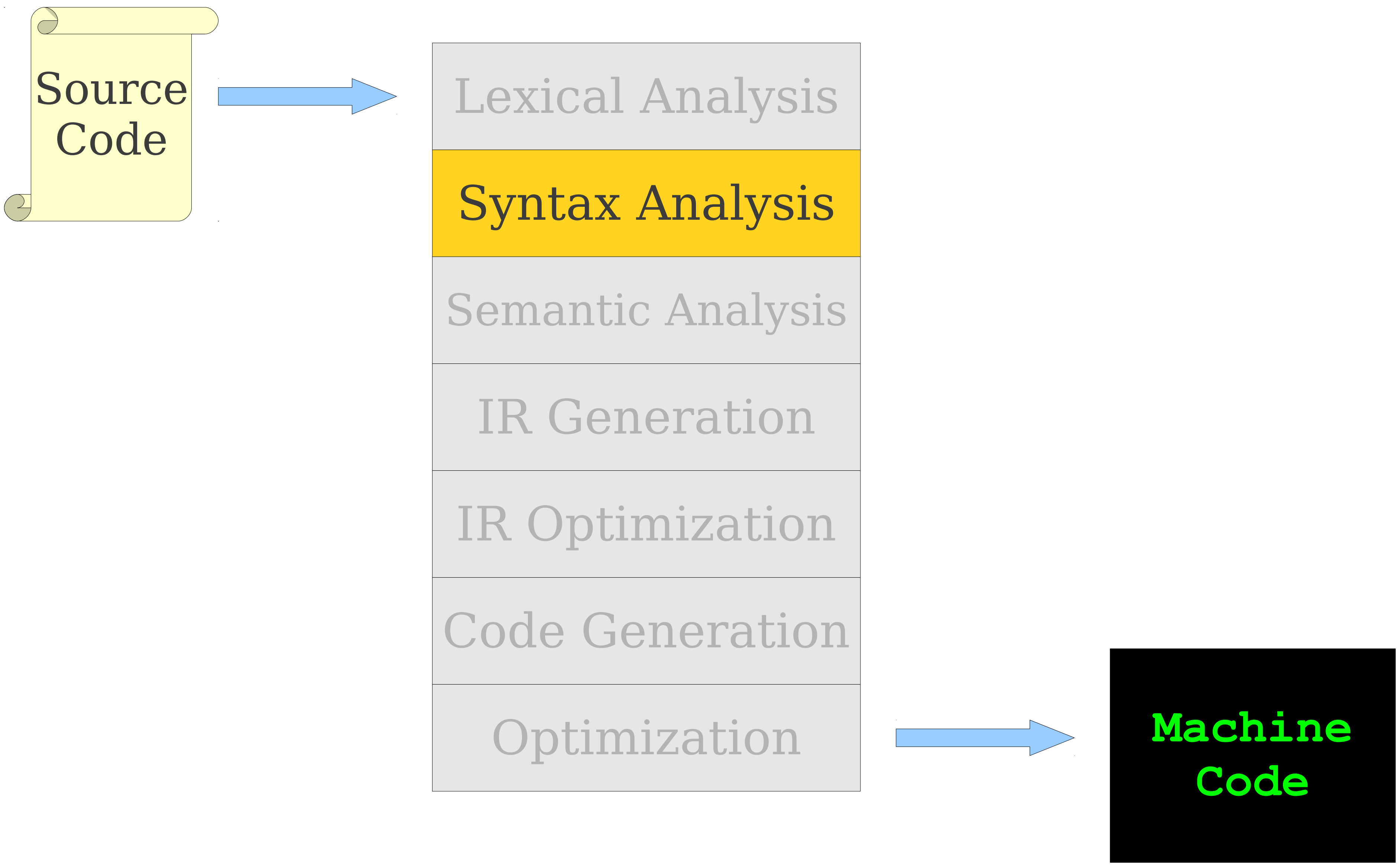
Syntax Analysis

- Introduction to parsing

Where We Are

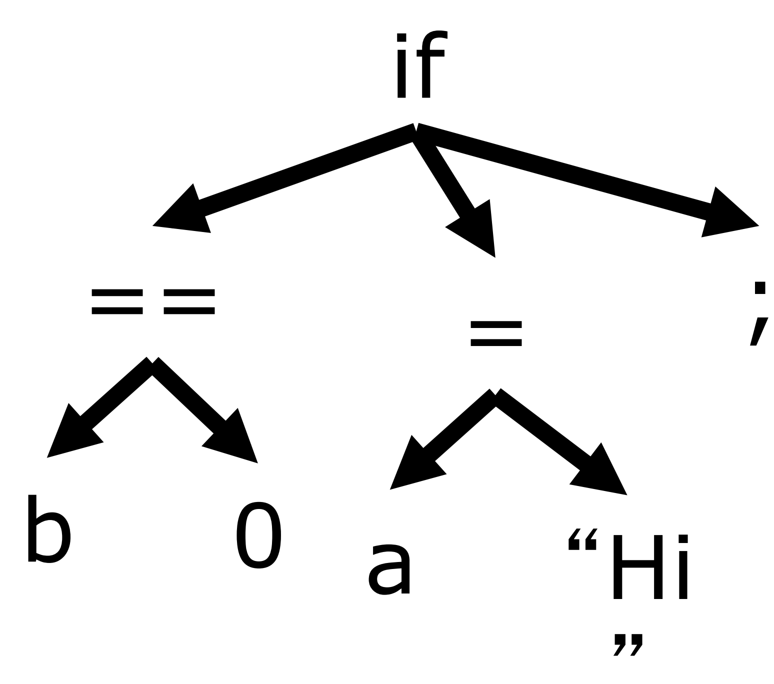

Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Overview of Syntax Analysis

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

Source code: if (b==0) a = “Hi”;

AST:



- 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

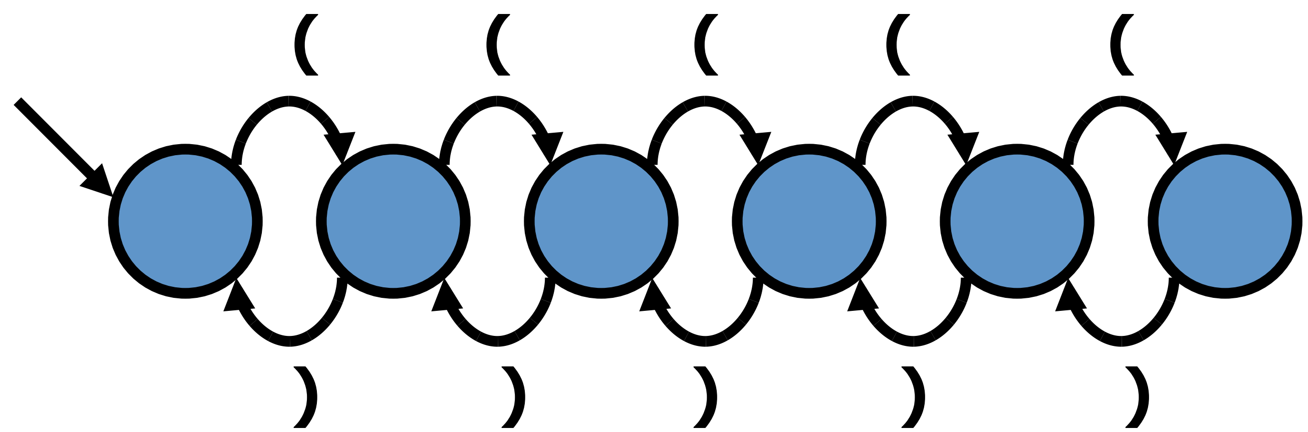
- Today: Formalism for syntax analysis
  - Grammars
  - Derivation
  - Ambiguity

Specify Language Syntax

- First problem: how to describe language syntax ?
  - Lexer: can describe tokens using __ ?
  - Regular expressions: easy to implement, efficient (DFA)
  - Can we use regular expressions 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)

- Example: A specification of the balanced-parenthesis language:
  - $S \rightarrow (S) S$
  - $S \rightarrow \varepsilon$
- The definition is recursive
- If a grammar accepts a string, there is a **derivation** of that string using the **productions** of the grammar
  - S => (S) ε => ((S) S) ε => ((ε) ε) ε => (())

CFG Terminology

- **Terminals**
  - Token or ε
- **Non-terminals**
  - variables
- **Start symbol**
  - Begins the derivation
- **Productions**
  - **replacement rules :** Specify how non-terminals may be expanded to form strings
  - LHS: single non-terminal, RHS: string of terminals (including ε) or non-terminals

> - $S \rightarrow (S) S$
> - $S \rightarrow \varepsilon$

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

A Notational Shorthand

$E \rightarrow$ `int`

$E \rightarrow E\ Op\ E$

$E \rightarrow$ `(`E`)`

$Op \rightarrow$ `+`

$Op \rightarrow$ `-`

$Op \rightarrow$ `*`

$Op \rightarrow$ `/`

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

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

- Vertical bar | 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
         |  …
```

Scanner vs. Parser

Language is a set of **strings**

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

Scanning:

- the **strings** are __?
  - source programs
- the **alphabet** is __?
  - the ASCII
- Formal Language is __?
  - Regular expression
- Machine to recognize the language?
  - Finite Automata

Parsing:

- The **strings** are__?
  - Sequence of token
- the **alphabet** is ___?
  - set of token-types returned by the lexical analyzer
- Formal Language is __?
  - Context Free Gramma
- Machine to recognize the language?
  - Pushdown automata => parsing algorithms for approximation

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. $\boldsymbol{\alpha}$, $\boldsymbol{\gamma}$, $\boldsymbol{\omega}$

Examples

- We might write an arbitrary production as

$$\mathbf{A} \to \boldsymbol{\omega}$$

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

$$\mathbf{A}\mathtt{t}$$

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

$$\mathbf{B} \to \boldsymbol{\alpha}\mathbf{A}\mathtt{t}\boldsymbol{\omega}$$

Derivation

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 $\alpha A \omega$ **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

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

- 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
  - abcbcba

- $S \rightarrow aXa$
- $X \rightarrow \varepsilon \mid bY$
- $Y \rightarrow \varepsilon \mid cXc$

Leftmost Derivations

```
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
      | EXPR = EXPR
      | …
```

Productions

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.

E → int | E Op E | (E)

Op → + | - | * | /

Related Derivations

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.

- Derivation: also a process of constructing a parse tree

Parse Trees

E → **`int`** | **E Op E** | **`(`E`)`**

Op → **`+`** | **`-`** | **`*`** | **`/`**

E

Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

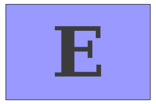
E

E

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

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

Parse Trees



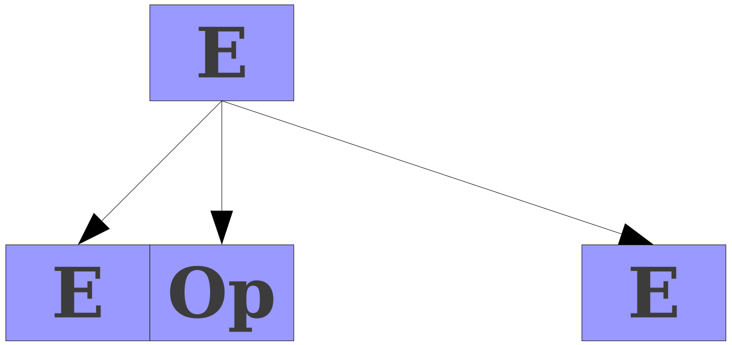
E

$\Rightarrow E\ Op\ E$

Parse Trees

E → **int** | **E Op E** | **(E)**

Op → **+** | **-** | ***** | **/**



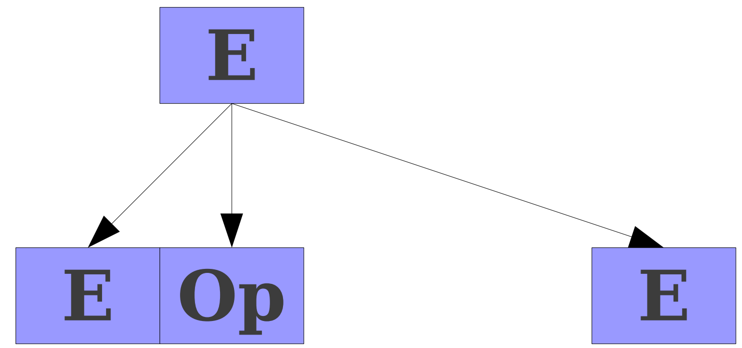
E

⇒ **E Op E**

Parse Trees

E → `int` | **E Op E** | `(`**E**

Op → `+` | `-` | `*` | `/`



E

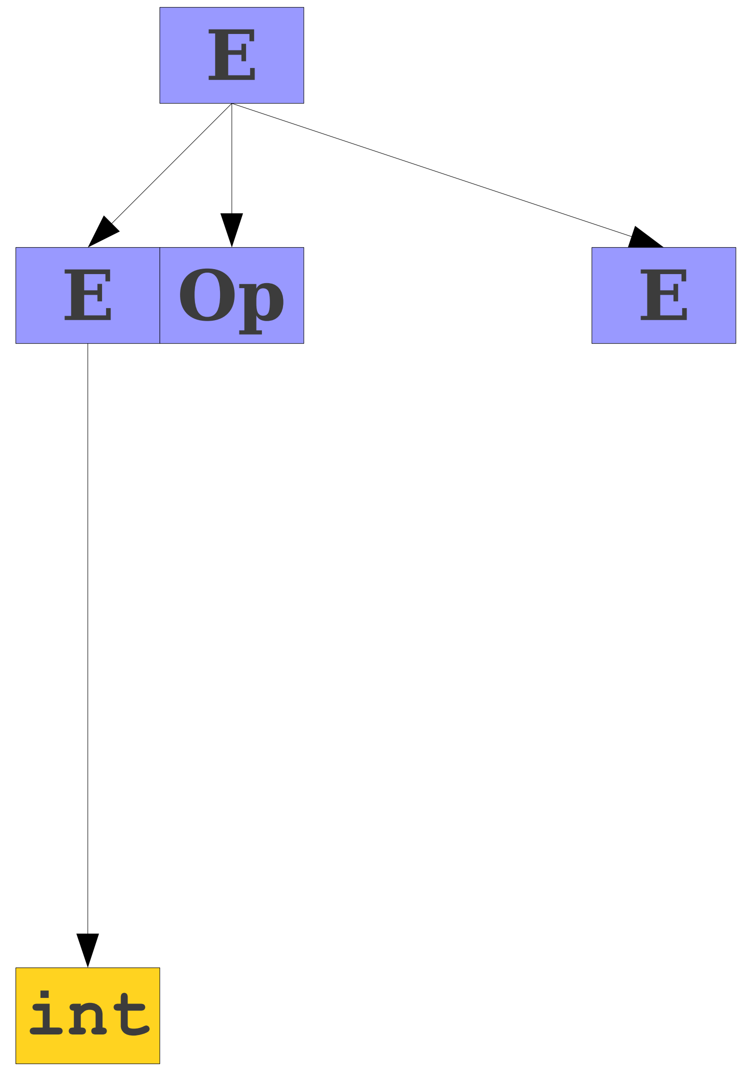
⇒ **E Op E**

⇒ `int` **Op E**

Parse Trees

E → **`int`** | **E Op E** | **`(E)`**

Op → **`+`** | **`-`** | **`*`** | **`/`**



E

⇒ **E Op E**

⇒ **`int`** **Op E**

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

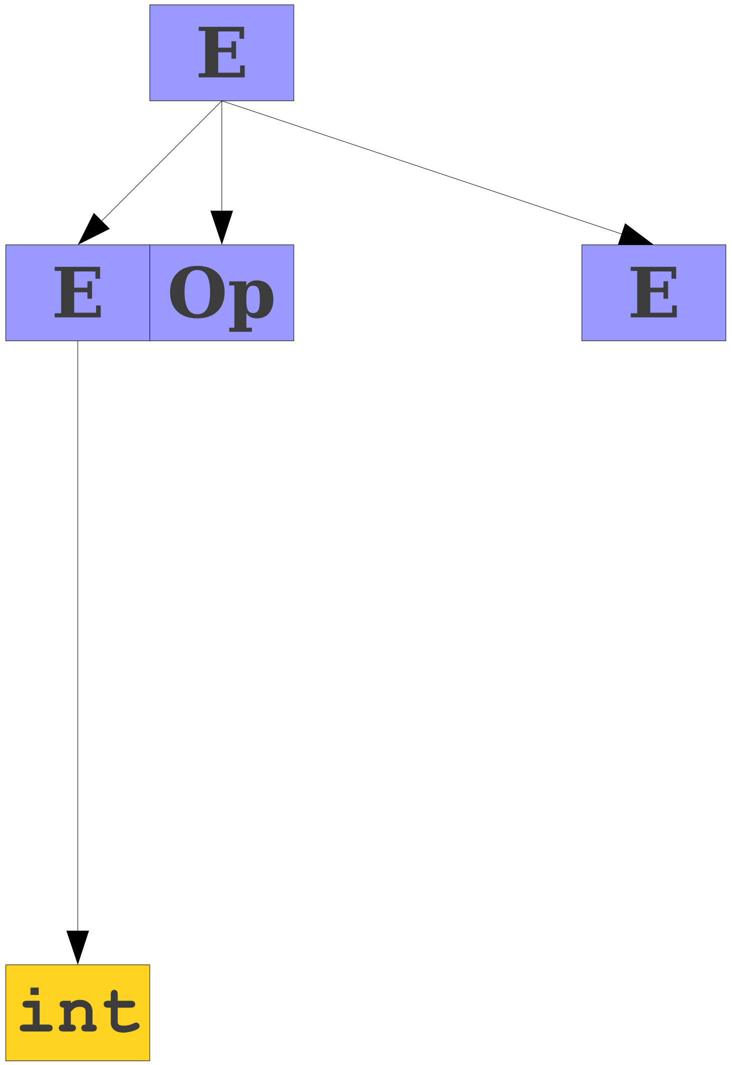
Parse Trees

E

⇒ **E Op E**

⇒ `int` **Op E**

⇒ `int *` **E**



Parse Trees

E → `int` | **E Op E** | `(E)`

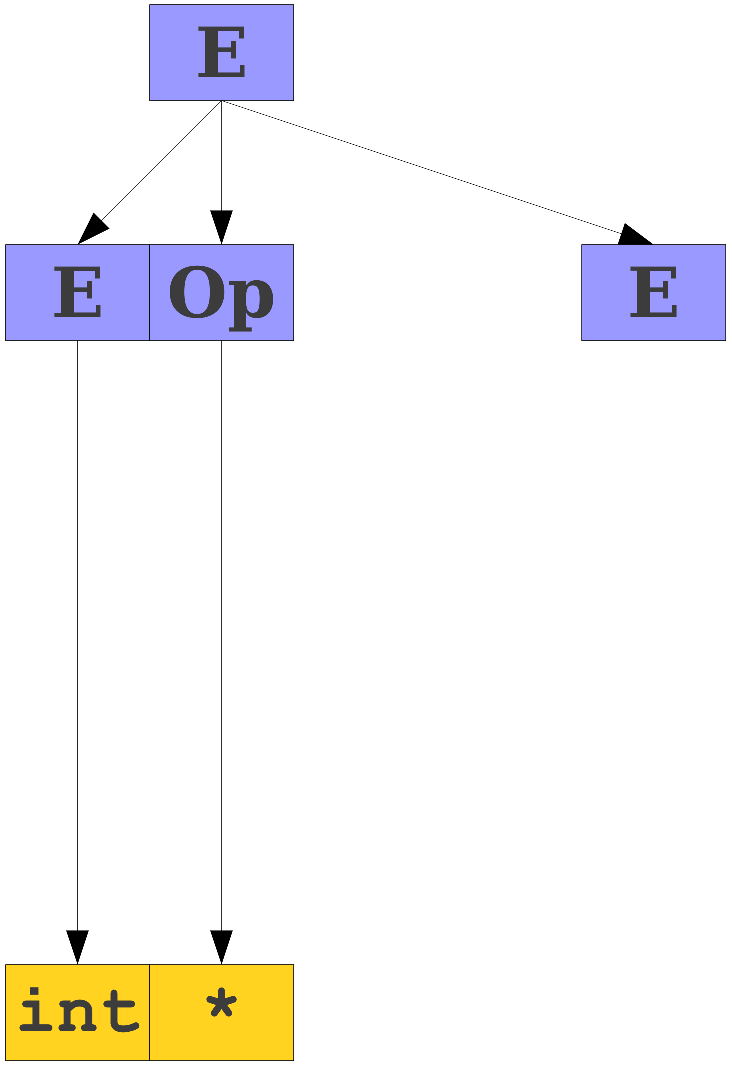
Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ `int` **Op E**

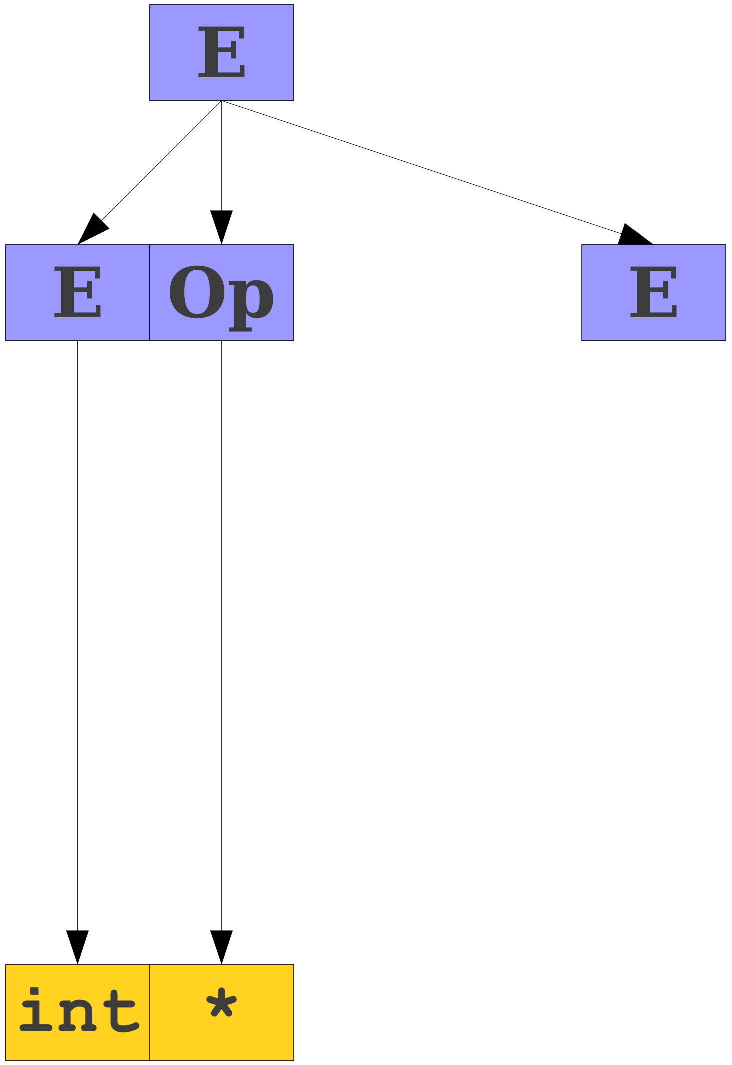
⇒ `int * ` **E**



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`



E

⇒ **E Op E**

⇒ `int` **Op E**

⇒ `int *` **E**

⇒ `int * (`**E**`)`

Parse Trees

E → int | E Op E | (E)

Op → + | - | * | /

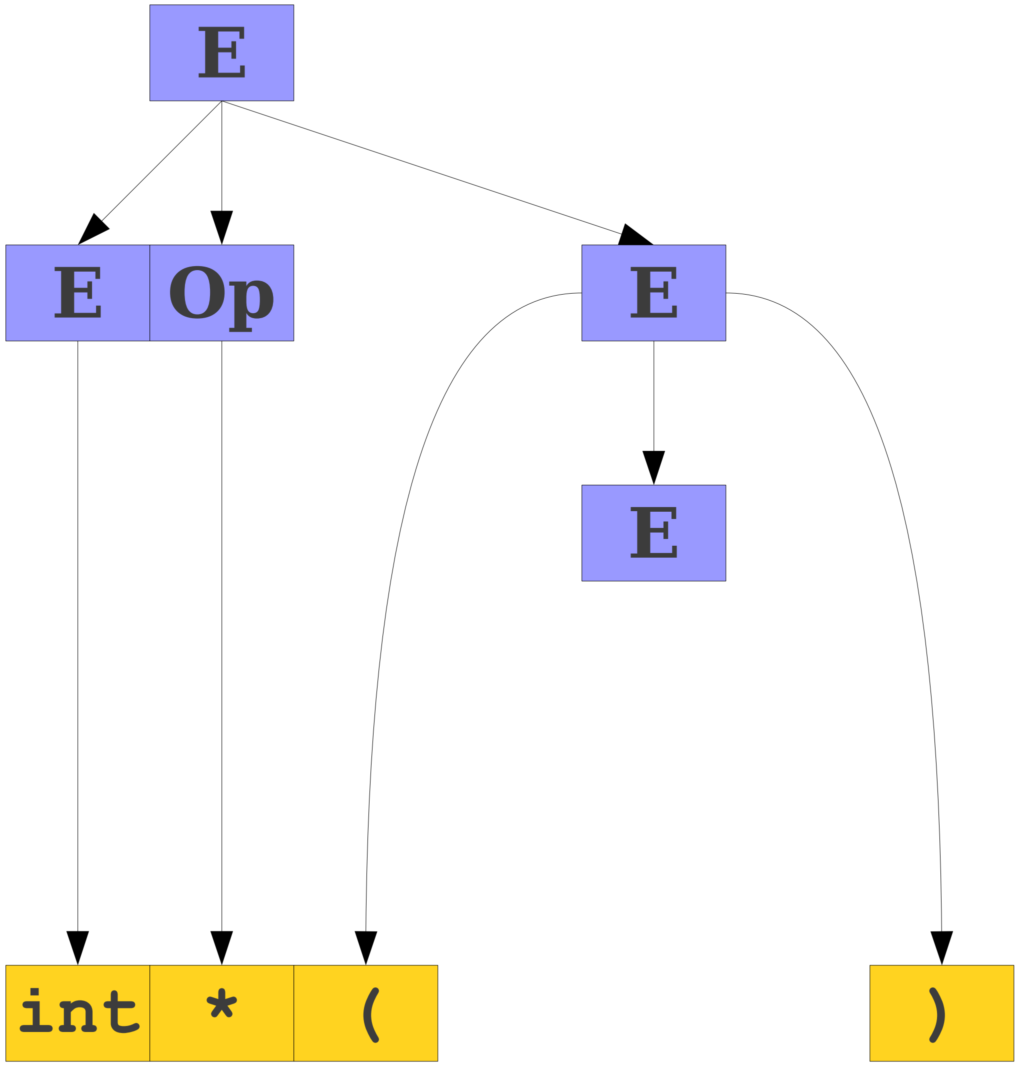
E

⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)



Parse Trees

E → int | E Op E | (E)

Op → + | - | * | /

E

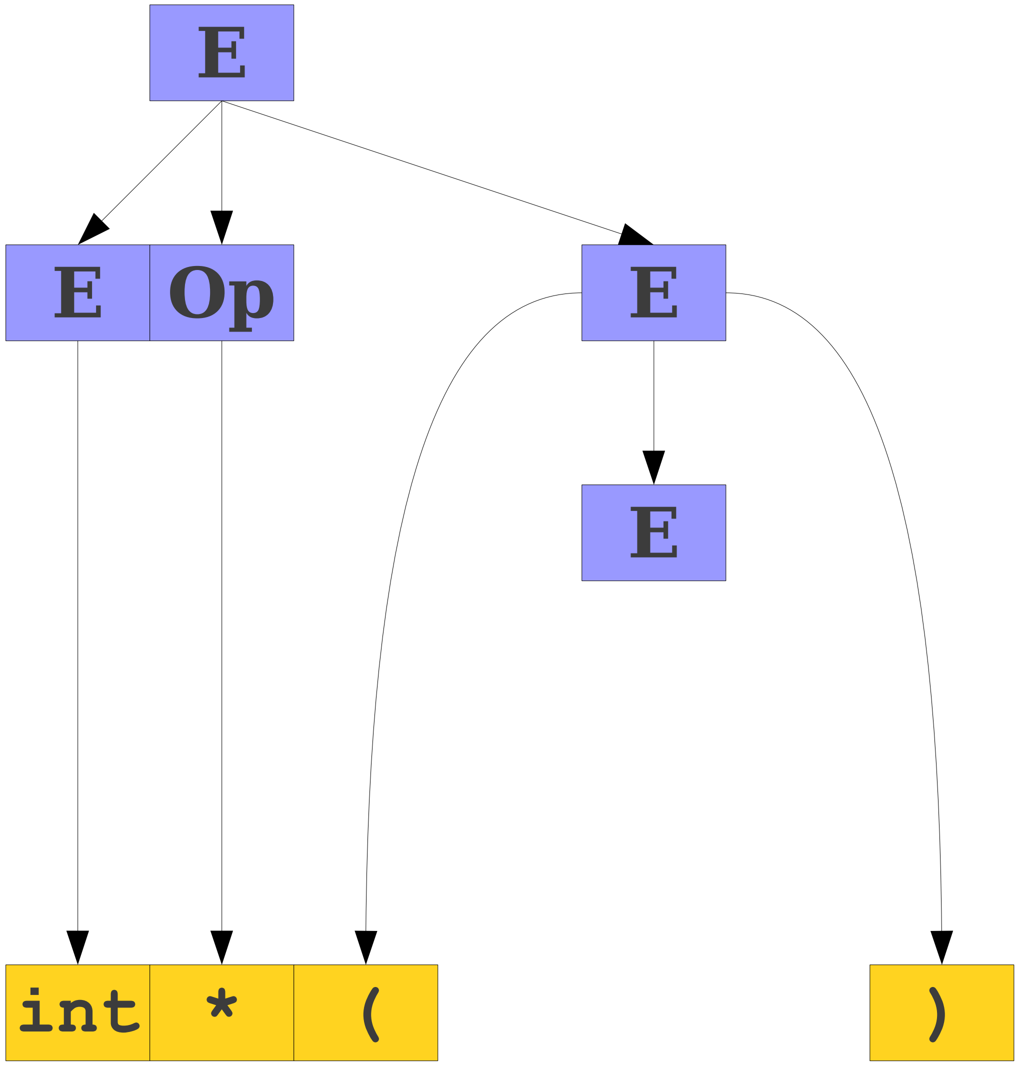
⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)



Parse Trees

E → int | E Op E | (E)

Op → + | - | * | /

E

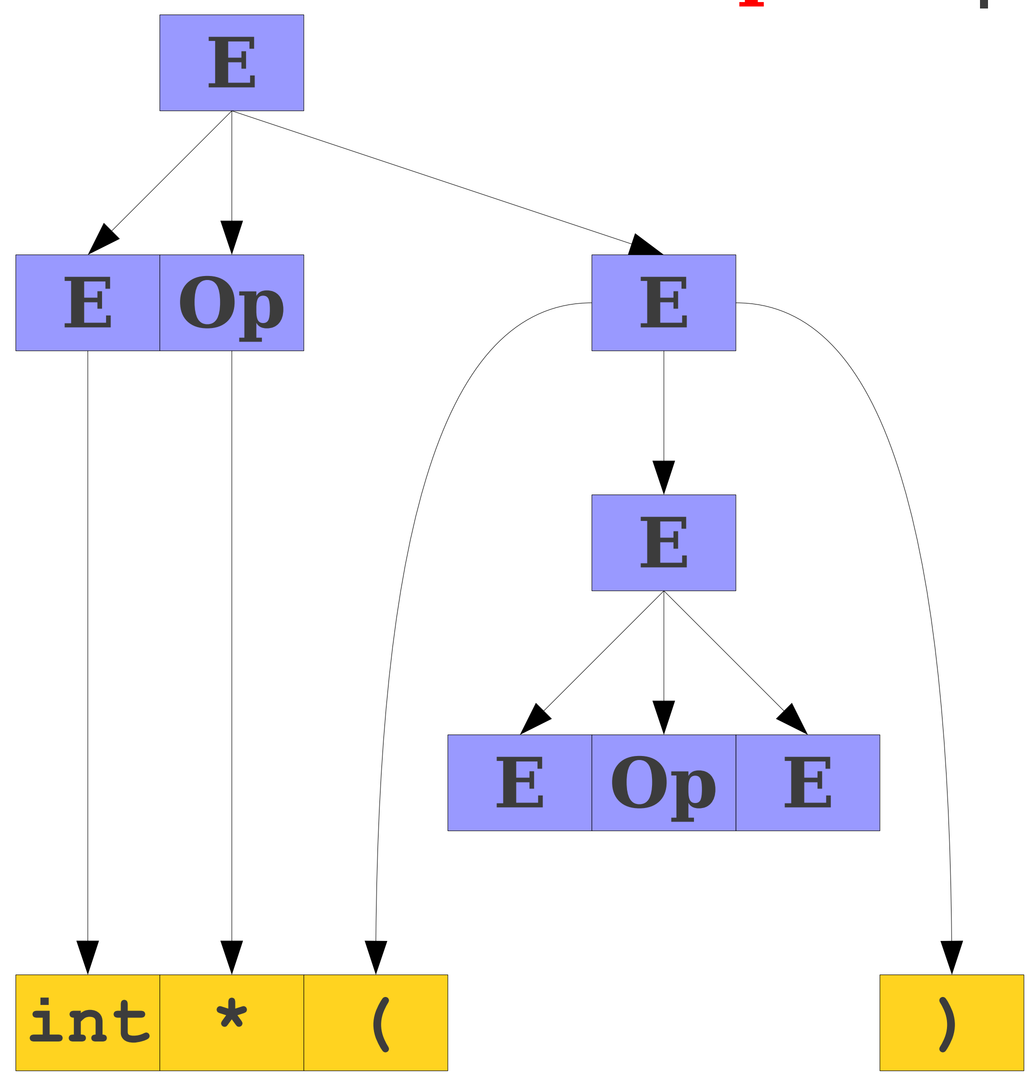
⇒ E Op E

⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)



Parse Trees

E → int | E Op E | (E)

Op → + | - | * | /

E

⇒ E Op E

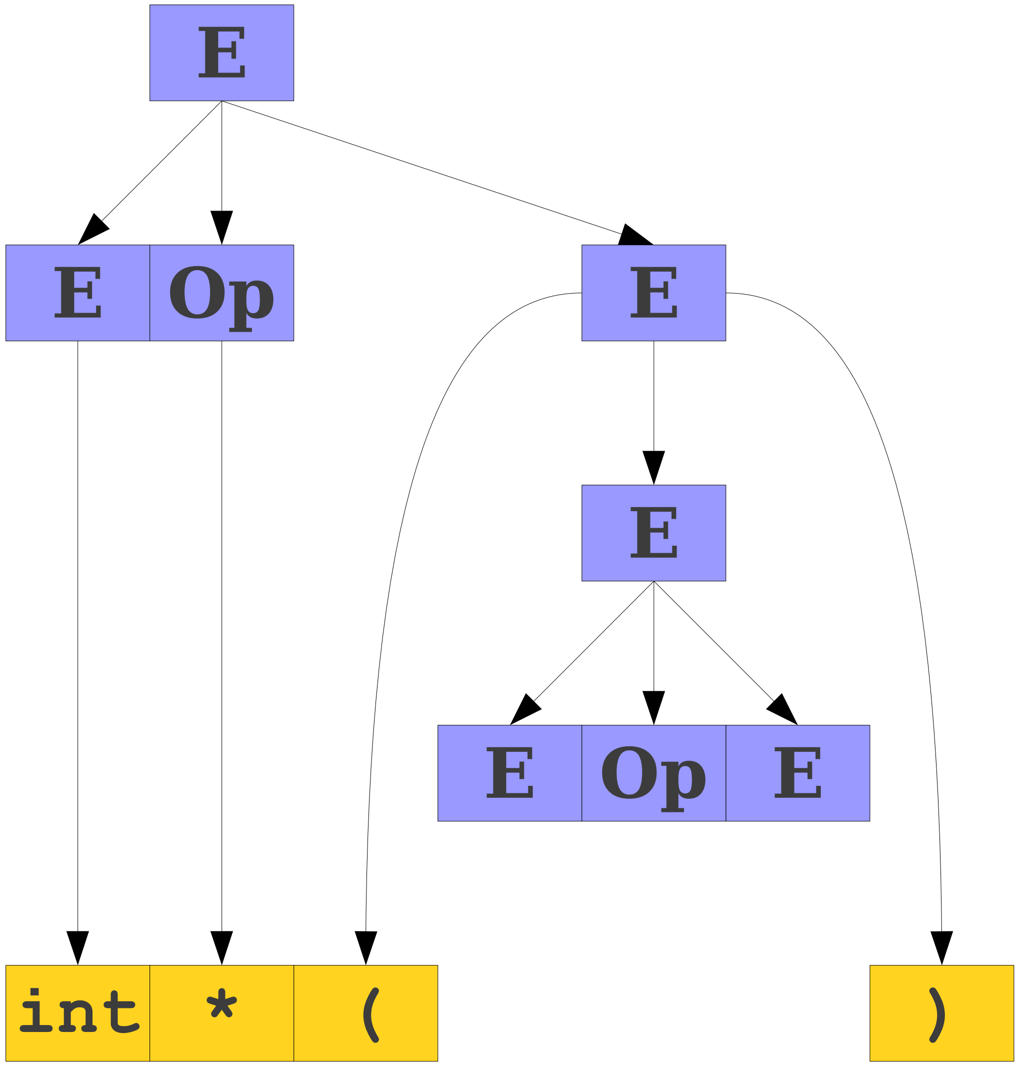
⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)



Parse Trees

E → int | E Op E | (E)

Op → + | - | * | /

E

⇒ E Op E

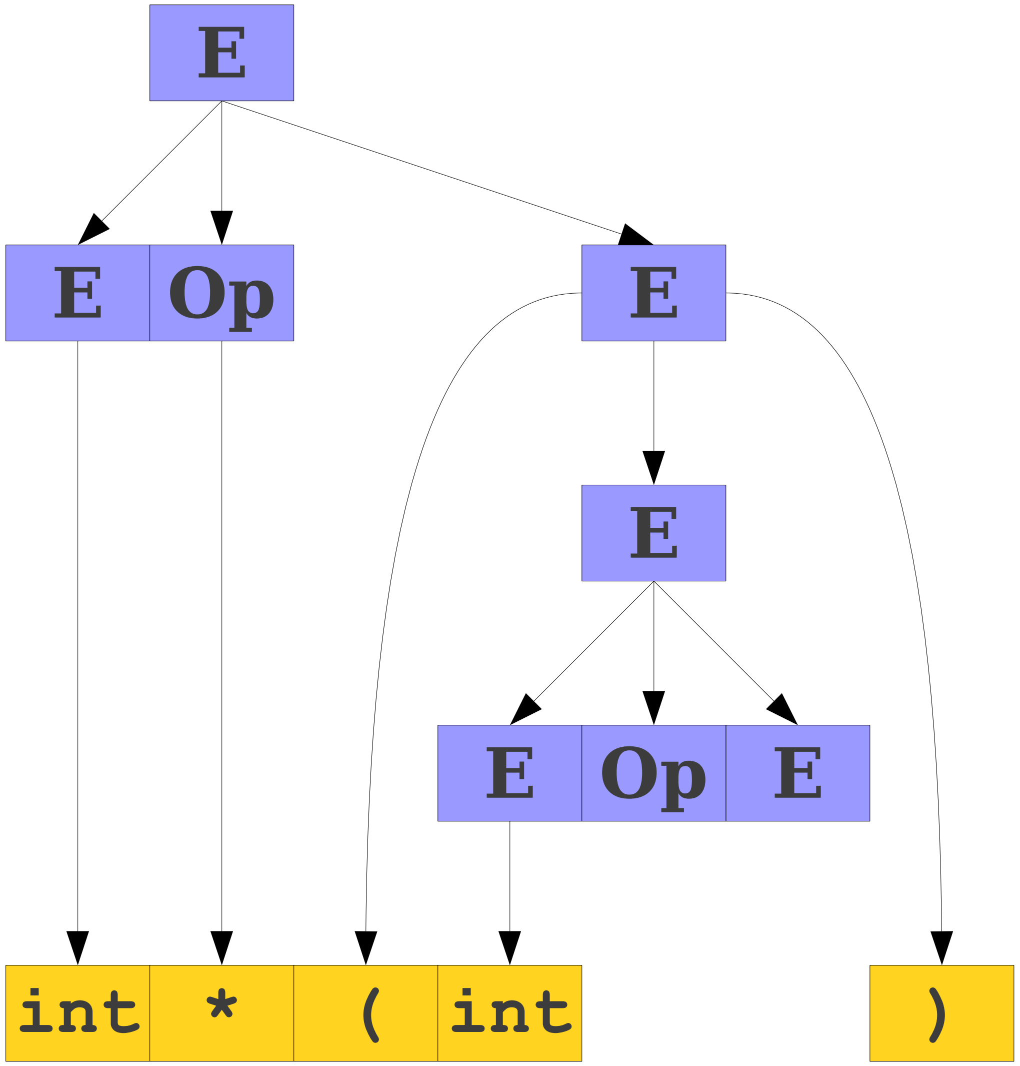
⇒ int Op E

⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ `int` **Op E**

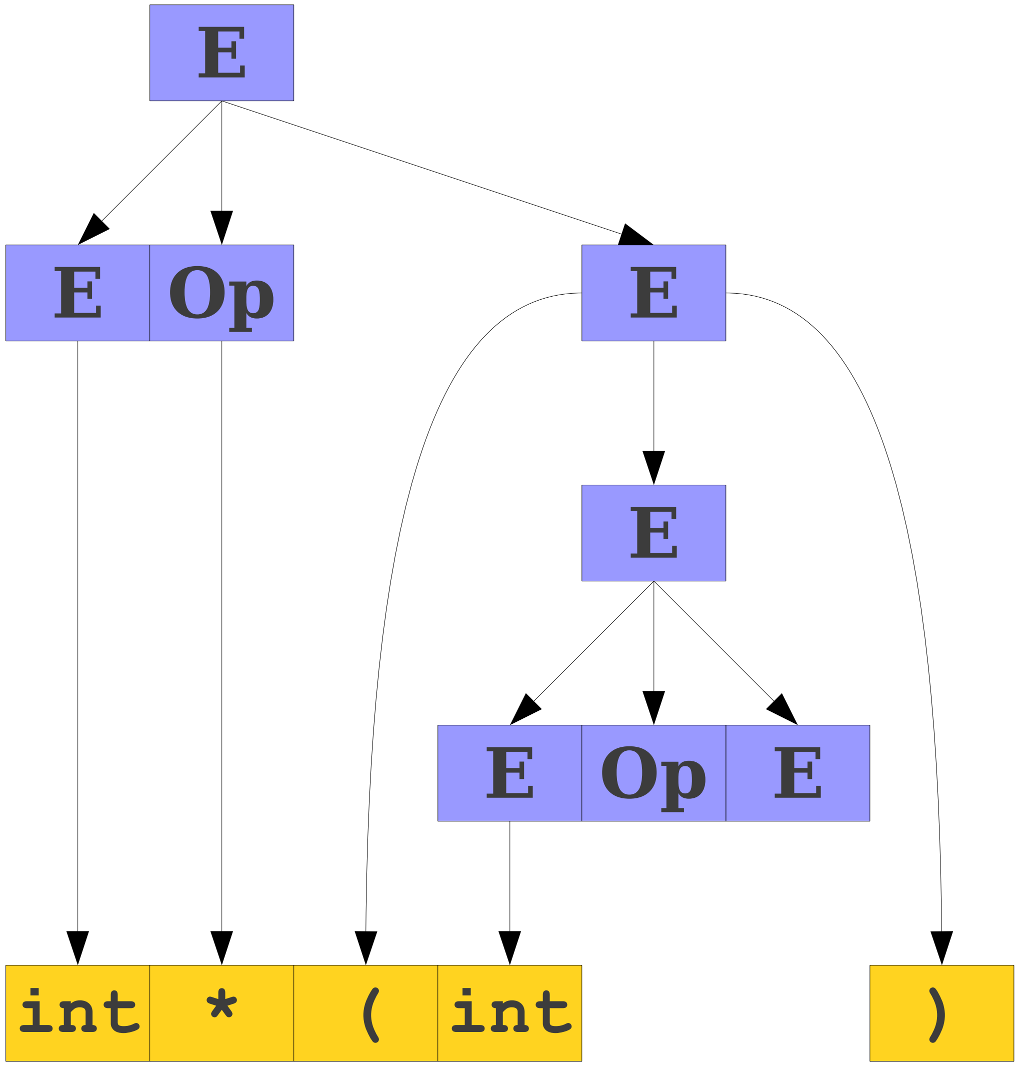
⇒ `int *` **E**

⇒ `int * (`**E**`)`

⇒ `int * (`**E Op E**`)`

⇒ `int * (int` **Op E**`)`

⇒ `int * (int +` **E**`)`



Parse Trees

E → **int** | **E Op E** | **(E)**

Op → **+** | **-** | ***** | **/**

E

⇒ E Op E

⇒ int Op E

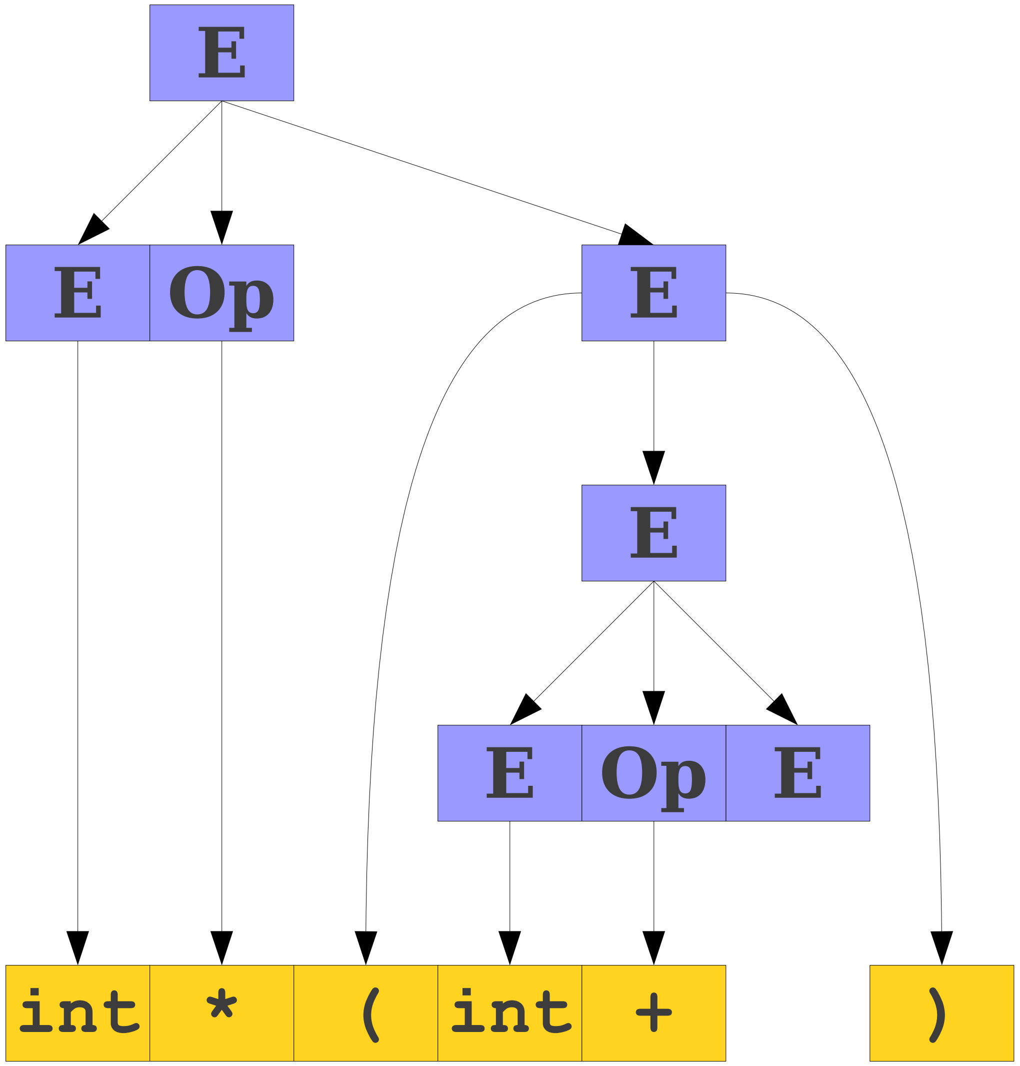
⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)



Parse Trees

E → int | **E Op E** | (**E**)

Op → + | - | * | /

E

⇒ **E Op E**

⇒ int **Op E**

⇒ int * **E**

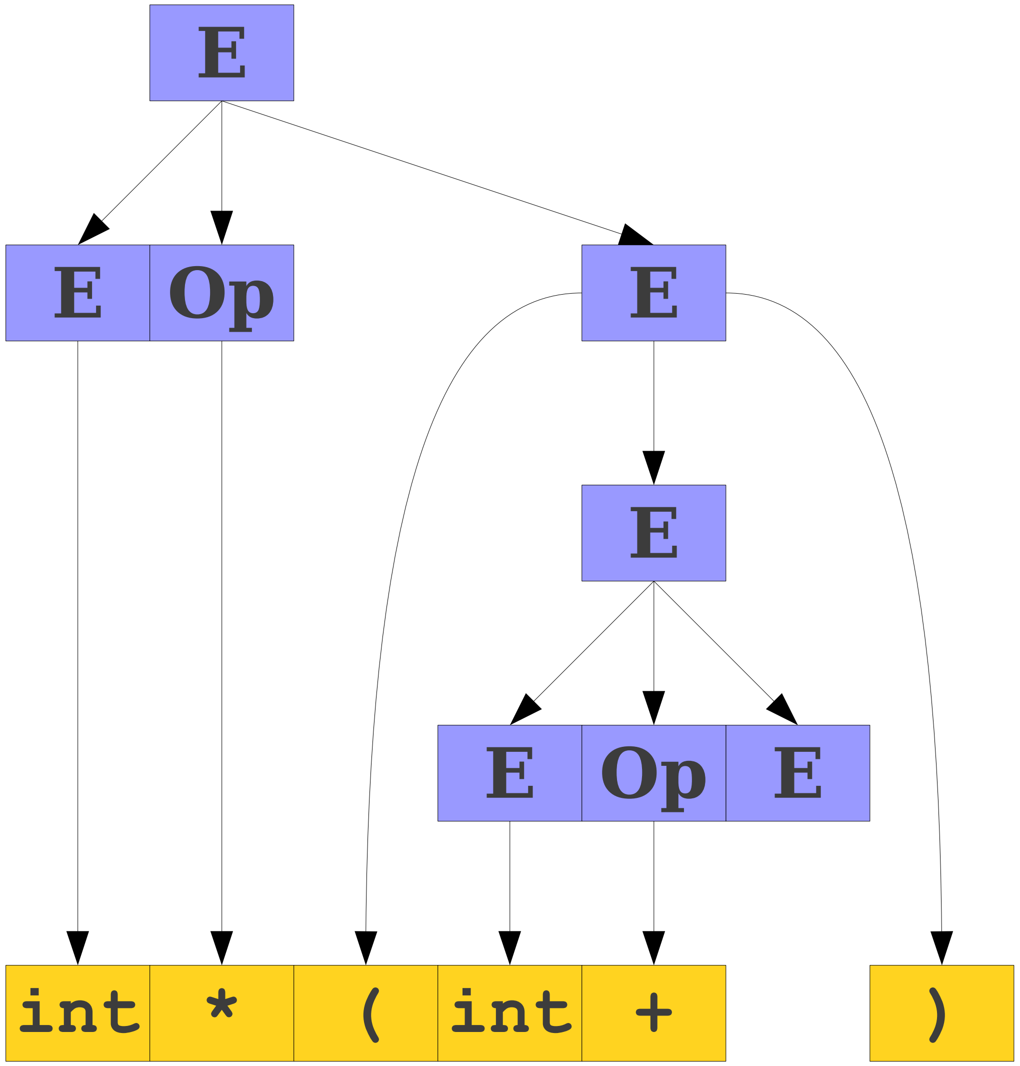
⇒ int * (**E**)

⇒ int * (**E Op E**)

⇒ int * (int **Op E**)

⇒ int * (int + **E**)

⇒ int * (int + int)



Parse Trees

E → int | E Op E | (E)

Op → + | - | * | /

E

⇒ E Op E

⇒ int Op E

⇒ int * E

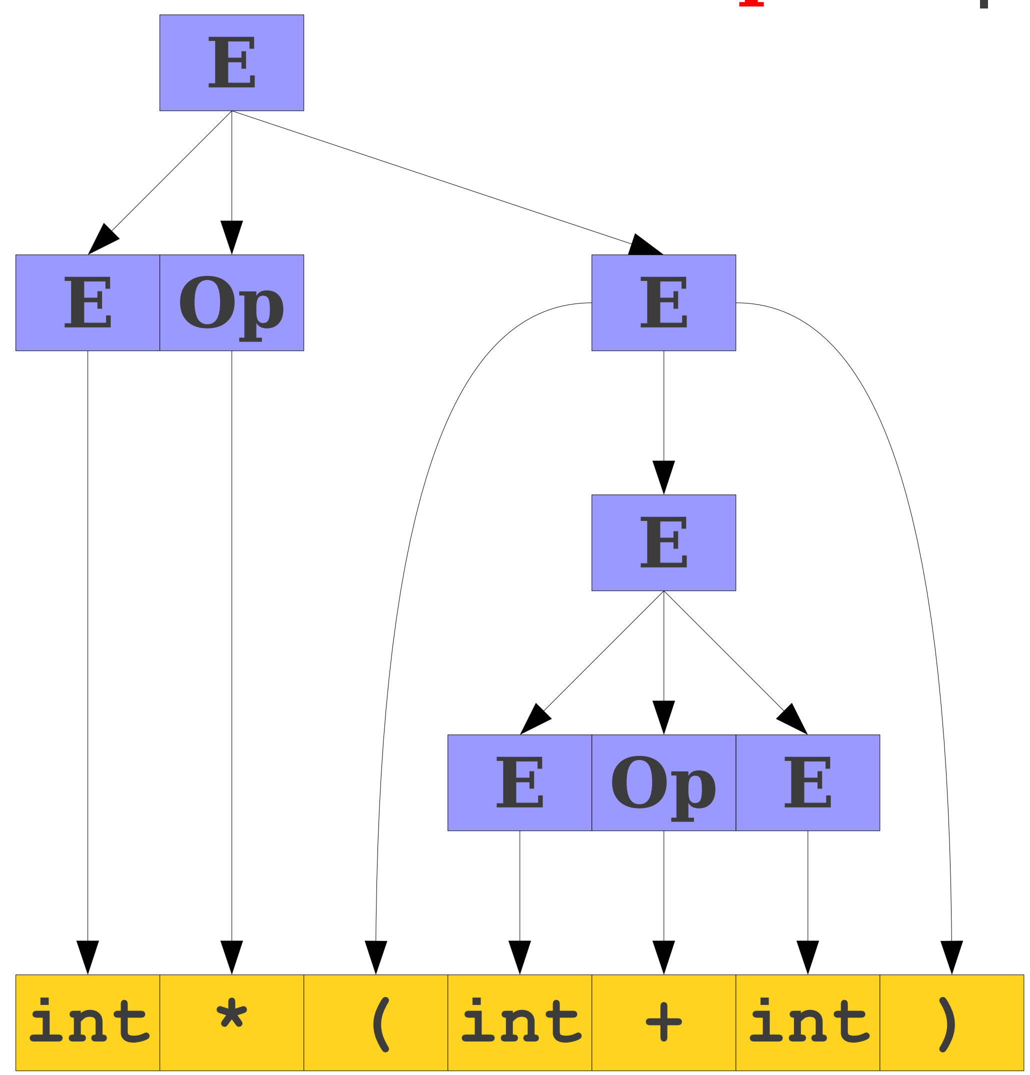
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)



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 → **`int`** | **E Op E** | **`(`E`)`**

Op → **`+`** | **`-`** | **`*`** | **`/`**

E

Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

E

Parse Trees

E → `int` | **E Op E** | `(`**E**

Op → `+` | `-` | `*` | `/`

E

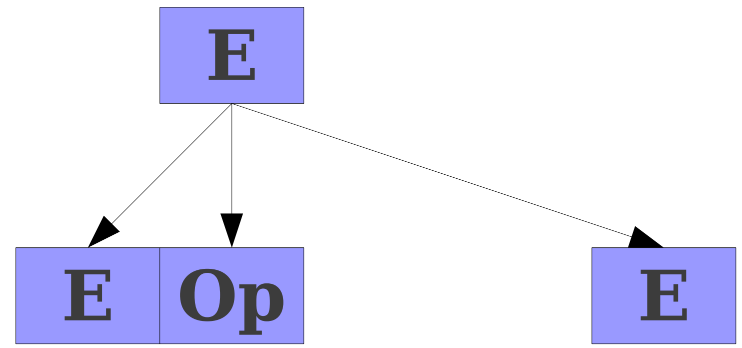
E

⇒ **E Op E**

Parse Trees

E → **int** | **E Op E** | **(E**

Op → **+** | **-** | ***** | **/**



E

⇒ **E Op E**

Parse Trees

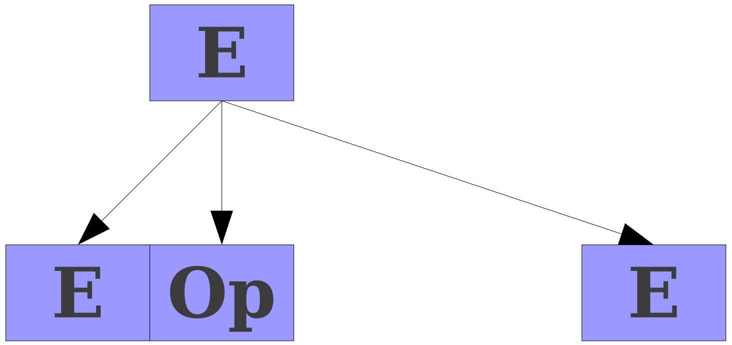
E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`



Parse Trees

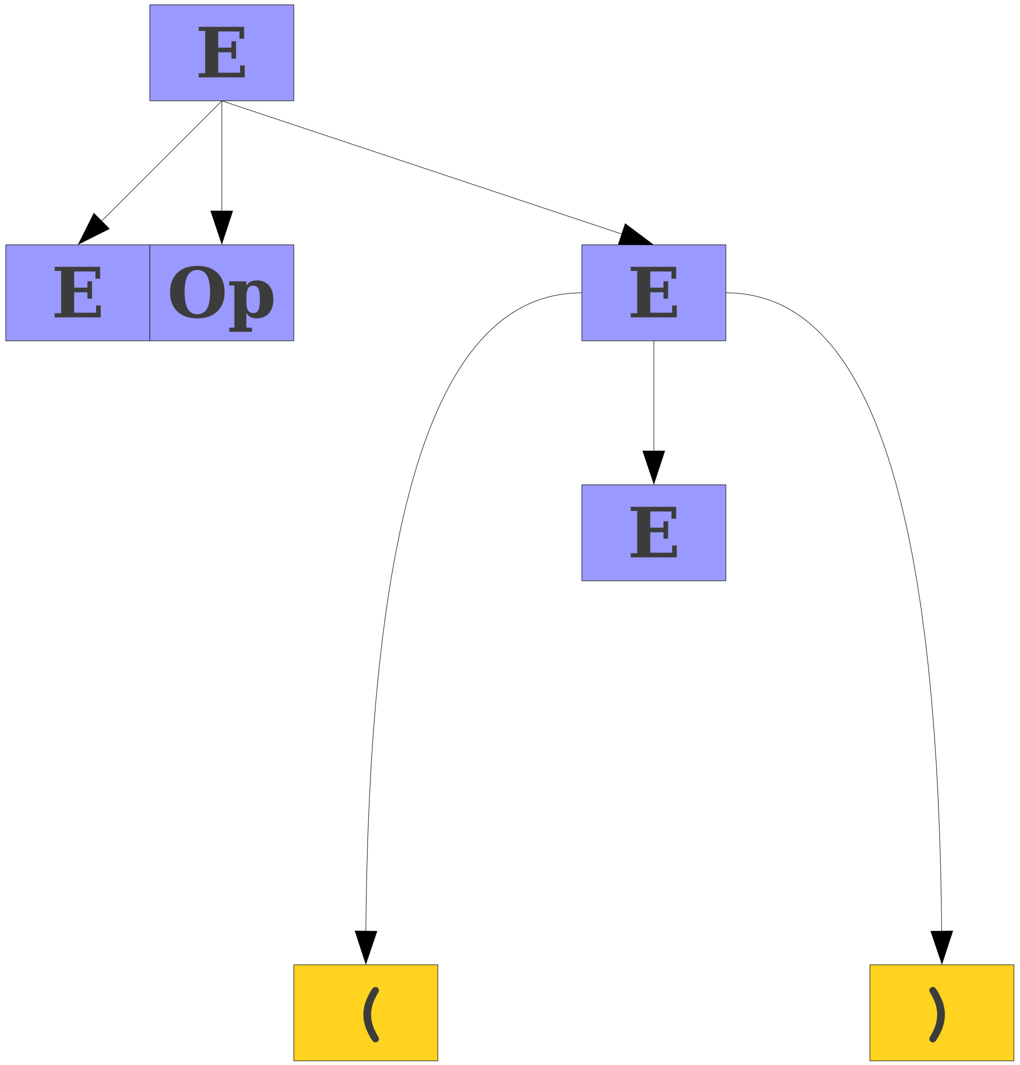
$E \rightarrow$ int | E Op E | (E)

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

E

⇒ E Op E

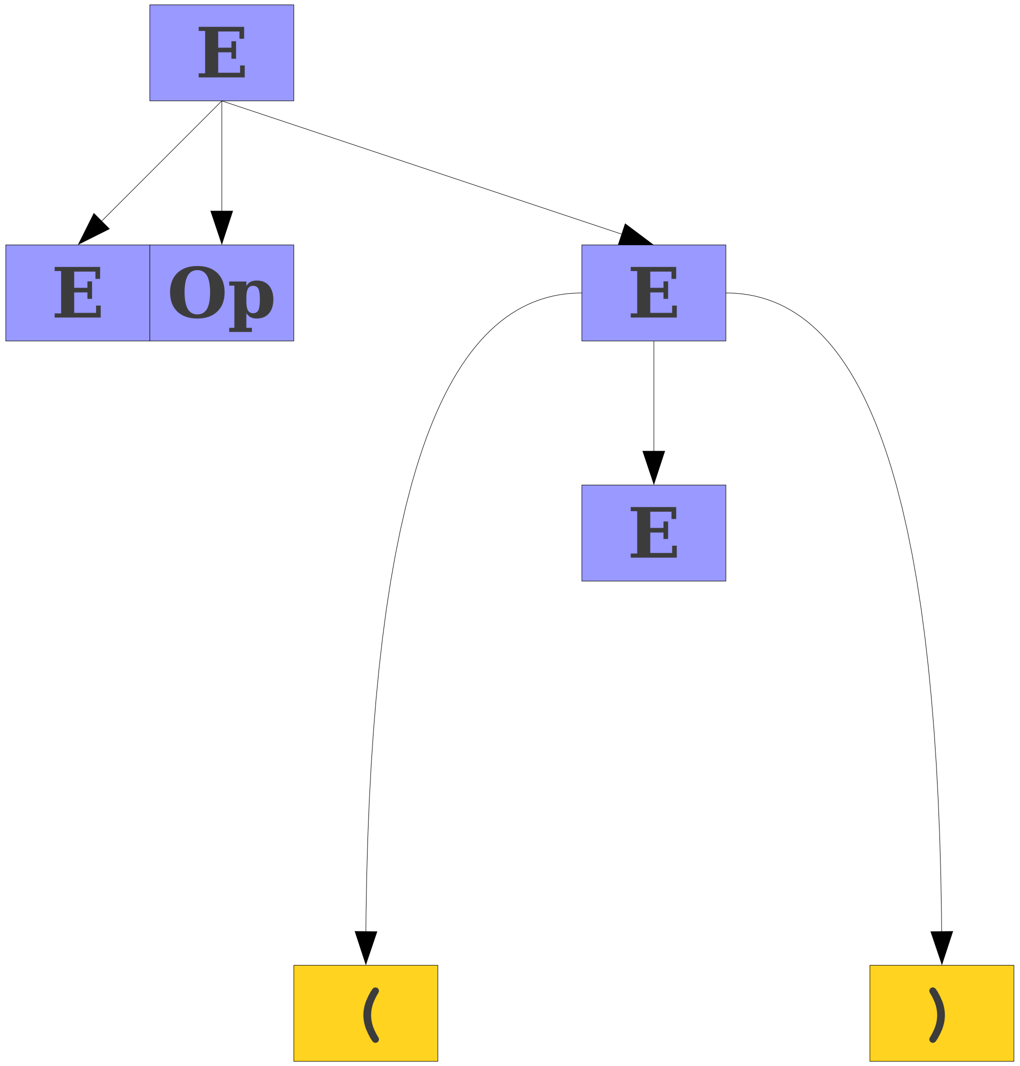
⇒ E Op (E)



Parse Trees

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

E
⇒ E Op E
⇒ E Op (E)
⇒ E Op (E Op E)



Parse Trees

E → int | E Op E | (E)

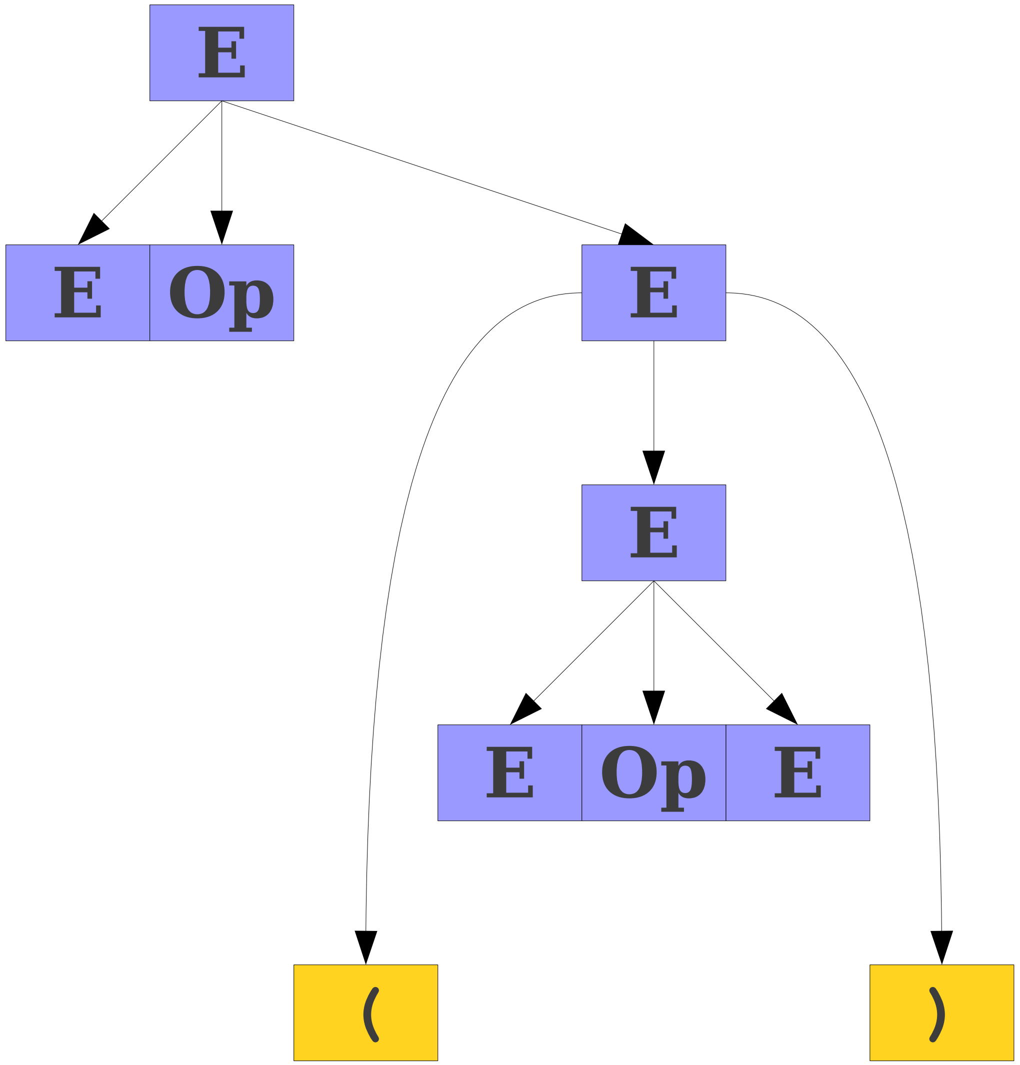
Op → + | - | * | /

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)



Parse Trees

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

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

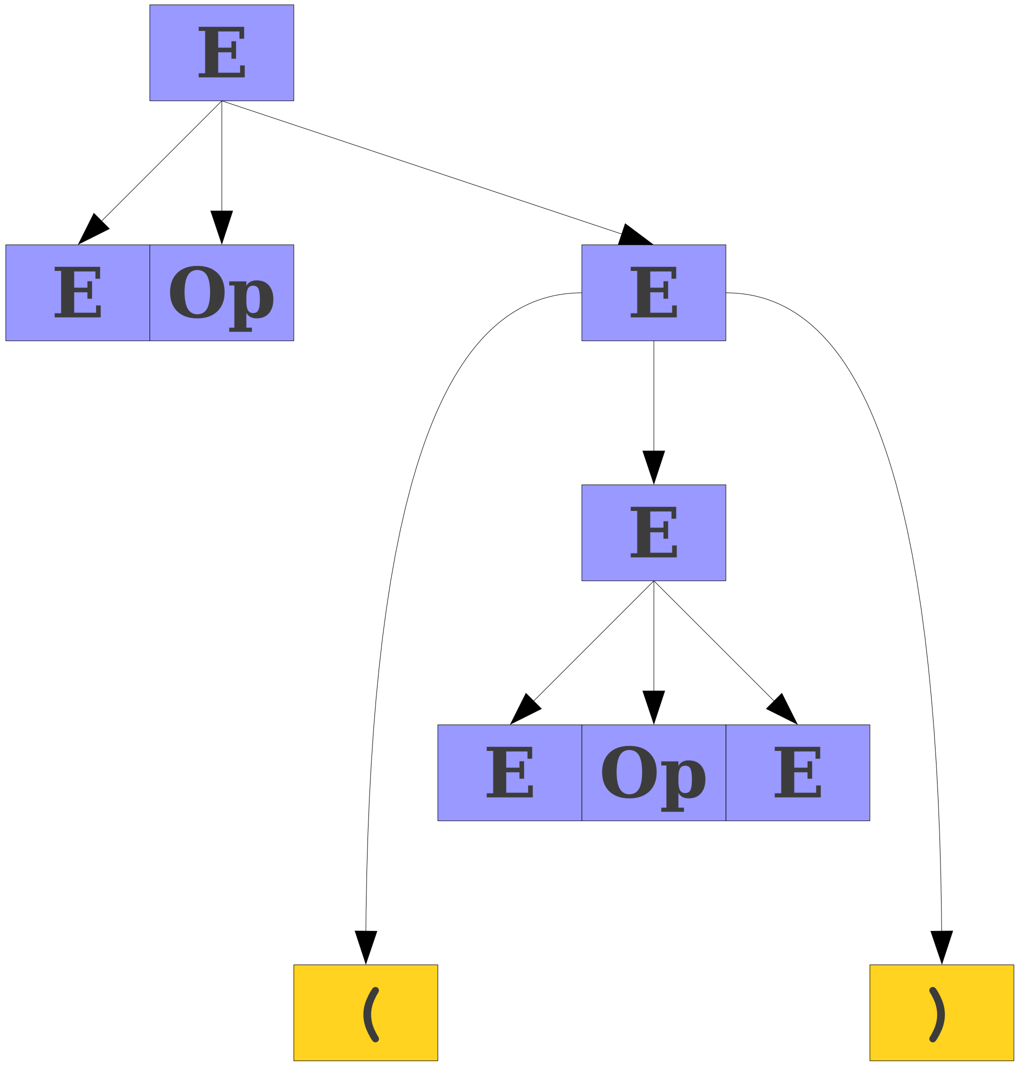
E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

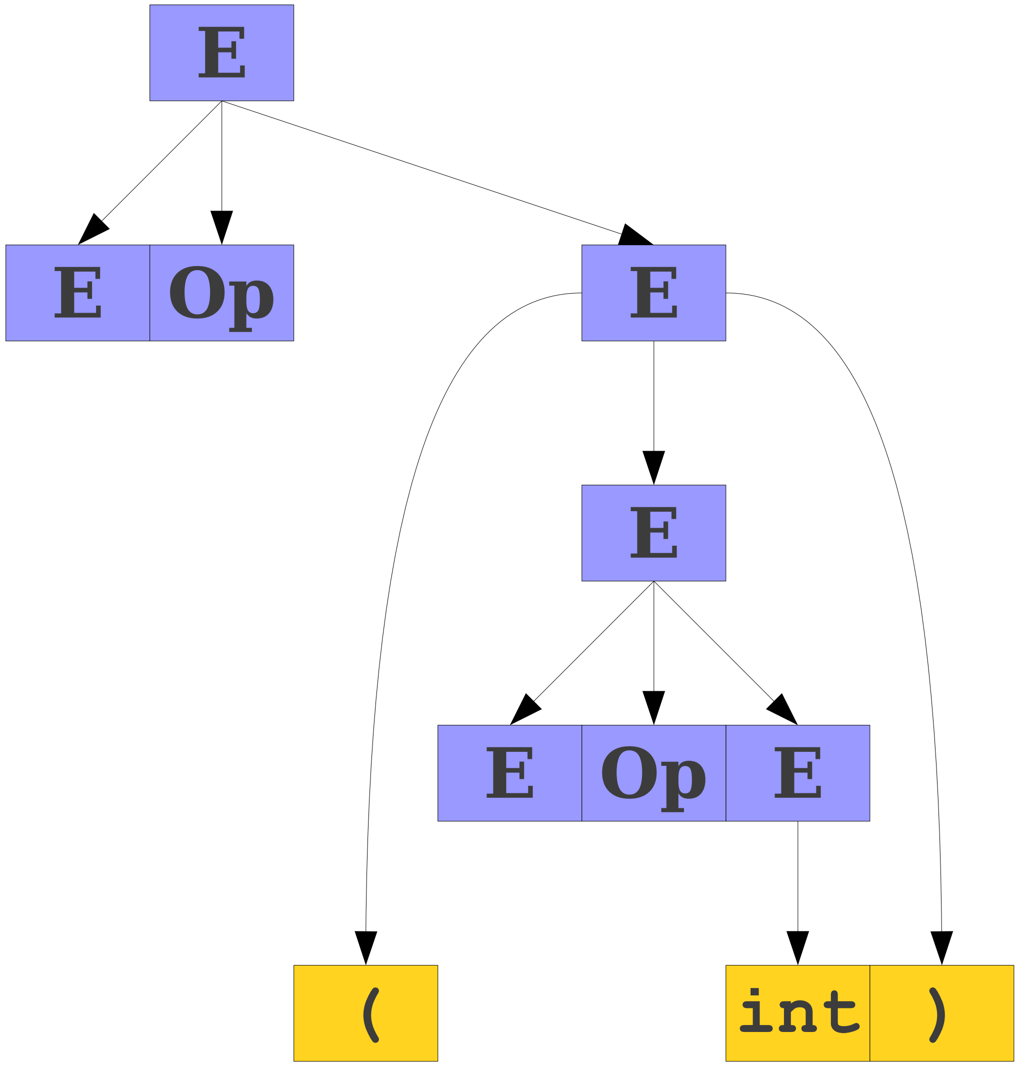
E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

⇒ **E Op** `(`**E Op** `int)`



Parse Trees

E → `int` | **E Op E** | `(`**E**

Op → `+` | `-` | `*` | `/`

E

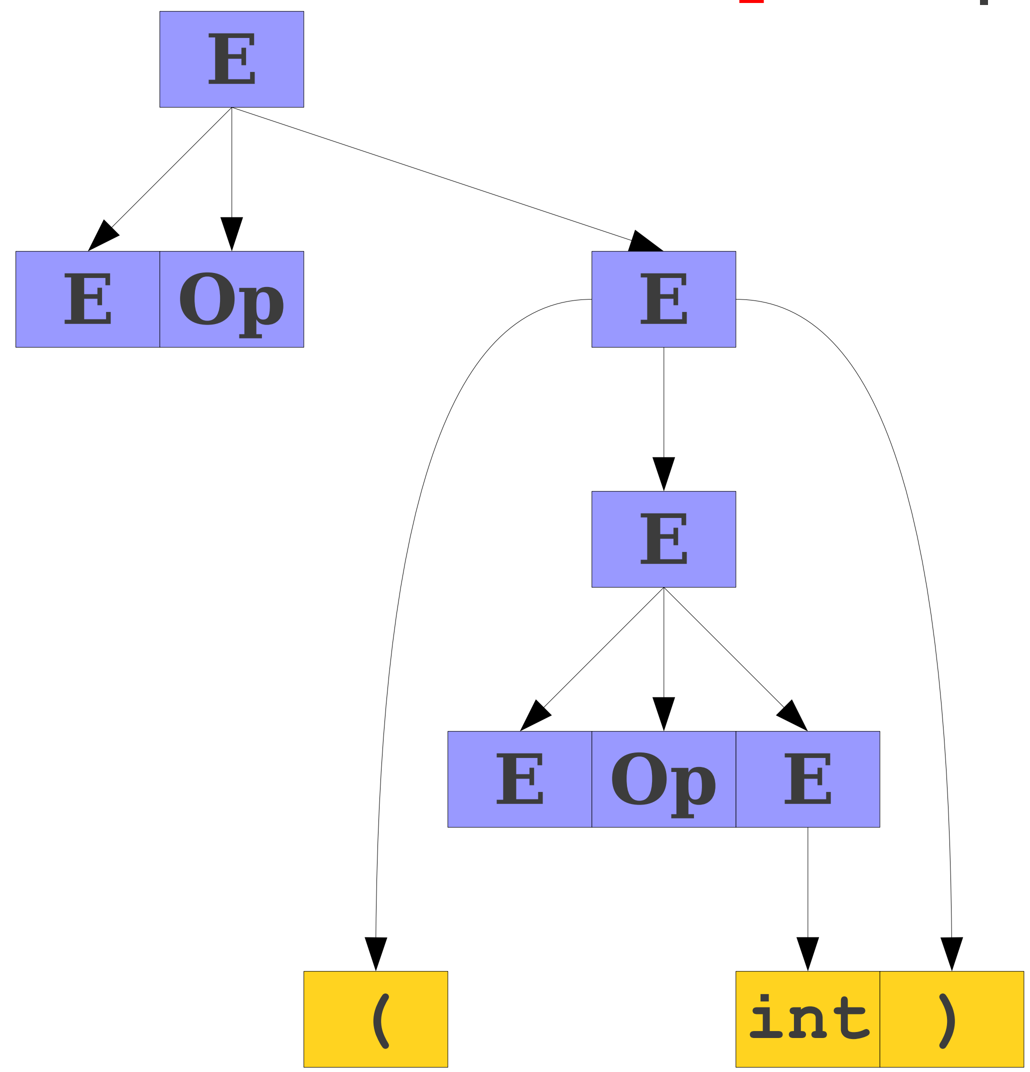
⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

⇒ **E Op** `(`**E Op** `int)`

⇒ **E Op** `(`**E** `+ int)`



Parse Trees

E → **int** | **E Op E** | **(E)**

Op → **+** | **-** | ***** | **/**

E

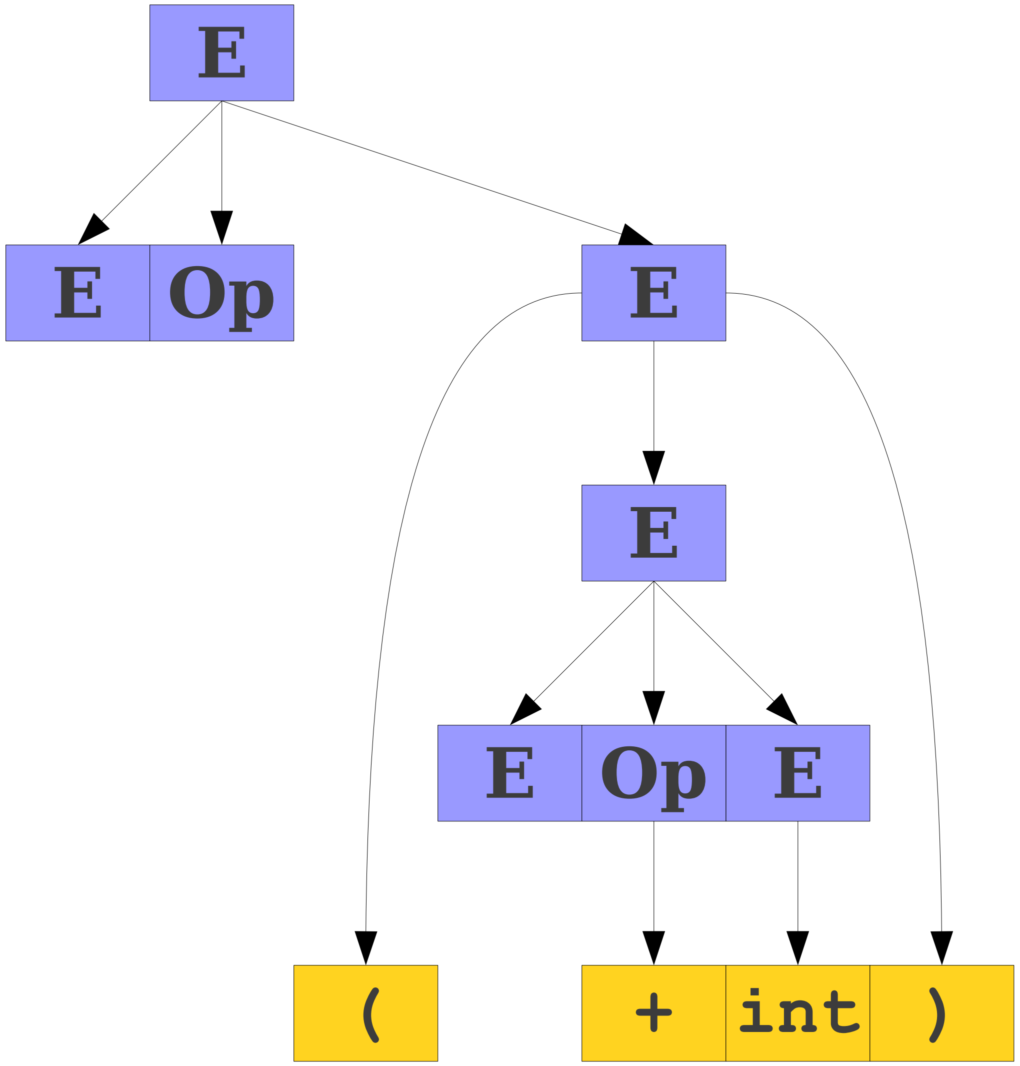
⇒ **E Op E**

⇒ **E Op (E)**

⇒ **E Op (E Op E)**

⇒ **E Op (E Op int)**

⇒ **E Op (E + int)**



Parse Trees

E → int | E Op E | (E)

Op → + | - | * | /

E

⇒ E Op E

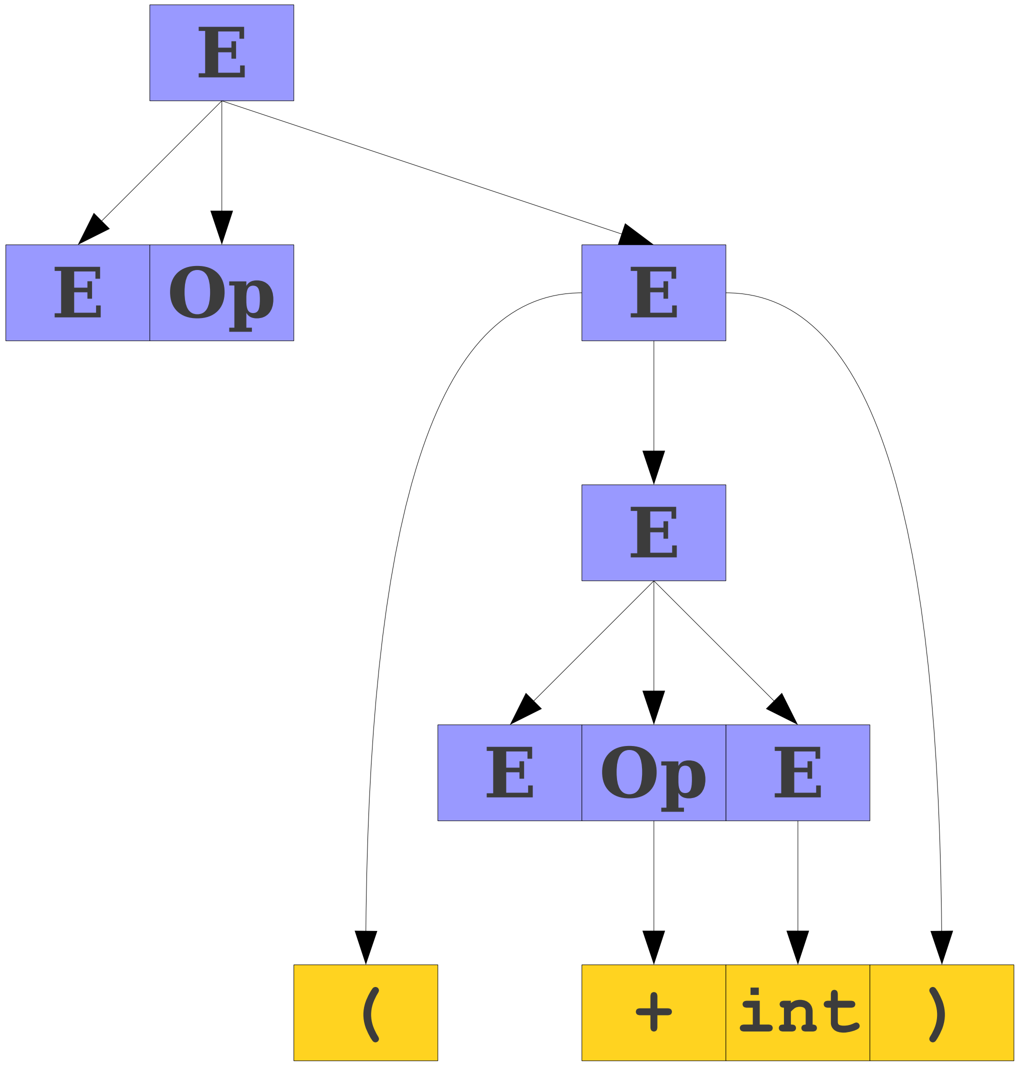
⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)



Parse Trees

E → `int` | **E Op E** | `(E)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

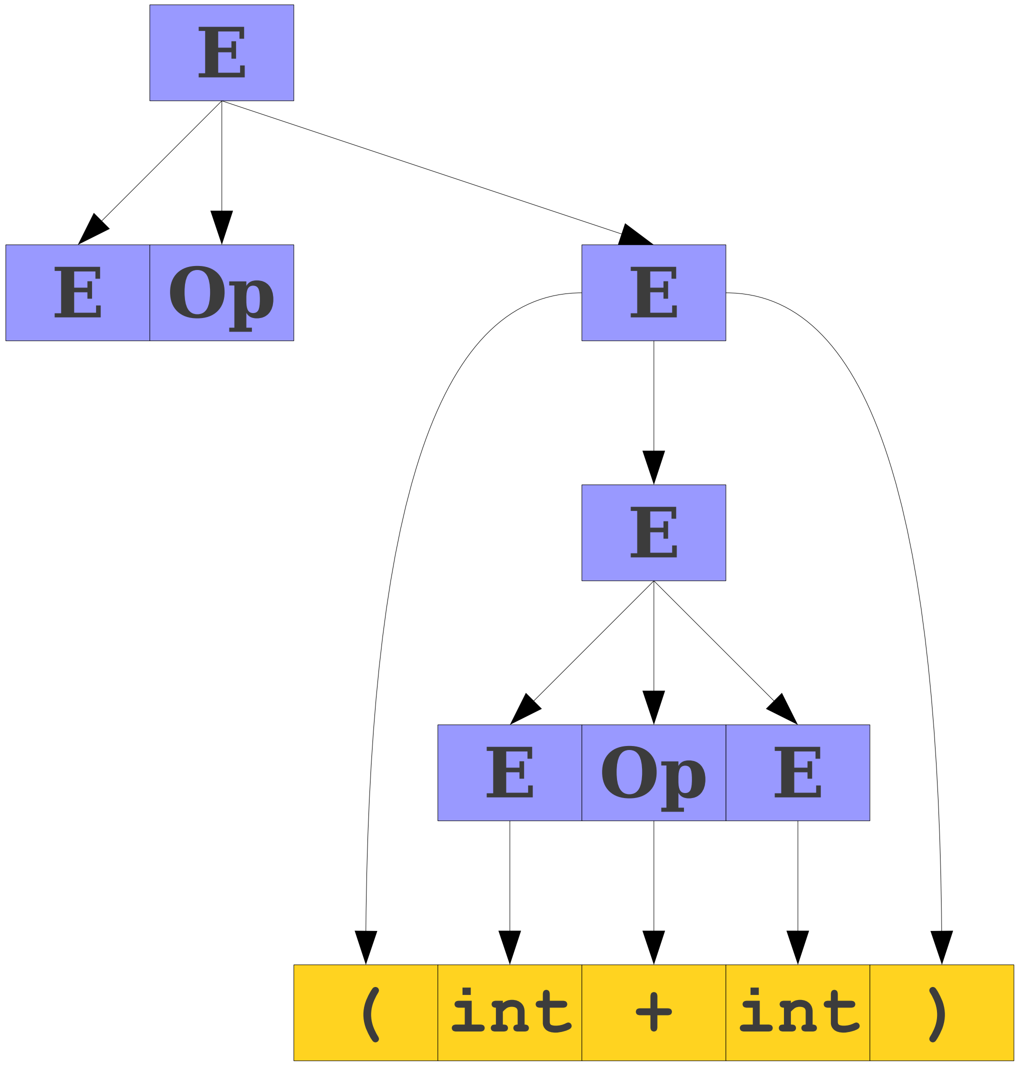
⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

⇒ **E Op** `(`**E Op** `int)`

⇒ **E Op** `(`**E** `+ int)`

⇒ **E Op** `(int + int)`



Parse Trees

E → int | E Op E | (E)

Op → + | - | * | /

E

⇒ E Op E

⇒ E Op (E)

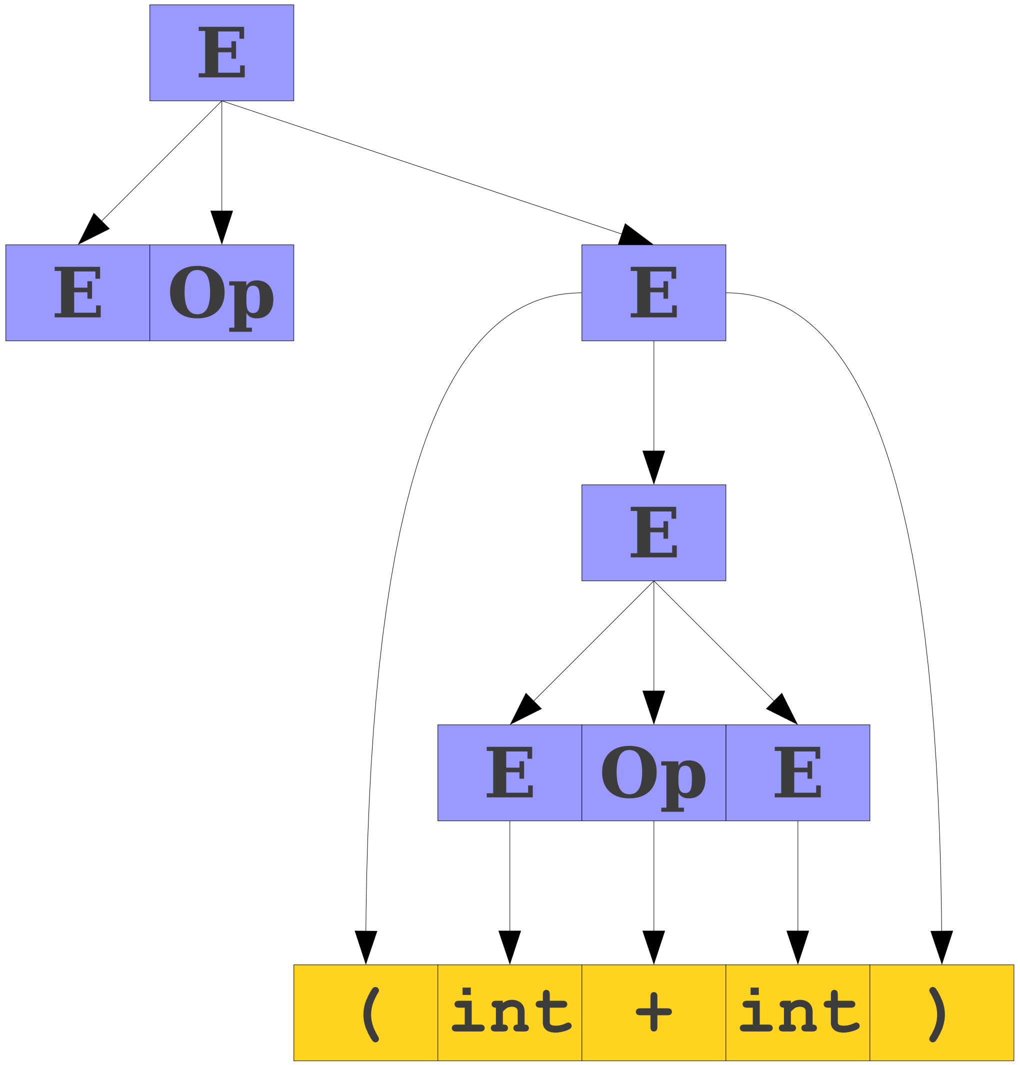
⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)

⇒ E * (int + int)



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

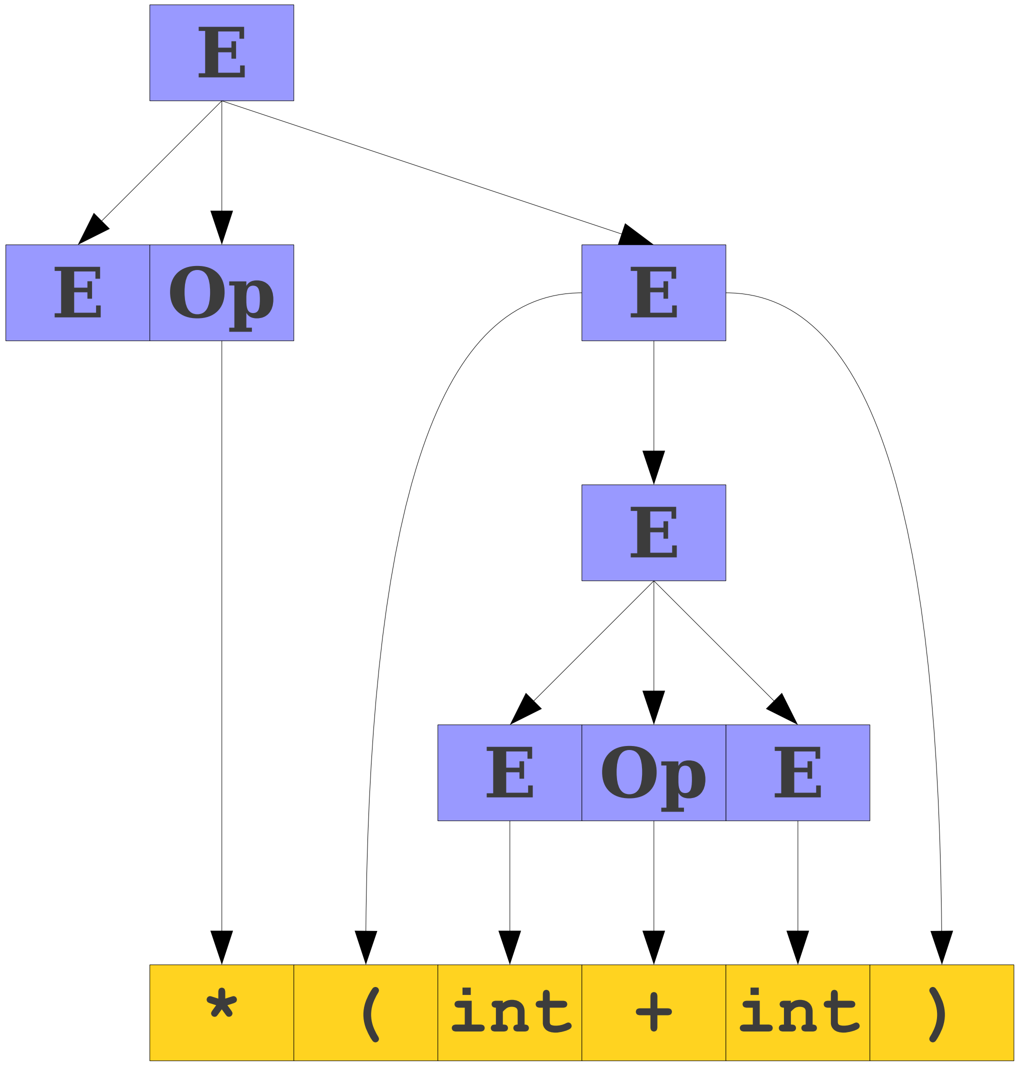
⇒ **E Op** `(`**E Op E**`)`

⇒ **E Op** `(`**E Op** `int)`

⇒ **E Op** `(`**E** `+ int)`

⇒ **E Op** `(int + int)`

⇒ **E** `* (int + int)`



Parse Trees

E → `int` | **E Op E** | `(E)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

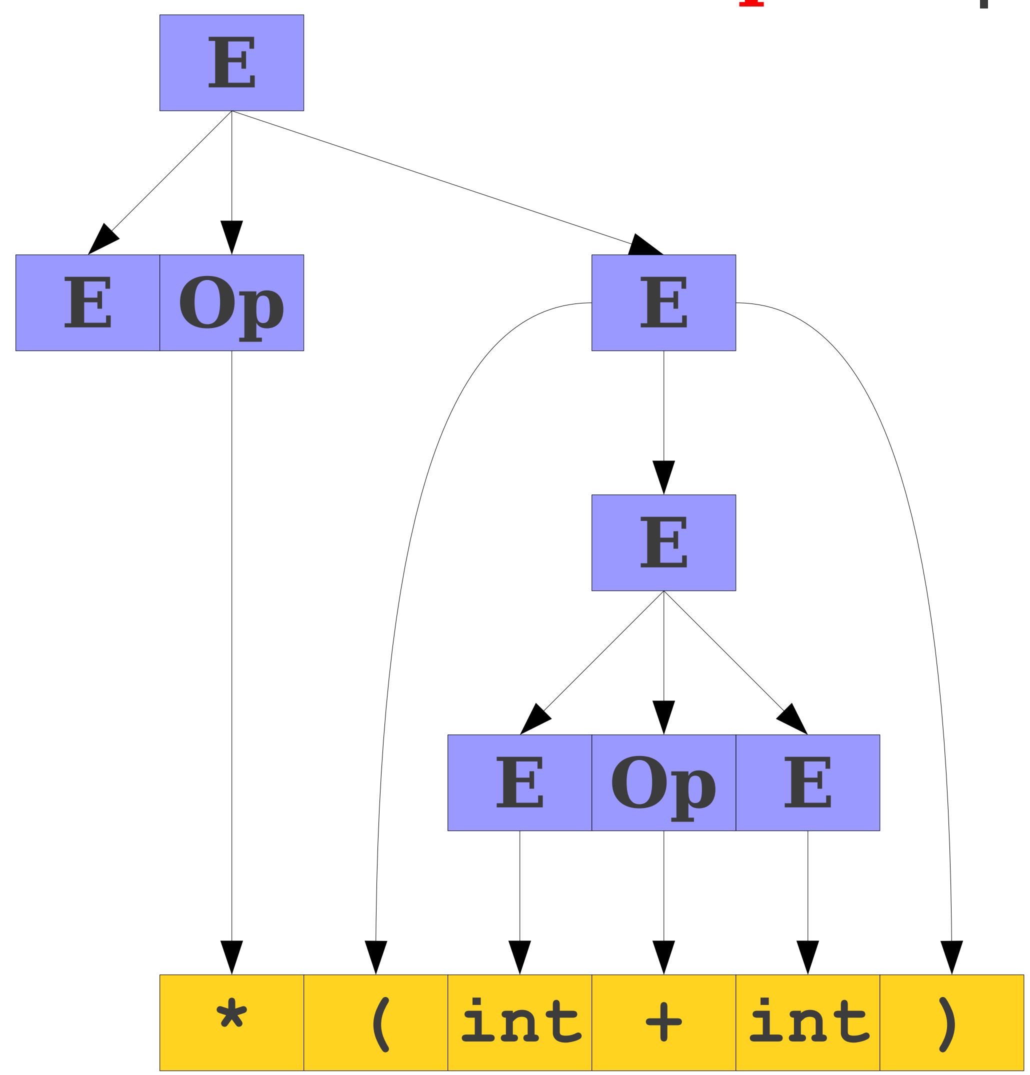
⇒ **E Op** `(`**E Op** `int)`

⇒ **E Op** `(`**E** `+ int)`

⇒ **E Op** `(int + int)`

⇒ **E** `* (int + int)`

⇒ `int * (int + int)`



Parse Trees

E → int | **E Op E** | (E)

Op → + | - | * | /

E

⇒ **E Op E**

⇒ **E Op** (**E**)

⇒ **E Op** (**E Op E**)

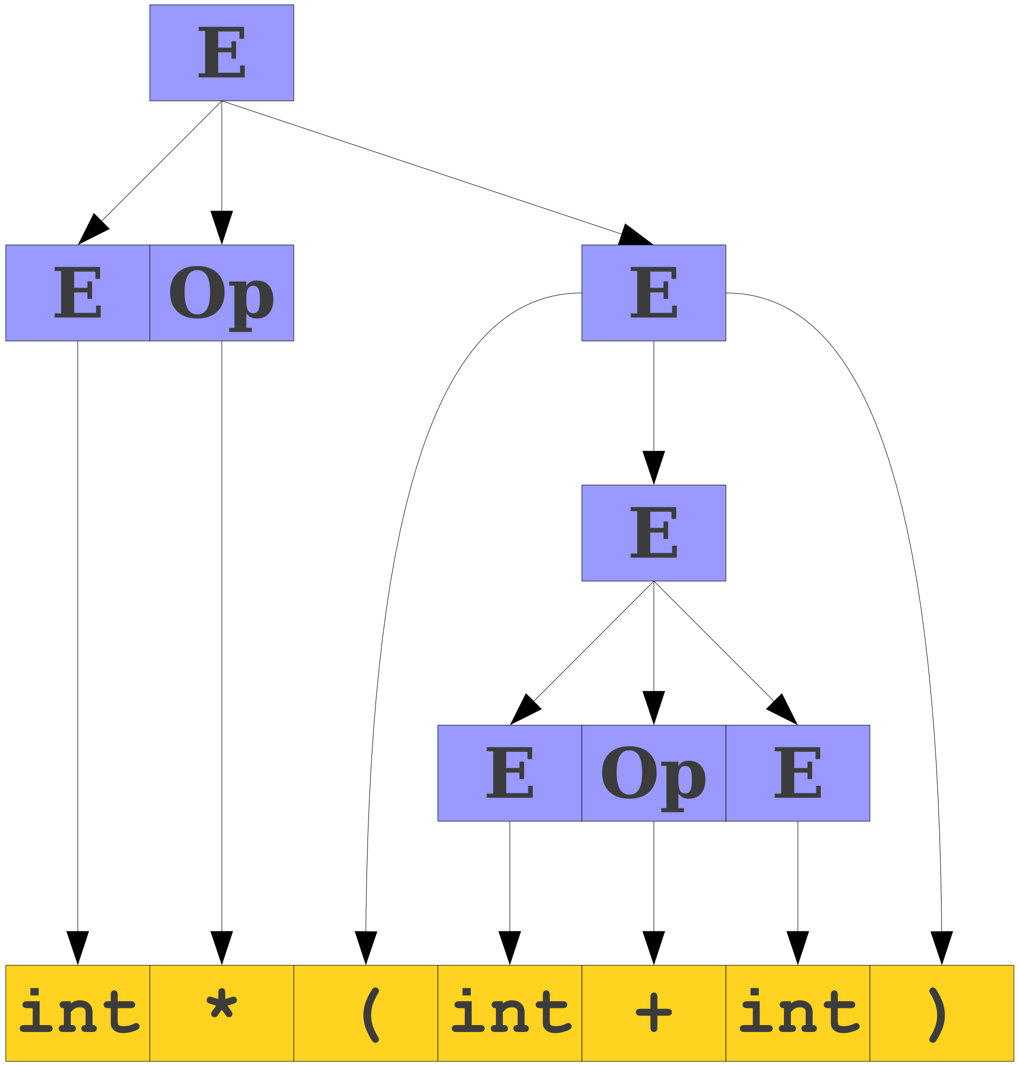
⇒ **E Op** (**E Op** int)

⇒ **E Op** (**E** + int)

⇒ **E Op** (int + int)

⇒ **E** * (int + int)

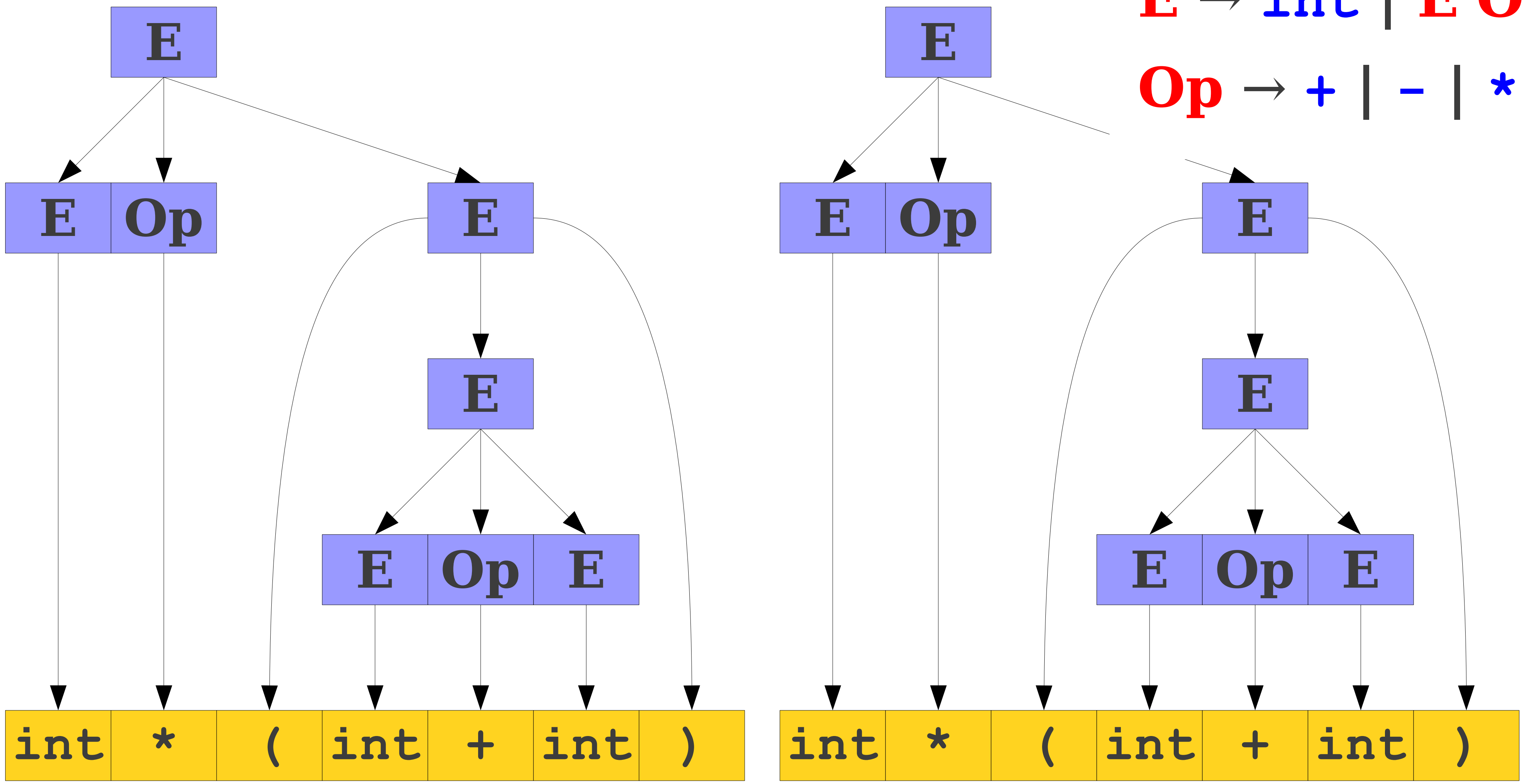
⇒ int * (int + int)



For Comparison

E → int | E Op E | (E

Op → + | - | * | /



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

But the order of the construction is different

Parse Trees

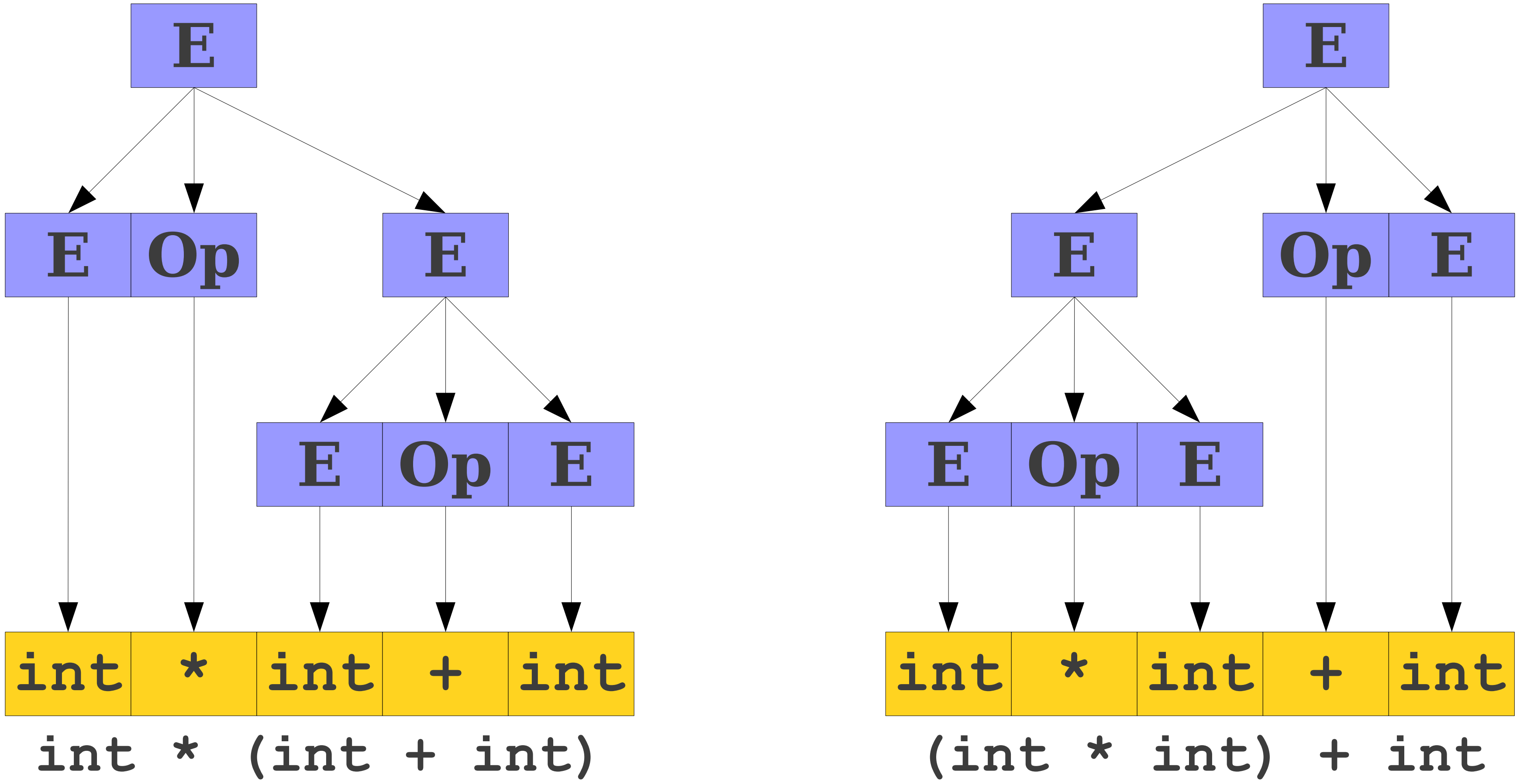
- A **parse tree** is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Inorder walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

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 next class ...

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*.

Is Ambiguity a Problem?

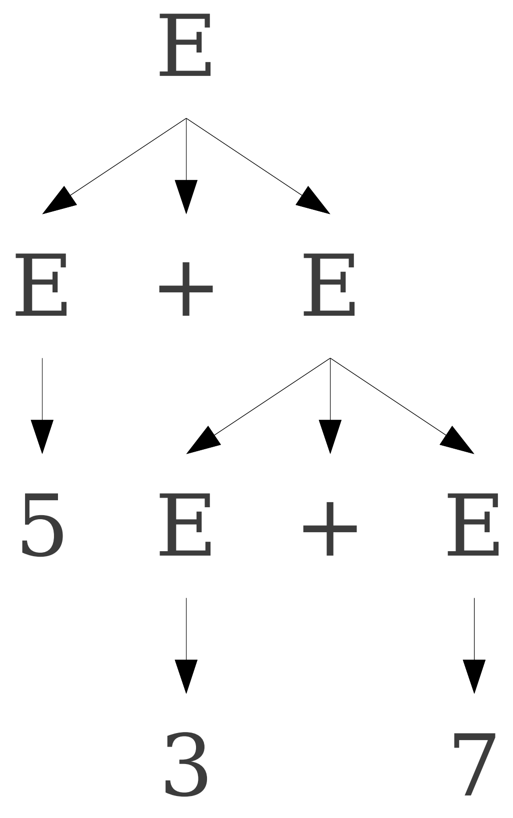
- Depends on **semantics**.

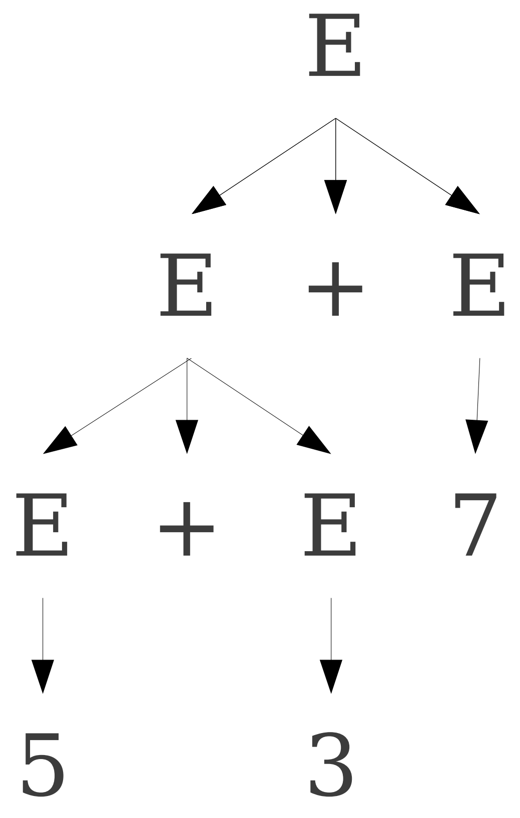
$$\mathbf{E} \rightarrow \texttt{int} \mid \mathbf{E} \; \texttt{+} \; \mathbf{E}$$

Is Ambiguity a Problem?

- Depends on **semantics**.

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





Is Ambiguity a Problem?

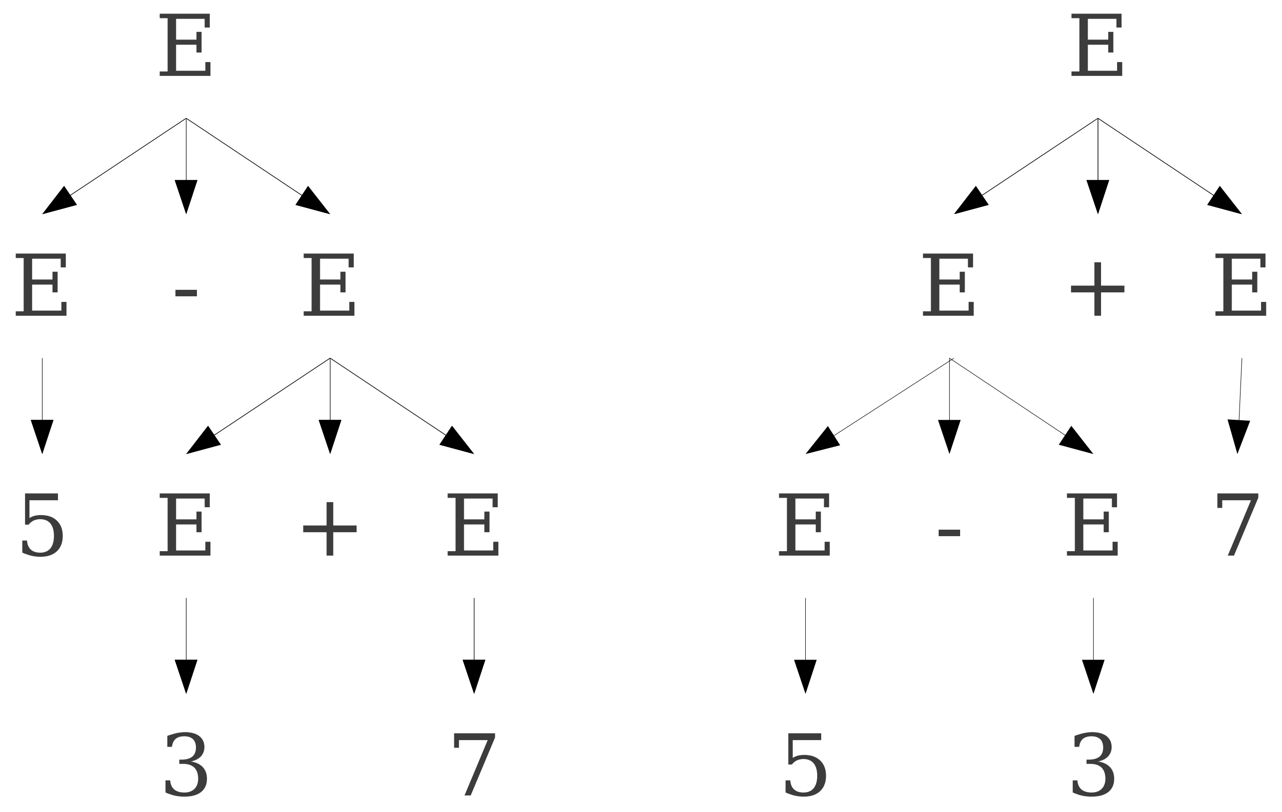
- Depends on **semantics**.

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

Is Ambiguity a Problem?

- Depends on **semantics**.

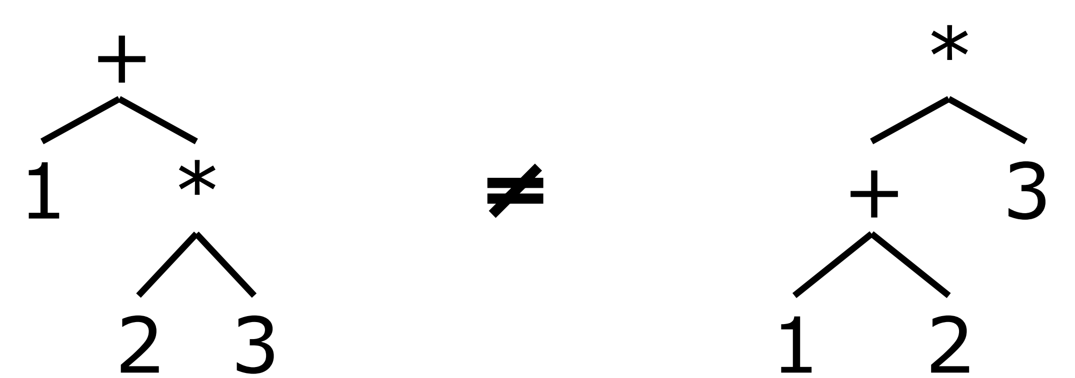
$$E \rightarrow \texttt{int} \mid E + E \mid E - E$$



Different Parse Trees

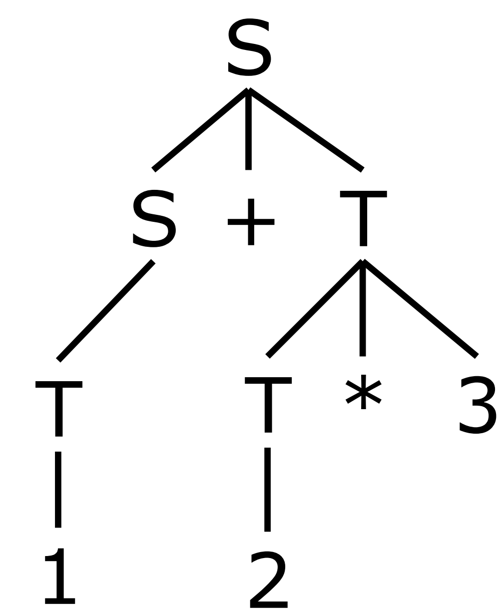
***S* → *S* + *S* | *S* * *S* | number**

- Consider expression 1 + 2 * 3
- Derivation 1: *S* →*S + S* →1 + *S* →1 + *S* * *S* → 1 + 2 * *S* →1 + 2 * 3
- Derivation 2: *S* →*S* * *S*→*S* * 3 →*S* + *S* * 3 →   *S* + 2 * 3 →1 + 2 * 3



Eliminating Ambiguity

- • Often can eliminate ambiguity by adding non-terminals & allowing recursion only on right or left
- *S* → *S* + *T* | *T*
- *T* → *T* * num | num
- • *T* non-terminal enforces precedence
- • Left-recursion : left-associativity



If-Then-Else

- How do we write a grammar for `if` statements?

- $S \rightarrow$ if (E) S
- $S \rightarrow$ if (E) S else S
- $S \rightarrow X = E$

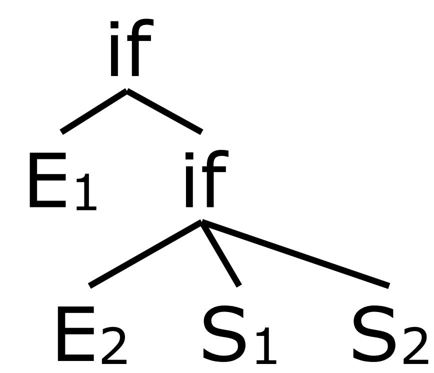
- Is this grammar OK?

No! Ambiguous

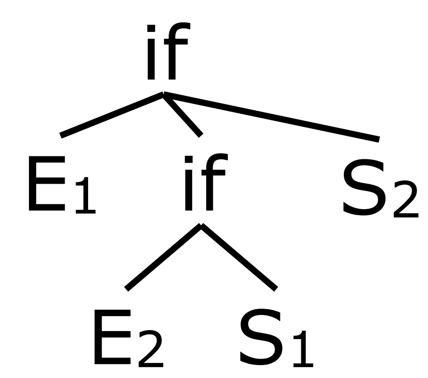
- if (*E*1) if (*E*2) *S*1 else *S*2

$S \rightarrow$ if (E) S
$S \rightarrow$ if (E) S else S
$S \rightarrow$ *other*

- *S* →if (*E*) *S*
- →if (*E*) if (*E*) *S* else *S*



- *S* →if (*E*) *S* else *S*
- →if (*E*) if (*E*) *S* 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 → matched | unmatched

matched → if (E) matched else matched | other

unmatched → if (E) statement |

if (E) matched else unmatched

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:

  R → **a** | **b** | **c** | …

  R → "**ε**"

  R → **RR**

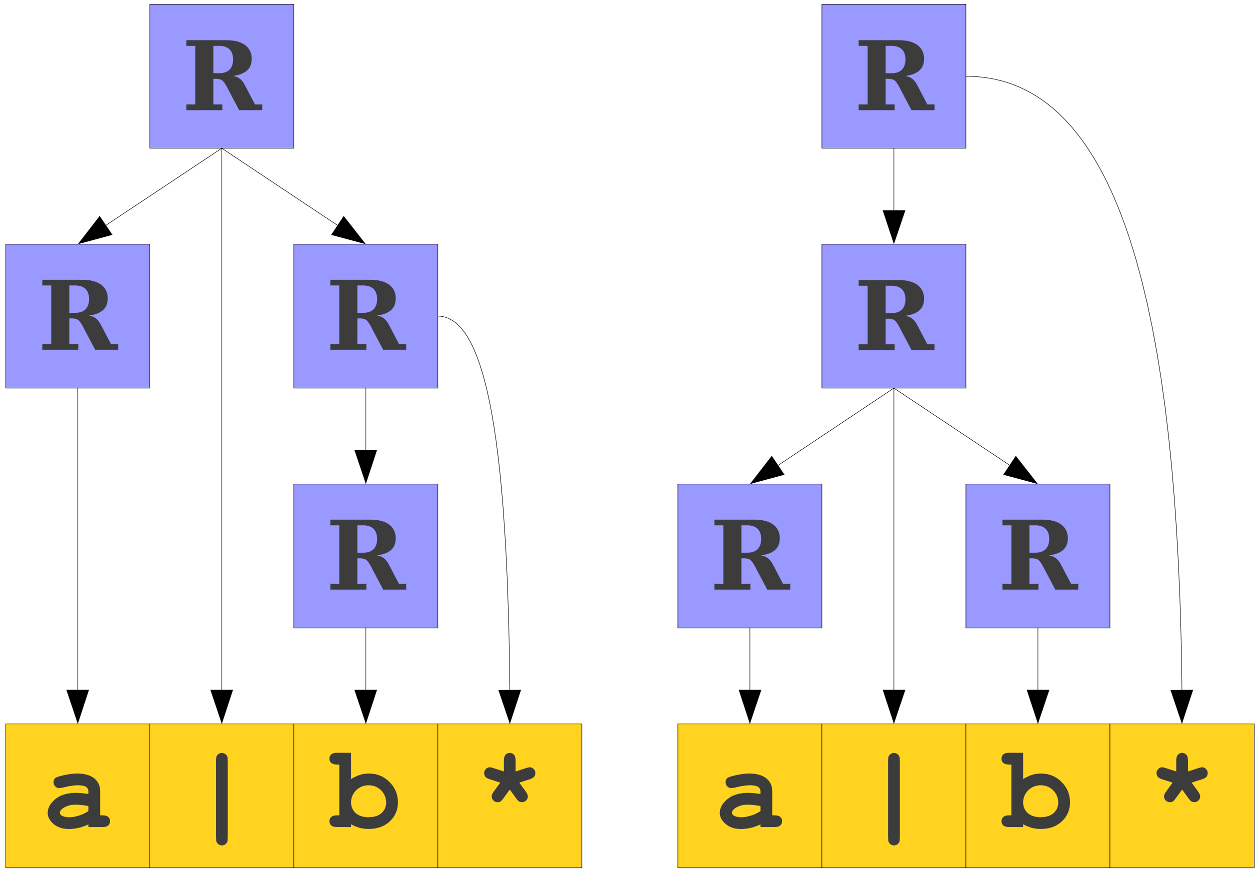
  R → **R** "**|**" **R**

  R → **R***

  R → **(R)**

An Ambiguous Grammar

R → a | b | c | …
R → "ε"
R → RR
R → R "|" R
R → R*
R → (R)



a | (b*)   (a | b)*

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

R → `a` | `b` | `c` | …

R → "`ε`"

R → **RR**

R → **R** "`|`" **R**

R → **R**`*`

R → `(`**R**`)`

R → **S** | **R** "`|`" **S**

S → **T** | **ST**

T → **U** | **T**`*`

U → `a` | `b` | `c` | …

U → "`ε`"

U → `(`**R**`)`

Why is this unambiguous?

R → **S** | **R** “|” **S**

S → **T** | **ST**

T → **U** | **T***

U → **a** | **b** | …

U → “**ε**”

U → **(R)**

Only generates
“atomic” expressions

Why is this unambiguous?

R → **S** | **R** “|” **S**

S → **T** | **ST**

T → **U** | **T***

U → a | b | …

U → “ε”

U → (**R**)

Puts stars onto atomic expressions

Only generates “atomic” expressions

Why is this unambiguous?

R → S | R "|" S

S → T | ST

T → U | T*

U → a | b | …

U → "ε"

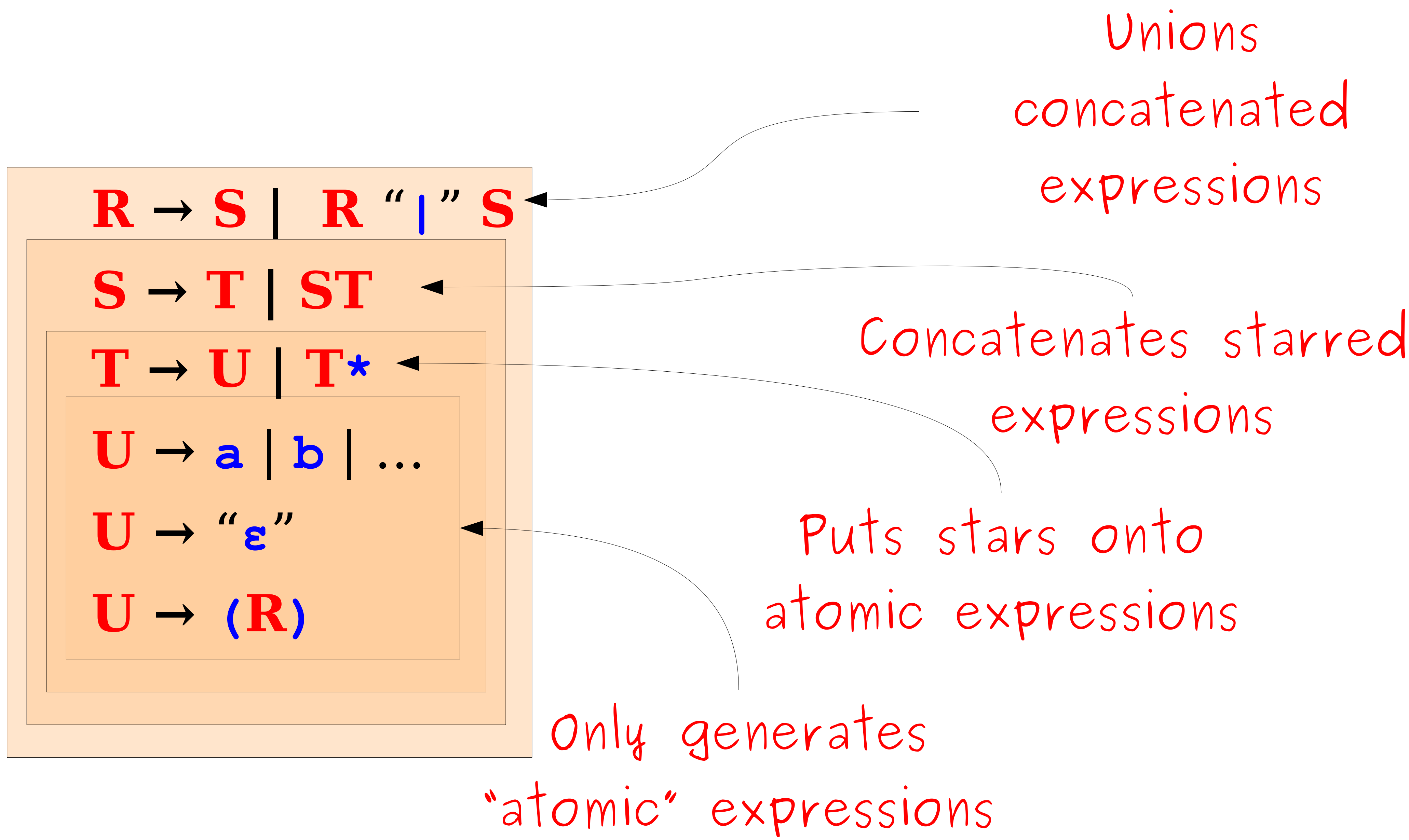
U → (R)

Concatenates starred expressions

Puts stars onto atomic expressions

Only generates "atomic" expressions

Why is this unambiguous?



R

R → S | R "|" S

S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)

a b | c | a *

R → S | R "|" S

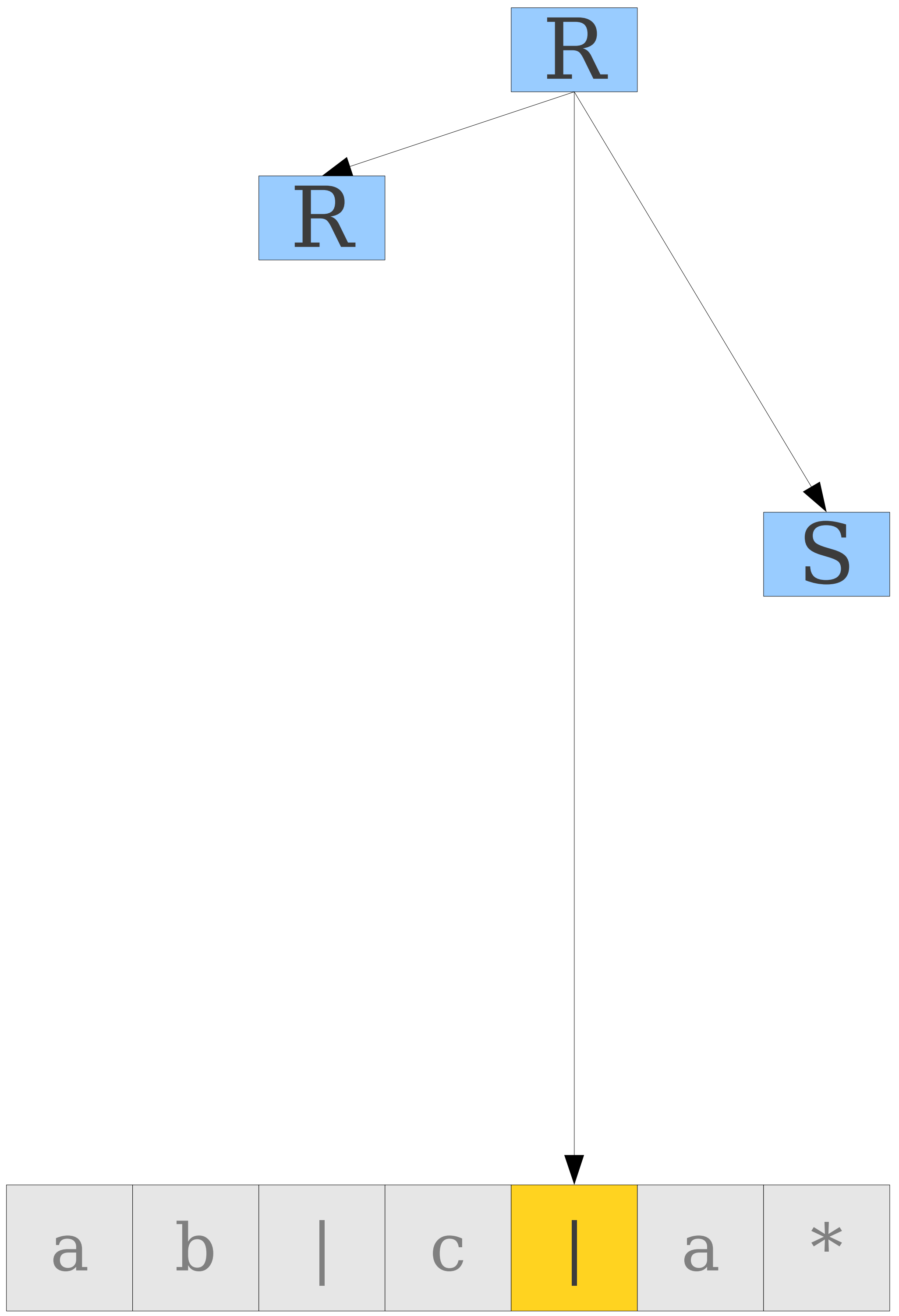
S → T | ST

T → U | T*

U → a | b | c | …

U → "ε"

U → (R)



R → **S** | **R** "|" **S**

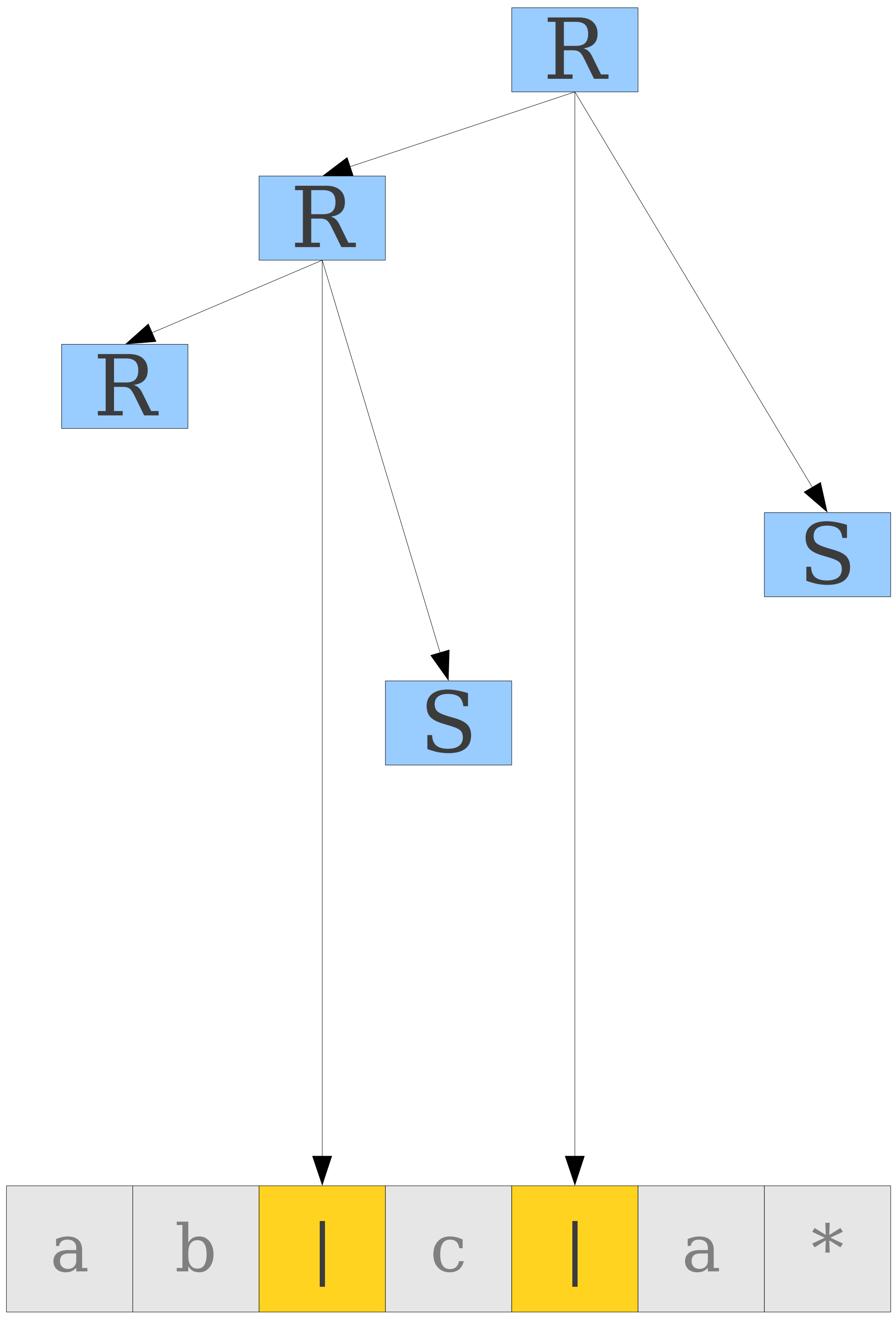
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → S | R "|" S

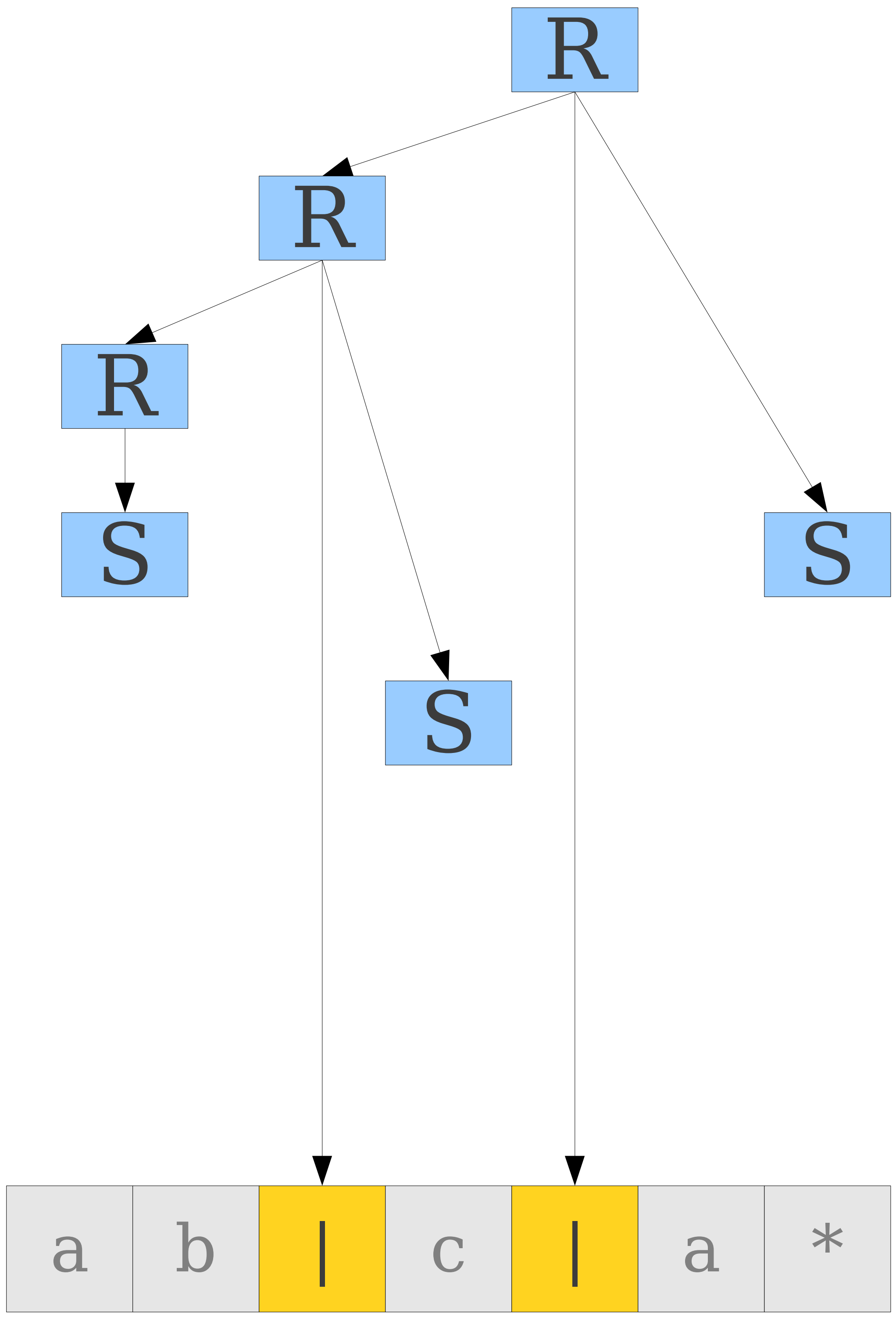
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



R → **S** | **R** "|" **S**

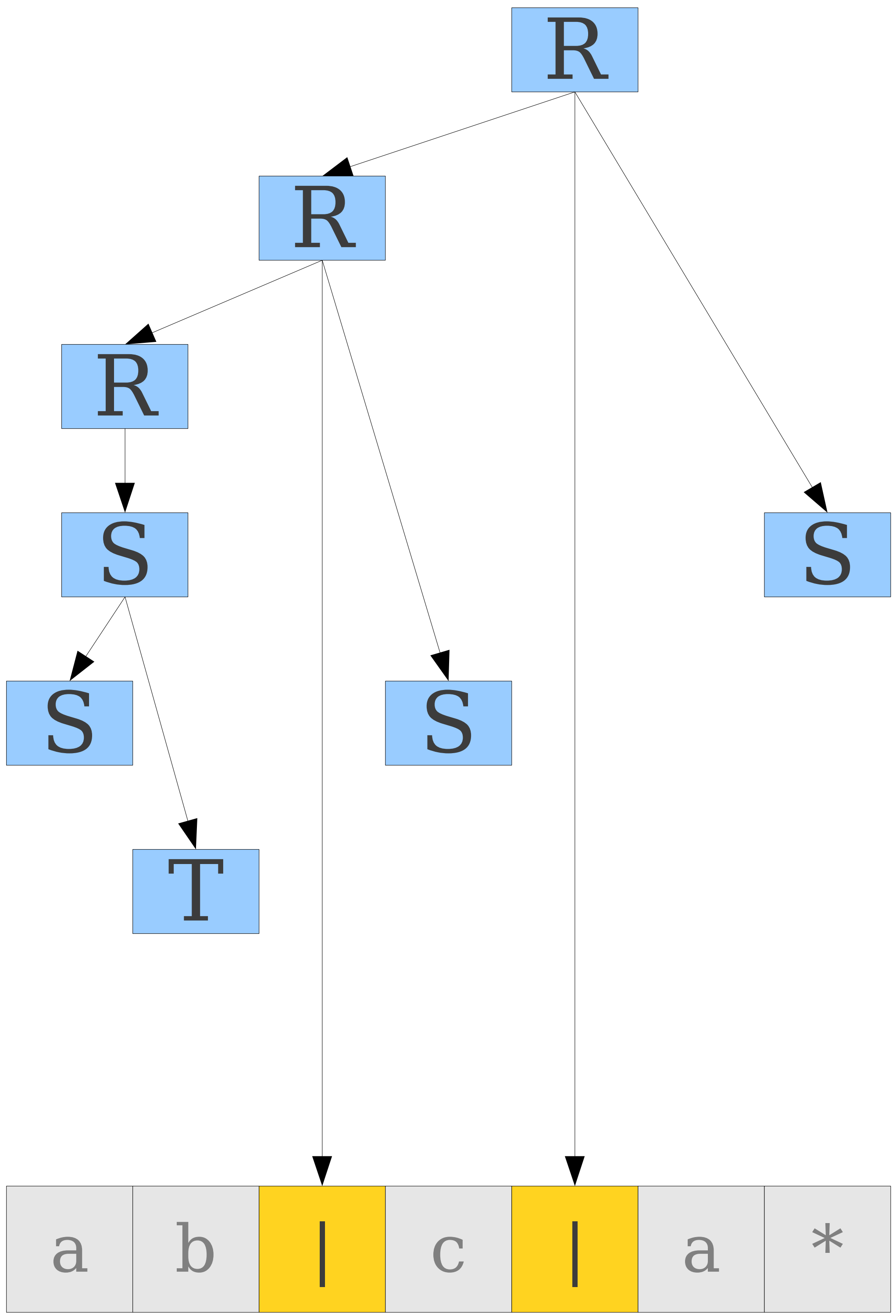
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → **S** | **R** "|" **S**

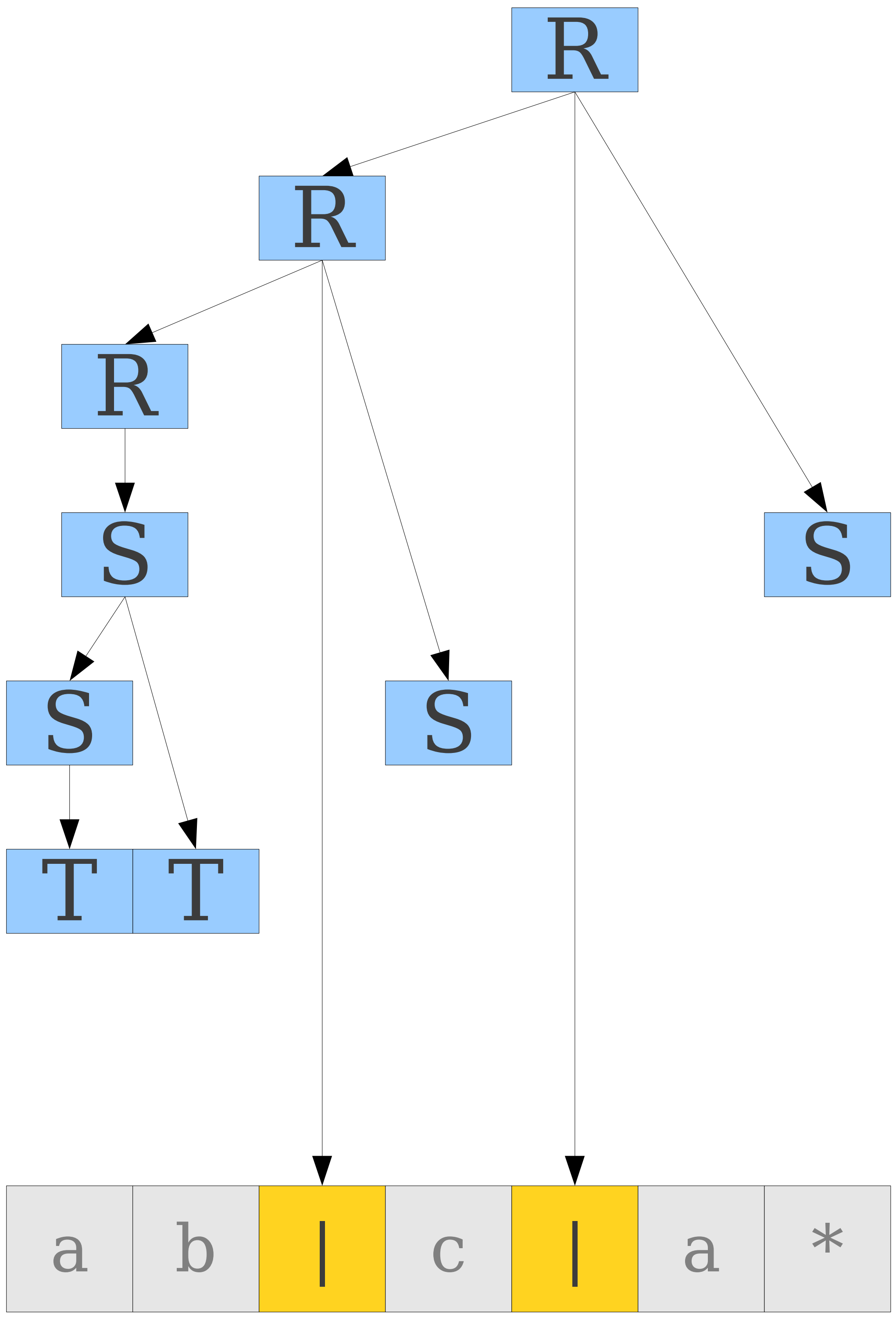
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → **S** | **R** "|" **S**

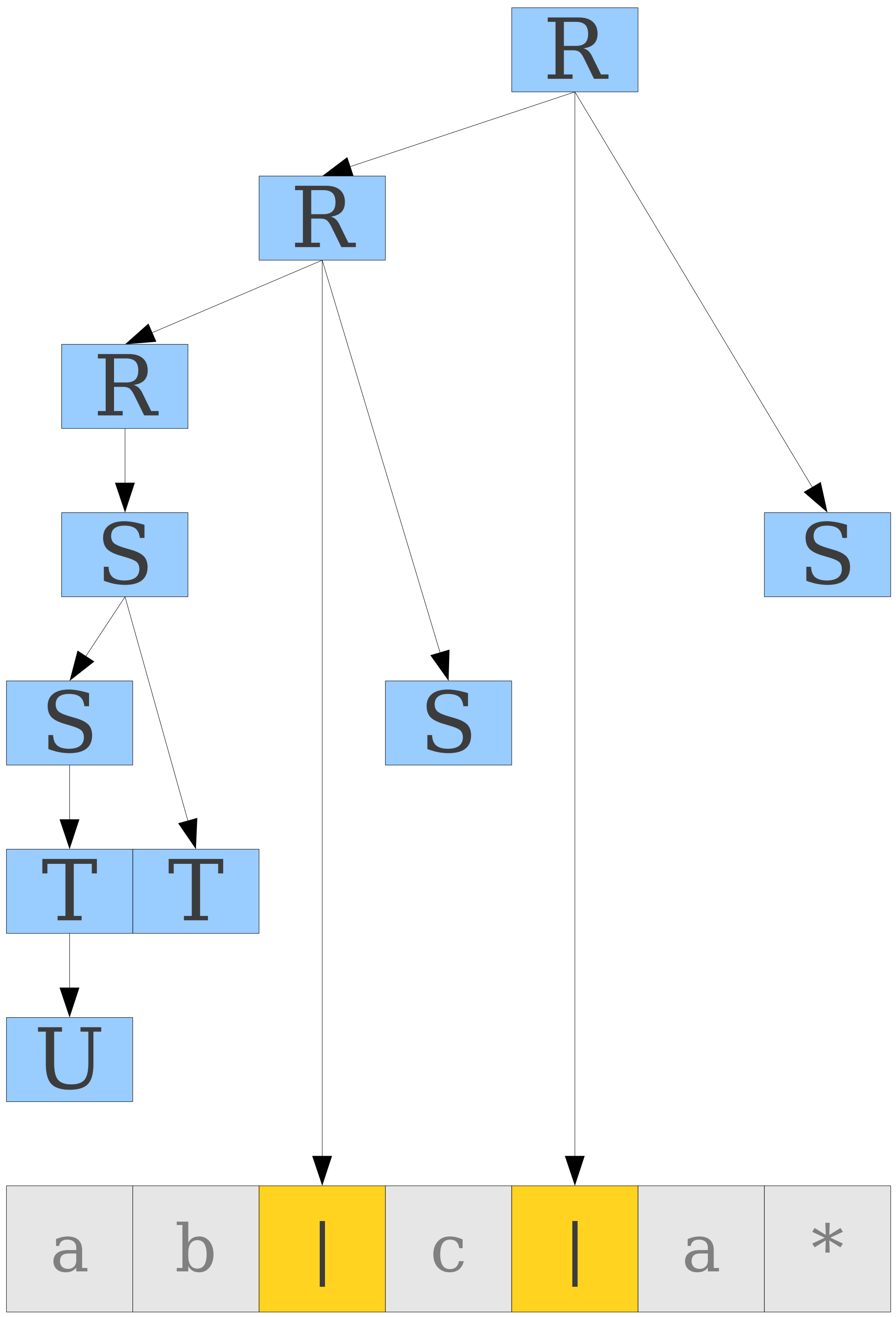
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → S | R "|" S

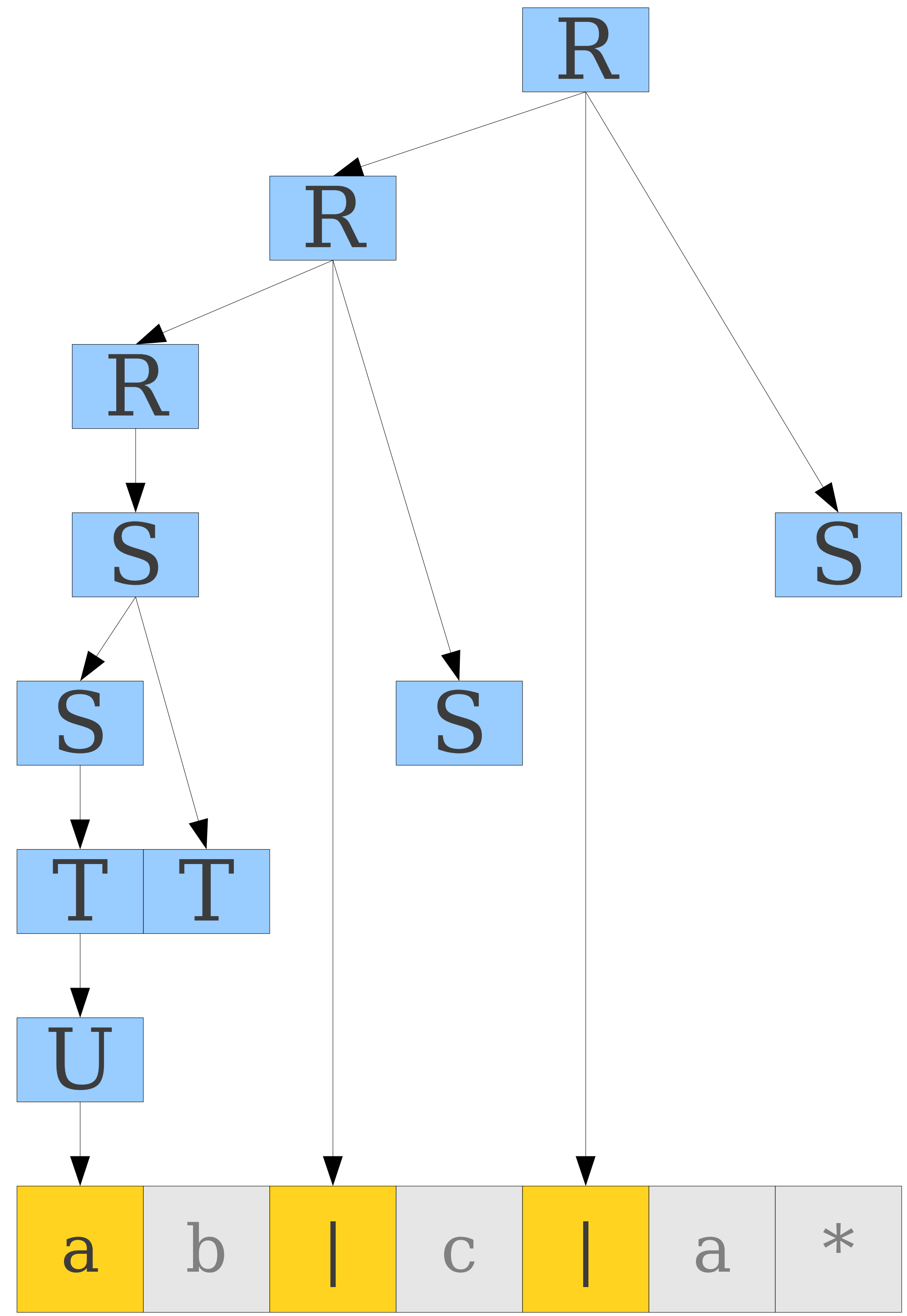
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



R → S | R “|” S

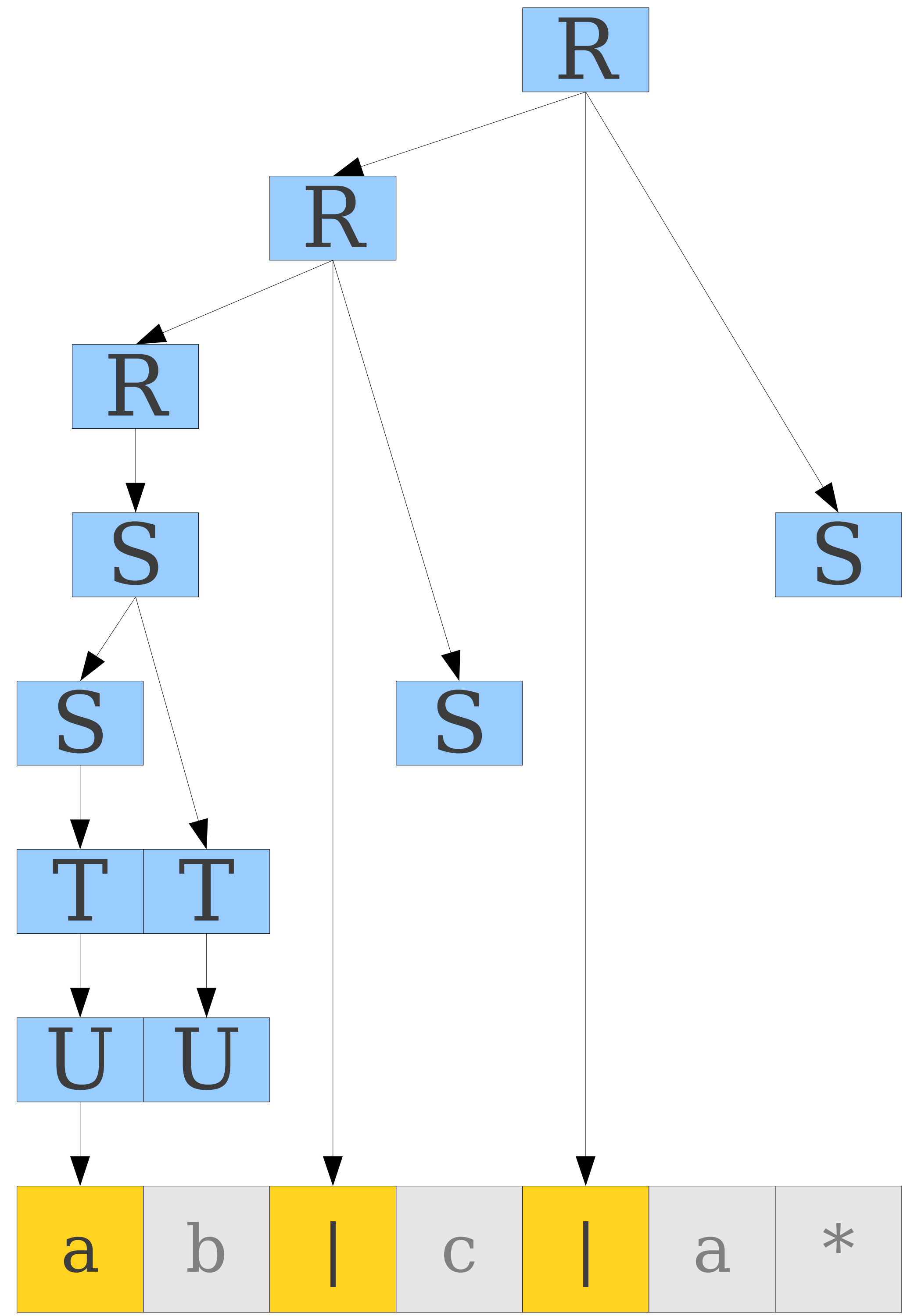
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R "|" S

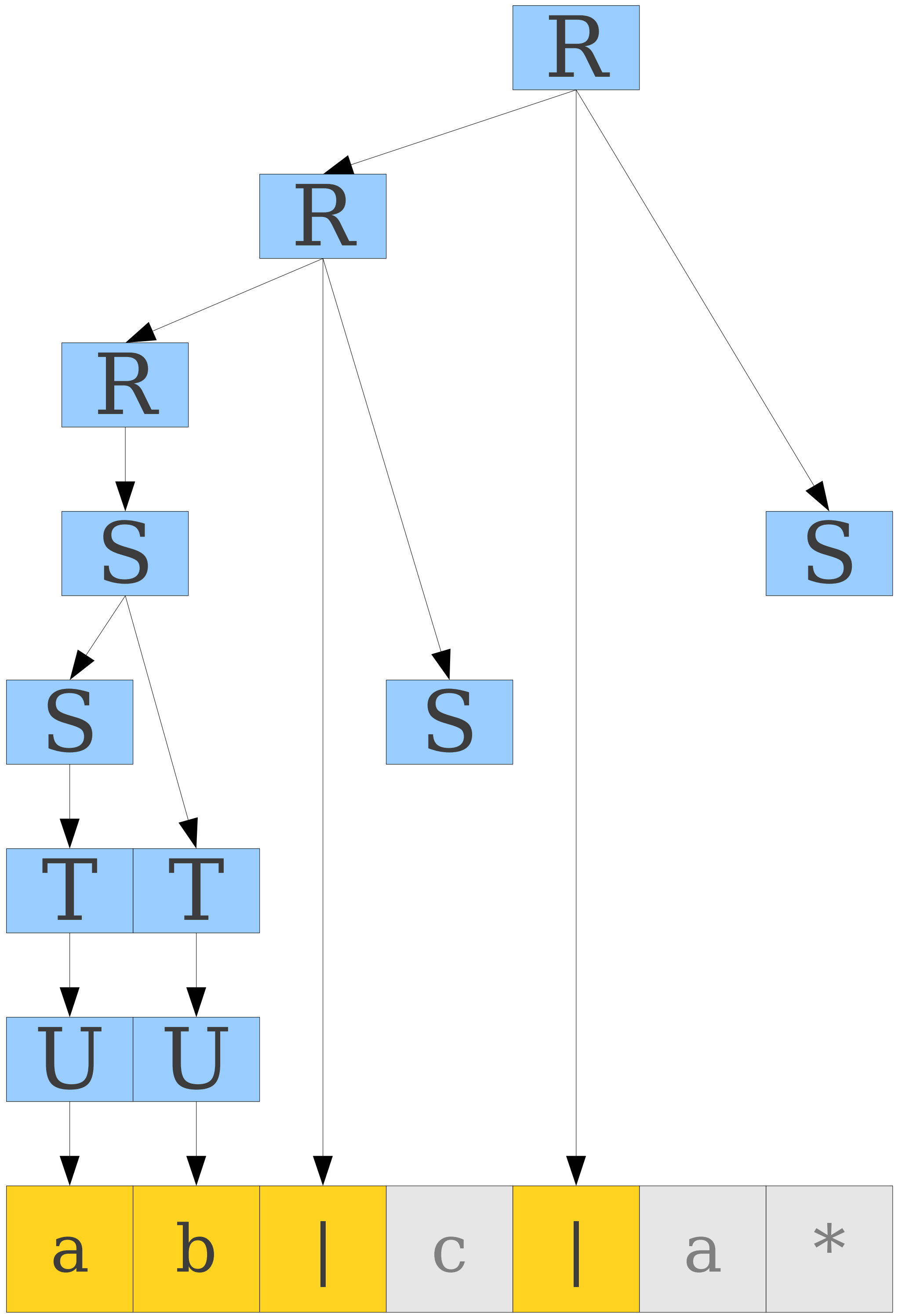
S → T | ST

T → U | T*

U → a | b | c | …

U → "ε"

U → (R)



R → **S** | **R** “|” **S**

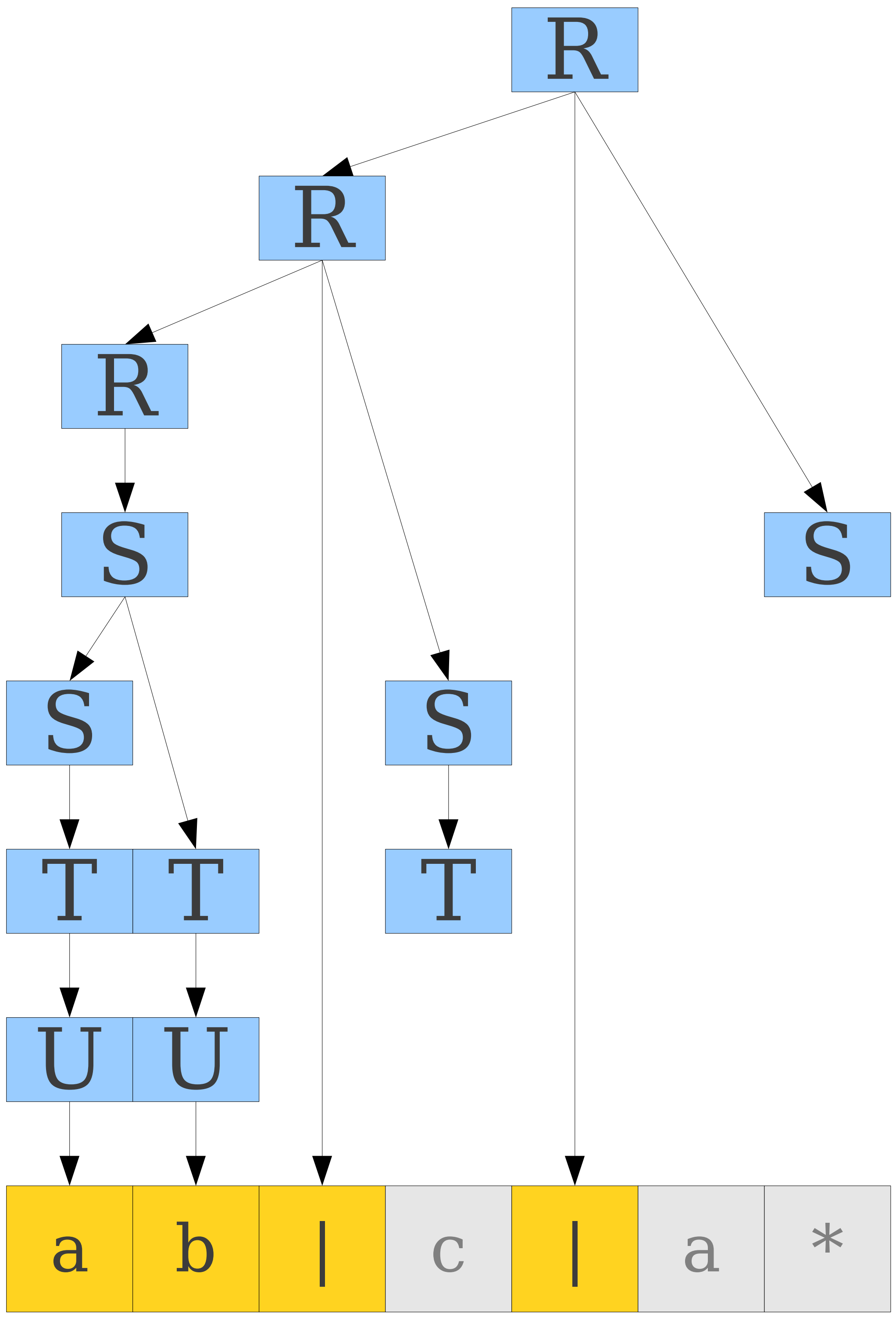
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → “ε”

U → (**R**)



R → **S** | **R** “|” **S**

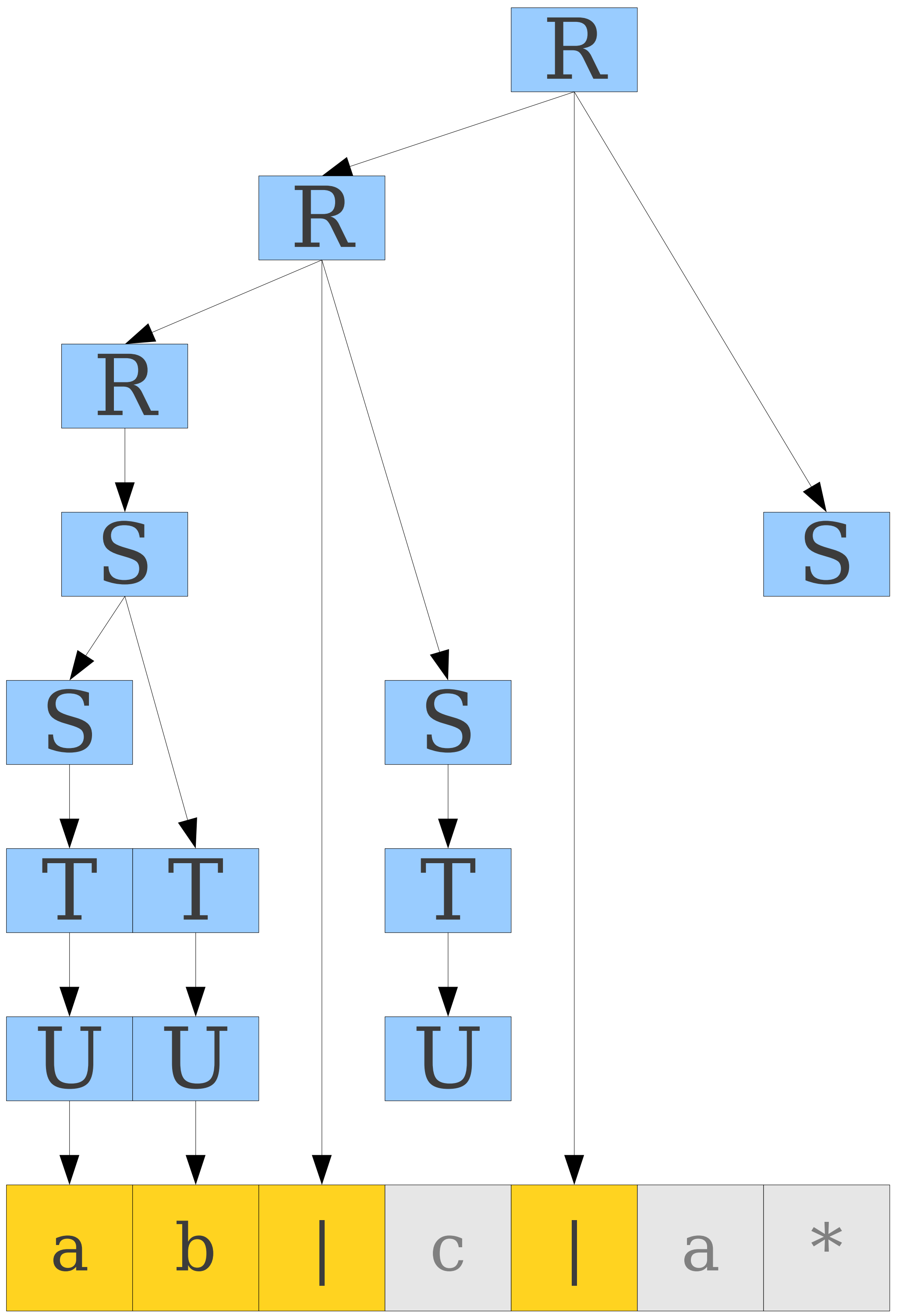
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → “ε”

U → (**R**)



R → S | R "|" S

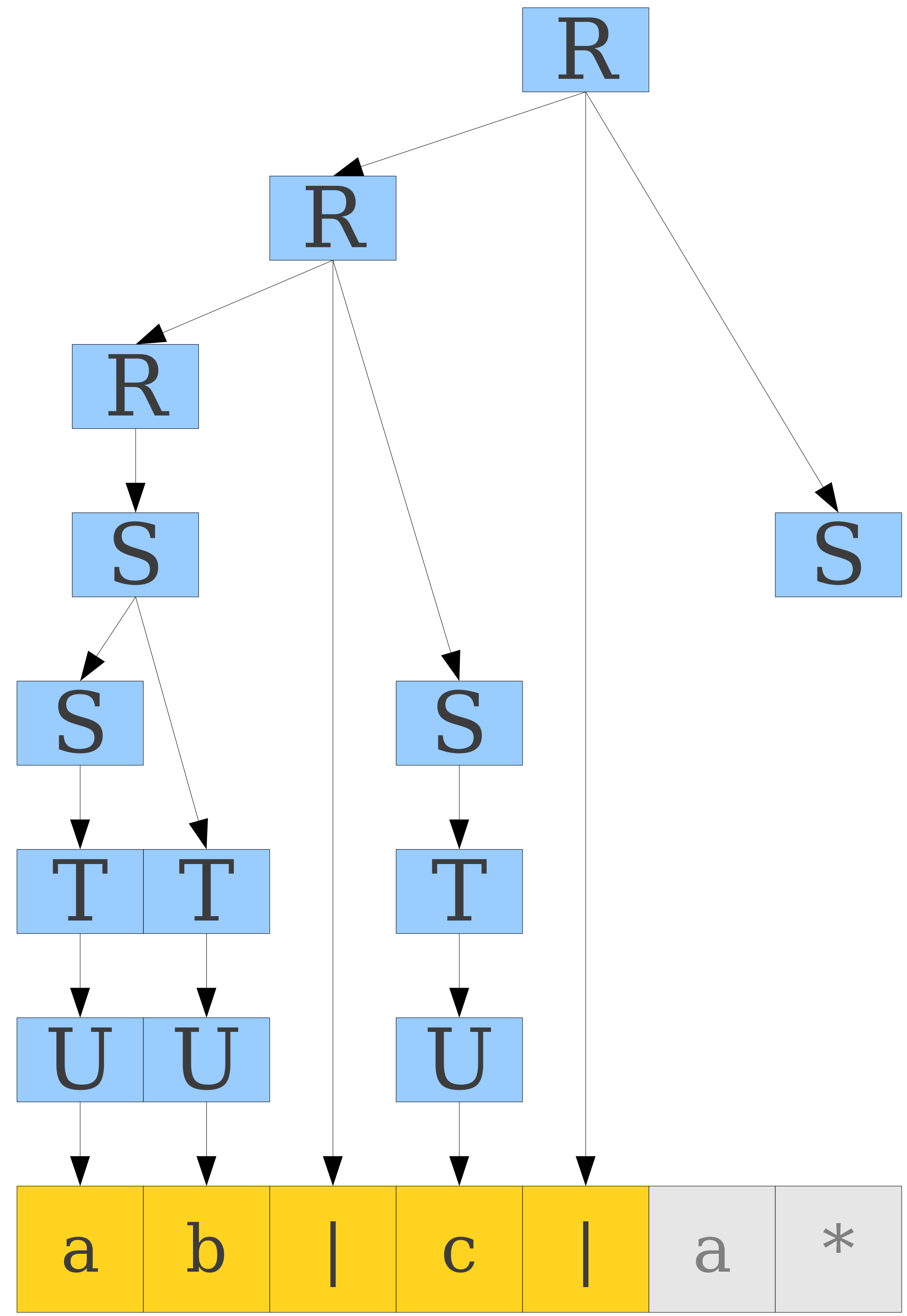
S → T | ST

T → U | T*

U → a | b | c | …

U → "ε"

U → (R)



R → **S** | **R** "|" **S**

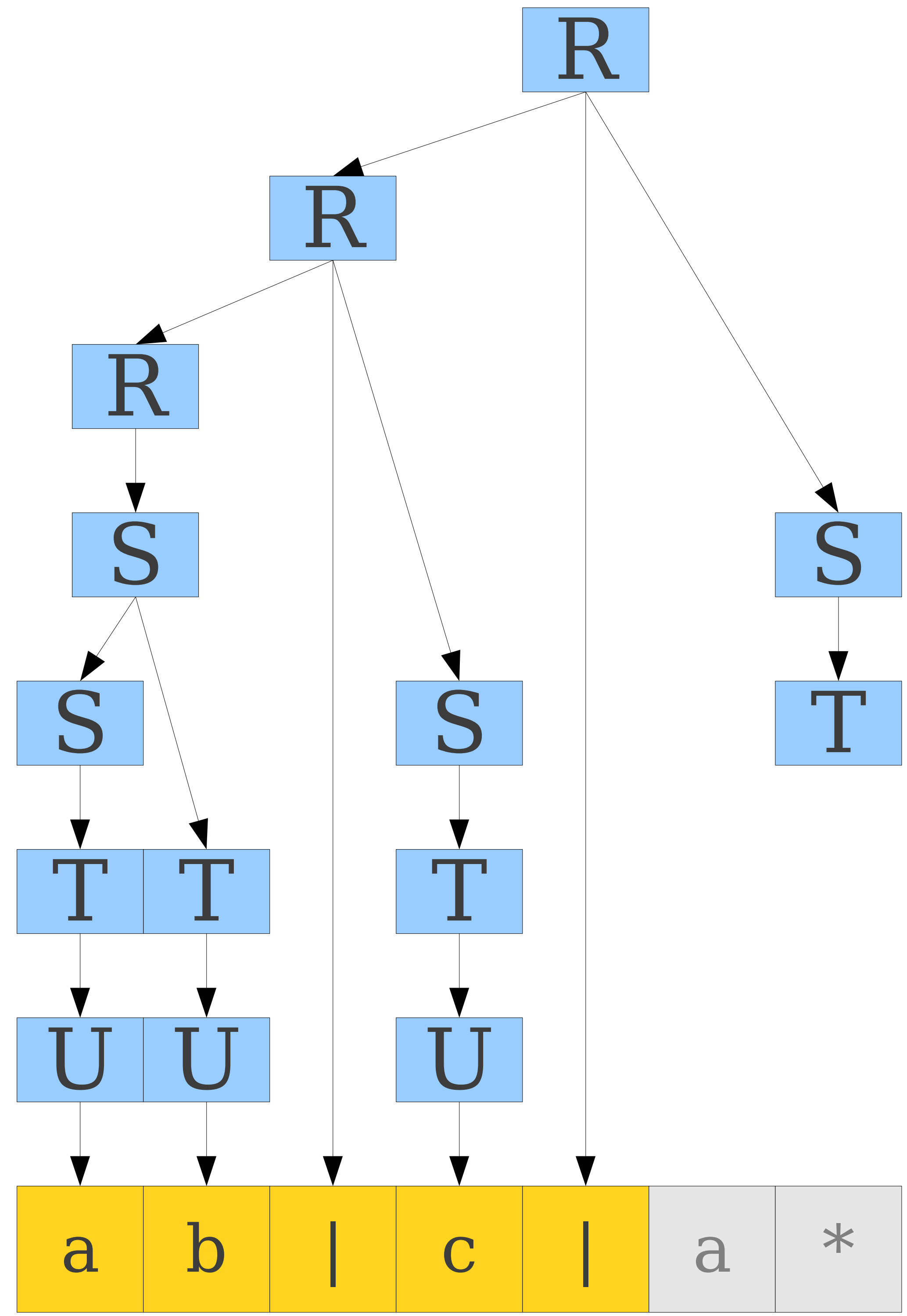
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | …

U → "ε"

U → (**R**)



R → **S** | **R** “|” **S**

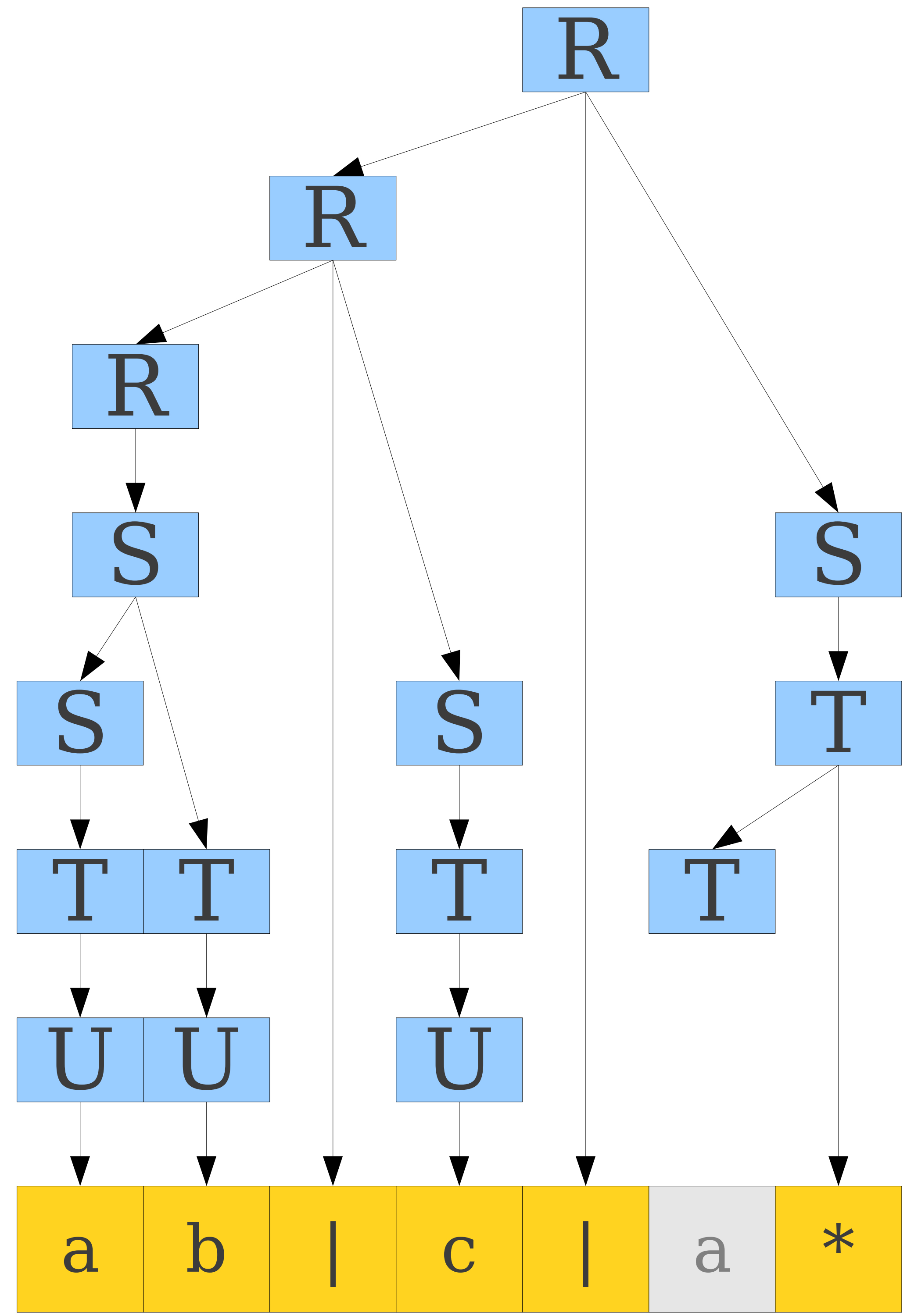
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | …

U → “ε”

U → (**R**)



R → S | R "|" S

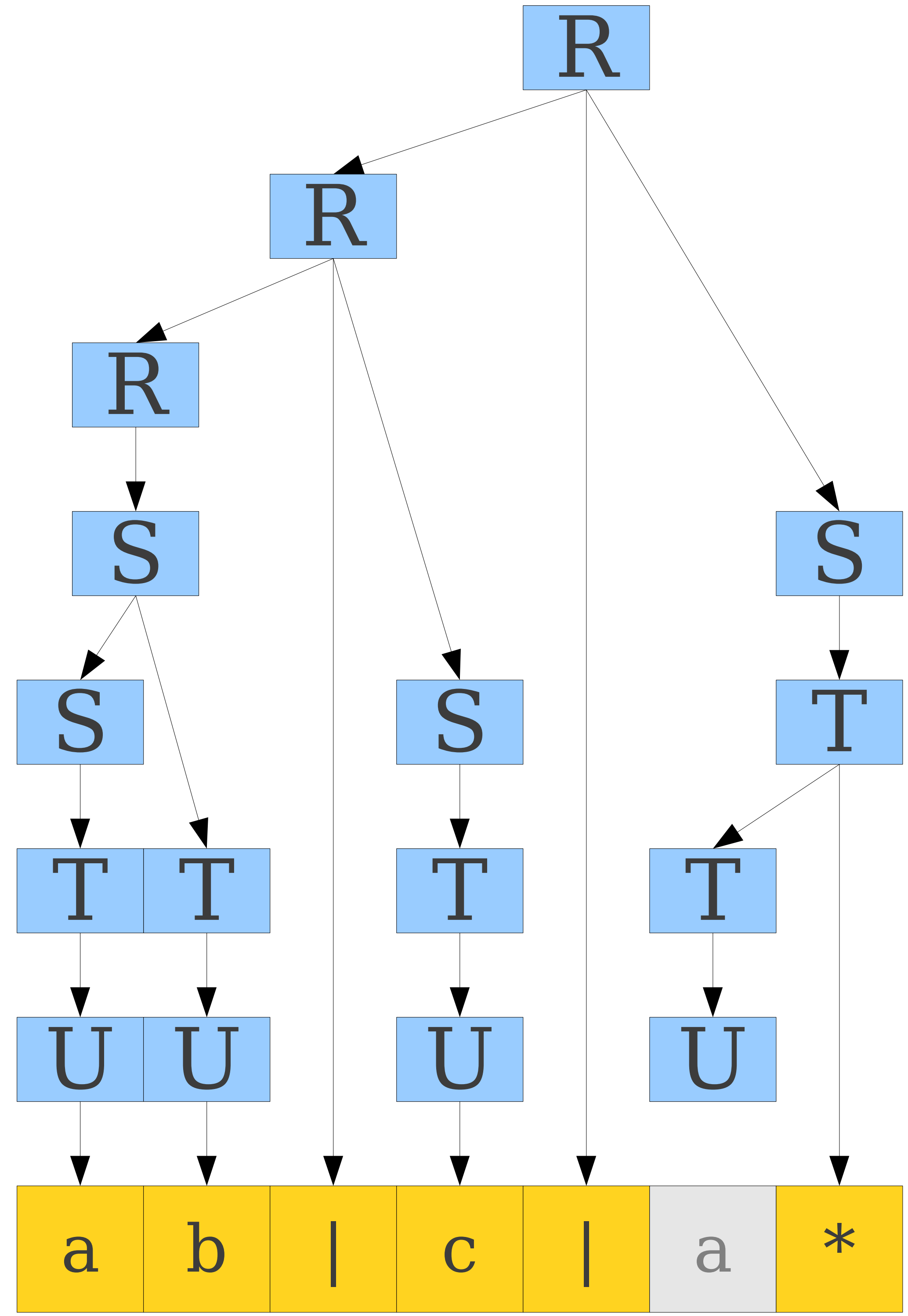
S → T | ST

T → U | T*

U → a | b | c | …

U → "ε"

U → (R)



R → S | R "|" S

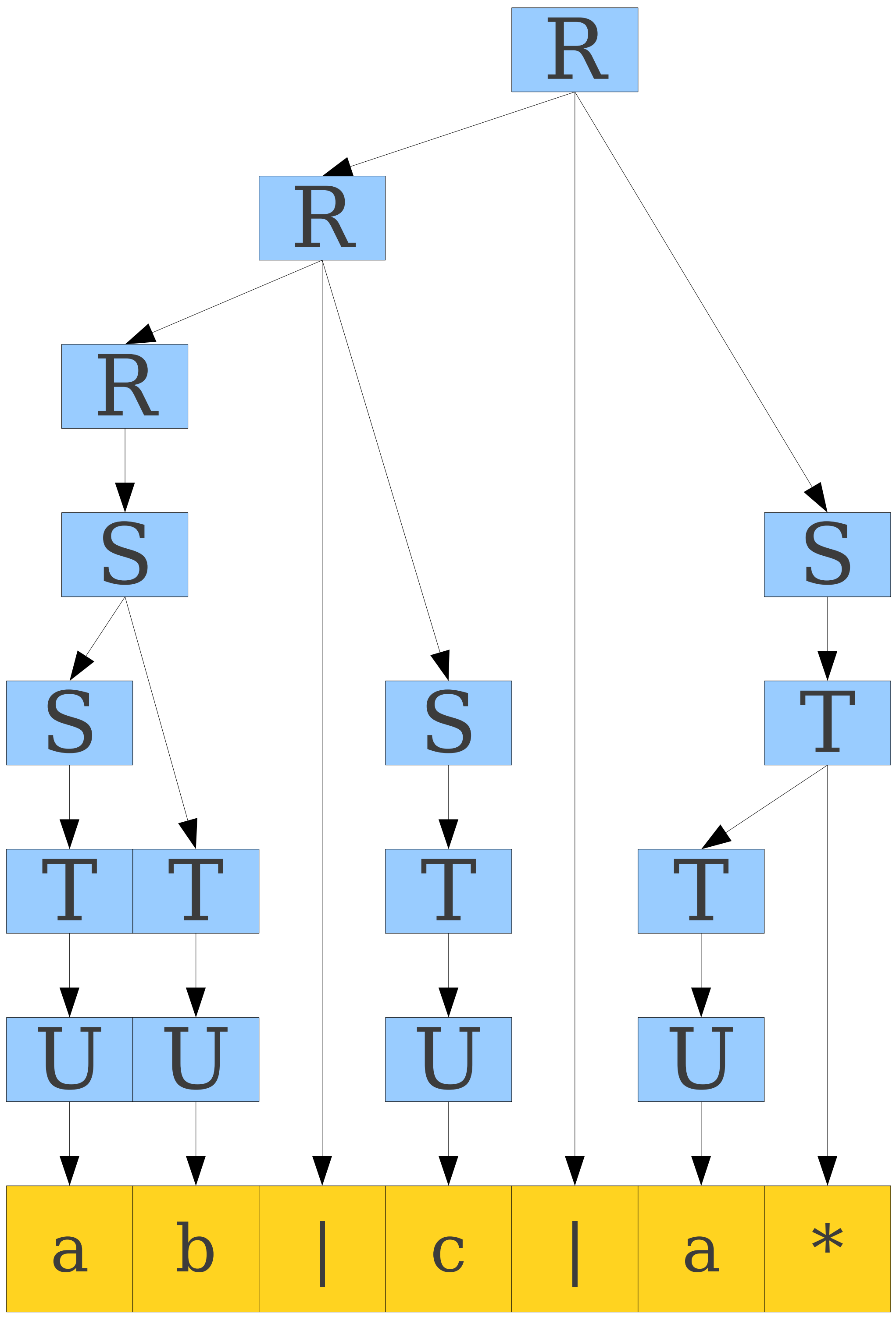
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



Precedence Declarations

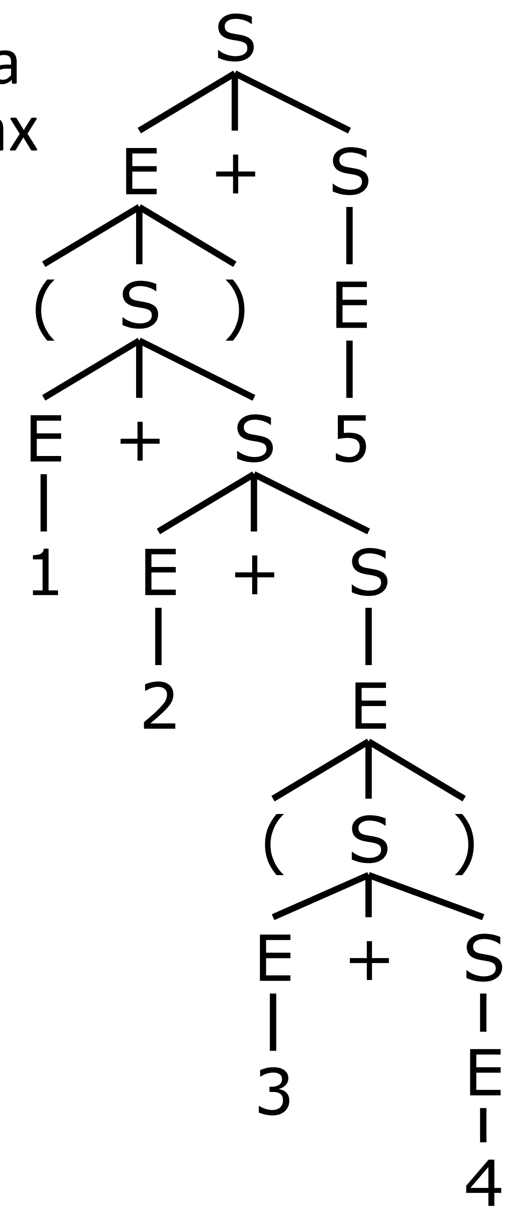
- If we leave the world of pure CFGs, we can often resolve ambiguities through **precedence declarations**.
  - e.g. multiplication has higher precedence than addition, but lower precedence than exponentiation.
- Allows for unambiguous parsing of ambiguous grammars.
- We'll see how this is implemented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

Abstract Syntax Trees (ASTs)

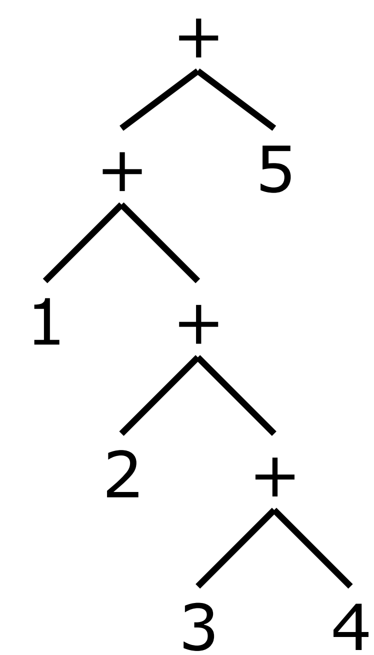
- A parse tree is a **concrete syntax tree**; it shows exactly how the text was derived.
- A more useful structure is an **abstract syntax tree**, which retains only the essential structure of the input.

Parse Tree vs. AST

- Parse Tree, aka concrete syntax



Abstract Syntax Tree



Discards/abstracts unneeded information

The Structure of a Parse Tree

R → **S** | **R** “|” **S**

S → **T** | **ST**

T → **U** | **T***

U → a | b | …

U → “ε”

U → (**R**)

a a | b *

The Structure of a Parse Tree

R → **S** | **R** "|" **S**

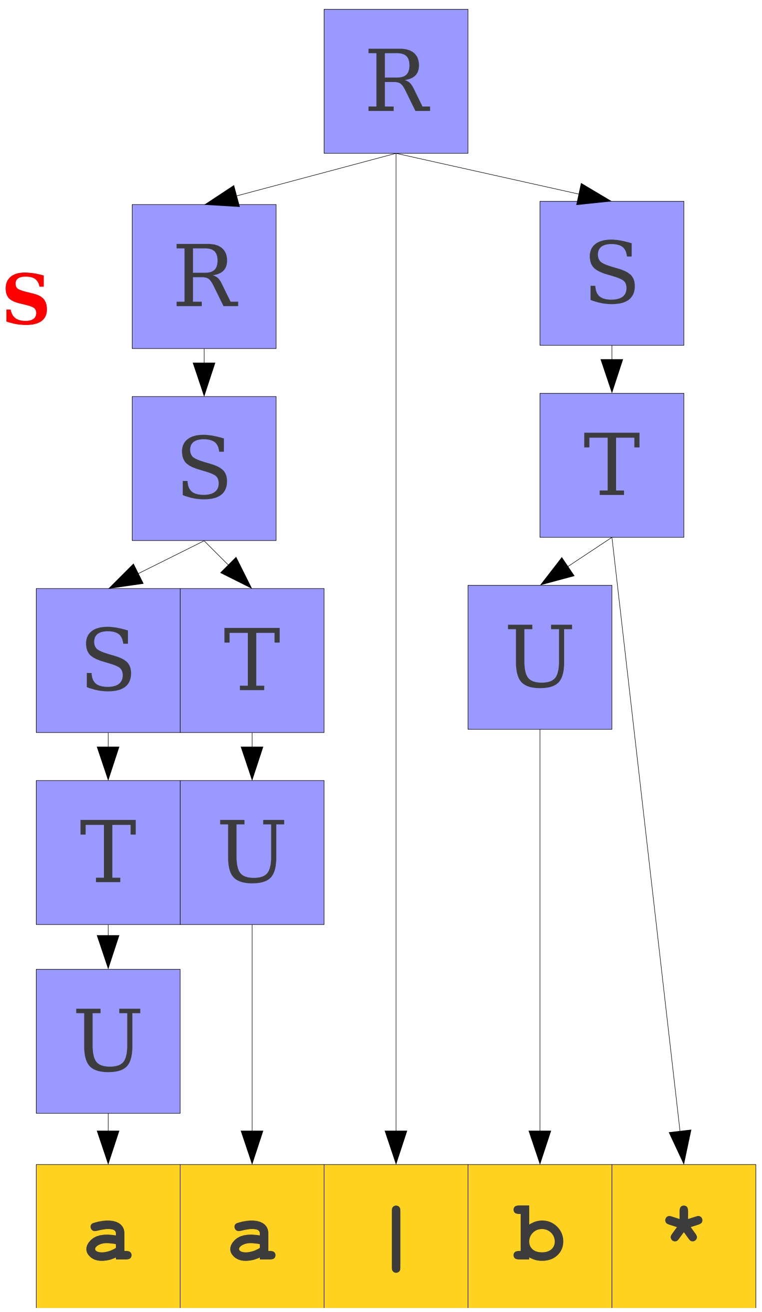
S → **T** | **ST**

T → **U** | **T*****

U → a | b | …

U → "ε"

U → (**R**)



The Structure of a Parse Tree

R → **S** | **R** "|" **S**

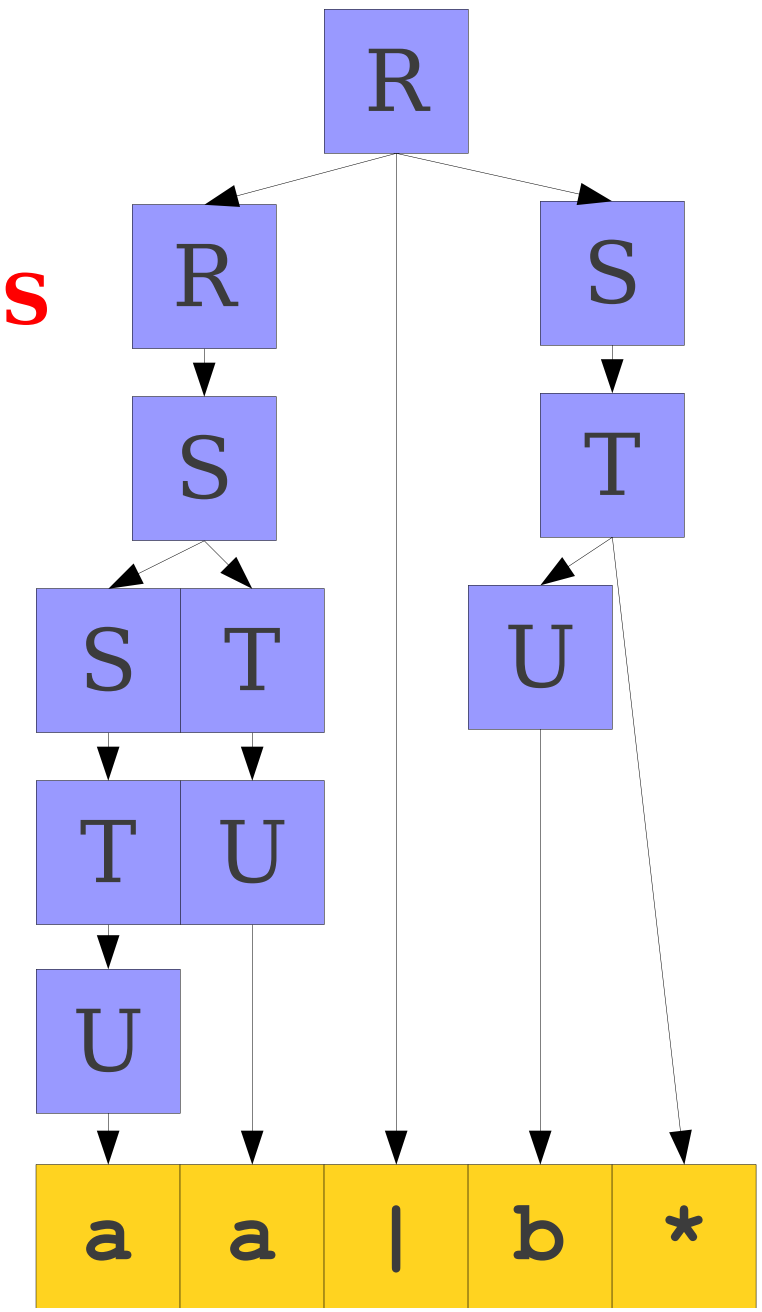
S → **T** | **ST**

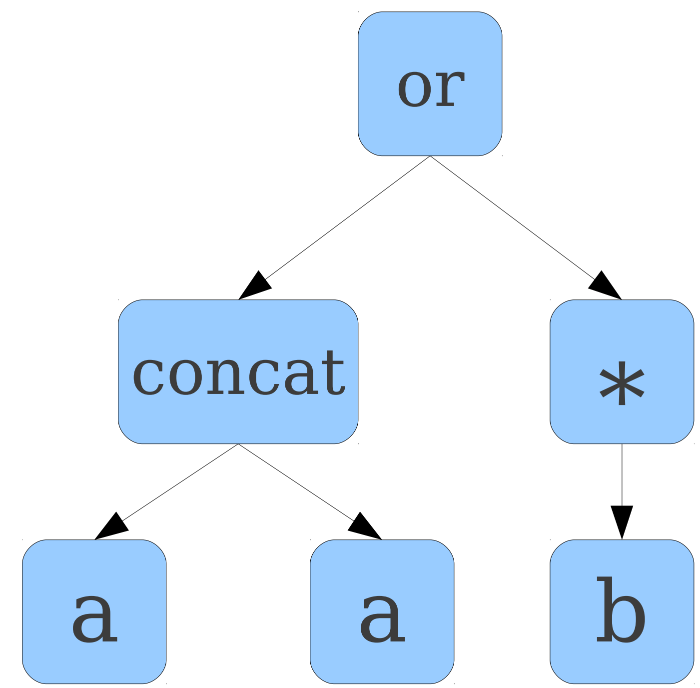
T → **U** | **T***

U → a | b | …

U → "ε"

U → (**R**)





R → **S** | **R** “|” **S**

S → **T** | **ST**

T → **U** | **T***

U → a | b | …

U → “ε”

U → (**R**)

a	(	b			c	)

R → **S** | **R** “|” **S**

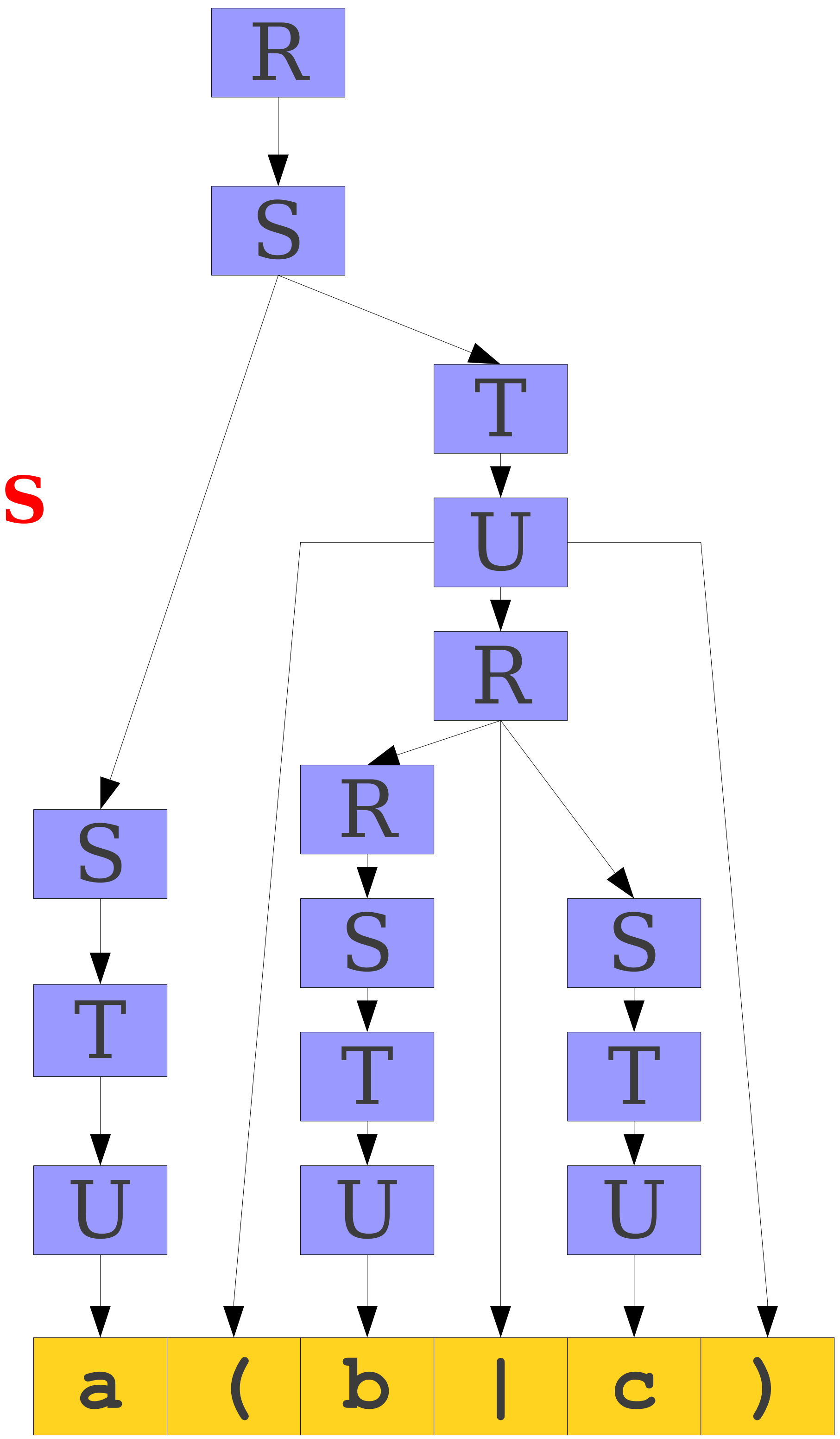
S → **T** | **ST**

T → **U** | **T***

U → a | b | …

U → “ε”

U → (**R**)



R → **S** | **R** "|" **S**

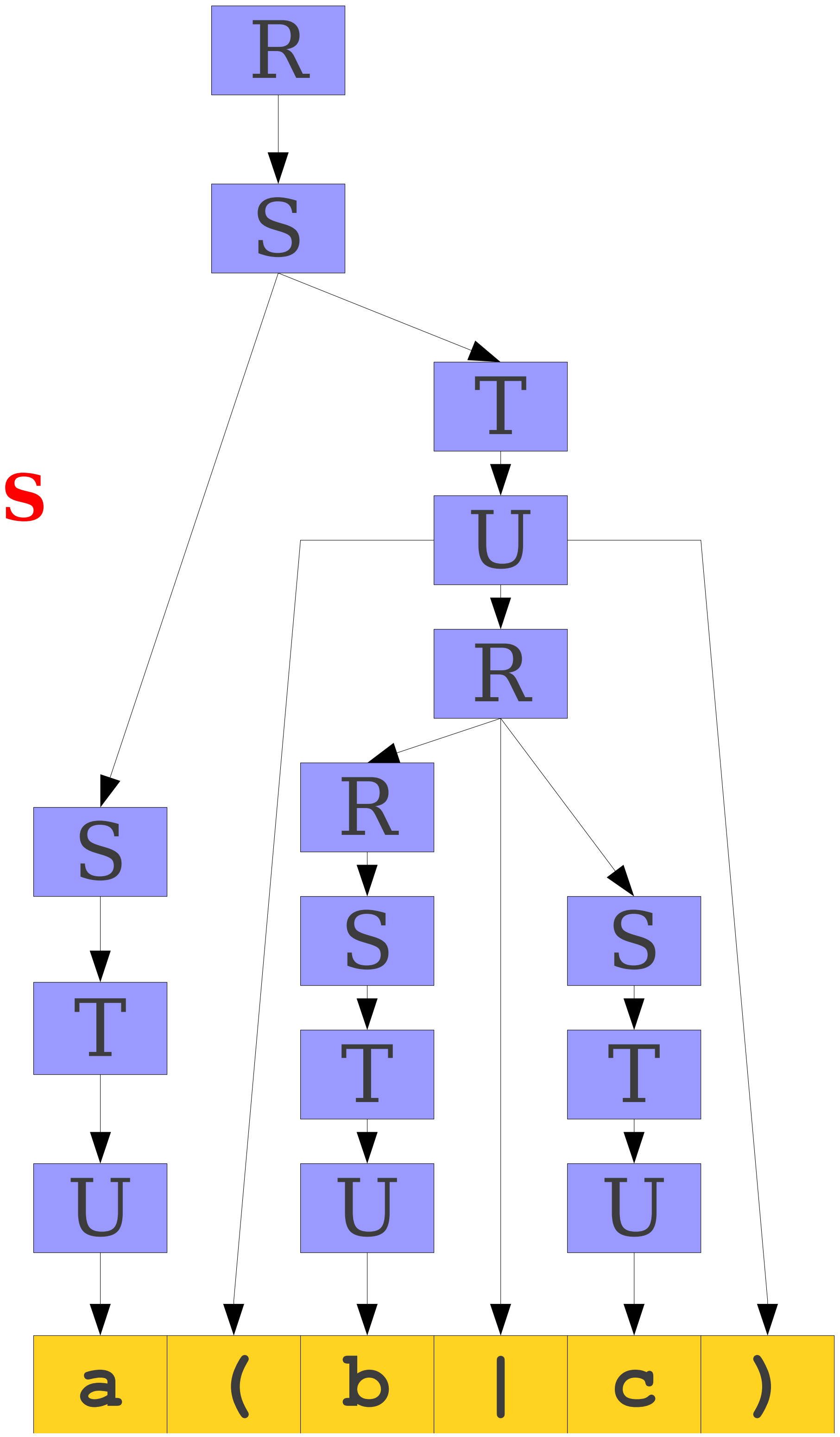
S → **T** | **ST**

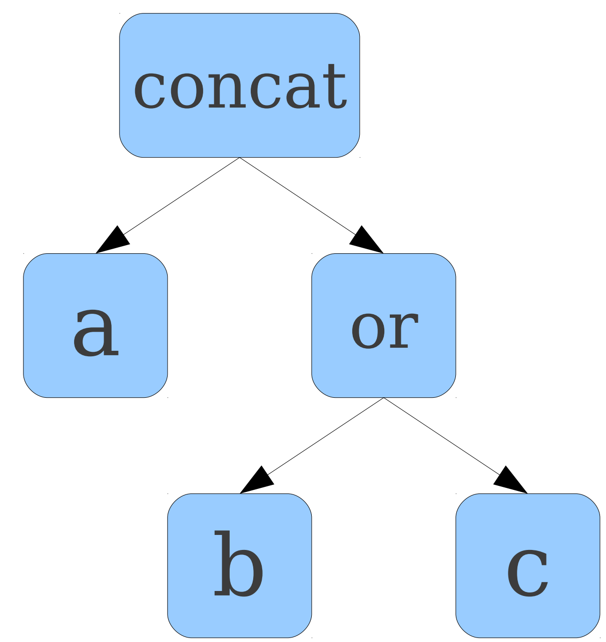
T → **U** | **T**`*`

U → `a` | `b` | …

U → "ε"

U → `(`**R**`)`





Summary

- Syntax analysis (**parsing**) extracts the structure from the tokens produced by the scanner.
- Languages are usually specified by **context-free grammars** (**CFG**s).
- 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** (**AST**s) contain an abstract representation of a program's syntax.
- **Semantic actions** associated with productions can be used to build ASTs.