CS 358: Principles of Programming Languages

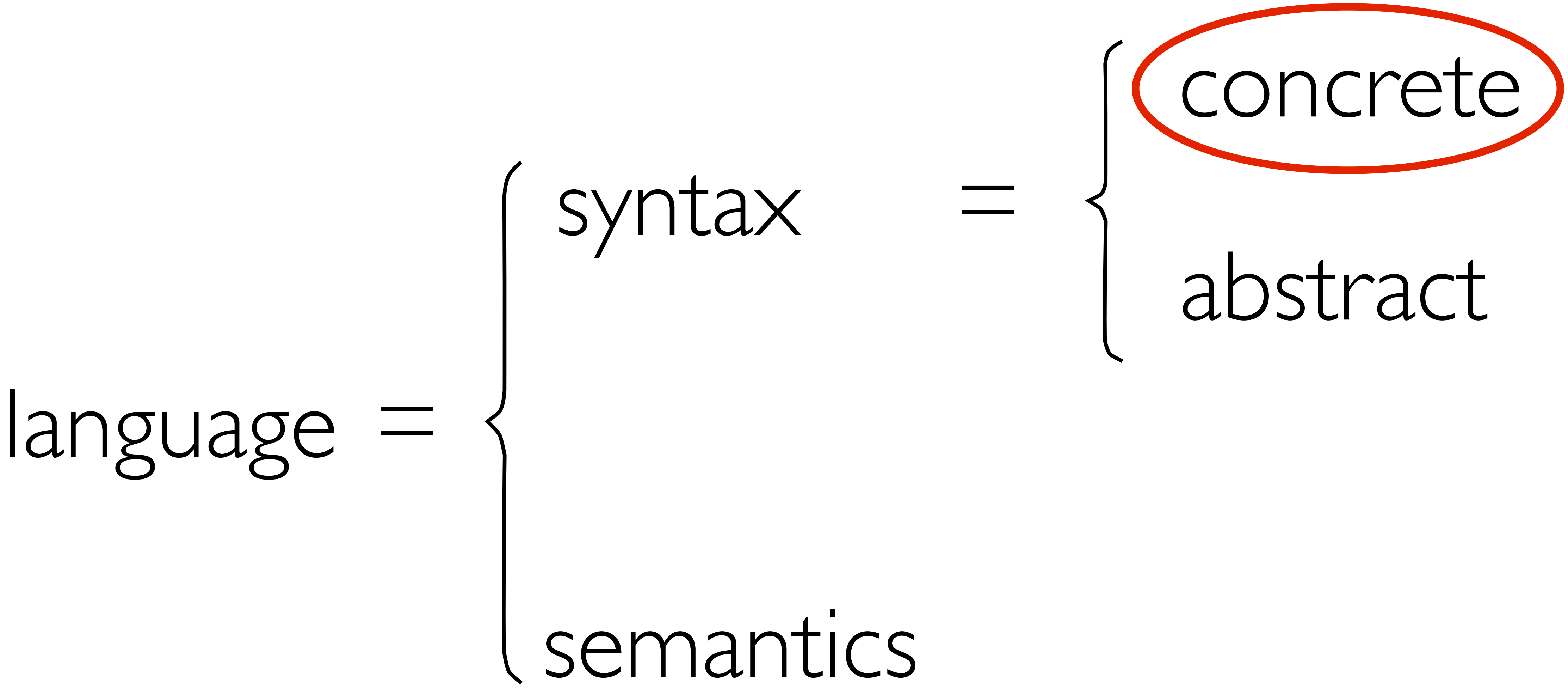
Andrew Tolmach, Mark P. Jones, Jingke Li, Katie Casamento
Portland State University

Winter 2025

Week 4: Describing Syntax

$$\text{language} = \begin{cases} \text{syntax} \\ \\ \text{semantics} \end{cases}$$

syntax: the written/spoken/symbolic/physical form; how things are communicated



language =

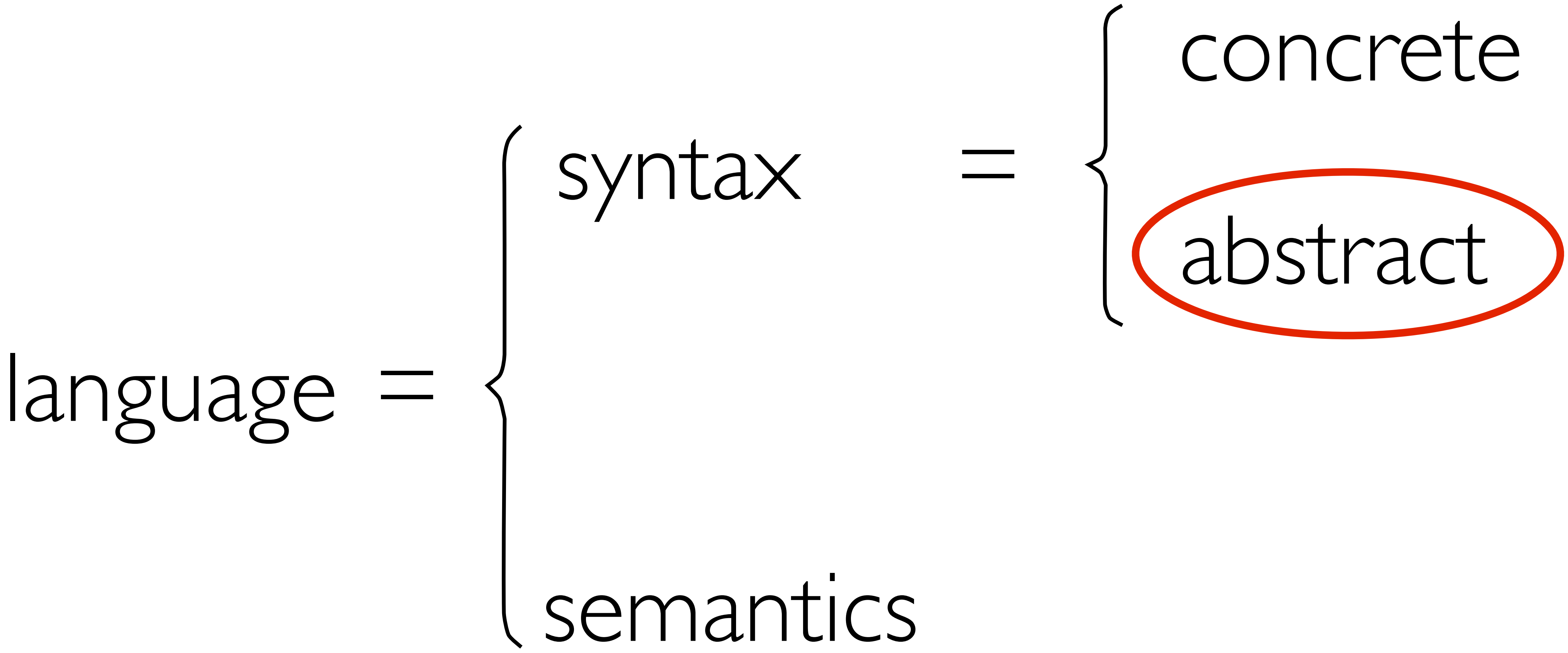
syntax =

concrete

abstract

semantics

concrete syntax: the representation of a program text in its source form as a sequence of bits/bytes/characters/lines



language =

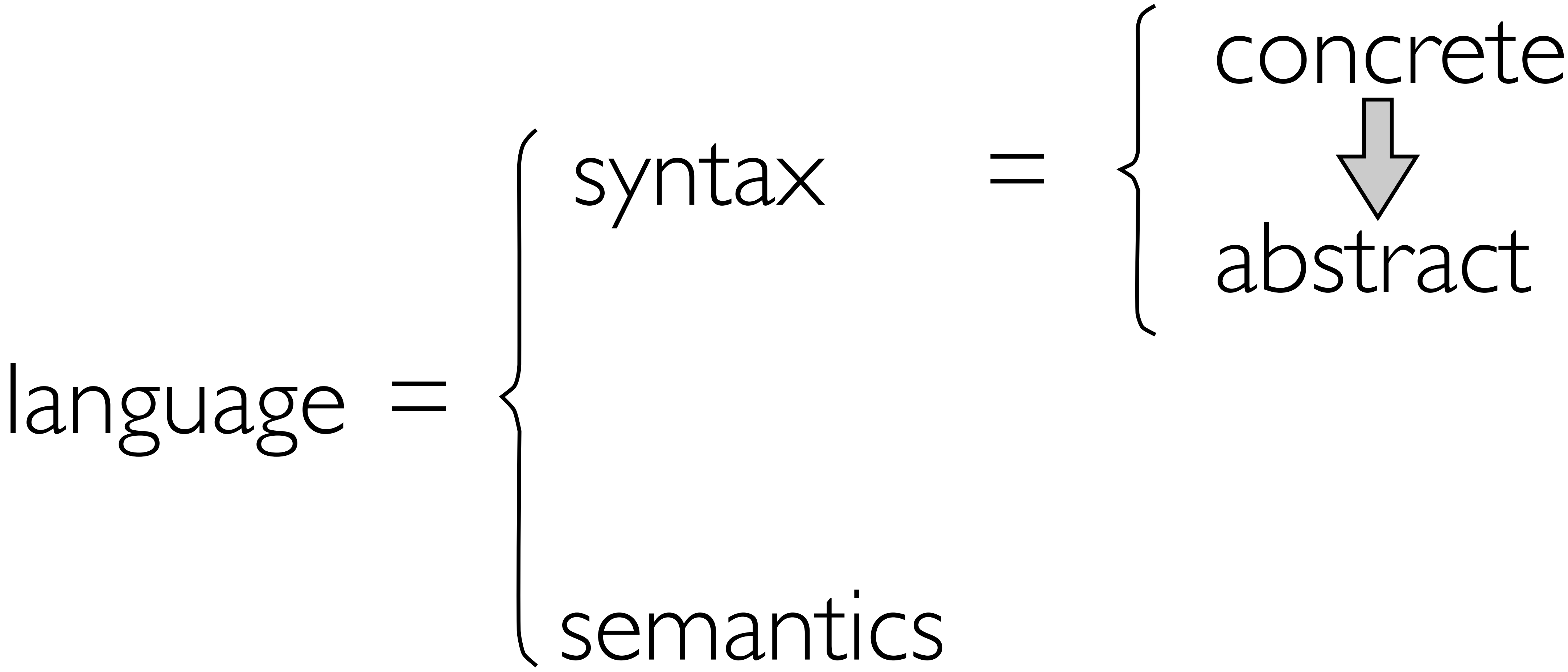
syntax =

concrete

abstract

semantics

abstract syntax: the representation of a program structure, independent of written form



language =

syntax =

concrete

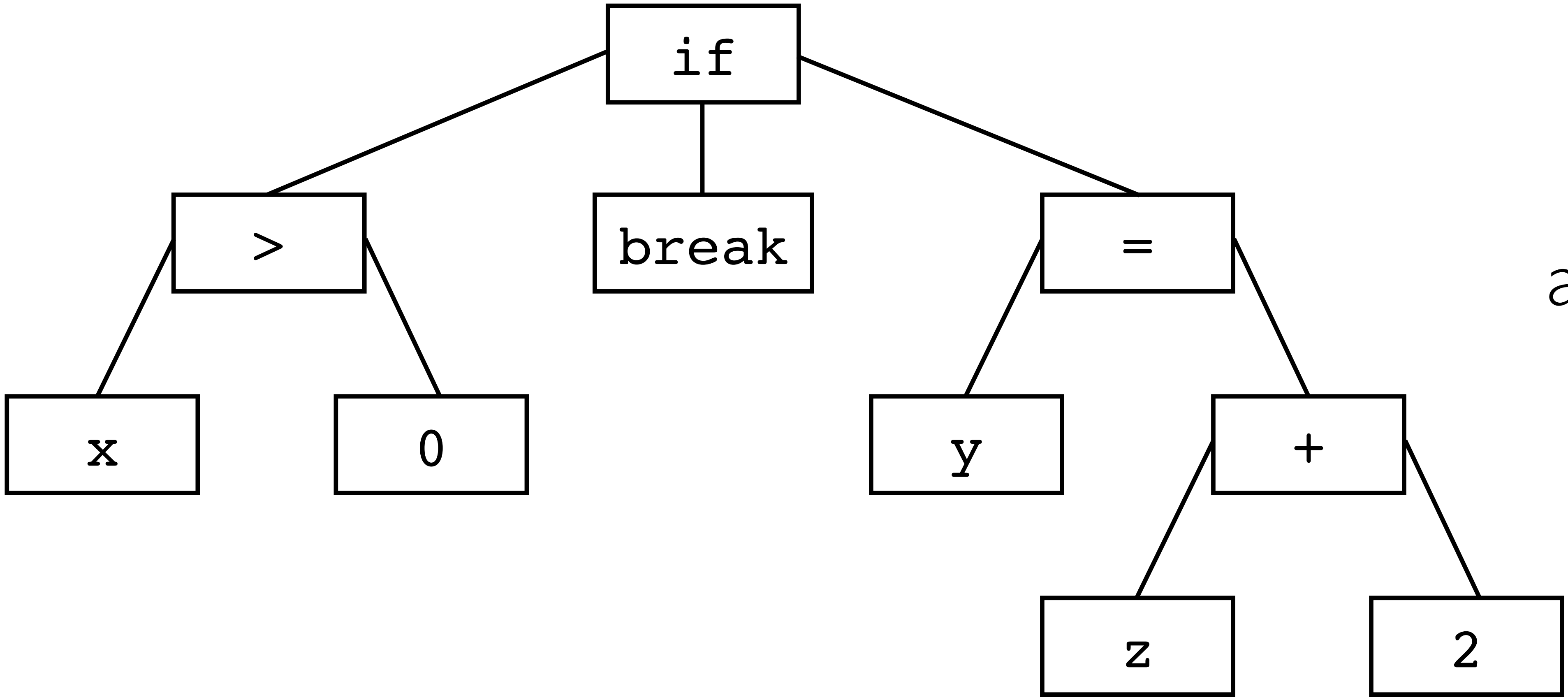
abstract

semantics

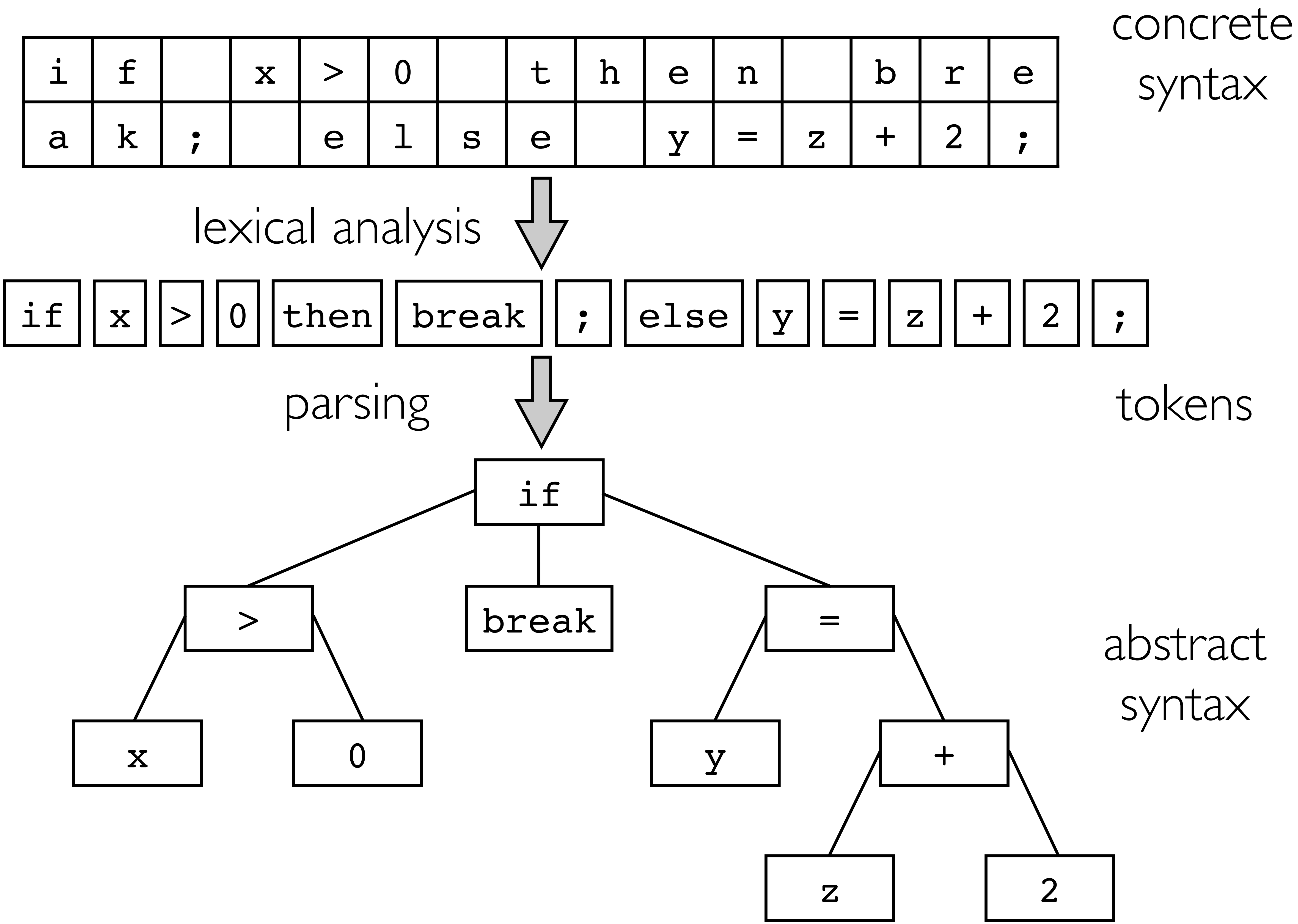
syntax analysis: This is one of the areas where theoretical computer science has had major impact on the practice of software development concrete syntax

i	f		x	>	0		t	h	e	n		b	r	e
a	k	;		e	l	s	e		y	=	z	+	2	;

syntax analysis



abstract syntax



concrete syntax

i	f		x	>	0		t	h	e	n		b	r	e
a	k	;		e	l	s	e		y	=	z	+	2	;

lexical analysis

if x > 0 then break ; else y = z + 2 ;

parsing

tokens

abstract syntax

Wanted!

- **We want methods for describing language syntax that are:**
  - clear, precise, and unambiguous
  - expressive (e.g., finite descriptions of infinite languages)
  - suitable for use in the implementation of practical syntax analysis tools (lexical analyzers, parsers, …)
- **Formal language theory provides a rigorous foundation for specifying syntax. In particular:**
  - Regular languages are well-suited to describing lexical syntax (what sequences of characters are valid tokens?)
  - Context-free languages are well-suited to describing grammatical structure (what sequences of tokens are valid expressions, statements, programs? and what AST structure do they correspond to?)

Formal languages

- Pick a set, A, of **symbols**, which we refer to as the **alphabet**
  - For lexical analysis, "symbols" are typically characters
  - For parsing, "symbols" are typically tokens
- The set of all finite strings of symbols taken from A is written as A*
- A **language** (over A) is a subset of A*

Examples

- If A = { 0, 1 }, then:

  A* = { "", "0", "1", "00", "01", "10", "11", "000", ... }

- Bytes form a finite language over A:

  Bytes = { $b_0\, b_1\, b_2\, b_3\, b_4\, b_5\, b_6\, b_7 \mid b_i \in \{0,1\}$ }

- Even length bitstreams form an infinite language over A

  Evens = {"", "00", "01", "10", "11"} ∪ { x y | x, y ∈ Evens }

string concatenation

English as a formal language

- With alphabet A = { "a", "b", "c", ... , "z" }:
  - A* = all strings with zero or more letters from A
  - Words = the (finite) subset of A* containing only those strings that are valid words in English e.g., "language" $\in$ Words, "jubmod" $\notin$ Words
  - English = the (infinite) subset of Words* containing only those strings that are valid sentences in English, e.g., ["languages", "are", "interesting"] $\in$ English
- But how do we specify these subsets?

Typical programming language as a formal language

- With alphabet A = ASCII or Unicode character set

  A* = all strings with zero or more letters from A

  Tokens = Keywords: `for if int public` ...
  Literals: `42 6.1e2 "abc" true` ...
  Operators: `+ - = == +=` ...
  Separators: `( ) { } ; .` ...
  Identifiers: all nonempty subsets of A* containing only letters (e.g.)

  ProgLang = the subset of Tokens* corresponding to valid programs

- But how do we specify these details?

Wanted!

- **We want methods for describing language syntax that are:**
  - clear, precise, and unambiguous
  - expressive (e.g., finite descriptions of infinite languages)
  - suitable for use in the implementation of practical syntax analysis tools (lexical analyzers, parsers, …)

- **Formal language theory provides a rigorous foundation for specifying syntax. In particular:**
  - Regular languages are well-suited to describing lexical syntax (what sequences of characters are valid tokens?)
  - Context-free languages are well-suited to describing grammatical structure (what sequences of tokens are valid expressions, statements, programs? and what AST structure do they correspond to?)

Regular Expressions

Regular expressions

- A widely used notation for describing simple patterns in text strings

- Notably useful for describing the lexical structure (format of tokens) of programming languages

  - Especially tokens corresponding to infinite sets, such as numeric literals and identifiers

- A set of strings matching a regular expression is called a regular language

- Every finite language is regular, as well as many infinite languages

Regular expressions (regexps)

Expression	Meaning
ε	**Empty**: matches the empty string
c	**Constant**: matches the single character **c**
$r_1 \mid r_2$	**Alternatives**: matches text matching r_1 or text matching r_2
$r_1\ r_2$	**Sequencing**: matches text matching r_1 followed by text matching r_2
r*	**Repetition**: matches a sequence of zero or more items, each of which matches **r**
(r)	**Grouping**: matches text matching **r**

Programming language examples

Regular expression	Describes
-	the operator **-**
if	the keyword **if**
for|if|int|…|public	(finite) set of all keywords
(0|1|2|3|4|5|6|7|8|9)*	(infinite) set of all decimal integer literals

Regular expression shorthands

Expression	Meaning
`r+`	**Repetition**: a sequence of one or more items, each of which matches `r`. `r+` = `rr*`.
`r?`	**Optional**: optional text matching `r`. `r?` = `(r | ε)`.
`[abc]`	**Character classes**: short for `(a|b|c)` Also allows ranges of characters. e.g., `[a-zA-Z]`.
`[^abc]`	**Negation character classes**: any character not in the specified class
`.`	**Wildcard**: matches any character (except newline)

These are derived forms (or "syntactic sugar") that can be constructed from the basic operations already shown

More programming language examples

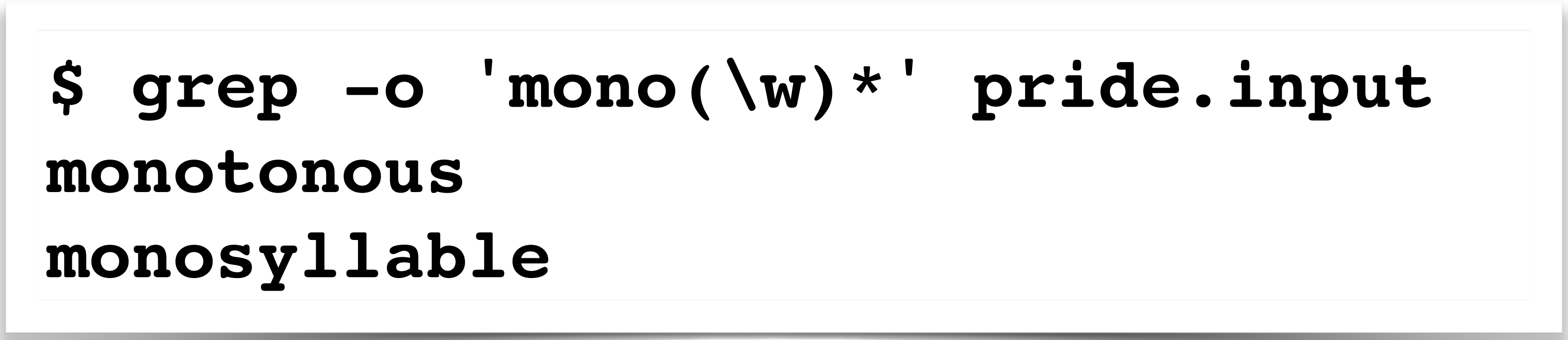
Regular expression	Describes
[A-Za-z$_][A-Za-z0-9$_]*	Java-style identifiers
0|[1-9][0-9]*	decimal integer literals (without leading zeros)
[\t\n]*	whitespace
#.*	Python-style comment
"[^"\n]*"	C string literals (simplified subset)
([0-9]*\.[0-9]+)|([0-9]+\.)	C floating point literals (simplified subset)

Other common uses of regular expressions

- Pattern matching primitives or libraries
  - in Perl, Ruby, JavaScript, Python, Java, C++, ...
  - Often with many extended features

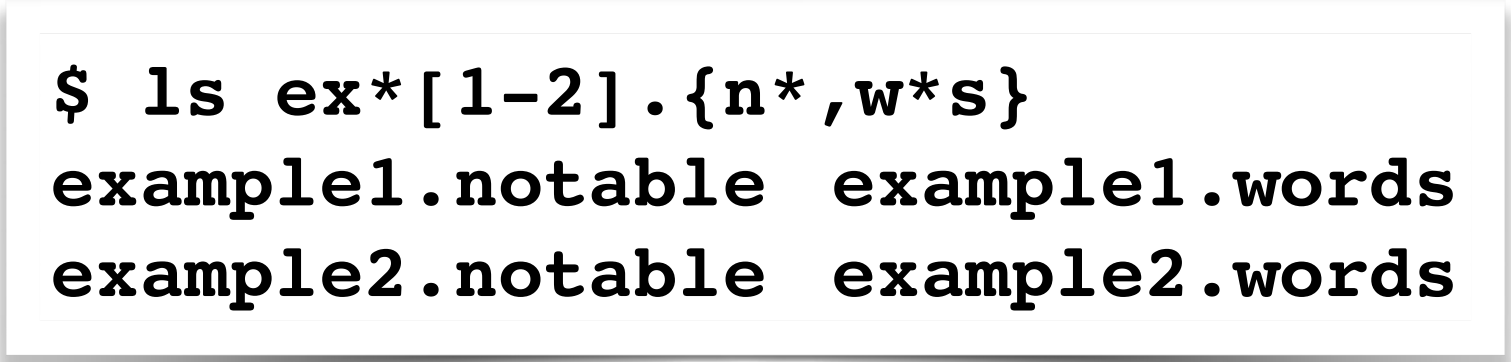
particularly useful in "scripting" paradigm

- Utility commands, e.g. unix grep:



```
$ grep -o 'mono(\w)*' pride.input
monotonous
monosyllable
```

- Filename expansion in unix shell ("globbing")
  - Concrete syntax is different, but concepts are similar



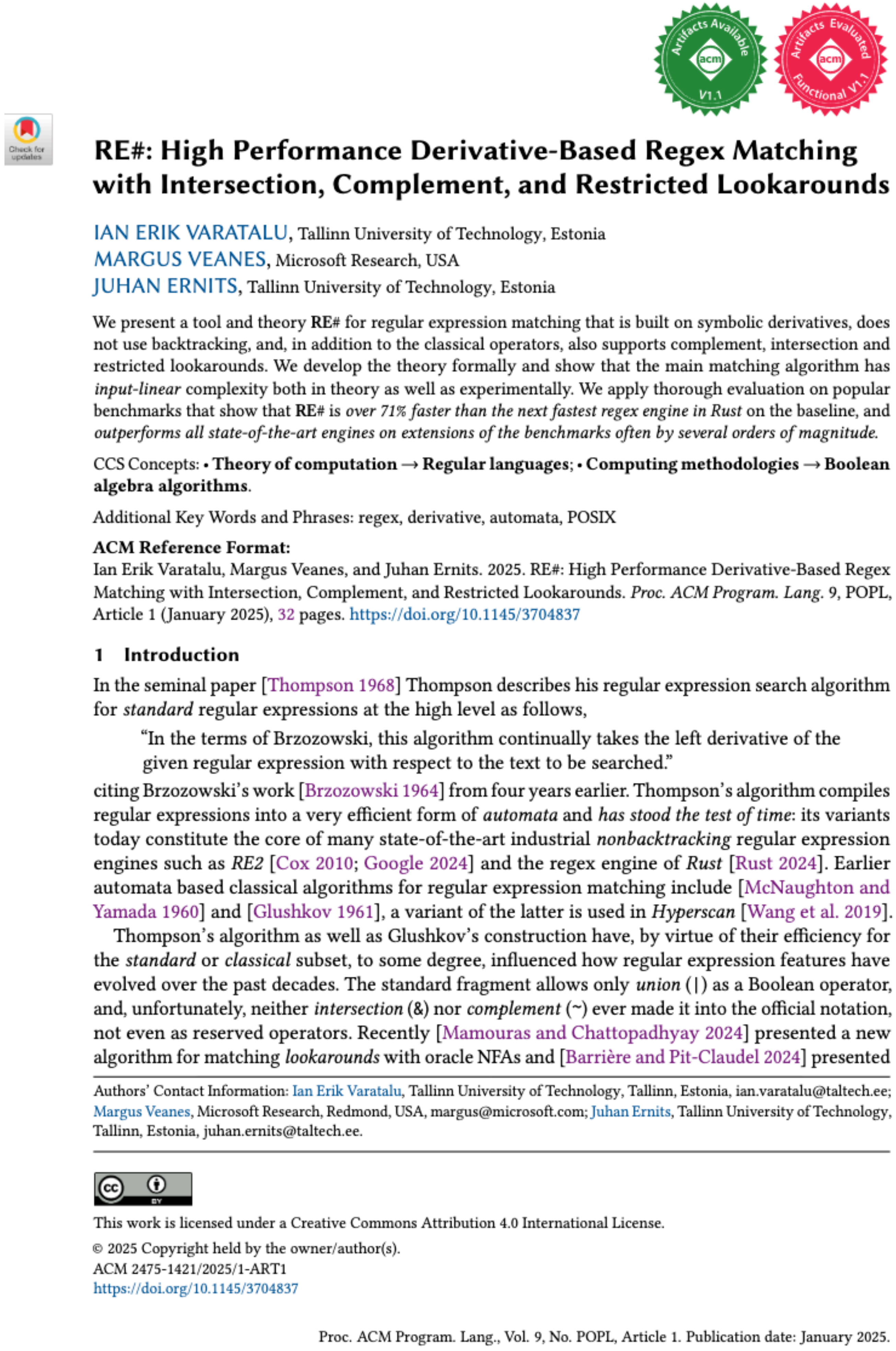
```
$ ls ex*[1-2].{n*,w*s}
example1.notable  example1.words
example2.notable  example2.words
```

`https://regex101.com/`

- Hexadecimal constants: `0x1fff` `0X03` etc.
- Add optional exponents to C floating constants: `6.02e+23` `0.04E2` `2e-80` etc.
- Challenge: C block comments
  - Start with `/*` and end with first `*/`
  - Can contain any characters, including newlines
- Bigger Challenge: Standard ML block comments
  - Similar, but can be nested
  - Start with `(*` and end with matching `*)`

Regular Expressions: Research continues…

ACM POPL 2025 Conference (Principles of Programming Languages)



RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds

IAN ERIK VARATALU, Tallinn University of Technology, Estonia
MARGUS VEANES, Microsoft Research, USA
JUHAN ERNITS, Tallinn University of Technology, Estonia

We present a tool and theory **RE#** for regular expression matching that is built on symbolic derivatives, does not use backtracking, and, in addition to the classical operators, also supports complement, intersection and restricted lookarounds. We develop the theory formally and show that the main matching algorithm has *input-linear* complexity both in theory as well as experimentally. We apply thorough evaluation on popular benchmarks that show that **RE#** is *over 71% faster than the next fastest regex engine in Rust* on the baseline, and *outperforms all state-of-the-art engines on extensions of the benchmarks often by several orders of magnitude.*

CCS Concepts: • **Theory of computation → Regular languages**; • **Computing methodologies → Boolean algebra algorithms**.

Additional Key Words and Phrases: regex, derivative, automata, POSIX

ACM Reference Format:
Ian Erik Varatalu, Margus Veanes, and Juhan Ernits. 2025. RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds. *Proc. ACM Program. Lang.* 9, POPL, Article 1 (January 2025), 32 pages. https://doi.org/10.1145/3704837

1 Introduction

In the seminal paper [Thompson 1968] Thompson describes his regular expression search algorithm for *standard* regular expressions at the high level as follows,

> "In the terms of Brzozowski, this algorithm continually takes the left derivative of the given regular expression with respect to the text to be searched."

citing Brzozowski's work [Brzozowski 1964] from four years earlier. Thompson's algorithm compiles regular expressions into a very efficient form of *automata* and *has stood the test of time*: its variants today constitute the core of many state-of-the-art industrial *nonbacktracking* regular expression engines such as *RE2* [Cox 2010; Google 2024] and the regex engine of *Rust* [Rust 2024]. Earlier automata based classical algorithms for regular expression matching include [McNaughton and Yamada 1960] and [Glushkov 1961], a variant of the latter is used in *Hyperscan* [Wang et al. 2019].

Thompson's algorithm as well as Glushkov's construction have, by virtue of their efficiency for the *standard* or *classical* subset, to some degree, influenced how regular expression features have evolved over the past decades. The standard fragment allows only *union* (|) as a Boolean operator, and, unfortunately, neither *intersection* (&) nor *complement* (~) ever made it into the official notation, not even as reserved operators. Recently [Mamouras and Chattopadhyay 2024] presented a new algorithm for matching *lookarounds* with oracle NFAs and [Barrière and Pit-Claudel 2024] presented

Authors' Contact Information: Ian Erik Varatalu, Tallinn University of Technology, Tallinn, Estonia, ian.varatalu@taltech.ee; Margus Veanes, Microsoft Research, Redmond, USA, margus@microsoft.com; Juhan Ernits, Tallinn University of Technology, Tallinn, Estonia, juhan.ernits@taltech.ee.

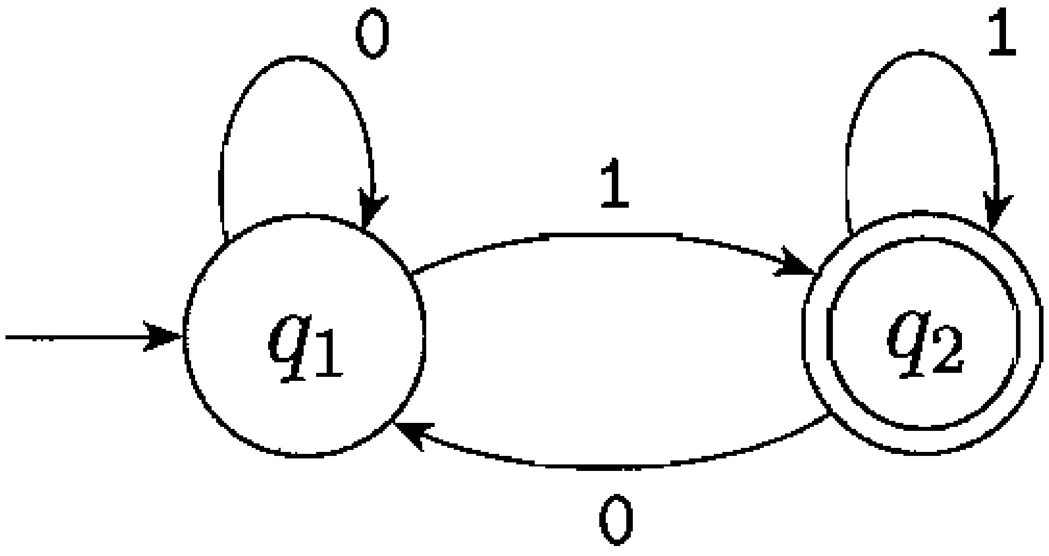
ACM 2475-1421/2025/1-ART1
https://doi.org/10.1145/3704837

Implementing Regular Expression matching

- Regular expression matching can be coded up in a typical programming language using simple conditionals and loops, using a bounded amount of memory

- More abstractly, it turns out that every regular language can be recognized by a finite automaton (or finite state machine)

  - E.g. this machine recognizes the language over A = {0,1} defined by the regular expression (0|1)*1



- These concepts and the proofs that regular expressions and finite automata have equal descriptive power are studied in depth in CS311

Beyond regular languages

- **Regular expressions work really well for describing lexical syntax …**
  - In fact, we only need their full power to describe a few kinds of tokens (the syntax of many tokens is trivial)
- **So perhaps we can use also use regular expressions to describe the large-scale syntax of expressions and programs?**
  - Unfortunately, only very limited kinds of languages are regular languages...

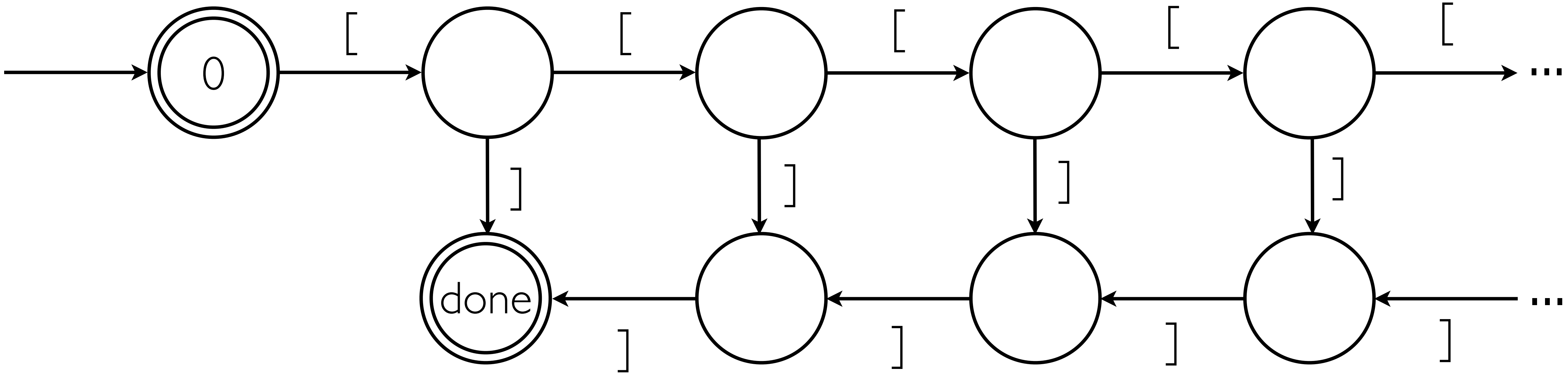
A simple language of Brackets

- Brackets = { "" } ∪ { [b] | b ∈ Brackets }
- So the "words" in Brackets are:

  "", "[ ]", "[[ ]]", "[[[ ]]]", "[[[[ ]]]]", "[[[[[ ]]]]]", …
- In other words, nested pairs of bracket characters:
  - a sequence of n open brackets …
  - … followed by exactly n close brackets
- A subset of any language that uses parentheses/brackets
- But is it a regular language?

Factoid: Brackets is not regular

- If Brackets were regular, then we could recognize it using a **finite** automaton that would look something like this:



- If s_n is the state that we can reach after n open brackets and n≠m, then $s_n \neq s_m$
- So this machine must have infinitely many states (at least one distinct state s_n for each n)
- So Brackets cannot be regular
- Informally, regular languages "can't keep count"

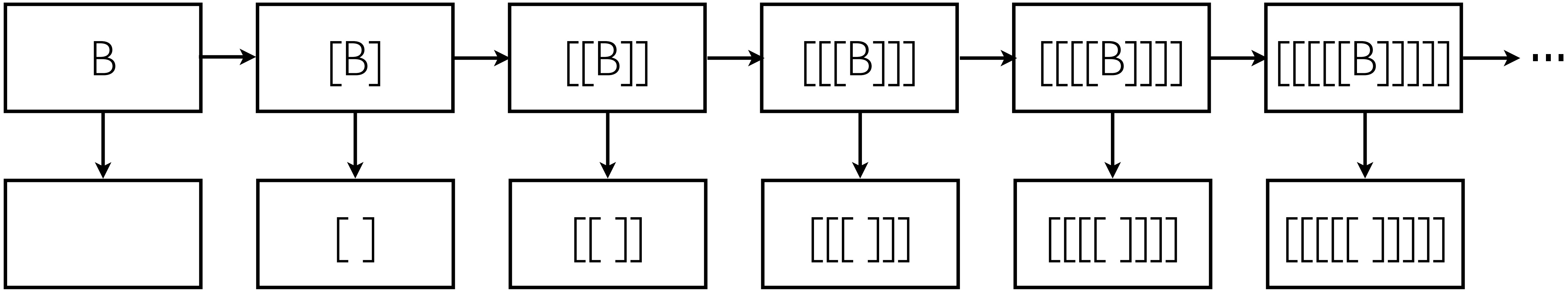
Iteration vs recursion

- Regular expressions don't allow recursion, just (unbounded) iteration
- But it is easy enough to give a simple recursive characterization for B ∈ Brackets:

B ⟶ meaning: B is empty

B ⟶ [B] meaning: B consists of an initial [symbol, another element from Brackets; and a closing] symbol

Generating brackets



- We have two rewrite rules:

B	⟶		replace a B with the empty string
B	⟶	[B]	replace a B with the string [B]

- Either rule can be used to rewrite an occurrence of B

- We say that "B **derives** s" if the string s can be obtained from B by repeated rewriting/replacement

Wanted!

- **We want methods for describing language syntax that are:**
  - clear, precise, and unambiguous
  - expressive (e.g., finite descriptions of infinite languages)
  - suitable for use in the implementation of practical syntax analysis tools (lexical analyzers, parsers, …)

- **Formal language theory provides a rigorous foundation for specifying syntax. In particular:**
  - Regular languages are well-suited to describing lexical syntax (what sequences of characters are valid tokens?)
  - Context-free languages are well-suited to describing grammatical structure (what sequences of tokens are valid expressions, statements, programs? and what AST structure do they correspond to?)

Context-free Grammars and Languages

Context-free grammars (CFGs)

A context-free grammar G = (T , N , P , S) consists of

a set T of **terminal** symbols (“tokens”)

a set N of **nonterminal** symbols

a set P of **productions**, each of which is a rule of the form n ⟶ w where n ∈ N, and w ∈ (T ∪ N)*

a **start symbol** S ∈ N

Example

- A CFG for Brackets: ({ [,] }, { B }, { B ⟶ , B ⟶ [B] }, B)

- In practice, it is often sufficient just to write down the productions for a CFG:

  the sets of terminals and nonterminals can usually be inferred from the productions

  the start symbol can either be identified explicitly, or assumed as the first nonterminal

- For example, we can describe Brackets by:

  B ⟶
  B ⟶ [B]

Example: a CFG for Boolean Expressions

E → E || E
E → E && E
E → ! E
E → true
E → false
E → let id = E in E
E → id
E → (E)

{ ||, &&, !, true, false, let, =, in, id }

{E} E

represents an infinite set of possible identifiers (names)

More examples

Arithmetic:

$E \longrightarrow \texttt{n}$
$E \longrightarrow (E)$
$E \longrightarrow E + E$
$E \longrightarrow E * E$

Regexps:

$R \longrightarrow \texttt{c}$
$R \longrightarrow \varepsilon$
$R \longrightarrow R\ R$
$R \longrightarrow R \mid R$
$R \longrightarrow R *$

Statements:

$S \longrightarrow \texttt{id := } E$
$S \longrightarrow \texttt{if } E \texttt{ then } S\ T$
$S \longrightarrow \texttt{while } E \texttt{ do } S$
$S \longrightarrow S \texttt{ ; } S$
$T \longrightarrow \texttt{else } S$
$T \longrightarrow$

CFGs:

$G \longrightarrow P$
$G \longrightarrow P\ G$
$P \longrightarrow \texttt{n -> } W$
$W \longrightarrow$
$W \longrightarrow \texttt{n } W$
$W \longrightarrow \texttt{t } W$

a "meta-level"grammar!

Context-free grammars and languages

- What is the relationship between context-free grammars and languages (i.e., sets of strings)?

- A **derivation** is a sequence of strings:

  $$s_1 \longrightarrow s_2 \longrightarrow s_3 \longrightarrow s_4 \longrightarrow s_5 \longrightarrow s_6 \longrightarrow s_7 \longrightarrow \ldots$$

  in which each string s_{i+1} is obtained from the previous string s_i by choosing a production $n \longrightarrow w$ and replacing an occurrence of n in s_i with w

- In this case, we say that s_1 **derives** s_i for each i = 1, 2, ...

- We say that a context-free grammar G = (T, N, P, S) **generates** the language that contains all strings in T* that can be derived from S

Brackets is a "context-free language"

- Any language that is generated from a context-free grammar is said to be a **context-free language**
- Sample derivations for Brackets:

B ⟶

B ⟶ [B] ⟶ [ ]

B ⟶ [B] ⟶ [[B]] ⟶ [[ ]]

…

B ⟶ [B] ⟶ [[B]] ⟶ [[[B]]] ⟶ [[[[B]]]] ⟶ [[[[ ]]]]

…

Derivations and parse trees

A language of arithmetic expressions

- Many computer languages are naturally described as context-free languages (i.e., using a context-free grammar)
- Example: a simple language of expressions:

  E ⟶ n (**n** is an integer literal)

  E ⟶ E + E

  E ⟶ E * E

  E ⟶ (E)

- Terminology:

  E is a nonterminal

  +, *****, **(**, **)**, and **n** are terminals (i.e., tokens)

Deriving expressions

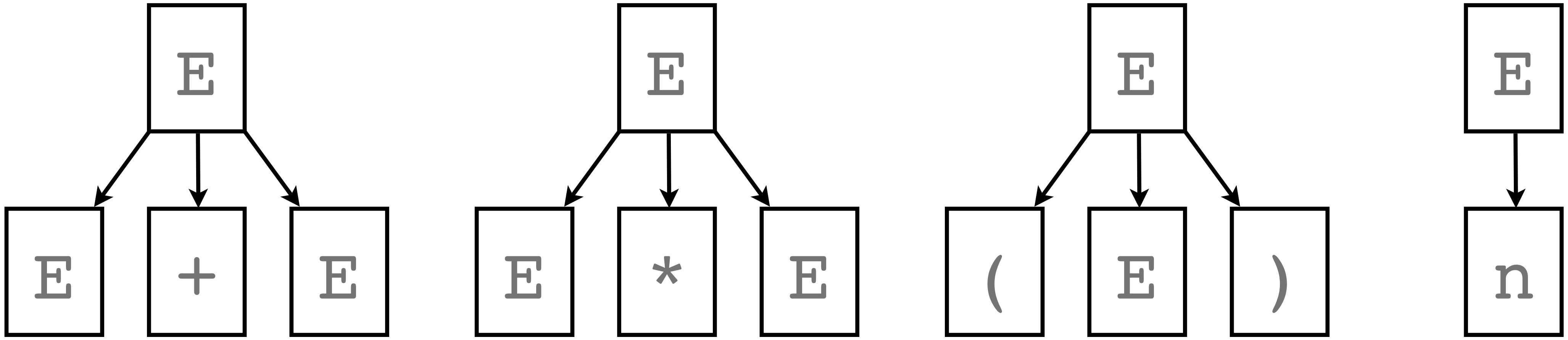
$$\begin{aligned} E &\longrightarrow n \\ E &\longrightarrow E + E \\ E &\longrightarrow E * E \\ E &\longrightarrow (E) \end{aligned}$$

For example, 1+(2*3) is an expression, as witnessed by this derivation:

$$\begin{aligned} E &\longrightarrow E + E \\ &\longrightarrow E + (E) \\ &\longrightarrow E + (E * E) \\ &\longrightarrow E + (E * 3) \\ &\longrightarrow E + (2 * 3) \\ &\longrightarrow 1 + (2 * 3) \end{aligned}$$

Productions in graphical form

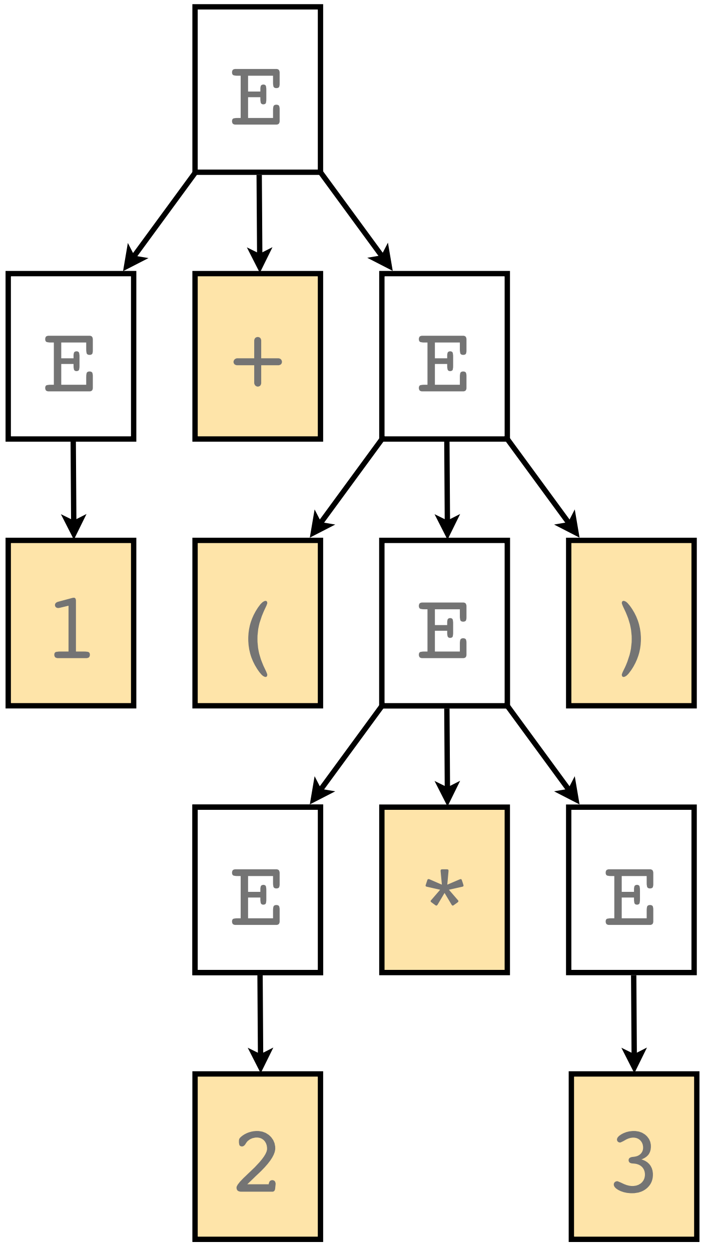
To understand the essential structure of a derivation, we will use a graphical notation to represent the productions in a grammar:



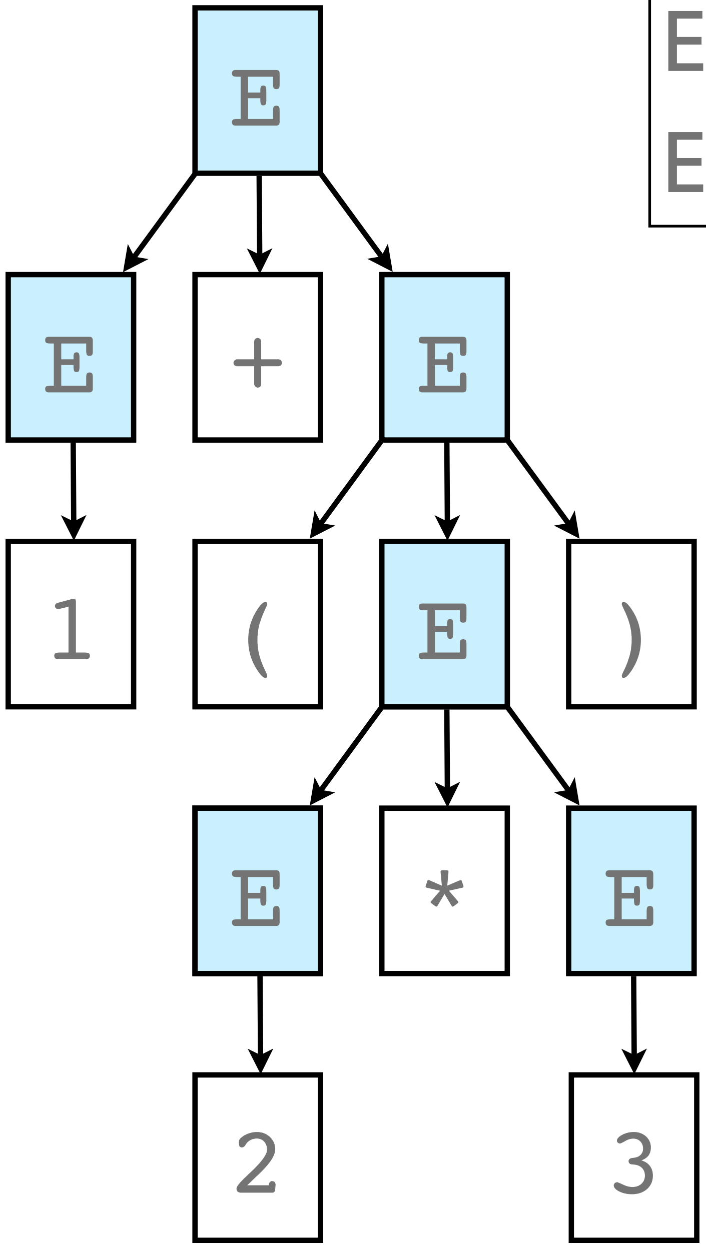
We can glue together productions to give a parse tree corresponding to a derivation

Parse tree for 1+(2*3)

E ⟶ n
E ⟶ E + E
E ⟶ E * E
E ⟶ (E)



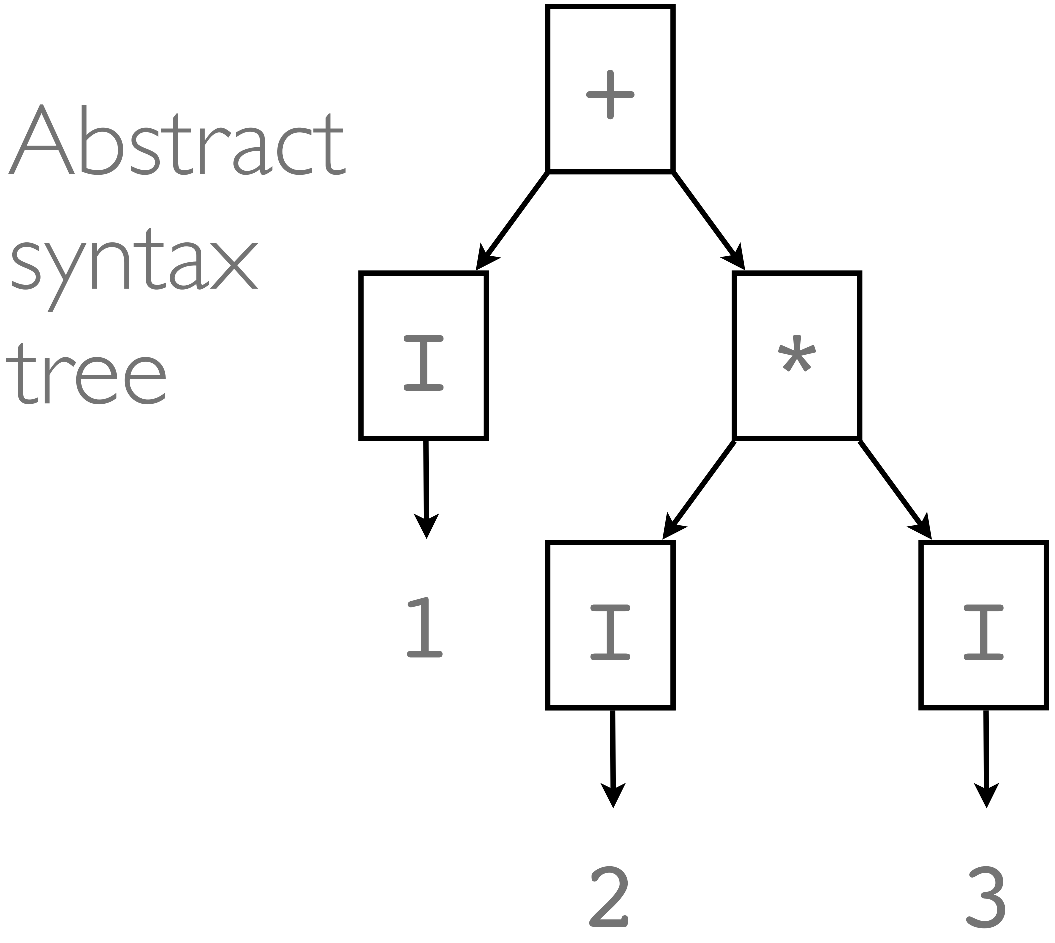
leaves are terminals, every terminal appears

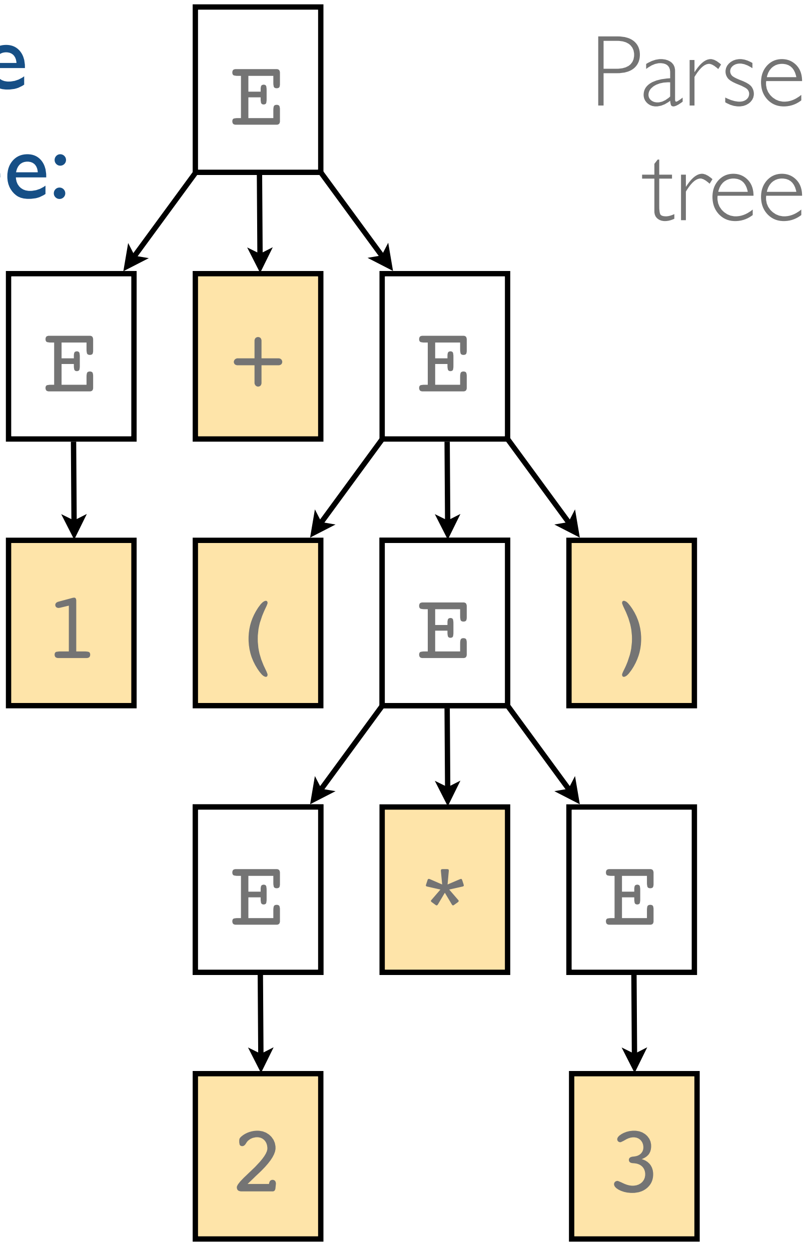


interior nodes correspond to productions

Parse trees vs abstract syntax trees

Because it describes a full derivation and all of the tokens in the input, there is usually more information in a parse tree than we need for an abstract syntax tree:





Alternative derivations

$$\begin{aligned} E &\longrightarrow n \\ E &\longrightarrow E + E \\ E &\longrightarrow E * E \\ E &\longrightarrow (E) \end{aligned}$$

1+(2*3) has more than one possible derivation

$$\begin{aligned} E &\longrightarrow E + E \\ &\longrightarrow E + (E) \\ &\longrightarrow E + (E * E) \\ &\longrightarrow E + (E * 3) \\ &\longrightarrow E + (2 * 3) \\ &\longrightarrow 1 + (2 * 3) \end{aligned}$$

original derivation

$$\begin{aligned} E &\longrightarrow E + E \\ &\longrightarrow 1 + E \\ &\longrightarrow 1 + (E) \\ &\longrightarrow 1 + (E * E) \\ &\longrightarrow 1 + (2 * E) \\ &\longrightarrow 1 + (2 * 3) \end{aligned}$$

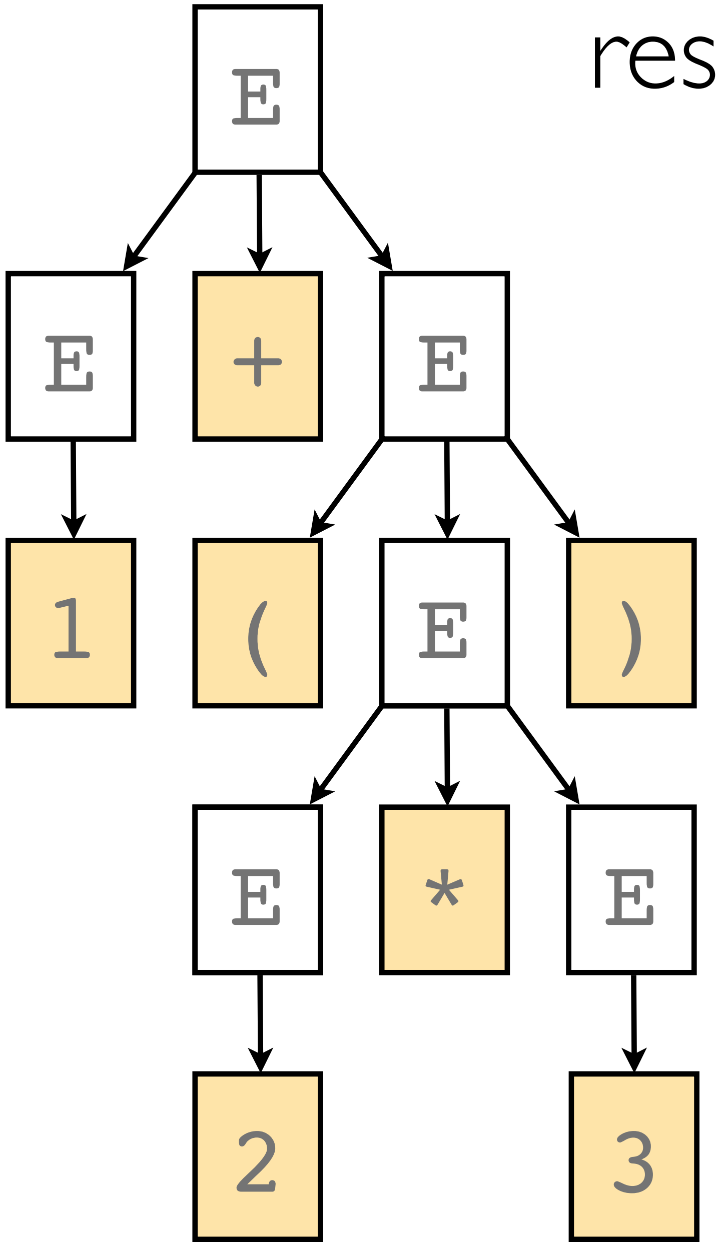
alternative derivation

Multiple derivations

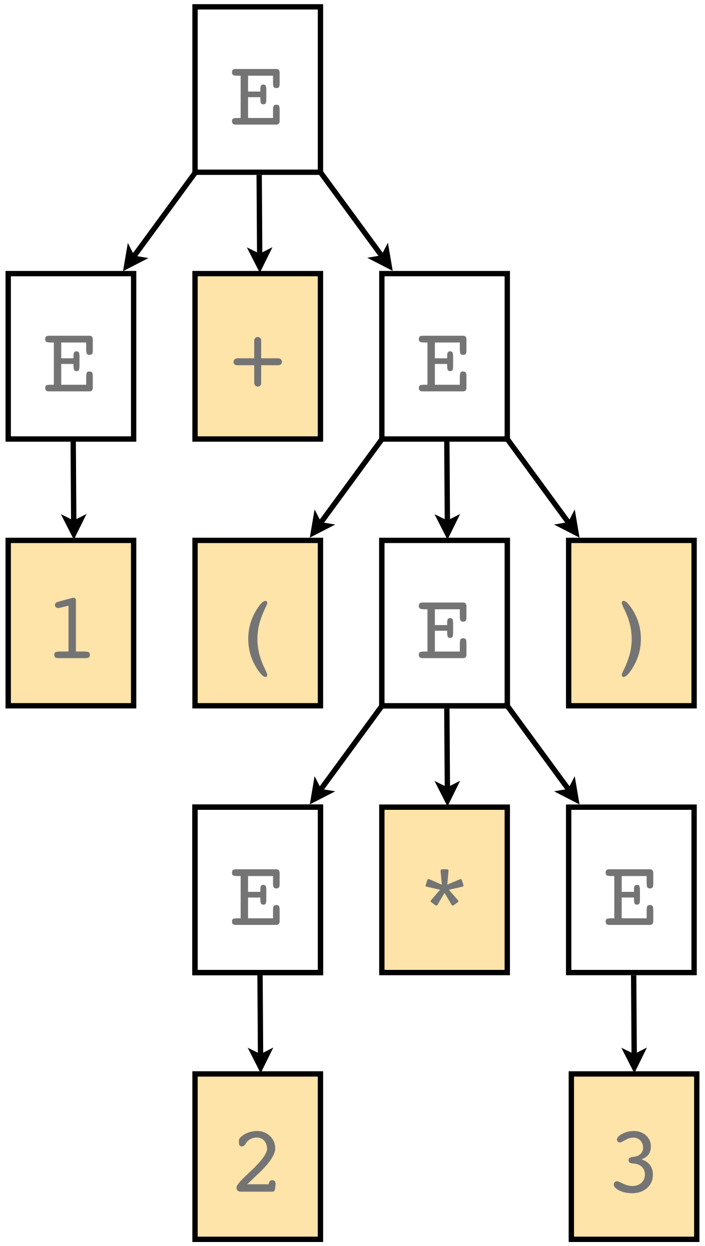
- In a right-most derivation, we replace the right-most nonterminal at each step
- In a left-most derivation, we replace the left-most nonterminal at each step
- There are many other, less systematic, derivation orders too
- Does it matter which derivation order we use?
- In general, no: what matters is the parse tree we get

1+(2*3)

for this example, end result is the same!



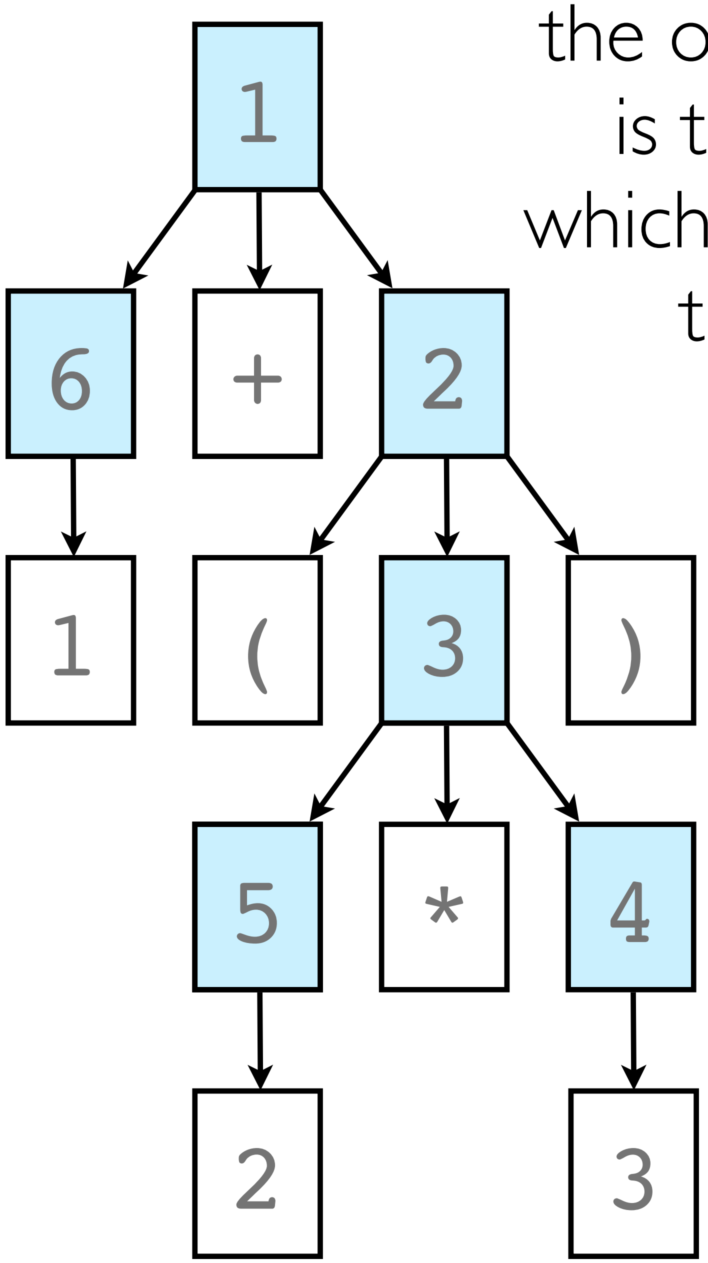
right-most



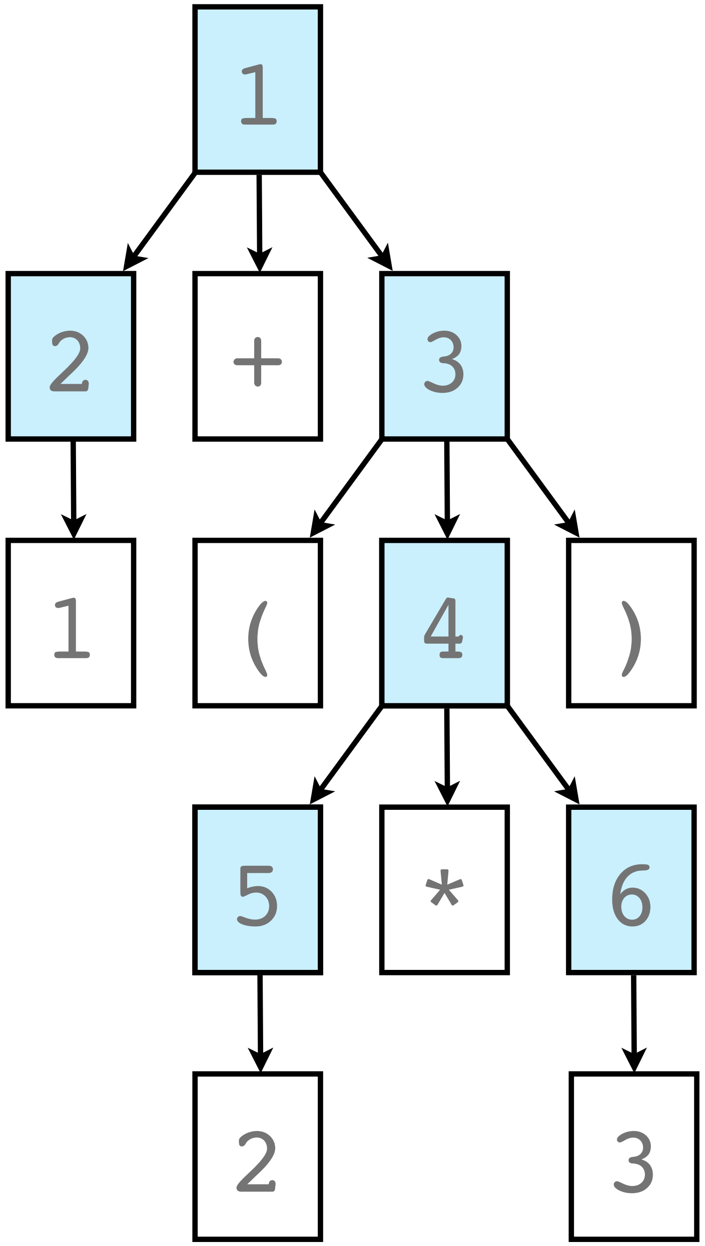
left-most

Right-most vs left-most

the only difference
is the order in
which we construct
the nodes



right-most



left-most

But derivation choices can matter

$$
\begin{aligned}
E &\longrightarrow n\\
E &\longrightarrow E + E\\
E &\longrightarrow E * E\\
E &\longrightarrow (E)
\end{aligned}
$$

Different production

For example, 1 + 2 * 3 is an expression (no parentheses):

$$
\begin{aligned}
E &\longrightarrow E + E\\
&\longrightarrow E + E * E\\
&\longrightarrow E + E * 3\\
&\longrightarrow E + 2 * 3\\
&\longrightarrow 1 + 2 * 3
\end{aligned}
$$

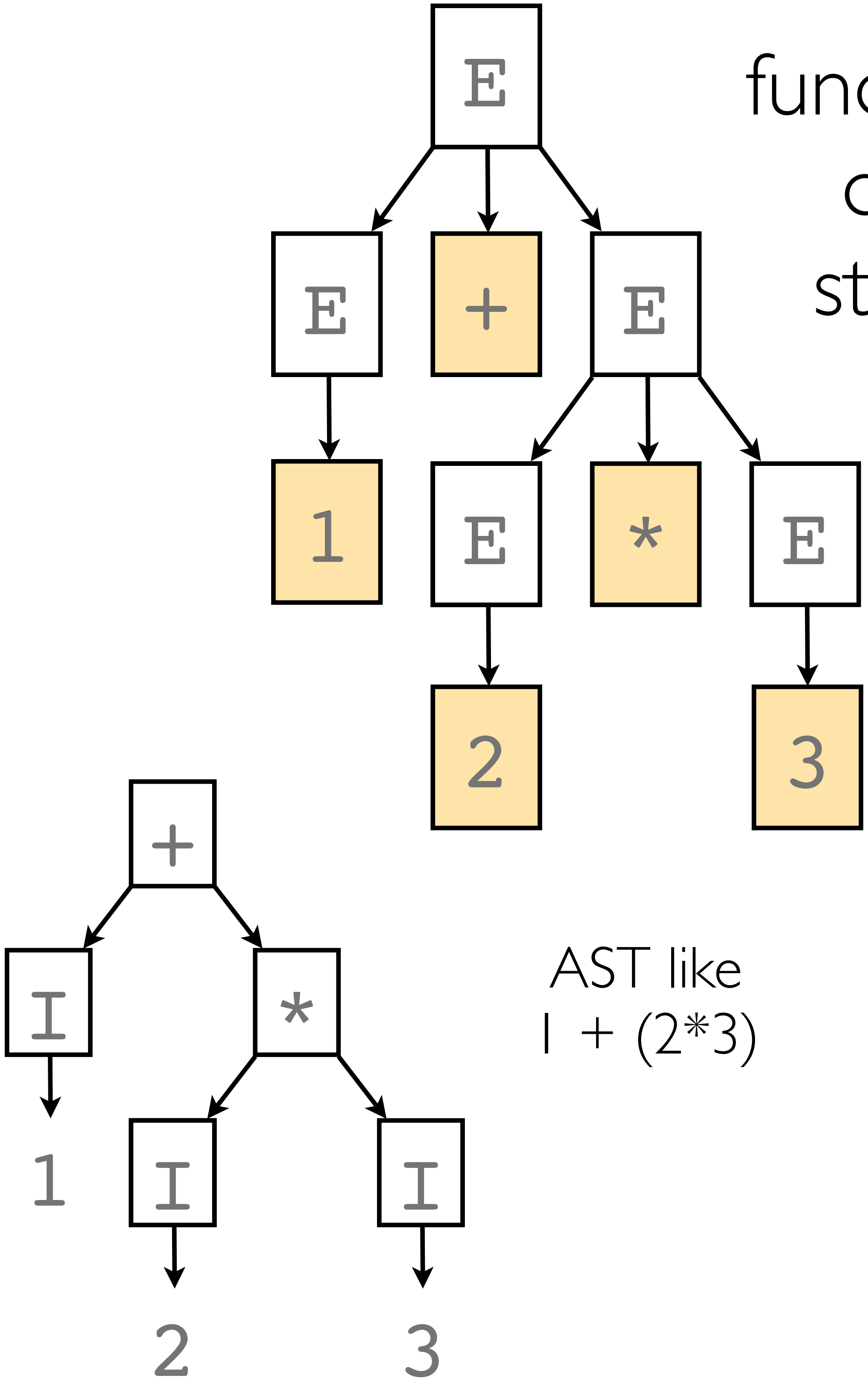
$$
\begin{aligned}
E &\longrightarrow E * E\\
&\longrightarrow E * 3\\
&\longrightarrow E + E * 3\\
&\longrightarrow E + 2 * 3\\
&\longrightarrow 1 + 2 * 3
\end{aligned}
$$

two valid derivations
(both right-most!),
but a fundamental difference this time

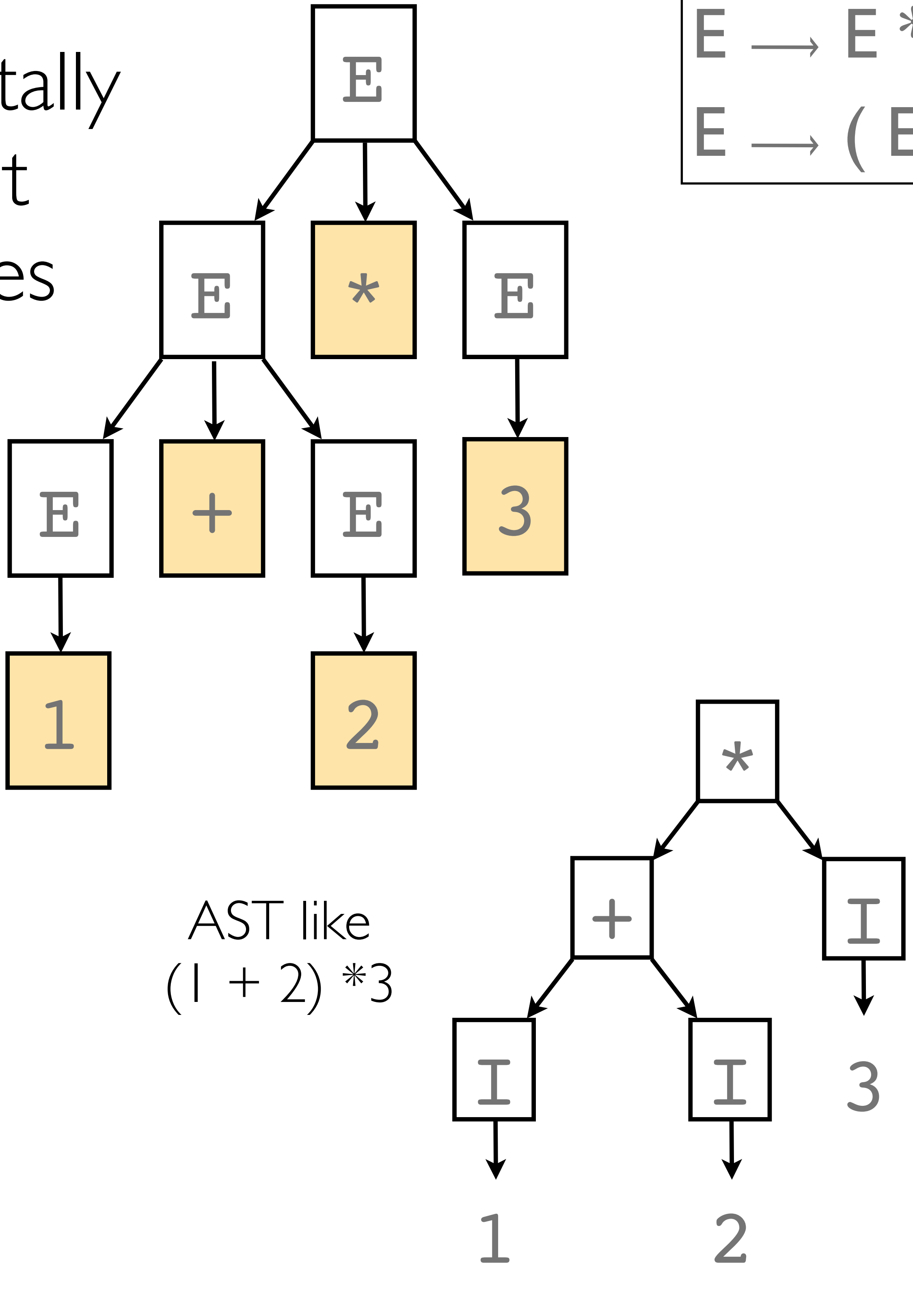
1+2*3

E ⟶ n
E ⟶ E + E
E ⟶ E * E
E ⟶ (E)

fundamentally different structures



AST like
1 + (2*3)



AST like
(1 + 2) *3

CFGs and parse trees

- Context-free grammars don't just define languages (i.e., they don’t just derive *sets of strings*) ...
- ... they actually define *sets of trees*!
- The strings in the corresponding context-free language can be recovered from the leaf nodes of each tree
- Parsing works in reverse: start with a string and try to construct the original tree
- What happens in there are <u>distinct</u> tree structures with the <u>same</u> sequence of symbols on their leaves?

Ambiguity

- A grammar is **ambiguous** if there is a string in the corresponding language with more than one parse tree
- Example: Our grammar for expressions is ambiguous because the string "1+2*3" has two distinct parse trees

  (We could find plenty of other examples of the same problem in this grammar, but finding just one is enough to demonstrate ambiguity)

- Ambiguity is a property of a grammar, NOT a language

  We can have multiple grammars describing the same language, some ambiguous and some unambiguous

Dealing with ambiguity

- Does it matter?

- **If any parse tree is as good as any other (i.e., they all have the same meaning), then just take whichever tree we get**

  Example: for regular expressions, $r_1(r_2r_3)$ matches exactly the same set of strings as $(r_1r_2)r_3$, so we can parse $r_1r_2r_3$ either way

- **If different trees have different meanings, then we need to choose between them:**

  Disambiguating rules (e.g., operator precedence)

  Rewrite the grammar to avoid ambiguity

Precedence and grouping (associativity)

- If ⊗ has **higher precedence** than ⊕, then a ⊗ b ⊕ c should be parsed in the same way as (a ⊗ b) ⊕ c
- If ⊗ **groups/associates** to the left, then a ⊗ b ⊗ c should be parsed in the same way as (a ⊗ b) ⊗ c
- If ⊗ **groups/associates** to the right, then a ⊗ b ⊗ c should be parsed in the same way as a ⊗ (b ⊗ c)
- If ⊗ is **nonassociative**, then a ⊗ b ⊗ c should be treated as an error

Order of operations

- There are widely used conventions for the precedence of standard arithmetic operations. (e.g., "PEMDAS")
- Parentheses first, then exponentiation, then multiplication and division, then addition and subtraction

  Example: (1+2)+3*4 == (1+2)+(3*4)
- But the final decision about what rules each operator should follow rests with the language designer
- Let's suppose we want to resolve our ambiguities using:

  * has higher precedence than +

  * and + both group to the left

An unambiguous grammar for expressions

Here is an unambiguous grammar for our language of expressions:

E → E + P   expressions
E → P

P → P * A   products
P → A

A → (E)   atoms
A → n   (**n** is an integer literal)

How were these properties met?

- * has higher precedence than +
- * and + both group to the left

An unambiguous grammar for expressions

Here is an unambiguous grammar for our language of expressions:

E → E + P	expressions are sums of products
E → P	
P → P * A	products of atoms
P → A	
A → (E)	atoms
A → n	

An unambiguous grammar for expressions

An expression is a sum of products, each of which is a product of atoms.

Example: if a_1, ..., a_8 are atoms, then:

$$\underbrace{\underbrace{a_1 * a_2 * a_3}_{\text{product}} + \underbrace{a_4 * a_5}_{\text{product}} + \underbrace{a_6}_{\text{product}} + \underbrace{a_7 * a_8}_{\text{product}}}_{\text{expr}}$$

must be parsed as:

$$(a_1 * a_2 * a_3) + (a_4 * a_5) + (a_6) + (a_7 * a_8)$$

Multiplication has been given a higher precedence than addition!

An unambiguous grammar for expressions

The right argument of + must be a product

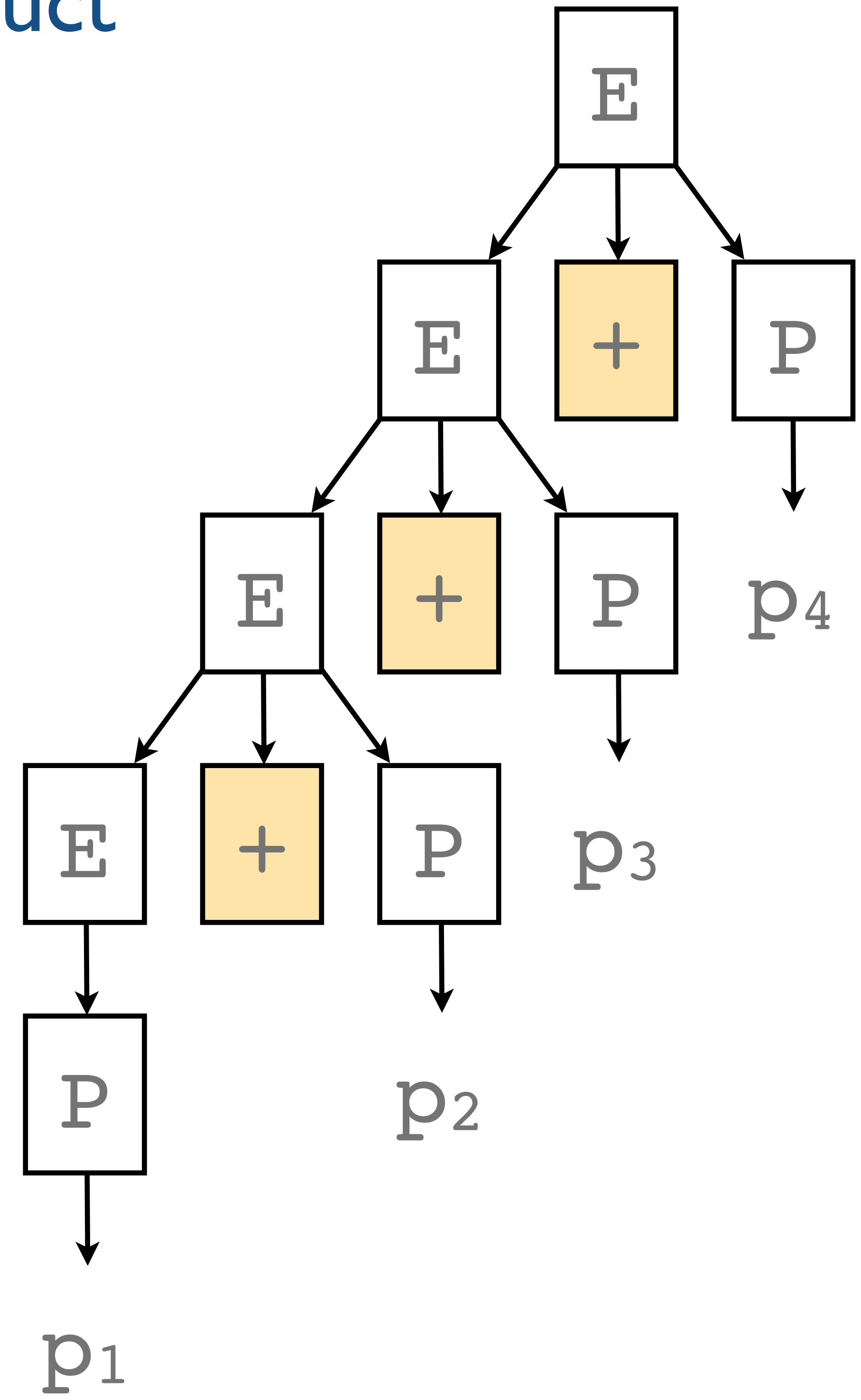
Example: if p_1, ..., p_4 are products, then:

$$\underbrace{\underbrace{\underbrace{\underbrace{\underbrace{p_1}_{\text{prod.}}}_{\text{expr}} + \underbrace{p_2}_{\text{prod.}}}_{\text{expr}} + \underbrace{p_3}_{\text{prod.}}}_{\text{expr}} + \underbrace{p_4}_{\text{prod.}}}_{\text{expr}}$$

must be parsed as:

$$(((p_1 + p_2) + p_3) + p_4)$$

In other words: + groups to the left



Controlling grouping

$$E \rightarrow E + P$$
$$E \rightarrow P$$

recursion on the left results in left grouping

$$P \rightarrow P * A$$
$$P \rightarrow A$$

similarly for *

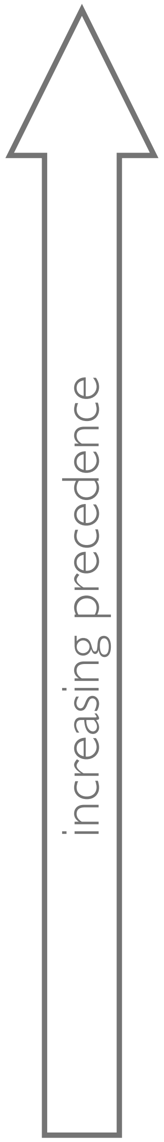
for right grouping, change the recursion

$$E \rightarrow P + E$$
$$E \rightarrow P$$

for nonassociative operators, no recursion on either side

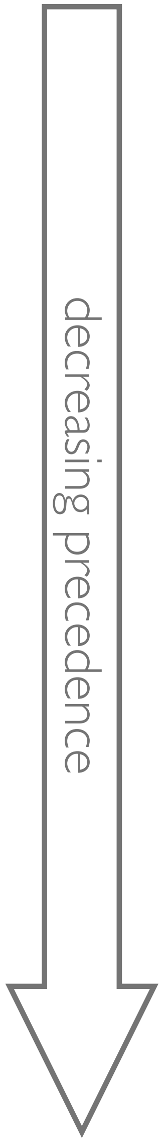
$$P \rightarrow A * A$$
$$P \rightarrow A$$

Fragments from the official Java grammar



```
MultiplicativeExpression:
    UnaryExpression
    MultiplicativeExpression * UnaryExpression
    MultiplicativeExpression / UnaryExpression
    MultiplicativeExpression % UnaryExpression

AdditiveExpression:
    MultiplicativeExpression
    AdditiveExpression + MultiplicativeExpression
    AdditiveExpression - MultiplicativeExpression

ShiftExpression:
    AdditiveExpression
    ShiftExpression << AdditiveExpression
    ShiftExpression >> AdditiveExpression
    ShiftExpression >>> AdditiveExpression

RelationalExpression:
    ShiftExpression
    RelationalExpression < ShiftExpression
    RelationalExpression > ShiftExpression
    RelationalExpression <= ShiftExpression
    RelationalExpression >= ShiftExpression
    RelationalExpression instanceof ReferenceType
```

```
EqualityExpression:
    RelationalExpression
    EqualityExpression == RelationalExpression
    EqualityExpression != RelationalExpression

AndExpression:
    EqualityExpression
    AndExpression & EqualityExpression

ExclusiveOrExpression:
    AndExpression
    ExclusiveOrExpression ^ AndExpression

InclusiveOrExpression:
    ExclusiveOrExpression
    InclusiveOrExpression | ExclusiveOrExpression

ConditionalAndExpression:
    InclusiveOrExpression
    ConditionalAndExpression && InclusiveOrExpression

ConditionalOrExpression:
    ConditionalAndExpression
    ConditionalOrExpression || ConditionalAndExpression

ConditionalExpression:
    ConditionalOrExpression
    ConditionalOrExpression ? Expression : ConditionalExpression
```



Implementing CFG parsers

- Parsing is the problem of determining whether a string is in a given language
- This problem is harder for context-free languages than it is for regular languages.
- Coding a CFG parser requires recursion (or, equivalently, an explicit working stack) of depth O(n) where n = input length
- CFG parsing can require roughly O(n^3) time in general, but most programming language CFGs can be parsed in O(n) time
- Parser generators are often used to generate parser code automatically from a CFG
  - e.g., the Earley parsing algorithm works for all CFGs

(E)BNF

- The format we've used to write down CFGs is common in theoretical descriptions
- Programming languages grammars are often written in a more specialized format called Backus-Naur Form (BNF) or Extended BNF (EBNF)
  - Makes the difference between alphabet symbols and meta-characters more explicit
  - Allows various abbreviations, e.g. the rules could be rewritten as

$$E \longrightarrow E + P$$
$$E \longrightarrow P$$

$E \longrightarrow E + P \mid P$ (multiple alternatives one one line)

$E \longrightarrow [\ E +\ ]\ P$ (optional)

$E \longrightarrow \{\ P +\ \}\ P$ (zero or more repetitions)

Regular Grammars

- Any regular language is also a context free language
- The grammar for a regular language can always be written using just rules of the following two forms:

  A → x (where A,B are non-terminals and
  A → xB x is a terminal)

- So we don't really need REs if we have CFGs. But the RE may be much more compact
- RE-like notations are often used as part of EBNF

Context free languages summary

- **Context free grammars can be used to describe a significantly larger family of languages than regular expressions**
  - including many of the languages we encounter in practice
- **Parse trees are graphical descriptions of derivations that**
  - can reflect the grammatical structure of the input
  - can highlight ambiguities in the grammar
  - include more detail than is typically needed for an AST
- **Grammars can be written so that operators are treated as having different precedence and grouping behaviors**
  - This can help to avoid ambiguity
- **Parsers for CFGs can be generated automatically**