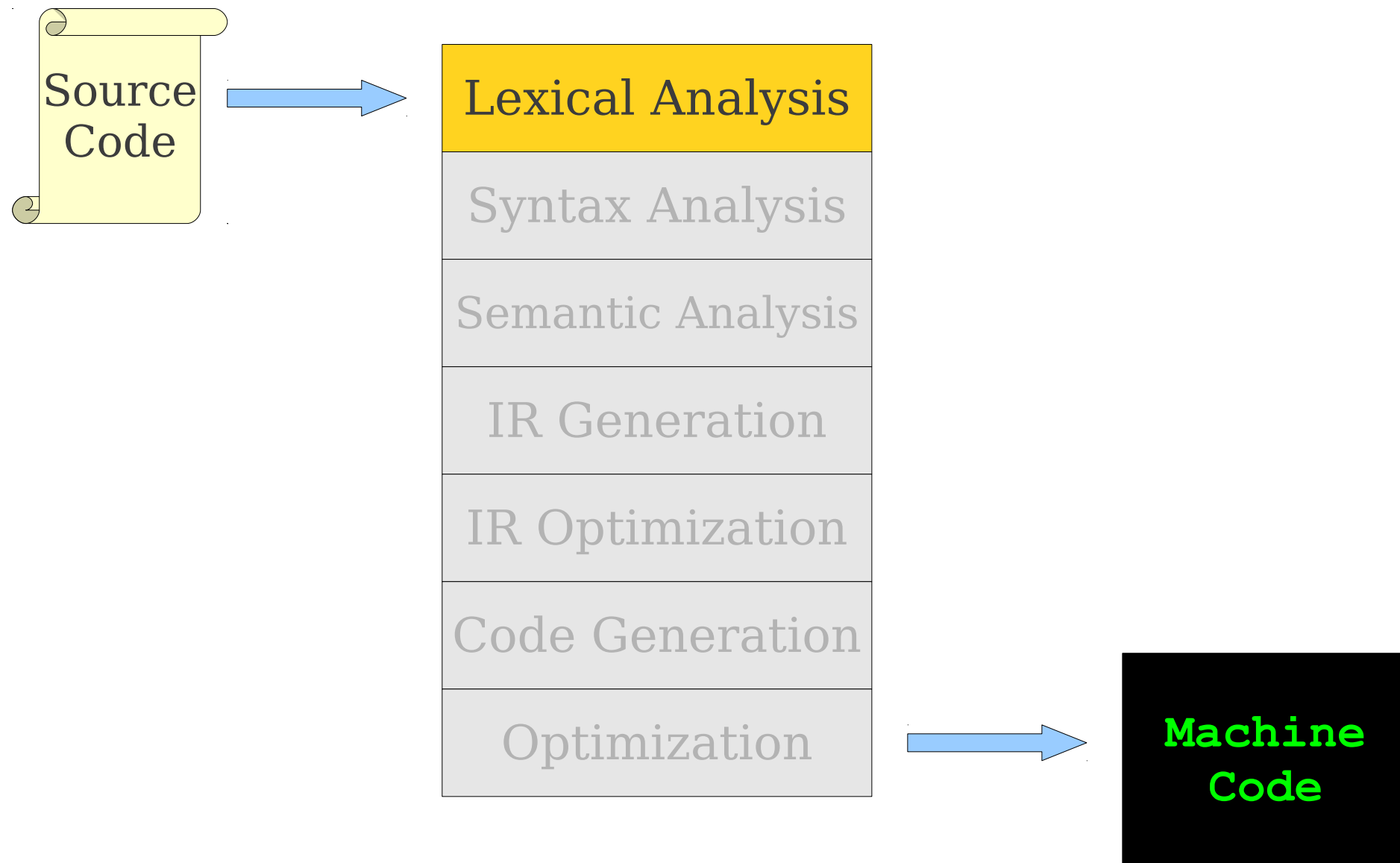


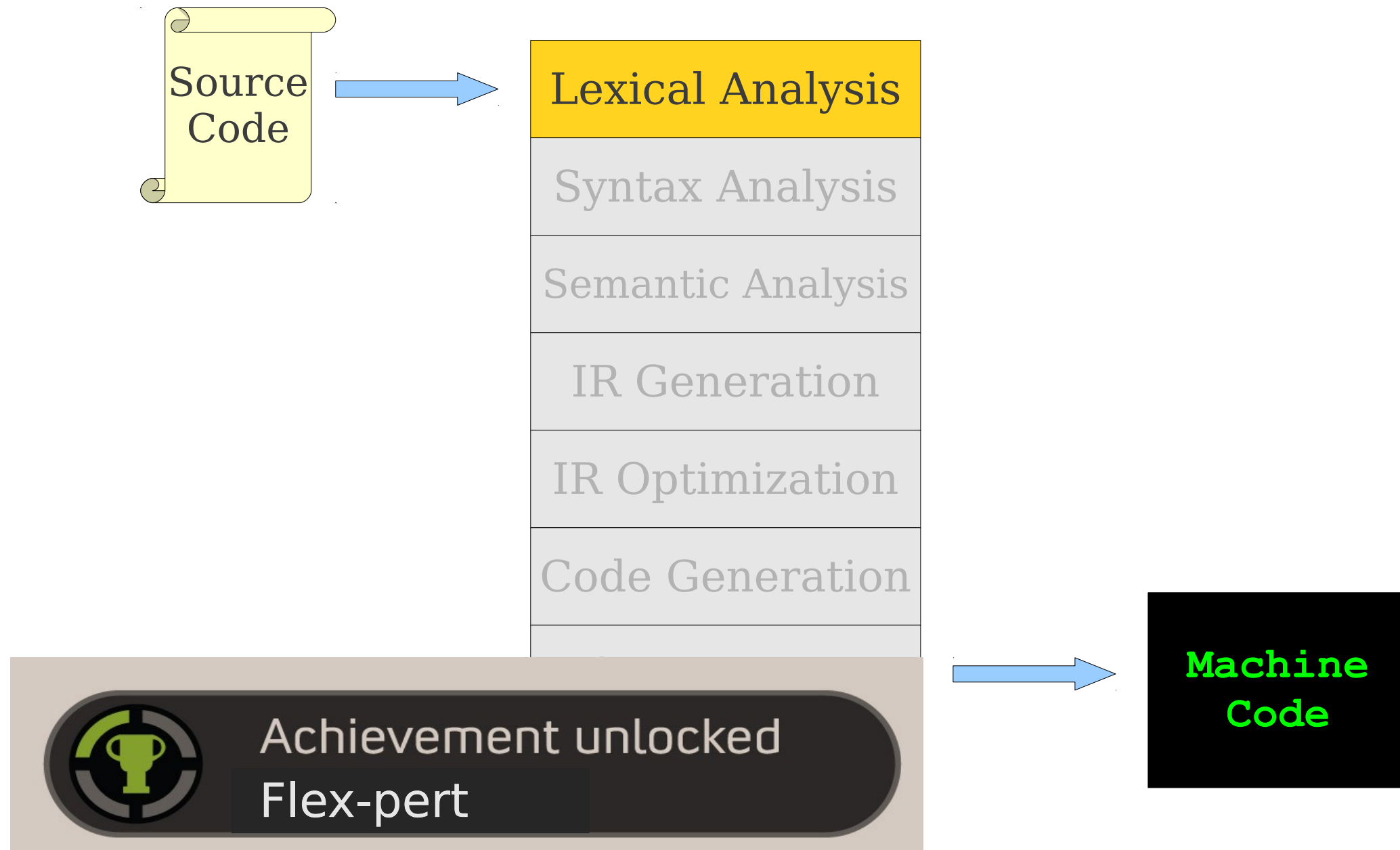
Syntax Analysis (parsing)

April 8, 2013

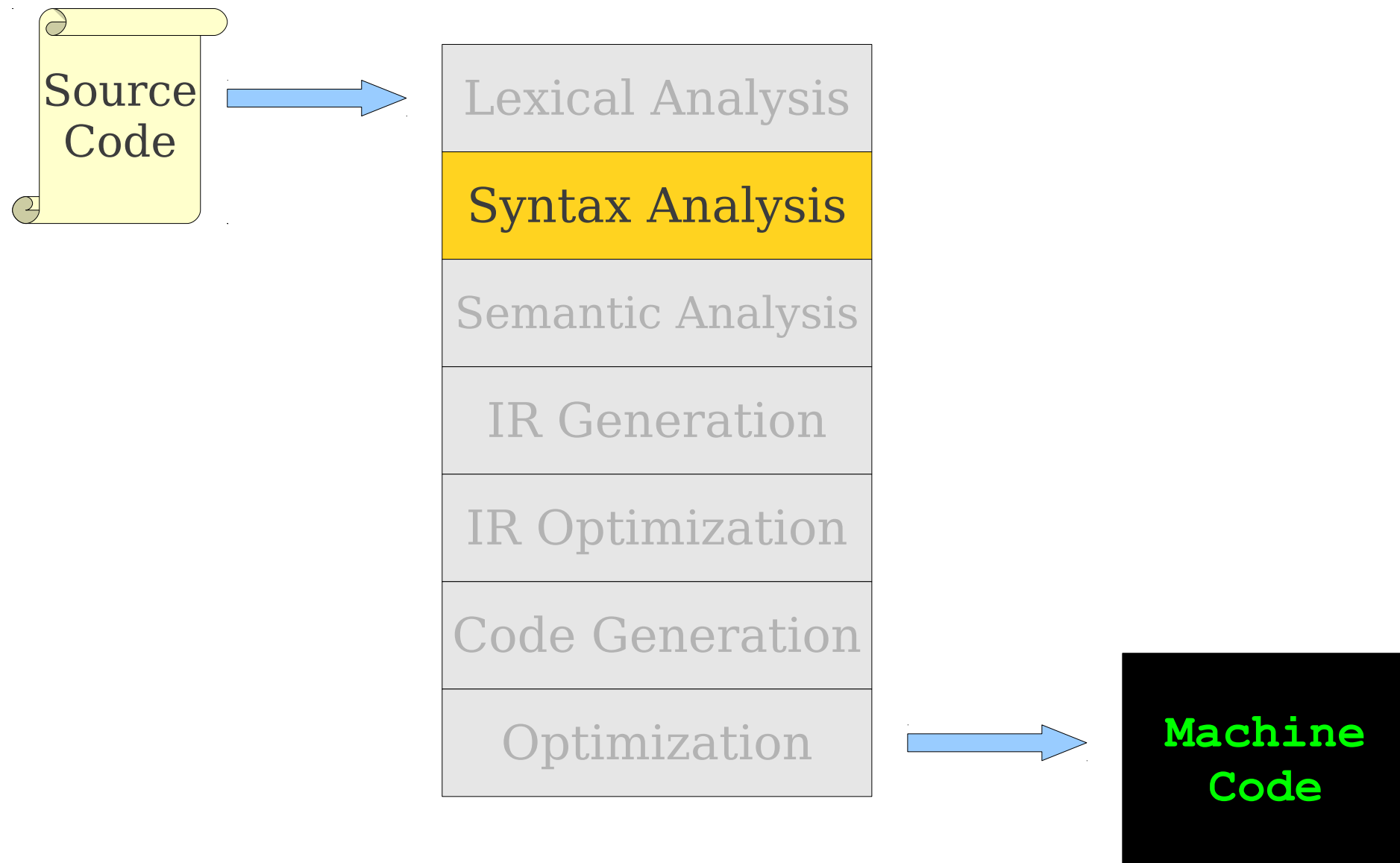
Where We Are



Where We Are



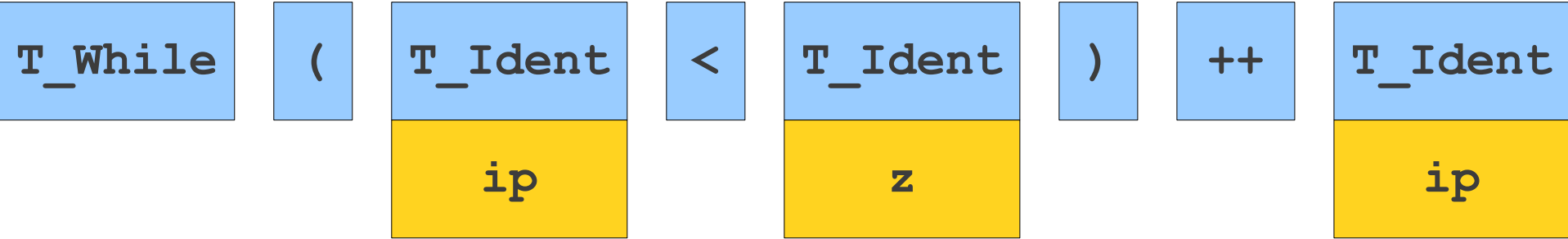
Where We Are



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

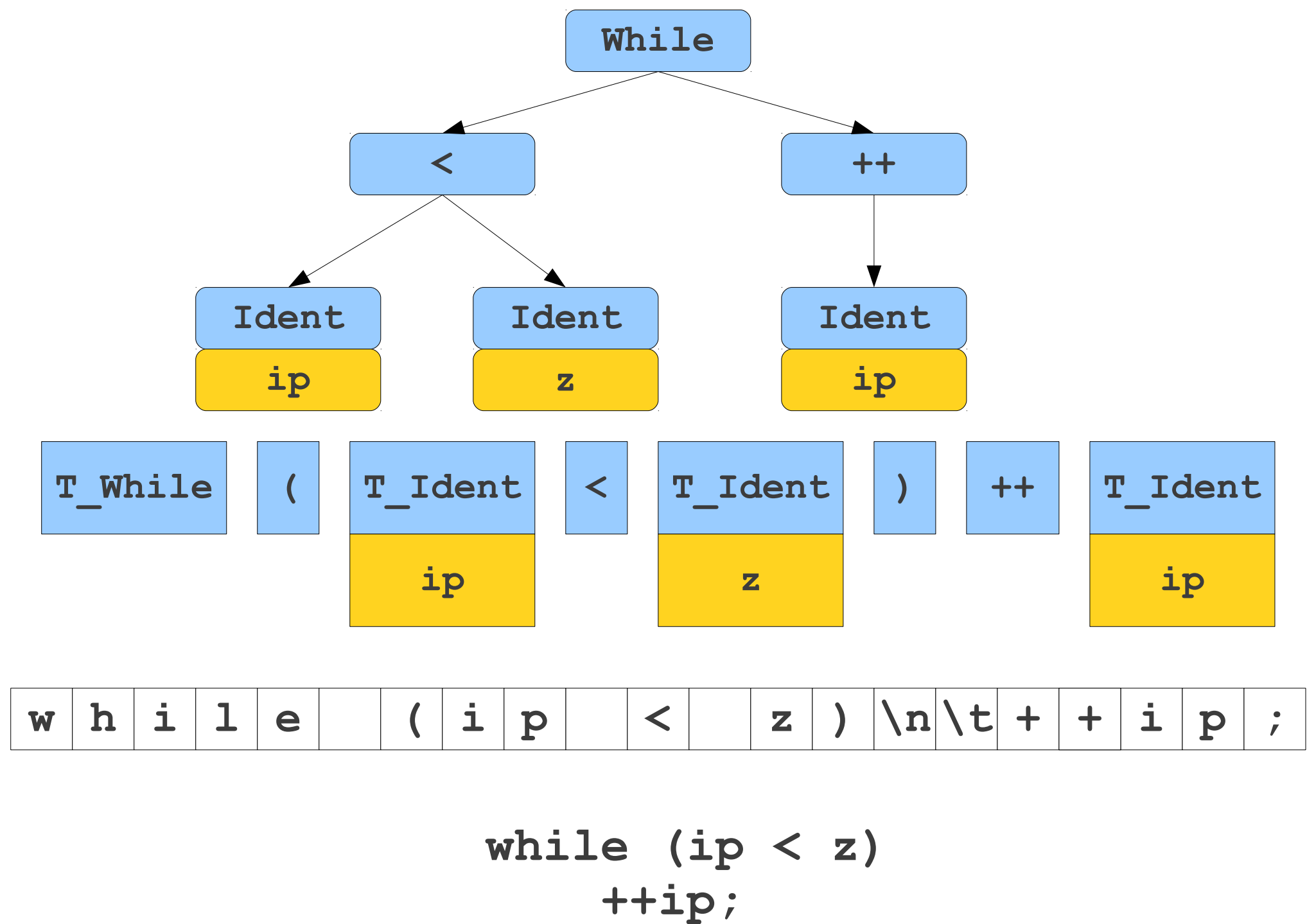
w	h	i	l	e		(	i	p		<		z	)	\n	\t	+	+	i	p	;
---	---	---	---	---	--	---	---	---	--	---	--	---	---	----	----	---	---	---	---	---

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



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

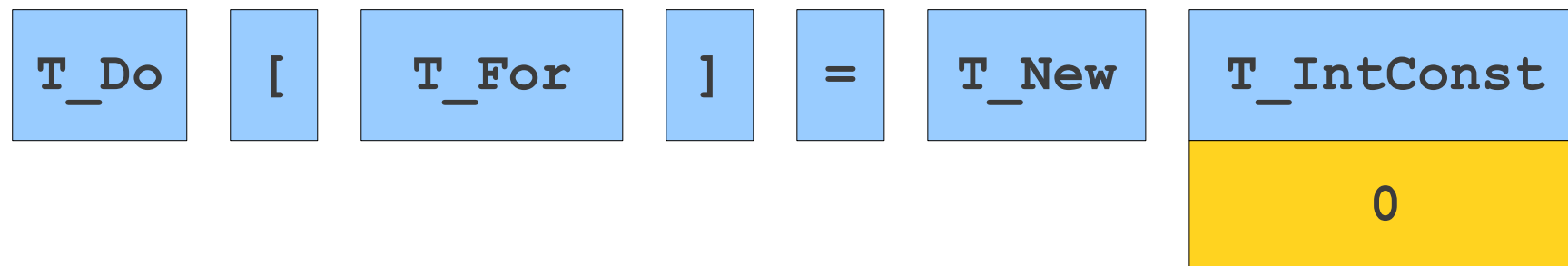
```
while (ip < z)
    ++ip;
```



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

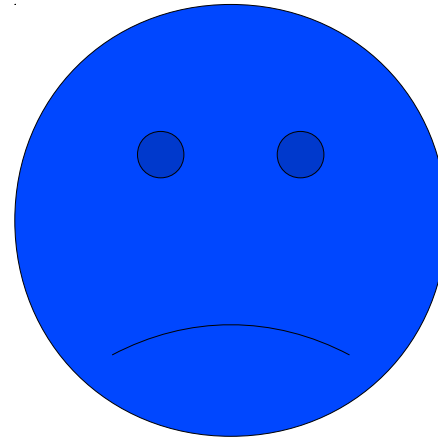
d	o	[	f	o	r	]		=		n	e	w		0	;
---	---	---	---	---	---	---	--	---	--	---	---	---	--	---	---

do[for] = new 0;



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

`do[for] = new 0;`



T_Do	[	T_For	]	=	T_New	T_IntConst
						0

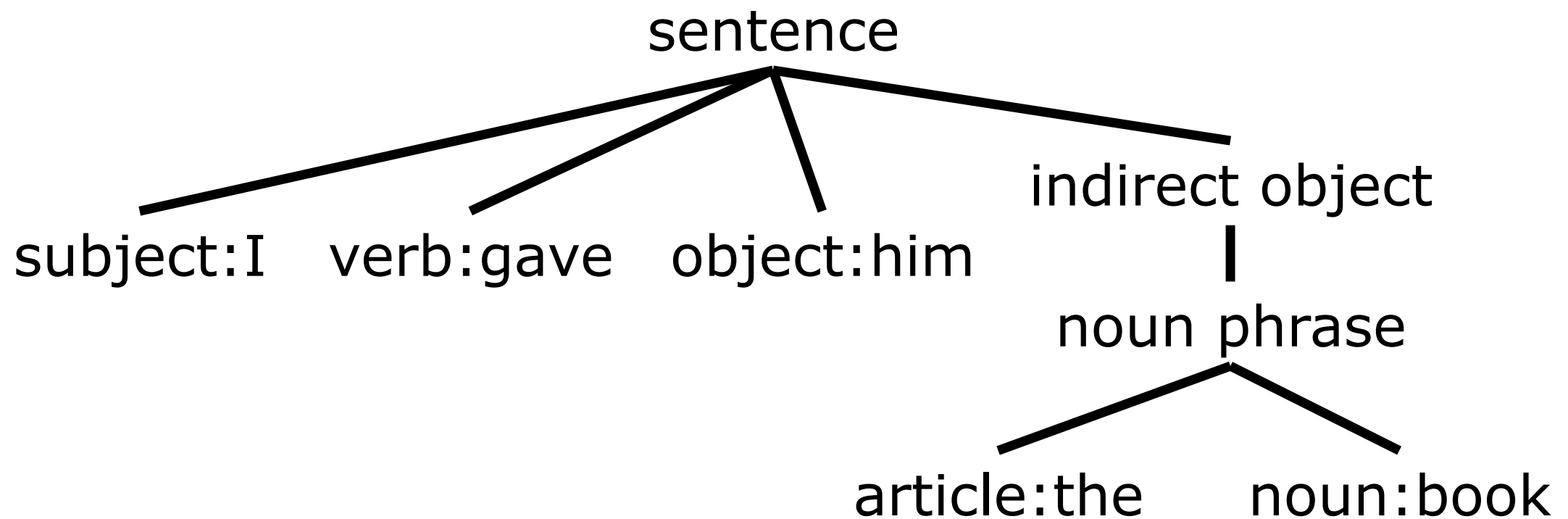
d	o	[	f	o	r	]		=		n	e	w		0	;
---	---	---	---	---	---	---	--	---	--	---	---	---	--	---	---

do[for] = new 0;

What is Parsing?

- Parsing: Recognizing whether a sentence (or program) is grammatically well formed and identifying the function of each component

“I gave him the book”

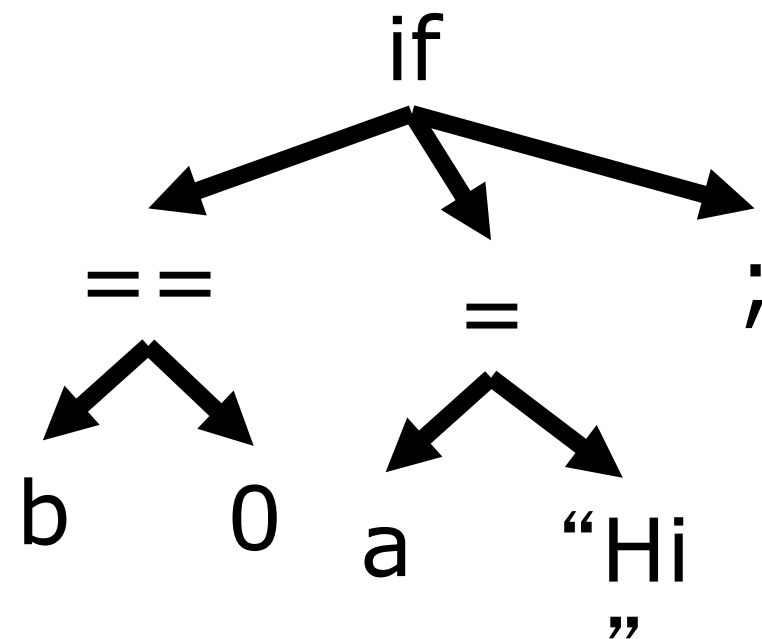


Overview of Syntax Analysis

- Input: stream of tokens from the lexer
- Output: Abstract Syntax Tree (AST)

Source code: `if (b==0) a = "Hi";`

AST:



Goal of Syntax Analysis

- Recovered the **structure** described by the token stream
- Report errors if the tokens do not properly encode a structure

What Parsing Doesn't Do

- Doesn't check: type agreement, variable declaration, variables initialization, etc.
- `int x = true;`
- `int y;`
- `z = f(y);`
- Deferred until semantic analysis

Outline

- Today: Formalism for syntax analysis
 - Grammars
 - Derivation
 - Ambiguity
 - Concrete vs. Abstract Syntax Tree
- Wednesday: Parsing algorithms
 - Top Down
 - Bottom up

Specify Language Syntax

Specify Language Syntax

- First problem: how to describe language syntax precisely and conveniently

Specify Language Syntax

- First problem: how to describe language syntax precisely and conveniently
- Last time: can describe tokens using ???

Specify Language Syntax

- First problem: how to describe language syntax precisely and conveniently
- Last time: can describe tokens using ???
- Regular expressions easy to implement, efficient (by converting to DFA)

Specify Language Syntax

- First problem: how to describe language syntax precisely and conveniently
- Last time: can describe tokens using ???
- Regular expressions easy to implement, efficient (by converting to DFA)
- Can we use regular expressions (on tokens) to specify programming language syntax?

Specify Language Syntax

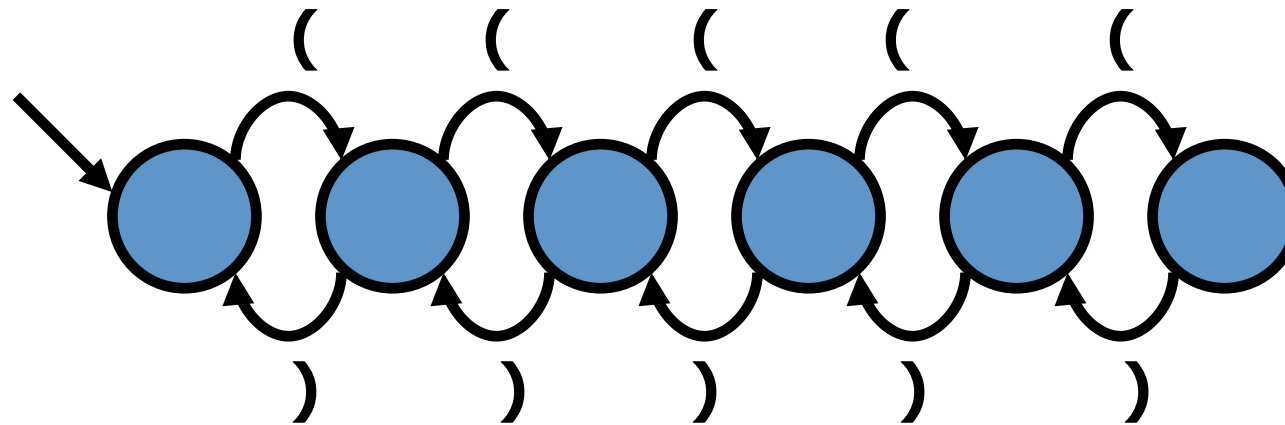
- First problem: how to describe language syntax precisely and conveniently
- Last time: can describe tokens using ???
- Regular expressions easy to implement, efficient (by converting to DFA)
- Can we use regular expressions (on tokens) to specify programming language syntax?

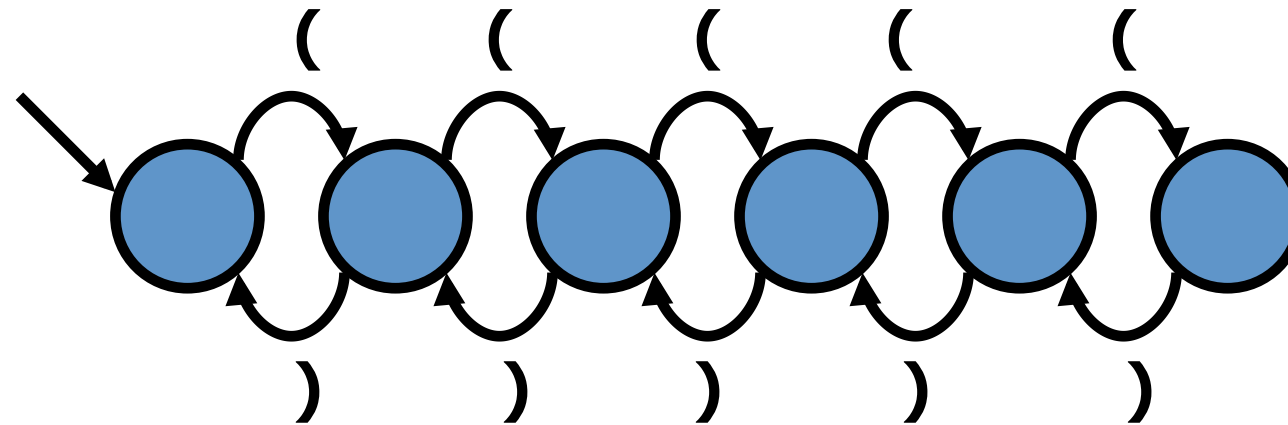
Specify Language Syntax

- First problem: how to describe language syntax precisely and conveniently
- Last time: can describe tokens using ???
- Regular expressions easy to implement, efficient (by converting to DFA)
- Can we use regular expressions (on tokens) to specify programming language syntax?

To answer this...

- Consider: language of all strings that contain balanced parentheses
 - $()$ $(())$ $(())()$ $(())()(())()$
 - $(())()$ $(())$ $(())()$
- Construct a Finite Automaton for this...?





- Limits of regular language: DFA has only finite number of states; cannot perform unbounded counting
- Need a More Powerful Representation

Context Free Grammar (CFG)

Context Free Grammar (CFG)

- A specification of the balanced-parenthesis language:
 - $S \rightarrow (S) S$
 - $S \rightarrow \varepsilon$
- The definition is recursive
- A context-free grammar: more expressive than regular expressions
 - $S \Rightarrow (S) \varepsilon \Rightarrow ((S) S) \varepsilon \Rightarrow ((\varepsilon) \varepsilon) \varepsilon \Rightarrow (())$
- If a grammar accepts a string, there is a derivation of that string using the productions of the grammar

Formal Language Terminology Review

Formal Language Terminology Review

- **Language** is a set of **strings**

Formal Language Terminology Review

- **Language** is a set of **strings**
- each **string** is a finite sequence of **symbols** taken from a finite **alphabet**

Formal Language Terminology Review

- **Language** is a set of **strings**
- each **string** is a finite sequence of **symbols** taken from a finite **alphabet**
- **parsing:**
 - the strings are source program,
 - the symbols are lexical tokens,
 - the alphabet is the set of token-types returned by the lexical analyzer
 - CFG describes a language

Formal Language Terminology Review

- **Language** is a set of **strings**
- each **string** is a finite sequence of **symbols** taken from a finite **alphabet**
- **parsing:**
 - the strings are source program,
 - the symbols are lexical tokens,
 - the alphabet is the set of token-types returned by the lexical analyzer
 - CFG describes a language

For scanning, alphabet is the ASCII

CFG Terminology

- **Terminals**

- Token or ϵ

- **Non-terminals**

- variables

- **Start symbol**

- Begins the derivation

- **Productions**

- Specify how non-terminals may be expanded to form strings (replacement rules)
- LHS: single non-terminal, RHS: string of terminals (including ϵ) or non-terminals

- $S \rightarrow (S) S$

- $S \rightarrow \epsilon$

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 → **/**

Non-terminal Symbols

Terminal Symbols

Production Rules

Start Symbols

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 (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)**

A Notational Shorthand

$E \rightarrow \text{int}$

$E \rightarrow E \text{ Op } E$

$E \rightarrow (E)$

$\text{Op} \rightarrow +$

$\text{Op} \rightarrow -$

$\text{Op} \rightarrow *$

$\text{Op} \rightarrow /$

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

- Vertical bar $\mid$ is shorthand for multiple productions

CFGs for Programming Languages

BLOCK	→	STMT { STMTS }
STMTS	→	ε STMT STMTS
STMT	→	EXPR; if (EXPR) BLOCK while (EXPR) BLOCK do BLOCK while (EXPR); BLOCK ...
EXPR	→	identifier constant EXPR + EXPR EXPR - EXPR EXPR * EXPR ...

RegExp: A Subset of CFG

- Regular expression definition of real numbers:
- $digit \rightarrow [0-9]$
- $posint \rightarrow digit^+$
- $int \rightarrow -? posint$
- $real \rightarrow int . (\epsilon \mid posint)$
- RE symbolic names are only *shorthand*: no recursion, so all symbols can be fully expanded:
- $real \rightarrow -? [0-9]^+ . (\epsilon \mid ([0-9]^+))$

Some CFG Notation

- Capital letters at the beginning of the alphabet will represent nonterminals.
 - i.e. **A, B, C, D**
- Lowercase letters at the end of the alphabet will represent terminals.
 - i.e. **t, u, v, w**
- Lowercase Greek letters will represent arbitrary strings of terminals and nonterminals.
 - i.e. **α , γ , ω**

Examples

- We might write an arbitrary production as

$$\mathbf{A} \rightarrow \omega$$

- We might write a string of a nonterminal followed by a terminal as

$$\mathbf{A}t$$

- We might write an arbitrary production containing a nonterminal followed by a terminal as

$$\mathbf{B} \rightarrow \alpha \mathbf{A}t\omega$$

Derivation

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$
 $\Rightarrow E * (E \text{ Op } E)$
 $\Rightarrow \text{int} * (E \text{ Op } E)$
 $\Rightarrow \text{int} * (\text{int Op } E)$
 $\Rightarrow \text{int} * (\text{int Op int})$
 $\Rightarrow \text{int} * (\text{int} + \text{int})$

- This sequence of steps is called a **derivation**.
- A string $\alpha A \omega$ **yields** string $\alpha \gamma \omega$ iff $A \rightarrow \gamma$ 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
$$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \beta$$
- If α derives β , we write $\alpha \Rightarrow^* \beta$.

Derivation

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$
 $\Rightarrow E * (E \text{ Op } E)$
 $\Rightarrow \text{int} * (E \text{ Op } E)$
 $\Rightarrow \text{int} * (\text{int} \text{ Op } E)$
 $\Rightarrow \text{int} * (\text{int} \text{ Op } \text{int})$
 $\Rightarrow \text{int} * (\text{int} + \text{int})$

- This sequence of steps is called a **derivation**.
- A string $\alpha A \omega$ **yields** string $\alpha \gamma \omega$ iff $A \rightarrow \gamma$ 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
$$\alpha \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \beta$$
- If α derives β , we write $\alpha \Rightarrow^* \beta$.

- Terminals: no replacement rules for them
- Terminals: tokens from the lexer

- Which of the strings are in the language of the given CFG?

- abcba
- acca
- aba
- abcbcbca

- $S \rightarrow aXa$

- $X \rightarrow \varepsilon \mid bY$

- $Y \rightarrow \varepsilon \mid cXc$

Leftmost Derivations

BLOCK	→	STMT { STMTS }	
STMTS	→	ε STMT STMTS	STMTS
STMT	→	EXPR; if (EXPR) BLOCK while (EXPR) BLOCK do BLOCK while (EXPR); BLOCK ...	⇒ STMT STMTS
EXPR	→	identifier constant EXPR + EXPR EXPR - EXPR EXPR * EXPR EXPR = EXPR ...	⇒ EXPR; STMTS ⇒ EXPR = EXPR; STMTS ⇒ id = EXPR; STMTS ⇒ id = EXPR + EXPR; STMTS ⇒ id = id + EXPR; STMTS ⇒ id = id + constant; STMTS ⇒ id = id + constant;

Productions

Derivations

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.

Related Derivations

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)**

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.

- **Derivation: also a process of constructing a parse tree**

Parse Trees

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

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

E

Parse Trees

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

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

E

E

Parse Trees

E

E
⇒ **E Op E**

E → **int** | **E Op E** | **(E)**

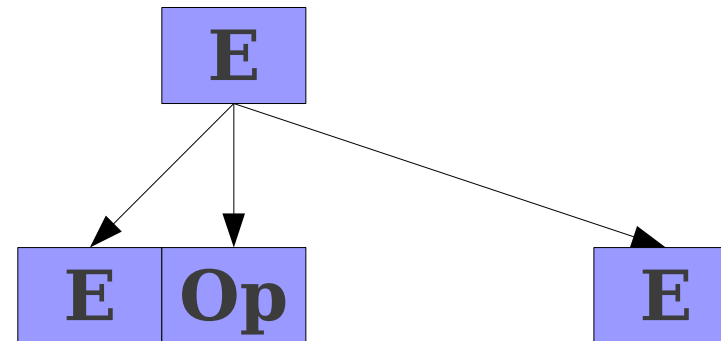
Op → **+** | **-** | ***** | **/**

Parse Trees

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

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

E
 $\Rightarrow$ **E Op E**

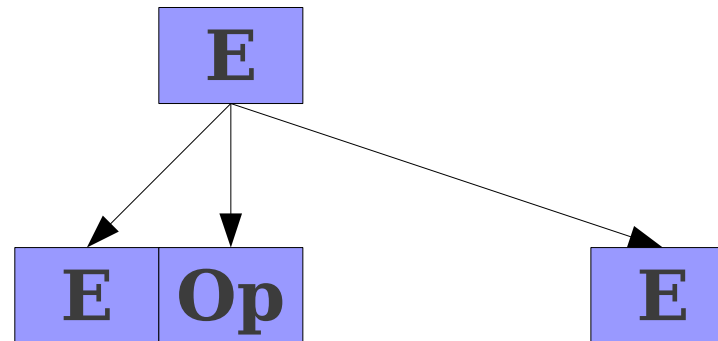


Parse Trees

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

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

E
 $\Rightarrow$ **E Op E**
 $\Rightarrow$ **int Op E**

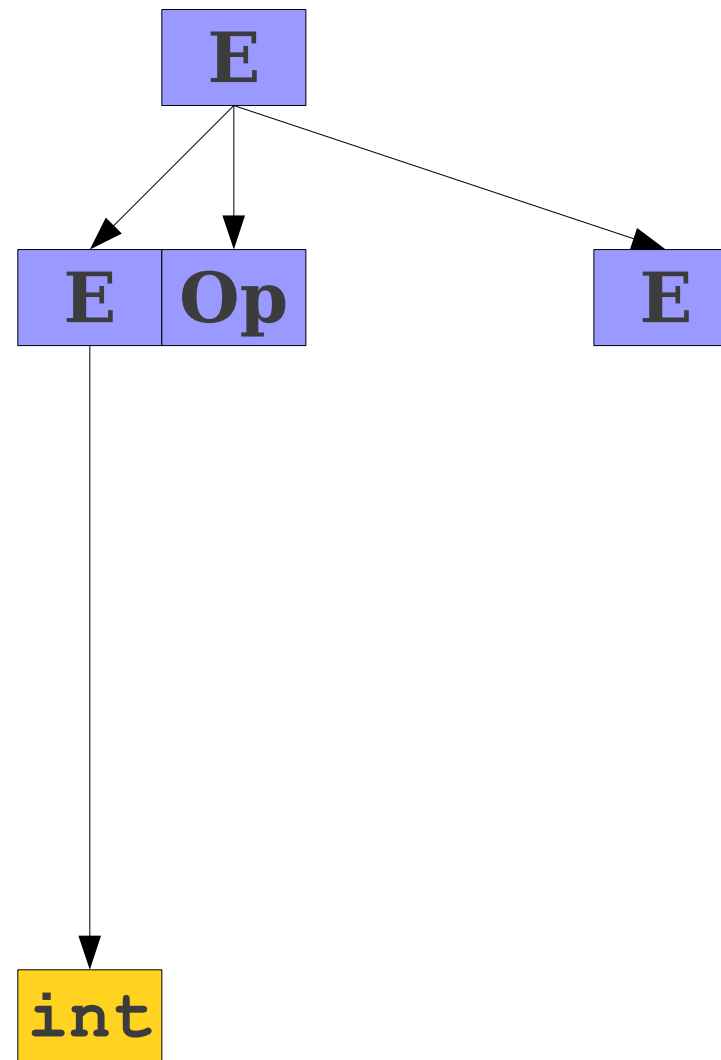


Parse Trees

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

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

E
 $\Rightarrow$ **E Op E**
 $\Rightarrow$ **int Op E**

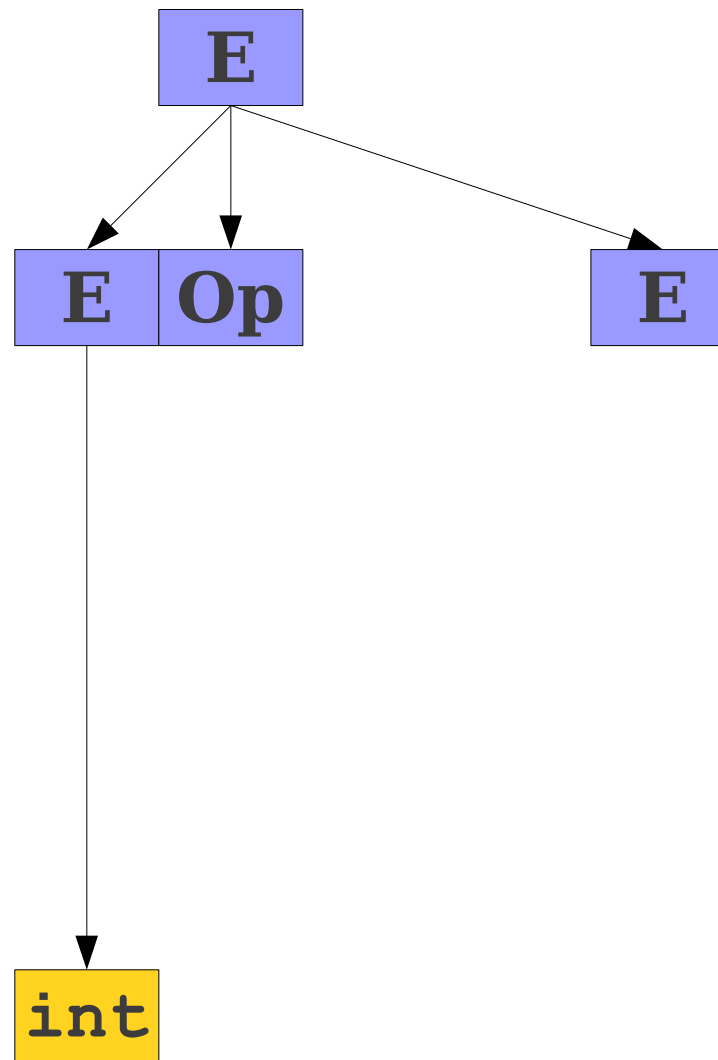


Parse Trees

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

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

E
 $\Rightarrow$ **E Op E**
 $\Rightarrow$ **int Op E**
 $\Rightarrow$ **int * E**

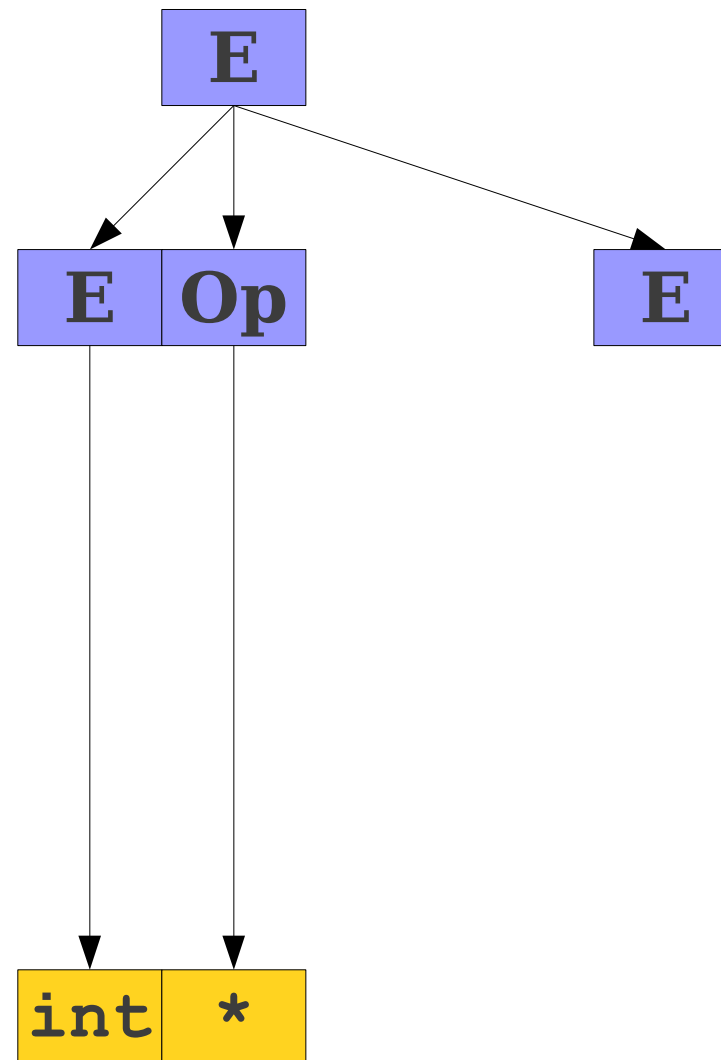


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow \text{int Op } E$
 $\Rightarrow \text{int } * E$

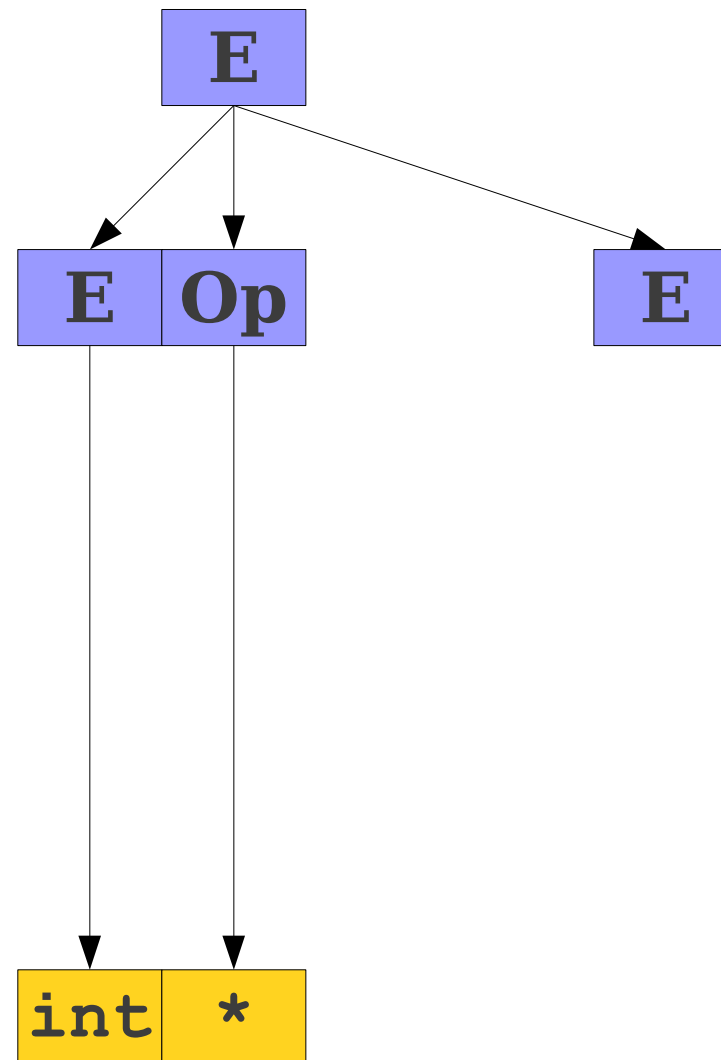


Parse Trees

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

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

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

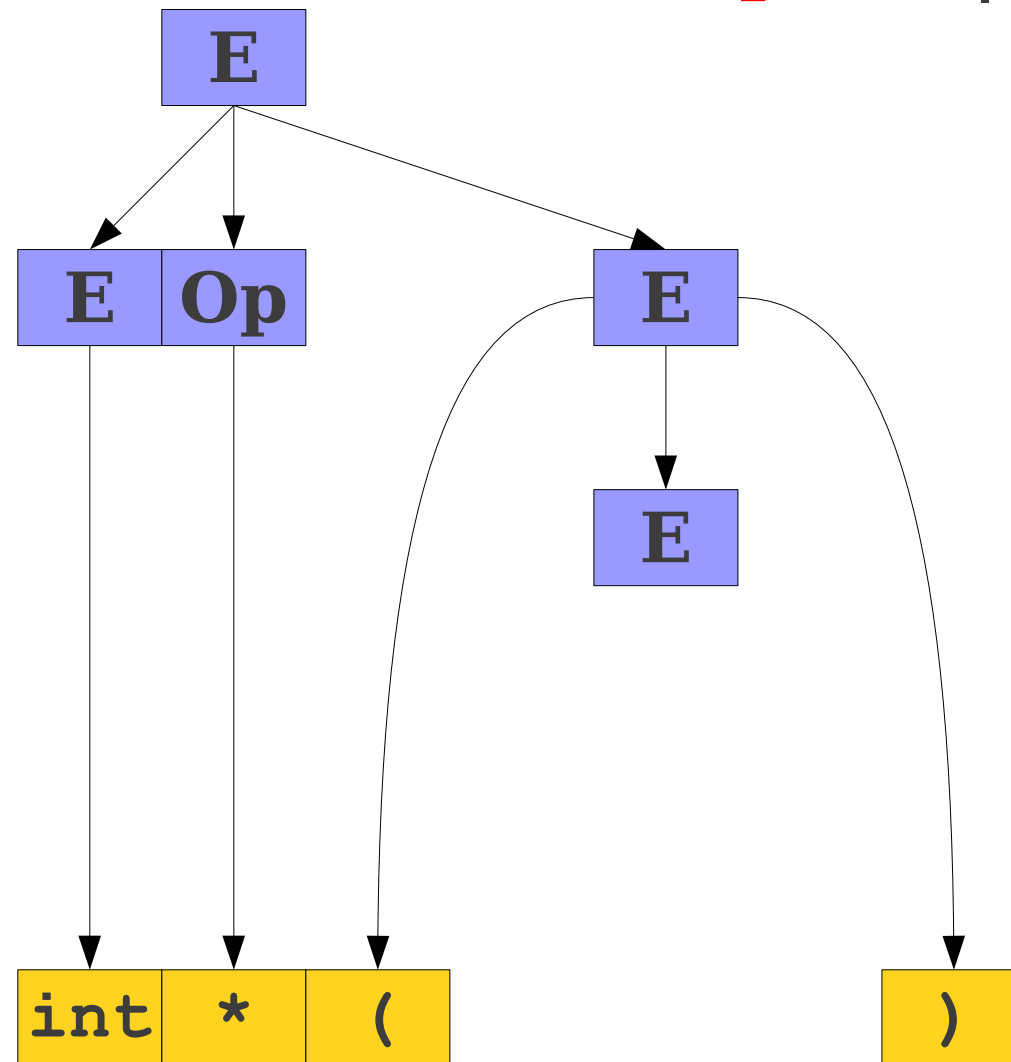


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

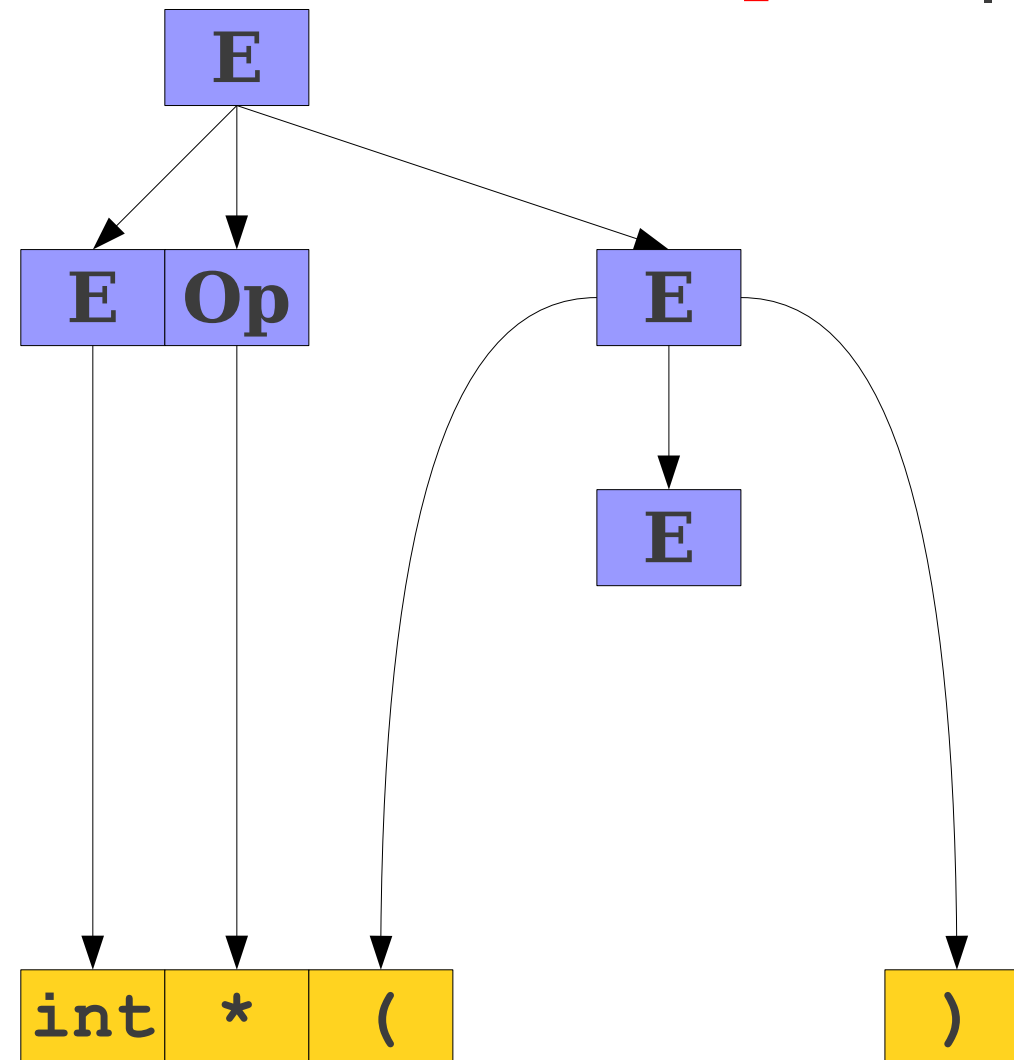
E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow \text{int Op } E$
 $\Rightarrow \text{int } * E$
 $\Rightarrow \text{int } * (E)$



Parse Trees

$$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$$

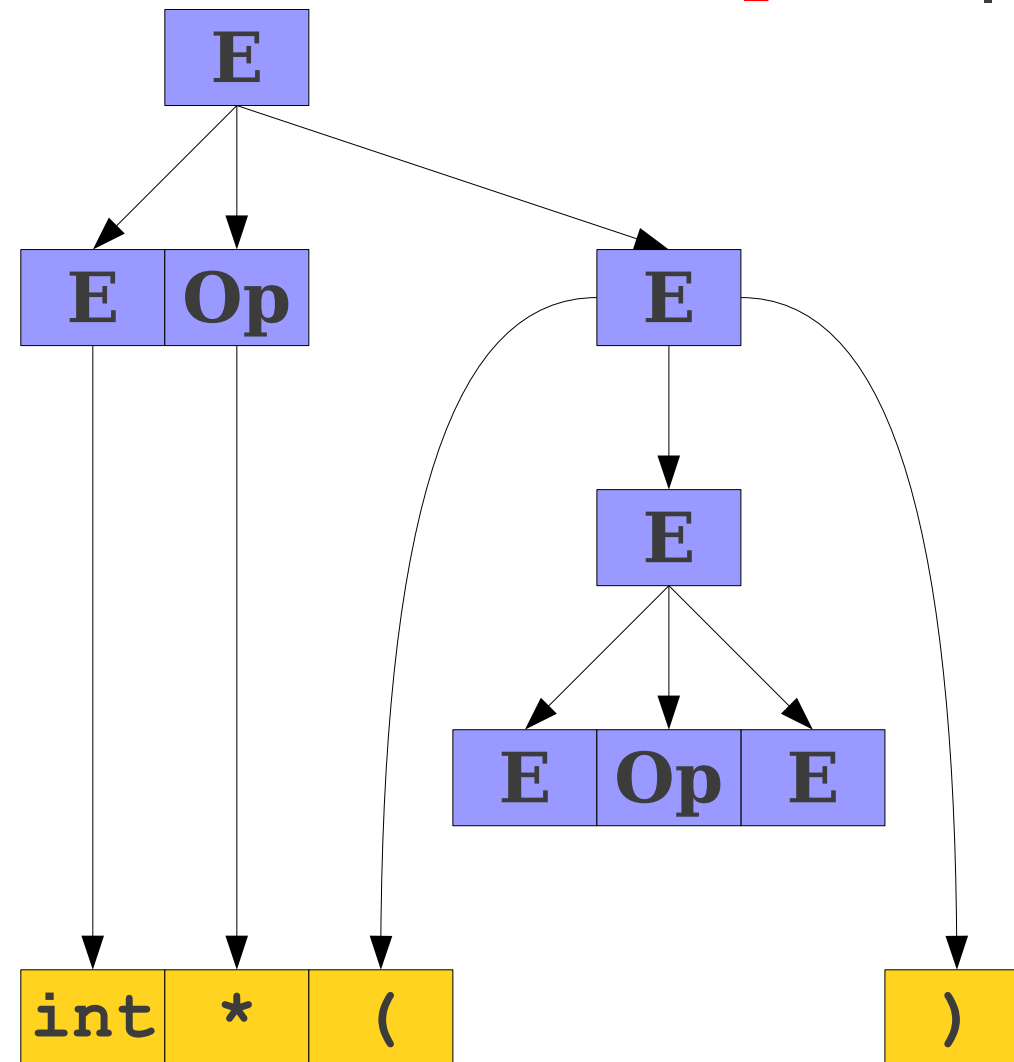
Op $\rightarrow$ + | - | * | /

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


Parse Trees

$$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$$

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

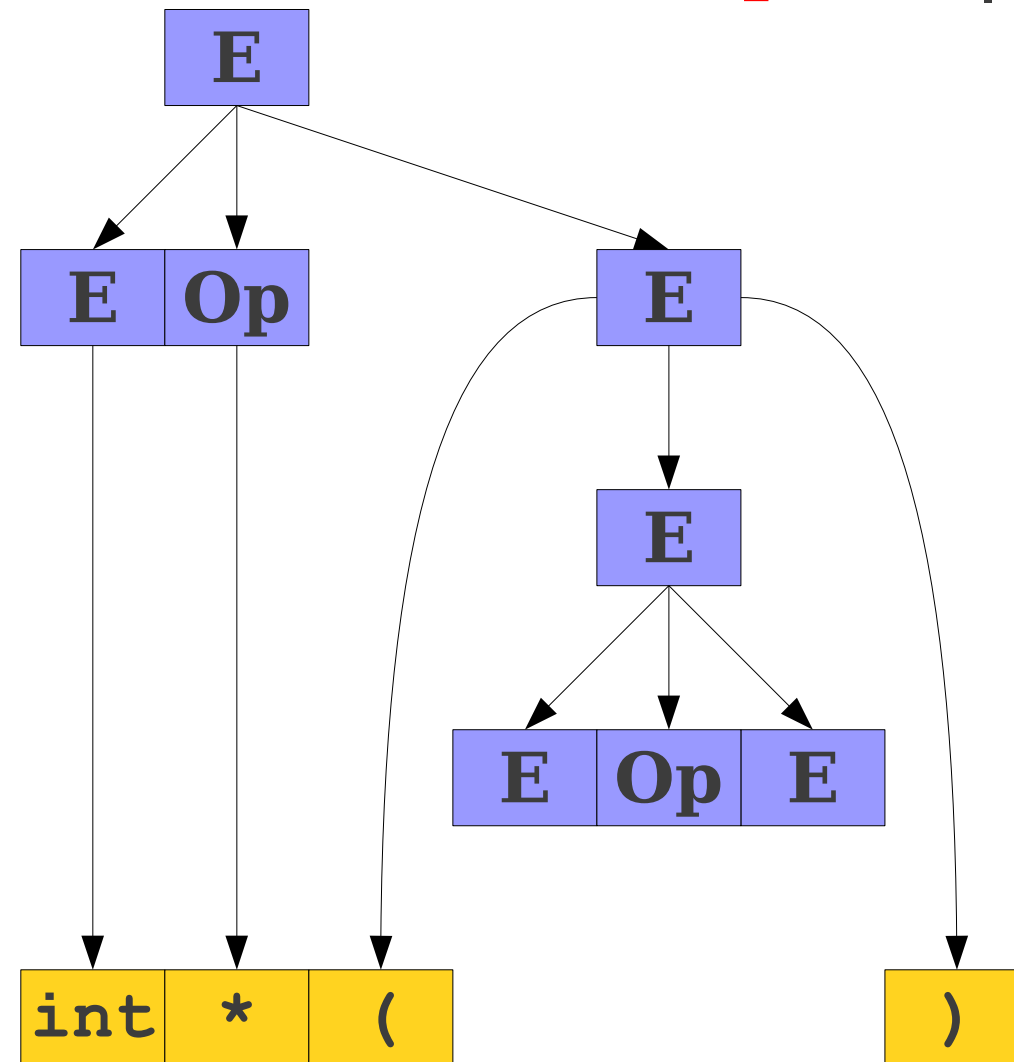
$$\begin{aligned} & \mathbf{E} \\ \Rightarrow & \mathbf{E \ Op \ E} \\ \Rightarrow & \mathbf{int \ Op \ E} \\ \Rightarrow & \mathbf{int \ * \ E} \\ \Rightarrow & \mathbf{int \ * \ (E)} \\ \Rightarrow & \mathbf{int \ * \ (E \ Op \ E)} \end{aligned}$$


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow \text{int Op } E$
 $\Rightarrow \text{int } * E$
 $\Rightarrow \text{int } * (E)$
 $\Rightarrow \text{int } * (E \text{ Op } E)$
 $\Rightarrow \text{int } * (\text{int Op } E)$



Parse Trees

$$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$$

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

$$\Rightarrow E$$

$$\Rightarrow E \text{ Op } E$$

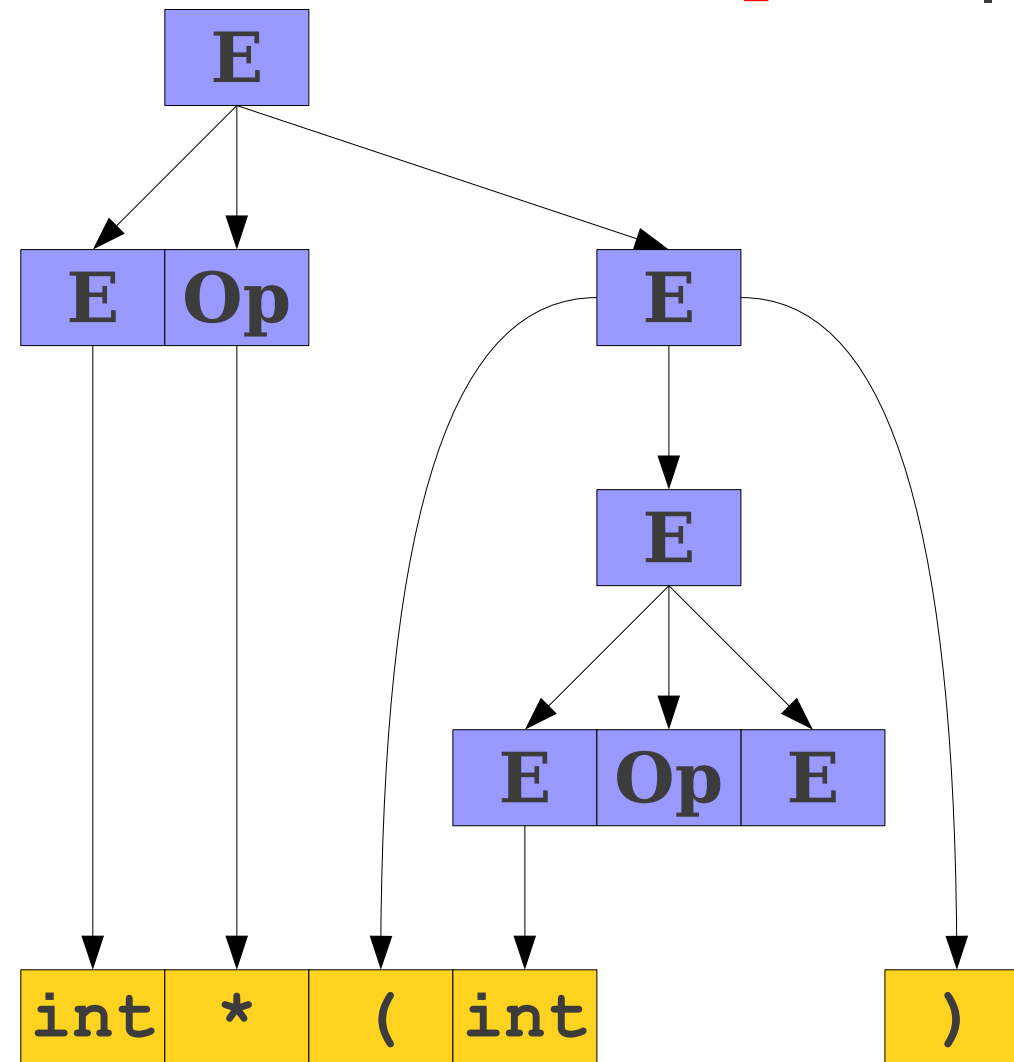
$$\Rightarrow \text{int Op } E$$

$$\Rightarrow \text{int} * E$$

$$\Rightarrow \text{int} * (E)$$

$$\Rightarrow \text{int} * (E \text{ Op } E)$$

$$\Rightarrow \text{int} * (\text{int Op } E)$$

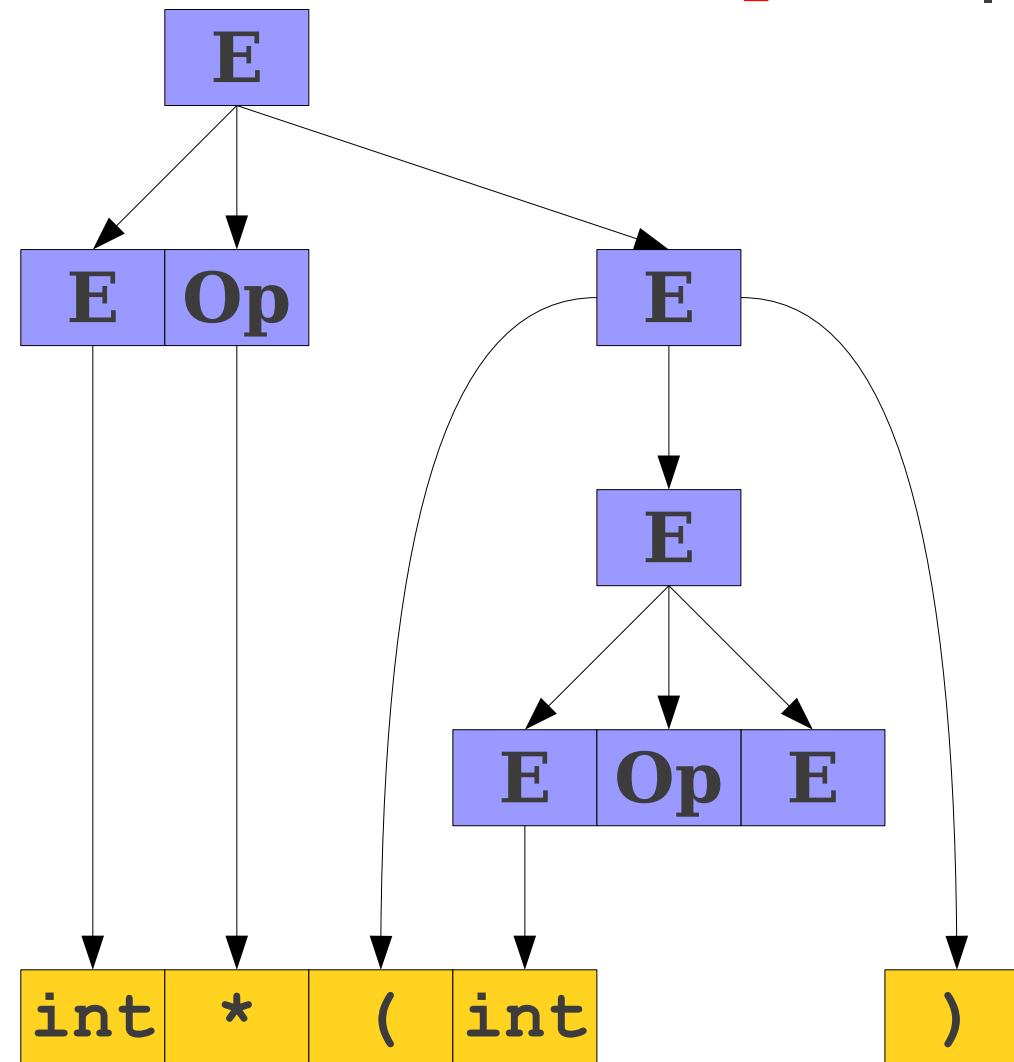


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow \text{int Op } E$
 $\Rightarrow \text{int } * E$
 $\Rightarrow \text{int } * (E)$
 $\Rightarrow \text{int } * (E \text{ Op } E)$
 $\Rightarrow \text{int } * (\text{int Op } E)$
 $\Rightarrow \text{int } * (\text{int } + E)$

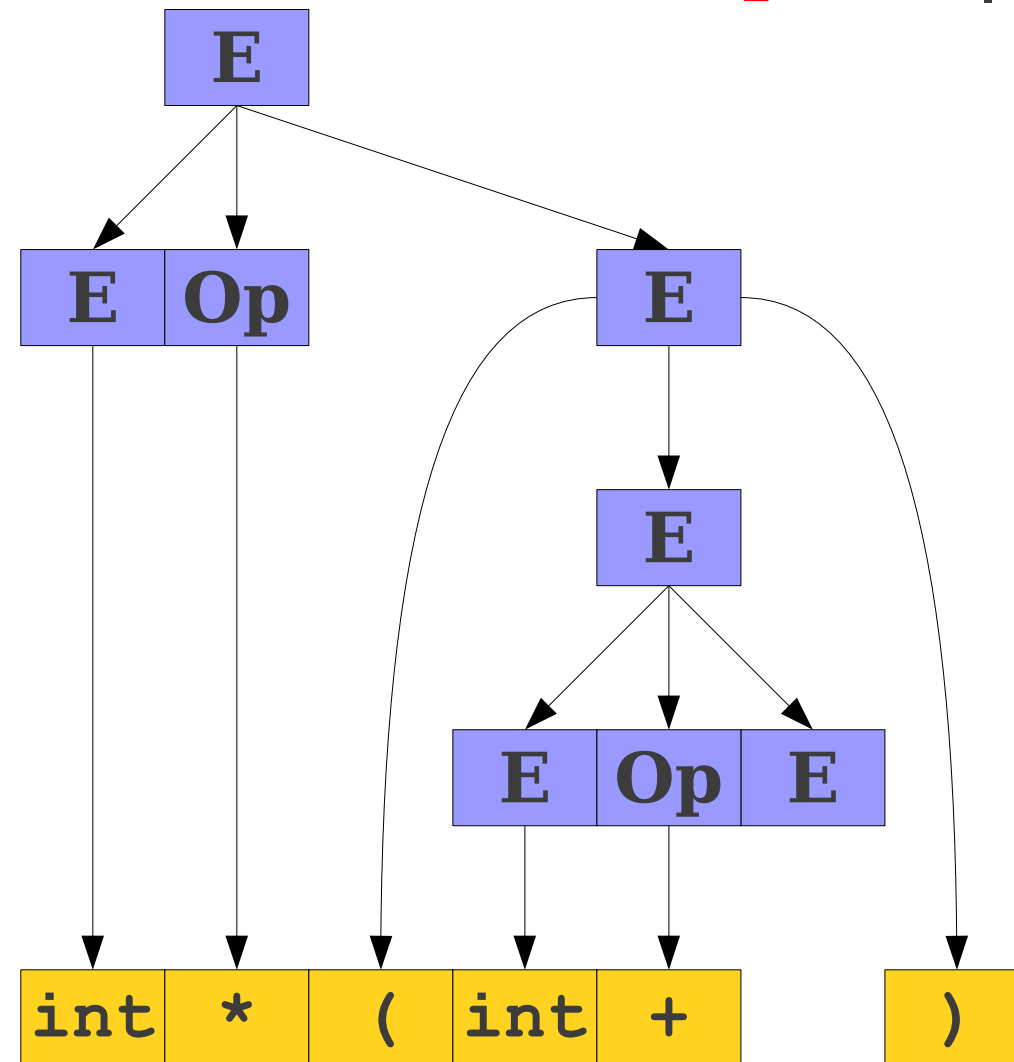


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow \text{int Op } E$
 $\Rightarrow \text{int } * E$
 $\Rightarrow \text{int } * (E)$
 $\Rightarrow \text{int } * (E \text{ Op } E)$
 $\Rightarrow \text{int } * (\text{int Op } E)$
 $\Rightarrow \text{int } * (\text{int } + E)$

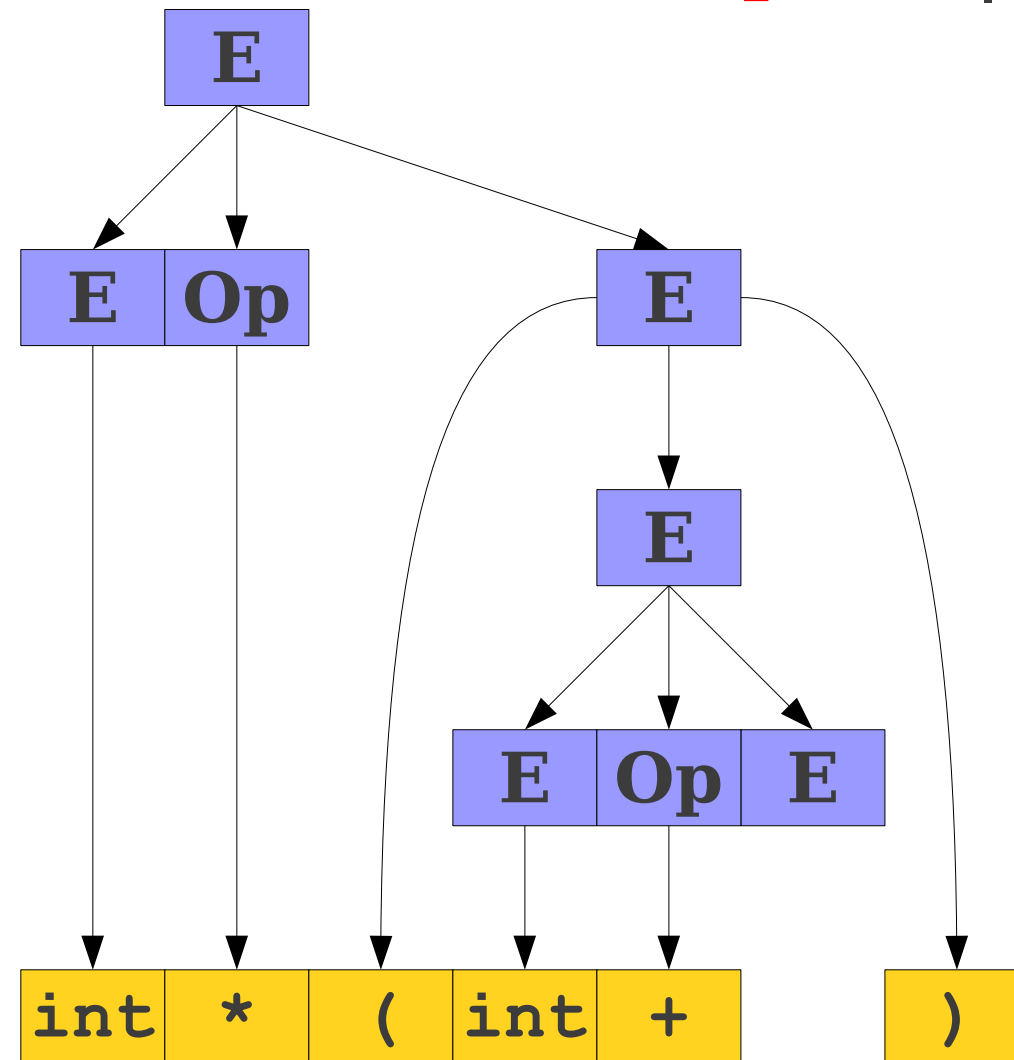


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow \text{int Op } E$
 $\Rightarrow \text{int } * E$
 $\Rightarrow \text{int } * (E)$
 $\Rightarrow \text{int } * (E \text{ Op } E)$
 $\Rightarrow \text{int } * (\text{int Op } E)$
 $\Rightarrow \text{int } * (\text{int } + E)$
 $\Rightarrow \text{int } * (\text{int } + \text{int})$

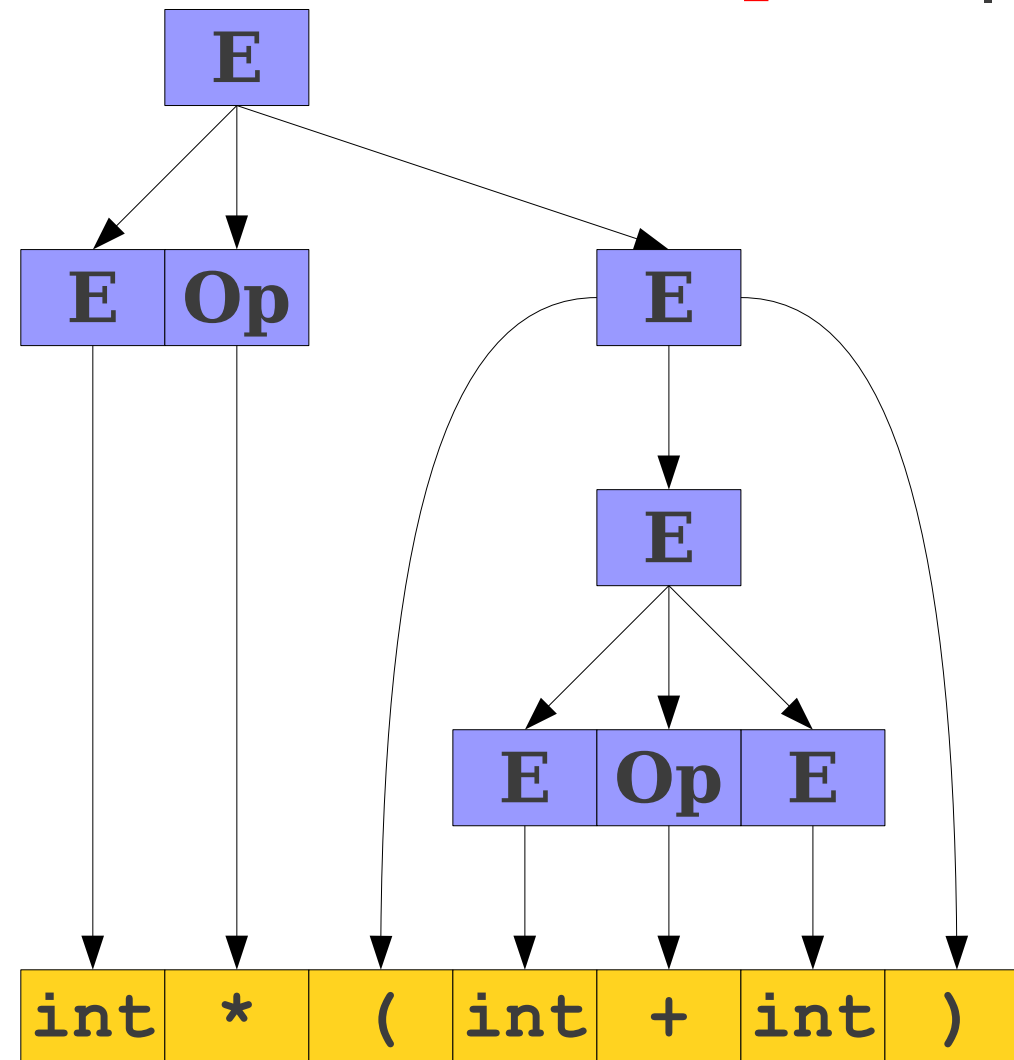


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow \text{int Op } E$
 $\Rightarrow \text{int } * E$
 $\Rightarrow \text{int } * (E)$
 $\Rightarrow \text{int } * (E \text{ Op } E)$
 $\Rightarrow \text{int } * (\text{int Op } E)$
 $\Rightarrow \text{int } * (\text{int } + E)$
 $\Rightarrow \text{int } * (\text{int } + \text{int})$

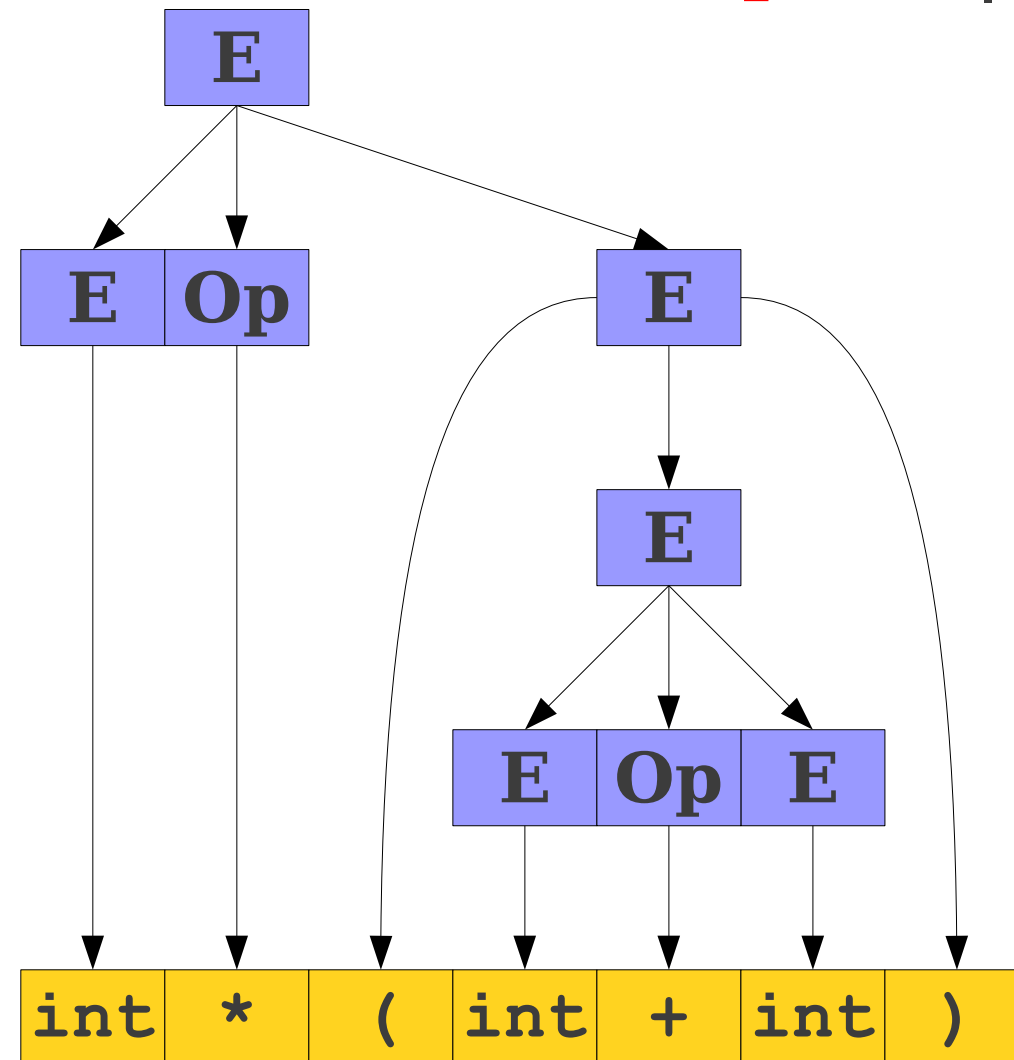


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow \text{int Op } E$
 $\Rightarrow \text{int } * E$
 $\Rightarrow \text{int } * (E)$
 $\Rightarrow \text{int } * (E \text{ Op } E)$
 $\Rightarrow \text{int } * (\text{int Op } E)$
 $\Rightarrow \text{int } * (\text{int } + E)$
 $\Rightarrow \text{int } * (\text{int } + \text{int})$



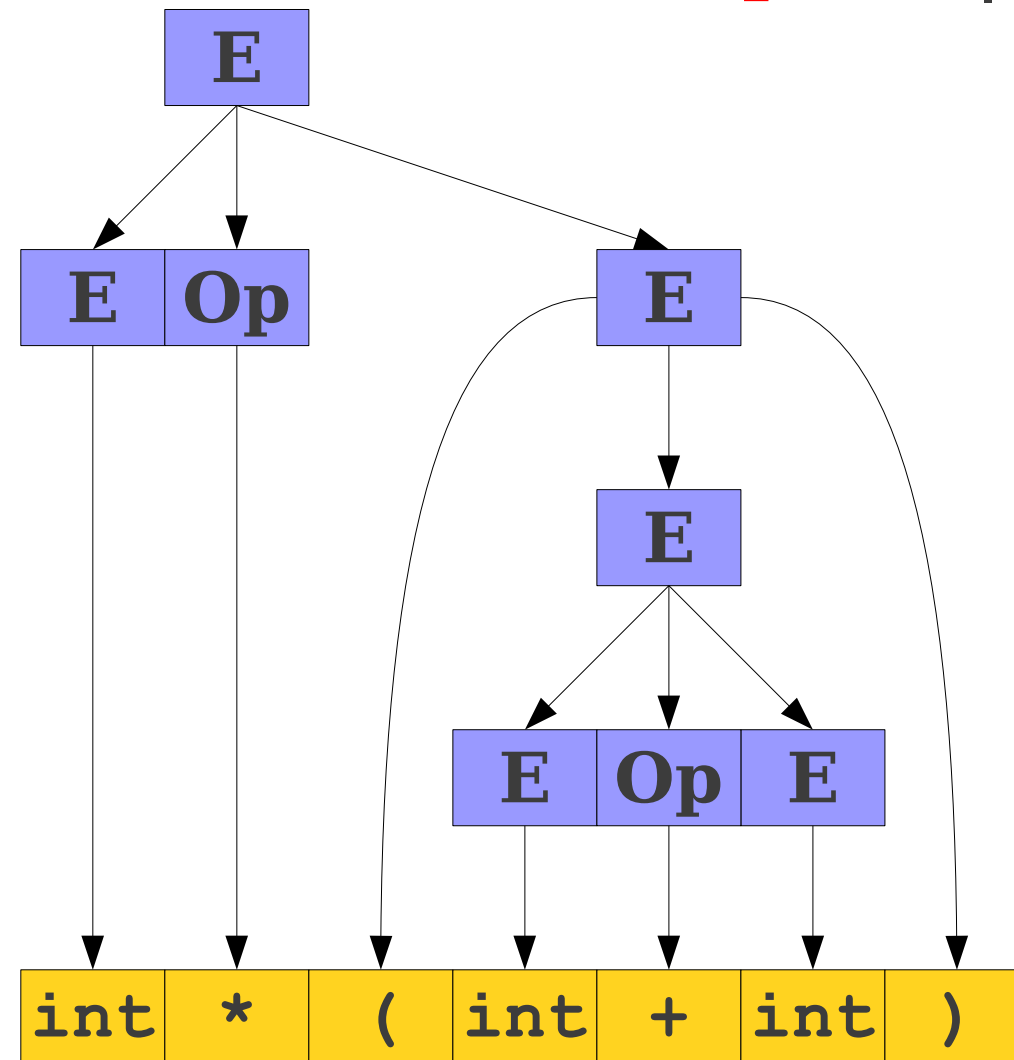
Start symbol is the root

Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow \text{int Op } E$
 $\Rightarrow \text{int } * E$
 $\Rightarrow \text{int } * (E)$
 $\Rightarrow \text{int } * (E \text{ Op } E)$
 $\Rightarrow \text{int } * (\text{int Op } E)$
 $\Rightarrow \text{int } * (\text{int } + E)$
 $\Rightarrow \text{int } * (\text{int } + \text{int})$



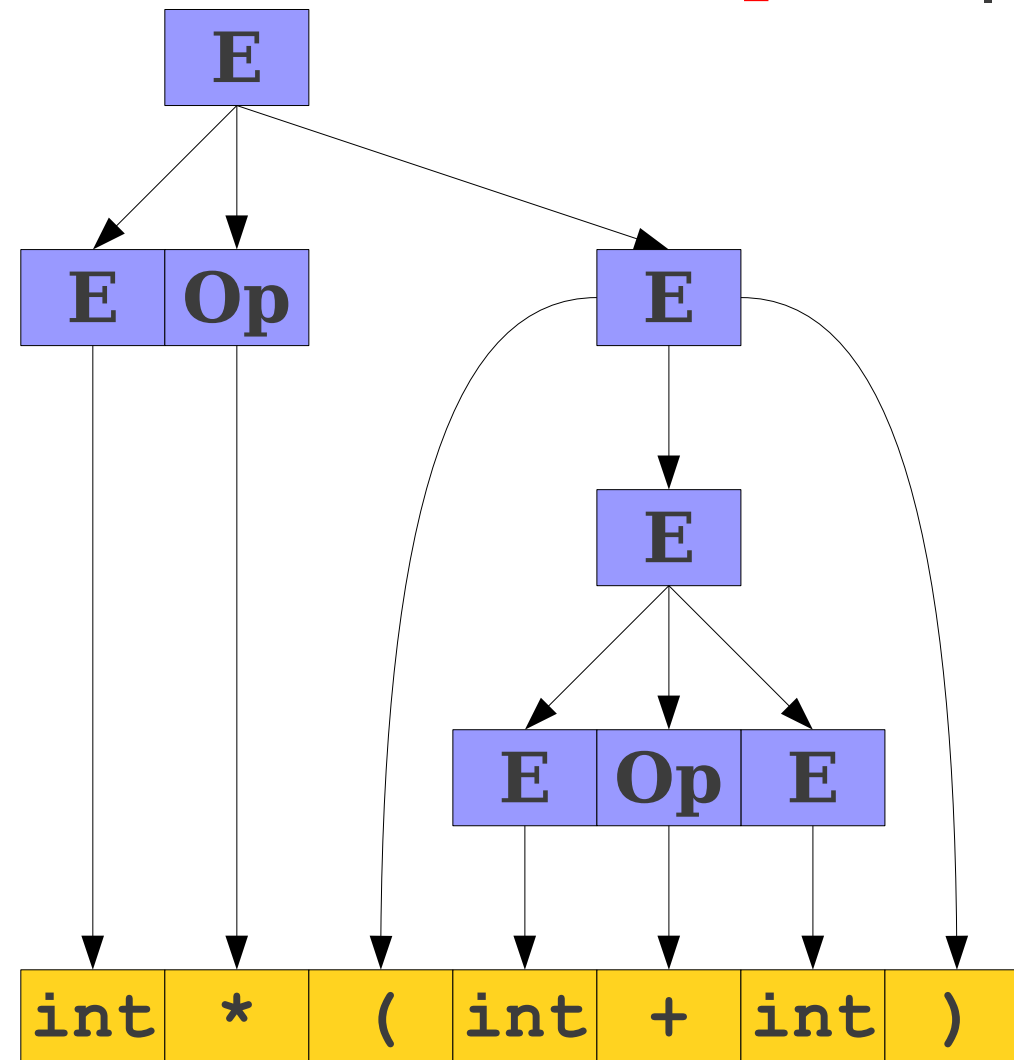
Start symbol is the root
Non-leaf nodes are non-terminals

Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow \text{int Op } E$
 $\Rightarrow \text{int } * E$
 $\Rightarrow \text{int } * (E)$
 $\Rightarrow \text{int } * (E \text{ Op } E)$
 $\Rightarrow \text{int } * (\text{int Op } E)$
 $\Rightarrow \text{int } * (\text{int } + E)$
 $\Rightarrow \text{int } * (\text{int } + \text{int})$



Start symbol is the root

Non-leaf nodes are non-terminals

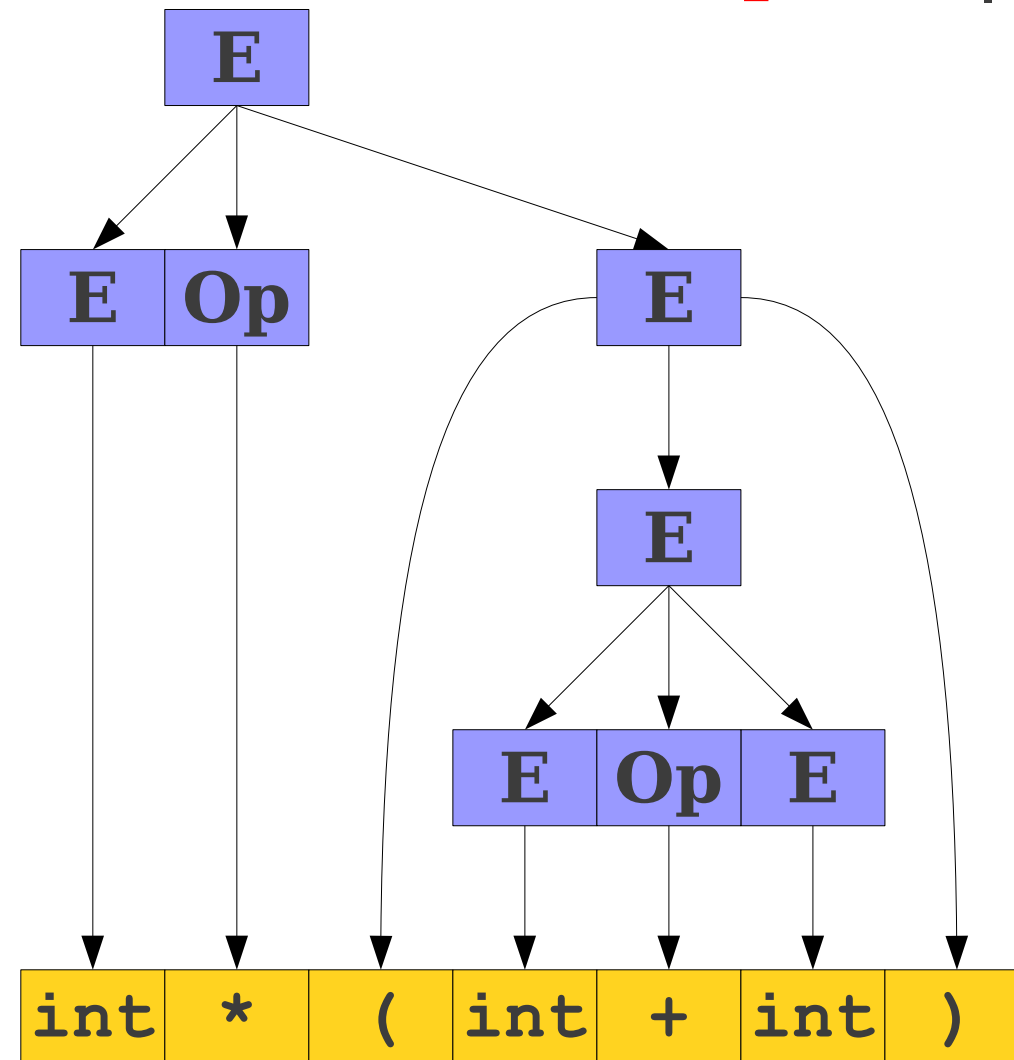
Leaf nodes are terminals

Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow \text{int Op } E$
 $\Rightarrow \text{int } * E$
 $\Rightarrow \text{int } * (E)$
 $\Rightarrow \text{int } * (E \text{ Op } E)$
 $\Rightarrow \text{int } * (\text{int Op } E)$
 $\Rightarrow \text{int } * (\text{int } + E)$
 $\Rightarrow \text{int } * (\text{int } + \text{int})$



Start symbol is the root

Non-leaf nodes are non-terminals

Leaf nodes are terminals

Inorder walk of the leaves is the generated string

Parse Trees

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

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

E

Parse Trees

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

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

E

E

Parse Trees

E

E
 $\Rightarrow$ **E Op E**

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

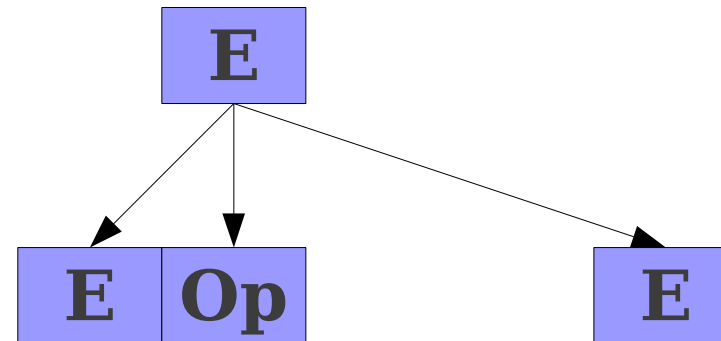
Op $\rightarrow$ **+** | **-** | ***** | **/**

Parse Trees

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

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

E
 $\Rightarrow$ **E Op E**

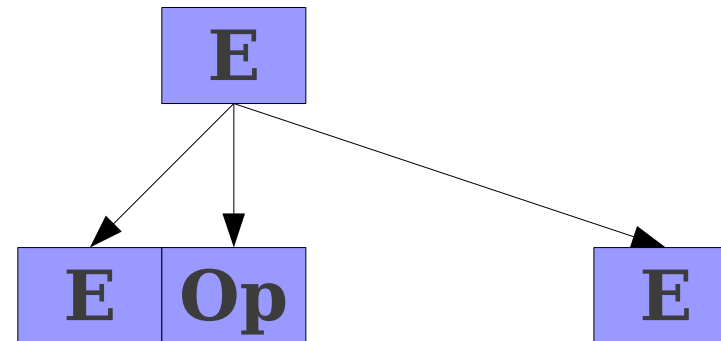


Parse Trees

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

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

E
 $\Rightarrow$ **E Op E**
 $\Rightarrow$ **E Op (E)**

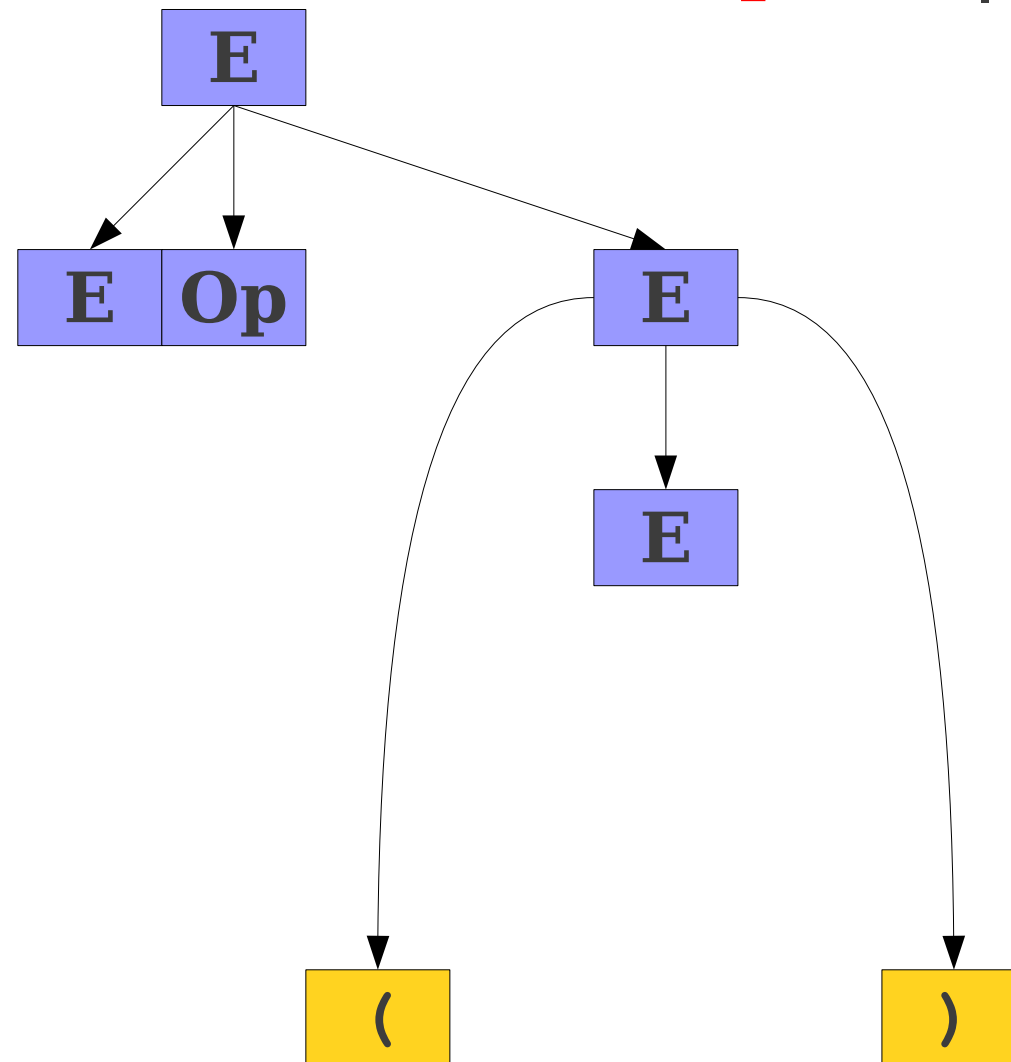


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$

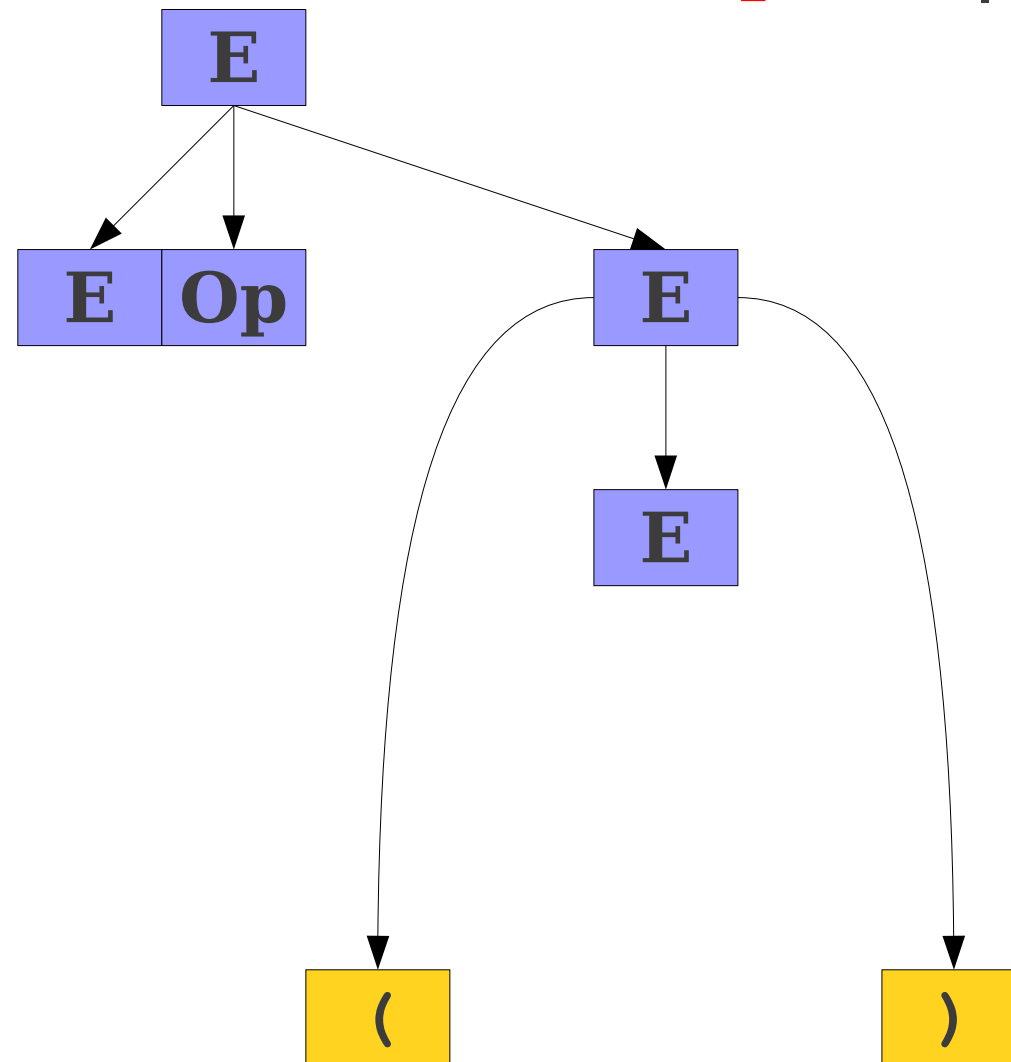


Parse Trees

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

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

E
 $\Rightarrow$ **E Op E**
 $\Rightarrow$ **E Op (E)**
 $\Rightarrow$ **E Op (E Op E)**

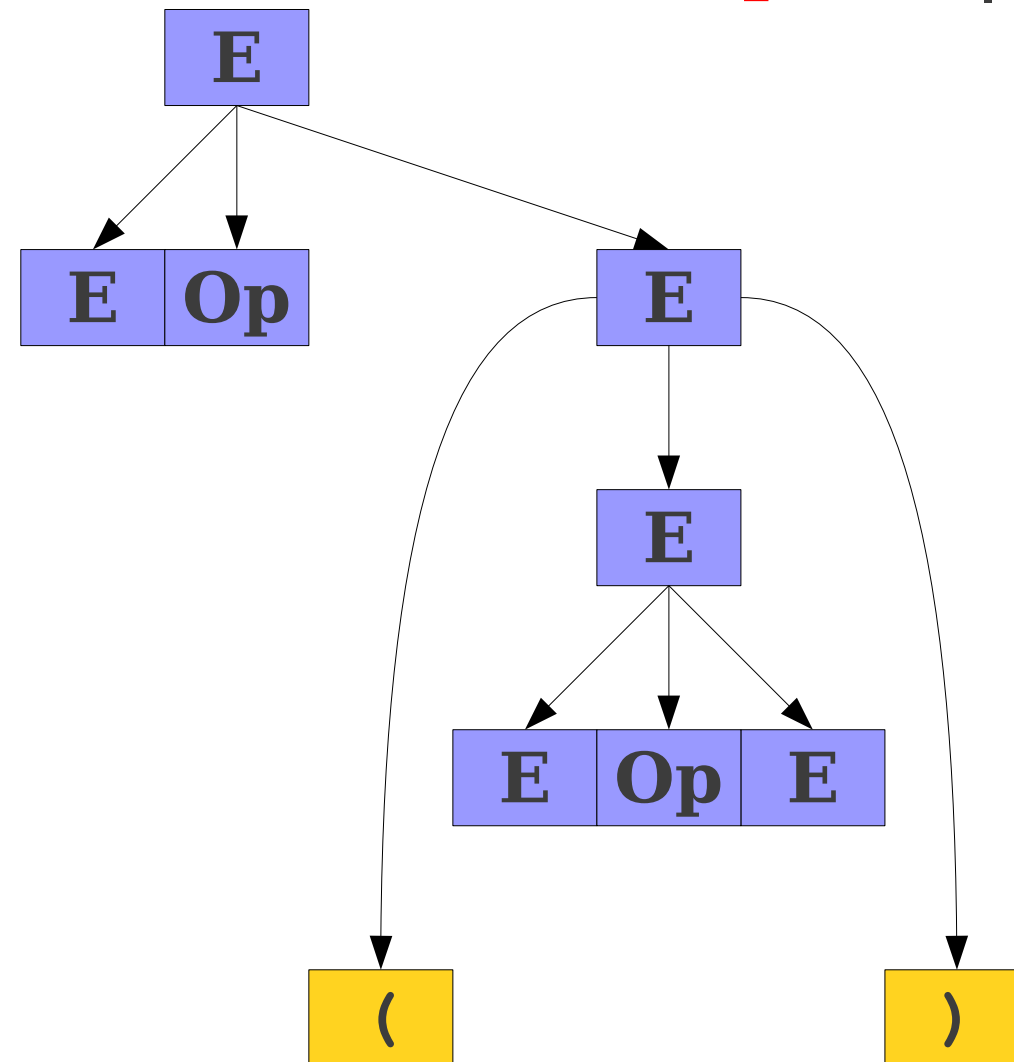


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$

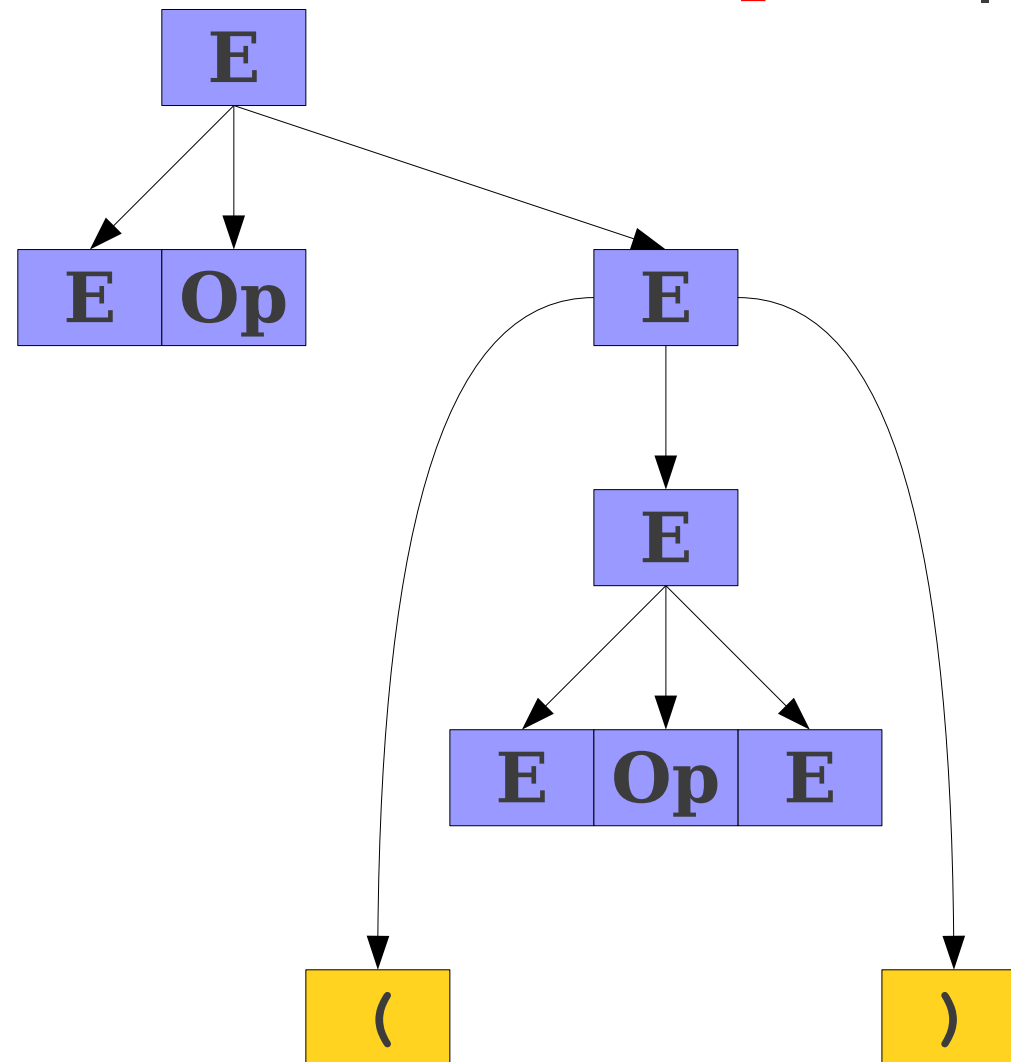


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } \text{int})$

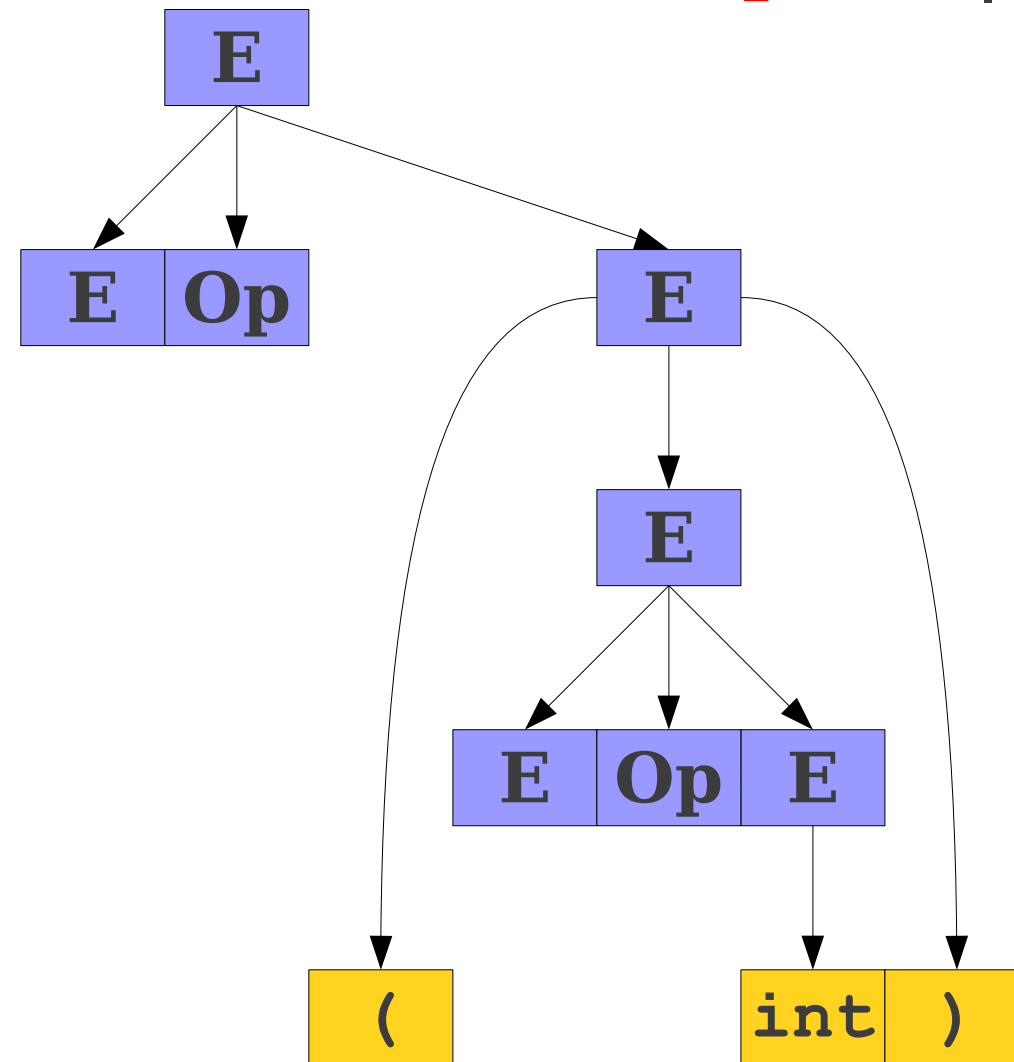


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } \text{int})$

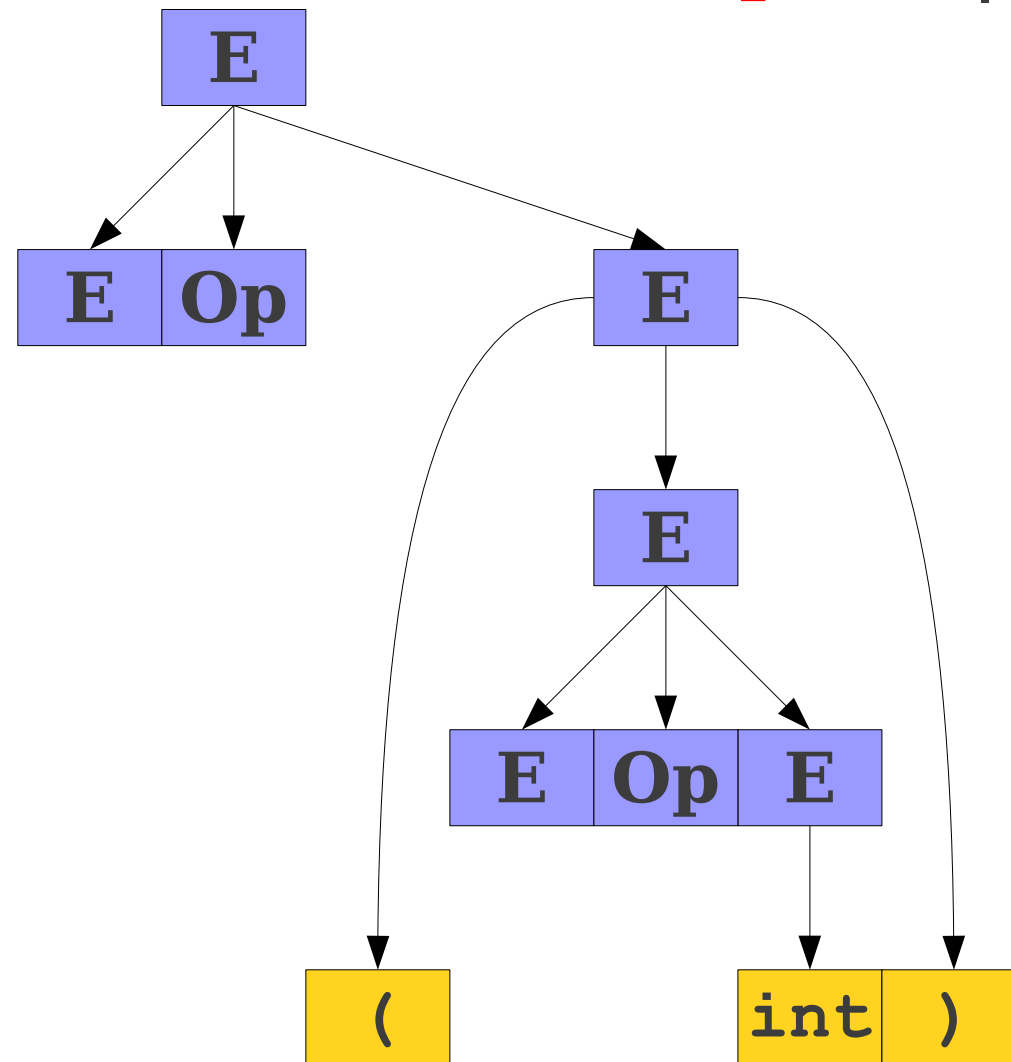


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } \text{int})$
 $\Rightarrow E \text{ Op } (E + \text{int})$

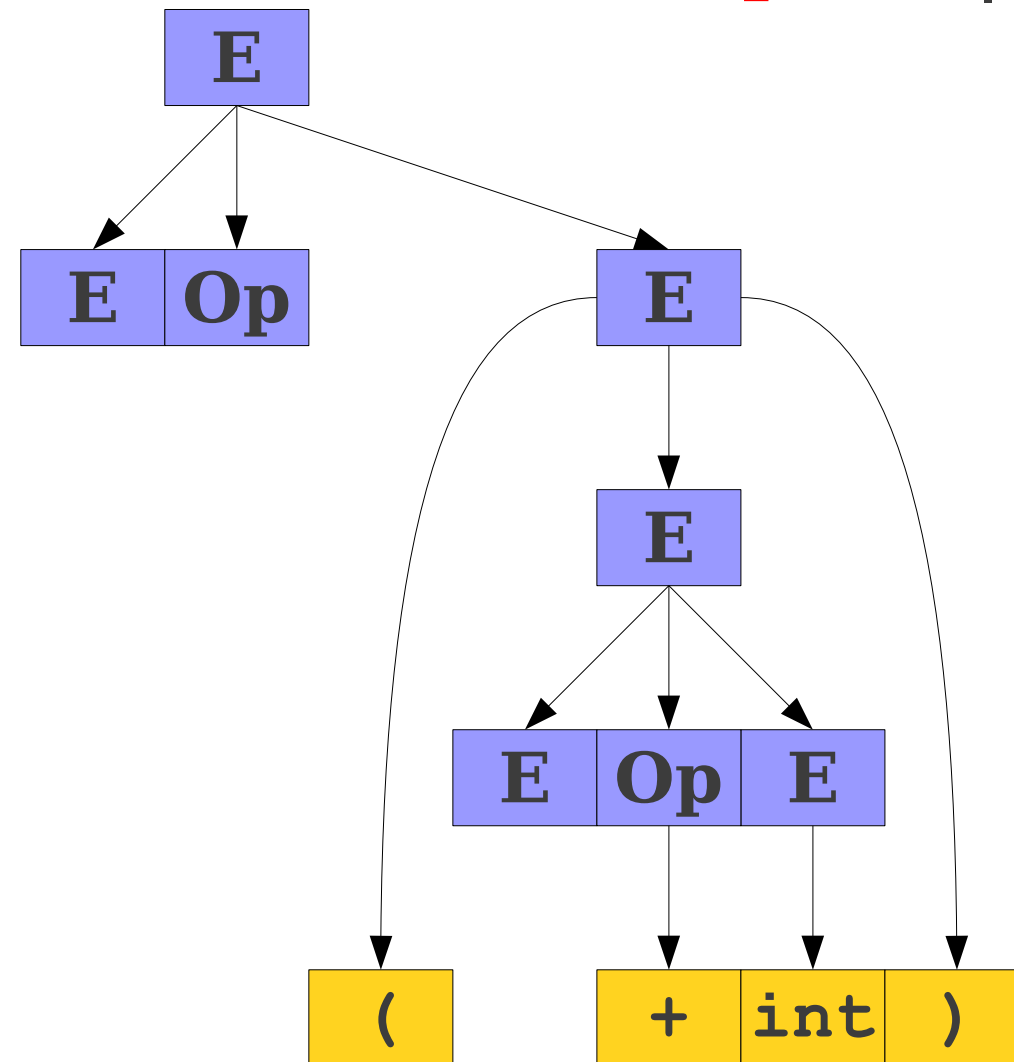


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } \text{int})$
 $\Rightarrow E \text{ Op } (E + \text{int})$

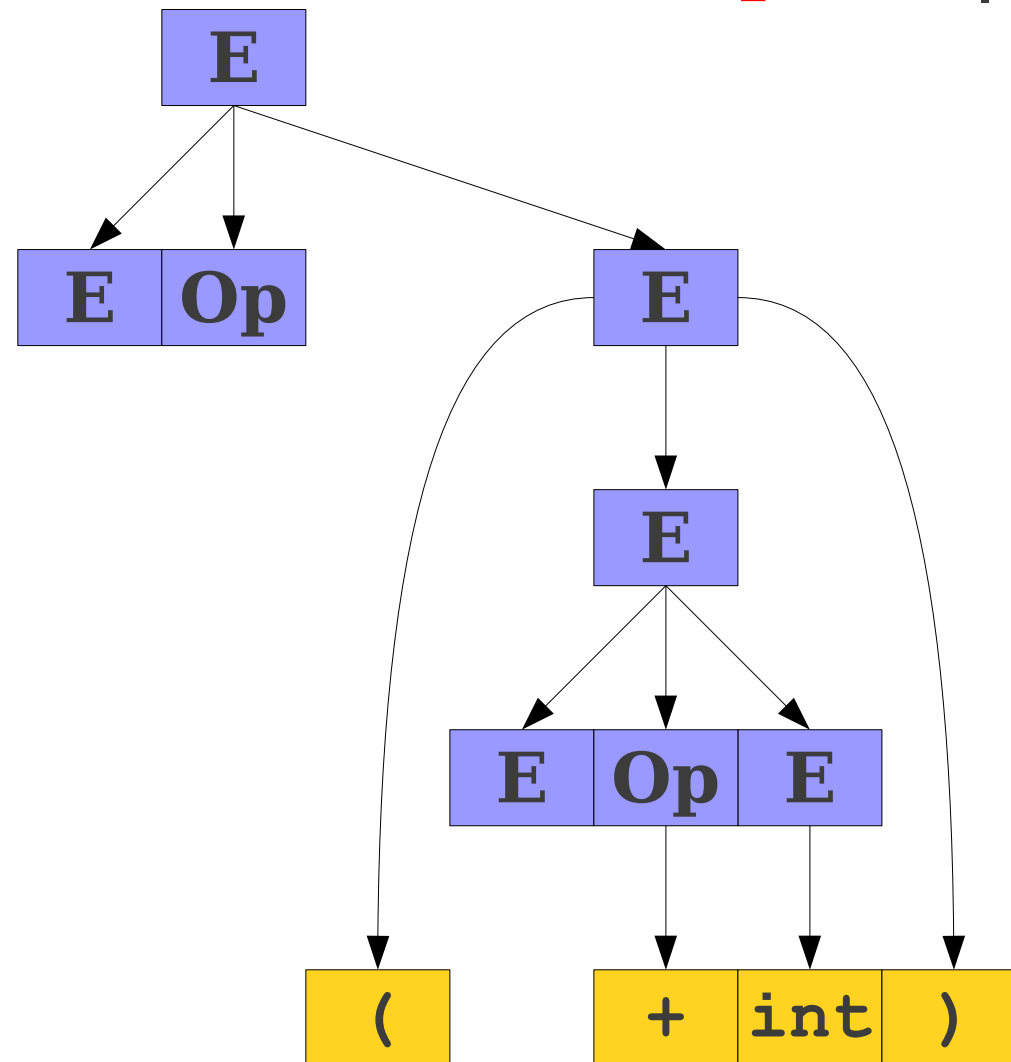


Parse Trees

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

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

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)**

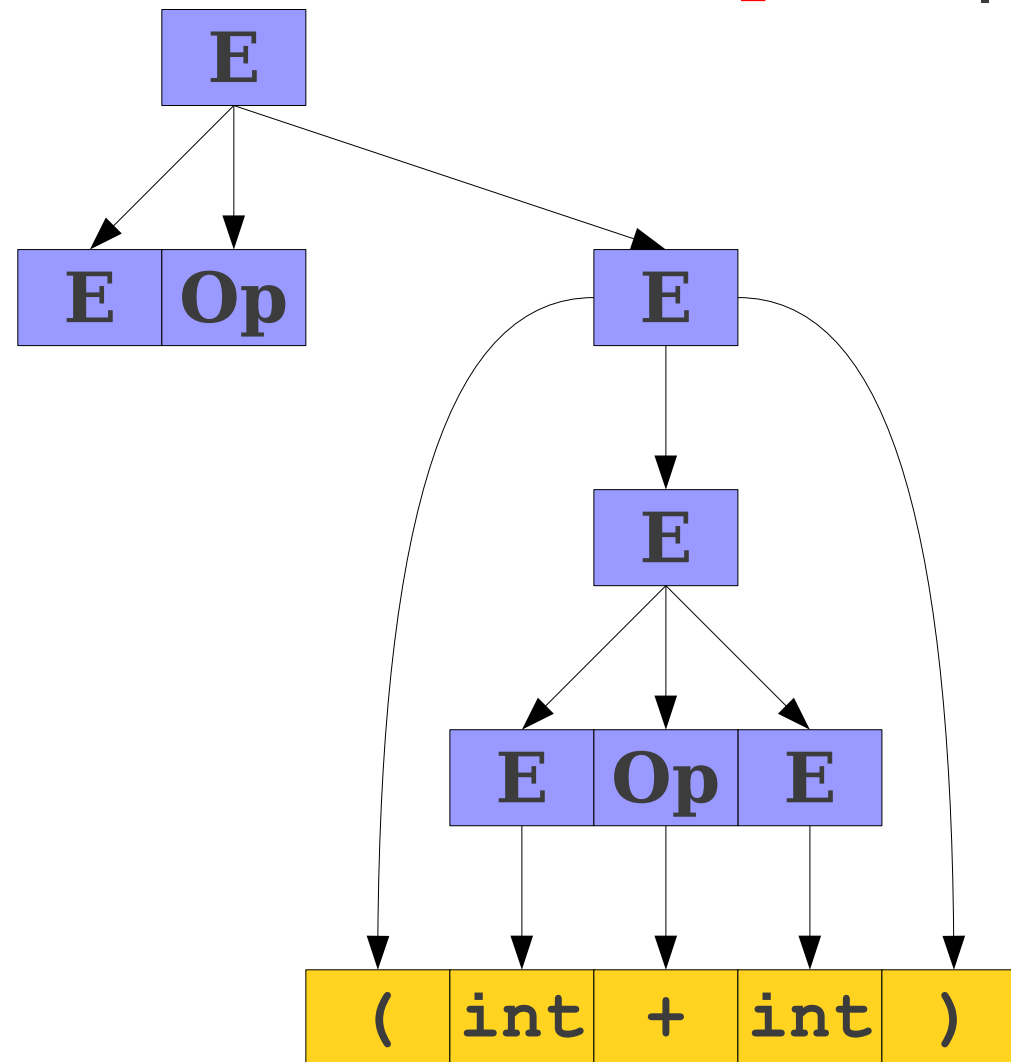


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } \text{int})$
 $\Rightarrow E \text{ Op } (E + \text{int})$
 $\Rightarrow E \text{ Op } (\text{int} + \text{int})$

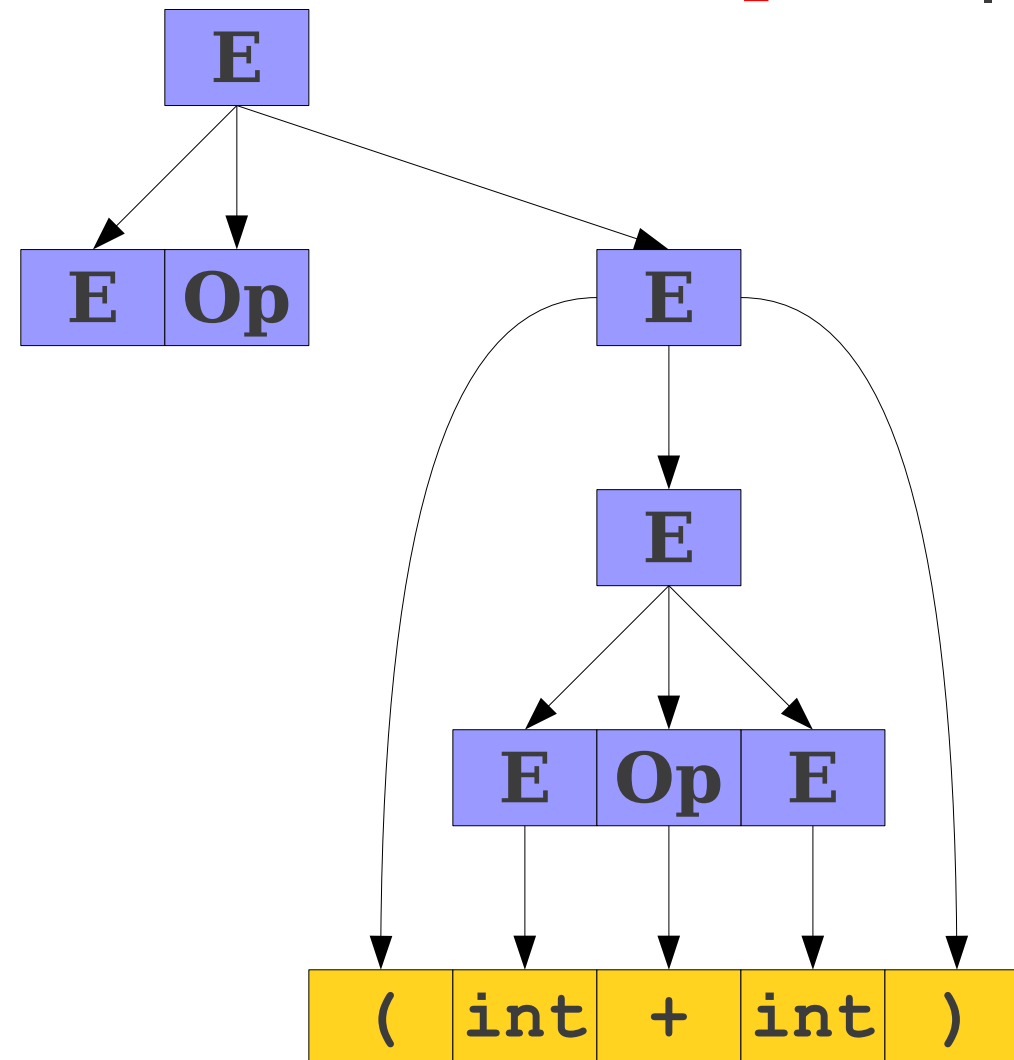


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

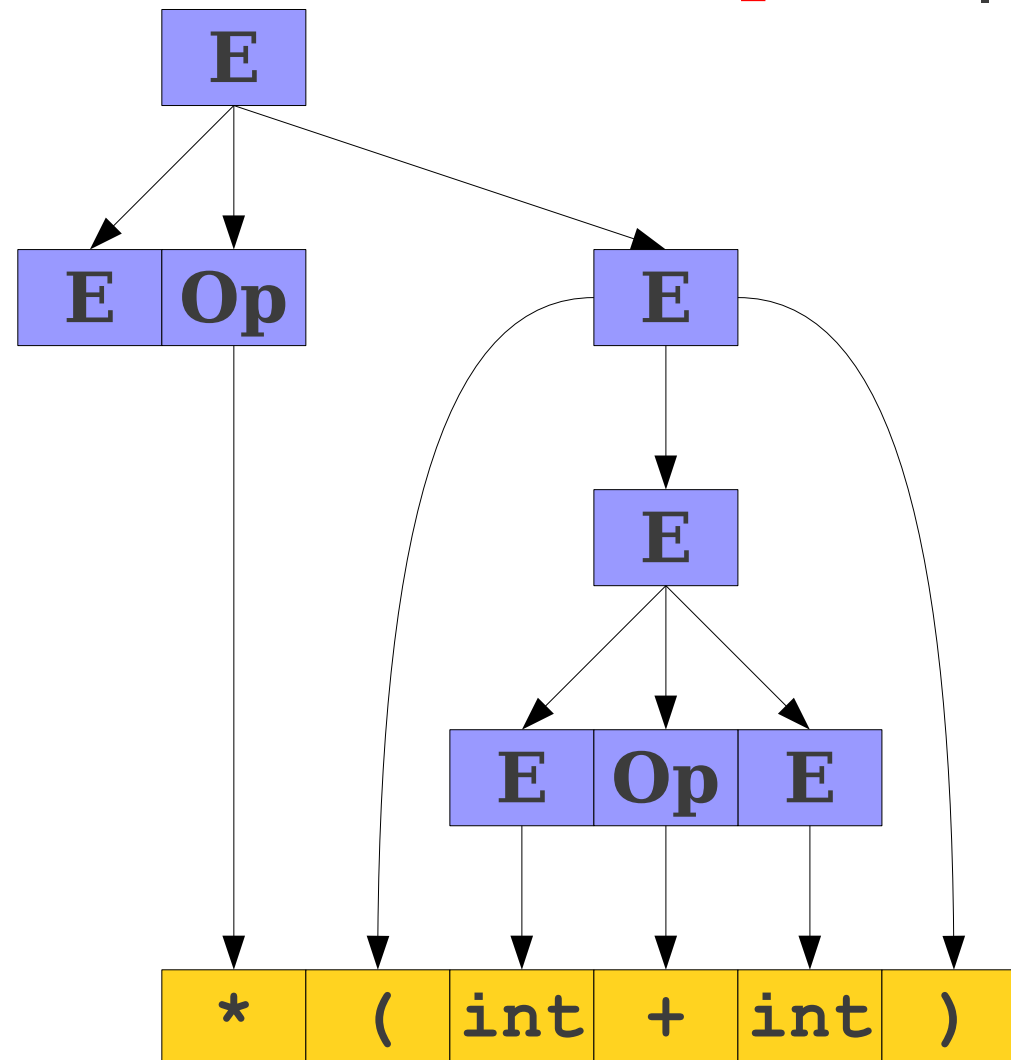
E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } \text{int})$
 $\Rightarrow E \text{ Op } (E + \text{int})$
 $\Rightarrow E \text{ Op } (\text{int} + \text{int})$
 $\Rightarrow E * (\text{int} + \text{int})$



Parse Trees

$$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$$

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

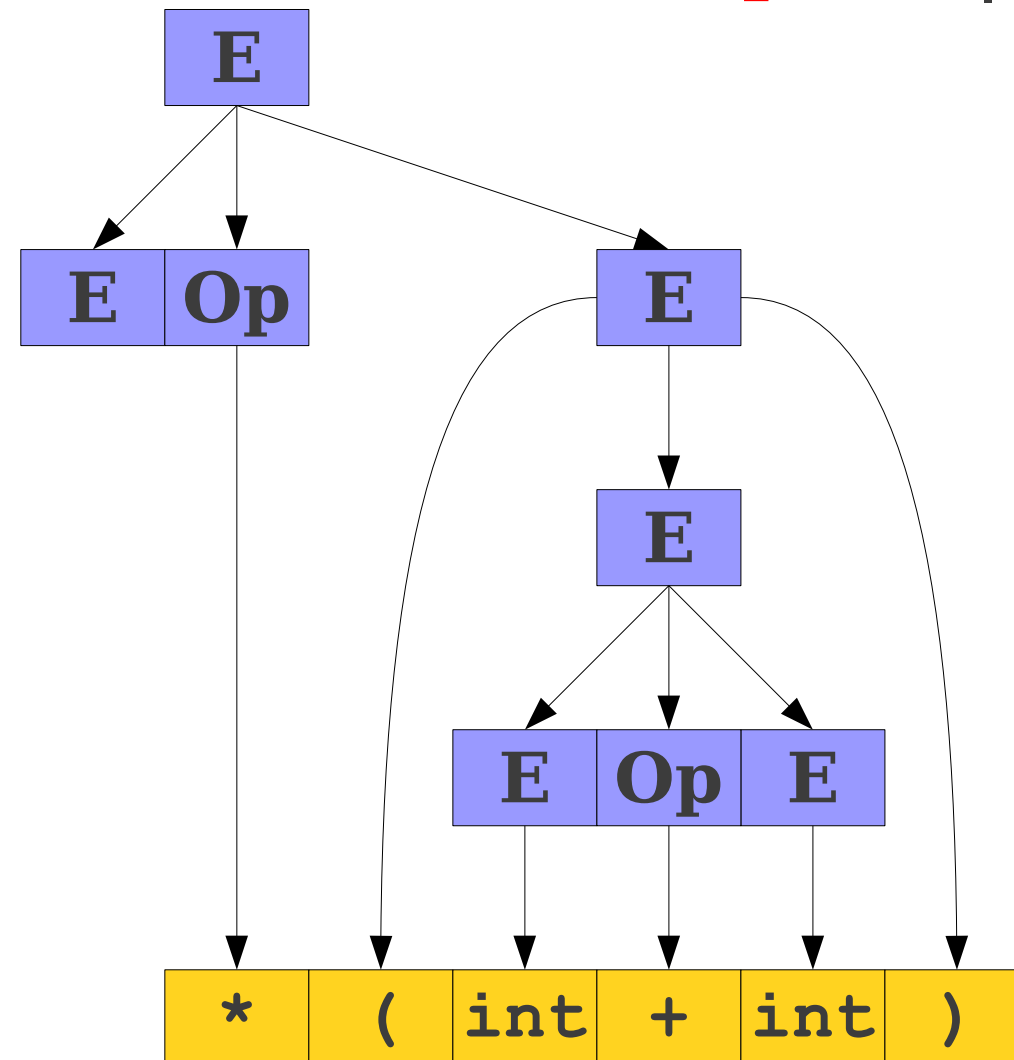
$$\begin{aligned} & \mathbf{E} \\ \Rightarrow & \mathbf{E} \text{ Op } \mathbf{E} \\ \Rightarrow & \mathbf{E} \text{ Op } (\mathbf{E}) \\ \Rightarrow & \mathbf{E} \text{ Op } (\mathbf{E} \text{ Op } \mathbf{E}) \\ \Rightarrow & \mathbf{E} \text{ Op } (\mathbf{E} \text{ Op } \text{int}) \\ \Rightarrow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


Parse Trees

$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } \text{int})$
 $\Rightarrow E \text{ Op } (E + \text{int})$
 $\Rightarrow E * (\text{int} + \text{int})$
 $\Rightarrow \text{int} * (\text{int} + \text{int})$

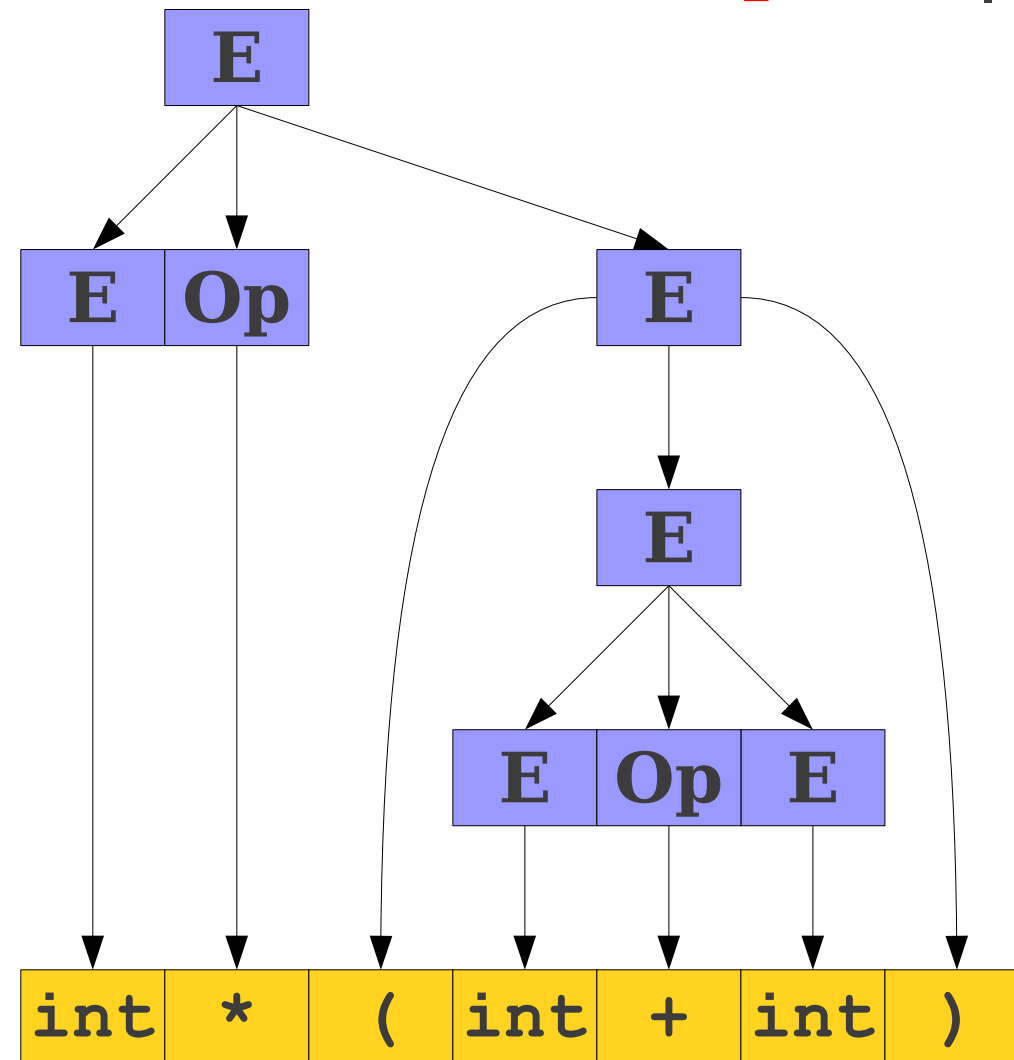


Parse Trees

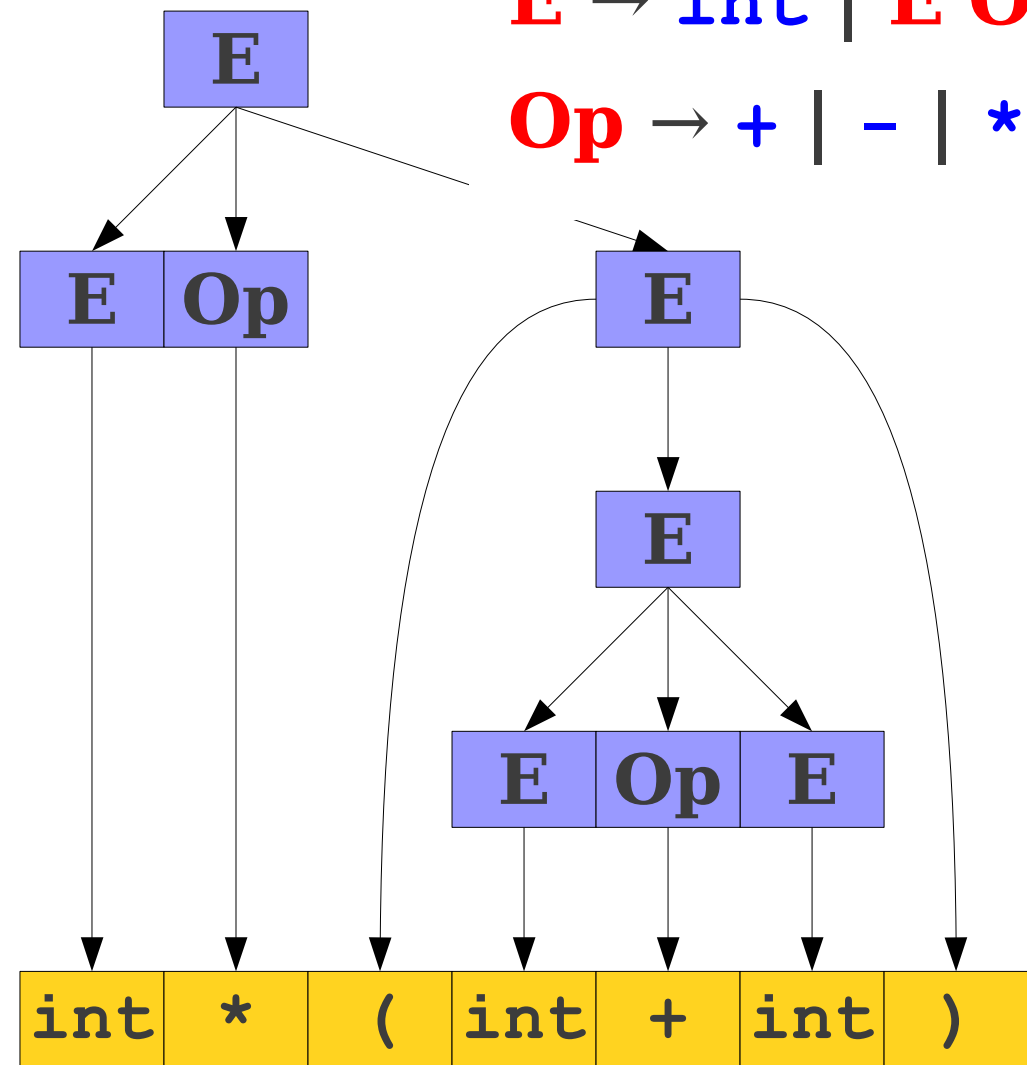
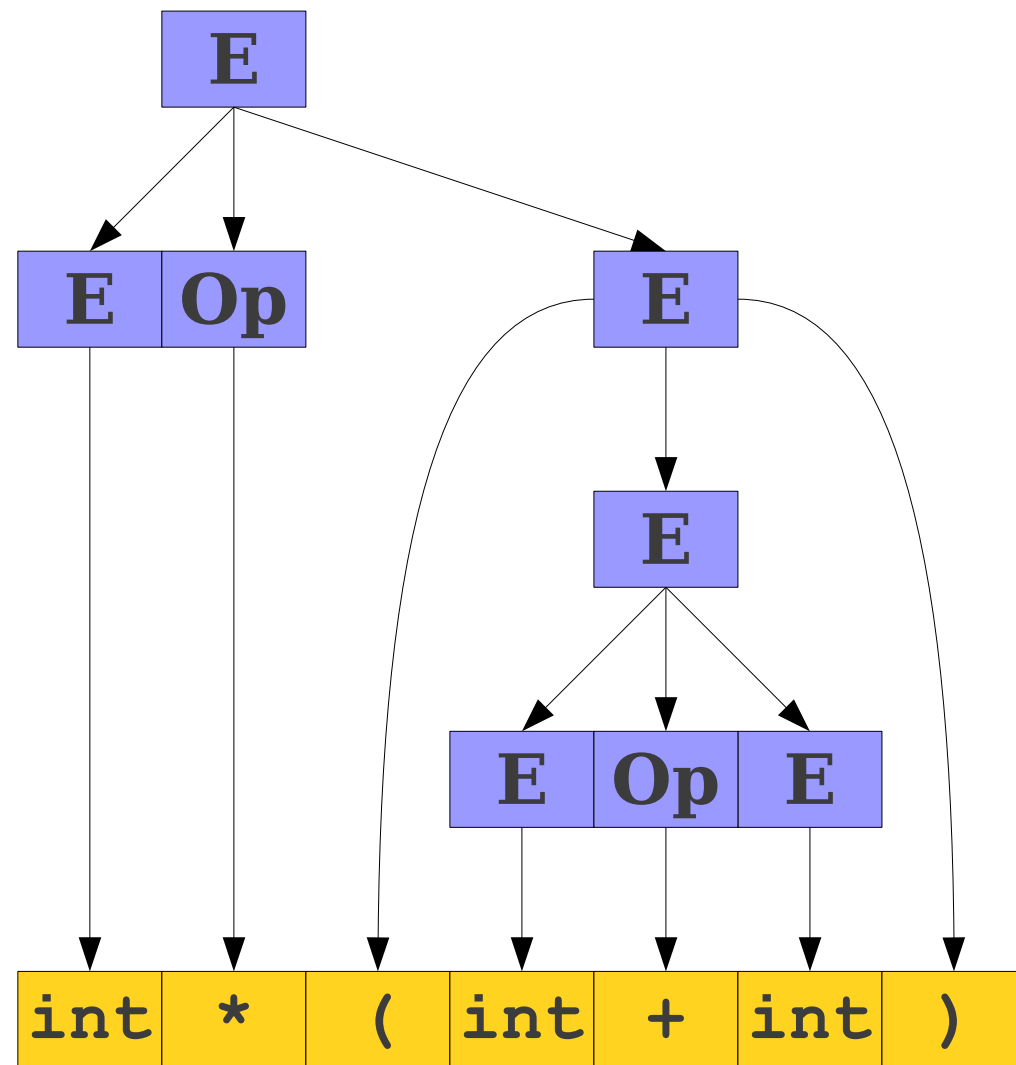
$E \rightarrow \text{int} \mid E \text{ Op } E \mid (E)$

$\text{Op} \rightarrow + \mid - \mid * \mid /$

E
 $\Rightarrow E \text{ Op } E$
 $\Rightarrow E \text{ Op } (E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } E)$
 $\Rightarrow E \text{ Op } (E \text{ Op } \text{int})$
 $\Rightarrow E \text{ Op } (E + \text{int})$
 $\Rightarrow E * (\text{int} + \text{int})$
 $\Rightarrow \text{int} * (\text{int} + \text{int})$



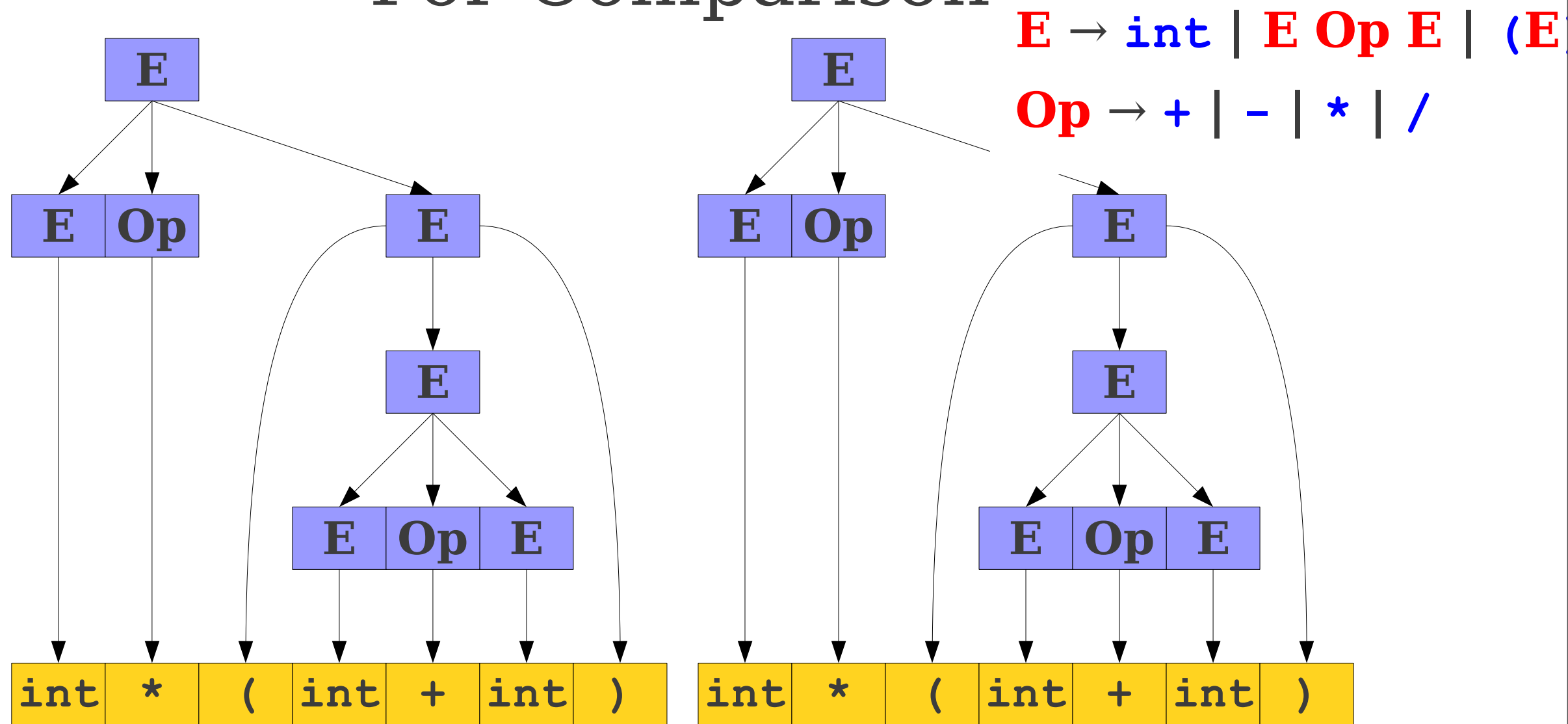
For Comparison



E → **int** | **E Op E** | **(E)**

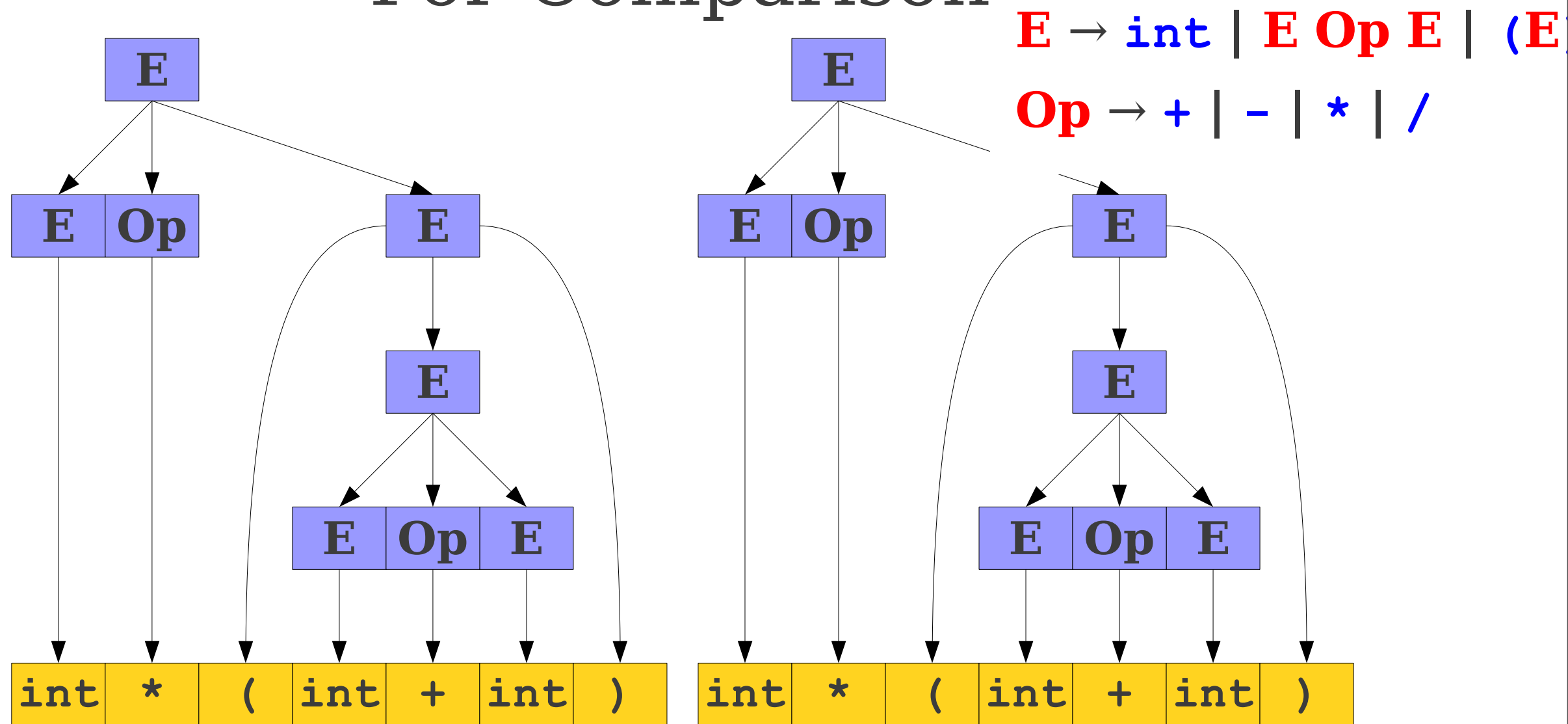
Op → **+** | **-** | ***** | **/**

For Comparison



Left-most derivation and right-most derivation generate the same parse tree

For Comparison



Left-most derivation and right-most derivation generate the same parse tree

The order of the construction is different

Parse Trees

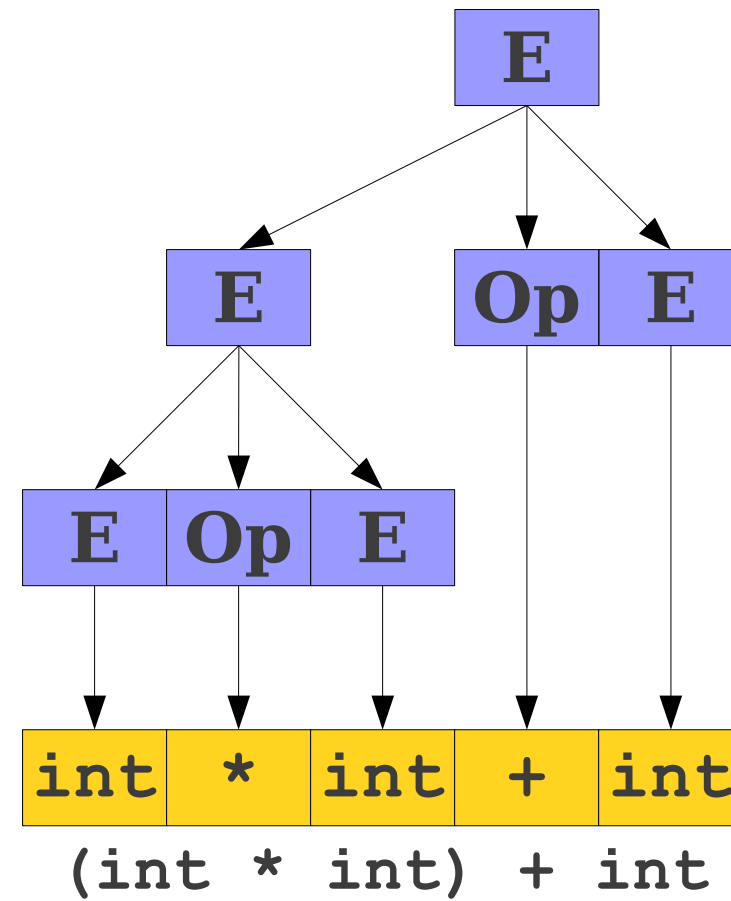
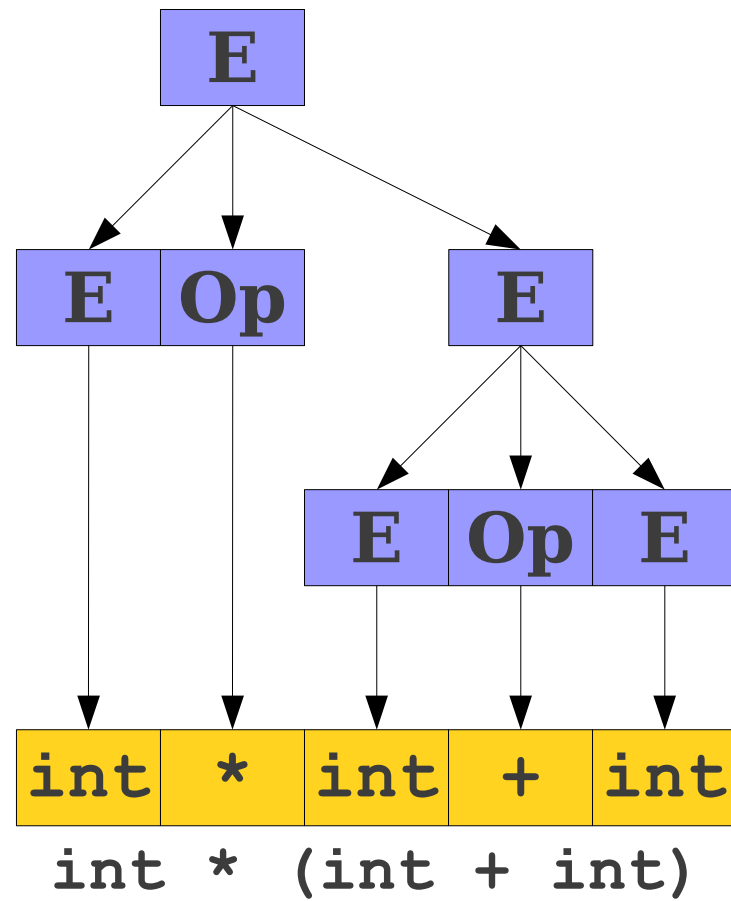
- A **parse tree** is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Inorder walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

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 will discuss how to do this more this Wednesday ...**

Challenges in Parsing

A Serious Problem



Ambiguity

- A CFG is said to be **ambiguous** if there is at least one string with two or more parse trees.
- Note that ambiguity is a property of *grammars*, not *languages*.
- There is no algorithm for converting an arbitrary ambiguous grammar into an unambiguous one.
 - Some languages are inherently ambiguous, meaning that no unambiguous grammar exists for them.
- There is no algorithm for detecting whether an arbitrary grammar is ambiguous.

Is Ambiguity a Problem?

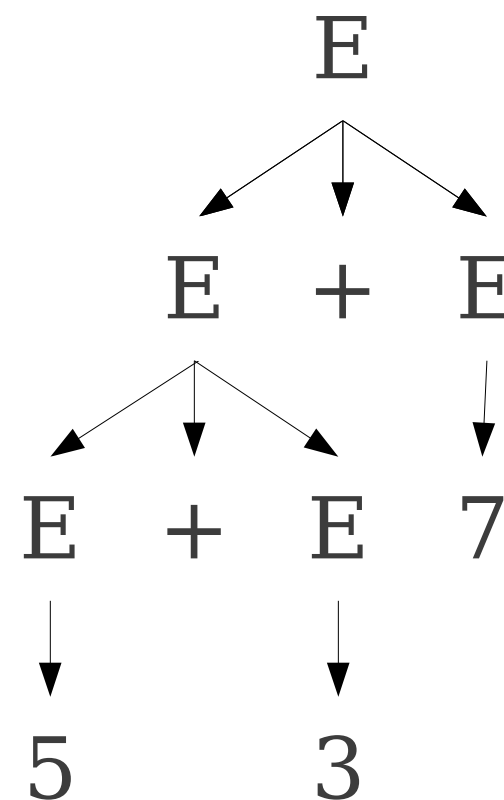
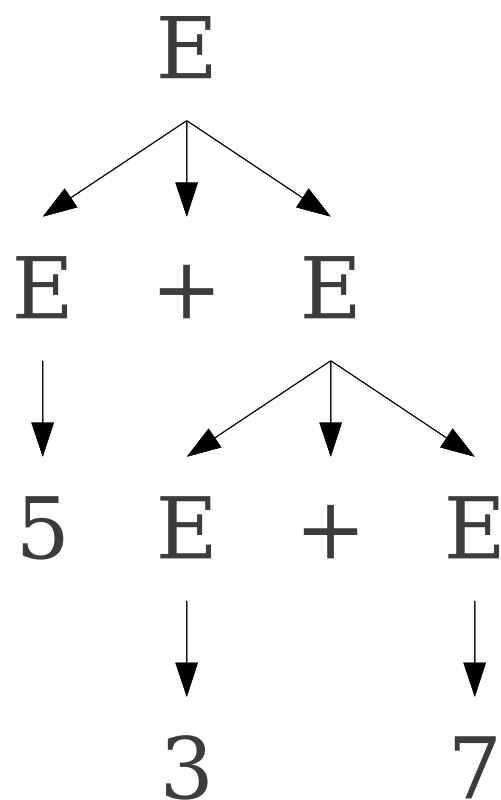
- Depends on **semantics**.

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

Is Ambiguity a Problem?

- Depends on **semantics**.

E $\rightarrow$ **int** | **E** + **E**



Is Ambiguity a Problem?

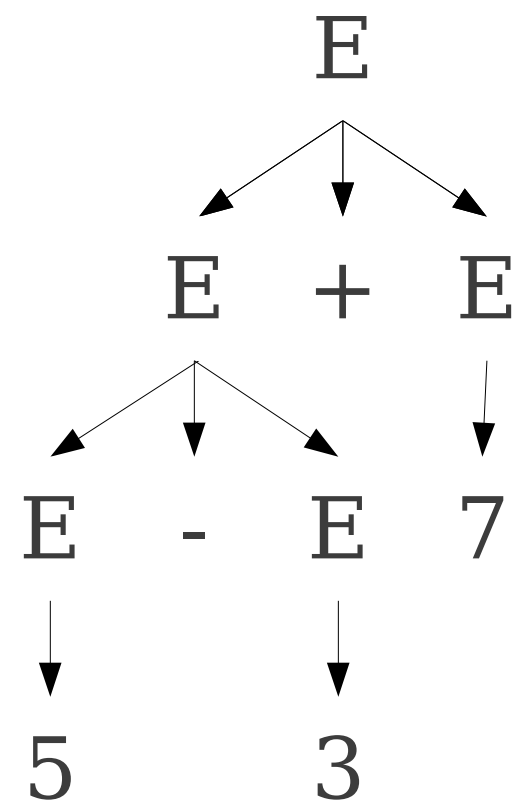
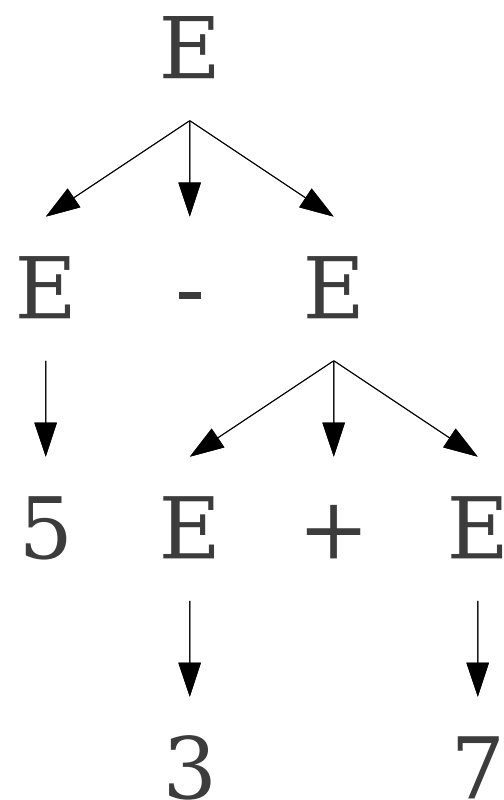
- Depends on **semantics**.

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

Is Ambiguity a Problem?

- Depends on **semantics**.

E $\rightarrow$ **int** | **E + E** | **E - E**



Different Parse Trees

$S \rightarrow S + S \mid S * S \mid \text{number}$

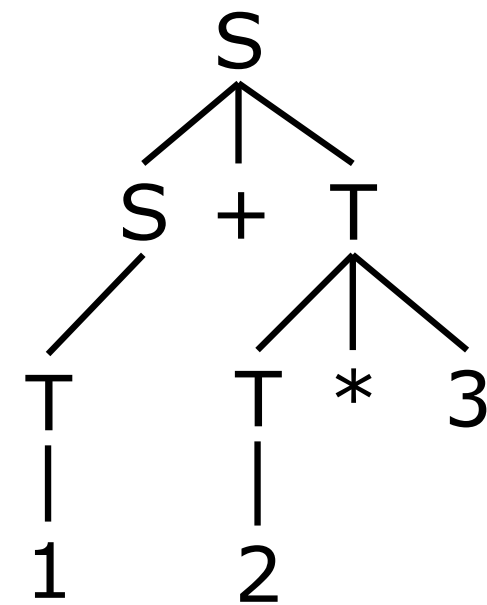
- Consider expression $1 + 2 * 3$
- Derivation 1: $S \rightarrow S + S \rightarrow 1 + S \rightarrow 1 + S * S \rightarrow 1 + 2 * S \rightarrow 1 + 2 * 3$
- Derivation 2: $S \rightarrow S * S \rightarrow S * 3 \rightarrow S + S * 3 \rightarrow S + 2 * 3 \rightarrow 1 + 2 * 3$



Eliminating Ambiguity



- Often can eliminate ambiguity by adding non-terminals & allowing recursion only on right or left
- $S \rightarrow S + T \mid T$
- $T \rightarrow T * \text{num} \mid \text{num}$
- T non-terminal enforces precedence
- Left-recursion : left-associativity



Another example:

Context-Free Grammars

- A regular expression can be
 - Any letter
 - ϵ
 - The concatenation of regular expressions.
 - The union of regular expressions.
 - The Kleene closure of a regular expression.
 - A parenthesized regular expression.

Context-Free Grammars

- This gives us the following CFG:

$$\mathbf{R} \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \dots$$

$$\mathbf{R} \rightarrow \text{"}\epsilon\text{"}$$

$$\mathbf{R} \rightarrow \mathbf{RR}$$

$$\mathbf{R} \rightarrow \mathbf{R} \text{"} \mid \text{"} \mathbf{R}$$

$$\mathbf{R} \rightarrow \mathbf{R}^*$$

$$\mathbf{R} \rightarrow (\mathbf{R})$$

An Ambiguous Grammar

$R \rightarrow a \mid b \mid c \mid \dots$

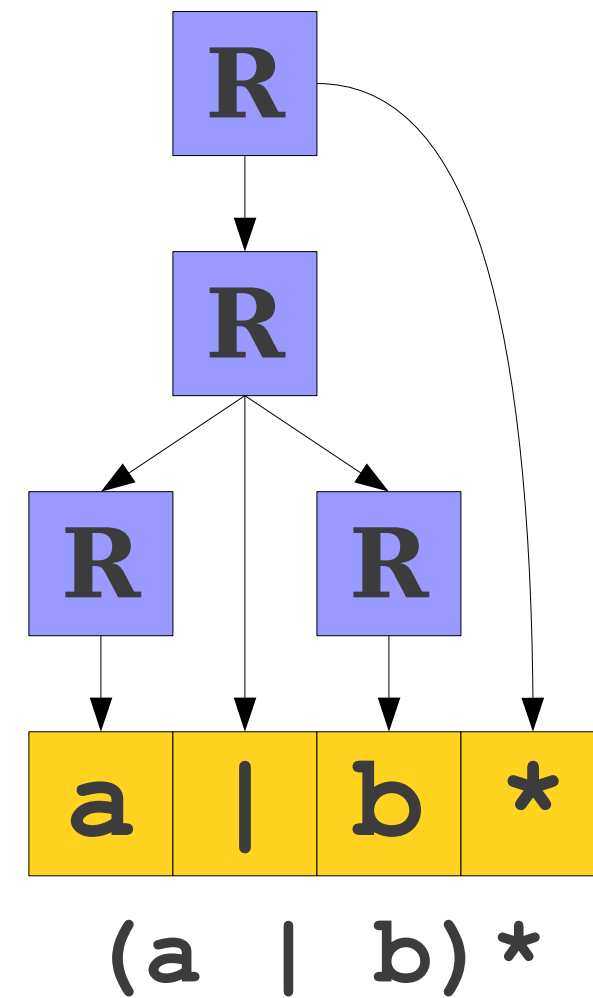
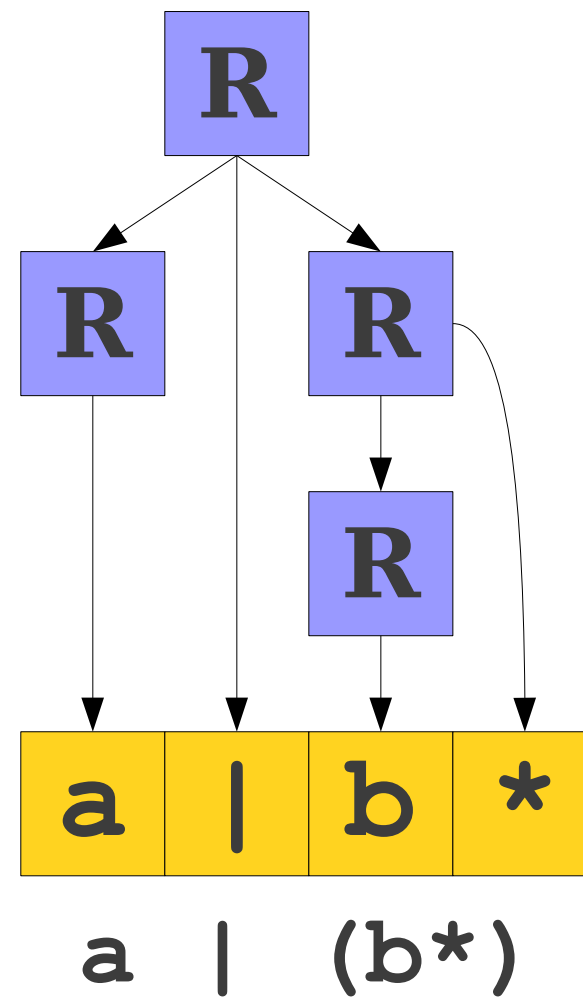
$R \rightarrow \epsilon$

$R \rightarrow RR$

$R \rightarrow R \mid R$

$R \rightarrow R^*$

$R \rightarrow (R)$



Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

$$\mathbf{R} \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \dots$$

$$\mathbf{R} \rightarrow \text{"}\epsilon\text{"}$$

$$\mathbf{R} \rightarrow \mathbf{RR}$$

$$\mathbf{R} \rightarrow \mathbf{R} \text{"}\mid\text{"} \mathbf{R}$$

$$\mathbf{R} \rightarrow \mathbf{R}^*$$

$$\mathbf{R} \rightarrow (\mathbf{R})$$

$$\mathbf{R} \rightarrow \mathbf{S} \mid \mathbf{R} \text{"}\mid\text{"} \mathbf{S}$$

$$\mathbf{S} \rightarrow \mathbf{T} \mid \mathbf{ST}$$

$$\mathbf{T} \rightarrow \mathbf{U} \mid \mathbf{T}^*$$

$$\mathbf{U} \rightarrow \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \dots$$

$$\mathbf{U} \rightarrow \text{"}\epsilon\text{"}$$

$$\mathbf{U} \rightarrow (\mathbf{R})$$

Why is this unambiguous?

$$R \rightarrow S \mid R \text{ “|” } S$$
$$S \rightarrow T \mid ST$$
$$T \rightarrow U \mid T^*$$
$$U \rightarrow a \mid b \mid \dots$$
$$U \rightarrow \text{“}\epsilon\text{”}$$
$$U \rightarrow (R)$$

only generates
“atomic” expressions

Why is this unambiguous?

$R \rightarrow S \mid R \text{ “|” } S$

$S \rightarrow T \mid ST$

$T \rightarrow U \mid T^*$

$U \rightarrow a \mid b \mid \dots$

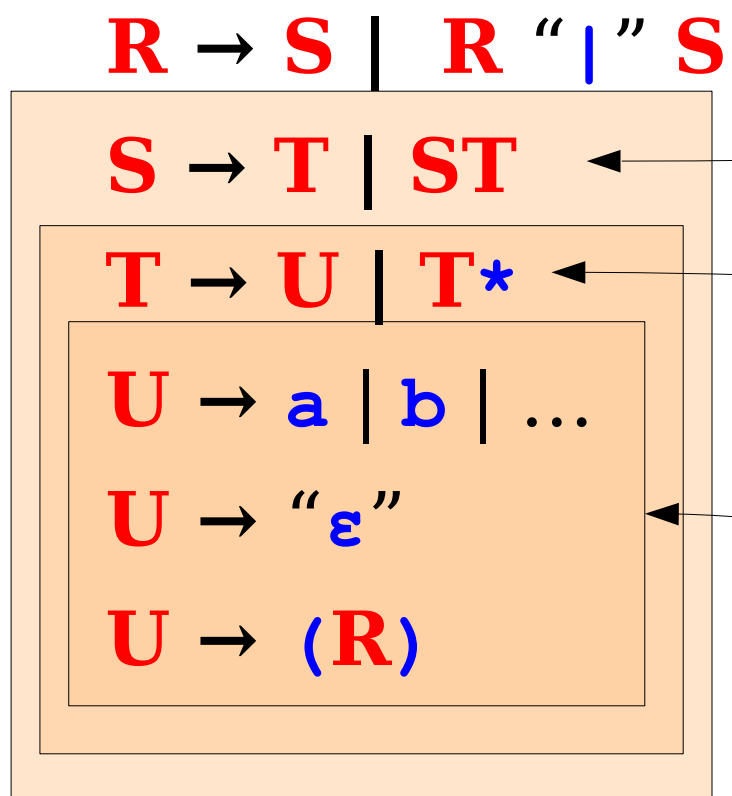
$U \rightarrow \epsilon$

$U \rightarrow (R)$

Puts stars onto
atomic expressions

only generates
“atomic” expressions

Why is this unambiguous?

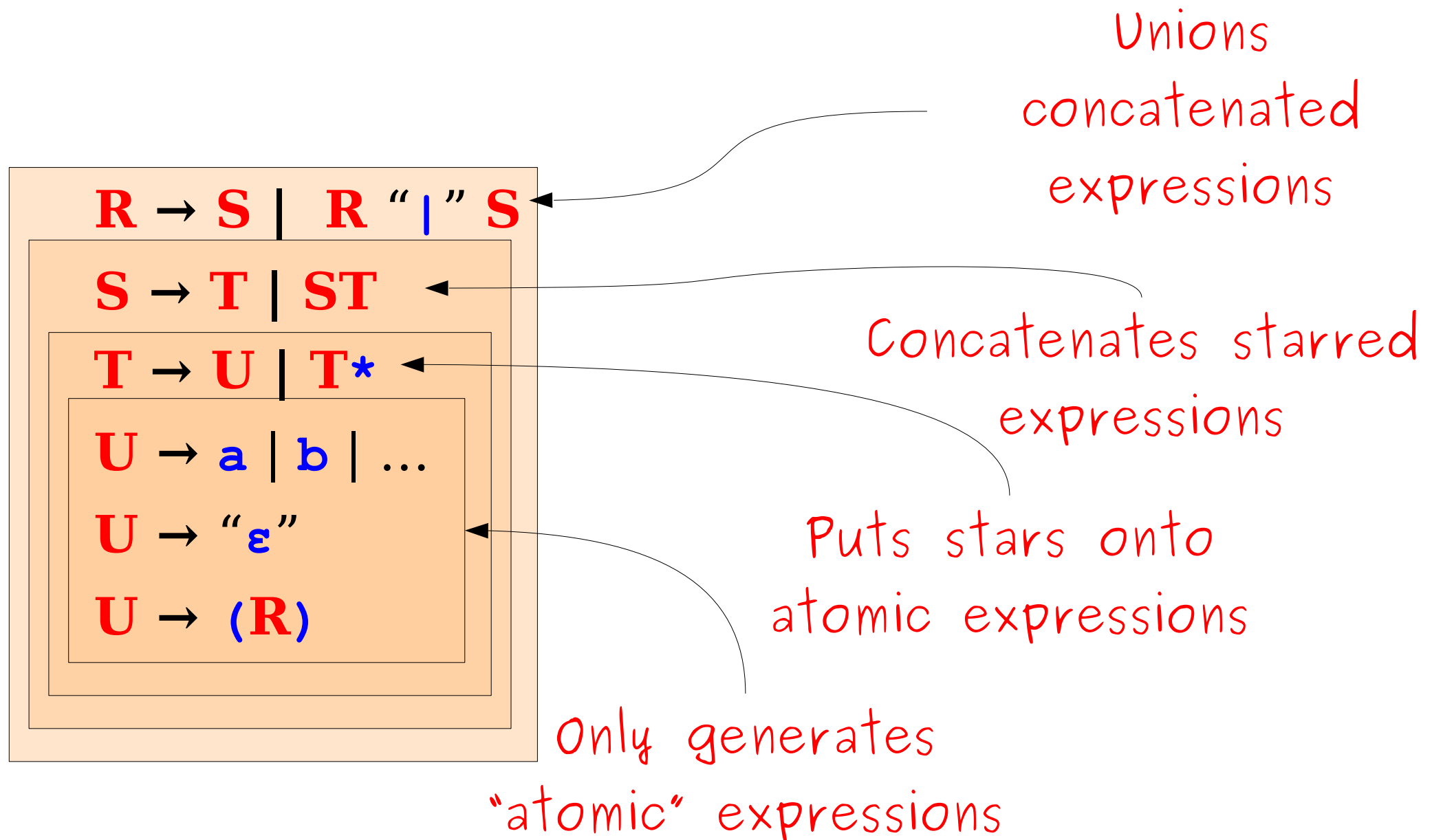


Concatenates starred
expressions

Puts stars onto
atomic expressions

Only generates
"atomic" expressions

Why is this unambiguous?



R

R $\rightarrow$ **S** | **R** “|” **S**

S $\rightarrow$ **T** | **ST**

T $\rightarrow$ **U** | **T***

U $\rightarrow$ **a** | **b** | **c** | ...

U $\rightarrow$ “ **ϵ** ”

U $\rightarrow$ (**R**)

a	b		c		a	*
---	---	--	---	--	---	---

R $\rightarrow$ **S** | **R** “|” **S**

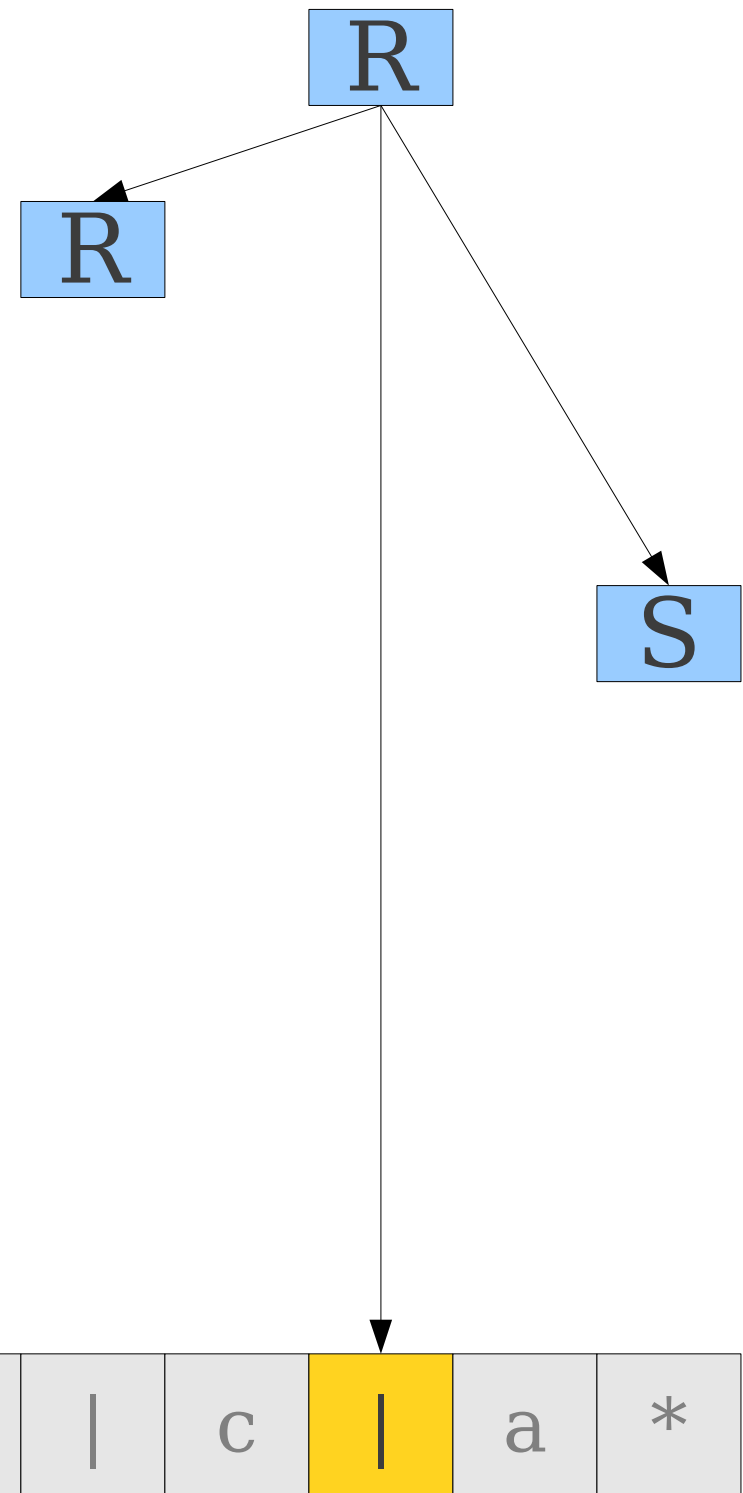
S $\rightarrow$ **T** | **ST**

T $\rightarrow$ **U** | **T***

U $\rightarrow$ **a** | **b** | **c** | ...

U $\rightarrow$ “ **ϵ** ”

U $\rightarrow$ (**R**)



R $\rightarrow$ **S** | **R** “ | ” **S**

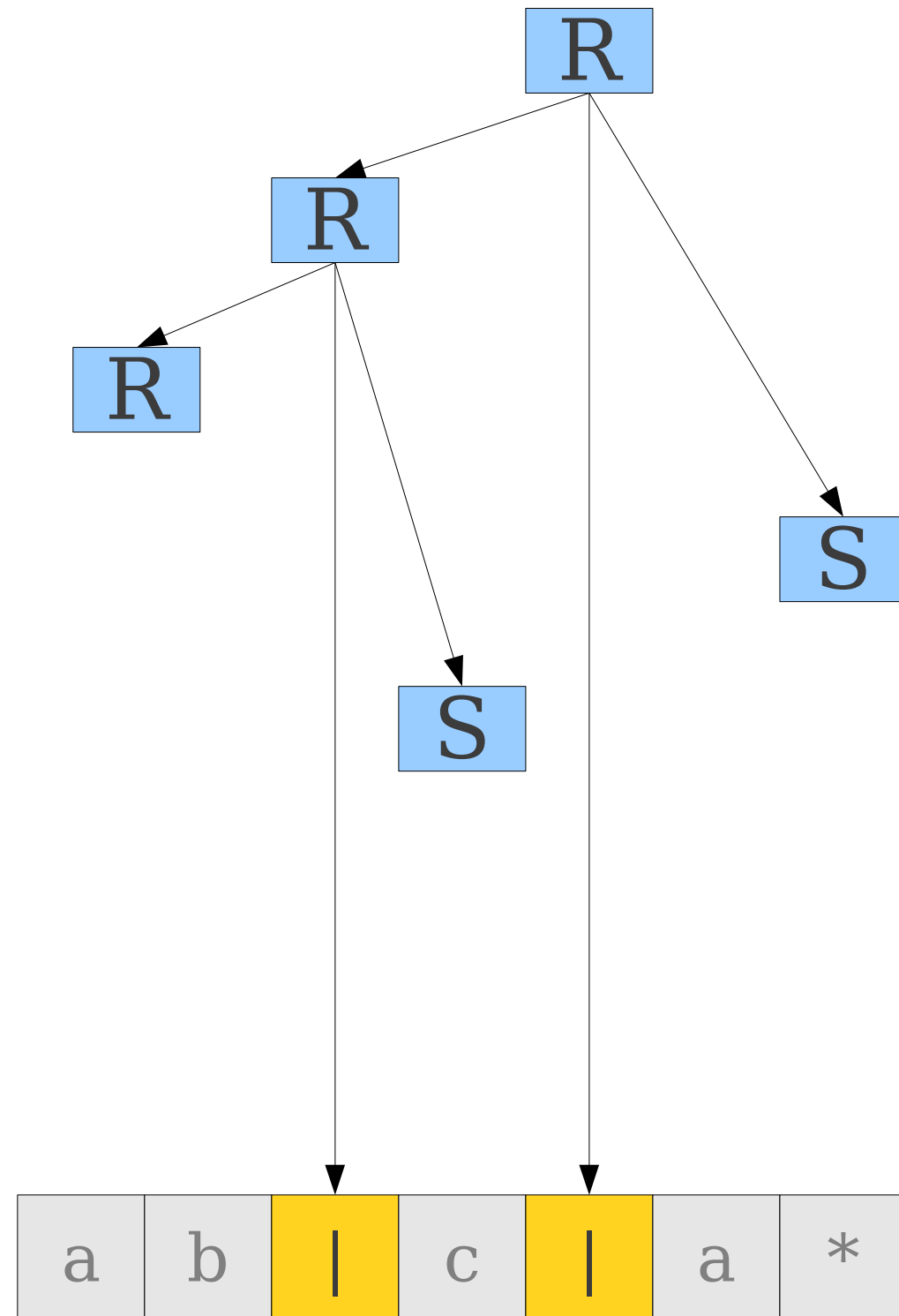
S $\rightarrow$ **T** | **ST**

T $\rightarrow$ **U** | **T***

U $\rightarrow$ **a** | **b** | **c** | ...

U $\rightarrow$ “ **ε** ”

U $\rightarrow$ **(R)**



$R \rightarrow S \mid R \text{ “|” } S$

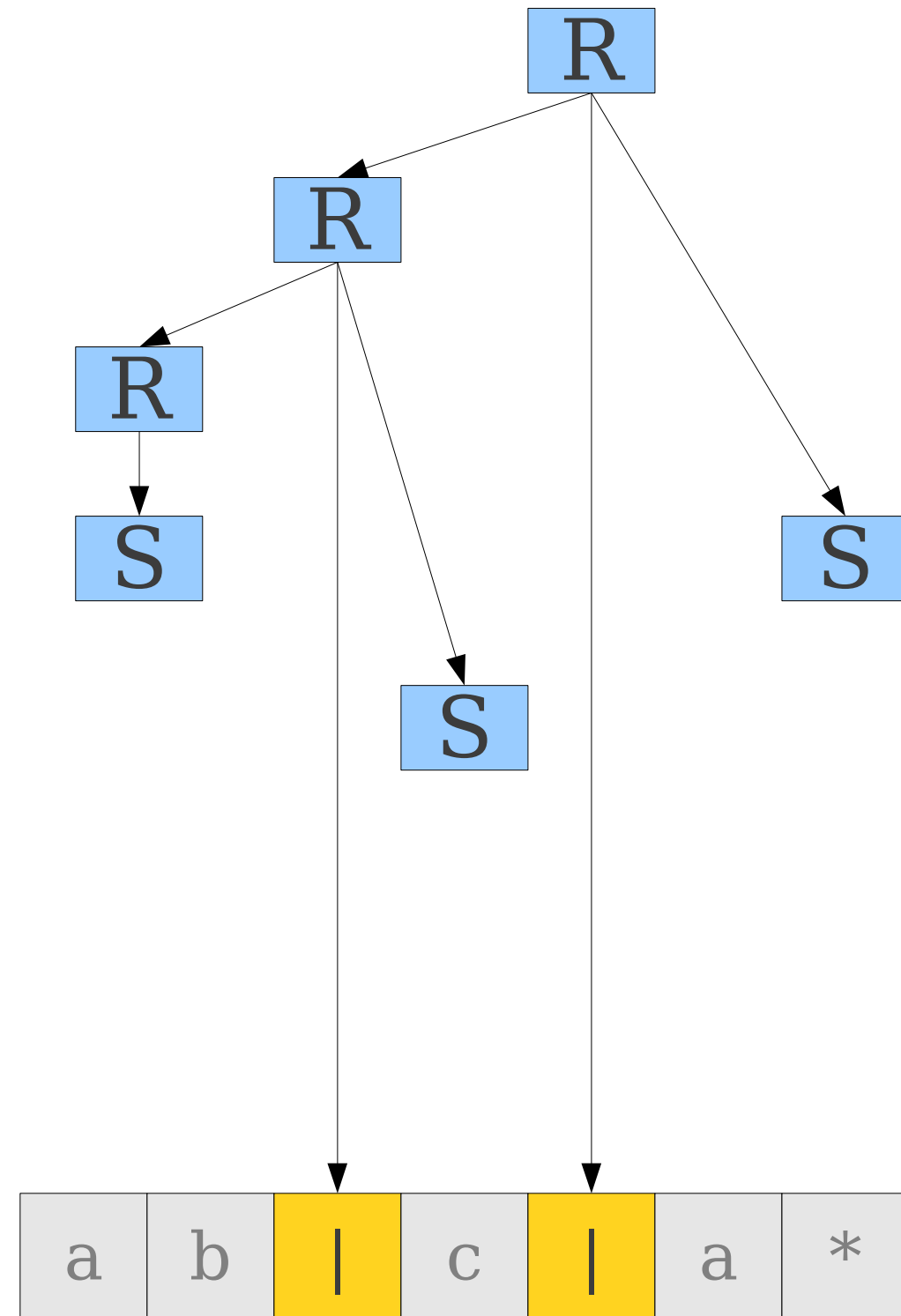
$S \rightarrow T \mid ST$

$T \rightarrow U \mid T^*$

$U \rightarrow a \mid b \mid c \mid \dots$

$U \rightarrow \text{“}\epsilon\text{”}$

$U \rightarrow (R)$



$R \rightarrow S \mid R \text{ “|” } S$

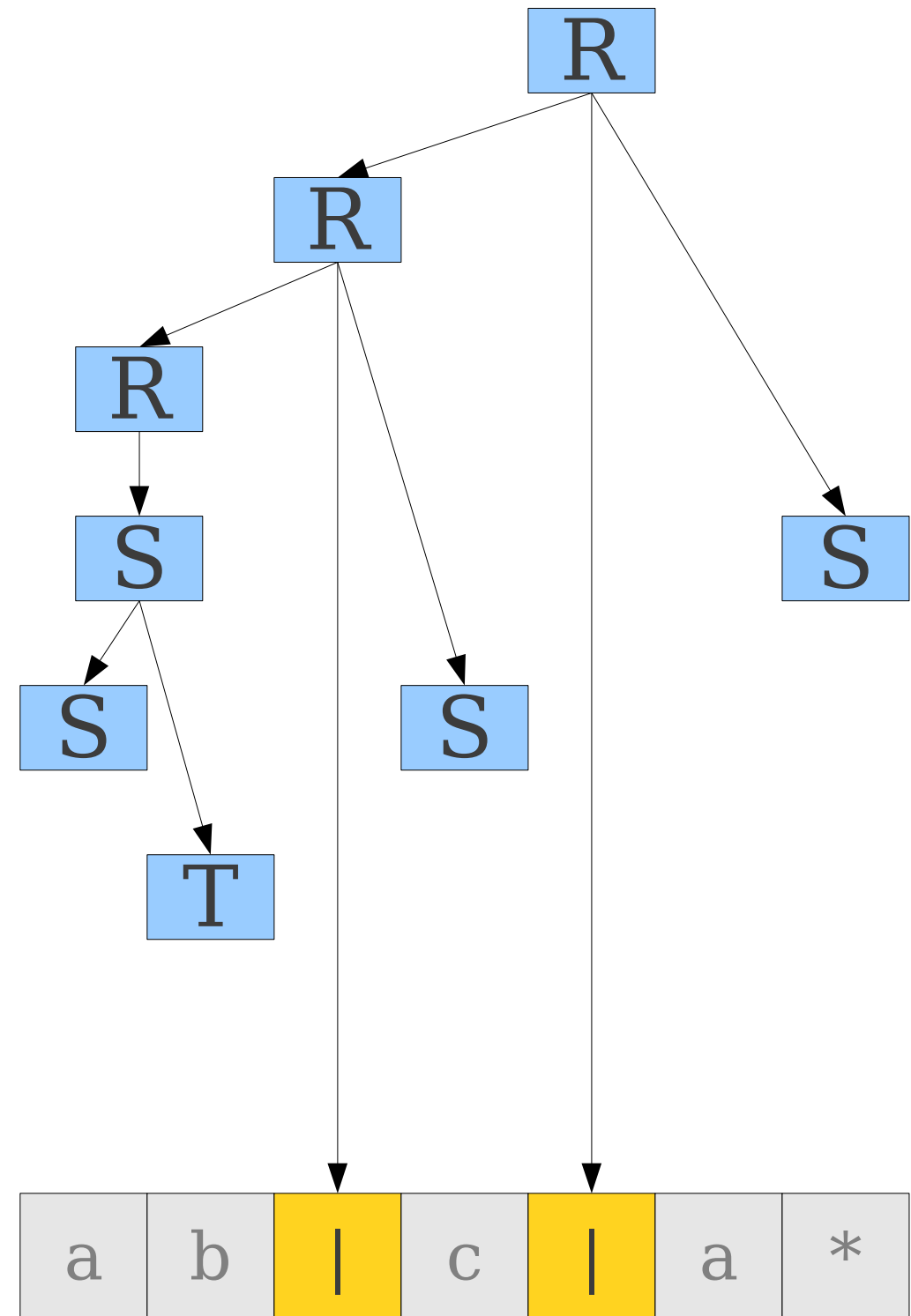
$S \rightarrow T \mid ST$

$T \rightarrow U \mid T^*$

$U \rightarrow a \mid b \mid c \mid \dots$

$U \rightarrow \text{“}\epsilon\text{”}$

$U \rightarrow (R)$



R $\rightarrow$ **S** | **R** “ | ” **S**

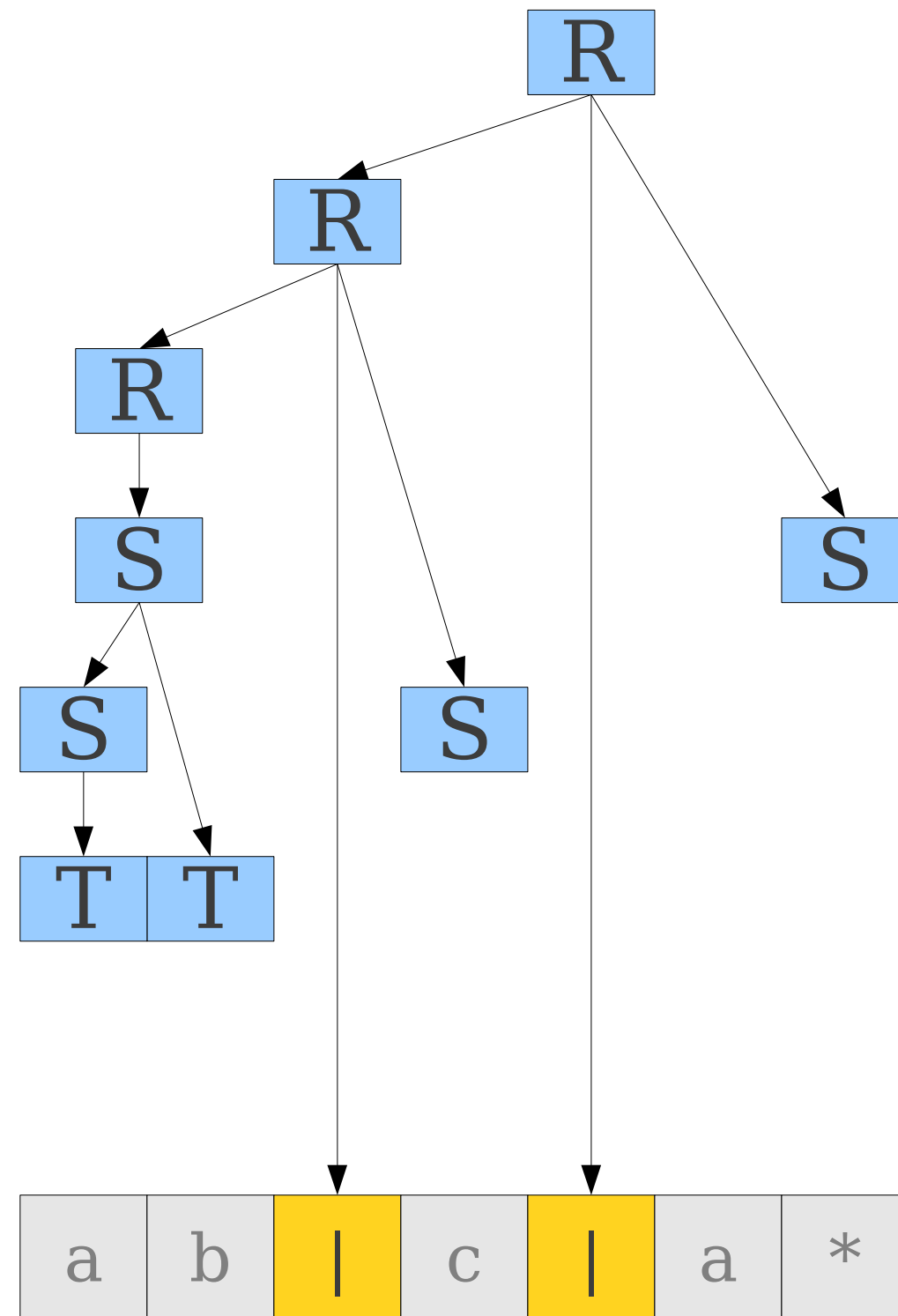
S $\rightarrow$ **T** | **ST**

T $\rightarrow$ **U** | **T***

U $\rightarrow$ **a** | **b** | **c** | ...

U → “**ε**”

U $\rightarrow$ **(R)**



$R \rightarrow S \mid R \text{ “|” } S$

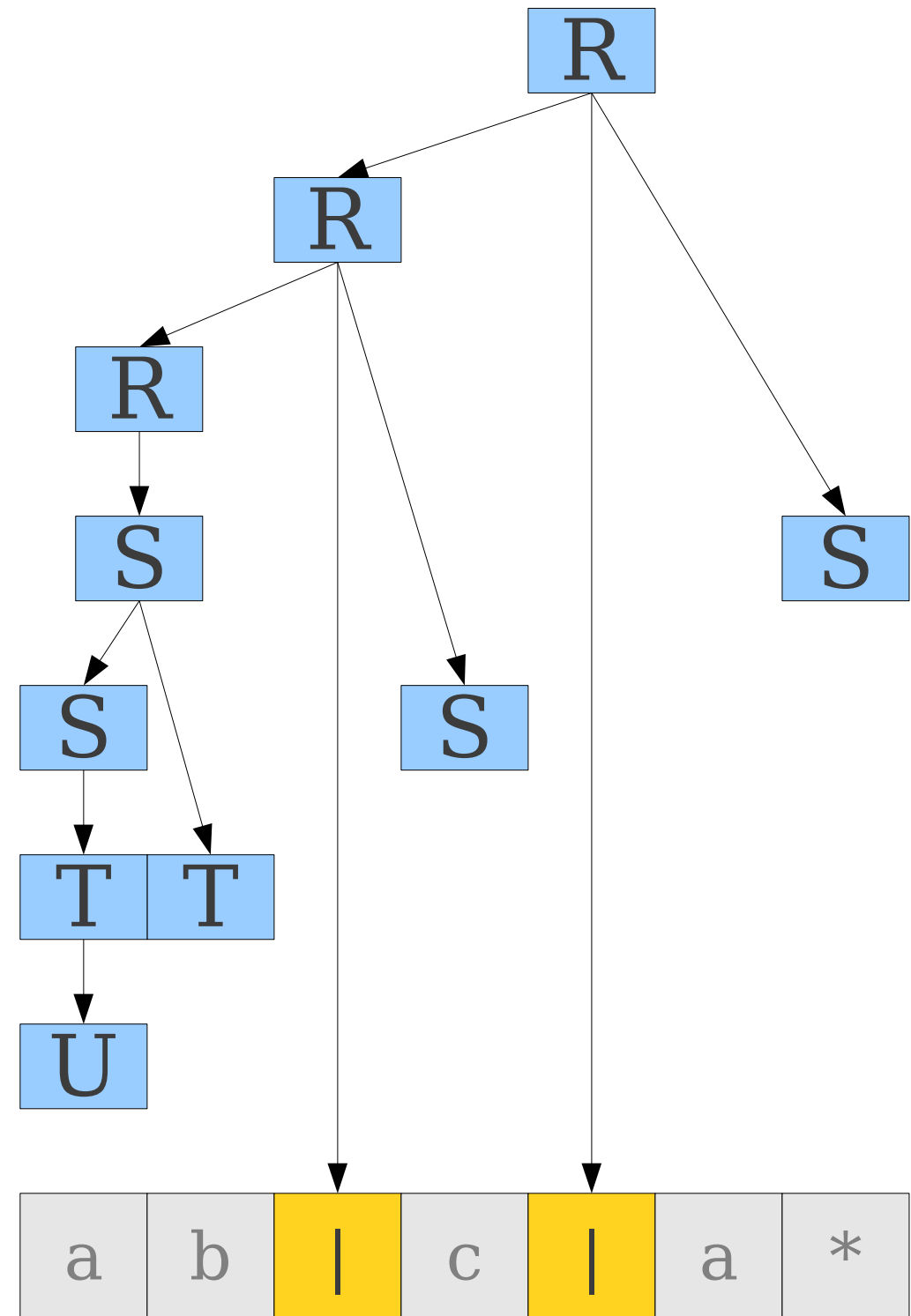
$S \rightarrow T \mid ST$

$T \rightarrow U \mid T^*$

$U \rightarrow a \mid b \mid c \mid \dots$

$U \rightarrow \text{“}\epsilon\text{”}$

$U \rightarrow (R)$



$R \rightarrow S \mid R \text{ “|” } S$

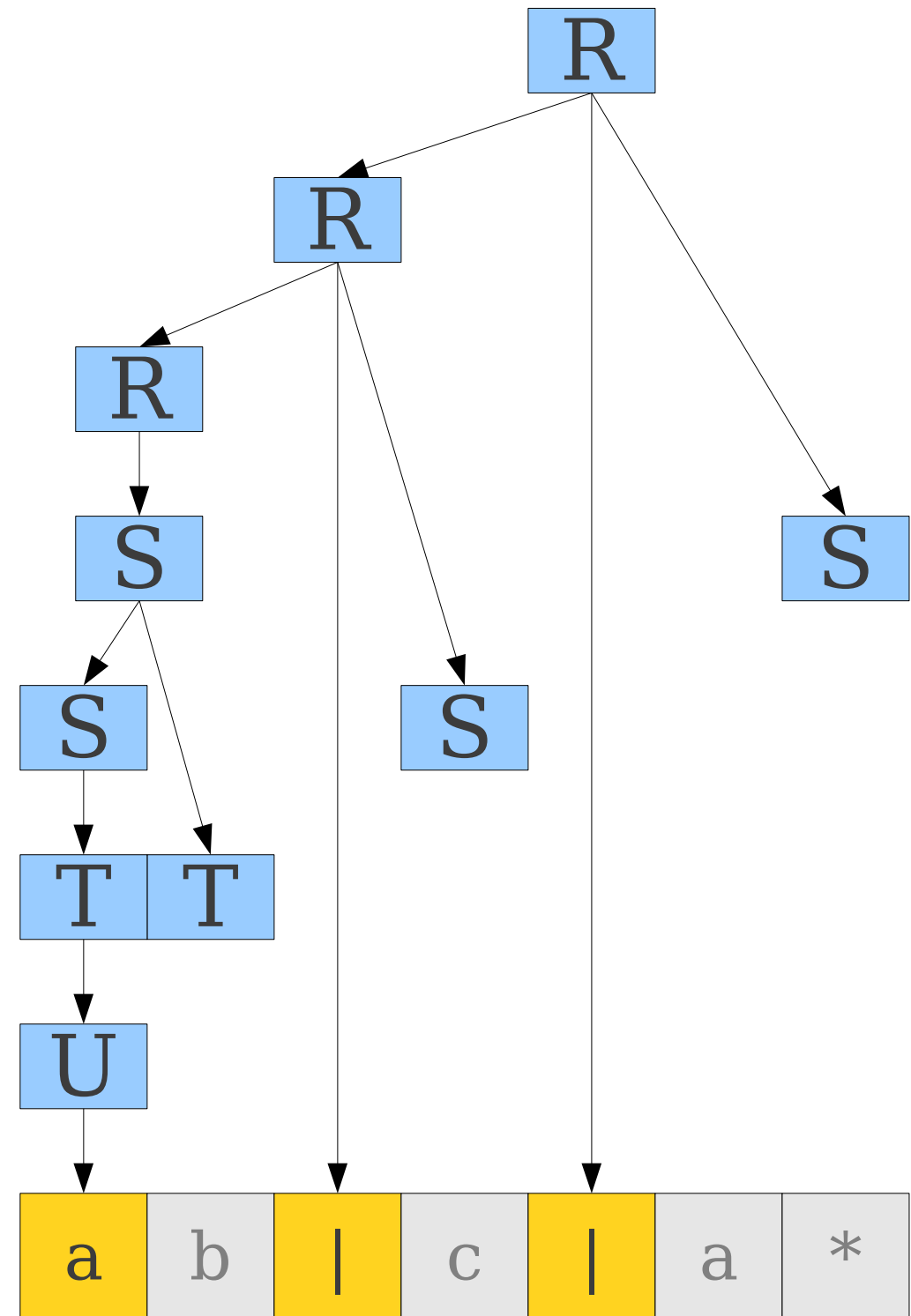
$S \rightarrow T \mid ST$

$T \rightarrow U \mid T^*$

$U \rightarrow a \mid b \mid c \mid \dots$

$U \rightarrow \text{“}\epsilon\text{”}$

$U \rightarrow (R)$



$R \rightarrow S \mid R \mid S$

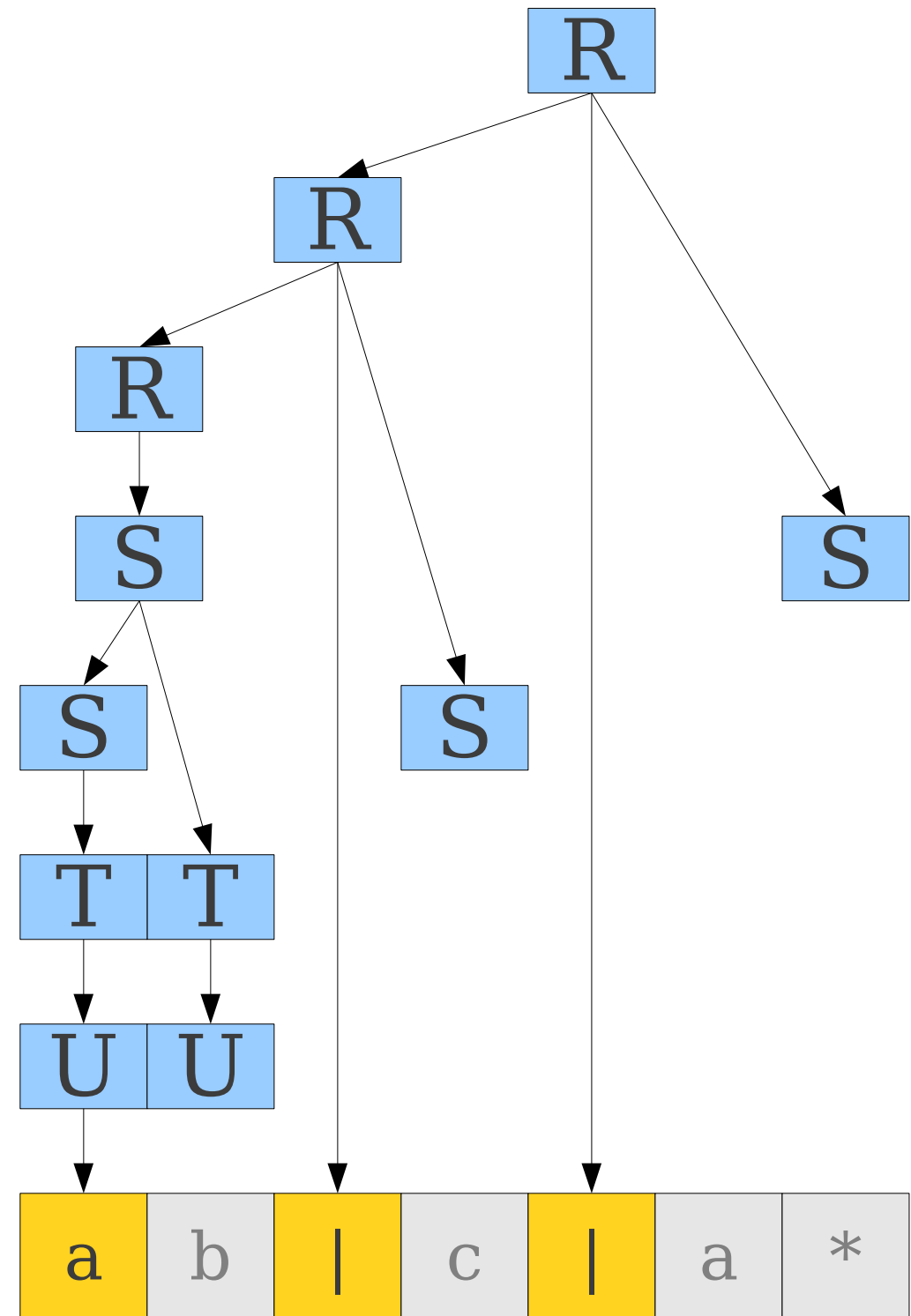
$S \rightarrow T \mid ST$

$T \rightarrow U \mid T^*$

$U \rightarrow a \mid b \mid c \mid \dots$

$U \rightarrow \epsilon$

$U \rightarrow (R)$



R $\rightarrow$ **S** | **R** “|” **S**

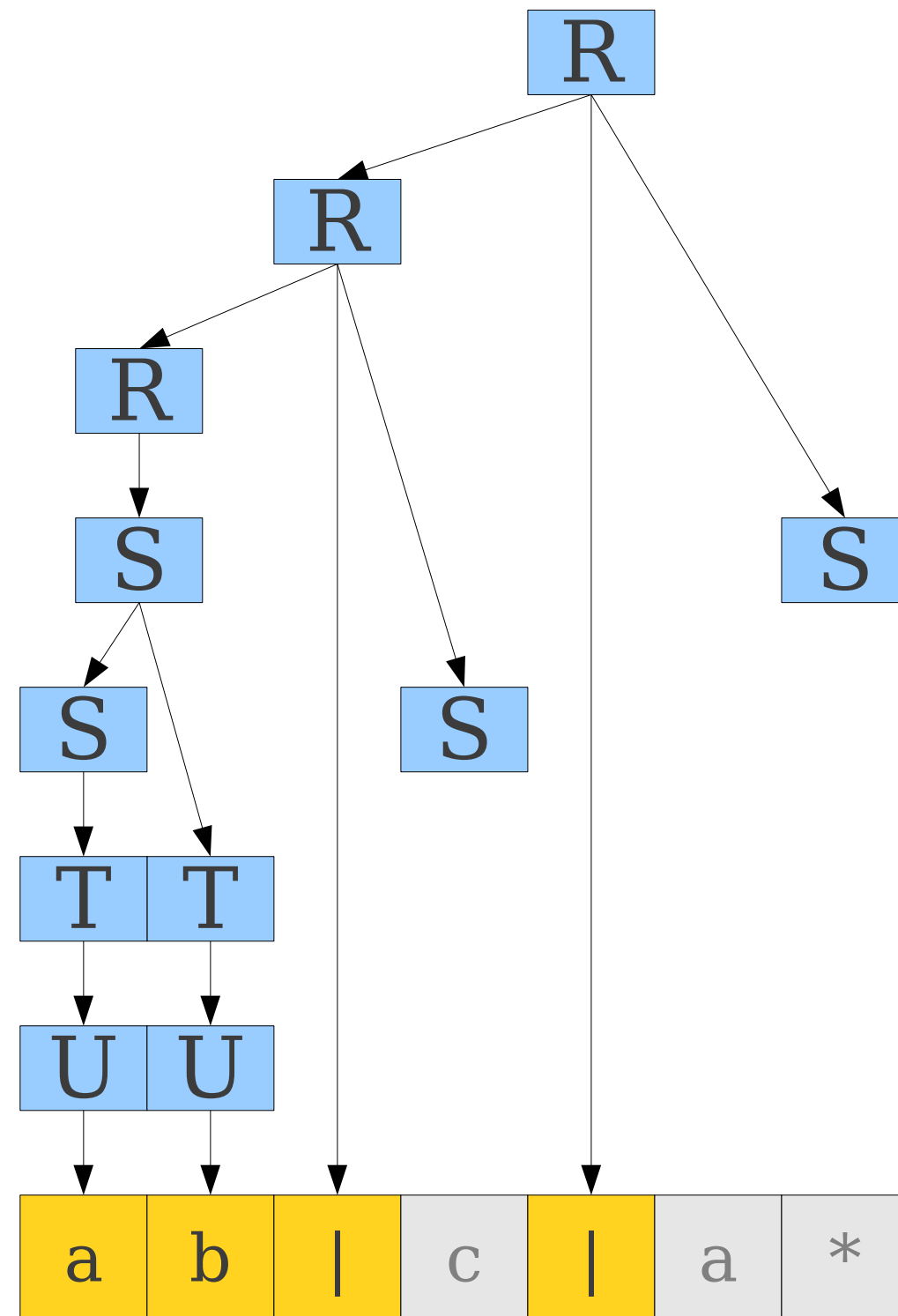
S $\rightarrow$ **T** | **ST**

T $\rightarrow$ **U** | **T***

U $\rightarrow$ **a** | **b** | **c** | ...

U $\rightarrow$ "ε"

U $\rightarrow$ **(R)**



$R \rightarrow S \mid R \mid S$

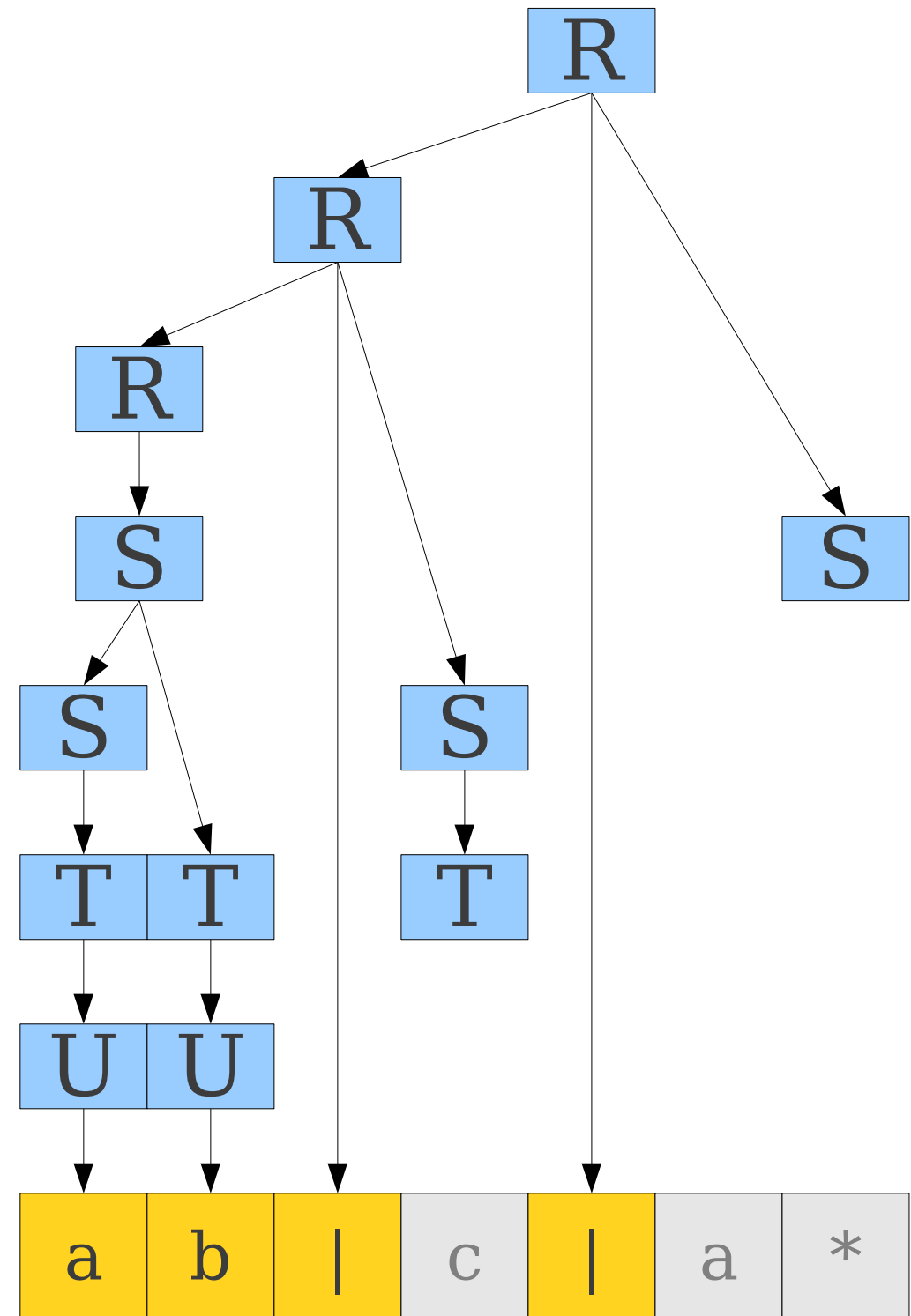
$S \rightarrow T \mid ST$

$T \rightarrow U \mid T^*$

$U \rightarrow a \mid b \mid c \mid \dots$

$U \rightarrow \epsilon$

$U \rightarrow (R)$



R $\rightarrow$ **S** | **R** “ | ” **S**

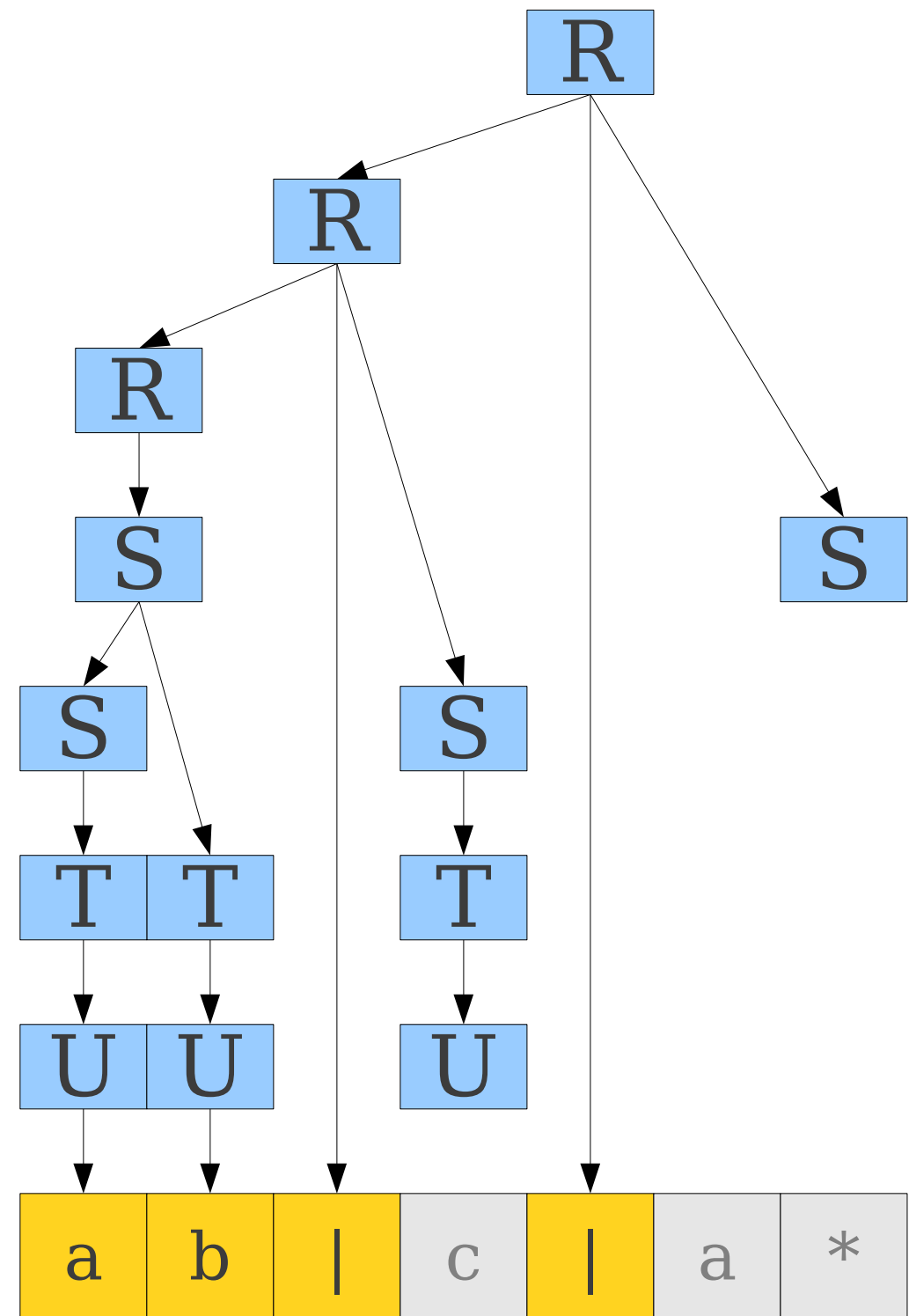
S $\rightarrow$ **T** | **ST**

T → **U** | **T***

U $\rightarrow$ **a** | **b** | **c** | ...

U $\rightarrow$ “ **ϵ** ”

U $\rightarrow$ **(R)**



R $\rightarrow$ **S** | **R** “ | ” **S**

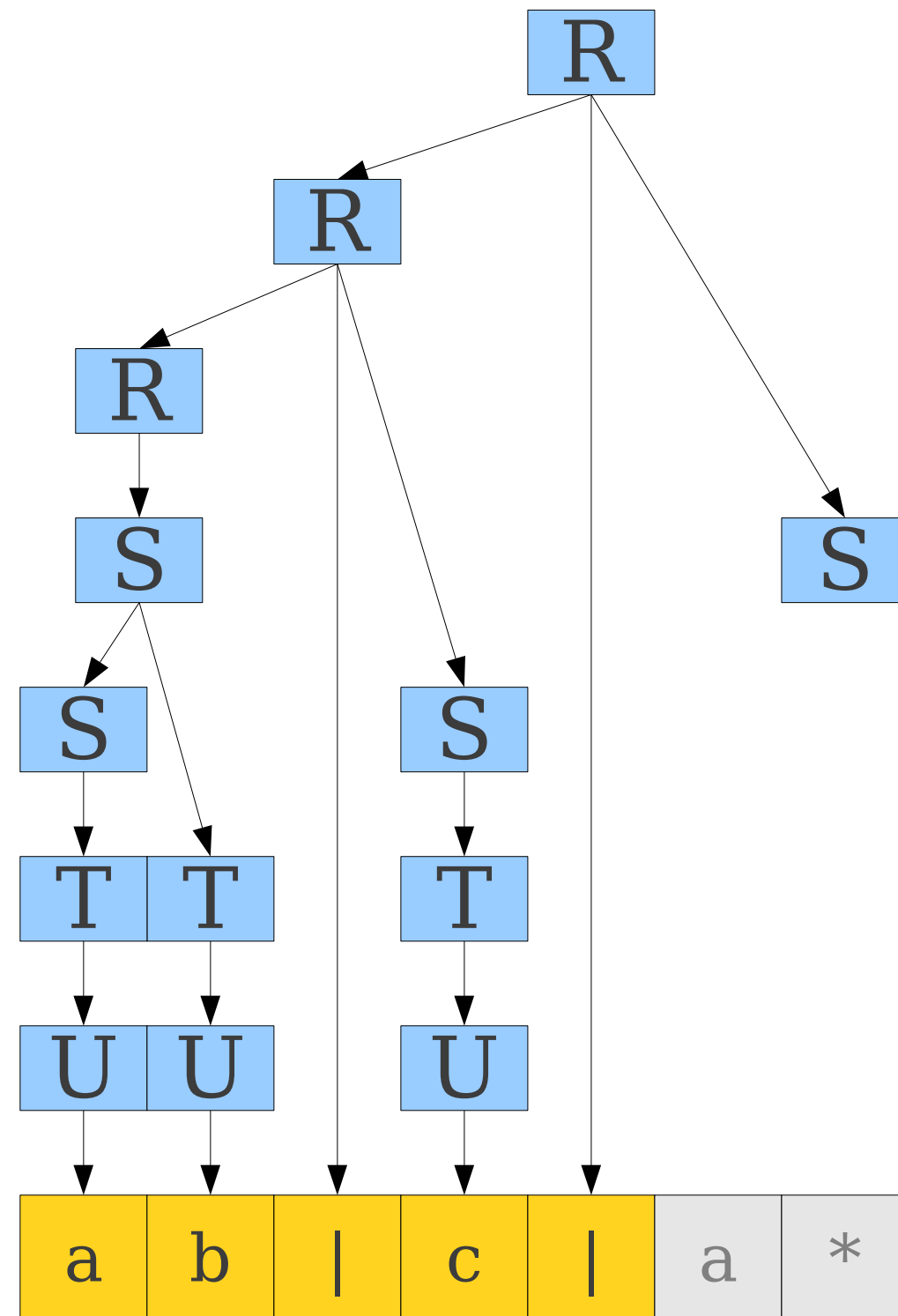
S $\rightarrow$ **T** | **ST**

T $\rightarrow$ **U** | **T***

U $\rightarrow$ **a** | **b** | **c** | ...

U $\rightarrow$ “ **ϵ** ”

U $\rightarrow$ **(R)**



R $\rightarrow$ **S** | **R** “|” **S**

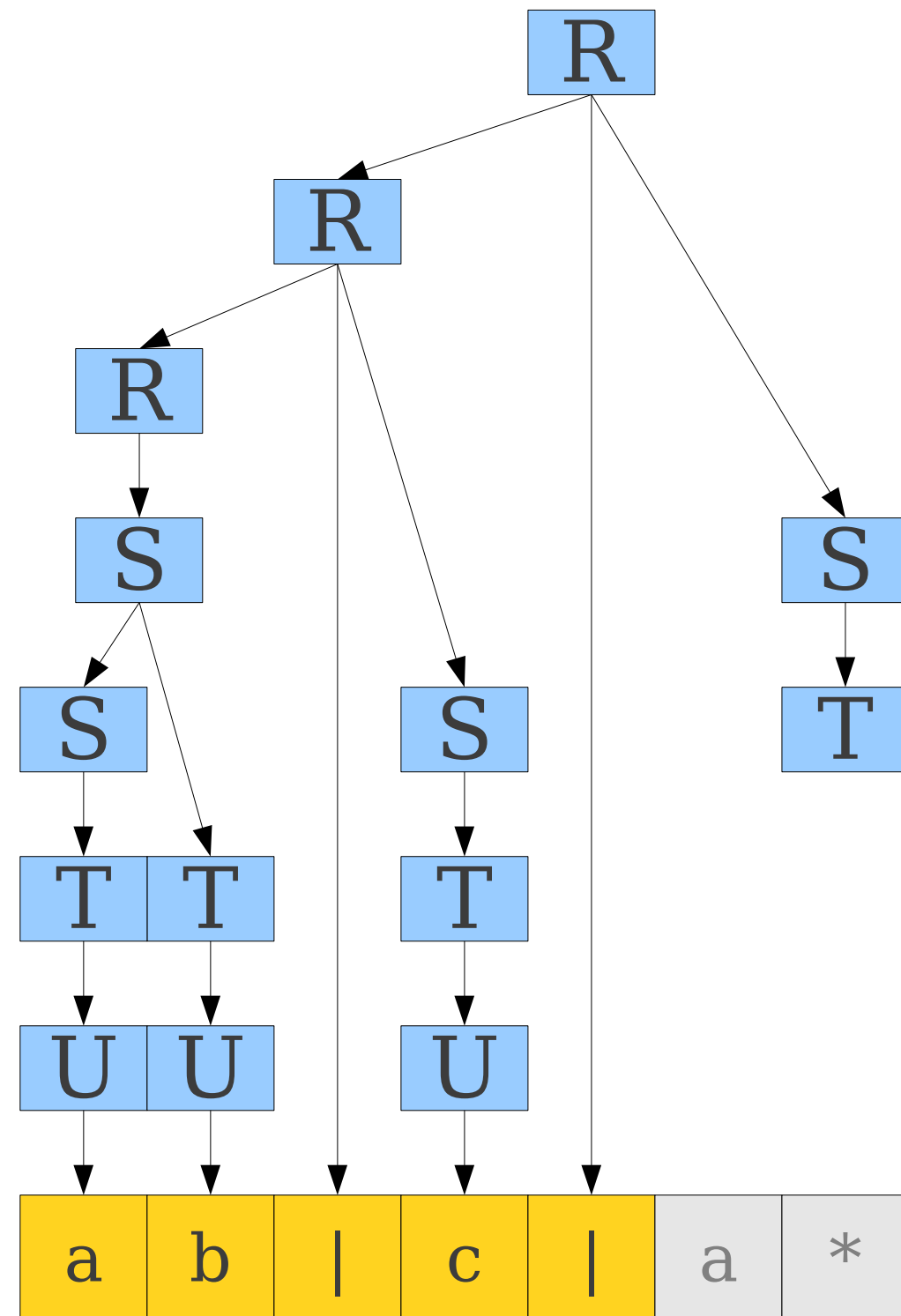
S $\rightarrow$ **T** | **ST**

T $\rightarrow$ **U** | **T***

U $\rightarrow$ **a** | **b** | **c** | ...

U → “**ε**”

U $\rightarrow$ **(R)**



R $\rightarrow$ **S** | **R** “ | ” **S**

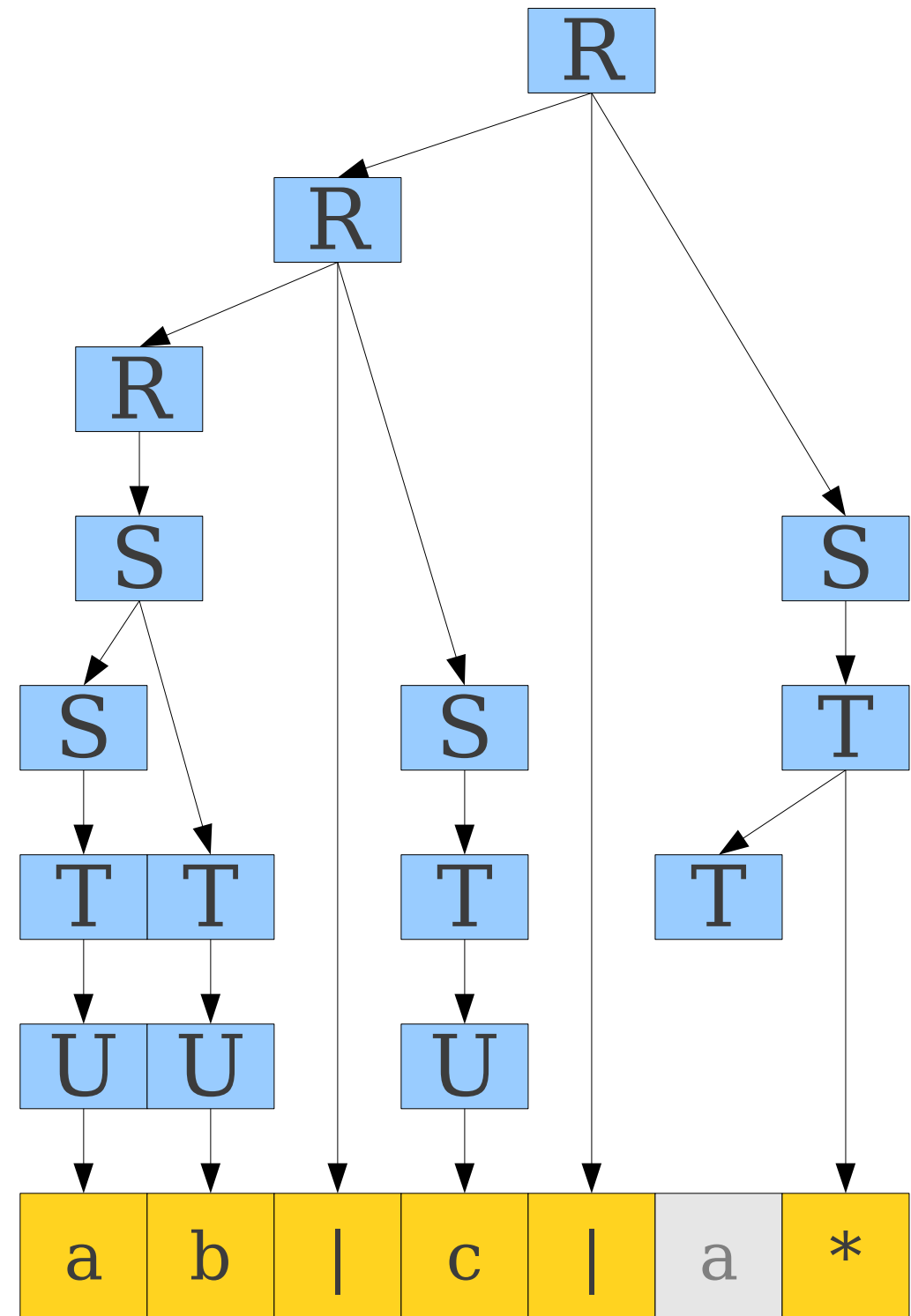
S $\rightarrow$ **T** | **ST**

T $\rightarrow$ **U** | **T***

U $\rightarrow$ **a** | **b** | **c** | ...

U → “ε”

U $\rightarrow$ **(R)**



$R \rightarrow S \mid R \text{ “|” } S$

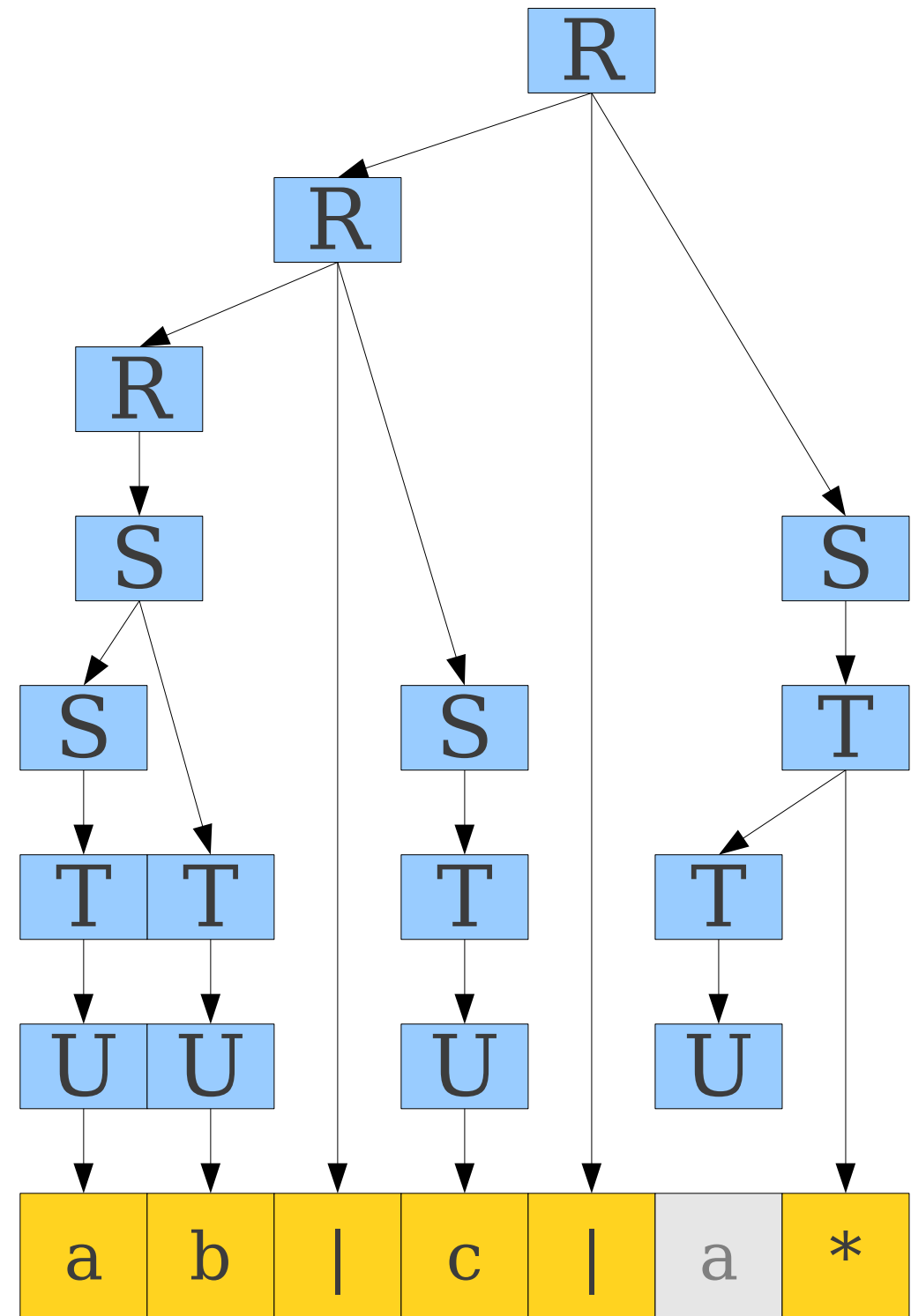
$S \rightarrow T \mid ST$

$T \rightarrow U \mid T^*$

$U \rightarrow a \mid b \mid c \mid \dots$

$U \rightarrow \text{“}\epsilon\text{”}$

$U \rightarrow (R)$



$R \rightarrow S \mid R \text{ “|” } S$

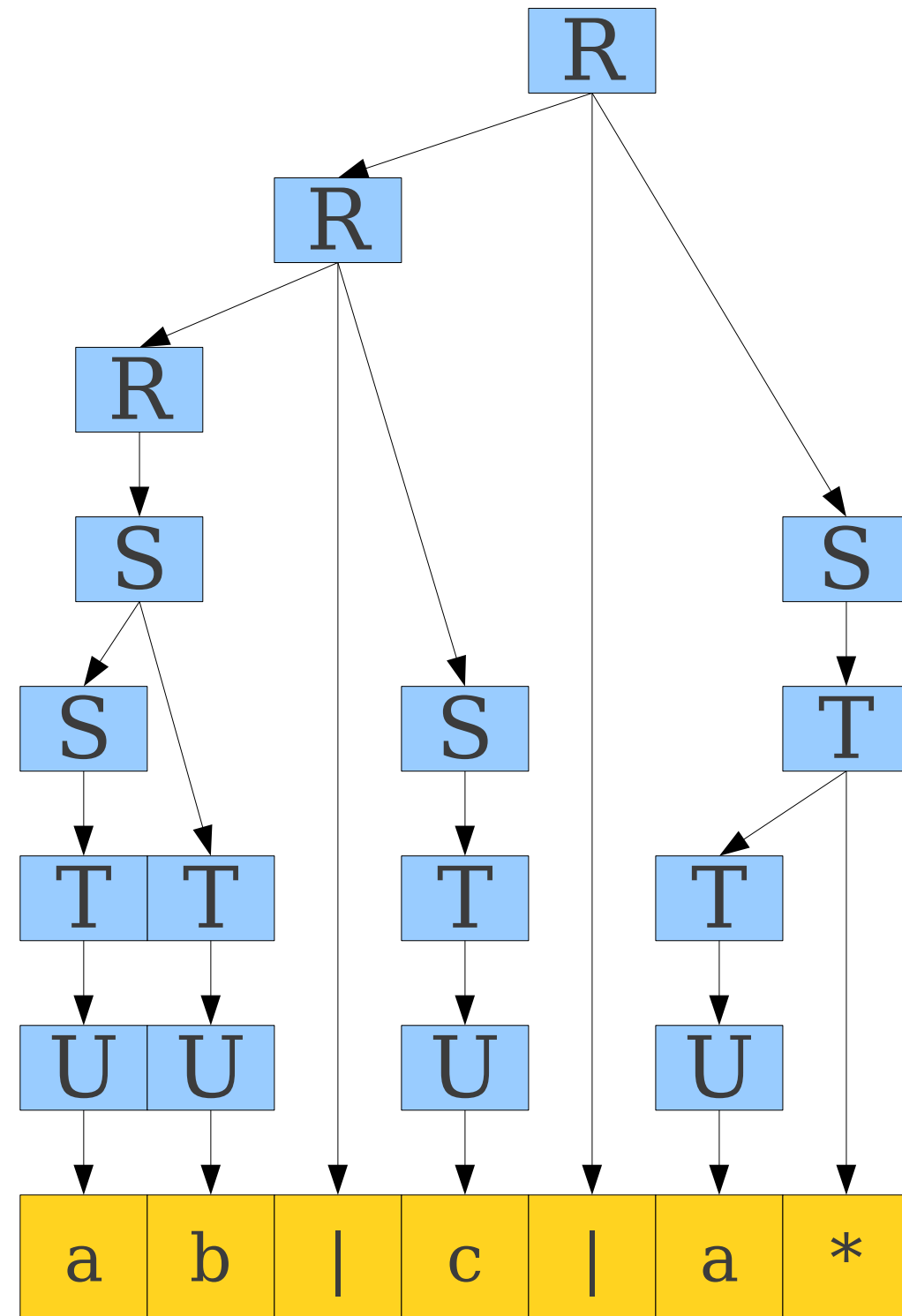
$S \rightarrow T \mid ST$

$T \rightarrow U \mid T^*$

$U \rightarrow a \mid b \mid c \mid \dots$

$U \rightarrow \text{“}\epsilon\text{”}$

$U \rightarrow (R)$



Precedence Declarations

- If we leave the world of pure CFGs, we can often resolve ambiguities through **precedence declarations**.
 - e.g. multiplication has higher precedence than addition, but lower precedence than exponentiation.
- Allows for unambiguous parsing of ambiguous grammars.
- We'll see how this is implemented later on.

The Structure of a Parse Tree

$R \rightarrow S \mid R \mid S$

$S \rightarrow T \mid ST$

$T \rightarrow U \mid T^*$

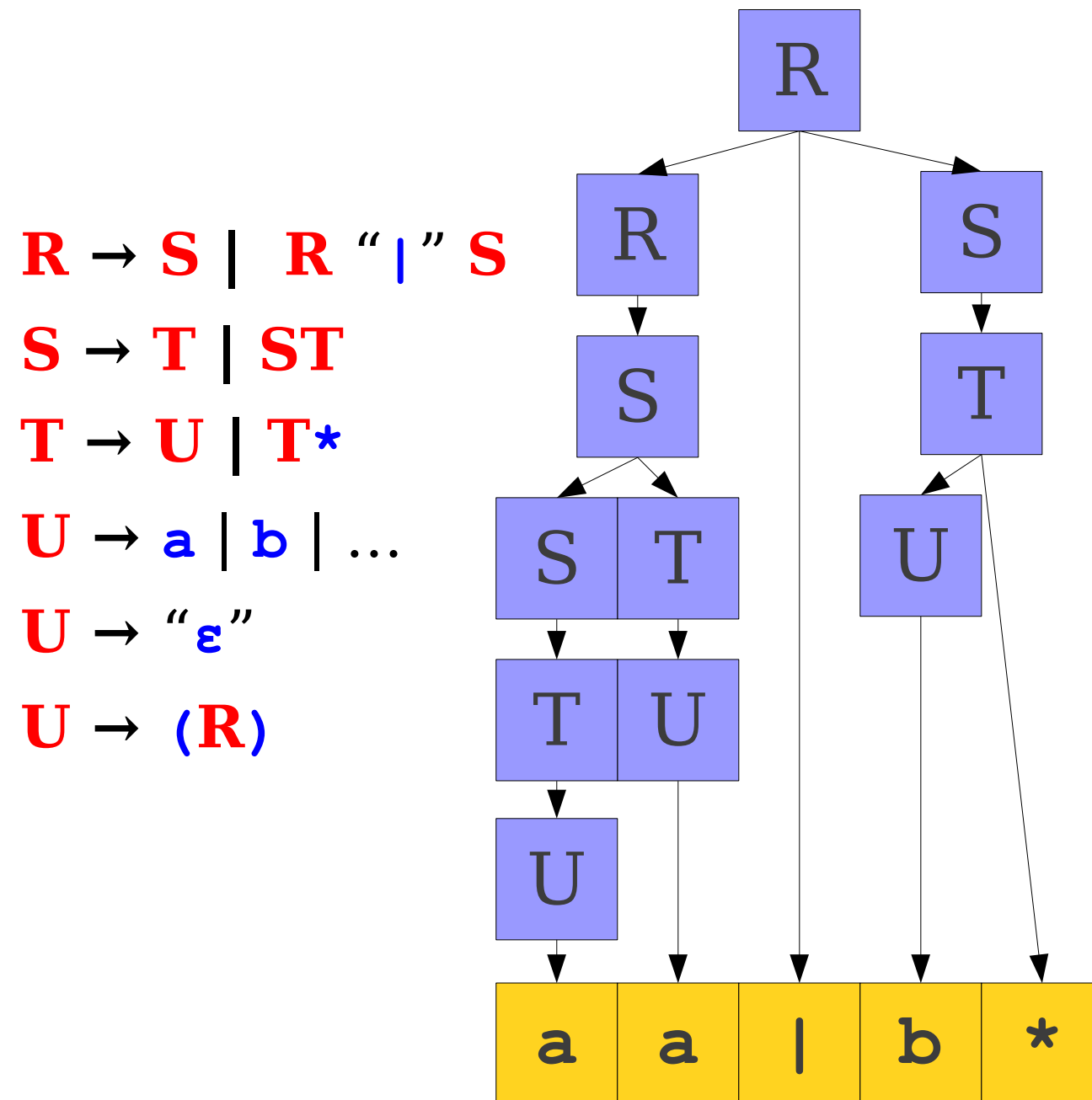
$U \rightarrow a \mid b \mid \dots$

$U \rightarrow \epsilon$

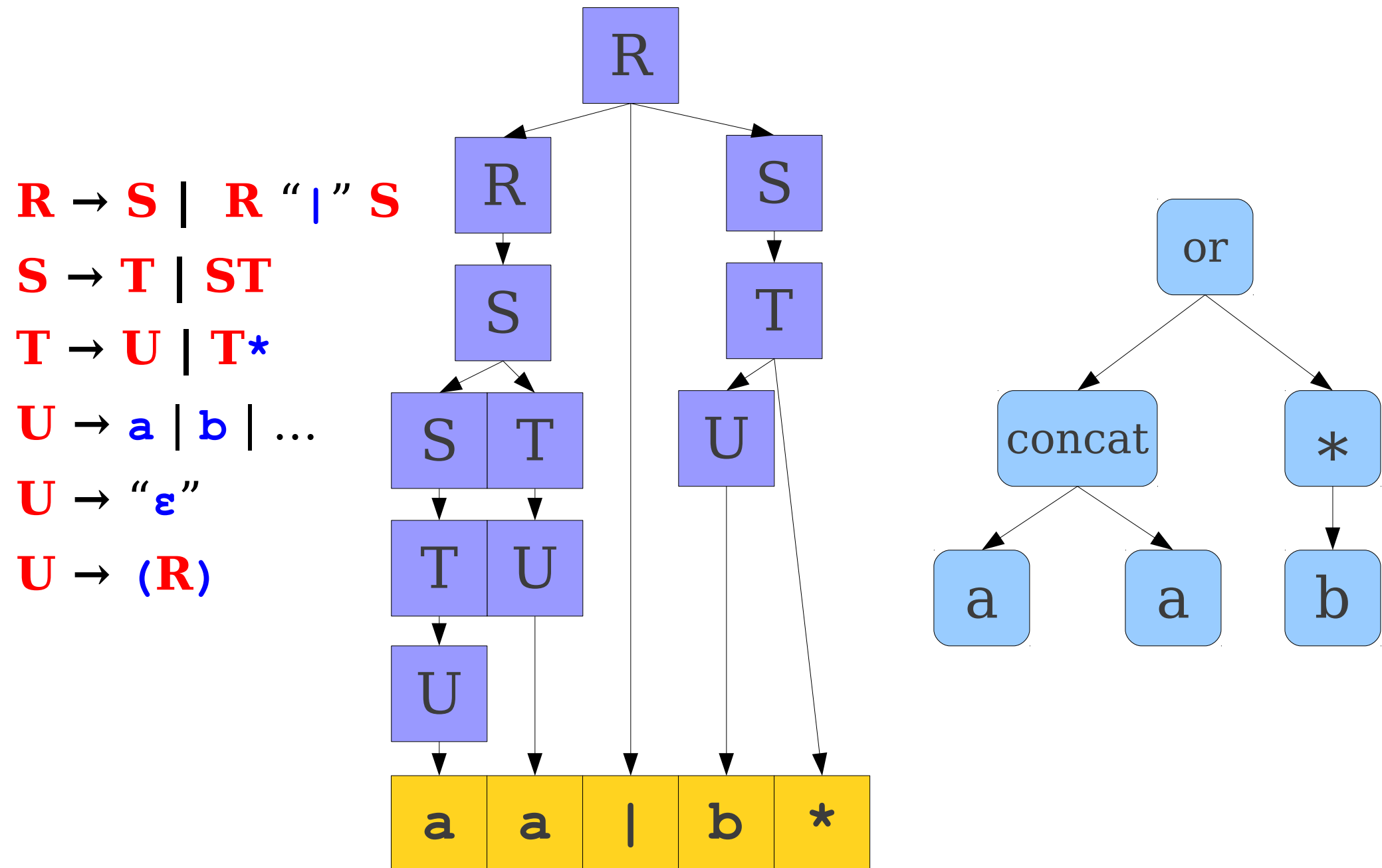
$U \rightarrow (R)$

a	a		b	*
---	---	--	---	---

The Structure of a Parse Tree



The Structure of a Parse Tree



R $\rightarrow$ **S** | **R** “|” **S**

S $\rightarrow$ **T** | **ST**

T $\rightarrow$ **U** | **T***

U $\rightarrow$ **a** | **b** | ...

U $\rightarrow$ “ **ϵ** ”

U $\rightarrow$ (**R**)

a	(	b		c	)
---	---	---	--	---	---

R $\rightarrow$ **S** | **R** “ | ” **S**

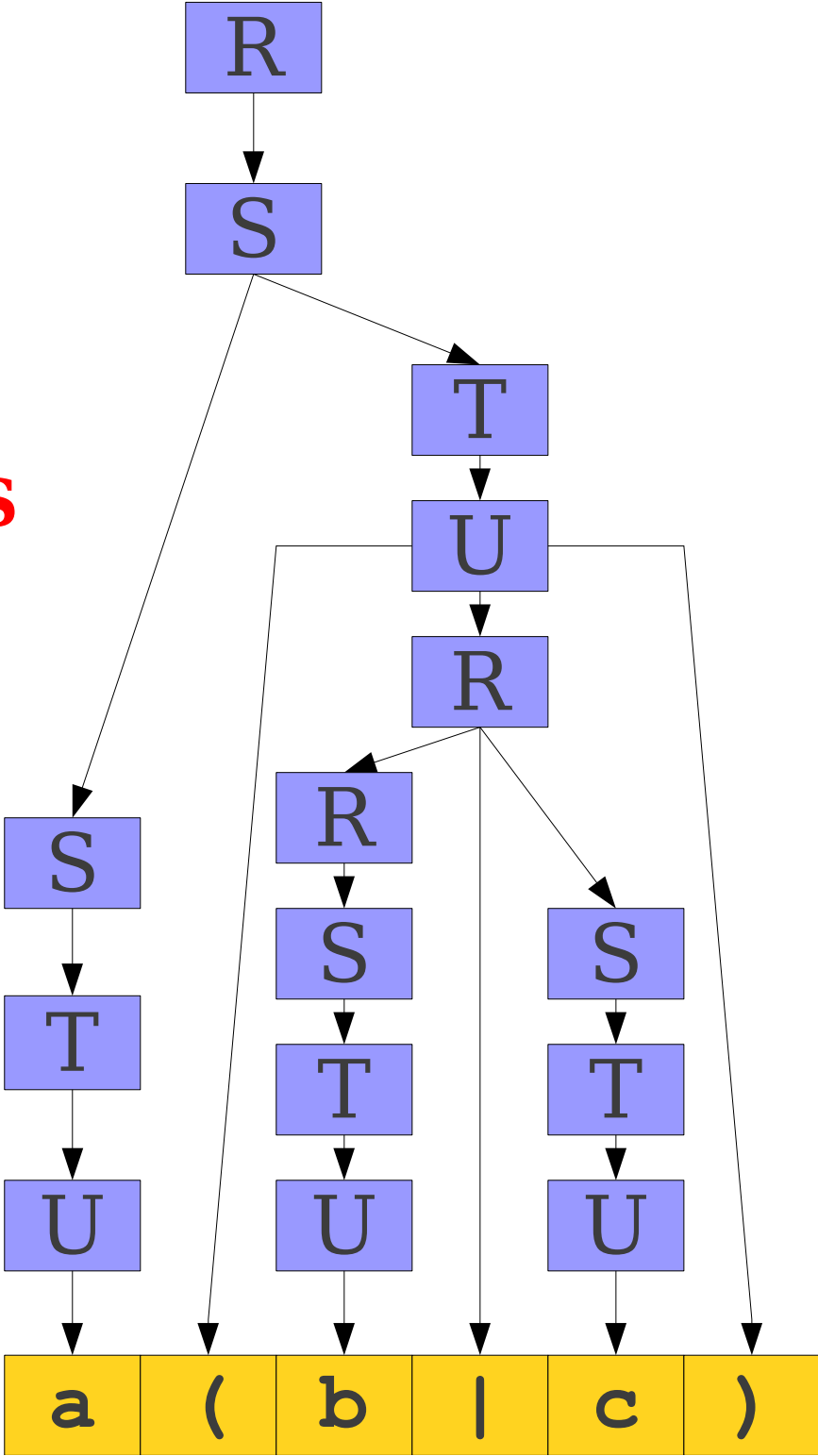
S $\rightarrow$ **T** | **ST**

T → **U** | **T***

U $\rightarrow$ **a** | **b** | ...

U → “ε”

U $\rightarrow$ **(R)**



$R \rightarrow S \mid R \mid S$

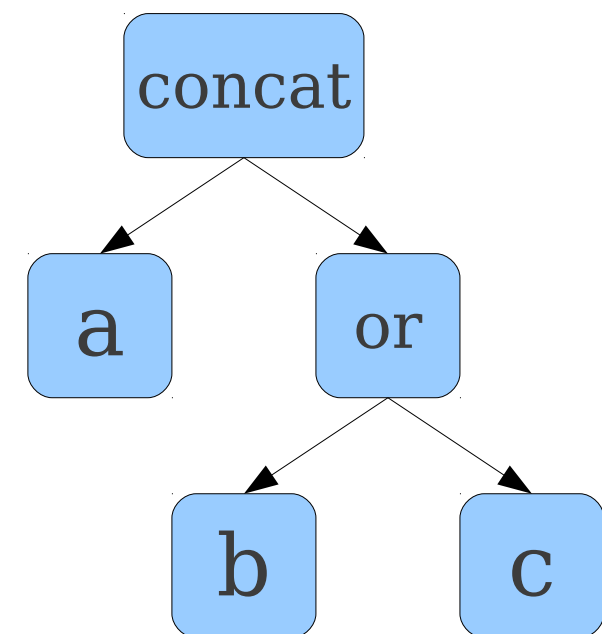
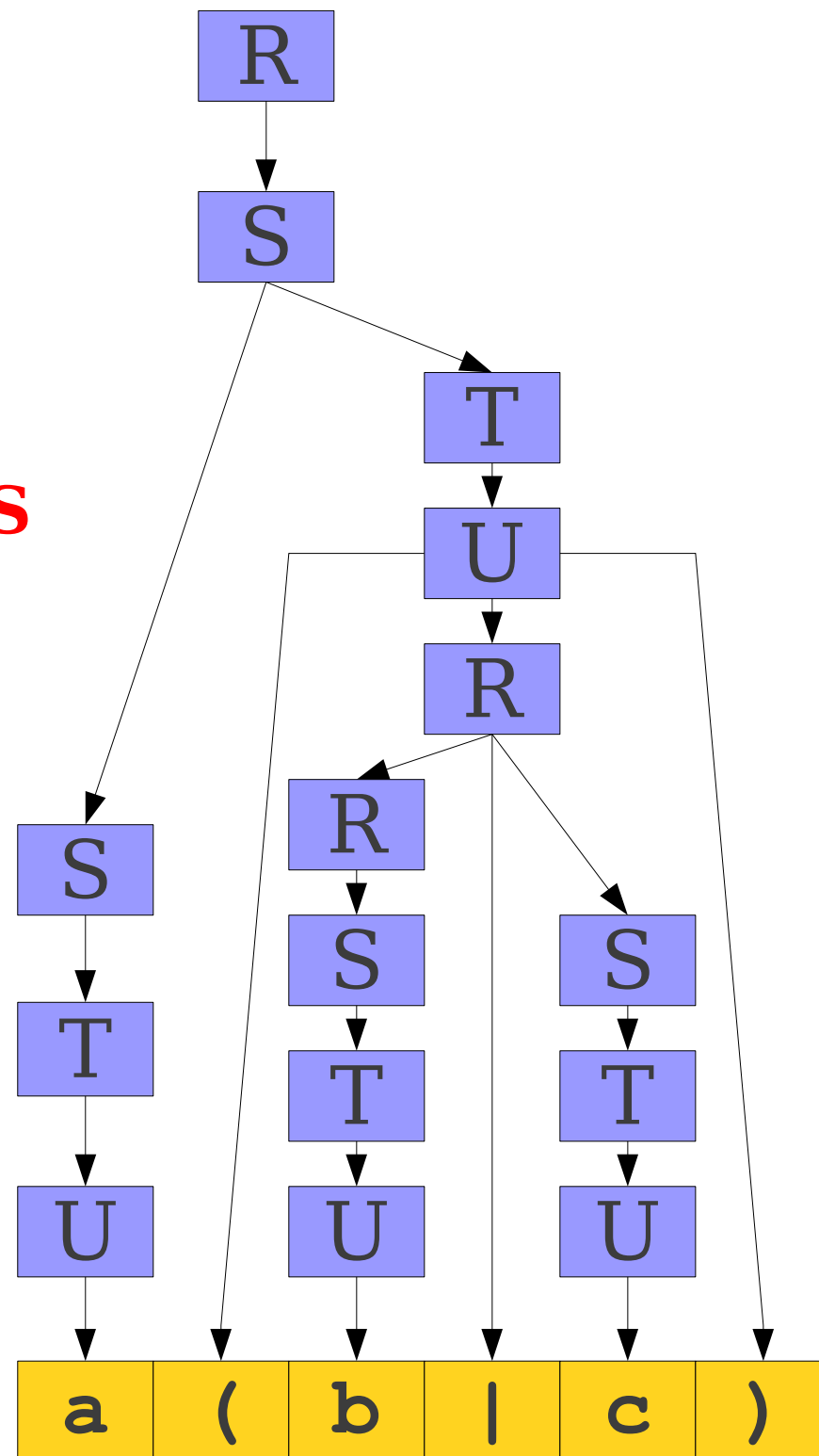
$S \rightarrow T \mid ST$

$T \rightarrow U \mid T^*$

$U \rightarrow a \mid b \mid \dots$

$U \rightarrow \epsilon$

$U \rightarrow (R)$



If-Then-Else

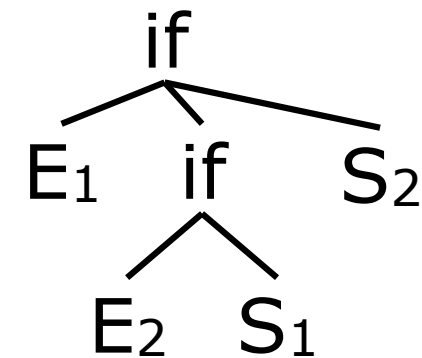
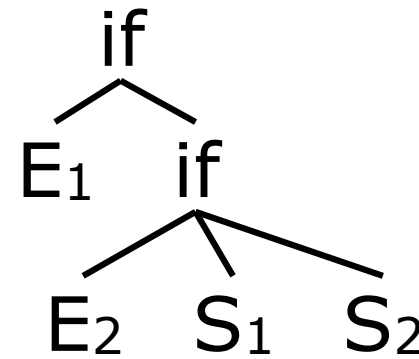
- How do we write a grammar for `if` statements?
- $S \rightarrow \text{if } (E) S$
- $S \rightarrow \text{if } (E) S \text{ else } S$
- $S \rightarrow X = E$
- Is this grammar OK?

No! Ambiguous

- $\text{if } (E_1) \text{ if } (E_2) S_1 \text{ else } S_2$

$S \rightarrow \text{if } (E) S$
$S \rightarrow \text{if } (E) S \text{ else } S$
$S \rightarrow \text{other}$

- $S \rightarrow \text{if } (E) S$
- $\rightarrow \text{if } (E) \text{ if } (E) S \text{ else } S$
- $S \rightarrow \text{if } (E) S \text{ else } S$
- $\rightarrow \text{if } (E) \text{ if } (E) S \text{ else } S$



- *Which “if” is the “else” attached to?*

Grammar for closest-if rule

- Want to rule out: if (E) if (E) S else S
- Problem: unmatched if may not occur as the “then” (consequent) clause of a containing “if”

statement $\rightarrow$ matched | unmatched

matched $\rightarrow$ if (E) matched else matched | other

unmatched $\rightarrow$ if (E) statement |

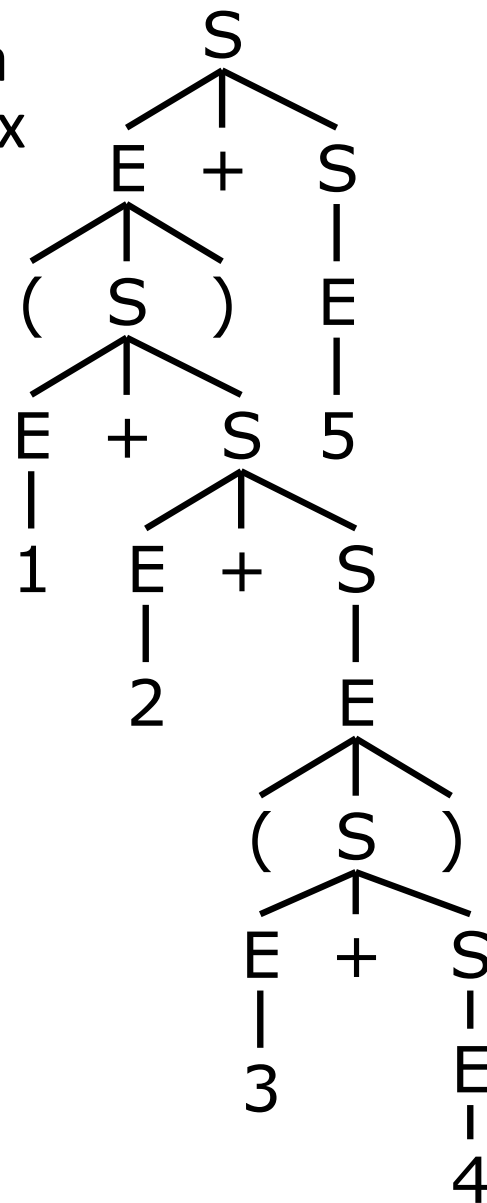
if (E) matched else unmatched

Abstract Syntax Trees (ASTs)

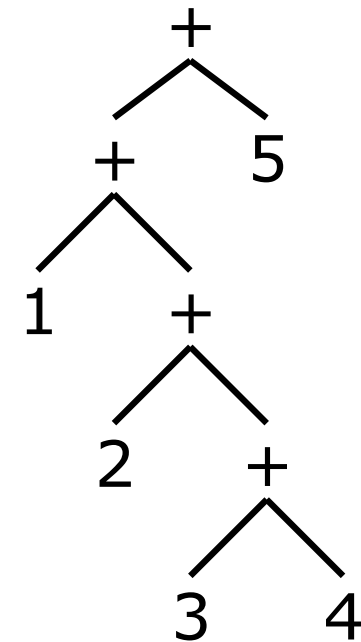
- A parse tree is a **concrete syntax tree**; it shows exactly how the text was derived.
- A more useful structure is an **abstract syntax tree**, which retains only the essential structure of the input.

Parse Tree vs. AST

- Parse Tree, aka concrete syntax



Abstract Syntax Tree



Discards/abstracts unnneeded information

Summary

- Syntax analysis (**parsing**) extracts the structure from the tokens produced by the scanner.
- Languages are usually specified by **context-free grammars (CFGs)**.
- A **parse tree** shows how a string can be **derived** from a grammar.
- A grammar is **ambiguous** if it can derive the same string multiple ways.
- There is no algorithm for eliminating ambiguity; it must be done by hand.
- **Abstract syntax trees (ASTs)** contain an abstract representation of a program's syntax.