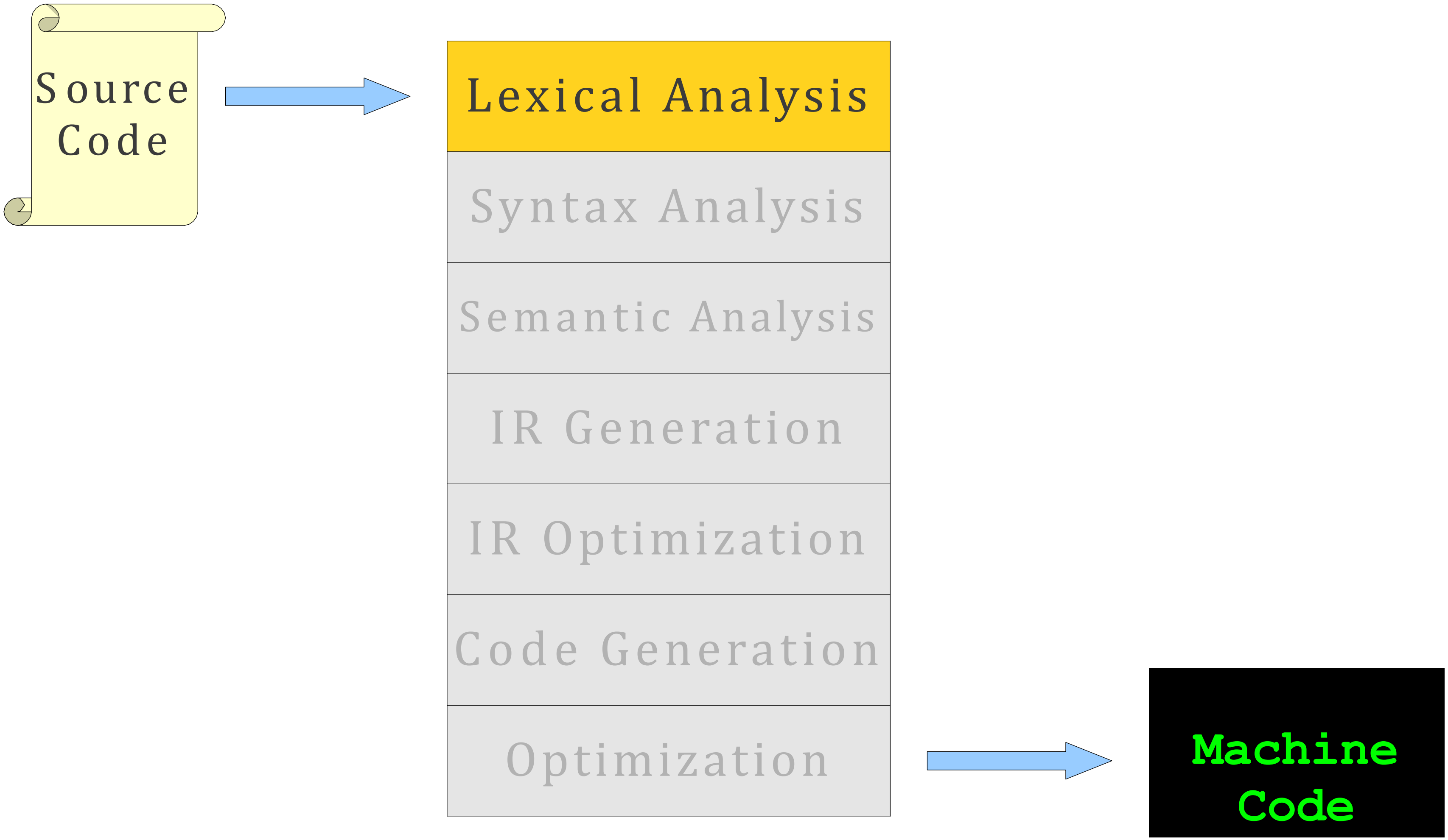
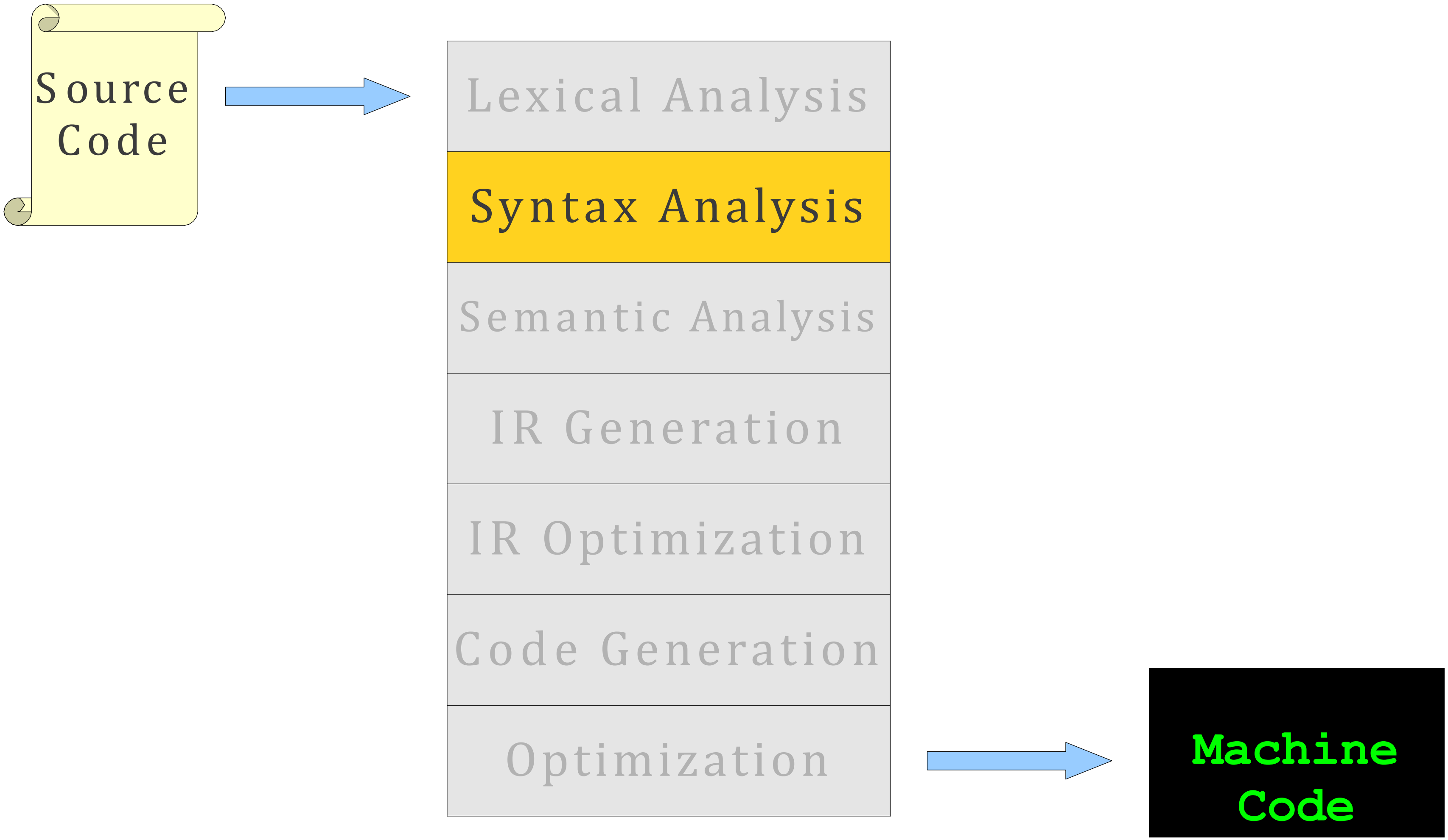
BBM 301 – Programming Languages

Lecture 3

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Where We Are


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

What is Syntax Analysis?

- After lexical analysis (scanning), we have a series of tokens.
- In syntax analysis (or parsing), we want to interpret what those tokens mean.
- Goal: Recover the *structure* described by that series of tokens.
- Goal: Report *errors* if those tokens do not properly encode a structure.

Lexical vs. Syntactic Analysis

Phase	Input	Output
Lexer	Sequence of characters	Sequence of tokens
Parser	Sequence of tokens	Parse tree

- **`Lex`** is a tool for writing lexical analyzers.
- **`Yacc`** is a tool for constructing parsers.

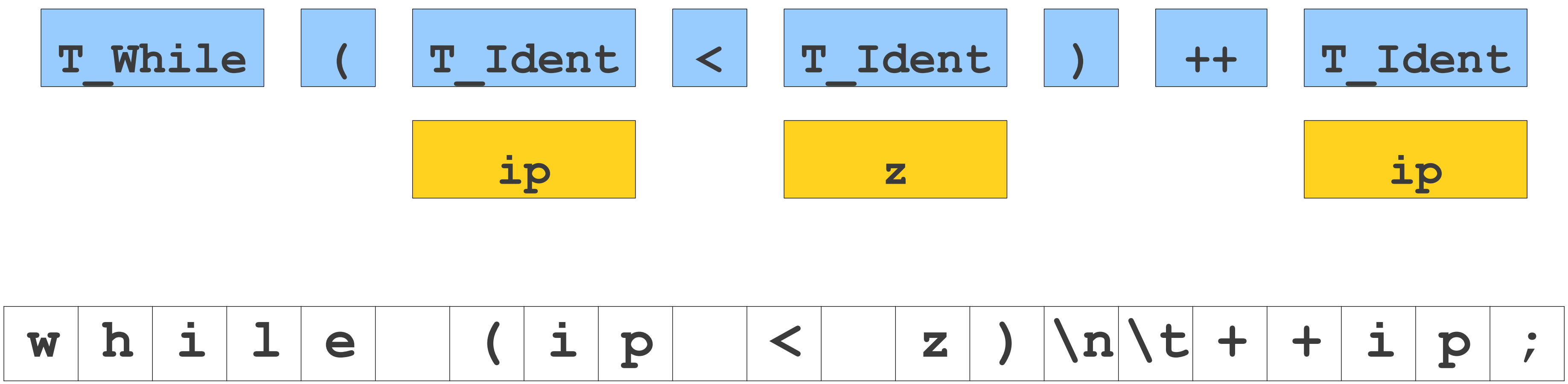
Slide credit: Wes Weimer

5

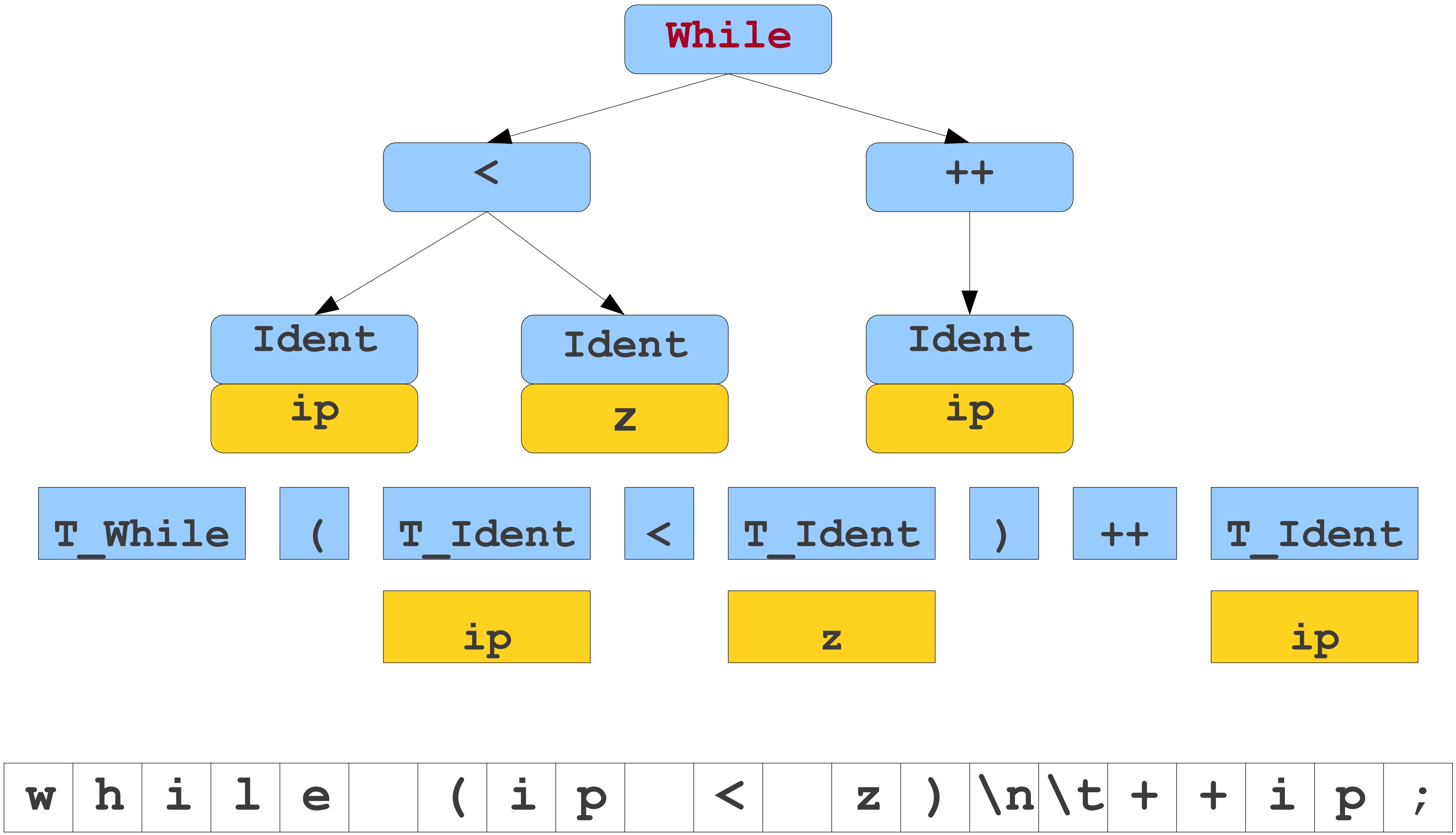
```
while (ip < z)
    ++ip;
```

w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;

```
while (ip < z)
    ++ip;
```




```
while (ip < z)
    ++ip;
```

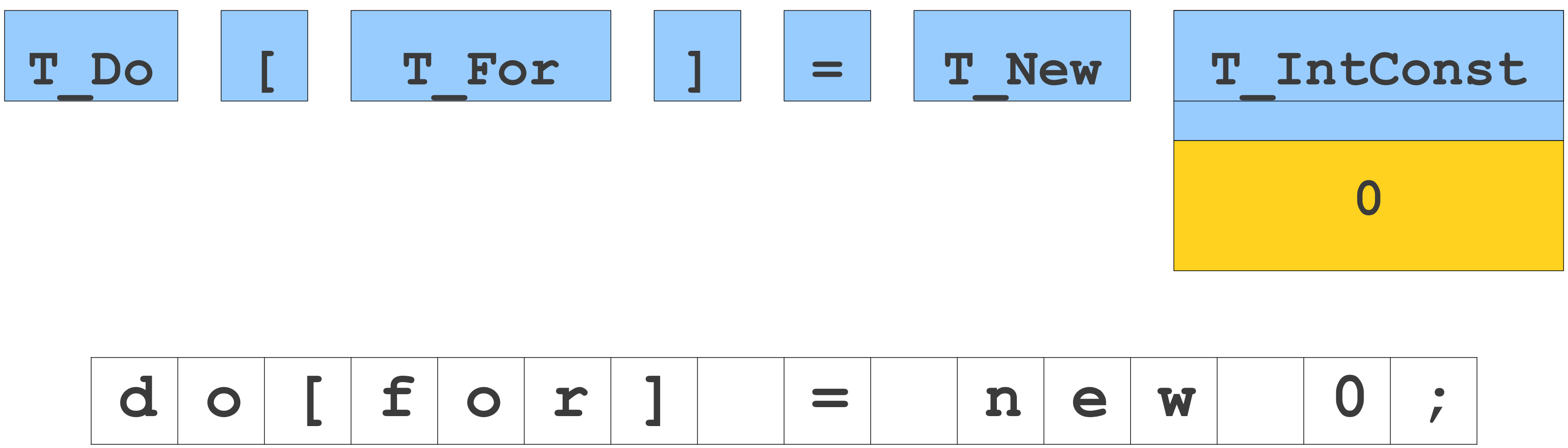

While
<
++
Ident
ip
Ident
z
Ident
ip
T_While
(
T_Ident
ip
<
T_Ident
z
)
++
T_Ident
ip
w h i l e (i p < z) \n \t + + i p ;

```
while (ip < z)
  ++ip;
```

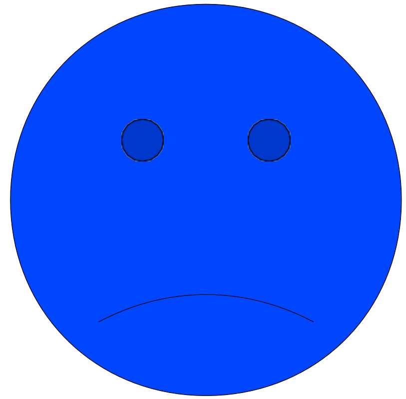
```
do[for] = new 0;
```

d	o	[	f	o	r	]		=		n	e	w		0	;

```
do[for] = new 0;
```




```
do[for] = new 0;
```



T_Do	[	T_For	]	=	T_New	T_IntConst
						0

d	o	[	f	o	r	]		=		n	e	w		0	;

```
do[for] = new 0;
```

Describing Syntax

- *Higher level constructs are given by syntax rules.*
- ***Syntax rules*** specify which strings from Σ* are in the language
- Examples: organization of the program, loop structures, assignment, expressions, subprogram definitions, and calls.

Formal Languages

- An **alphabet** is a set Σ of symbols that act as letters.
- A **language** over Σ is a set of strings made from symbols in Σ.
- When scanning, our alphabet was ASCII or Unicode characters. We produced tokens.
- When parsing, our alphabet is the set of tokens produced by the scanner.

The Limits of Regular Languages

- **When scanning, we used regular expressions to define each token.**
- **Unfortunately, regular expressions are (usually) too weak to define programming languages.**
  - Cannot define a regular expression matching all expressions with properly balanced parentheses.
  - Cannot define a regular expression matching all functions with properly nested block structure.
- We need a more powerful formalism.

Context-Free Grammars

- A **context-free grammar** (or **CFG**) is a formalism for defining languages.
- Can define the **context-free languages**, a strict superset of the the regular languages.

- **L = {one or more zeros followed by one or more ones}**

- **0^+1^+ : Regular expression**

- **S -> AB**
- **A-> 0A | 0**
- **B-> 1B | 1**

Is the following language regular?

L = {number of 0s followed by equal number of 1s}

L = $\{0^n 1^n, n>=0\}$

Context Free Grammars

$L = \{a^n b^n, n >= 0\}$

S -> aSb | empty

$L = \{a^n b^n, n >= 1\}$

S -> aSb | ab

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$S \rightarrow \texttt{a*b}$$

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$S \rightarrow Ab$$

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$
\begin{aligned}
S &\rightarrow A\mathtt{b} \\
A &\rightarrow A\mathtt{a} \mid \varepsilon
\end{aligned}
$$

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$S \rightarrow \texttt{a(b|c*)}$$

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\mathbf{S} \to \mathtt{a}\mathbf{X}$$
$$\mathbf{X} \to \mathtt{(b|c*)}$$

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$S \rightarrow aX$$
$$X \rightarrow b \mid c^*$$

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$S \rightarrow aX$$
$$X \rightarrow b \mid C$$

Not Notational Shorthand

- The syntax for regular expressions does not carry over to CFGs.
- Cannot use *, |, or parentheses.

$$\begin{aligned} S &\rightarrow \mathtt{a}X \\ X &\rightarrow \mathtt{b} \mid C \\ C &\rightarrow C\mathtt{c} \mid \varepsilon \end{aligned}$$

Context-Free Grammars

- **Context-Free Grammars**
  - Developed by Noam Chomsky in the mid-1950s
  - Language generators, meant to describe the syntax of natural languages
  - Define the class of context-free languages
  - Programming languages are contained in the class of CFL's.

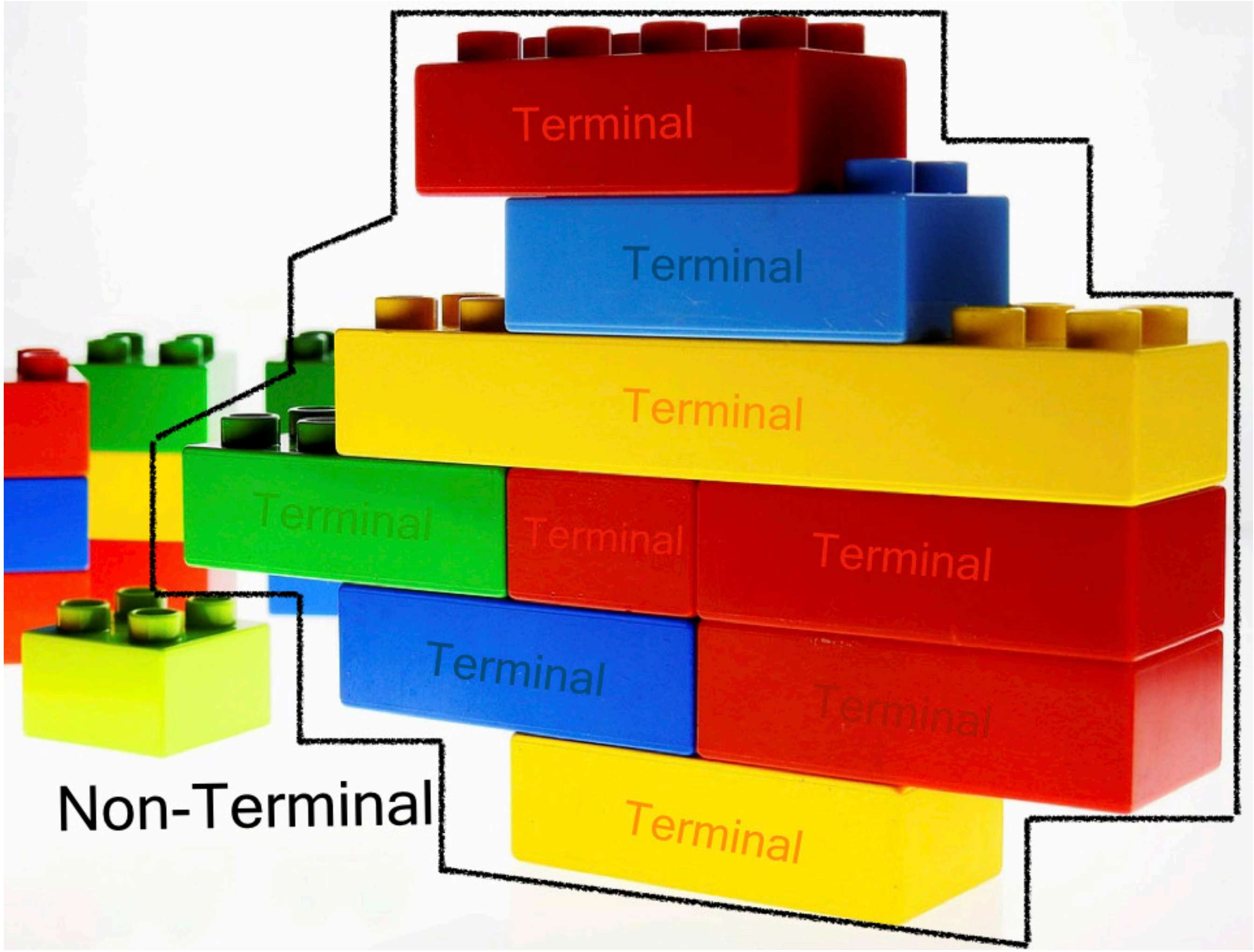
Elements of Syntax

- An alphabet of symbols
- Symbols are terminal and non-terminal
  - Terminals cannot be broken down
  - Non-terminals can be broken down further
- Grammar rules that express how symbols are combined to make legal sentences
- Rules are of the general form

```
non-terminal -> list of zero or more terminals or non-terminals
```

- One uses rules to recognize (parse) or generate legal sentences

Additional Notes on Terminals and NonTerminals



Source: https://tomassetti.me/ebnf/

Backus-Naur Form (BNF)

- A notation to describe the syntax of programming languages.
- Named after
  - John Backus – Algol 58
  - Peter Naur – Algol 60
- A ***metalanguage*** is a language used to describe another language.
- **BNF is a metalanguage used to describe PLs.**

BNF Fundamentals

- BNF uses abstractions for syntactic structures.

  <LHS> → <RHS>
- LHS: abstraction being defined
- RHS: definition
- "→" means "can have the form"
- Sometimes ::= is used for →

BNF Fundamentals

- Example, Java ***assignment*** statement can be represented by the abstraction **<assign>**
- **<assign> → <var> = <expression>**
- This is a ***rule*** or ***production***
- Here, **<var>** and **<expression>** must also be defined.
- example instances of this abstraction can be

```
total = sub1 + sub2
myVar = 4
```

BNF Fundamentals

- These abstractions are called **Variables** or **Nonterminals** of a Grammar.
- Lexemes and tokens are the **Terminals** of a grammar.
- Nonterminals are often enclosed in angle brackets
- Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

BNF Fundamentals

- A formal definition of ***rule***:

  A ***rule*** has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals

  <LHS> → <RHS>

- **Grammar:** a finite non-empty set of rules

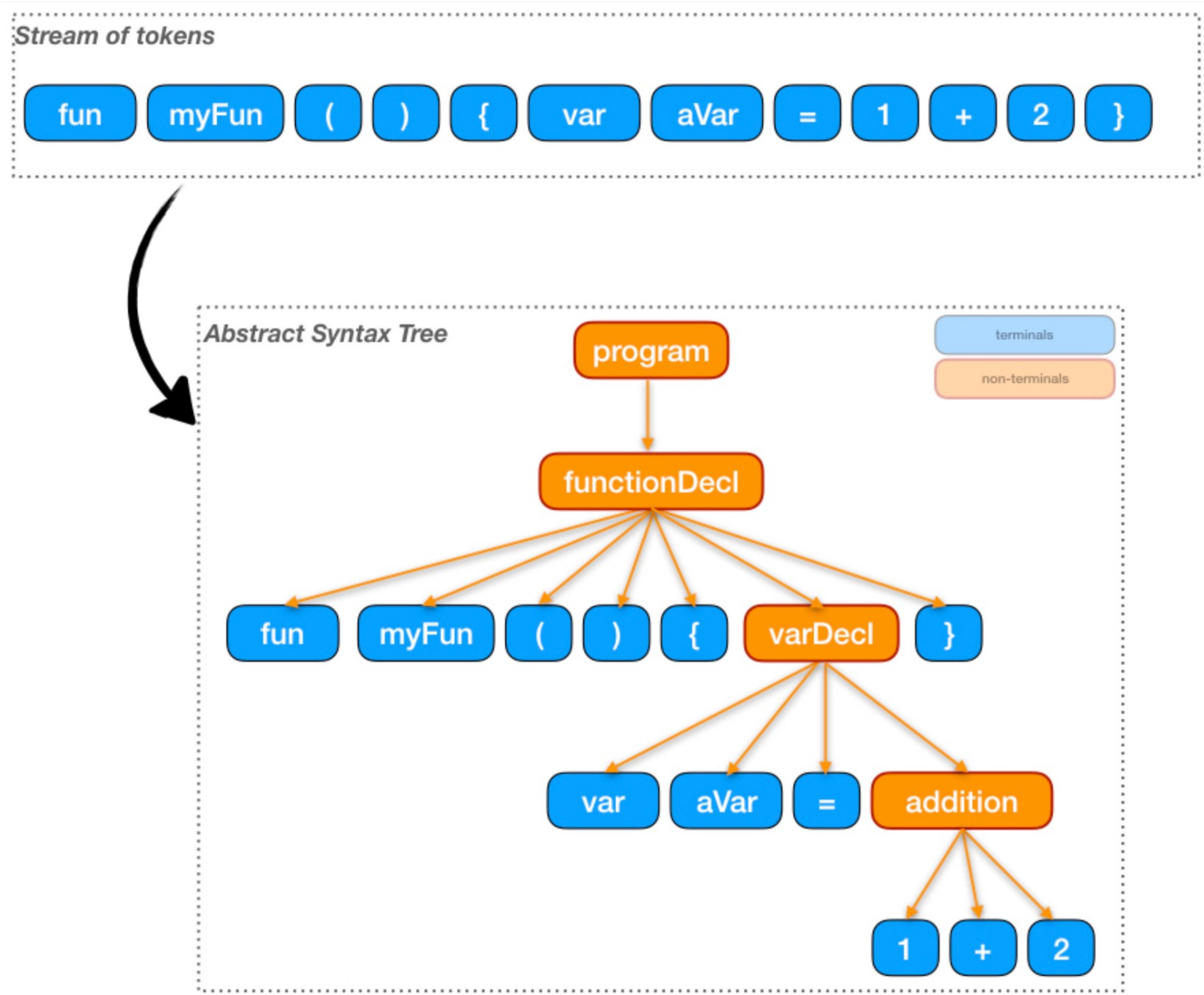
Terminals

- Let's see some typical terminals:
  - *identifiers:* these are the names used for variables, classes, functions, methods and so on.
  - *keywords:* almost every language uses keywords. They are exact strings that are used to indicate the start of a definition (think about class in Java or def in Python), a modifier (public, private, static, final, etc.) or control flow structures (while, for, until, etc.)

Terminals

- *literals:* these permit to define values in our languages. We can have string literals, numeric literal, char literals, boolean literals (but we could consider them keywords as well), array literals, map literals, and more, depending on the language
- *separators and delimiters:* like colons, semicolons, commas, parenthesis, brackets, braces
- *whitespaces:* spaces, tabs, newlines.
- *comments*

Terminals and Non-terminals



Source: https://tomassetti.me/ebnf/

Non-terminals

- Examples of non-terminals are:
  - *program/document*: represent the entire file
  - *module/classes:* group several declarations togethers
  - *functions/methods:* group statements together
  - *statements:* these are the single instructions. Some of them can contain other statements. Example : loops
  - *expressions:* are typically used within statements and can be composed in various ways

Examples

An initial example

- Consider the sentence **"Mary greets John"**

- A simple grammar

  <sentence> ::= <subject><predicate>

  <subject> ::= Mary

  <predicate> ::= <verb><object>

  <verb> ::= greets

  <object> ::= John

Alternations

- Multiple definitions can be separated by | (OR).

  <object> ::= John | Alfred

- This adds "Mary greets Alfred" to legal sentences

  <subject> ::= Mary | John | Alfred

  <object> ::= Mary | John | Alfred

- Alternation to the previous grammar

  <sentence> ::= <subject><predicate>

  <subject> ::= <noun>

  <predicate> ::= <verb><object>

  <verb> ::= greets

  <object> ::= <noun>

  <noun> ::= Mary | John | Alfred

Infinite Number of Sentences

<object> ::= John |

John again |

John again and again |

....

Instead use recursive definition

<object> ::= John |

John <repeat factor>

<repeat factor> ::= again |

again and <repeat factor>

A rule is recursive if its LHS appears in its RHS

Identifiers

<identifier> → <letter> |
<identifier><letter> |
<identifier><digit>

PASCAL/Ada If Statement

<if_stmt> → if <logic_expr> then <stmt>
<if_stmt> → if <logic_expr> then <stmt> else <stmt>

Or

<if_stmt> → if <logic_expr> then <stmt>
| if <logic_expr> then <stmt> else <stmt>

Another example

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

E → int
E → E Op E
E → (E)
Op → +
Op → -
Op → *
Op → /

E
⇒ E Op E
⇒ E Op int
⇒ int Op int
⇒ int / int

Arithmetic Expressions

- Suppose we want to describe all legal arithmetic expressions using addition, subtraction, multiplication, and division.
- Here is one possible CFG:

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

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

Context-Free Grammars

- Formally, a context-free grammar is a collection of four objects:
  - A set of **nonterminal symbols** (or **variables**),
  - A set of **terminal symbols**,
  - A set of **production rules** saying how each nonterminal can be converted by a string of terminals and nonterminals, and
  - A **start symbol** that begins the derivation.

E → `int`

E → E Op E

E → `(`E`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

A Notational Shorthand

$$\mathbf{E} \rightarrow \texttt{int}$$
$$\mathbf{E} \rightarrow \mathbf{E}\ \mathbf{Op}\ \mathbf{E}$$
$$\mathbf{E} \rightarrow \texttt{(}\mathbf{E}\texttt{)}$$
$$\mathbf{Op} \rightarrow \texttt{+}$$
$$\mathbf{Op} \rightarrow \texttt{-}$$
$$\mathbf{Op} \rightarrow \texttt{*}$$
$$\mathbf{Op} \rightarrow \texttt{/}$$

A Notational Shorthand

$$\mathbf{E} \rightarrow \texttt{int} \mid \mathbf{E}\ \mathbf{Op}\ \mathbf{E} \mid \texttt{(}\mathbf{E}\texttt{)}$$

$$\mathbf{Op} \rightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}$$

Derivations

```
   E
⇒ E Op E
⇒ E Op (E)
⇒ E Op (E Op E)
⇒ E * (E Op E)
⇒ int * (E Op E)
⇒ int * (int Op E)
⇒ int * (int Op int)
⇒ int * (int + int)
```

- This sequence of steps is called a **derivation**.
- A string $\alpha \mathbf{A} \omega$ **yields** string $\alpha\gamma\omega$ iff $\mathbf{A} \rightarrow \gamma$ is a production.
- If α yields β, we write $\alpha \Rightarrow \beta$.
- We say that α **derives** β iff there is a sequence of strings where

  $$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \beta$$

- If α derives β, we write $\alpha \Rightarrow^* \beta$.

Related Derivations

E → int | E Op E | (E)

Op → + | - | * | /

- E
- ⇒ E Op E
- ⇒ int Op E
- ⇒ int * E
- ⇒ int * (E)
- ⇒ int * (E Op E)
- ⇒ int * (int Op E)
- ⇒ int * (int + E)
- ⇒ int * (int + int)

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)

⇒ E * (int + int)

⇒ int * (int + int)

Grammars and Derivations

- A grammar is a generative device for defining languages
- The sentences of the language are **generated** through a sequence of applications of the rules, starting from the special nonterminal called ***start symbol***.
- Such a generation is called a **derivation**.
- Start symbol represents a complete program. So it is usually named as <program>.

An Example Grammar

```
<program>     → begin <stmt_list> end
<stmt_list> → <stmt> |
               <stmt> ; <stmt_list>
<stmt>         → <var> := <expression>
<var>          → A | B | C
<expression>→ <var> |
                 <var> <arith_op> <var>
<arith_op>   →           + | - | * | /
```

Derivation

- In order to check if a given string represents a valid program in the language, we try to derive it in the grammar.
- Derivation starts from the start symbol <program>.
- At each step we replace a nonterminal with its definition (RHS of the rule).

Derivations

- Every string of symbols in a derivation is a ***sentential form***
- A ***sentence*** is a sentential form that has only terminal symbols
- A ***leftmost derivation*** is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost

Leftmost Derivations

- A **leftmost derivation** is a derivation in which each step expands the leftmost nonterminal.
- A **rightmost derivation** is a derivation in which each step expands the rightmost nonterminal.
-

An Example Derivation

Derive string:

begin A := B; C := A * B end

```
<program>     → begin <stmt_list> end
<stmt_list> → <stmt> | <stmt> ; <stmt_list>
<stmt>        → <var> := <expression>
<var>         → A | B | C
<expression>→ <var> | <var> <arith_op> <var>
<arith_op>  →           + | - | * | /
```

<program> ⇒ begin <stmt_list> end

Leftmost derivation

⇒ begin <stmt> ; <stmt_list> end

⇒ begin <var> := <expression>; <stmt_list> end

Rightmost derivation

⇒ begin <stmt> ; <stmt_list> end

⇒ begin <stmt>; <stmt> end

Derive string:
begin A := B; C := A * B end

```
<program>    → begin <stmt_list> end
<stmt_list>  → <stmt> | <stmt> ; <stmt_list>
<stmt>       → <var> := <expression>
<var>        → A | B | C
<expression> → <var> | <var> <arith_op> <var>
<arith_op>   →        + | - | * | /
```

<program> ⇒ begin <stmt_list> end
⇒ begin <stmt> ; <stmt_list> end
⇒ begin <var> := <expression>; <stmt_list> end
⇒ begin A := <expression>; <stmt_list> end
⇒ begin A := B; <stmt_list> end
⇒ begin A := B; <stmt> end
⇒ begin A := B; <var> := <expression> end
⇒ begin A := B; C := <expression> end
⇒ begin A := B; C := <var><arith_op><var> end
⇒ begin A := B; C := A <arith_op> <var> end
⇒ begin A := B; C := A * <var> end
⇒ begin A := B; C := A * B end

If always the leftmost nonterminal is replaced, then it is called **leftmost derivation**.

Derive string:

begin A := B; C := A * B end

```
<program>     → begin <stmt_list> end
<stmt_list> → <stmt> | <stmt> ; <stmt_list>
<stmt>        → <var> := <expression>
<var>         → A | B | C
<expression>→ <var> | <var> <arith_op> <var>
<arith_op>   →       + | - | * | /
```

<program> ⇒ begin <stmt_list> end
⇒ begin <stmt> ; <stmt_list> end
⇒ begin <stmt> ; <stmt> end
⇒ begin <stmt> ; <var> := <expression> end
⇒ begin <stmt> ; <var> := <var><arith_op><var> end
⇒ begin <stmt> ; <var> := <var><arith_op> B end
⇒ begin <stmt> ; <var> := <var> * B end
⇒ begin <stmt> ; <var> := A * B end
⇒ begin <stmt> ; C := A * B end
⇒ begin <var> := <expression>; C := A * B end
⇒ begin <var> := <var>; C := A * B end
⇒ begin <var> := B ; C := A * B end
⇒ begin A := B; C := A * B end

If always the rightmost nonterminal is replaced, then it is called **rightmost derivation**.

Derive string:
begin A := B; C := A * B end

```
<program>     → begin <stmt_list> end
<stmt_list> → <stmt> | <stmt> ; <stmt_list>
<stmt>        → <var> := <expression>
<var>         → A | B | C
<expression>→ <var> | <var> <arith_op> <var>
<arith_op>  →         + | - | * | /
```

<program>
⇒ begin <stmt_list> end
⇒ begin <stmt> ; <stmt_list> end
⇒ begin <var> := <expression>; <stmt_list> end
⇒ begin A := <expression>; <stmt_list> end
⇒ begin A := B; <stmt_list> end
⇒ begin A := B; <stmt> end
⇒ begin A := B; <var> := <expression> end
⇒ begin A := B; C := <expression> end
⇒ begin A := B; C := <var><arith_op><var> end
⇒ begin A := B; C := A <arith_op> <var> end
⇒ begin A := B; C := A * <var> end
⇒ begin A := B; C := A * B end

Leftmost derivation

<program>
⇒ begin <stmt_list> end
⇒ begin <stmt> ; <stmt_list> end
⇒ begin <stmt> ; <stmt> end
⇒ begin <stmt> ; <var> := <expression> end
⇒ begin <stmt> ; <var> := <var><arith_op><var> end
⇒ begin <stmt> ; <var> := <var><arith_op> B end
⇒ begin <stmt> ; <var> := <var> * B end
⇒ begin <stmt> ; <var> := A * B end
⇒ begin <stmt> ; C := A * B end
⇒ begin <var> := <expression>; C := A * B end
⇒ begin <var> := <var>; C := A * B end
⇒ begin <var> := B ; C := A * B end
⇒ begin A := B; C := A * B end

Rightmost derivation

Related Derivations

$E \rightarrow$ `int` | E Op E | `(`E`)`

Op $\rightarrow$ `+` | `-` | `*` | `/`

E	E
⇒ E Op E	⇒ E Op E
⇒ int Op E	⇒ E Op (E)
⇒ int * E	⇒ E Op (E Op E)
⇒ int * (E)	⇒ E Op (E Op int)
⇒ int * (E Op E)	⇒ E Op (E + int)
⇒ int * (int Op E)	⇒ E Op (int + int)
⇒ int * (int + E)	⇒ E * (int + int)
⇒ int * (int + int)	⇒ int * (int + int)

Derivations Revisited

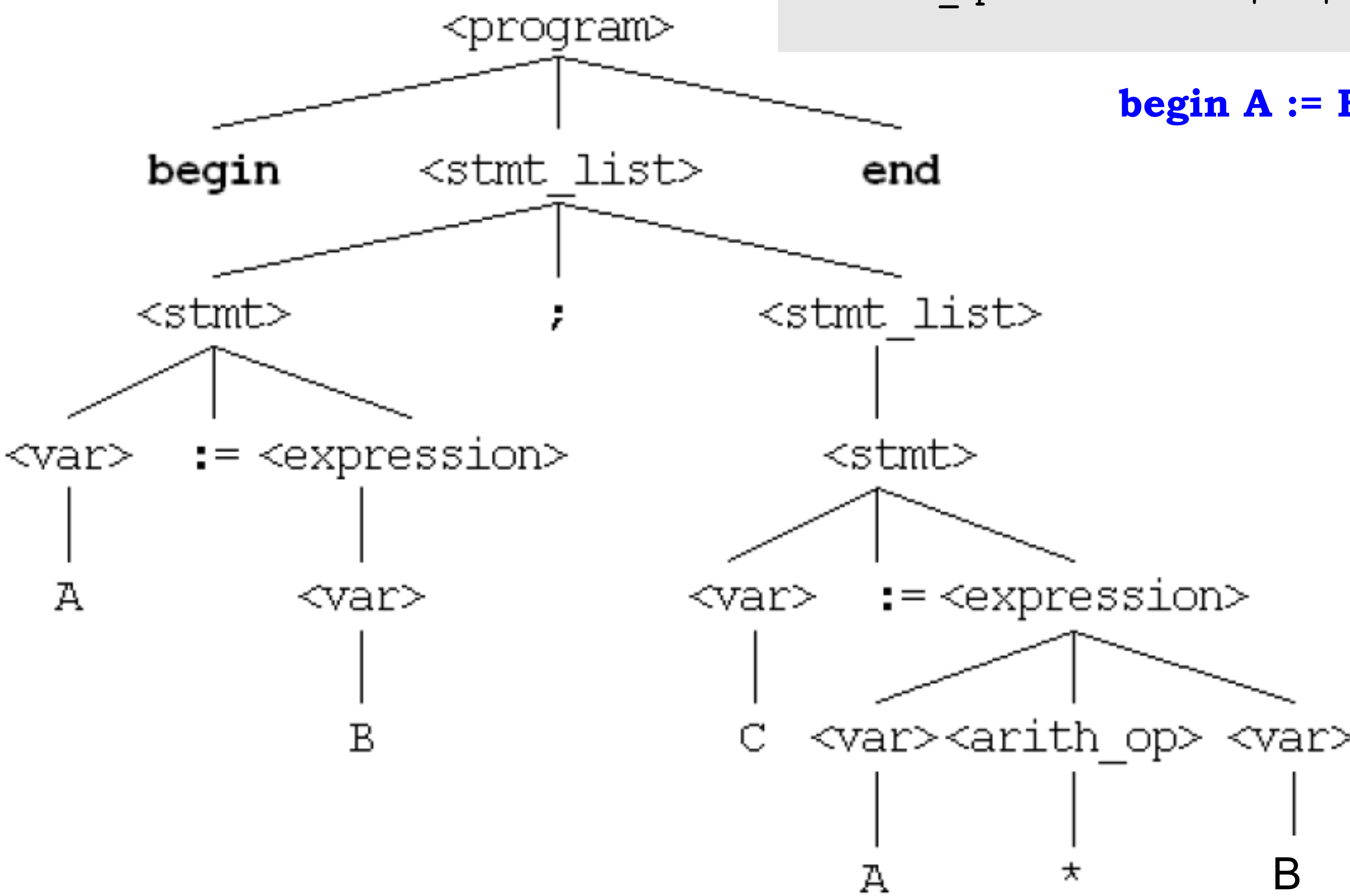
- A derivation encodes two pieces of information:
  - What productions were applied produce the resulting string from the start symbol?
  - In what order were they applied?
- Multiple derivations might use the same productions, but apply them in a different order.

Parse Tree

A hierarchical representation of a derivation

```
<program>    → begin <stmt_list> end
<stmt_list> → <stmt> | <stmt> ; <stmt_list>
<stmt>       → <var> := <expression>
<var>        → A | B | C
<expression>→ <var> | <var> <arith_op> <var>
<arith_op>  →        + | - | * | /
```

begin A := B; C := A * B end



Arithmetic expressions revisited

$$E \to \texttt{int} \mid E \; Op \; E \mid \texttt{(} E \texttt{)}$$

$$Op \to \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}$$

Parse Trees

E

Parse Trees

E

E

Parse Trees

E

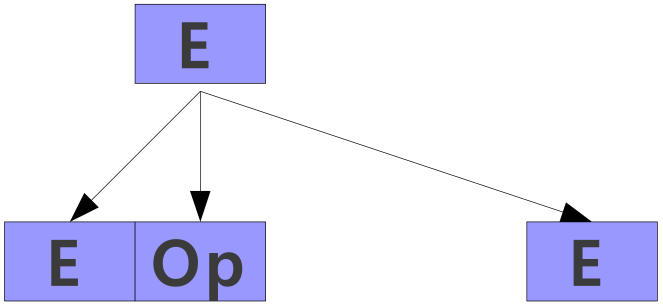
E

$\Rightarrow$ E Op E

Parse Trees

E

$\Rightarrow$ E Op E

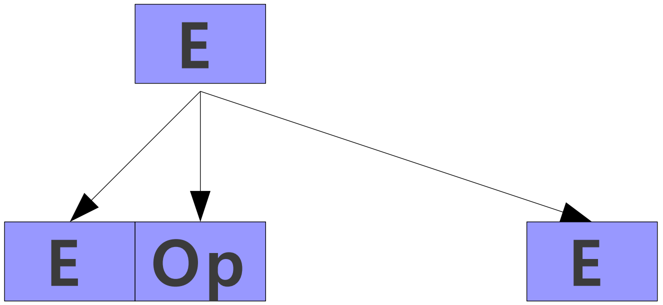


Parse Trees

E

⇒ E Op E

⇒ `int` Op E

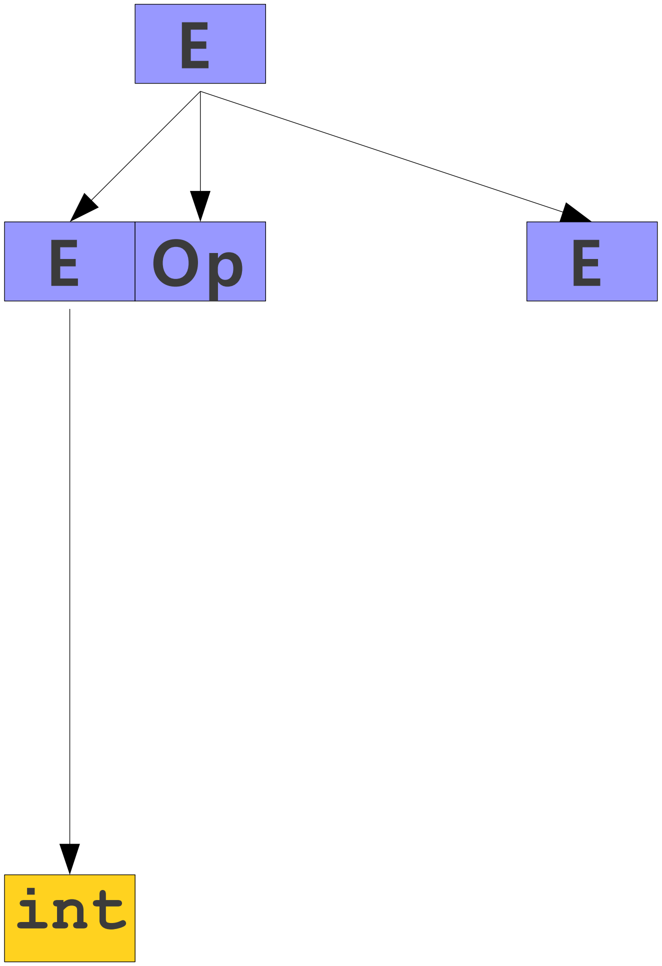


Parse Trees

E

$\Rightarrow$ E Op E

$\Rightarrow$ `int` Op E



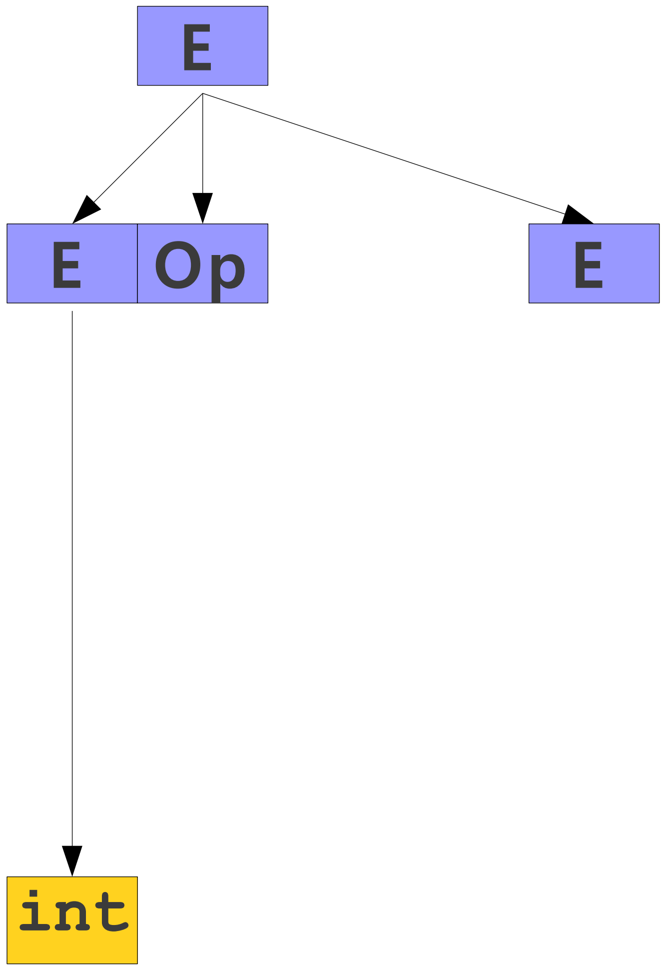
Parse Trees

E

⇒ E Op E

⇒ int Op E

⇒ int * E



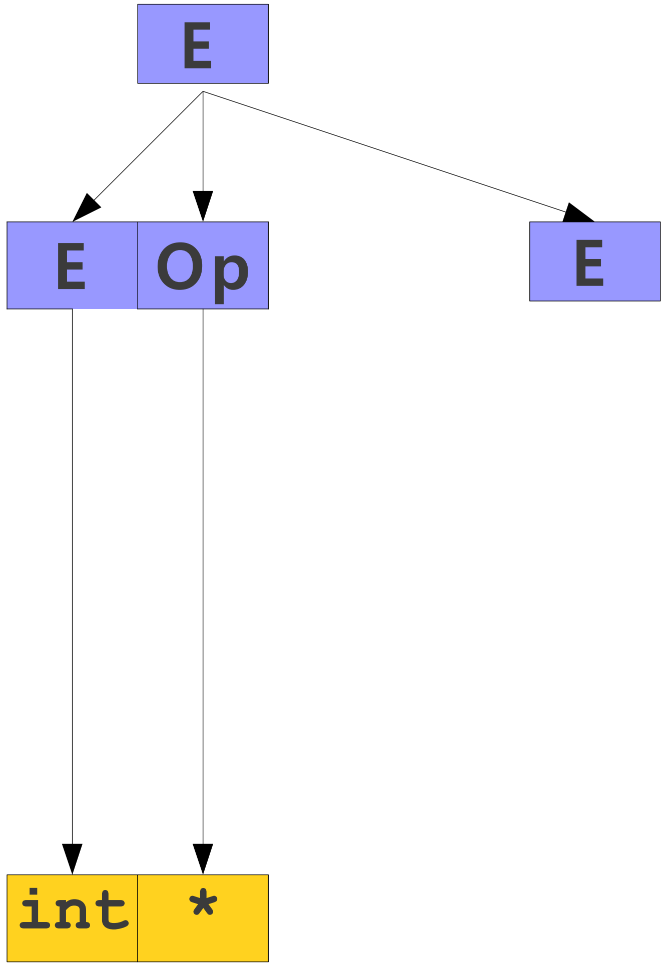
Parse Trees

E

⇒ E Op E

⇒ `int` Op E

⇒ `int * ` E



Parse Trees

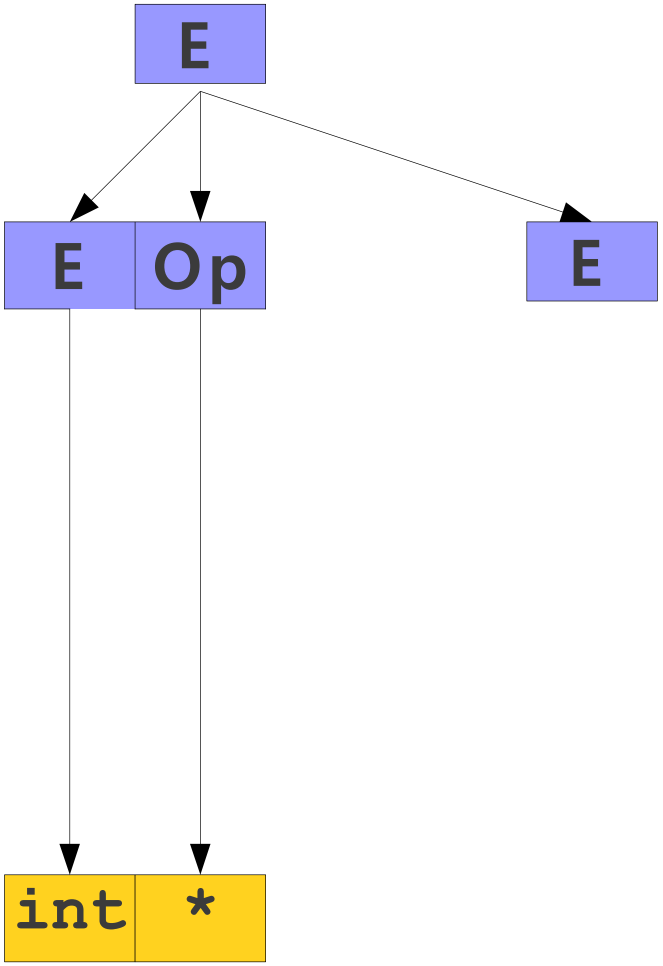
E

$\Rightarrow$ E Op E

$\Rightarrow$ int Op E

$\Rightarrow$ int * E

$\Rightarrow$ int * (E)



Parse Trees

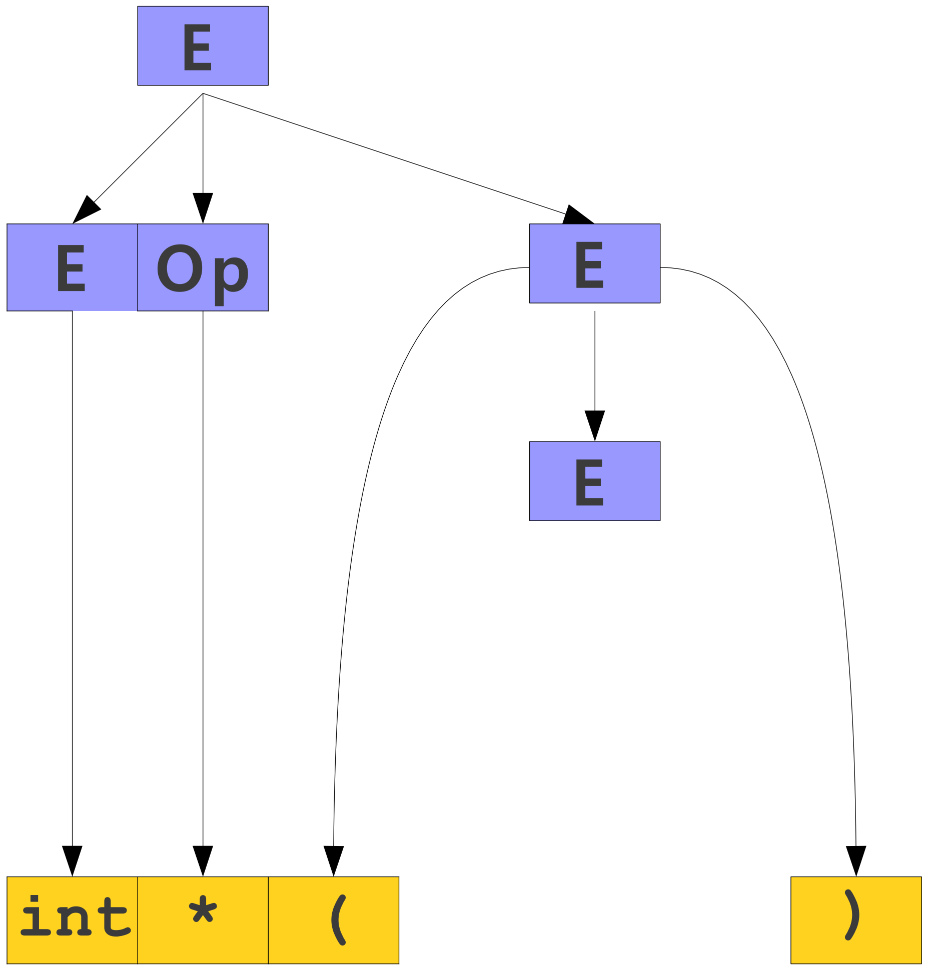
E

⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)



Parse Trees

E

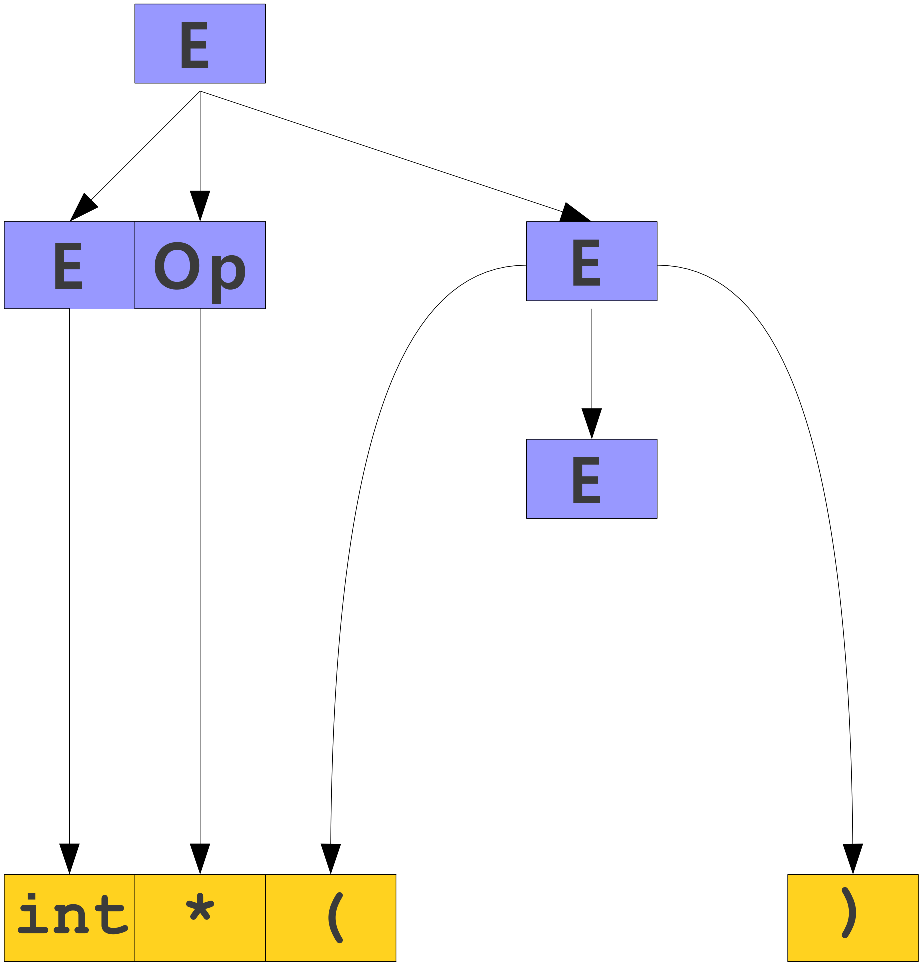
⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)



Parse Trees

E

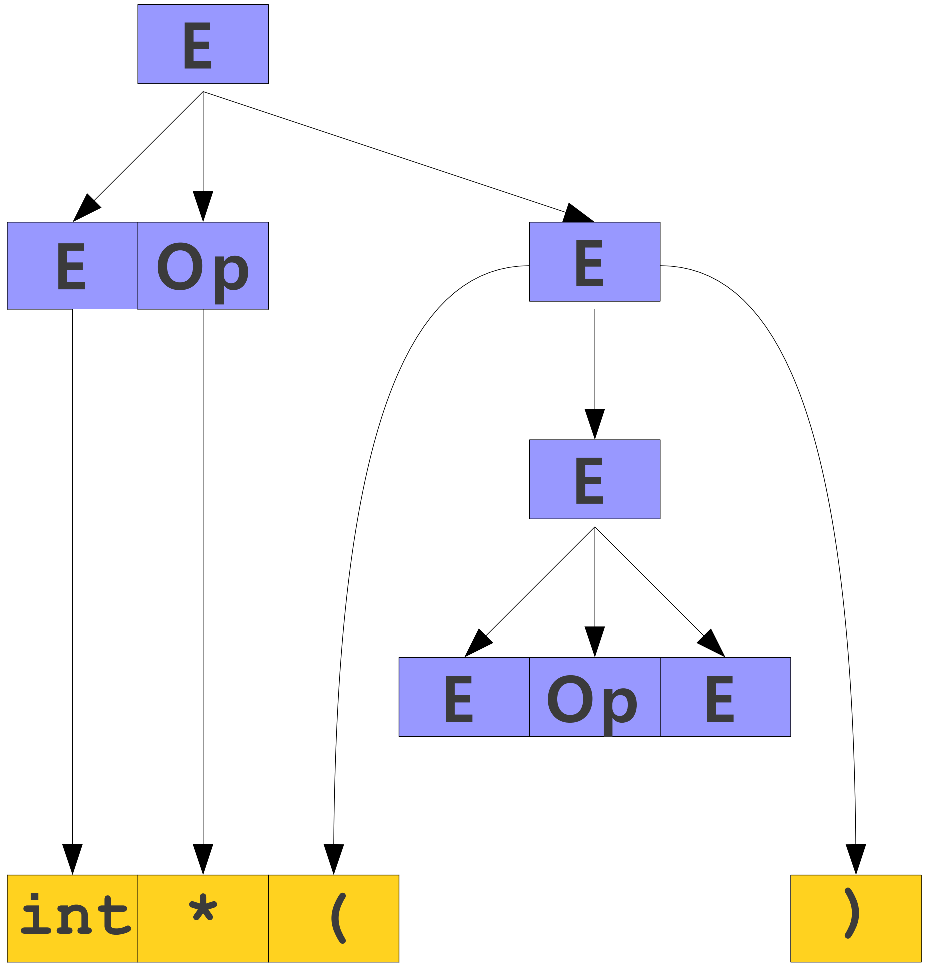
⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)



Parse Trees

E

⇒ E Op E

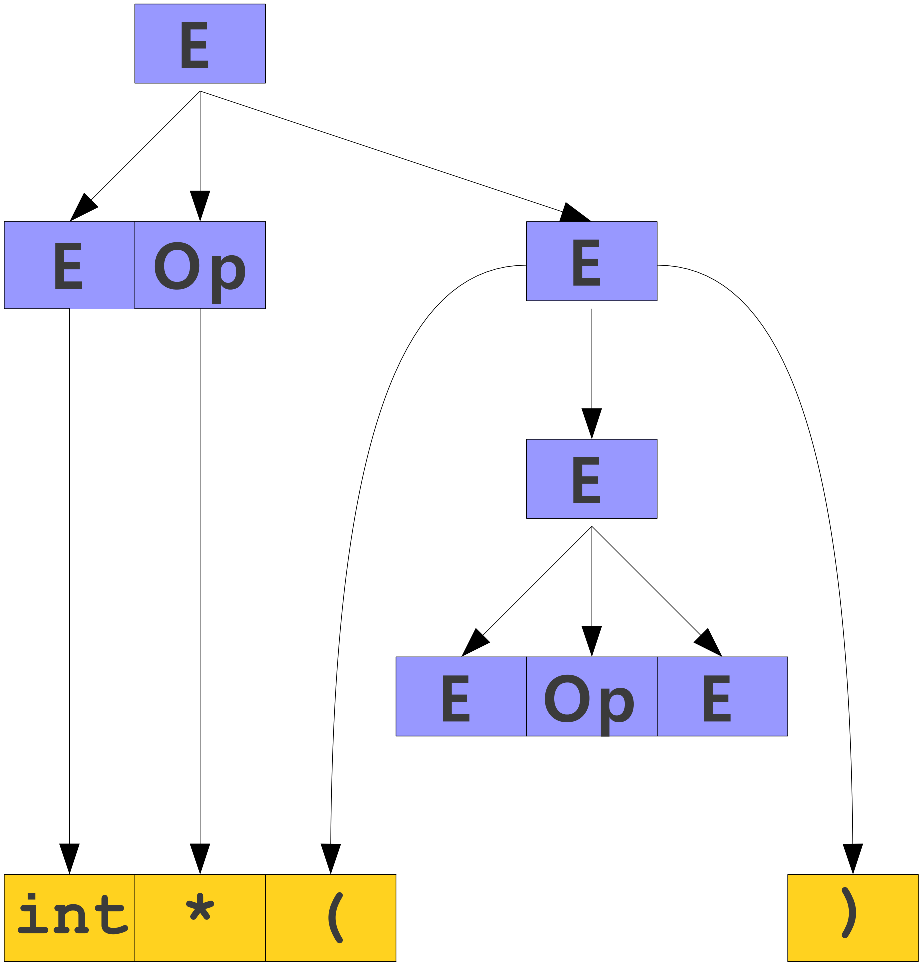
⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)



Parse Trees

E

⇒ E Op E

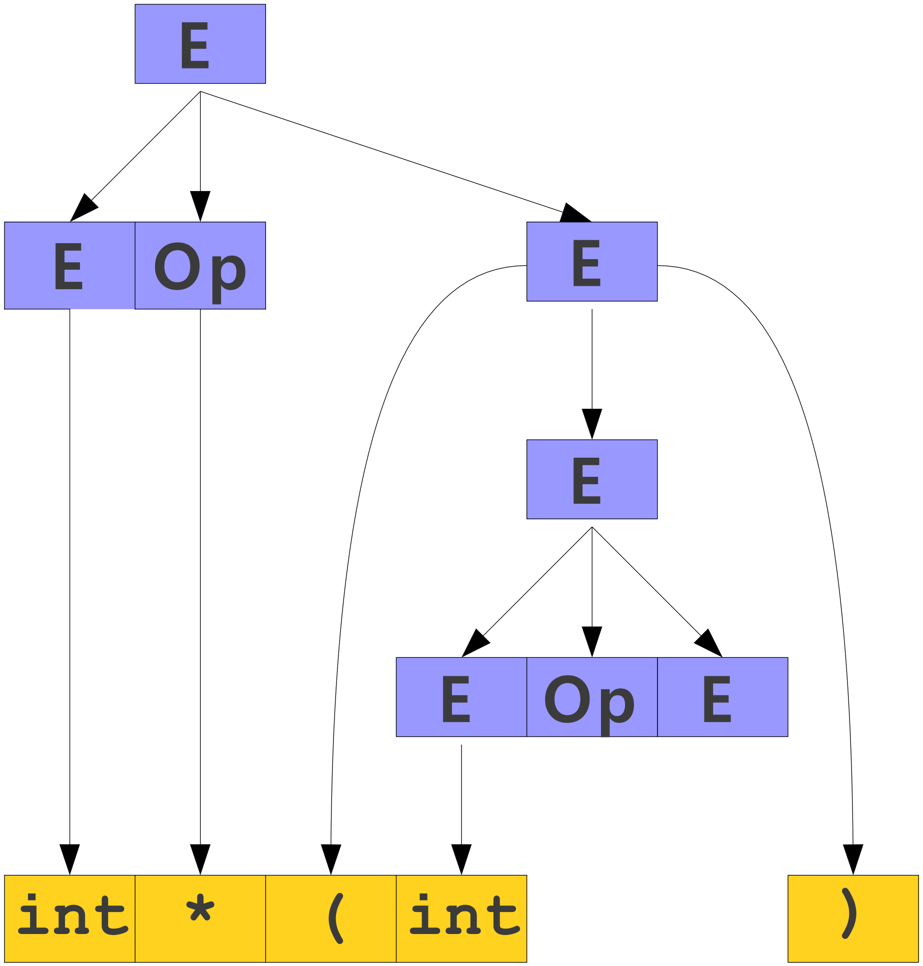
⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)



Parse Trees

E

⇒ E Op E

⇒ int Op E

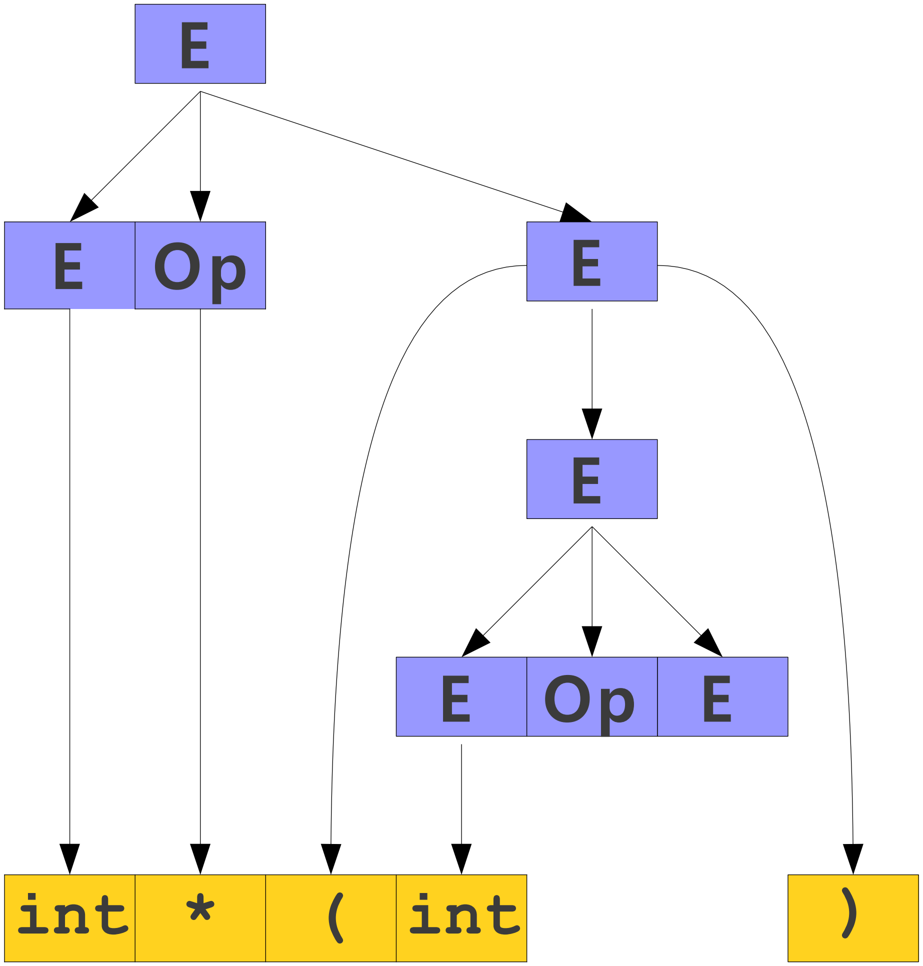
⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

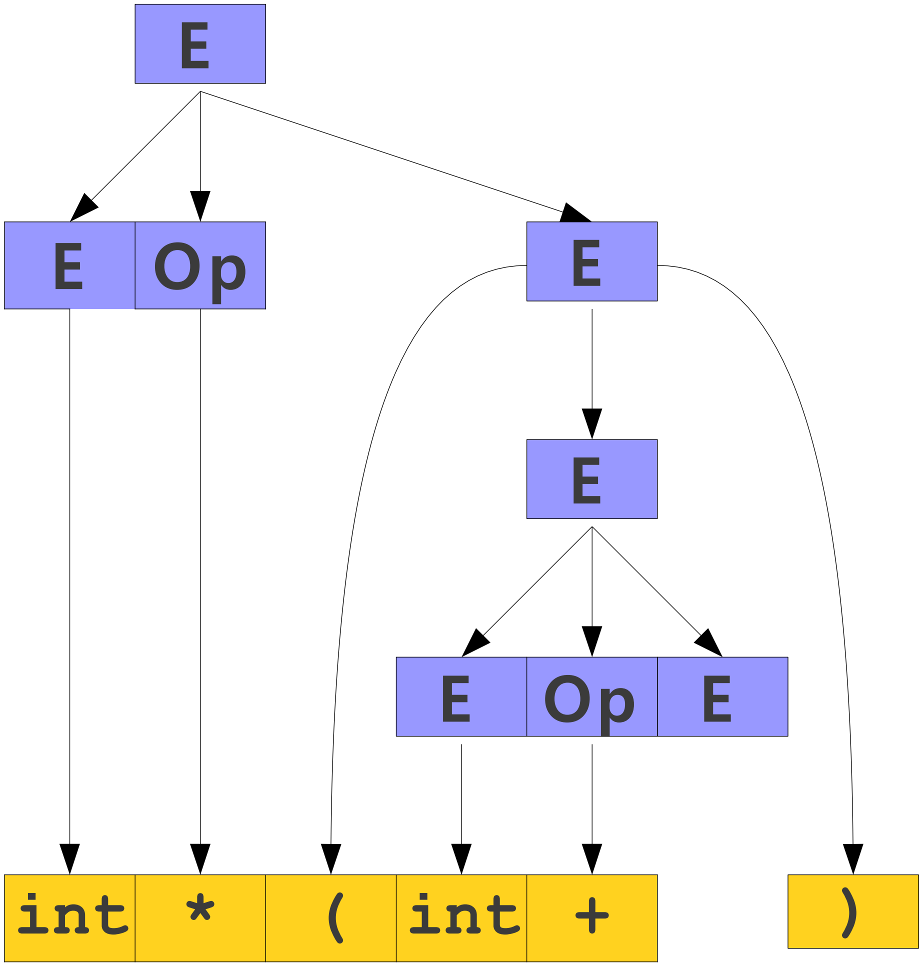
⇒ int * (int Op E)

⇒ int * (int + E)



Parse Trees

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



Parse Trees

E

⇒ E Op E

⇒ int Op E

⇒ int * E

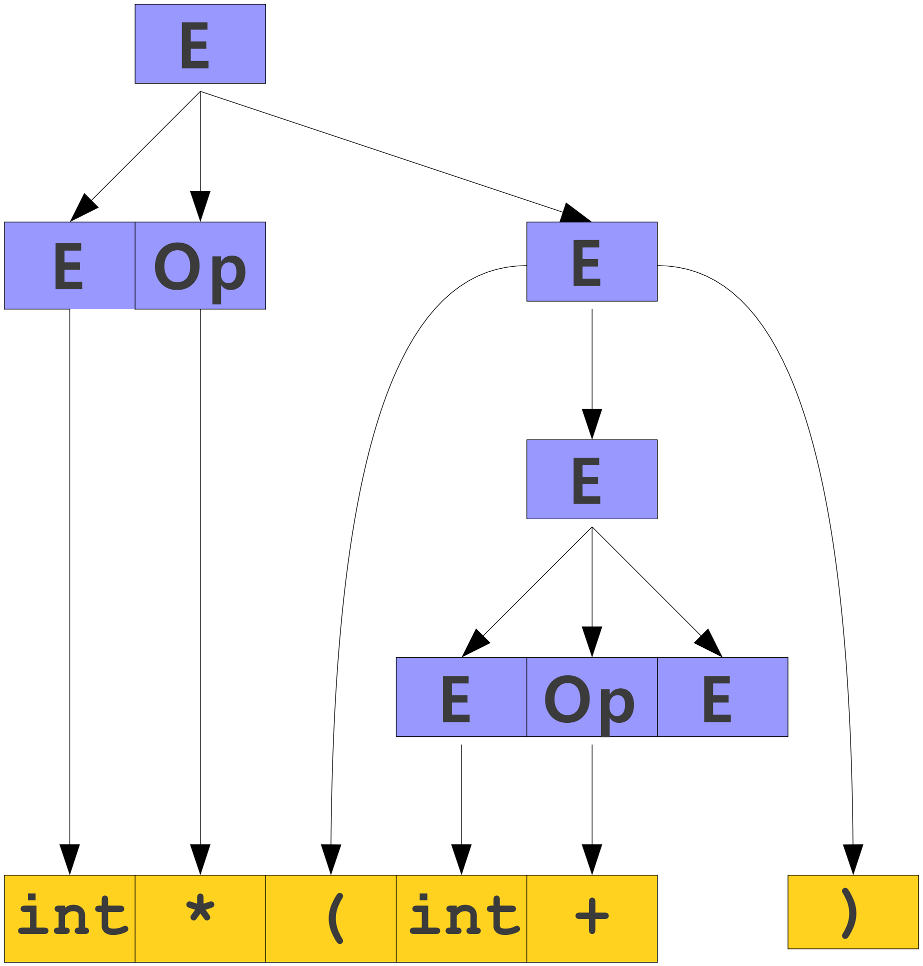
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)



Parse Trees

E

⇒ E Op E

⇒ int Op E

⇒ int * E

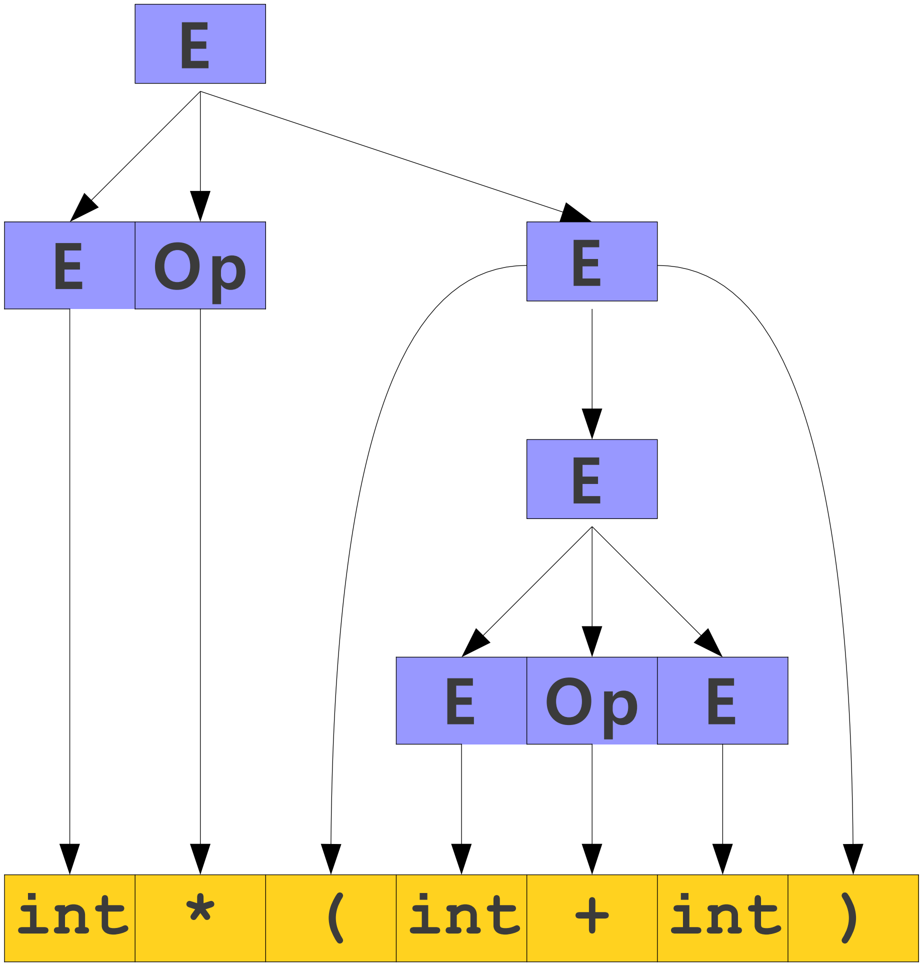
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)



Parse Trees

E

Parse Trees

E

E

Parse Trees

E

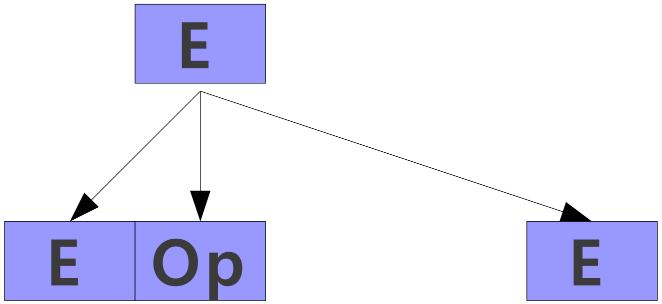
E

$\Rightarrow$ E Op E

Parse Trees

E

$\Rightarrow$ E Op E

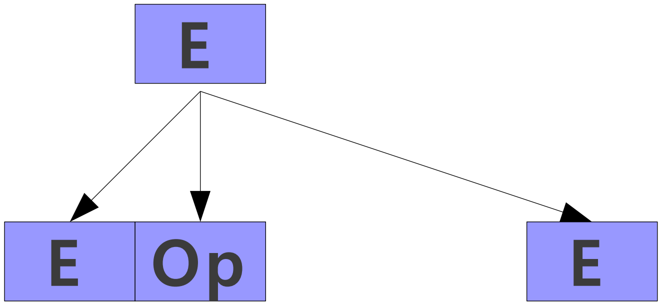


Parse Trees

E

$\Rightarrow$ E Op E

$\Rightarrow$ E Op (E)

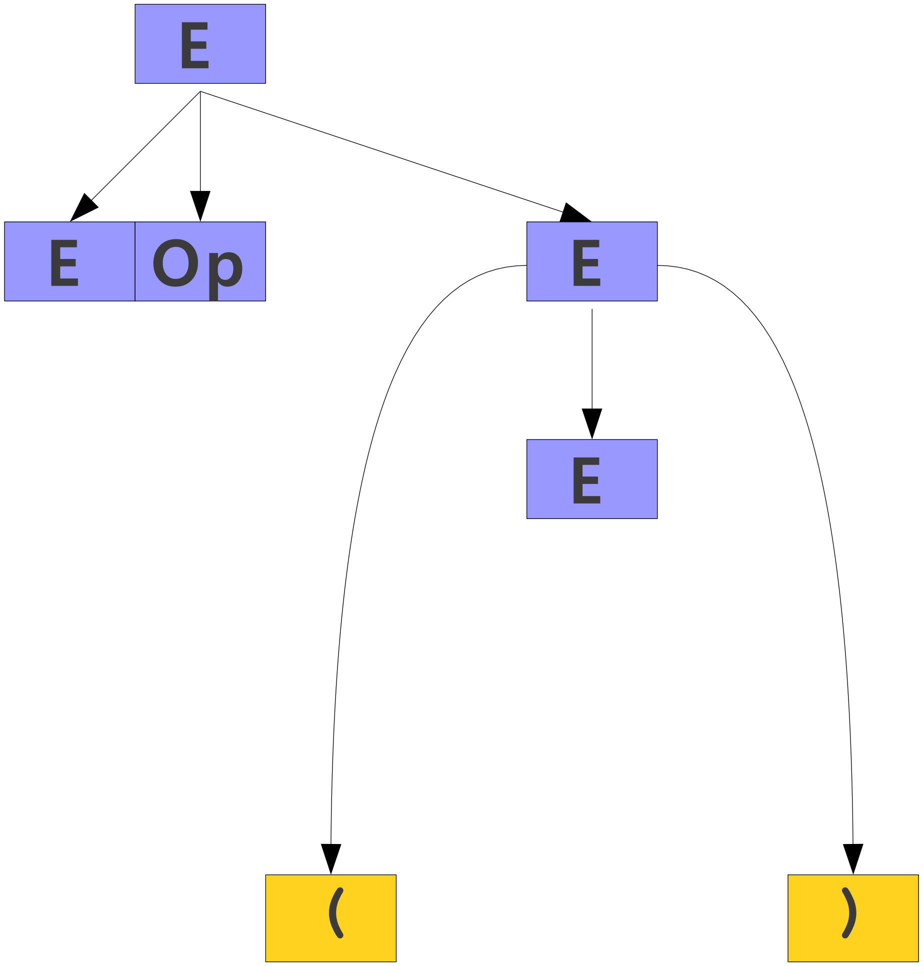


Parse Trees

E

⇒ E Op E

⇒ E Op (E)



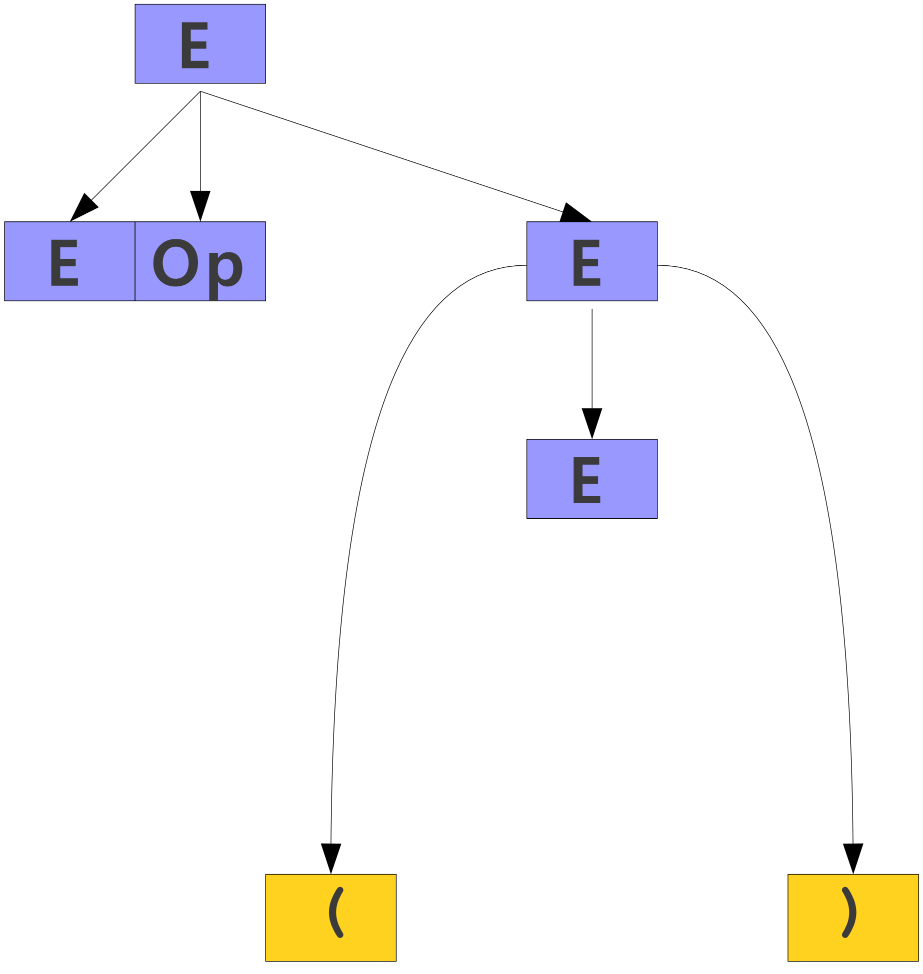
Parse Trees

E

$\Rightarrow$ E Op E

$\Rightarrow$ E Op (E)

$\Rightarrow$ E Op (E Op E)



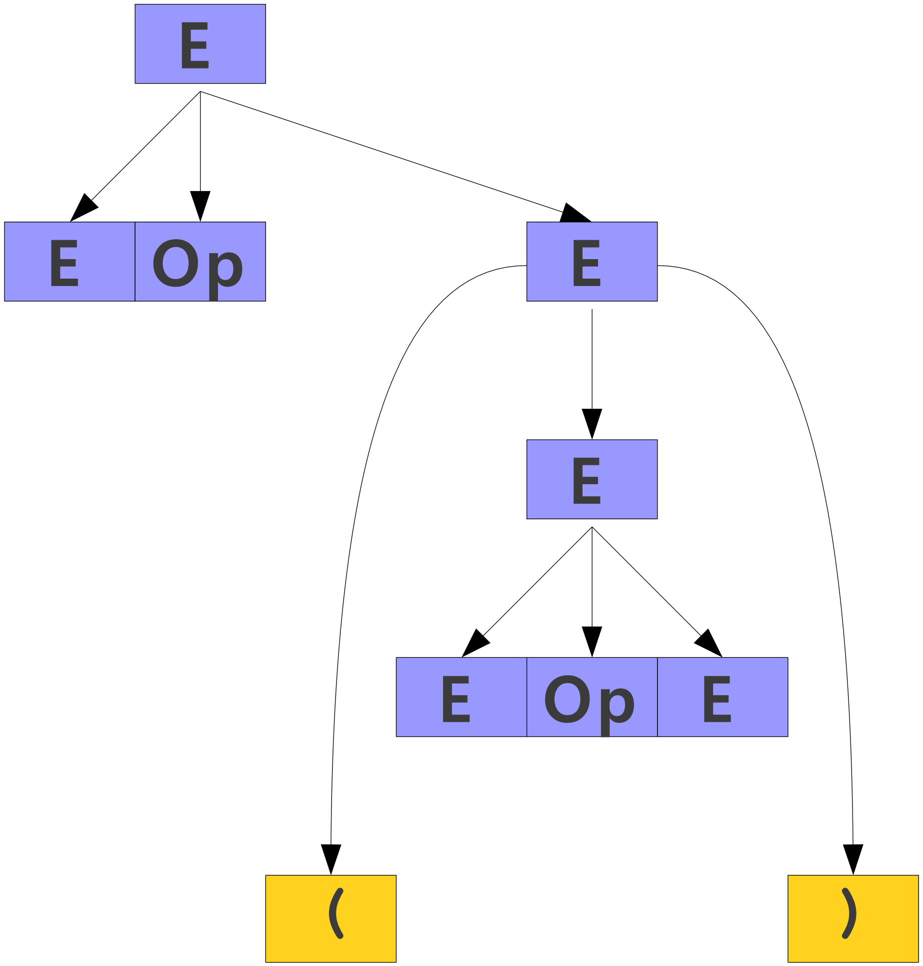
Parse Trees

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)



Parse Trees

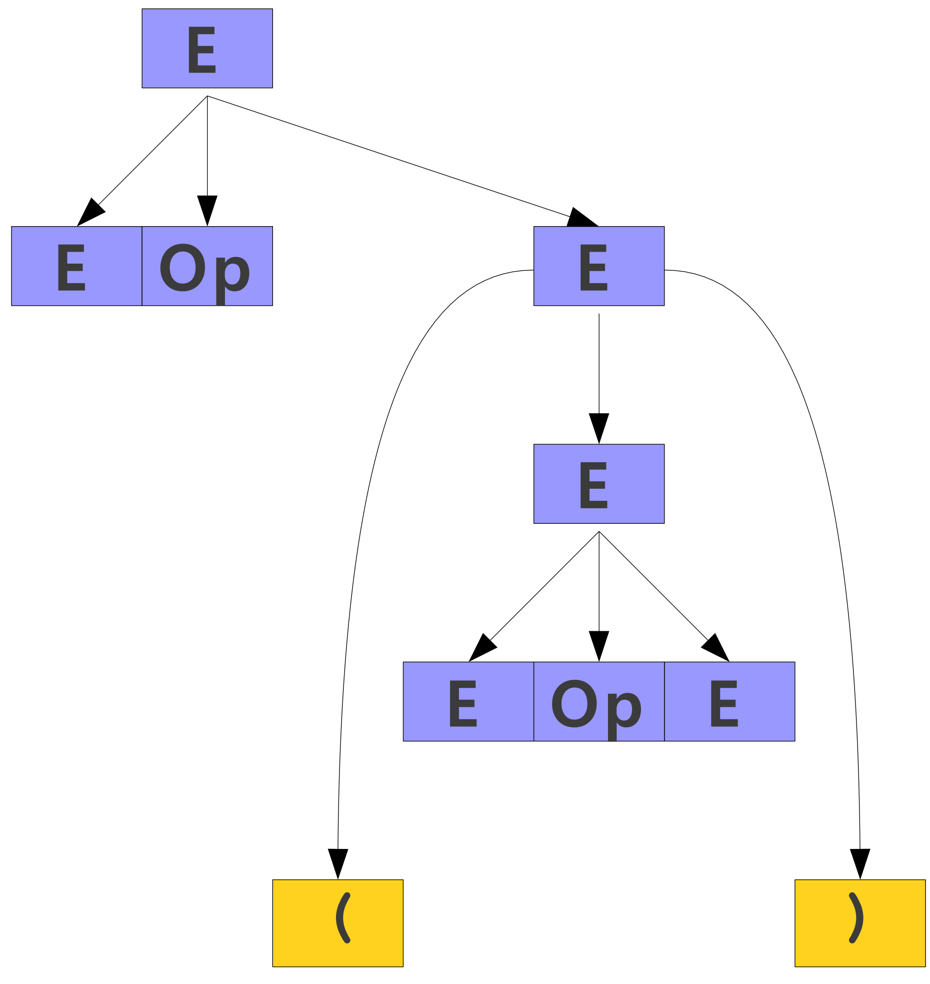
E

$\Rightarrow$ E Op E

$\Rightarrow$ E Op (E)

$\Rightarrow$ E Op (E Op E)

$\Rightarrow$ E Op (E Op `int`)



Parse Trees

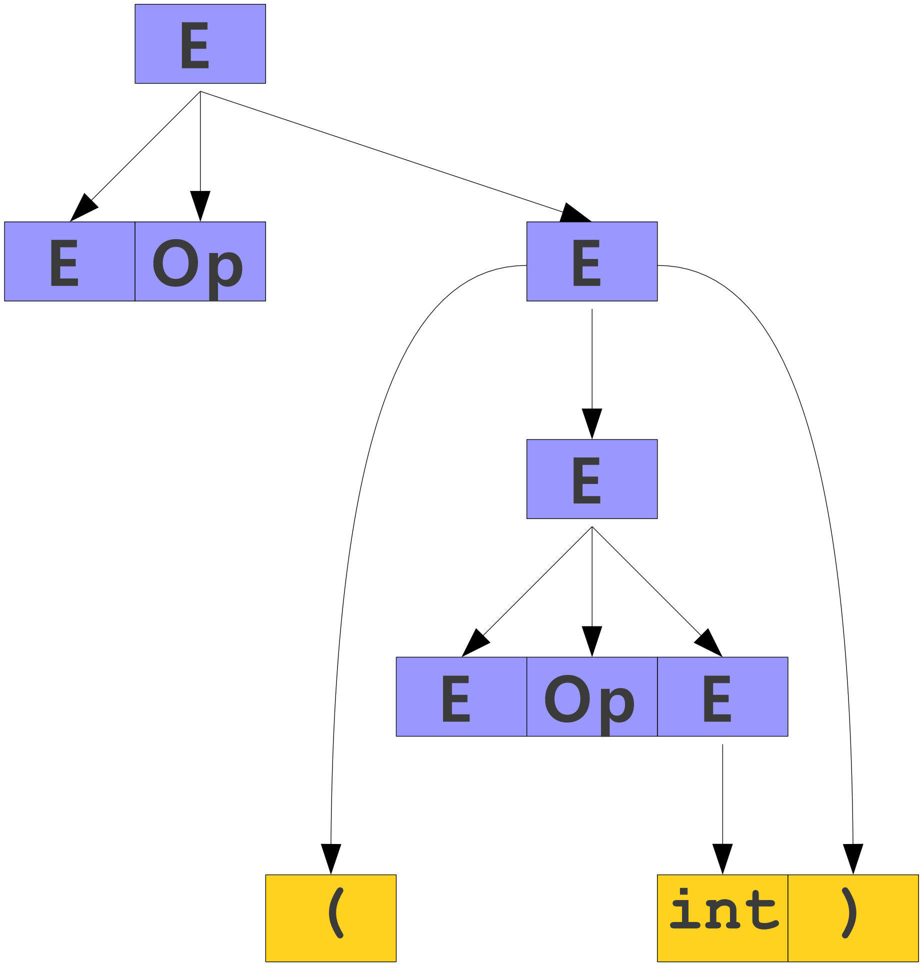
E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)



Parse Trees

E

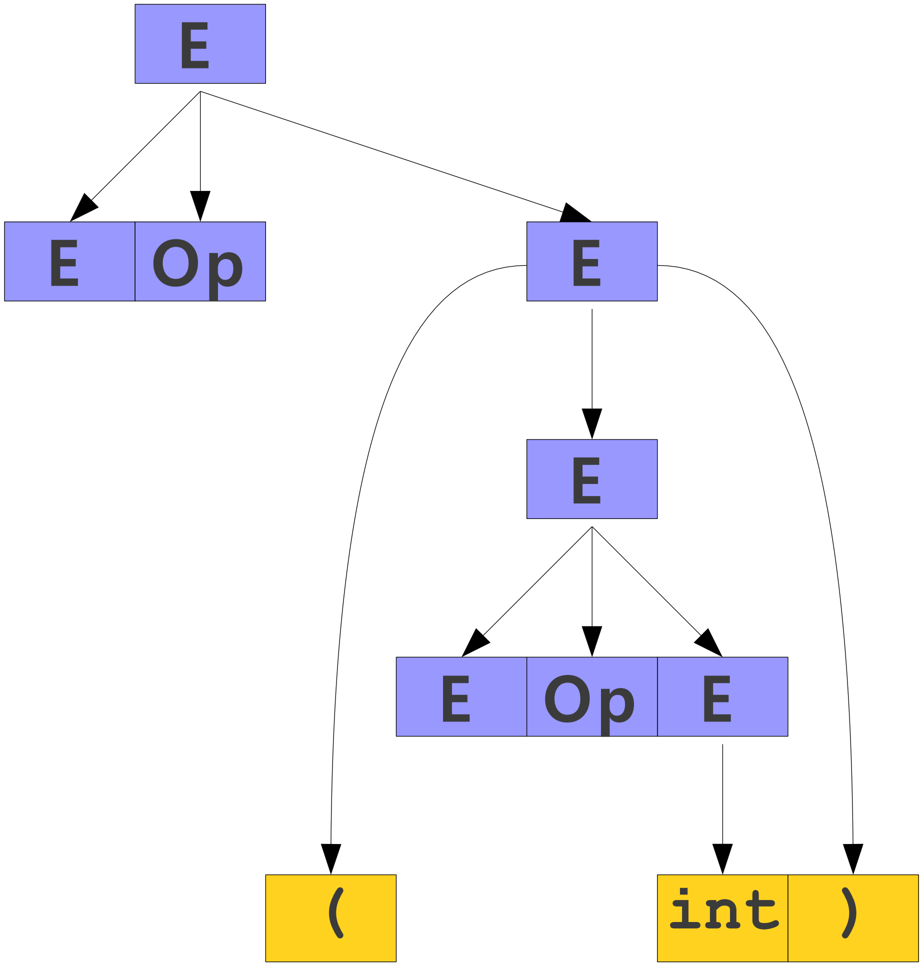
⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op `int`)

⇒ E Op (E + `int`)



Parse Trees

E

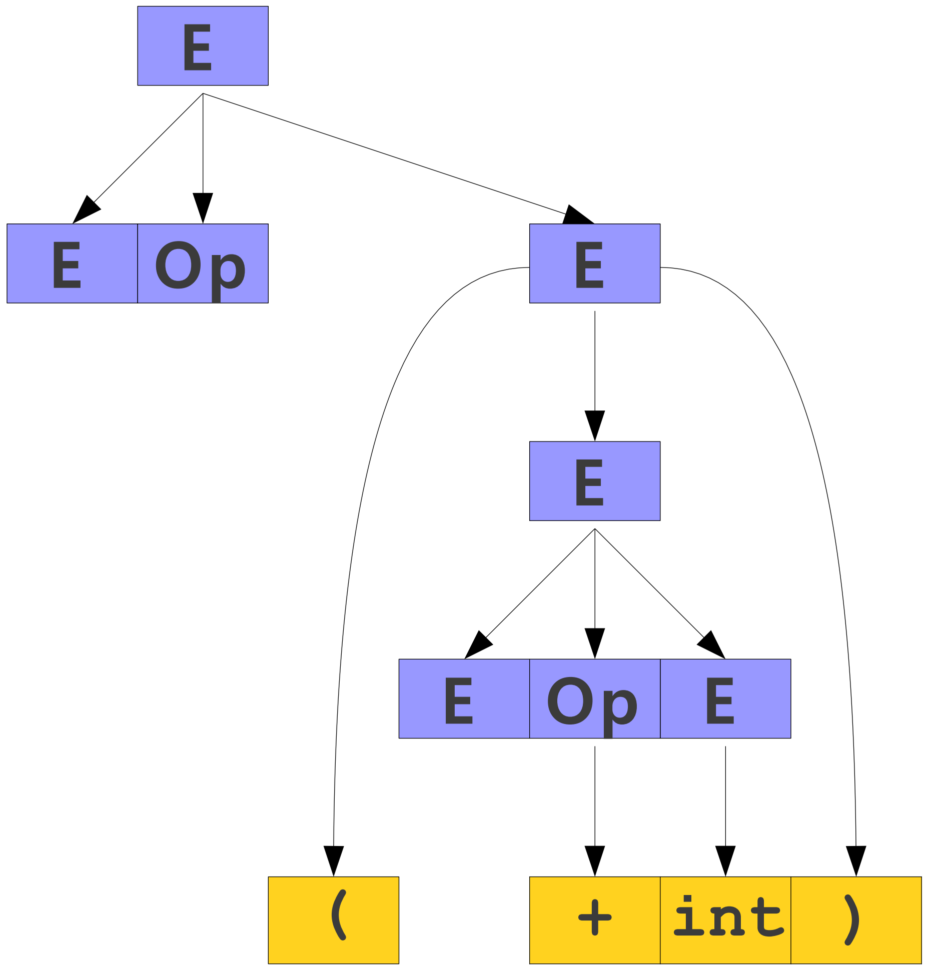
⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)



Parse Trees

E

⇒ E Op E

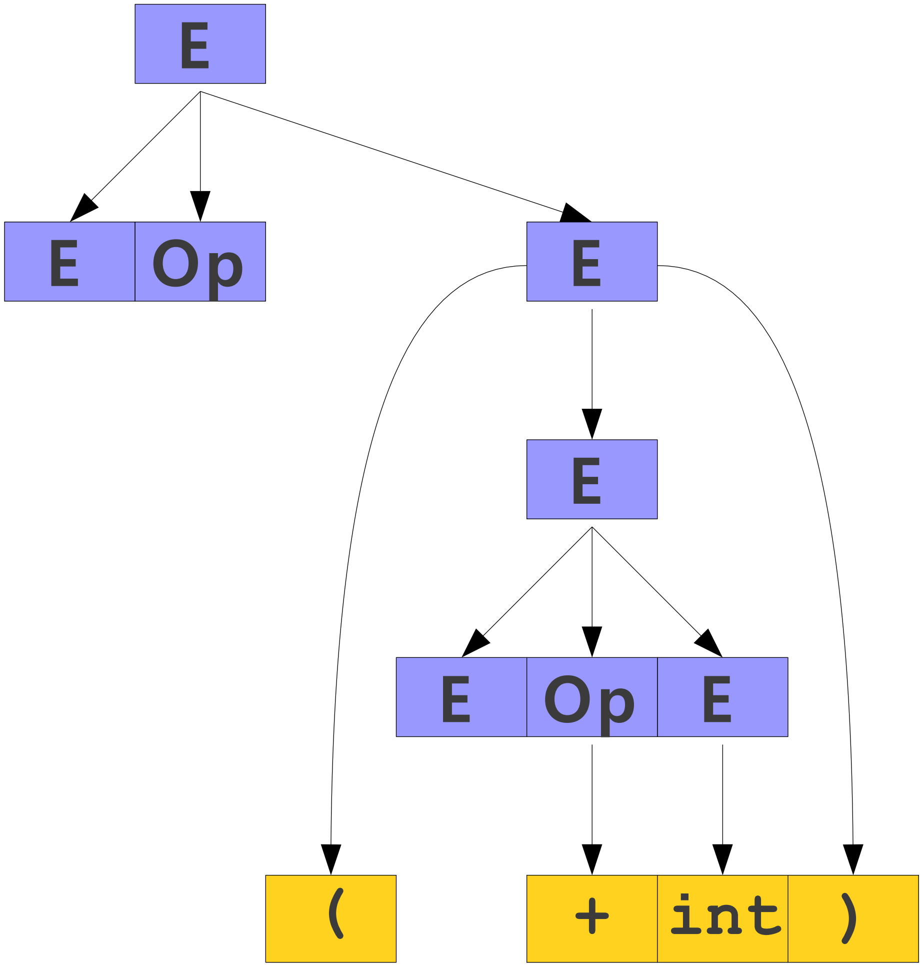
⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)



Parse Trees

E

⇒ E Op E

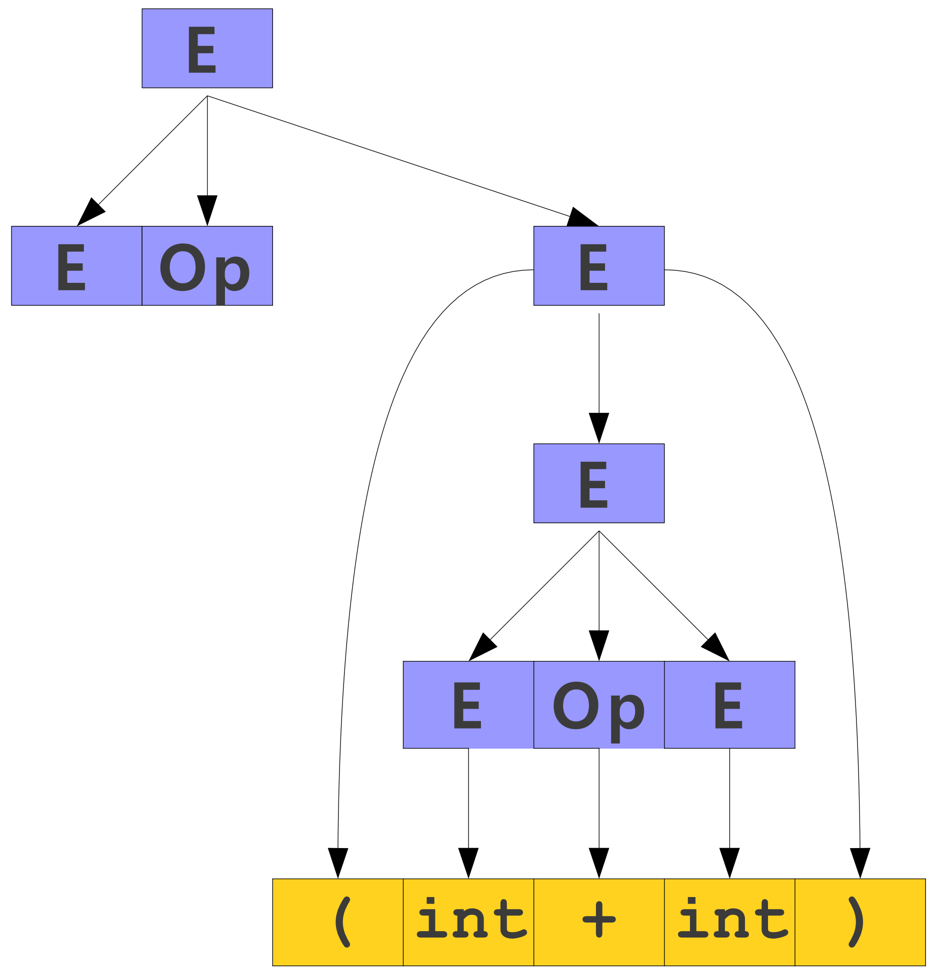
⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)



Parse Trees

E

$\Rightarrow$ E Op E

$\Rightarrow$ E Op (E)

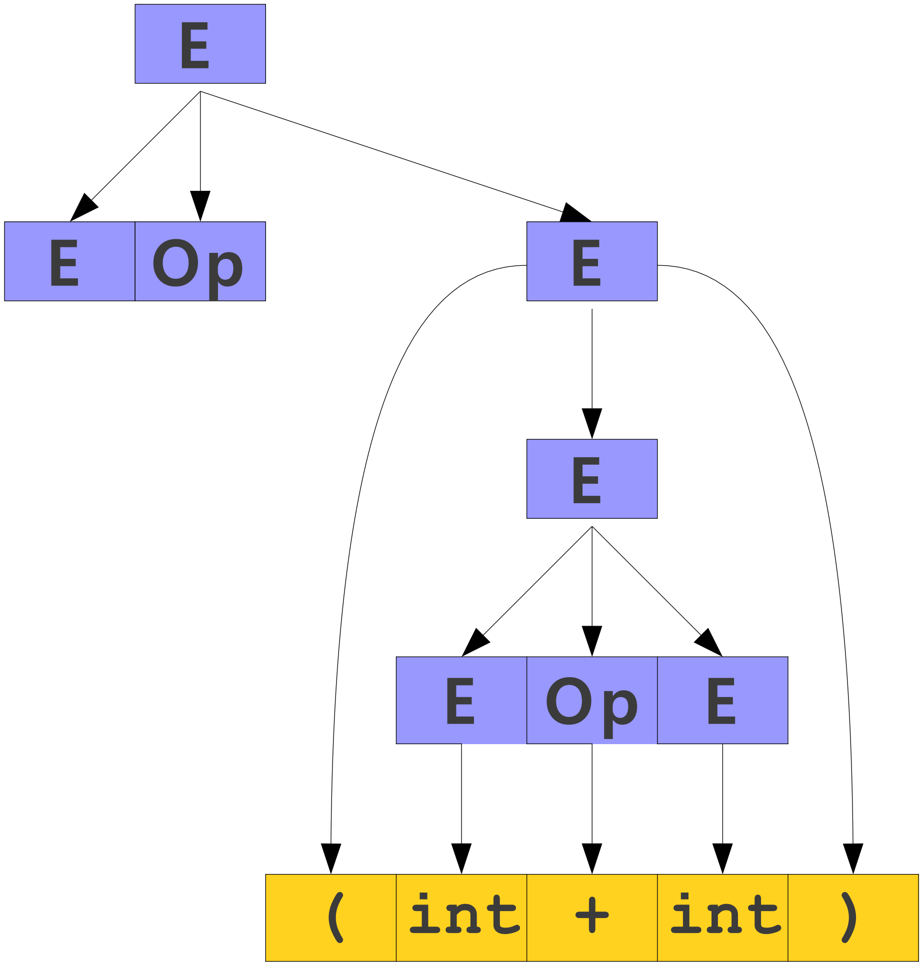
$\Rightarrow$ E Op (E Op E)

$\Rightarrow$ E Op (E Op int)

$\Rightarrow$ E Op (E + int)

$\Rightarrow$ E Op (int + int)

$\Rightarrow$ E * (int + int)



Parse Trees

E

⇒ E Op E

⇒ E Op (E)

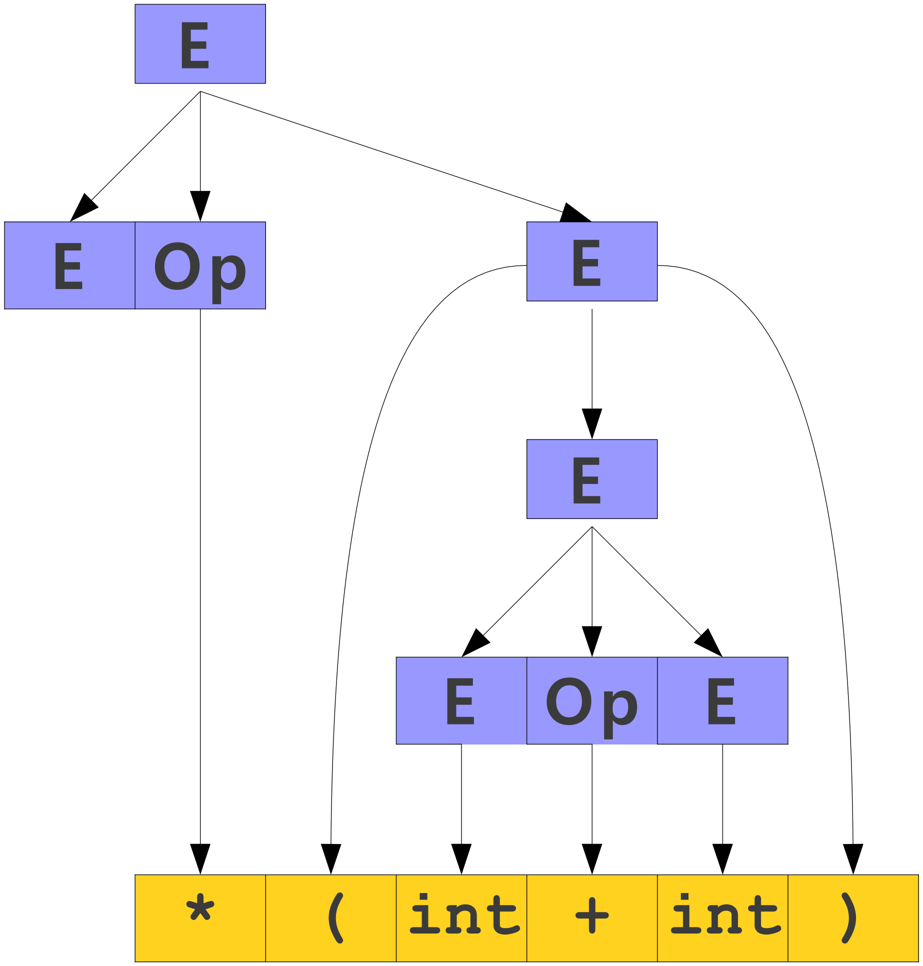
⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)

⇒ E * (int + int)



Parse Trees

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

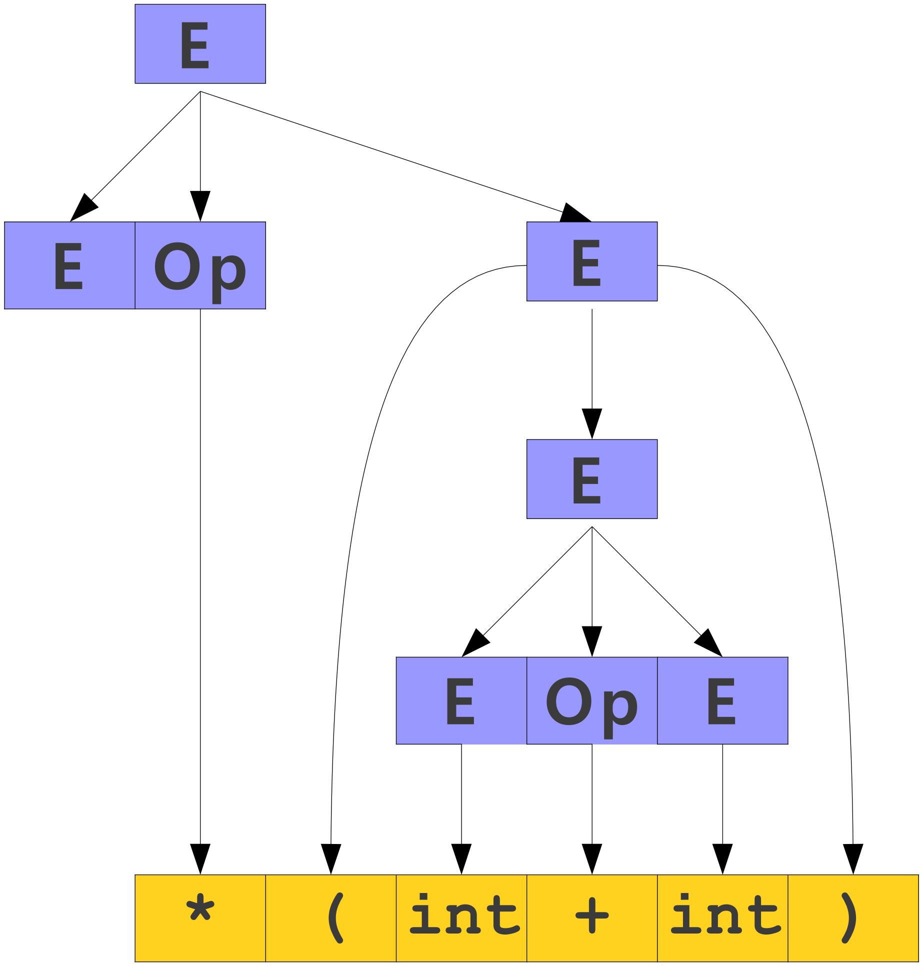
⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)

⇒ E * (int + int)

⇒ int * (int + int)



Parse Trees

E

$\Rightarrow$ E Op E

$\Rightarrow$ E Op (E)

$\Rightarrow$ E Op (E Op E)

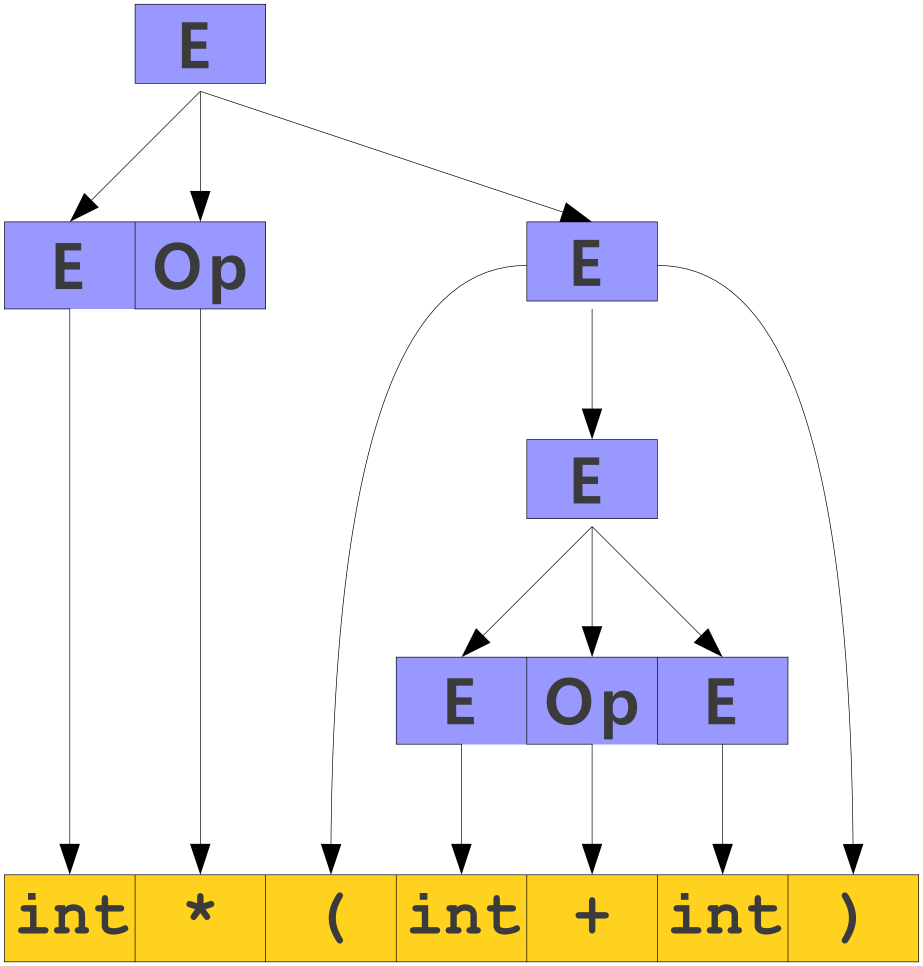
$\Rightarrow$ E Op (E Op int)

$\Rightarrow$ E Op (E + int)

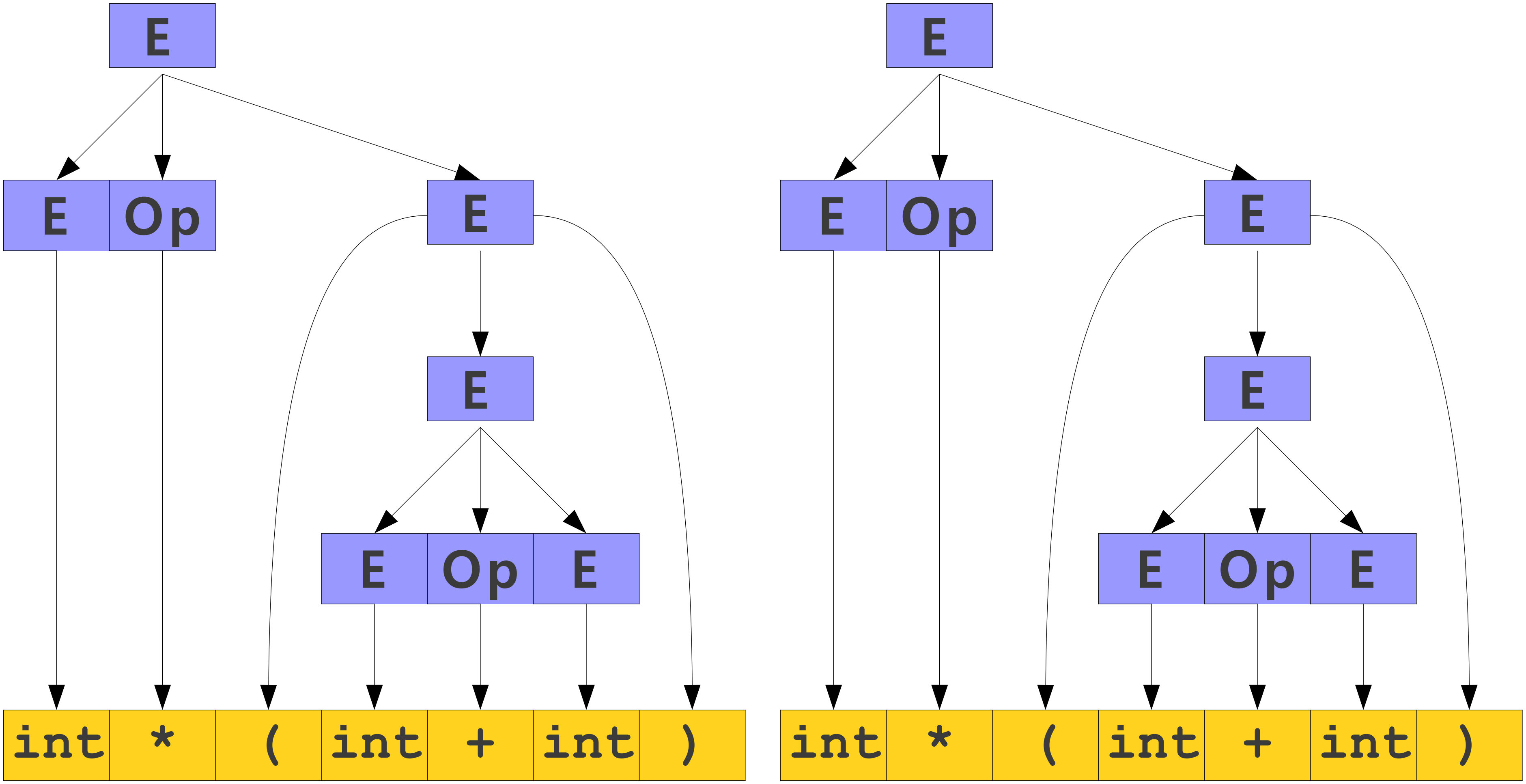
$\Rightarrow$ E Op (int + int)

$\Rightarrow$ E * (int + int)

$\Rightarrow$ int * (int + int)



For Comparison



Parse Trees

- A parse tree is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Inorder walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

The Goal of Parsing

- Goal of syntax analysis: Recover the **structure** described by a series of tokens.
- If language is described as a CFG, goal is to recover a parse tree for the the input string.
  - Usually we do some simplifications on the tree; more on that later.
- We'll discuss how to do this next week.

Exercise
Simple example for PLs

<expr> ::= <expr> <operator> <expr> |<var>

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

<var> ::= a | b | c | …

<var> ::= <signed number>

<signed number> ::= + <number> | - <number>

<number> ::= <number> <digit> | <digit>

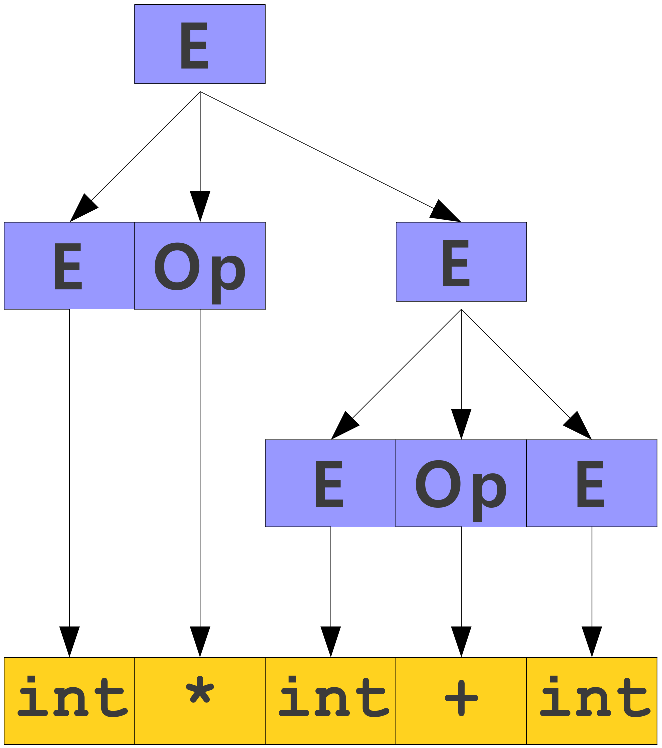
Ambiguity (Belirsizlik) in Grammars

- A grammar is ***ambiguous*** if and only if it generates a sentential form that has two or more distinct parse trees

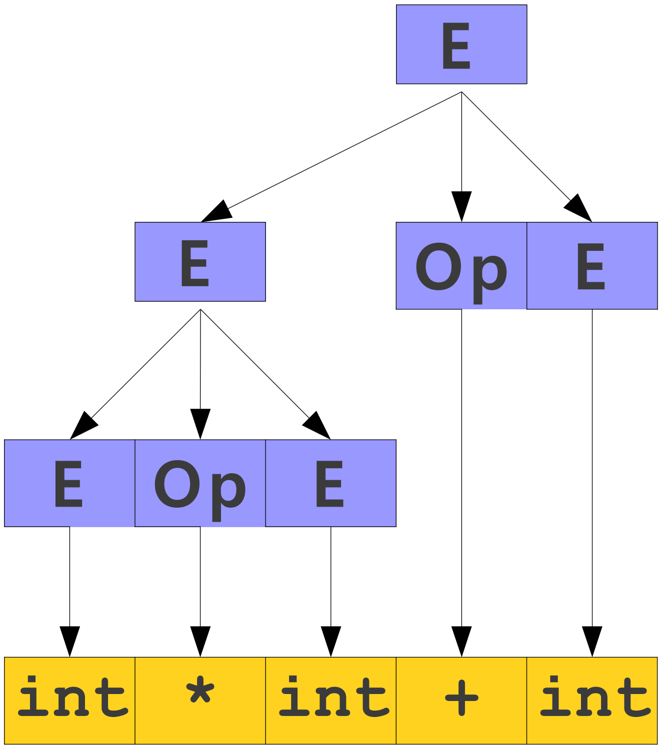
A Serious Problem

E → int | E Op E | (E)

Op → + | - | * | /



int * (int + int)



(int * int) + int

Is Ambiguity a Problem?

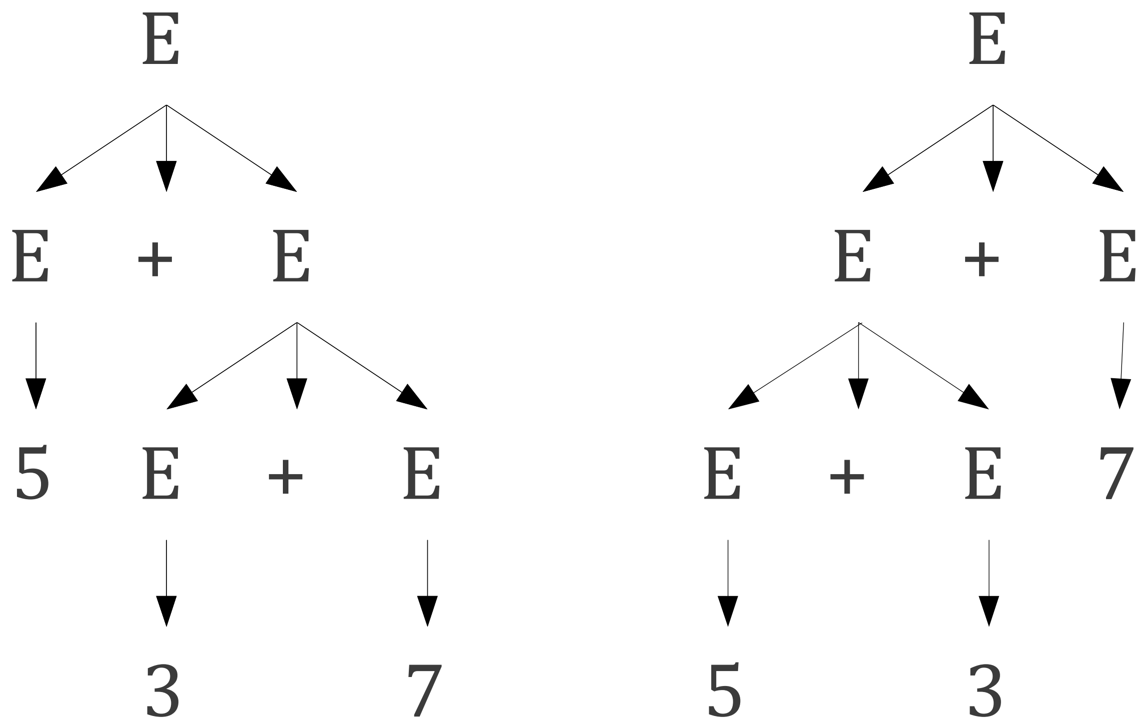
- Depends on **semantics**.

$$\mathbf{E} \rightarrow \mathtt{int} \mid \mathbf{E} \ \mathtt{+} \ \mathbf{E}$$

Is Ambiguity a Problem?

- Depends on **semantics**.

$$E \rightarrow \texttt{int} \mid E + E$$



Is Ambiguity a Problem?

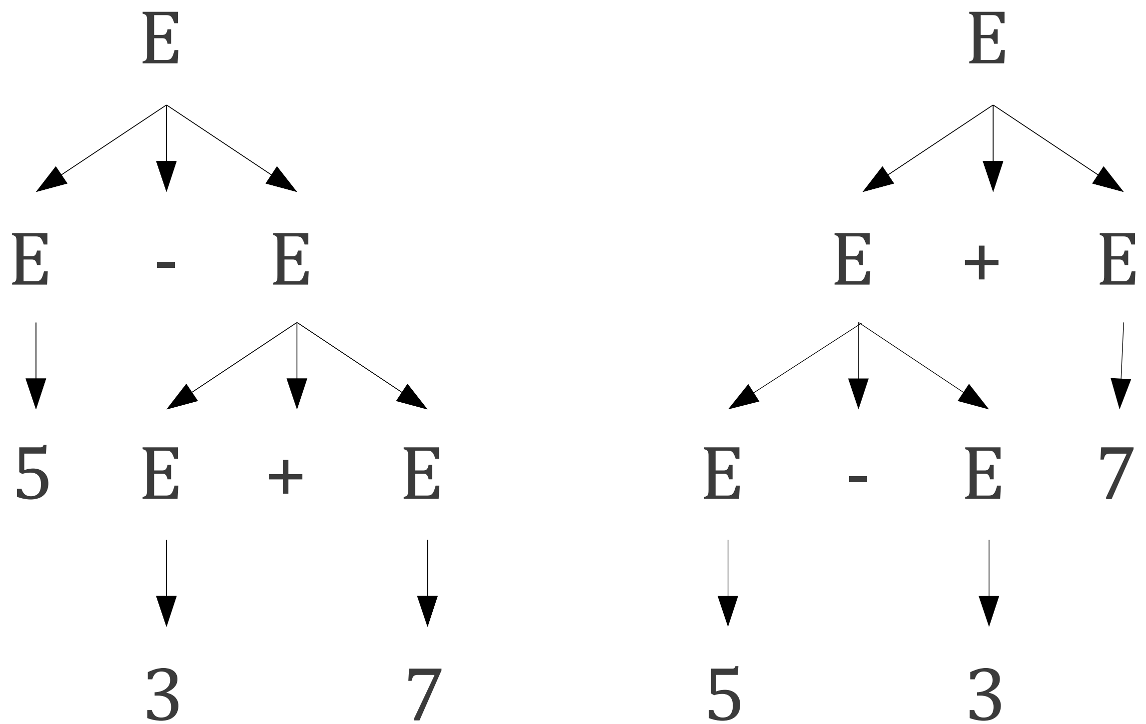
- Depends on **semantics**.

$$\mathbf{E} \rightarrow \mathtt{int} \mid \mathbf{E} + \mathbf{E} \mid \mathbf{E} - \mathbf{E}$$

Is Ambiguity a Problem?

- Depends on **semantics**.

$$E \rightarrow \texttt{int} \mid E + E \mid E - E$$



Example

- Given the following grammar

<assign> ::= <id> = <expr>

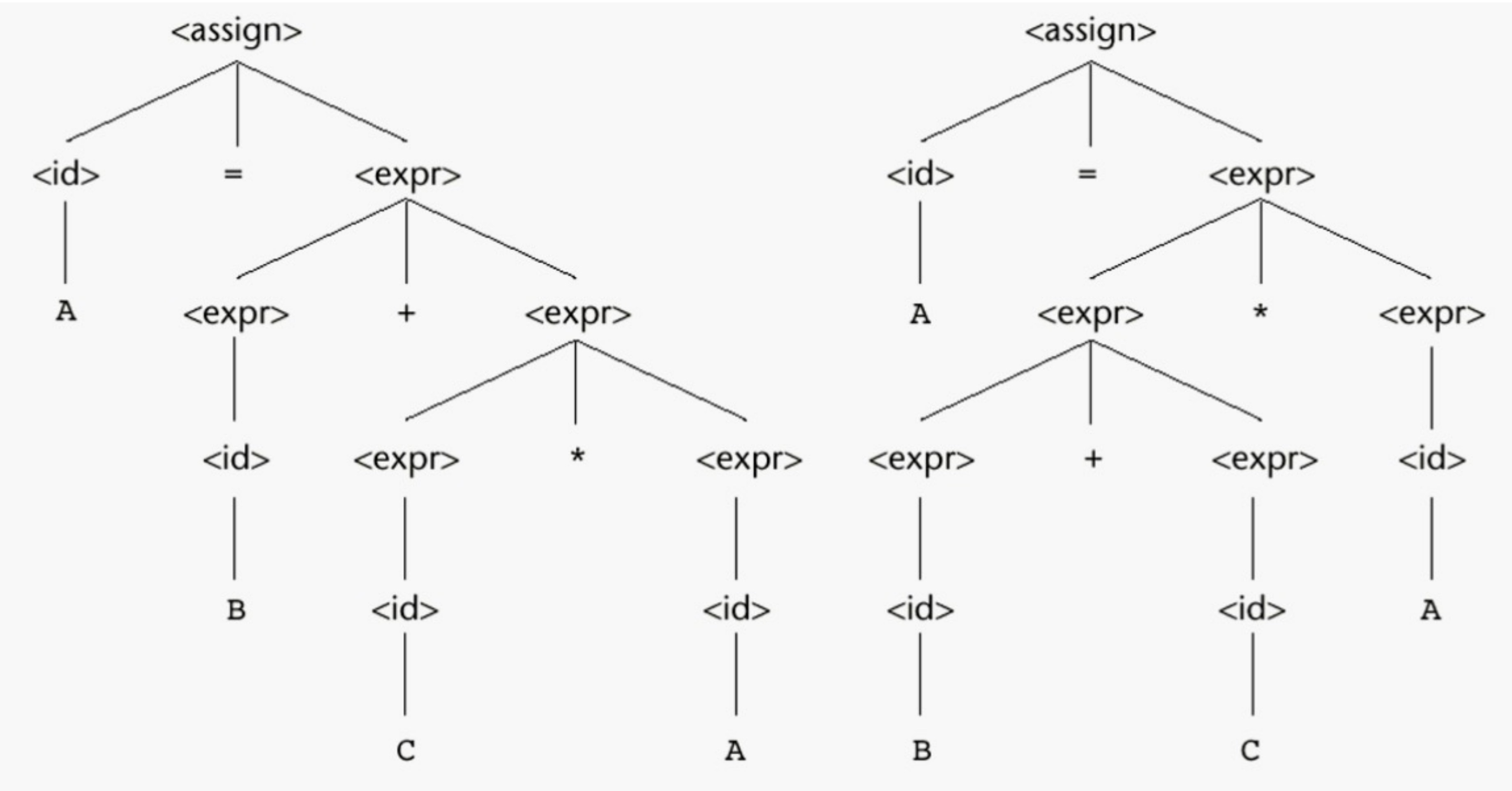
<id> ::= A | B | C

<expr> ::= <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Parse Tree(s) for A = B + C * A

Two Parse Trees for A = B + C * A

<assign> ::= <id> = <expr>
<id> ::= A | B | C
<expr> ::= <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>



Two Leftmost derivations for A = B + C * A

```
<assign> ::= <id> = <expr>
<id> ::= A | B | C
<expr> ::=  <expr> + <expr>
          | <expr> * <expr>
          | (<expr>)
          | <id>
```

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <expr> + <expr>
         => A = <id> + <expr>
         => A = B + <expr>
         => A = B + <expr> * <expr>
         => A = B + <id> * <expr>
         => A = B + C * <expr>
         => A = B + C * <id>
         => A = B + C * A
```

```
<assign> => <id> = <expr>
         => A = <expr>
         => A = <expr> * <expr>
         => A = <expr> + <expr> * <expr>
         => A = <id> + <expr> * <expr>
         => A = B + <expr> * <expr>
         => A = B + <id> * <expr>
         => A = B + C * <expr>
         => A = B + C * <id>
         => A = B + C * A
```

<assign> ::= <id> = <expr>

<id> ::= A | B | C

<expr> ::= <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Two Rightmost derivations for A = B + C * A

<assign> => <id> = <expr>
=> <id> = <expr> + <expr>
=> <id> = <expr> + <expr> * <expr>
=> <id> = <expr> + <expr> * <id>
=> <id> = <expr> + <expr> * A
=> <id> = <expr> + <id> * A
=> <id> = <expr> + C * A
=> <id> = <id> + C * A
=> <id> = B + C * A
=> A = B + C * A

<assign> => <id> = <expr>
=> <id> = <expr> * <expr>
=> <id> = <expr> * <id>
=> <id> = <expr> * A
=> <id> = <expr> + <expr> * A
=> <id> = <expr> + <id> * A
=> <id> = <expr> + C * A
=> <id> = <id> + C * A
=> <id> = B + C * A
=> A = B + C * A

A = B + C * A

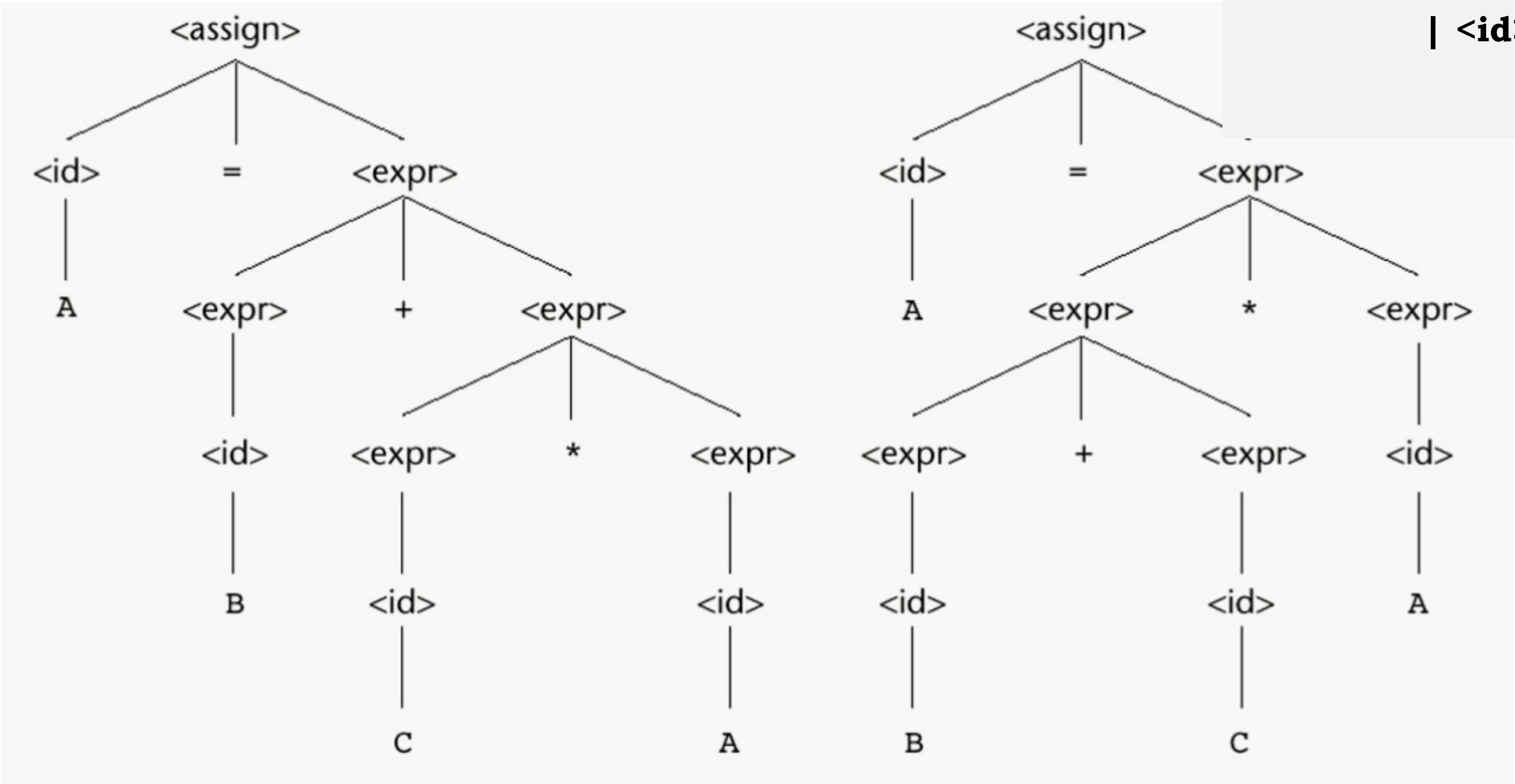
A = 3 , B = 4, C = 5

3 + 4 * 5

<assign> ::= <id> = <expr>

<id> ::= A | B | C

<expr> ::= <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>



3 + [4 * 5]

[3 + 4] * 5

A = B + C + A

A = 3 , B = 4, C = 5

3 + 4 + 5

<assign> ::= <id> = <expr>

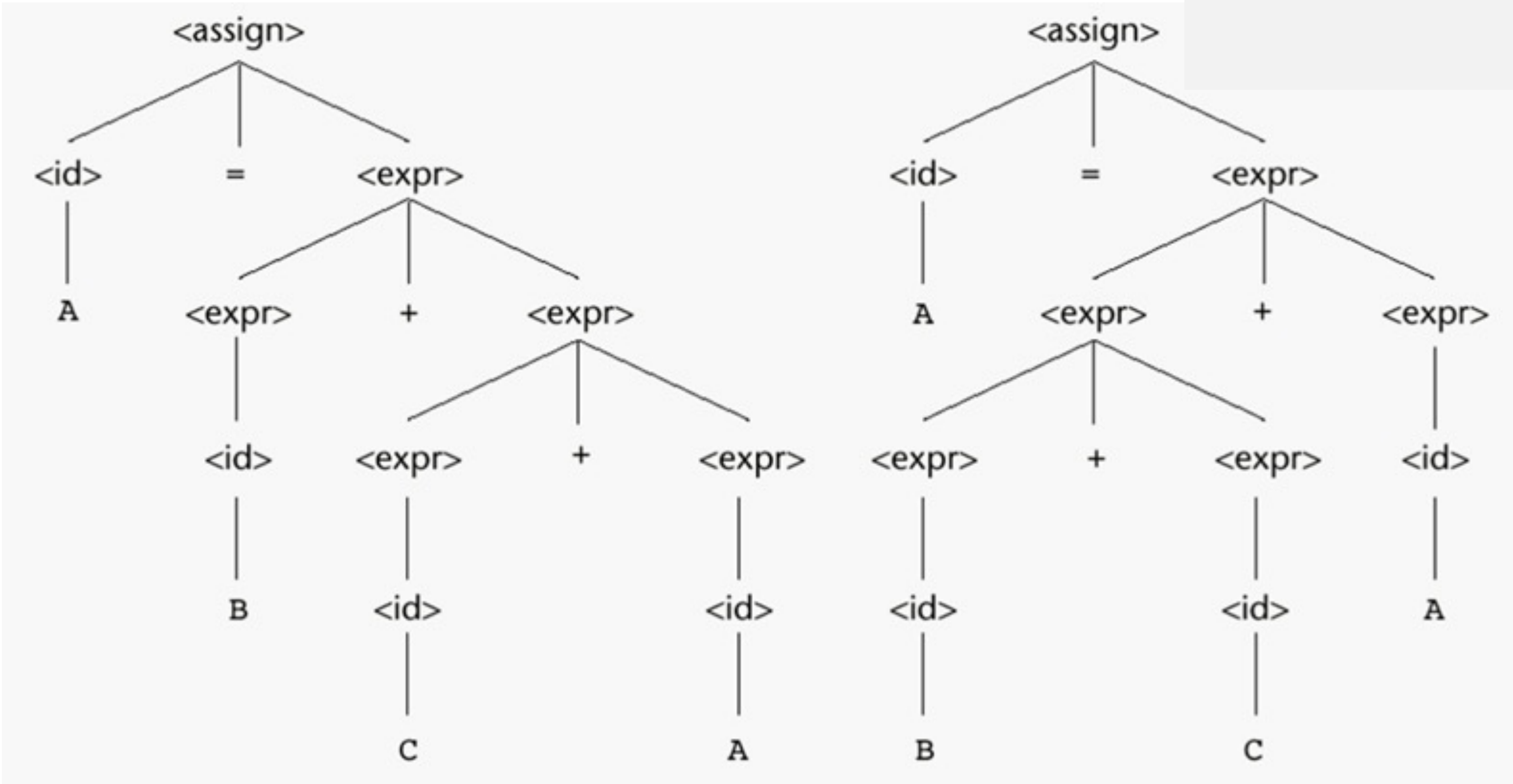
<id> ::= A | B | C

<expr> ::= <expr> + <expr>

| <expr> * <expr>

| (<expr>)

| <id>



3 + [4 + 5]

[3 + 4] + 5

Single leftmost derivation for A = B + C

<assign> ::= <id> = <expr>
<id> ::= A | B | C
<expr> ::= <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> + <expr>
=> A = <id> + <expr>
=> A = B + <expr>
=> A = B + <id>
=> A = B + C

There is also a single rightmost derivation

And a single parse tree for A = B + C

Finding at least one string with more than a single parse tree
(or more than a single leftmost derivation
Or more than a single rightmost derivation)
is sufficient to prove ambiguity of a grammar

Handling Ambiguity

- The grammar of a PL must not be ambiguous
- There are solutions for correcting the ambiguity
  - Operator precedence
  - Associativity rules

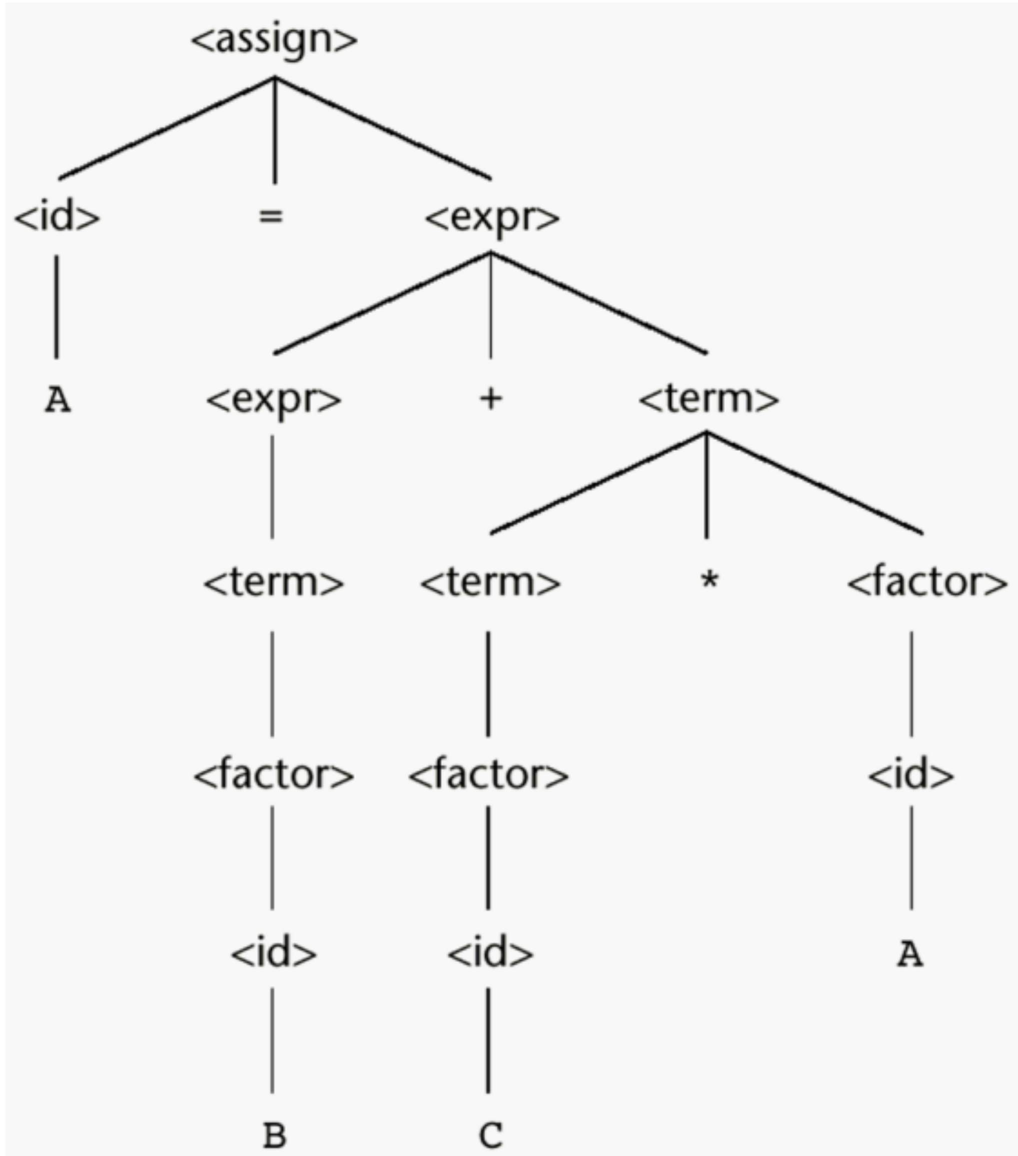
Operator Precedence

- In mathematics * operation has a higher precedence than +
- This can be implemented with extra nonterminals

```
<assign> ::= <id> = <expr>
<id> ::= A | B | C
<expr> ::= <expr> + <expr>
         | <expr> * <expr>
         | (<expr>)
         | <id>
```

```
<assign> ::= <id> = <expr>
<id> ::= A | B | C
<expr> ::= <expr> + <term>
         | <term>
<term> ::= <term> * <factor>
         | <factor>
<factor> ::= (<expr>)
           | <id>
```

Unique Parse Tree for A = B + C * A



Associativity of Operators

- What about equal precedence operators?

- In math addition and multiplication are associative
  A+B+C = (A+B)+C = A+(B+C)

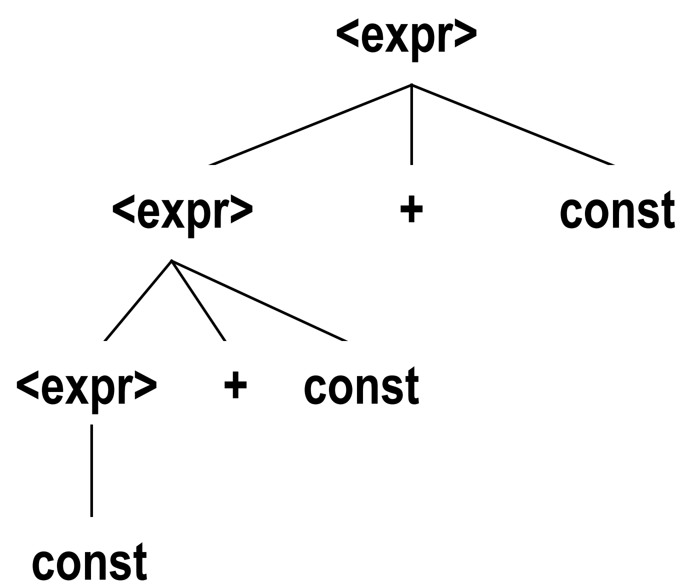
- However computer arithmetic may not be associative
- Ex: for floating point addition where floating points values store 7 digits of accuracy, adding eleven numbers together where one of the numbers is 10^7 and the others are 1 result would be $1.000001 * 10^7$ only if the ten 1s are added first

- Subtraction and division are not associative
  A/B/C/D = ?   ((A/B)/C)/D ≠A/(B/(C/D))

Associativity of Operators

- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Associativity

- In a BNF rule, if the LHS appears at the beginning of the RHS, the rule is said to be left recursive
- Left recursion specifies left associativity

<expr> ::= <expr> + <term>

| <term>

- Similar for the right recursion
- In most of the languages exponential is defined as a right associative operation

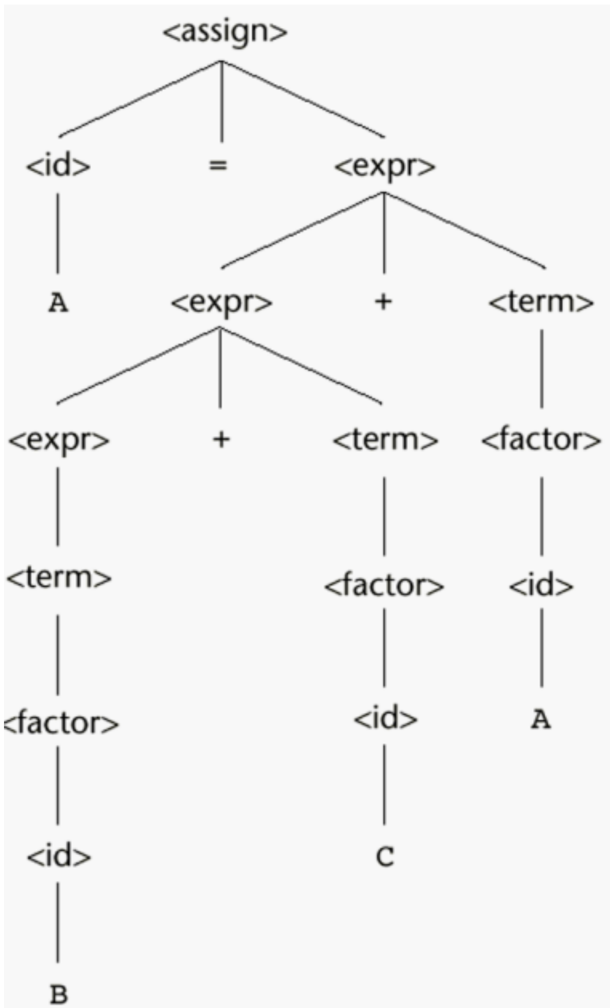
<factor> ::= <expr> ** <factor>

| <expr>

<expr> ::= (<expr>)

| <id>

A parse tree for A = B + C + A illustrating the associativity of addition



Left associativity
Left addition is lower than the right addition

Is this ambiguous?

<stmt> ::= <if_stmt> | <other_stmt>

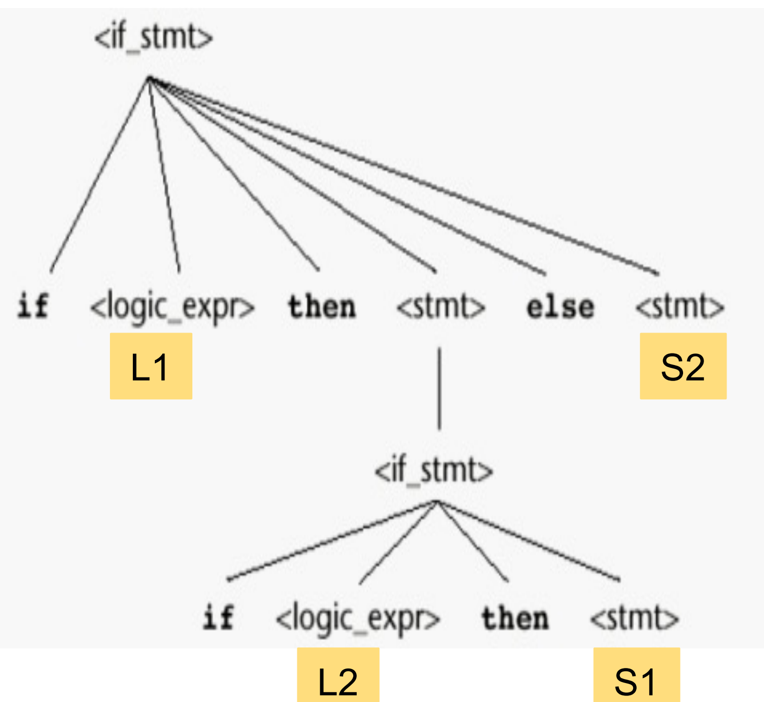
<if_stmt> ::= if <logic_expr> then <stmt>

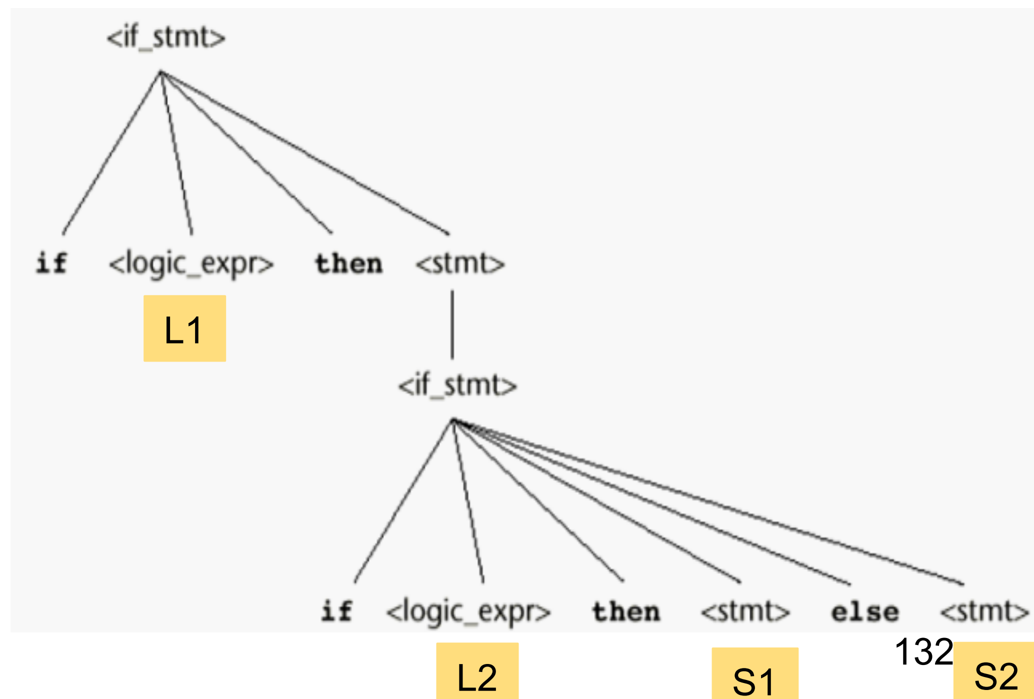
| if <logic_expr> then <stmt> else <stmt>

<other_stmt> ::= S1 | S2

<logic_expr> ::= L1 | L2

Derive for : If L1 then if L2 then S1 else S2





An Unambiguous grammar for "if then else"

- Dangling else problem: there are more if then else
- To design an unambiguous if-then-else statement we have to decide which `if` a dangling `else` belongs to
- Most PL adopt the following rule:
  - "an else is matched with the closest previous unmatched if statement"
  - (unmatched if = else-less if)

Has a unique parse tree

```
<stmt> ::= <matched> | <unmatched>
<matched> ::= if <logic_expr> then <matched> else <matched>
    | any non-if-statement
<unmatched> ::= if <logic_expr> then <stmt>
   | if <logic_expr> then <matched> else <unmatched>
```

Draw the parse tree

<stmt> ::= <matched> | <unmatched>
<matched> ::= if <logic_expr> then <matched> else <matched>
| <other_stmt>
<unmatched> ::= if <logic_expr> then <stmt>
| if <logic_expr> then <matched> else <unmatched>

If L1 then if L2 then S1 else S2