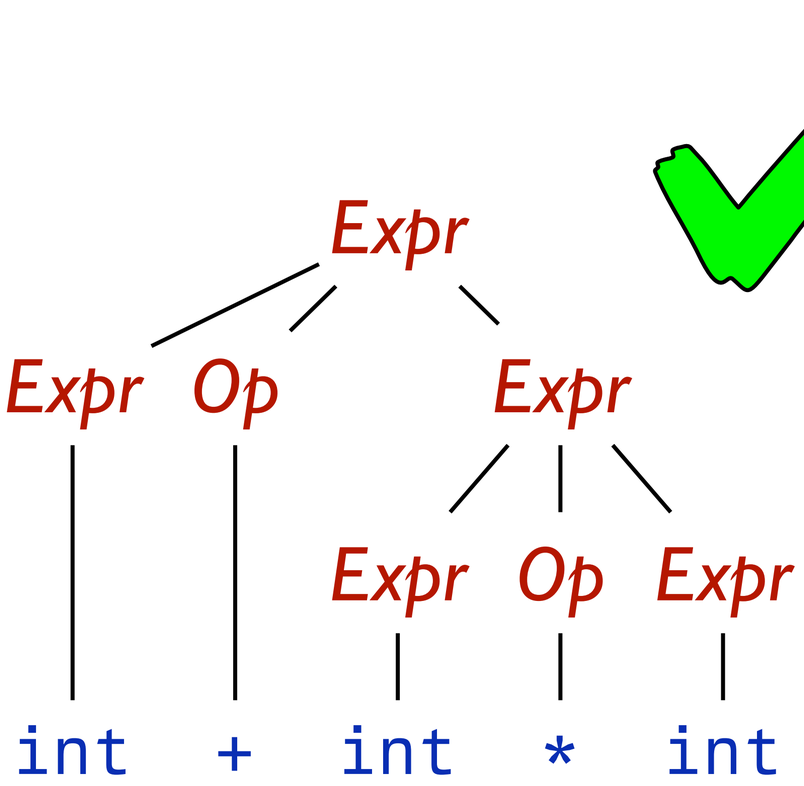
Ambiguity is ubiquitous in natural language…

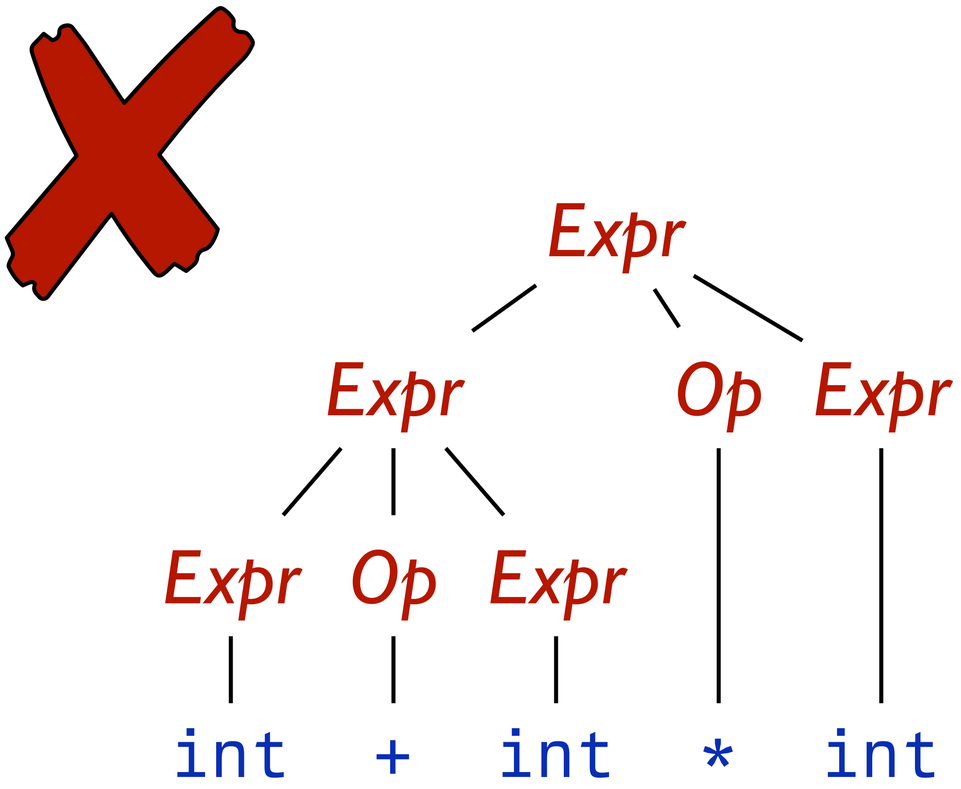


bb.ustc.edu.cn

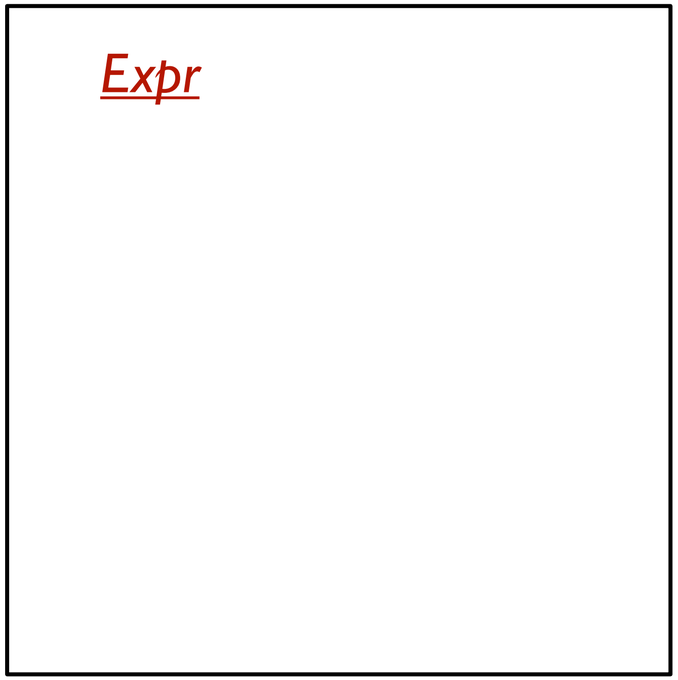
A grammar is *ambiguous* if there is more than one way to parse a string.

Ambiguity is ubiquitous in natural language… but it's a bad thing in a programming language!





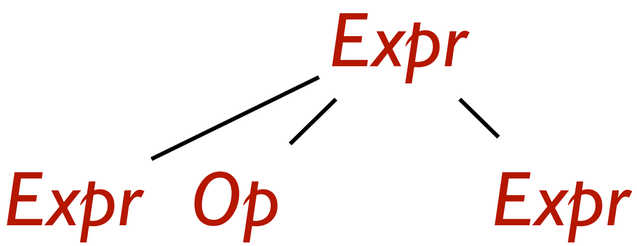
Equivalently, we could say a CFG is ambiguous if there is at least one string that has multiple *leftmost derivations*, where you always expand the leftmost variable before any others.



Expr

Expr

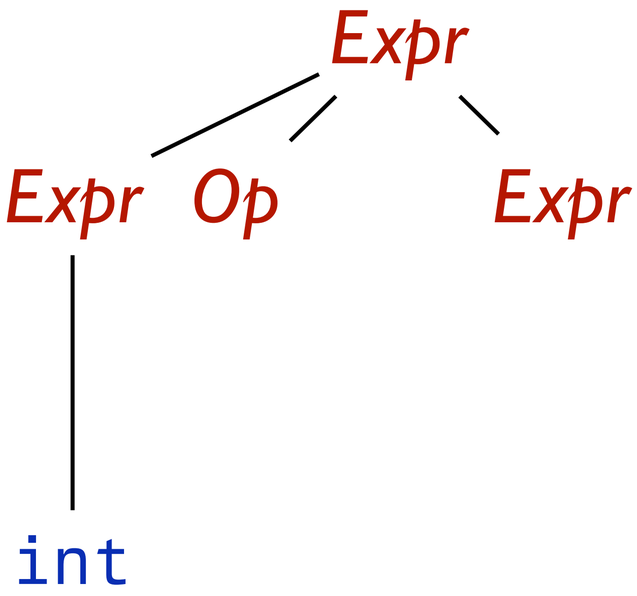
⇒ *Expr Op Expr*



Expr

⇒ *Expr Op Expr*

⇒ `int` *Op Expr*

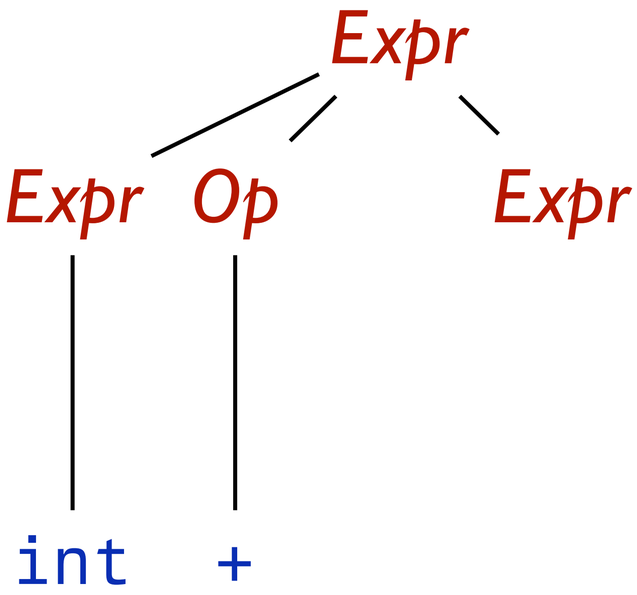


Expr

⇒ *Expr* *Op* *Expr*

⇒ `int` *Op* *Expr*

⇒ `int` `+` *Expr*



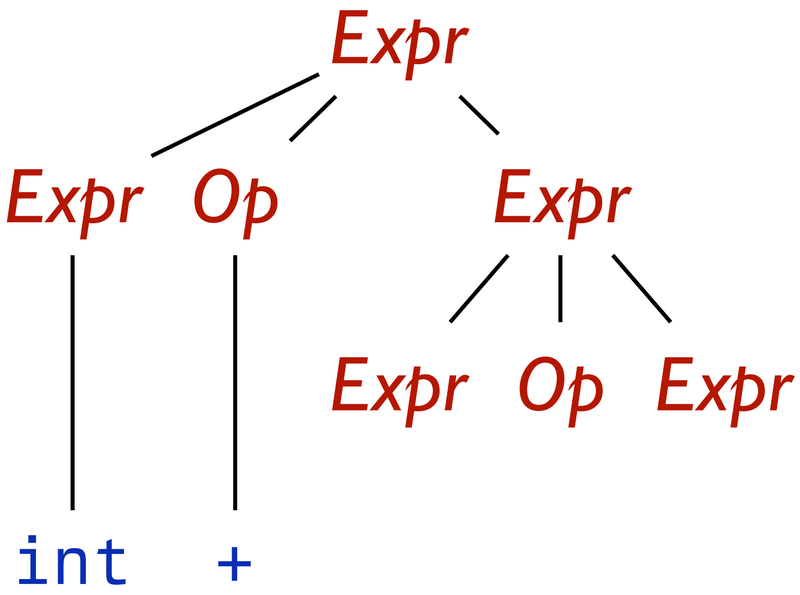
Expr

⇒ *Expr* *Op* *Expr*

⇒ `int` *Op* *Expr*

⇒ `int` `+` *Expr*

⇒ `int` `+` *Expr* *Op* *Expr*



Expr

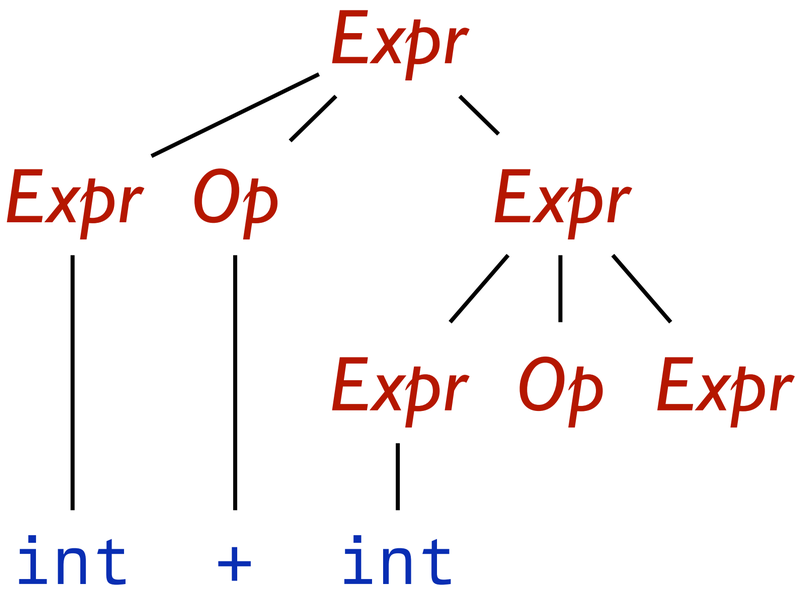
⇒ *Expr Op Expr*

⇒ `int` *Op Expr*

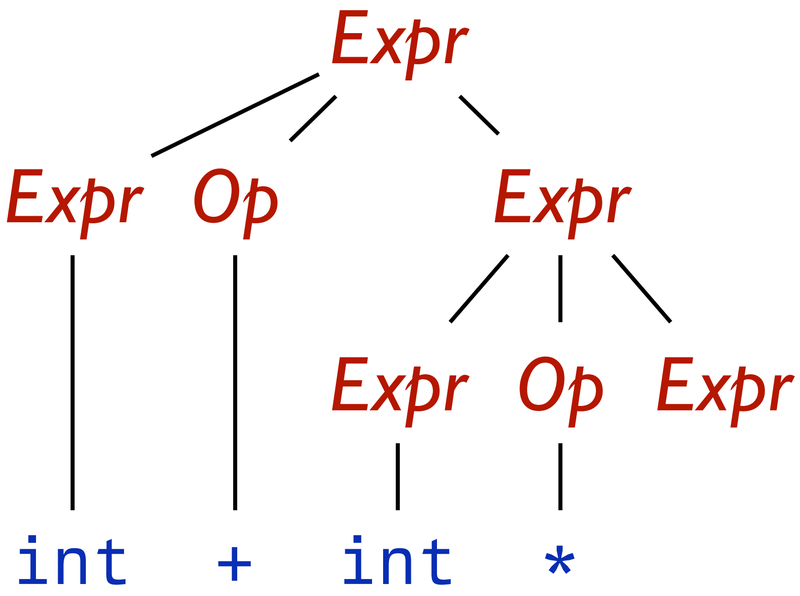
⇒ `int +` *Expr*

⇒ `int +` *Expr Op Expr*

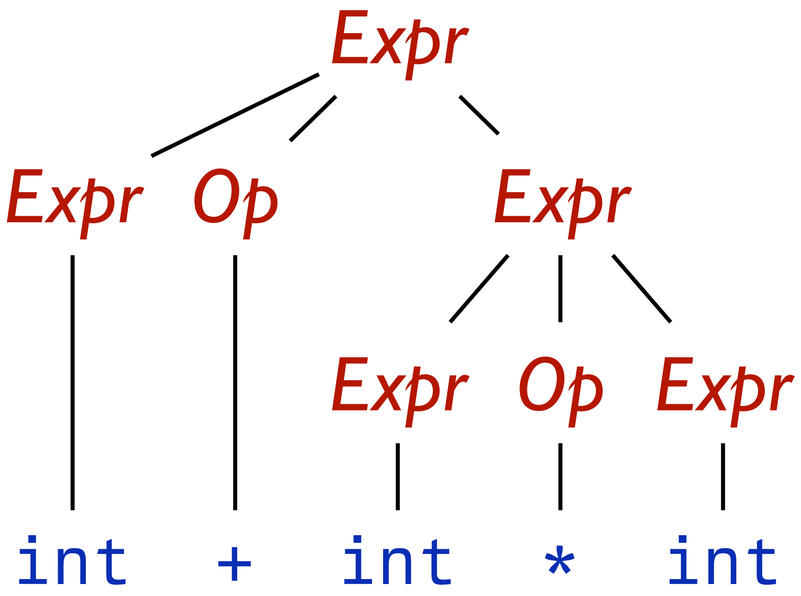
⇒ `int + int` *Op Expr*



Expr
⇒ *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr*
⇒ `int +` *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int + int` *Op* *Expr*
⇒ `int + int *` *Expr*

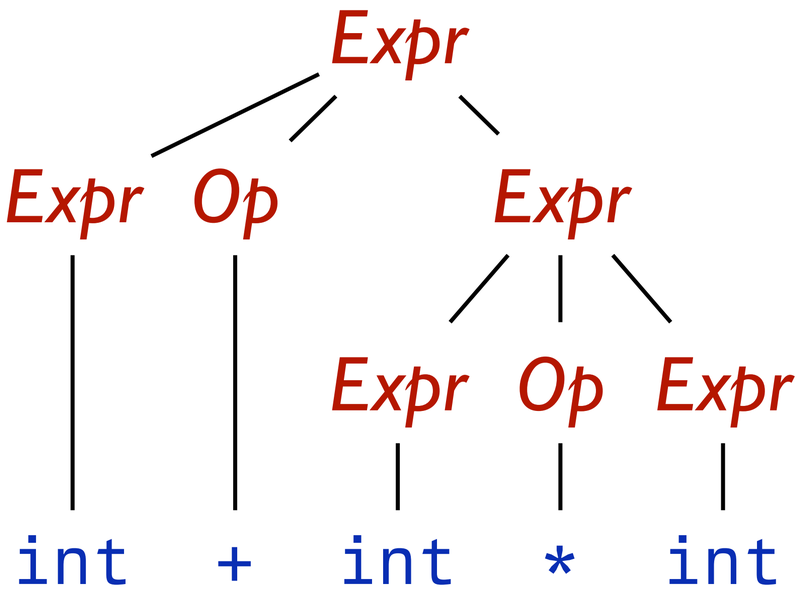


Expr
⇒ *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr*
⇒ `int +` *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int + int` *Op* *Expr*
⇒ `int + int *` *Expr*
⇒ `int + int * int`



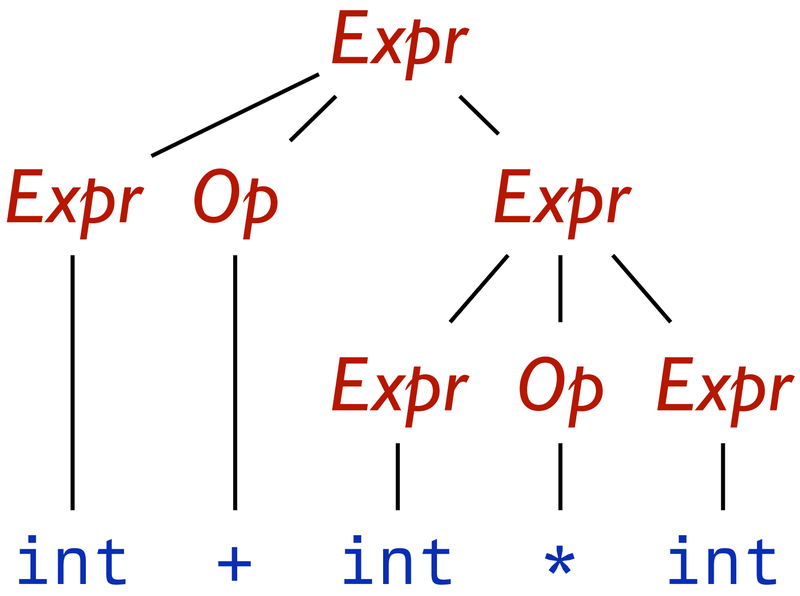
Expr
⇒ *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr*
⇒ `int +` *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int + int` *Op* *Expr*
⇒ `int + int *` *Expr*
⇒ `int + int * int`

Expr

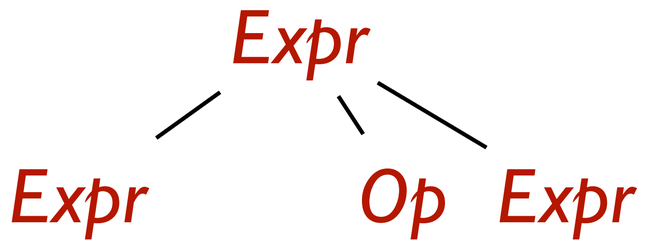


Expr

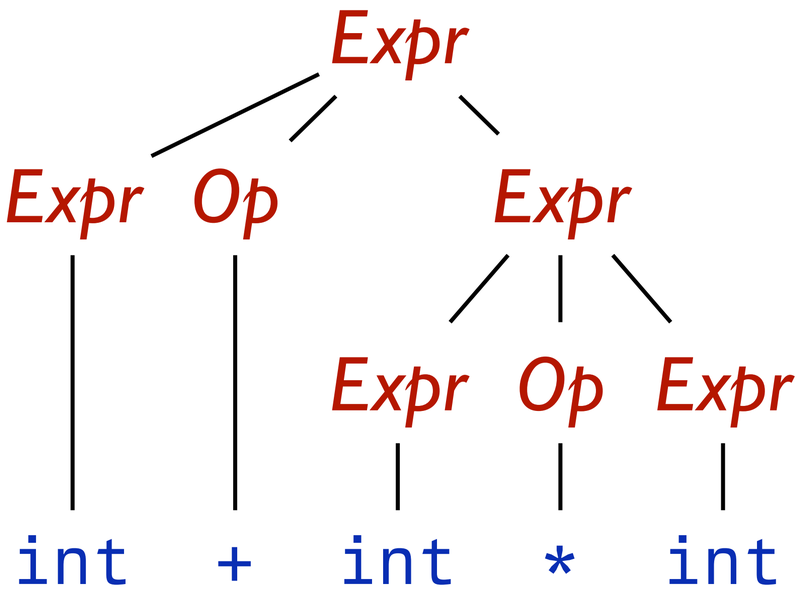
Expr
⇒ *Expr Op Expr*
⇒ `int` *Op Expr*
⇒ `int +` *Expr*
⇒ `int +` *Expr Op Expr*
⇒ `int + int` *Op Expr*
⇒ `int + int *` *Expr*
⇒ `int + int * int`



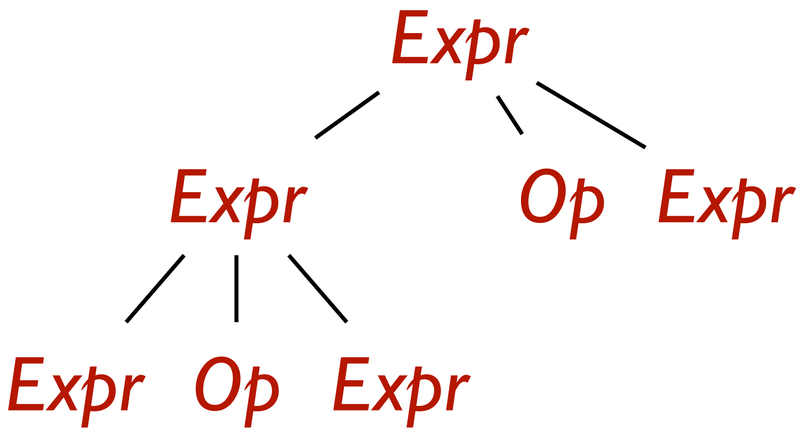
Expr
⇒ *Expr Op Expr*



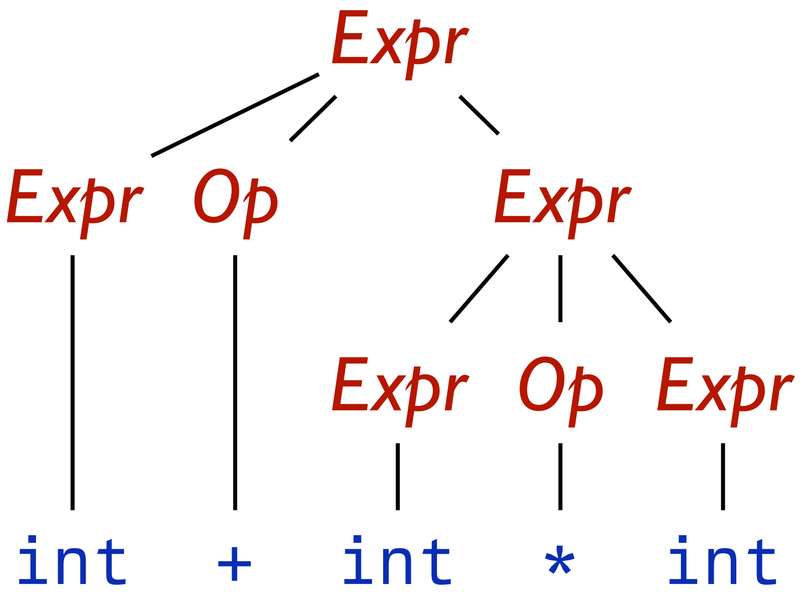
Expr
⇒ *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr*
⇒ `int +` *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int + int` *Op* *Expr*
⇒ `int + int *` *Expr*
⇒ `int + int * int`



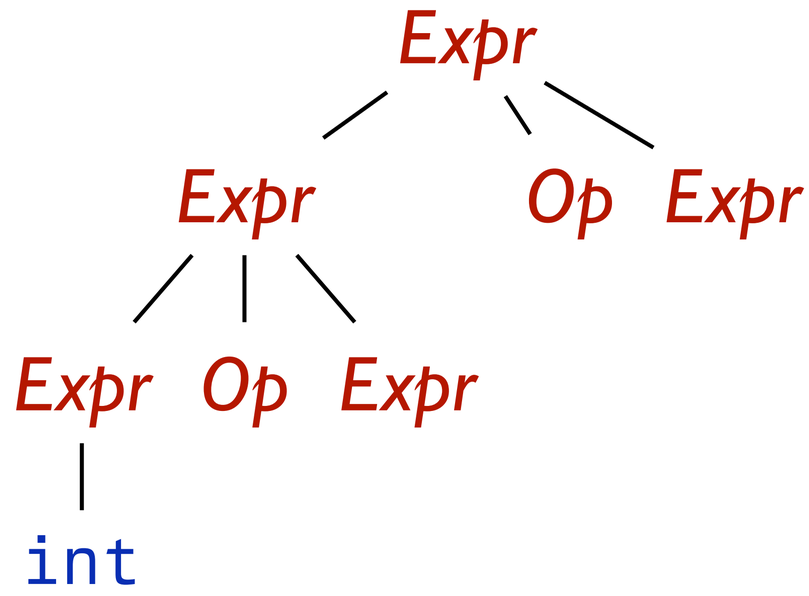
Expr
⇒ *Expr* *Op* *Expr*
⇒ *Expr* *Op* *Expr* *Op* *Expr*



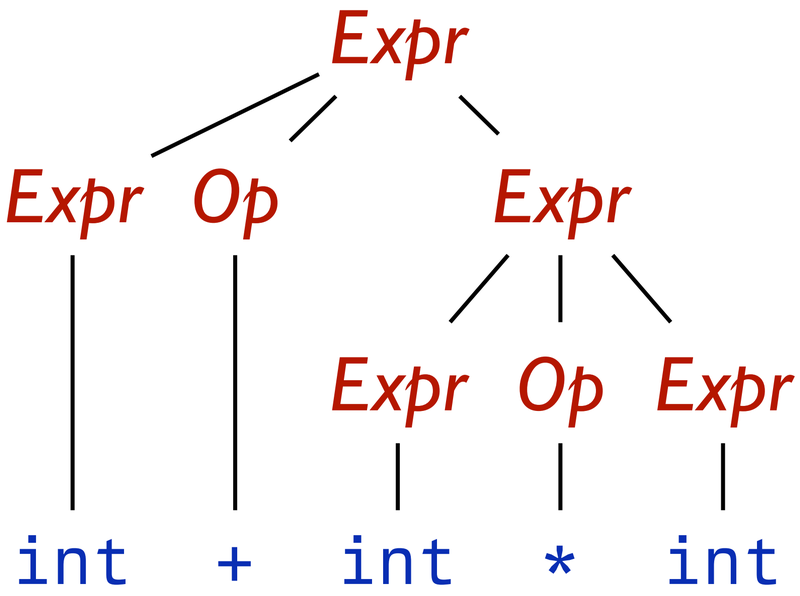
Expr
⇒ *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr*
⇒ `int +` *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int + int` *Op* *Expr*
⇒ `int + int *` *Expr*
⇒ `int + int * int`



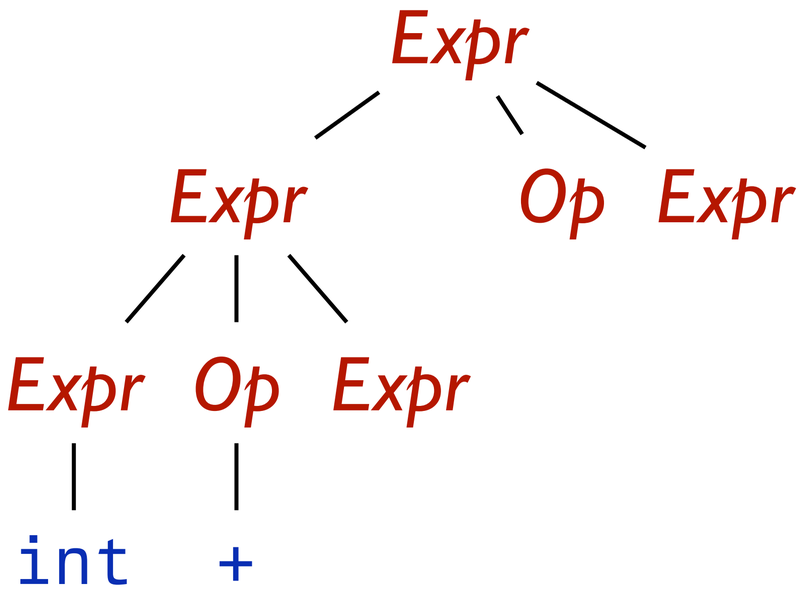
Expr
⇒ *Expr* *Op* *Expr*
⇒ *Expr* *Op* *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr* *Op* *Expr*



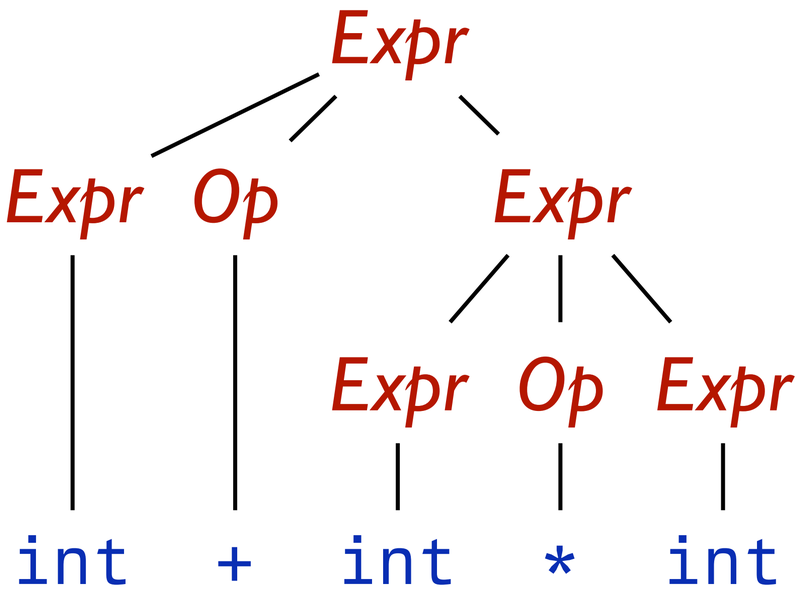
Expr
⇒ *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr*
⇒ `int +` *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int + int` *Op* *Expr*
⇒ `int + int *` *Expr*
⇒ `int + int * int`



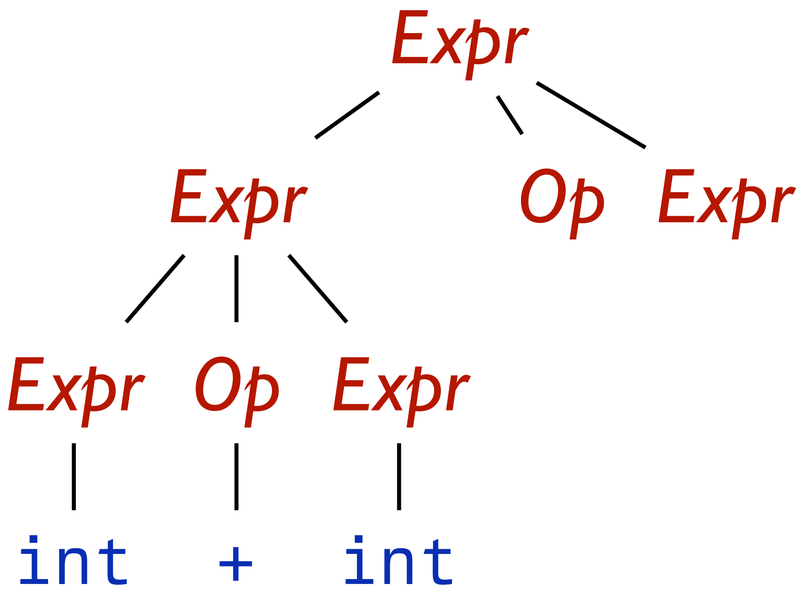
Expr
⇒ *Expr* *Op* *Expr*
⇒ *Expr* *Op* *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr* *Op* *Expr*
⇒ `int +` *Expr* *Op* *Expr*



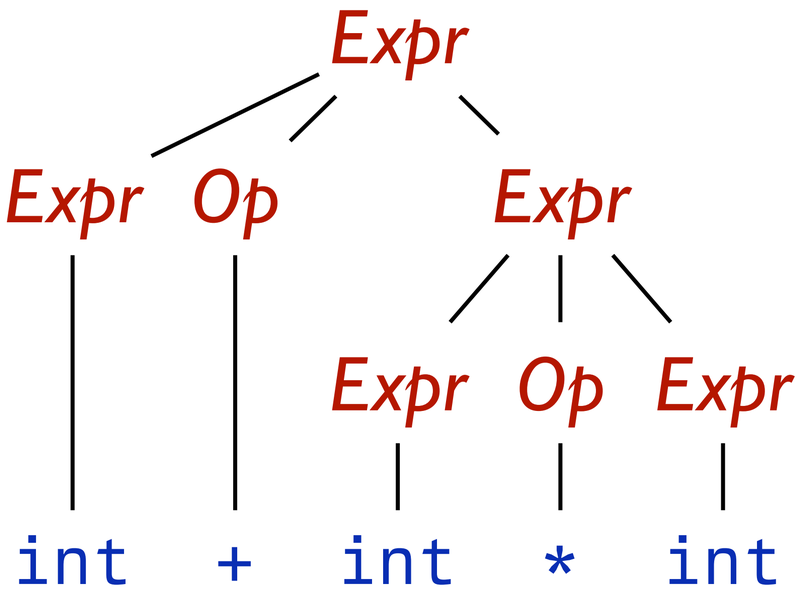
Expr
⇒ *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr*
⇒ `int +` *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int + int` *Op* *Expr*
⇒ `int + int *` *Expr*
⇒ `int + int * int`



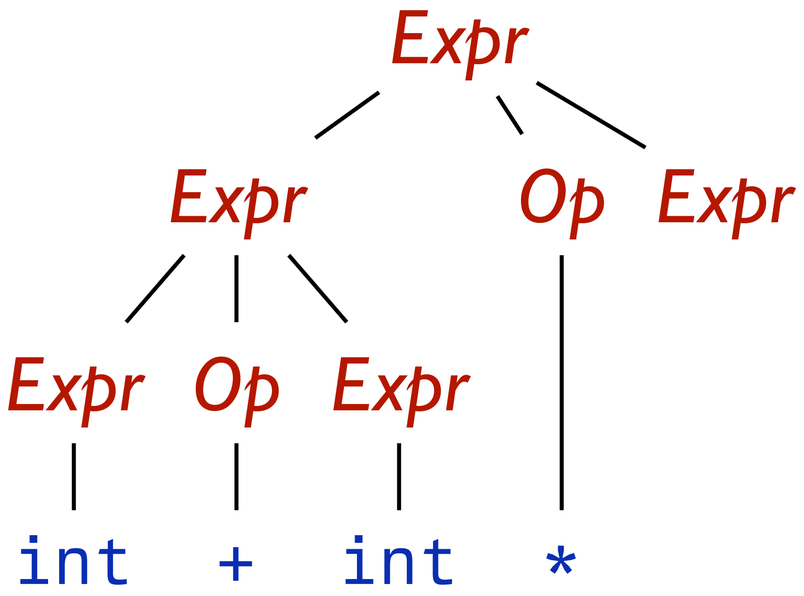
Expr
⇒ *Expr* *Op* *Expr*
⇒ *Expr* *Op* *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr* *Op* *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int + int` *Op* *Expr*



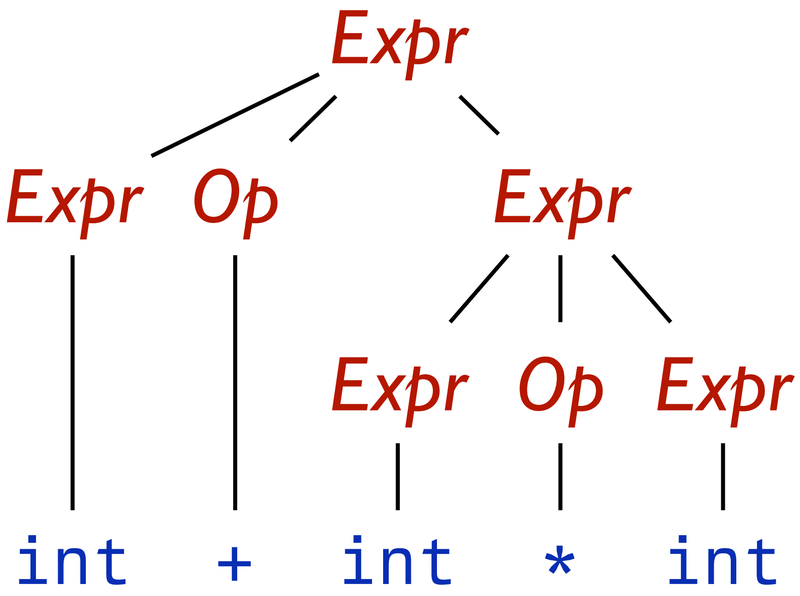
Expr
⇒ *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr*
⇒ `int +` *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int + int` *Op* *Expr*
⇒ `int + int *` *Expr*
⇒ `int + int * int`



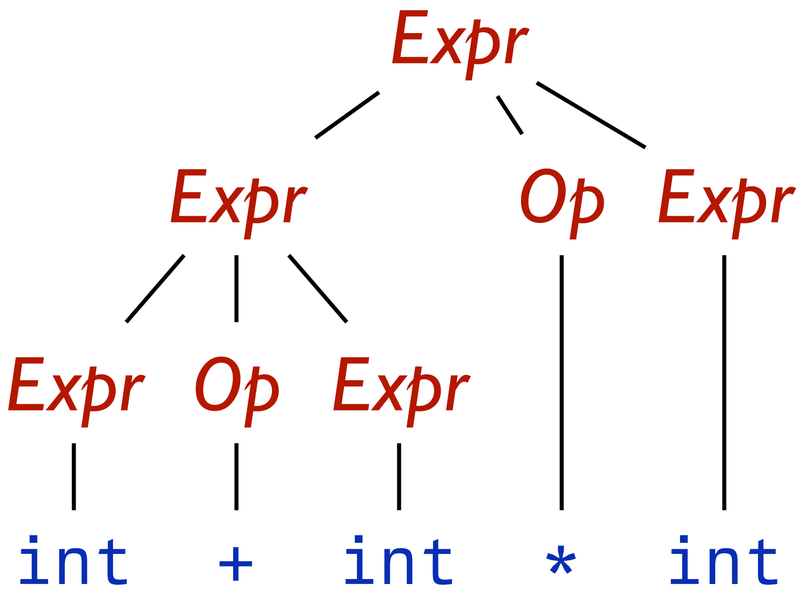
Expr
⇒ *Expr* *Op* *Expr*
⇒ *Expr* *Op* *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr* *Op* *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int + int` *Op* *Expr*
⇒ `int + int *` *Expr*



Expr
⇒ *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr*
⇒ `int +` *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int +` `int` *Op* *Expr*
⇒ `int + int *` *Expr*
⇒ `int + int * int`



Expr
⇒ *Expr* *Op* *Expr*
⇒ *Expr* *Op* *Expr* *Op* *Expr*
⇒ `int` *Op* *Expr* *Op* *Expr*
⇒ `int +` *Expr* *Op* *Expr*
⇒ `int +` `int` *Op* *Expr*
⇒ `int + int *` *Expr*
⇒ `int + int * int`



$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \mid \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid (\mathit{Expr}) \\
\mathit{Op} &\rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

How do we remove this ambiguity from the grammar?

$$
\begin{aligned}
\mathit{Expr} &\rightarrow \texttt{int} \mid \mathit{Expr}\ \mathit{Op}\ \mathit{Expr} \mid \texttt{(}\mathit{Expr}\texttt{)} \\
\mathit{Op} &\rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

By adding new variables, we can enforce an *order* for generating a string in the language:

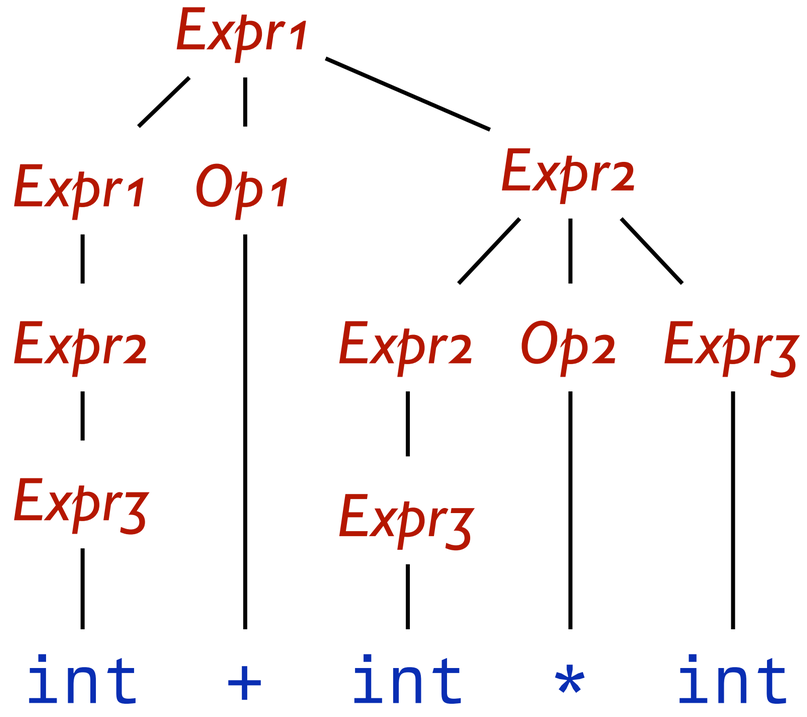
$$
\begin{aligned}
\mathit{Expr1} &\rightarrow \mathit{Expr1}\ \mathit{Op1}\ \mathit{Expr2} \mid \mathit{Expr2} \\
\mathit{Op1} &\rightarrow \texttt{+} \mid \texttt{-} \\
\\
\mathit{Expr2} &\rightarrow \mathit{Expr2}\ \mathit{Op2}\ \mathit{Expr3} \mid \mathit{Expr3} \\
\mathit{Op2} &\rightarrow \texttt{*} \mid \texttt{/} \\
\\
\mathit{Expr3} &\rightarrow \texttt{(}\mathit{Expr}\texttt{)} \mid \texttt{int}
\end{aligned}
$$

The grammar and the parse tree are more complicated now, but we only have one possible parse tree (and, thus, one leftmost derivation) for this string.

Expr1 → *Expr1 Op1 Expr2* | *Expr2*
Op1 → `+` | `-`

Expr2 → *Expr2 Op2 Expr3* | *Expr3*
Op2 → `*` | `/`

Expr3 → `(`*Expr1*`)` | `int`



Where does that leave us?

A language *L* is called a ***context-free language*** (or CFL) if there is a CFG *G* such that *L* = *L*(*G*).

The language for a grammar is ***ambiguous*** if there's more than one way to parse a string.

Ambiguity is a property of *grammars*, not *languages*.

There can be multiple grammars for the same language, where some are ambiguous and some aren't.

A language L is *inherently ambiguous* if *every* CFG for L is ambiguous.

See Problem 2.29 in Sipser for an example of an inherently ambiguous language.

Questions for next time

How do we design context-free grammars for context-free languages?

What languages are context-free?

How are context-free and regular languages related?

Acknowledgments

This lecture incorporates material from:

David Chiang, University of Notre Dame

Keith Schwarz and Amy Liu, Stanford University

Michael Sipser, *Introduction to the Theory of Computation*