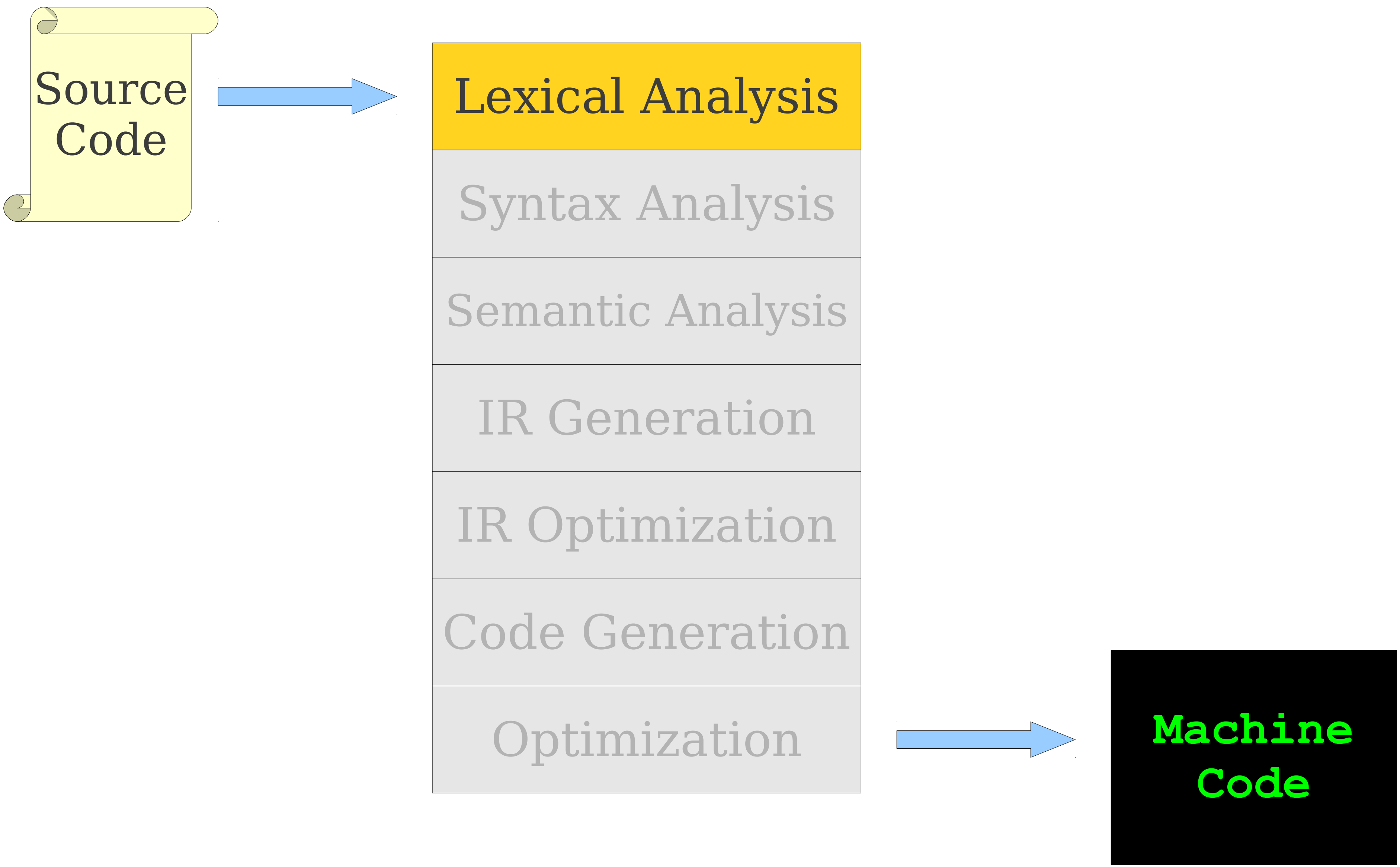
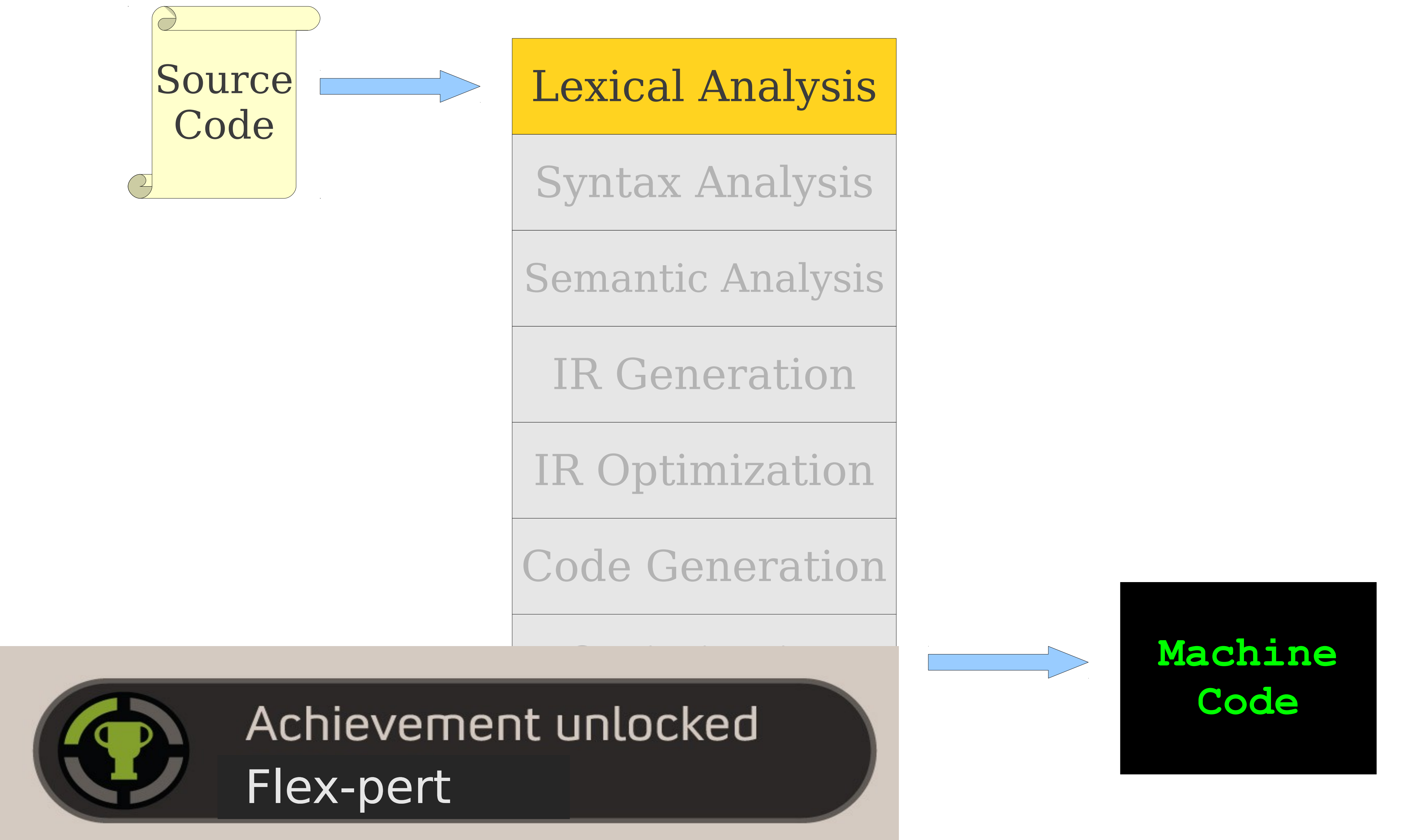
Syntax Analysis

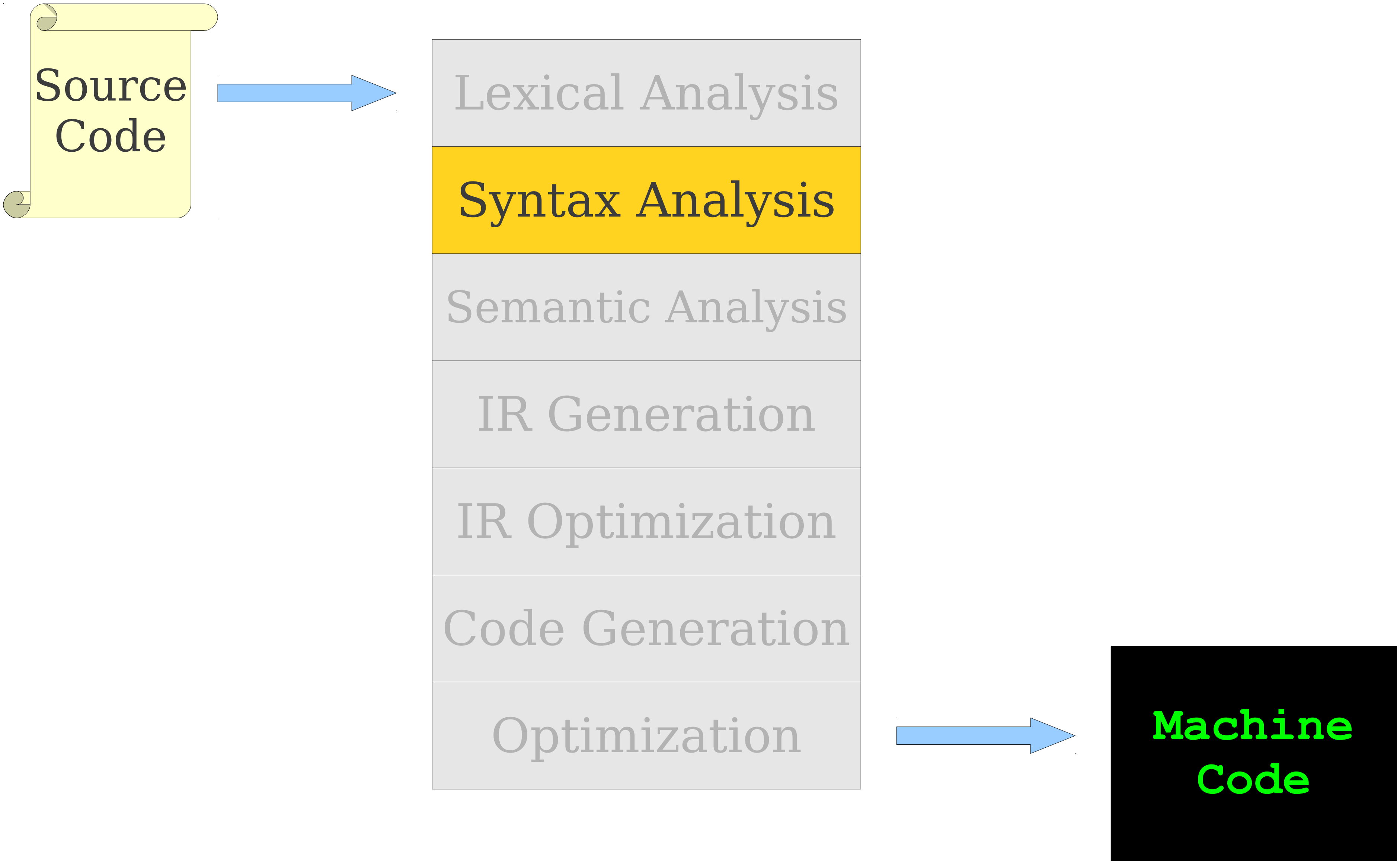
Where We Are


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
Machine Code
Achievement unlocked
Flex-pert

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.

Outline

- Today: Formalisms for syntax analysis.
  - Context-Free Grammars
  - Derivations
  - Concrete and Abstract Syntax Trees
  - Ambiguity
- Next Week: Parsing algorithms.
  - Top-Down Parsing
  - Bottom-Up Parsing

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.
- CFGs are best explained by 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)`

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`

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

E → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

A Notational Shorthand

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

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

Not Notational Shorthand

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

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

Not Notational Shorthand

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

$$\mathbf{S} \to \mathbf{A}\mathtt{b}$$

Not Notational Shorthand

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

$$\mathbf{S} \rightarrow \mathbf{A}\mathtt{b}$$

$$\mathbf{A} \rightarrow \mathbf{A}\mathtt{a} \mid \varepsilon$$

Not Notational Shorthand

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

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

Not Notational Shorthand

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

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}$$

$$\mathbf{X} \rightarrow \mathtt{(b|c^*)}$$

Not Notational Shorthand

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

S → `a`**X**
X → `b` | `c*`

Not Notational Shorthand

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

$$\mathbf{S} \to \mathtt{a}\mathbf{X}$$
$$\mathbf{X} \to \mathtt{b} \mid \mathbf{C}$$

Not Notational Shorthand

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

$$\mathbf{S} \rightarrow \mathtt{a}\mathbf{X}$$
$$\mathbf{X} \rightarrow \mathtt{b} \mid \mathbf{C}$$
$$\mathbf{C} \rightarrow \mathbf{C}\mathtt{c} \mid \varepsilon$$

More Context-Free Grammars

- Chemicals!
  - $C_{19}H_{14}O_5S$
  - $Cu_3(CO_3)_2(OH)_2$
  - MnO_4^-
  - S^{2-}

Form → **Cmp** | **Cmp Ion**

Cmp → **Term** | **Term Num** | **Cmp Cmp**

Term → **Elem** | `(`**Cmp**`)`

Elem → `H` | `He` | `Li` | `Be` | `B` | `C` | …

Ion → `+` | `-` | **IonNum** `+` | **IonNum** `-`

IonNum → `2` | `3` | `4` | ...

Num → `1` | **IonNum**

CFGs for Chemistry

Form → **Cmp** | **Cmp Ion**

Cmp → **Term** | **Term Num** | **Cmp Cmp**

Term → **Elem** | `(`**Cmp**`)`

Elem → `H` | `He` | `Li` | `Be` | `B` | `C` | …

Ion → `+` | `-` | **IonNum** `+` | **IonNum** `-`

IonNum → `2` | `3` | `4` | ...

Num → `1` | **IonNum**

Form
⇒ **Cmp Ion**
⇒ **Cmp Cmp Ion**
⇒ **Cmp Term Num Ion**
⇒ **Term Term Num Ion**
⇒ **Elem Term Num Ion**
⇒ `Mn` **Term Num Ion**
⇒ `Mn` **Elem Num Ion**
⇒ `MnO` **Num Ion**
⇒ `MnO` **IonNum Ion**
⇒ MnO_4 **Ion**
⇒ MnO_4^-

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
       | …
```

Some CFG Notation

- We will be discussing generic transformations and operations on CFGs over the next two weeks.
- Let's standardize our notation.

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

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

- 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

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

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 $\alpha A \omega$ **yields** string $\alpha \gamma \omega$ iff $A \to \gamma$ 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 $\alpha \Rightarrow^* \beta$.

Leftmost Derivations

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

```
  STMTS
⇒ STMT STMTS
⇒ EXPR; STMTS
⇒ EXPR = EXPR; STMTS
⇒ id = EXPR; STMTS
⇒ id = EXPR + EXPR; STMTS
⇒ id = id + EXPR; STMTS
⇒ id = id + constant; STMTS
⇒ id = id + constant;
```

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.
- These will be of great importance when we talk about parsing next week.

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.

Parse Trees

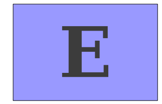
E

Parse Trees

E

E

Parse Trees



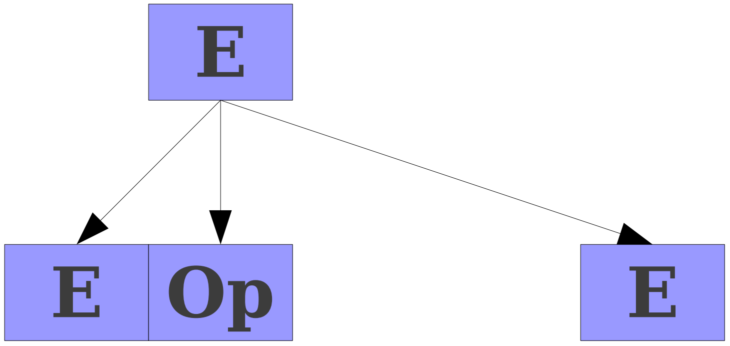
E

⇒ **E Op E**

Parse Trees

E

⇒ **E Op E**

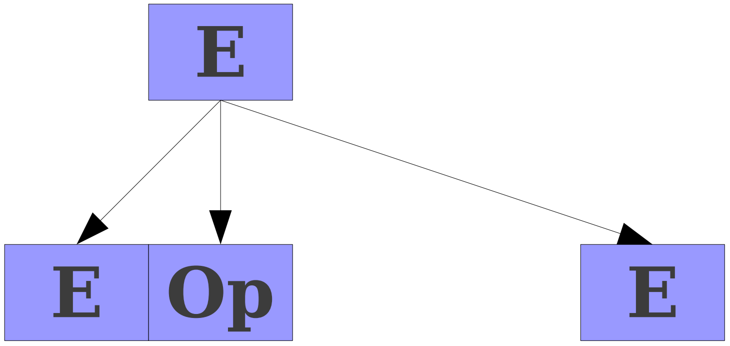


Parse Trees

E

⇒ E Op E

⇒ `int` Op E

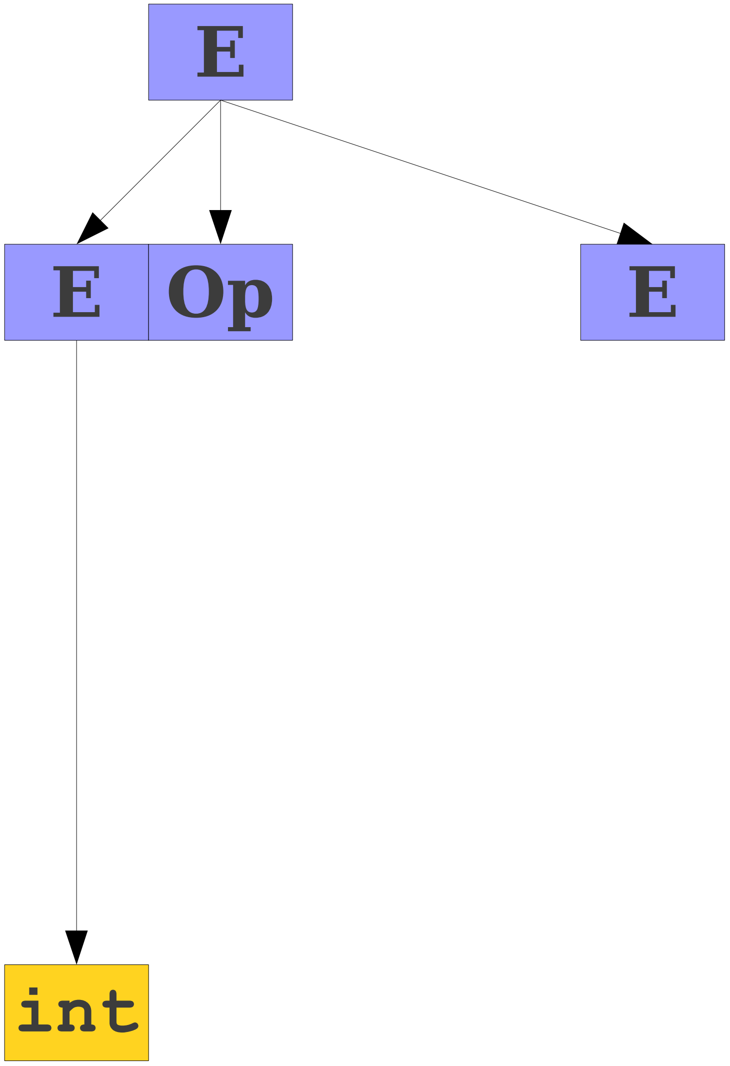


Parse Trees

E

⇒ **E Op E**

⇒ `int` **Op E**



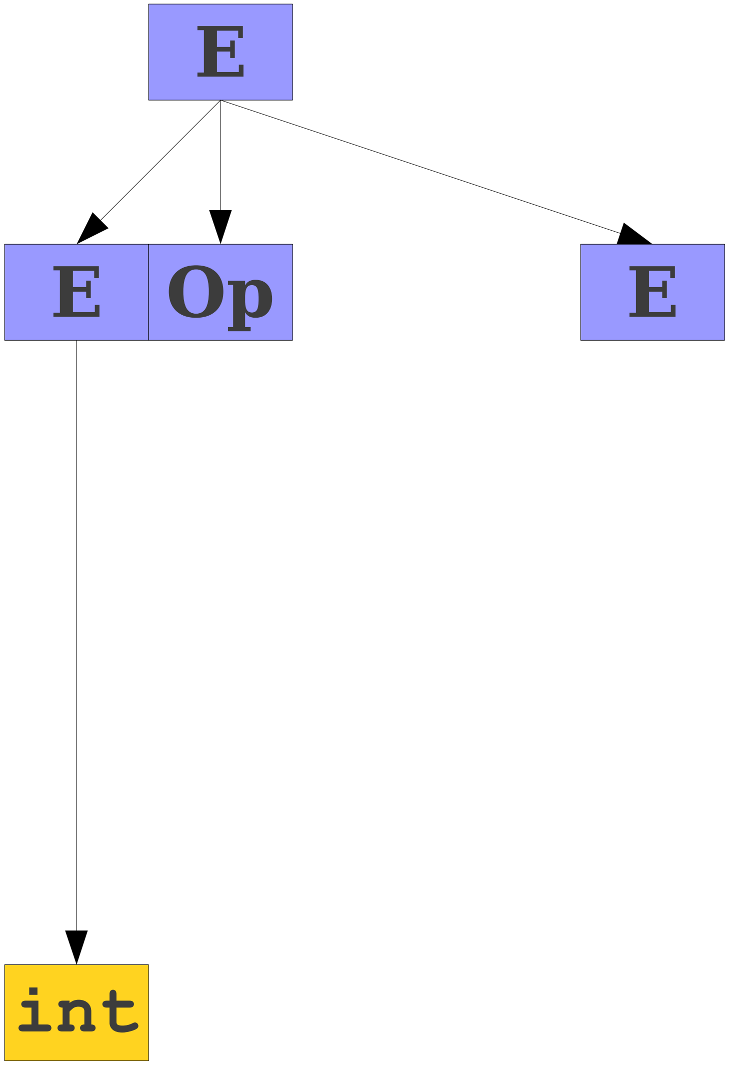
Parse Trees

E

⇒ E Op E

⇒ `int` Op E

⇒ `int * ` E



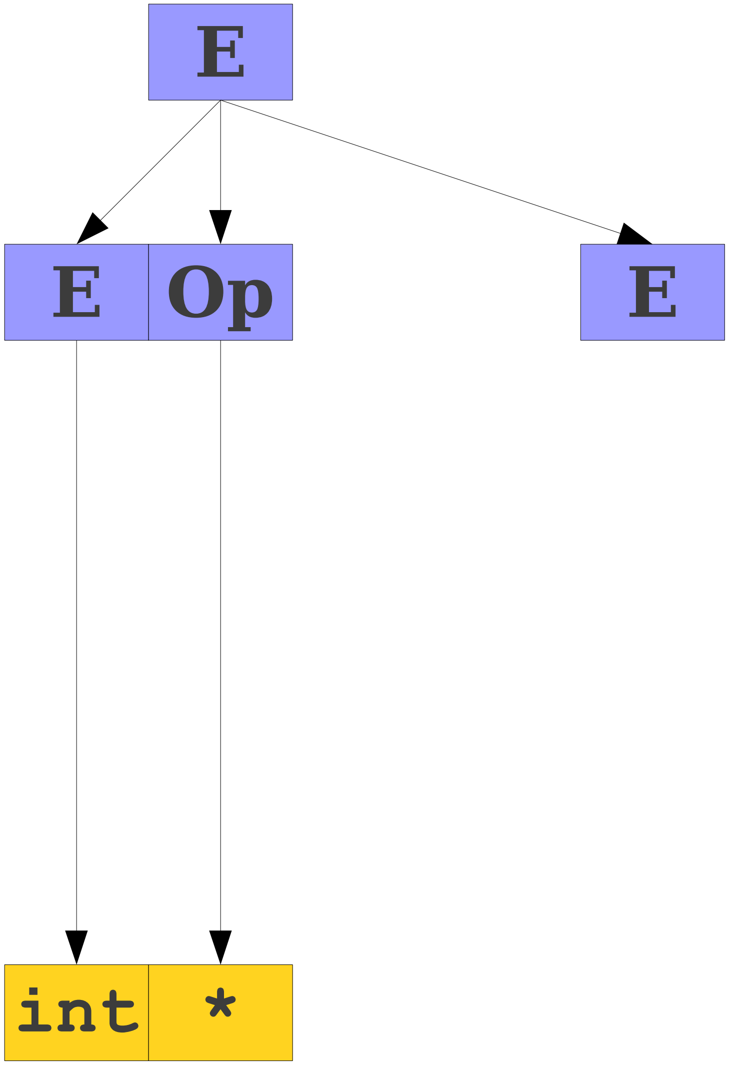
Parse Trees

E

⇒ **E Op E**

⇒ `int` **Op E**

⇒ `int * `**E**



Parse Trees

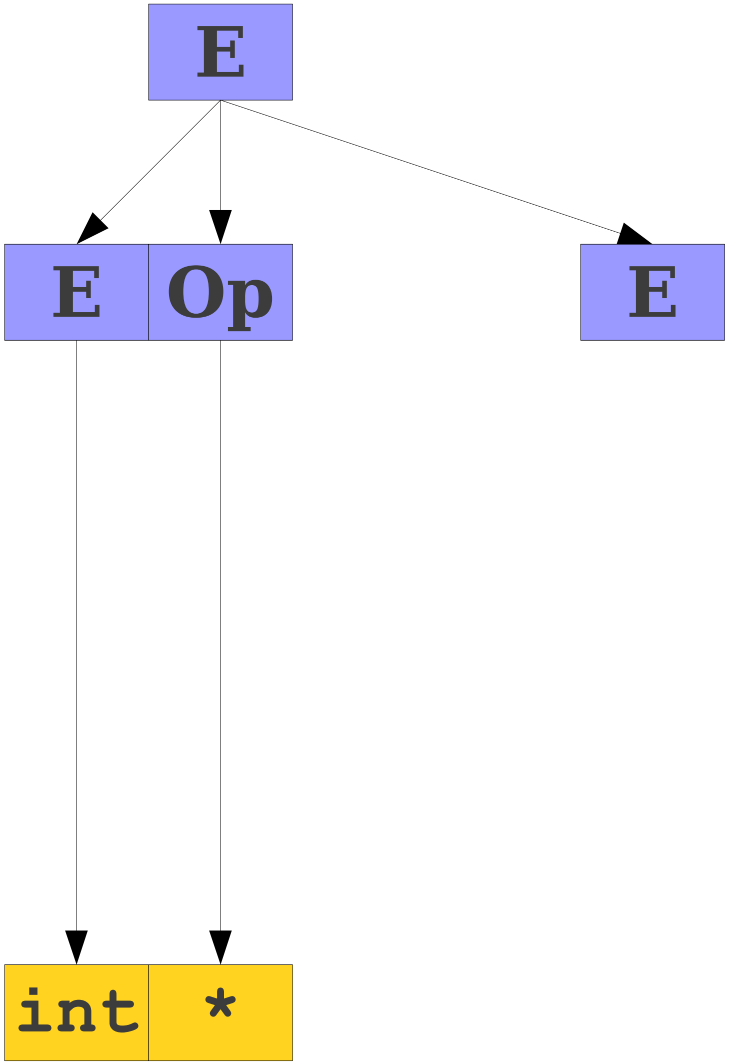
E

⇒ **E Op E**

⇒ `int` **Op E**

⇒ `int * `**E**

⇒ `int * (`**E**`)`



Parse Trees

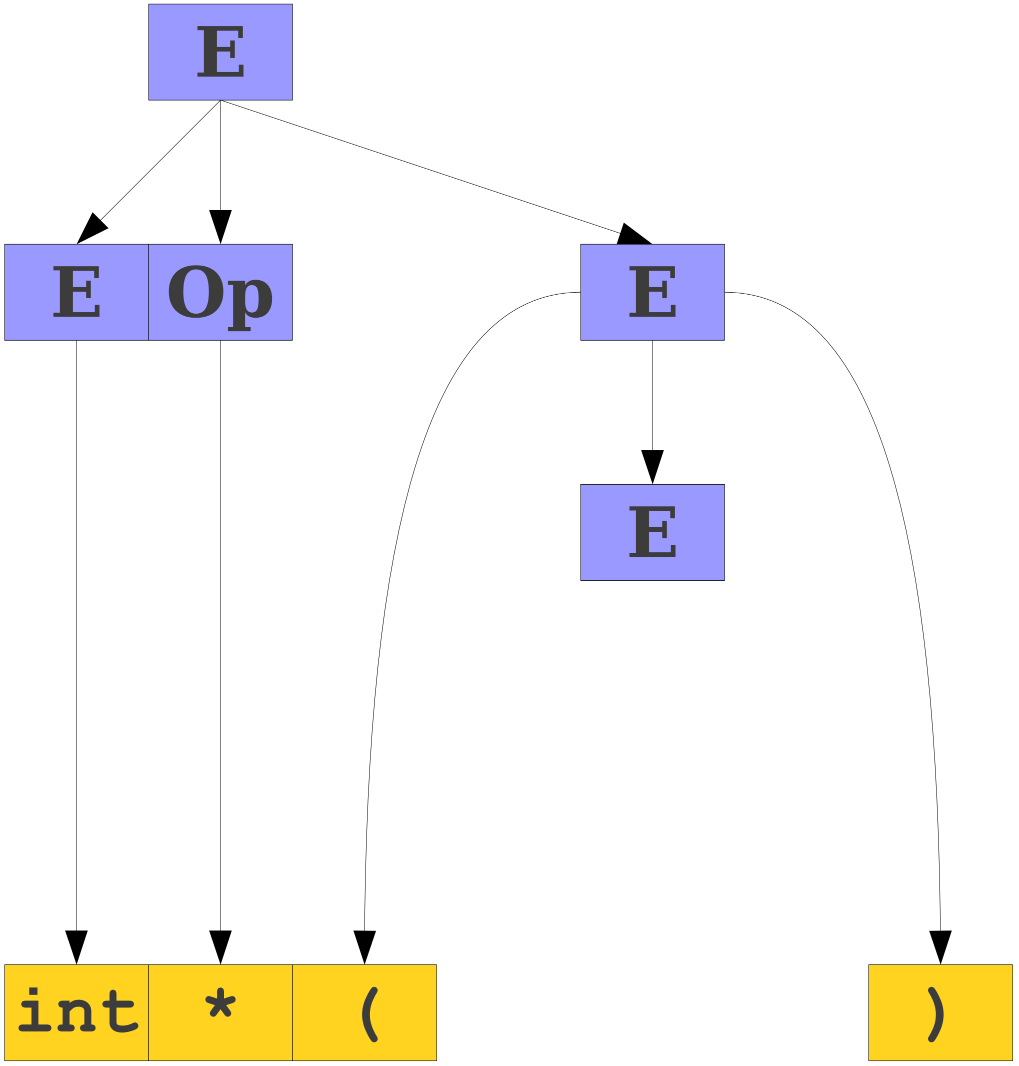
E

⇒ E Op E

⇒ `int` Op E

⇒ `int *` E

⇒ `int * (`E`)`



Parse Trees

E

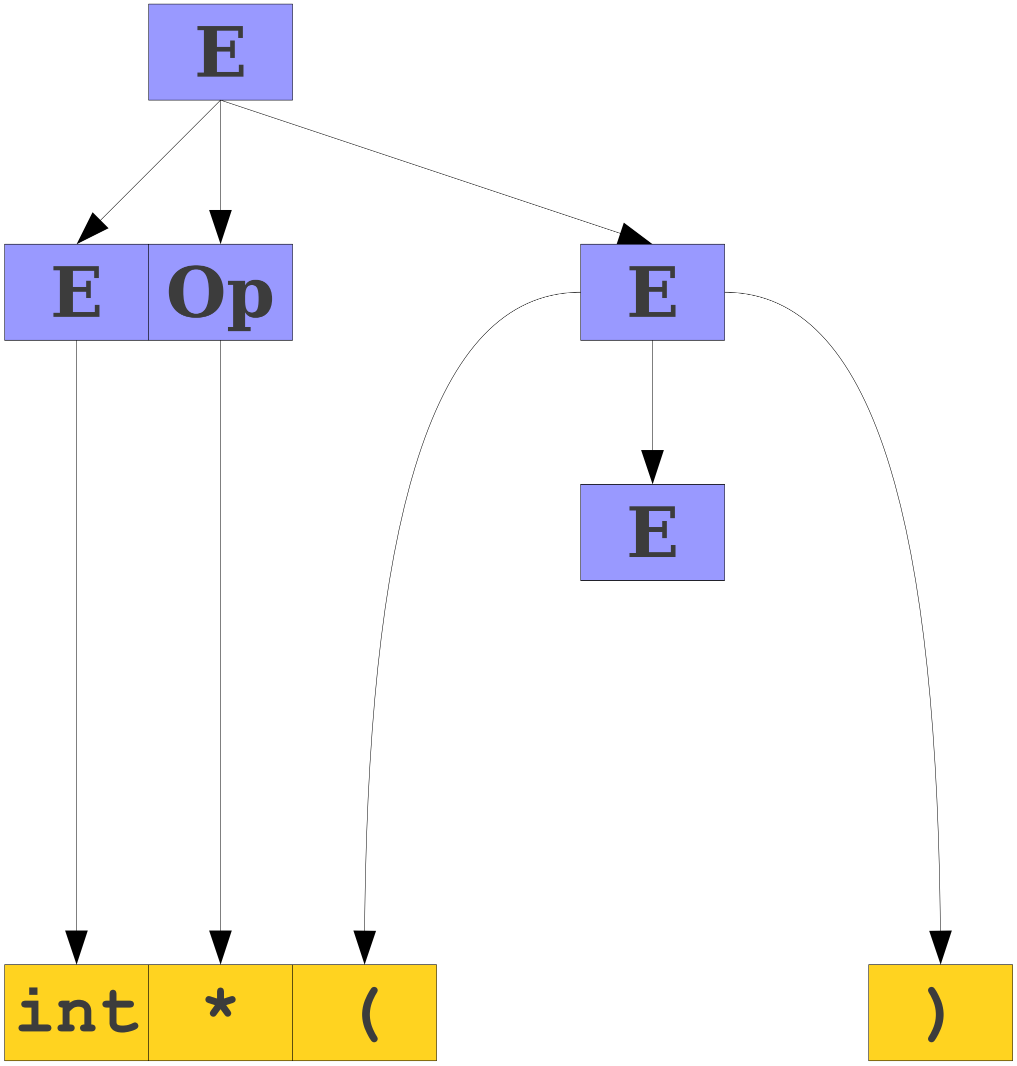
⇒ E Op E

⇒ int Op E

⇒ int * E

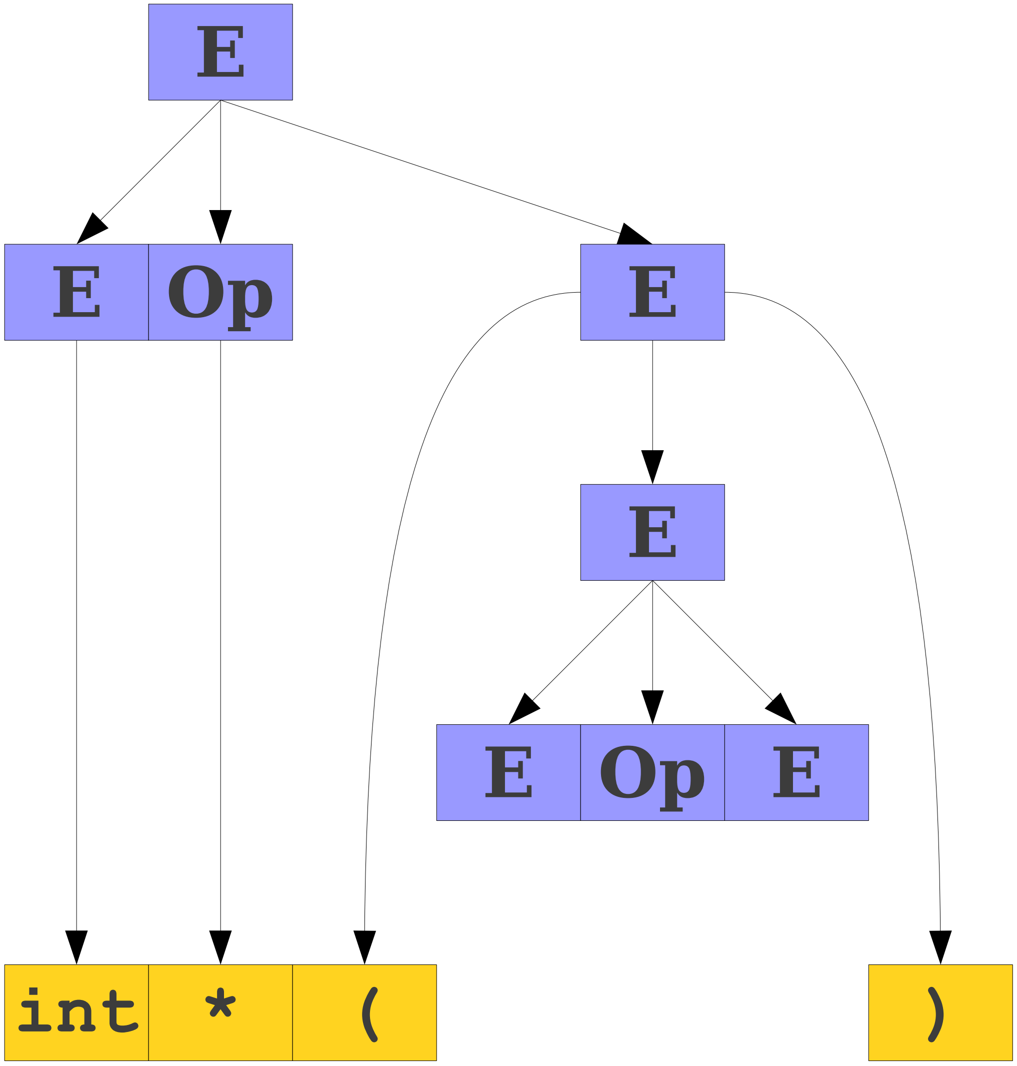
⇒ int * (E)

⇒ int * (E Op E)



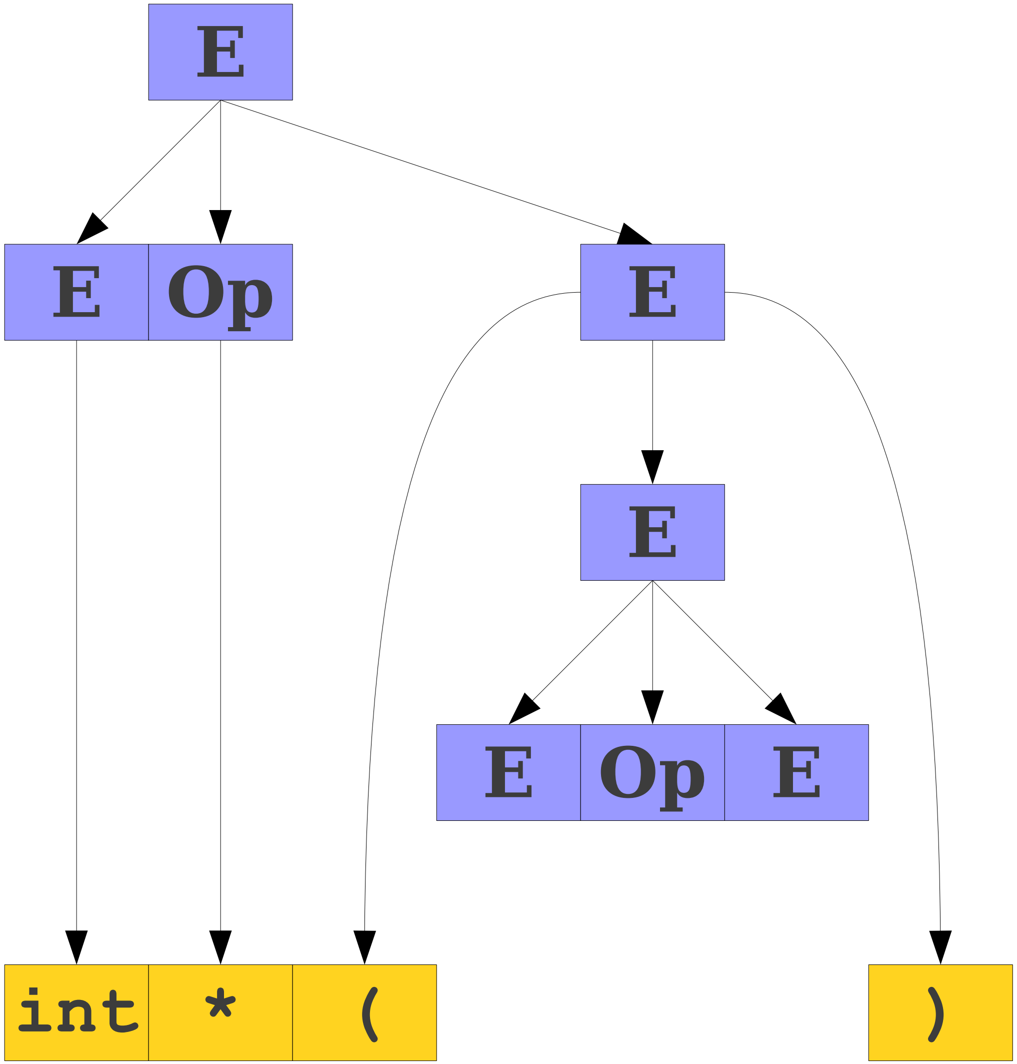
Parse Trees

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**

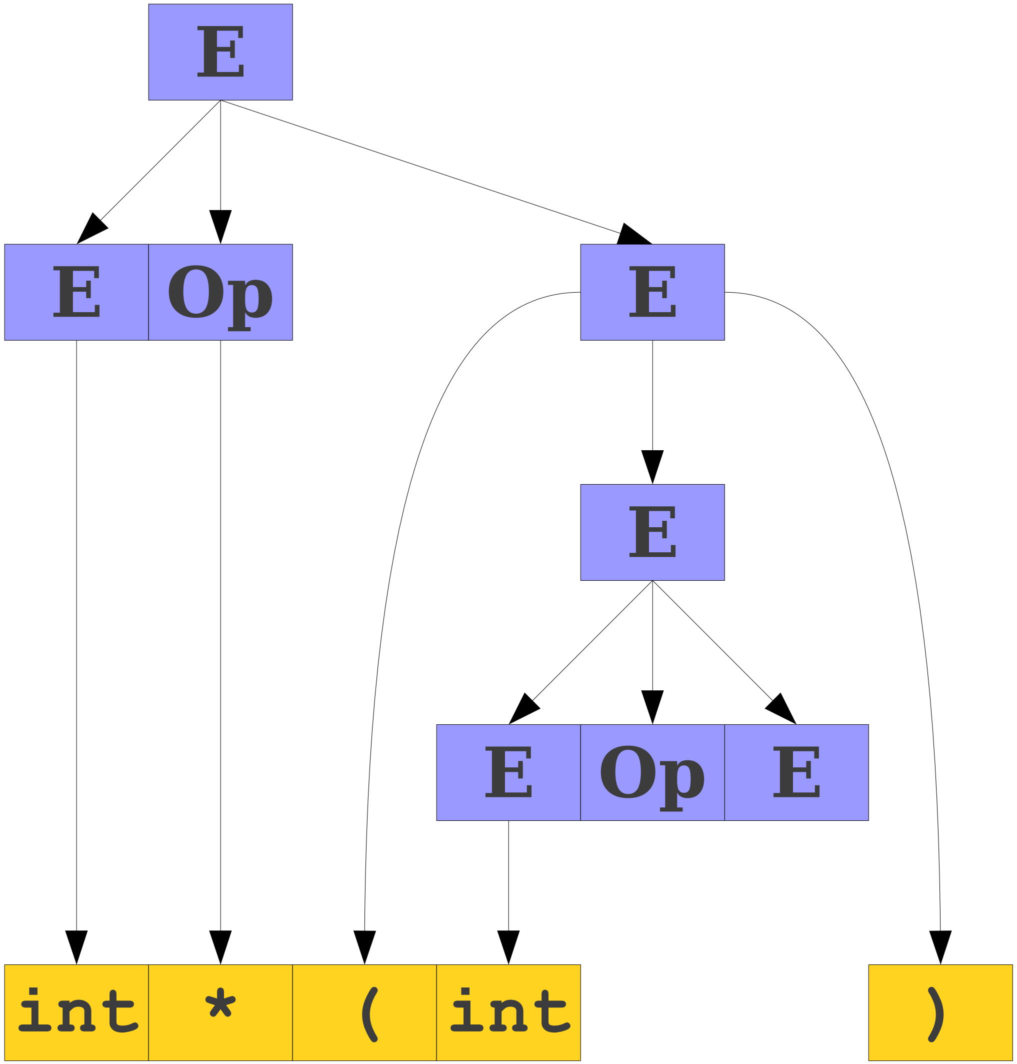
⇒ `int` **Op E**

⇒ `int *` **E**

⇒ `int * (`**E**`)`

⇒ `int * (`**E Op E**`)`

⇒ `int * (int` **Op E**`)`



Parse Trees

E

⇒ E Op E

⇒ int Op E

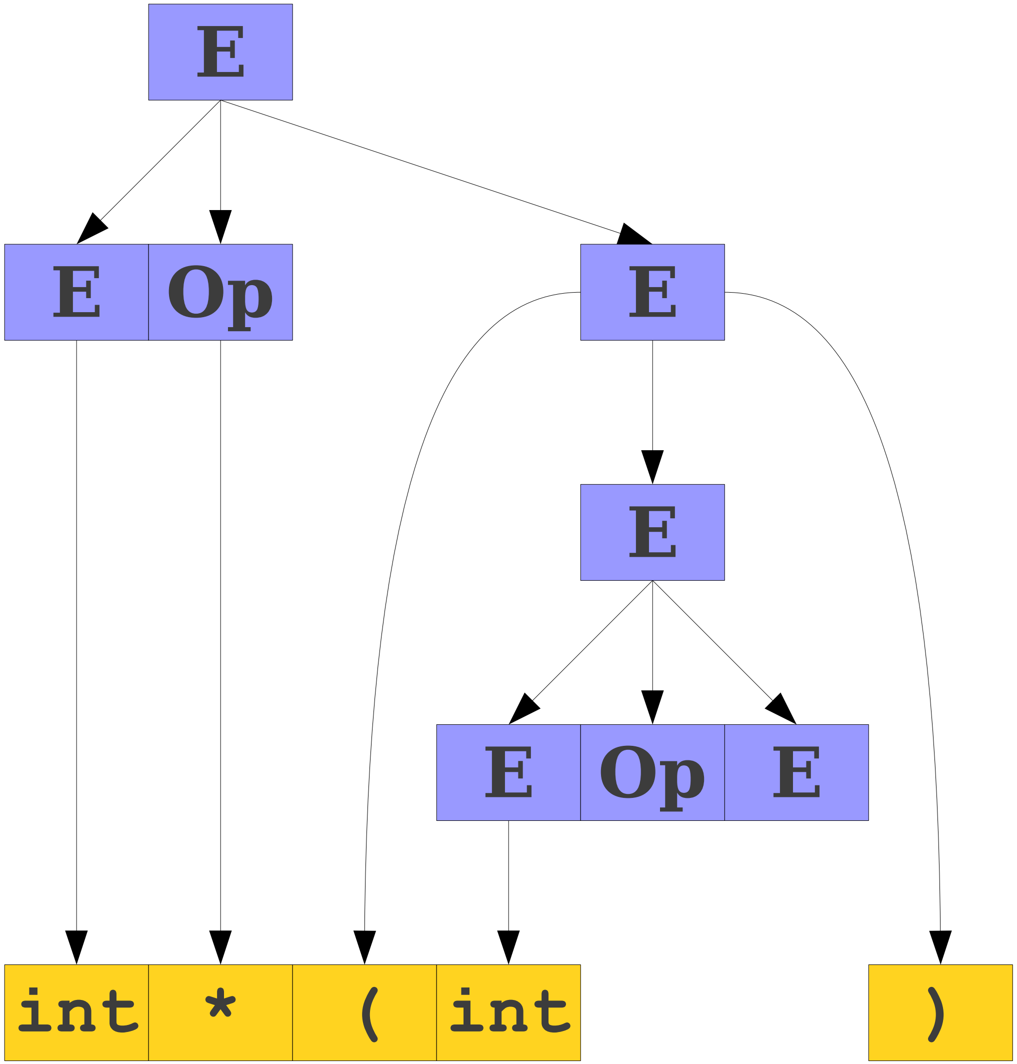
⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)



Parse Trees

E

⇒ E Op E

⇒ int Op E

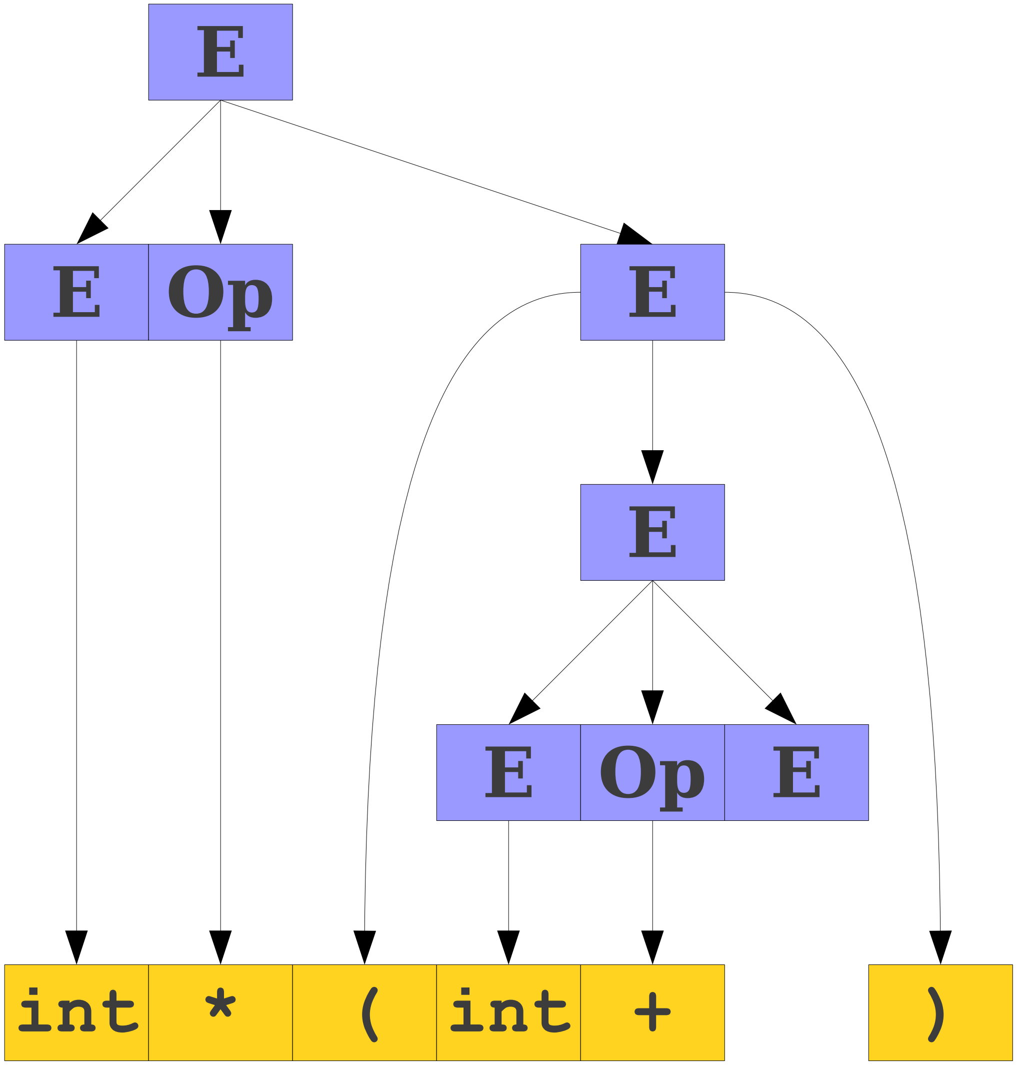
⇒ int * E

⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)



Parse Trees

E

⇒ E Op E

⇒ int Op E

⇒ int * E

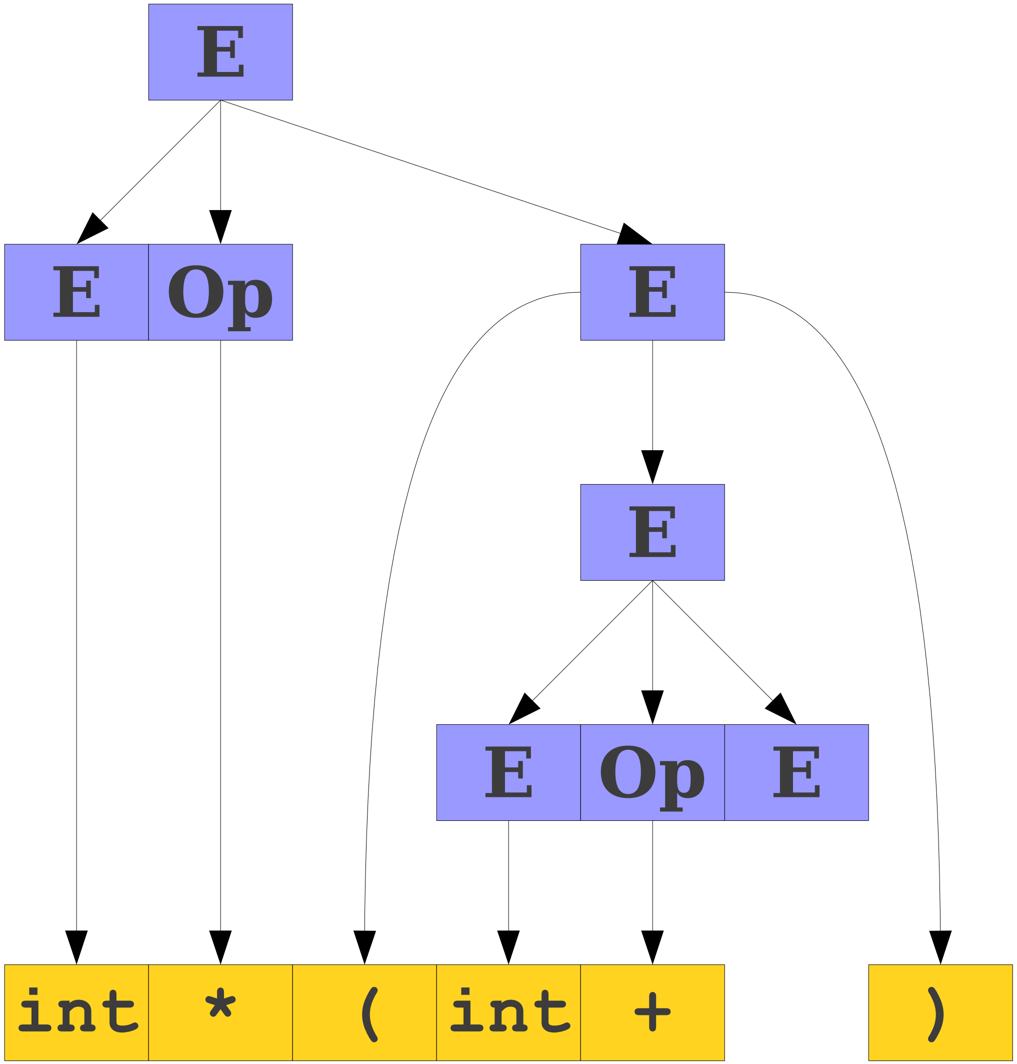
⇒ int * (E)

⇒ 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

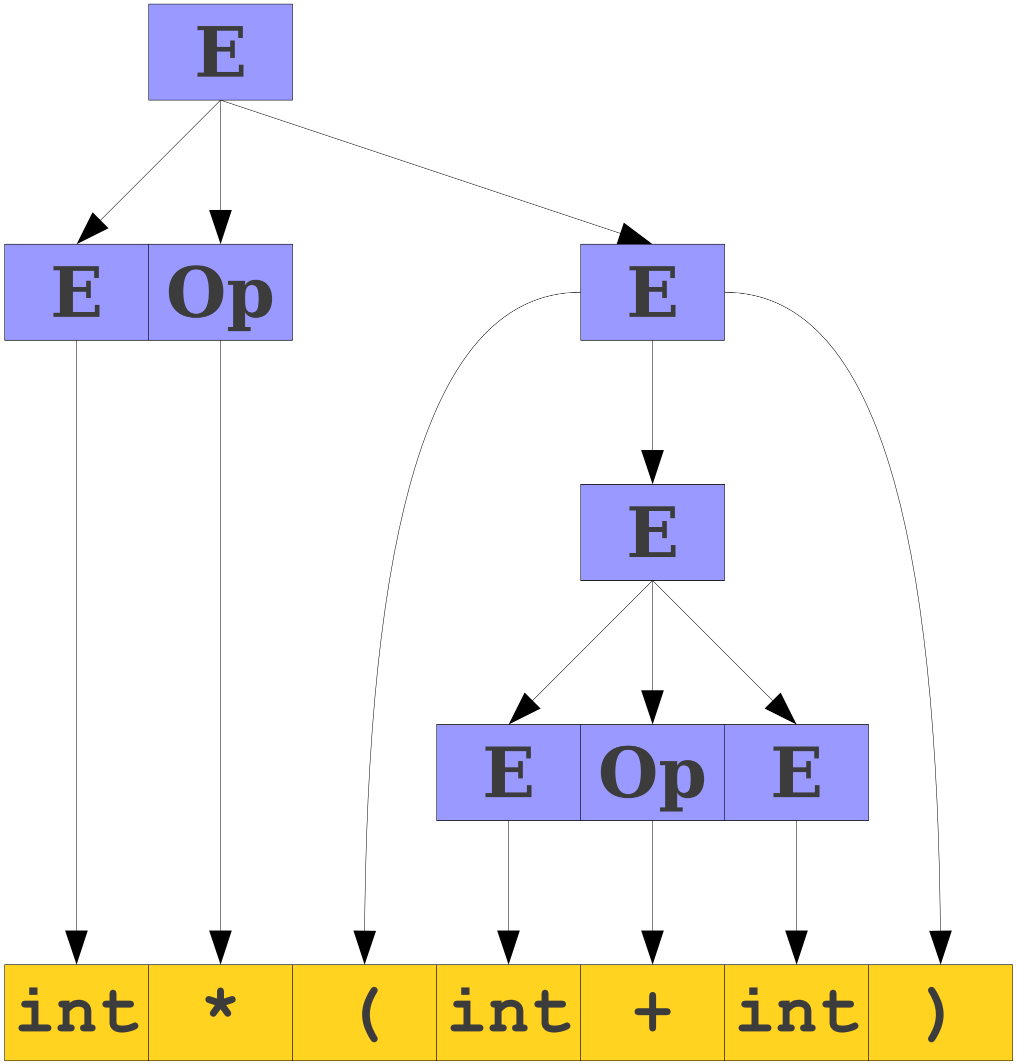
⇒ int * (E)

⇒ int * (E Op E)

⇒ int * (int Op E)

⇒ int * (int + E)

⇒ int * (int + int)



Parse Trees

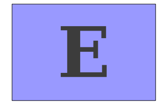
E

Parse Trees

E

E

Parse Trees



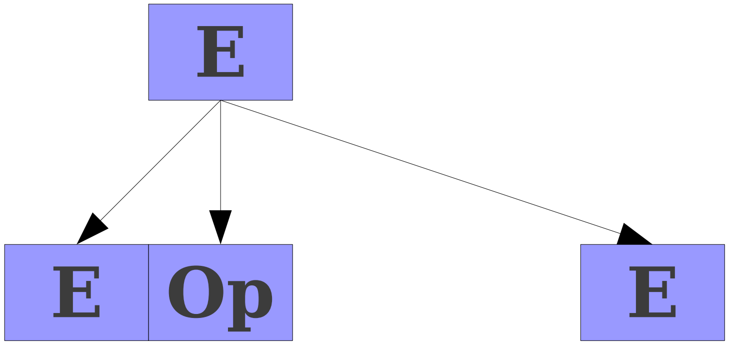
E

⇒ **E Op E**

Parse Trees

E

⇒ E Op E

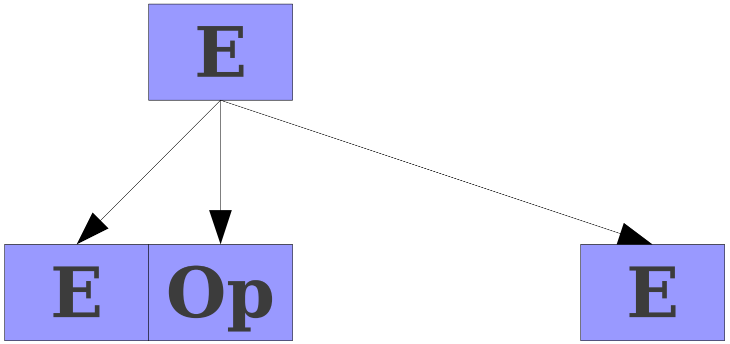


Parse Trees

E

⇒ **E Op E**

⇒ **E Op (E)**

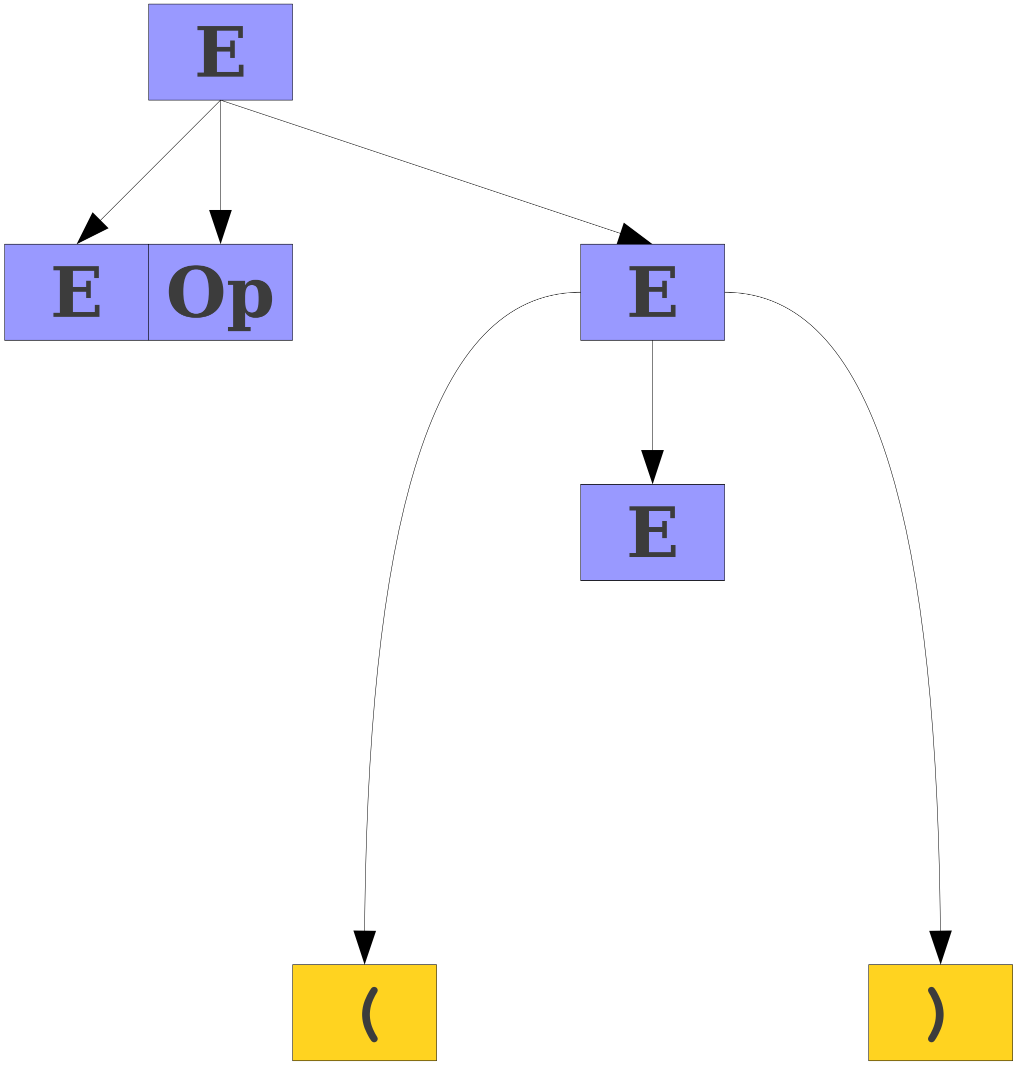


Parse Trees

E

⇒ **E Op E**

⇒ **E Op (E)**



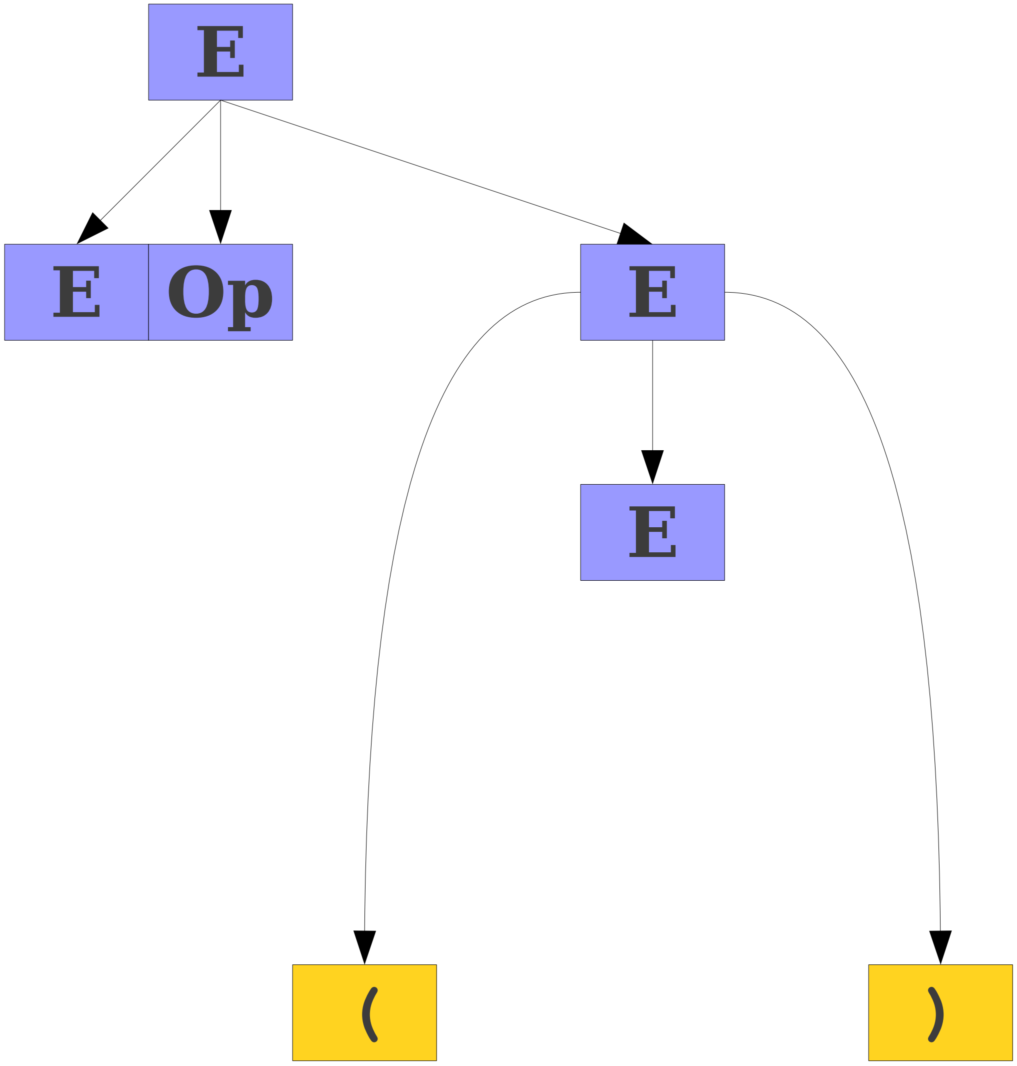
Parse Trees

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)



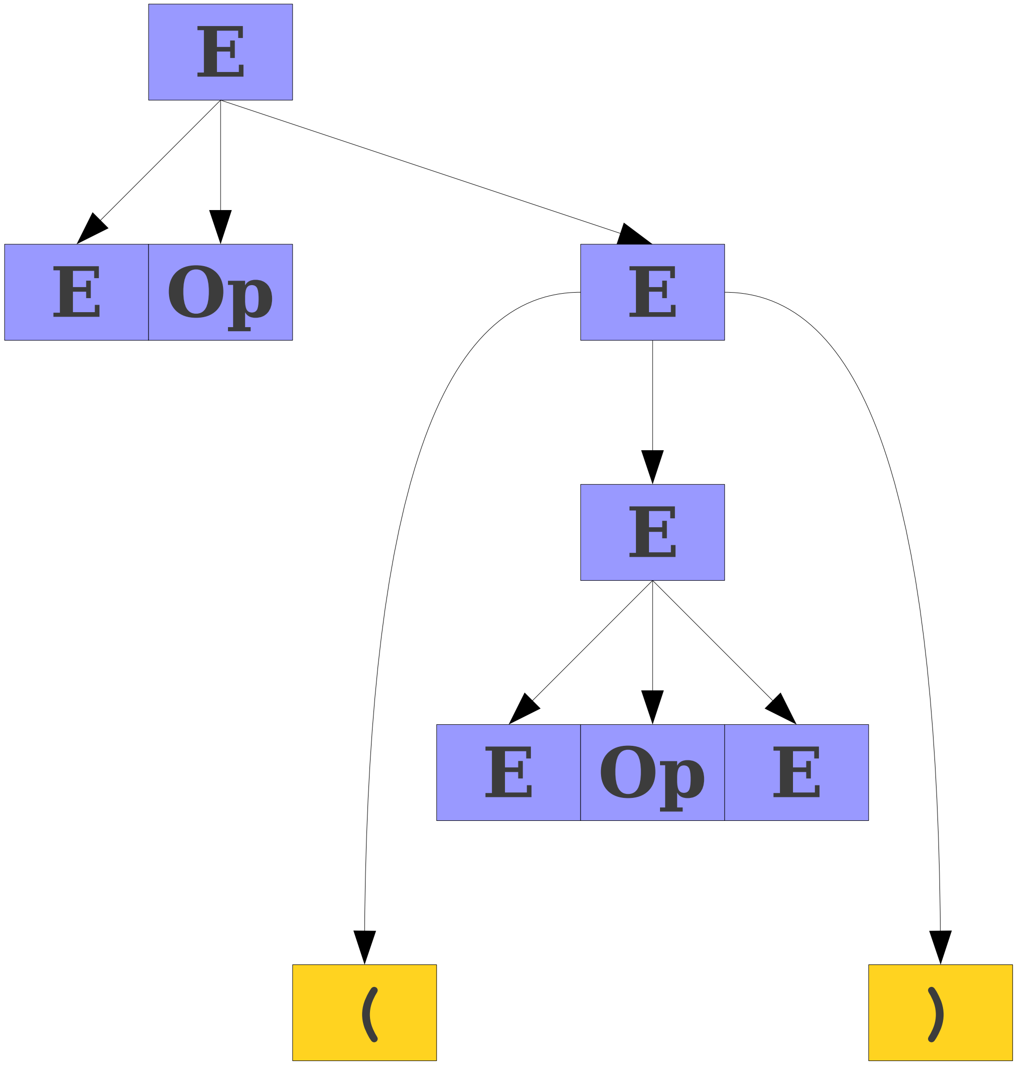
Parse Trees

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)



Parse Trees

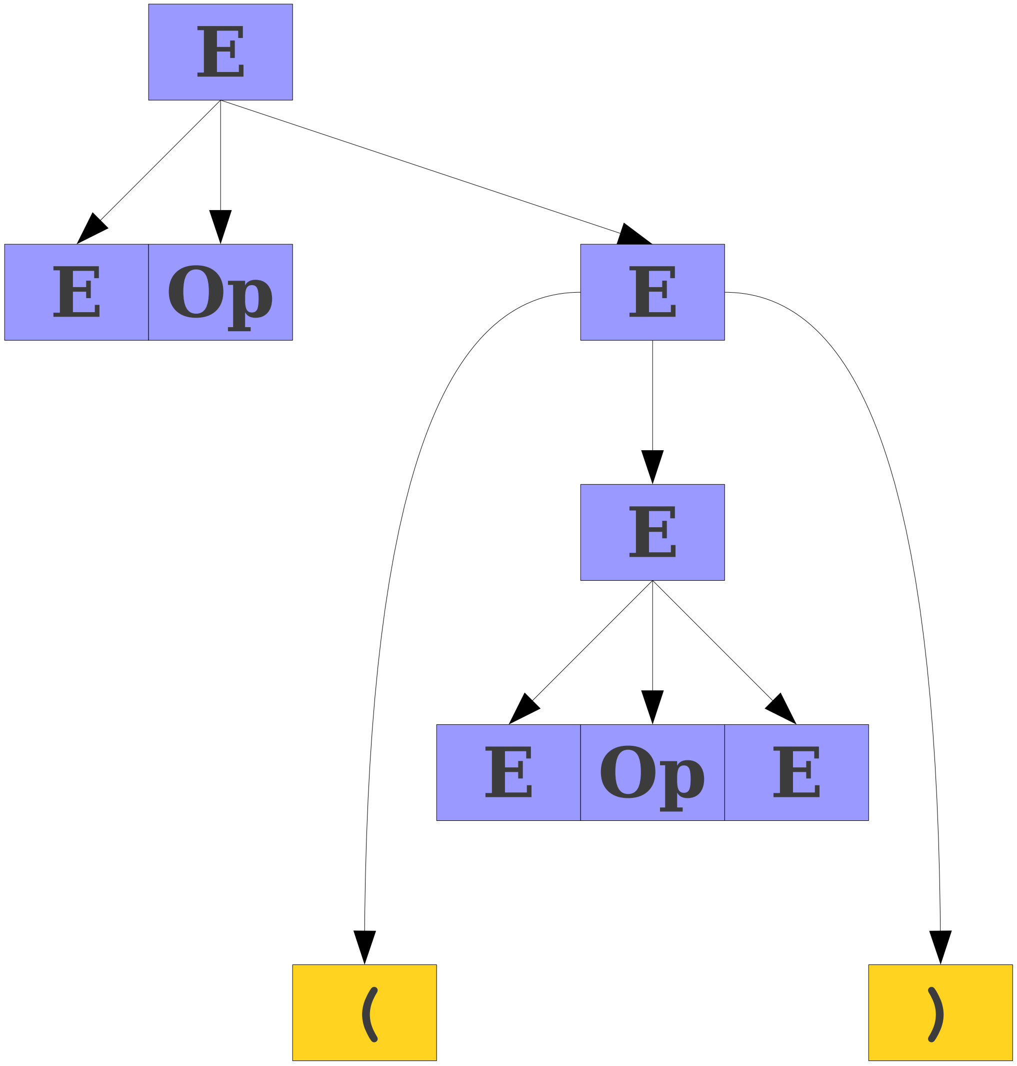
E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)



Parse Trees

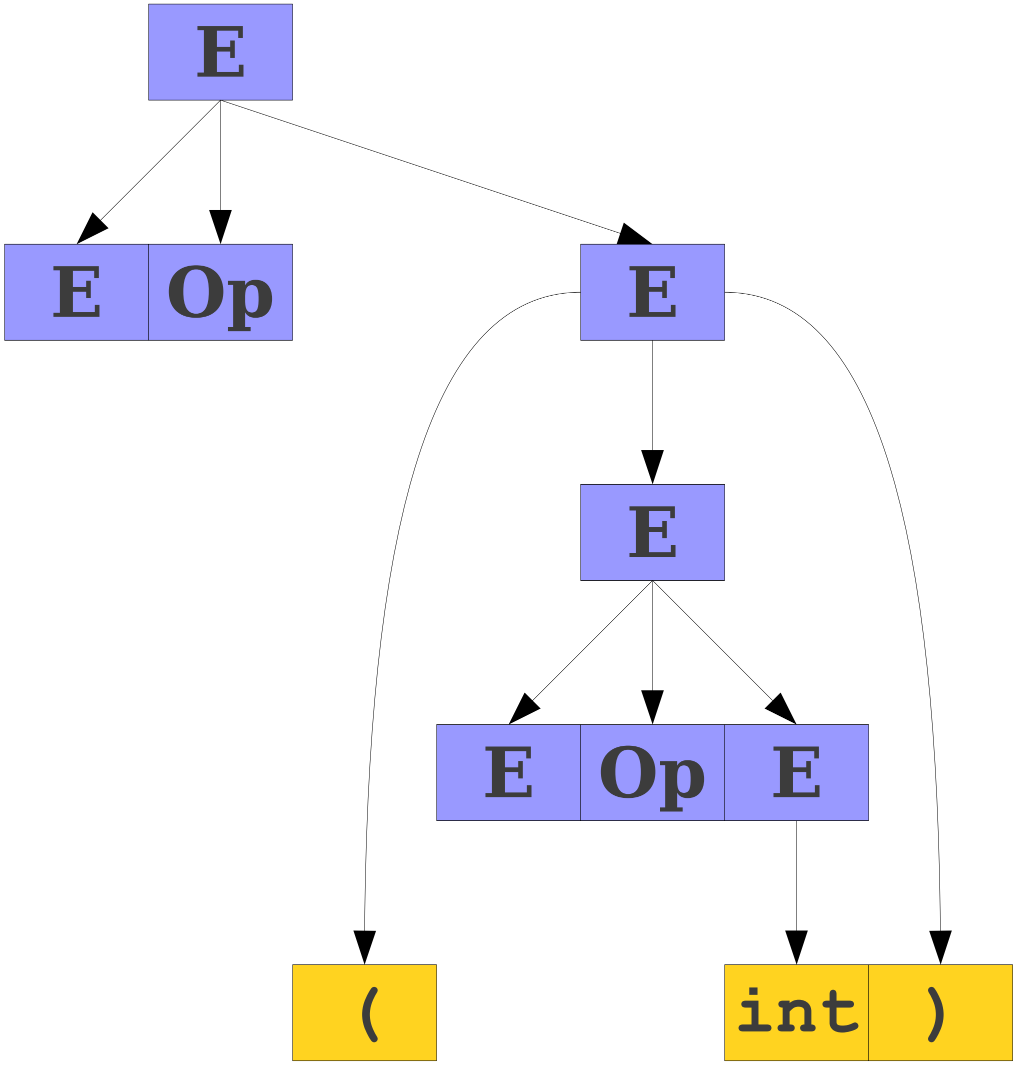
E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)



Parse Trees

E

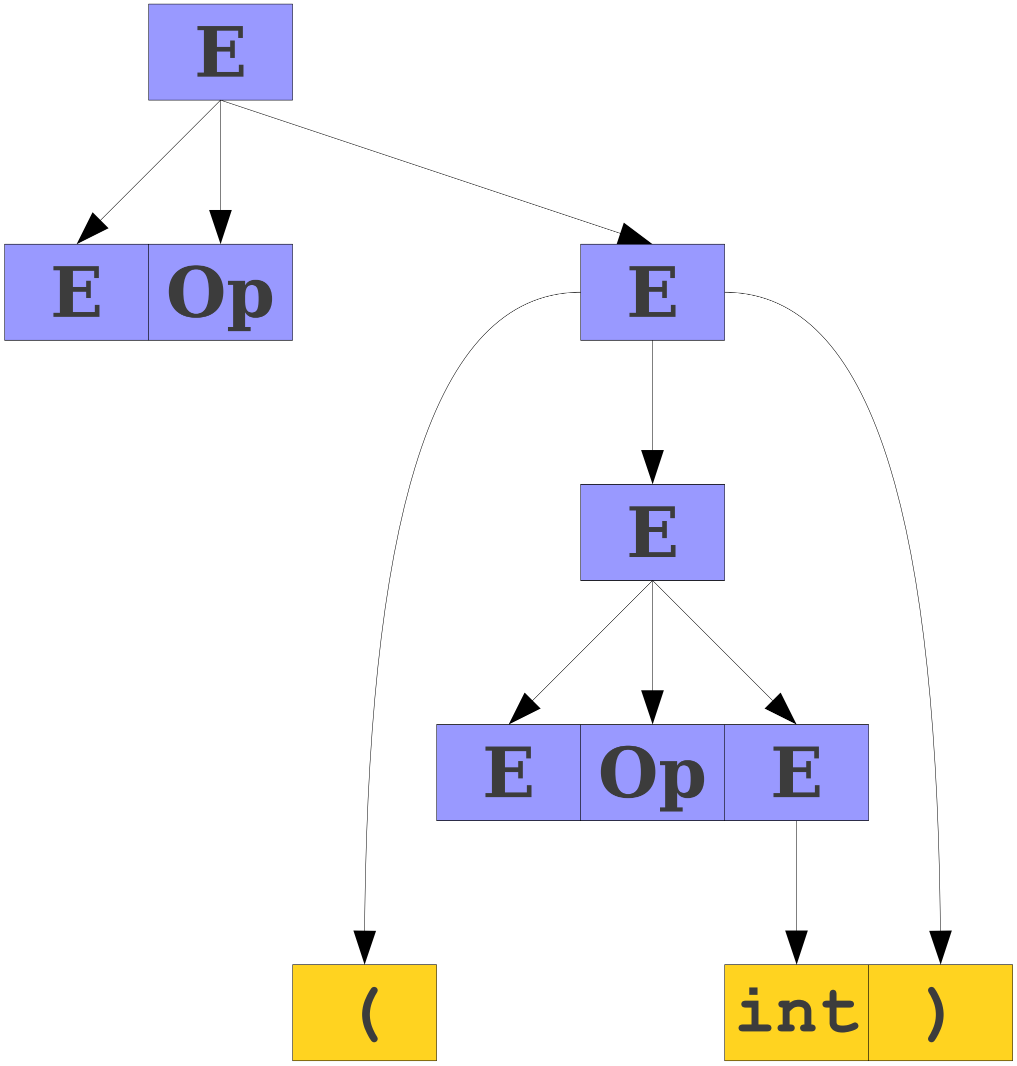
⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)



Parse Trees

E

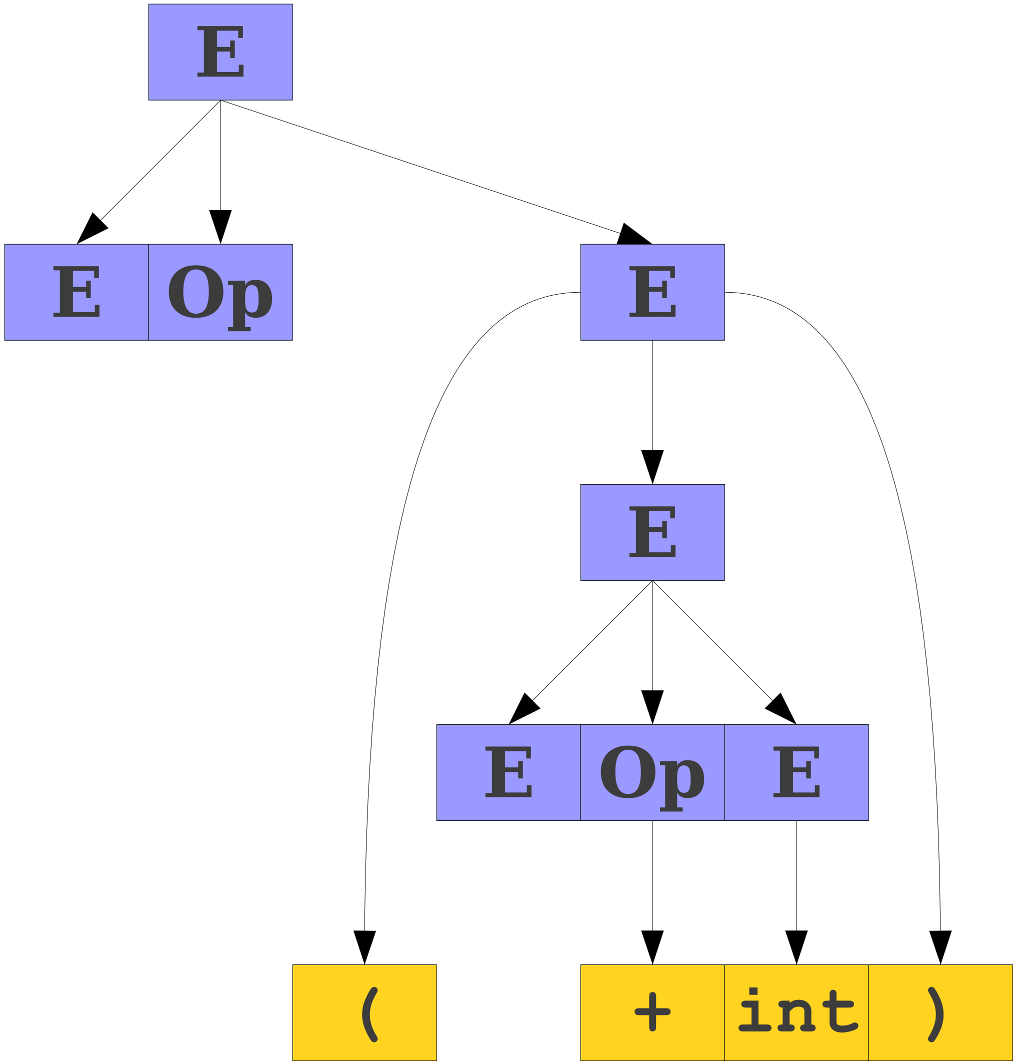
⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)



Parse Trees

E

⇒ **E Op E**

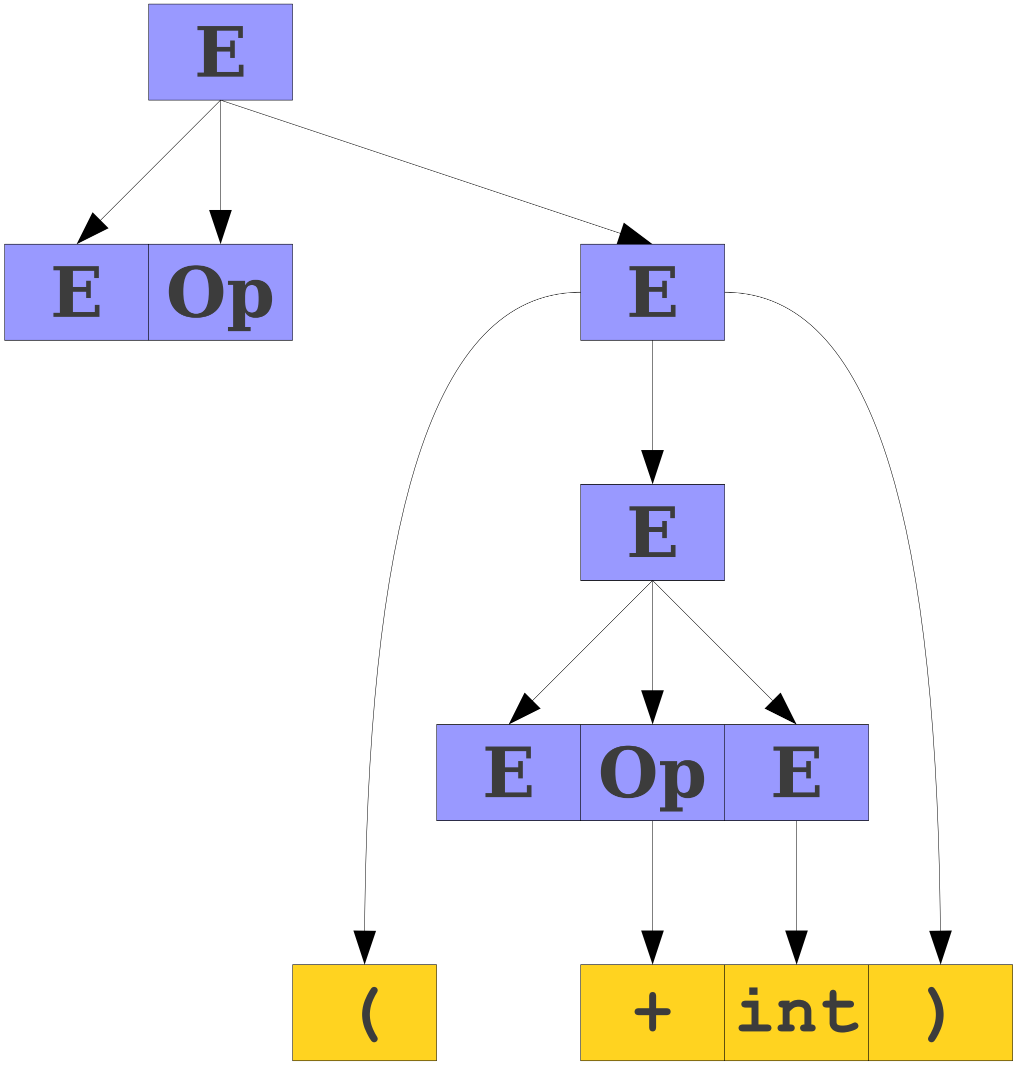
⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

⇒ **E Op** `(`**E Op** `int)`

⇒ **E Op** `(`**E** `+ int)`

⇒ **E Op** `(int + int)`



Parse Trees

E

⇒ E Op E

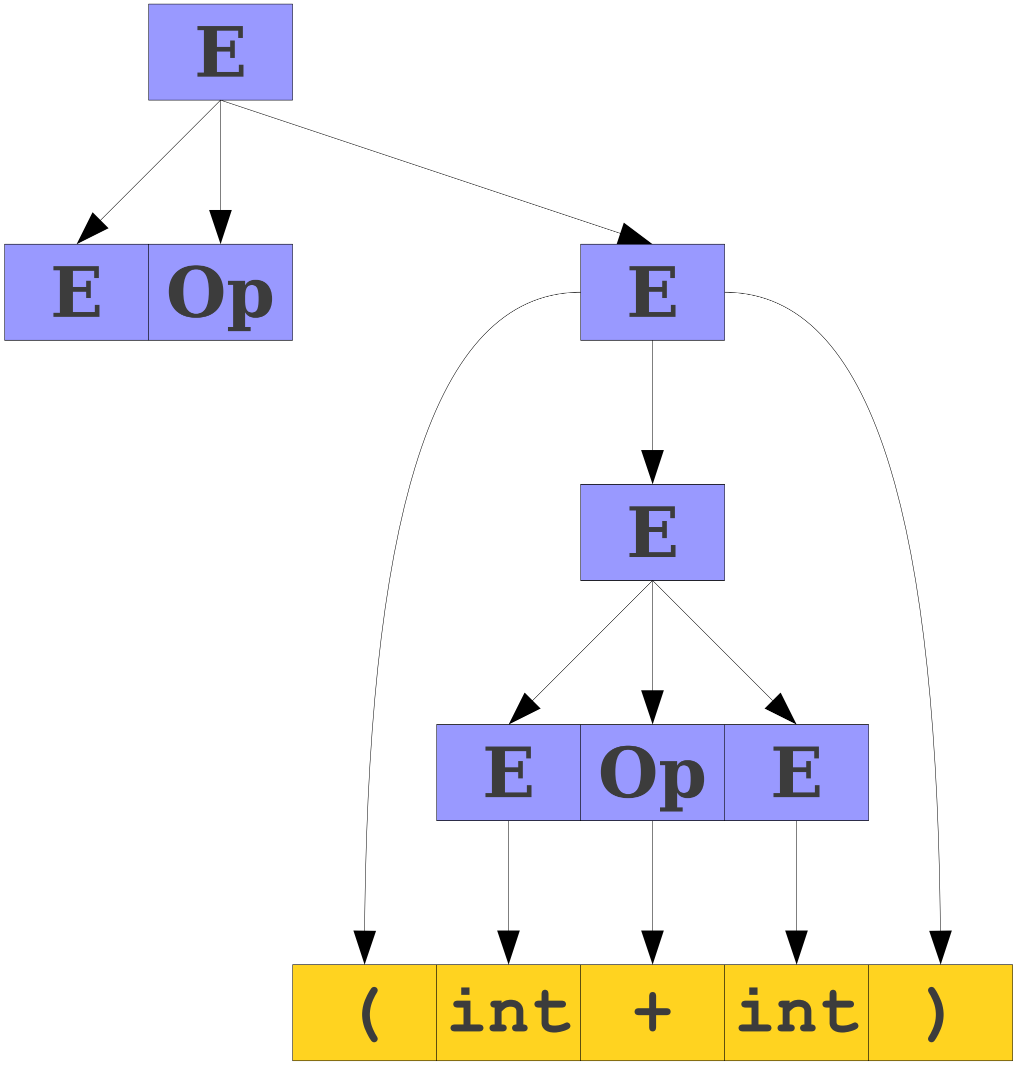
⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

⇒ E Op (int + int)



Parse Trees

E

⇒ E Op E

⇒ E Op (E)

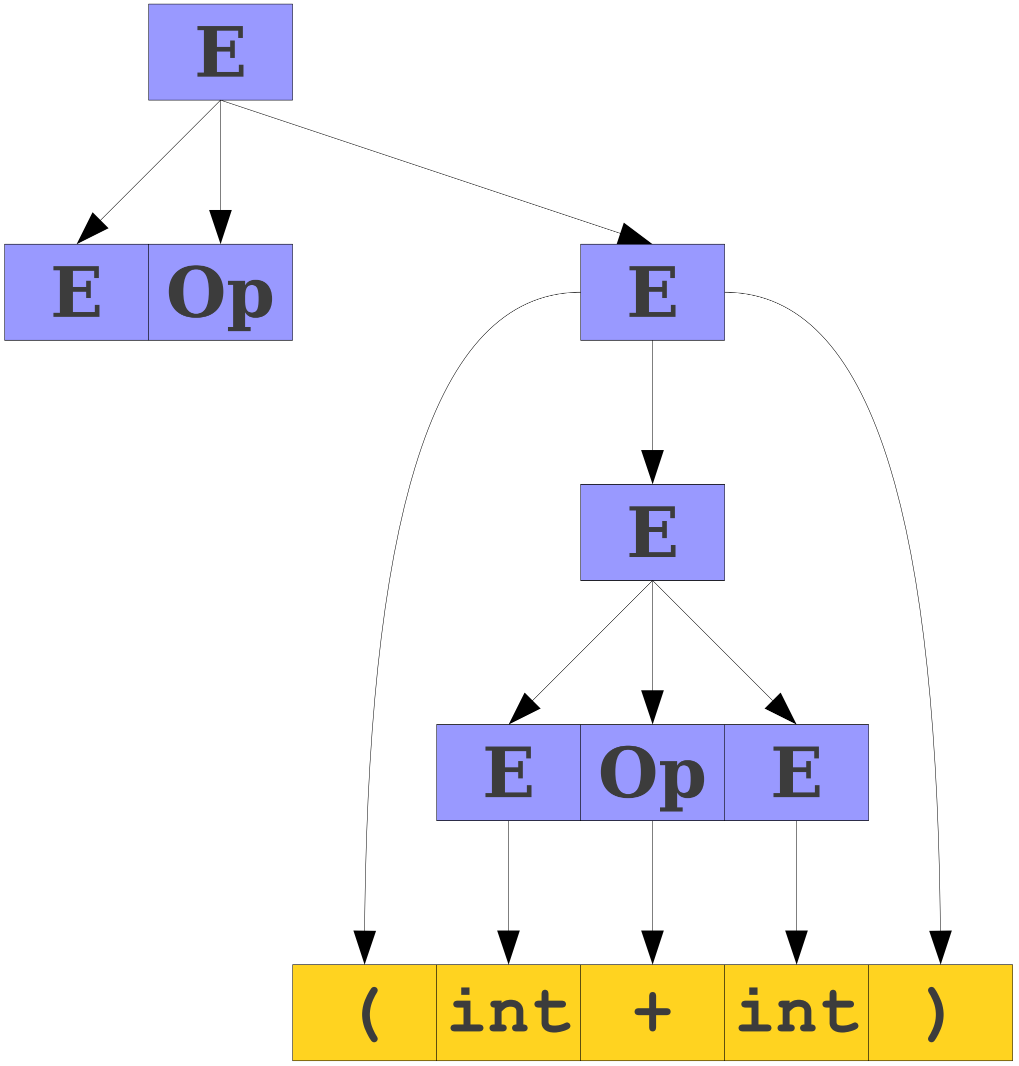
⇒ E Op (E Op E)

⇒ E Op (E Op int)

⇒ E Op (E + int)

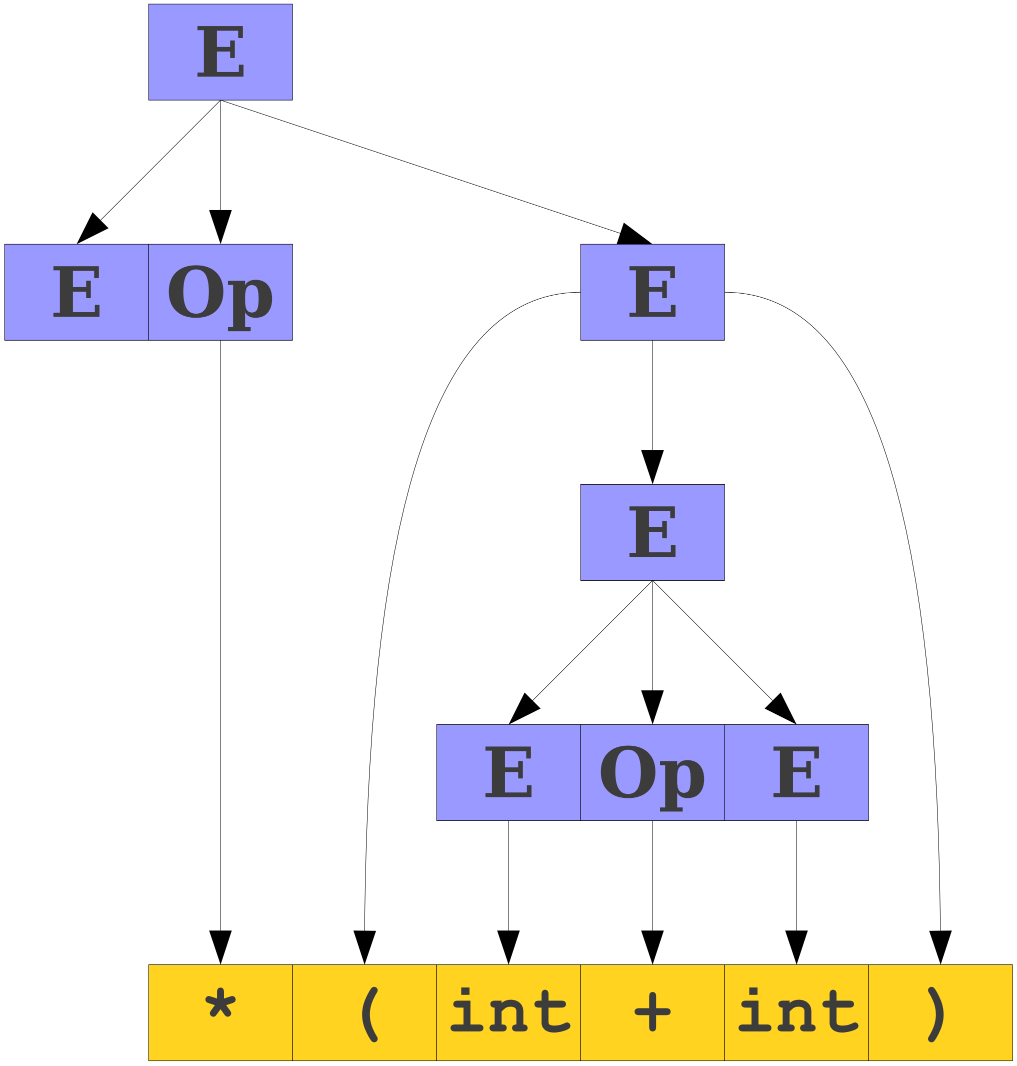
⇒ E Op (int + int)

⇒ 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)



Parse Trees

E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

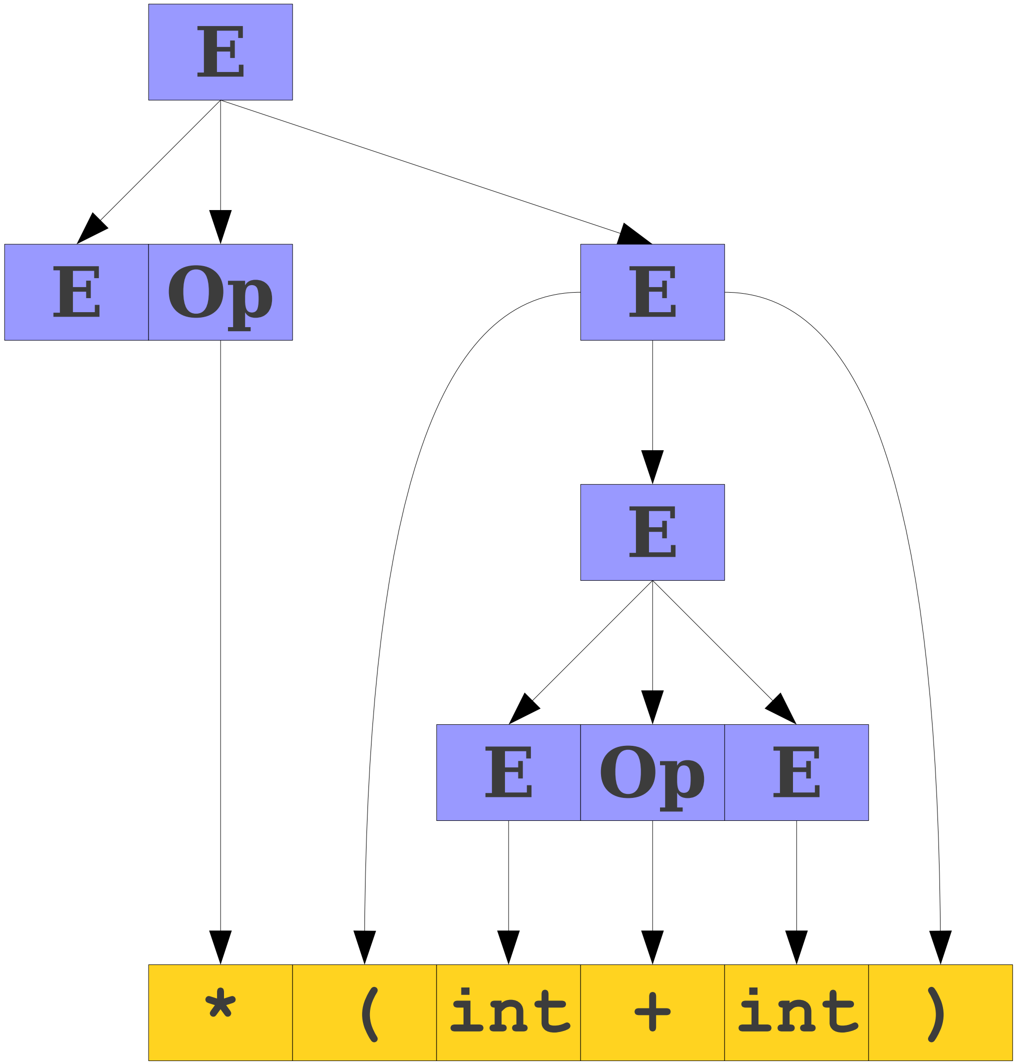
⇒ **E Op** `(`**E Op** `int)`

⇒ **E Op** `(`**E** `+ int)`

⇒ **E Op** `(int + int)`

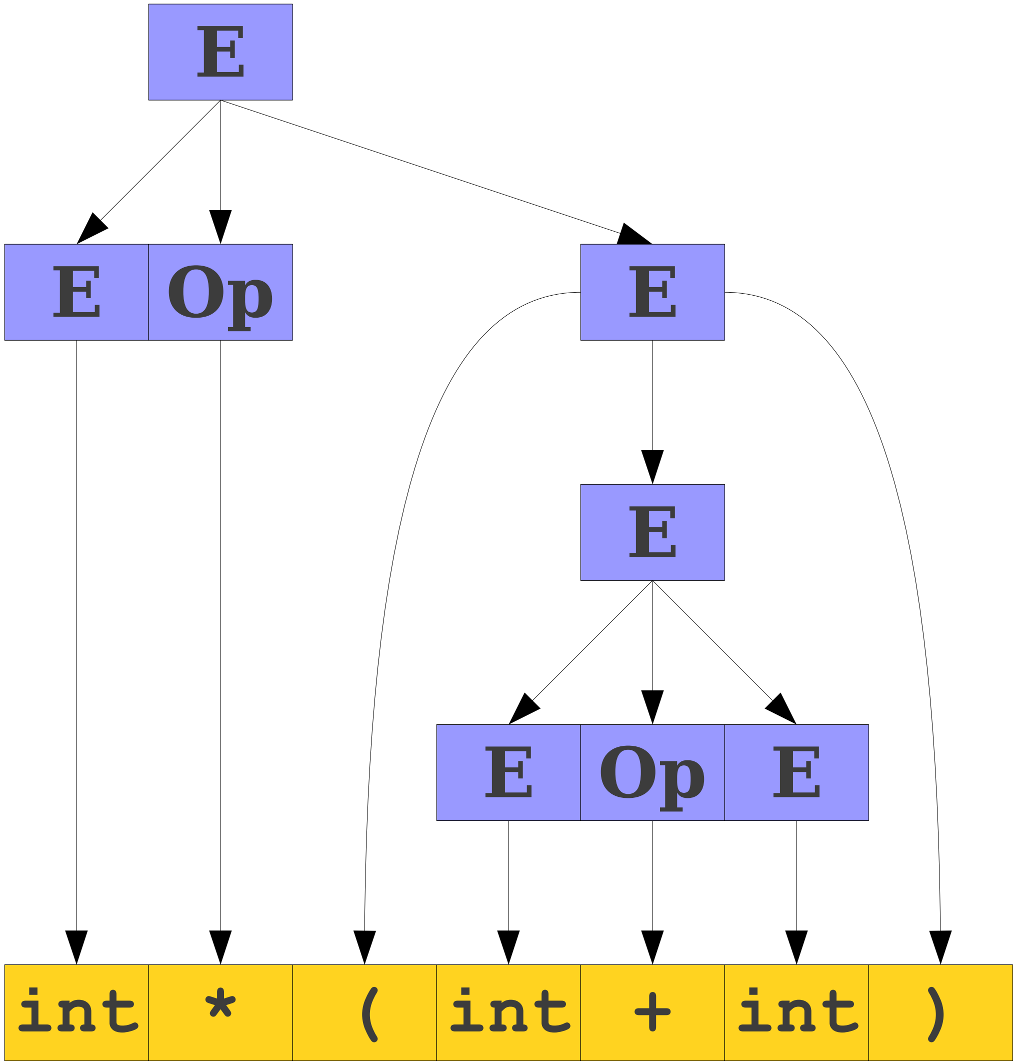
⇒ **E** `* (int + int)`

⇒ `int * (int + int)`

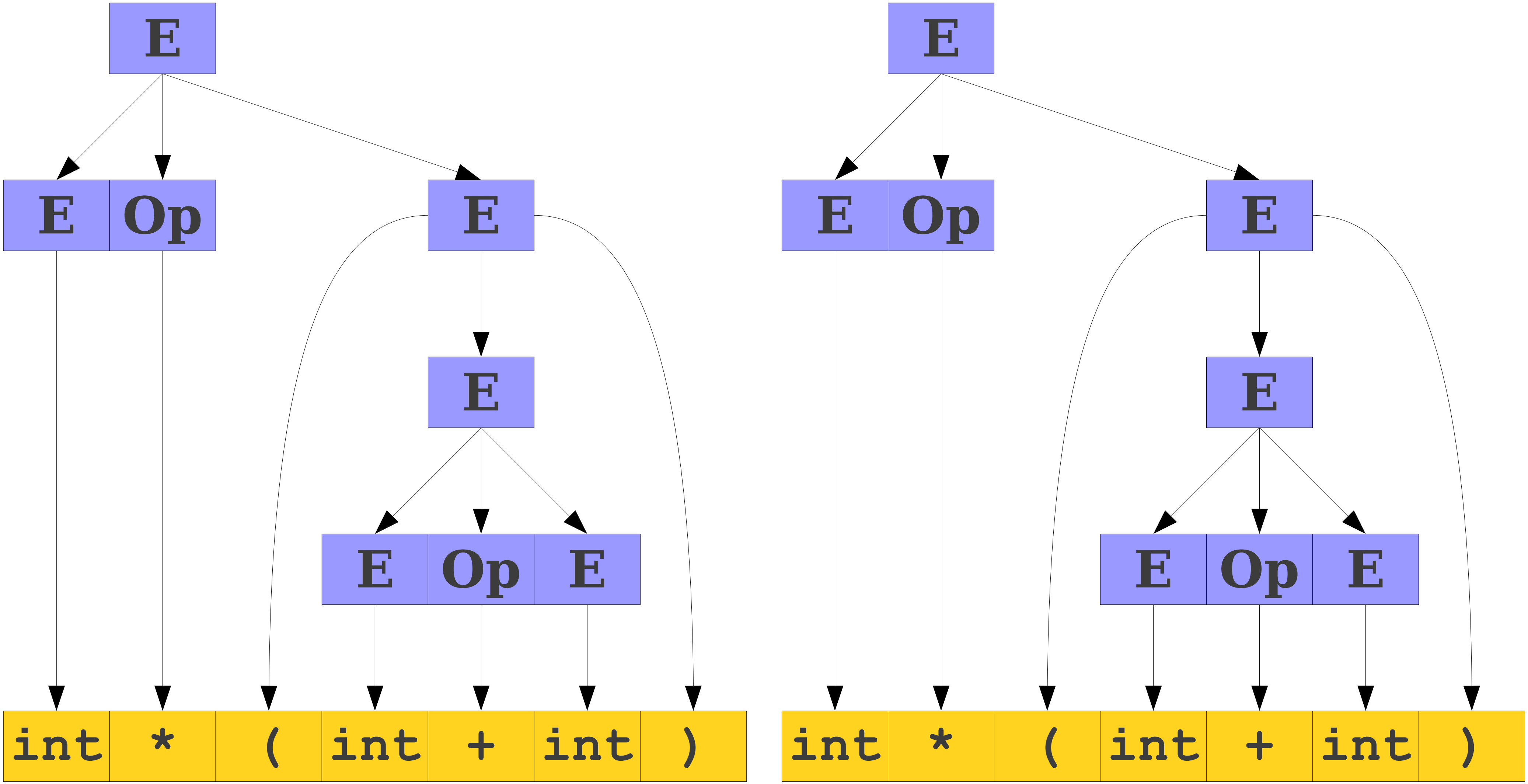


Parse Trees

E
⇒ 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



Parse Trees

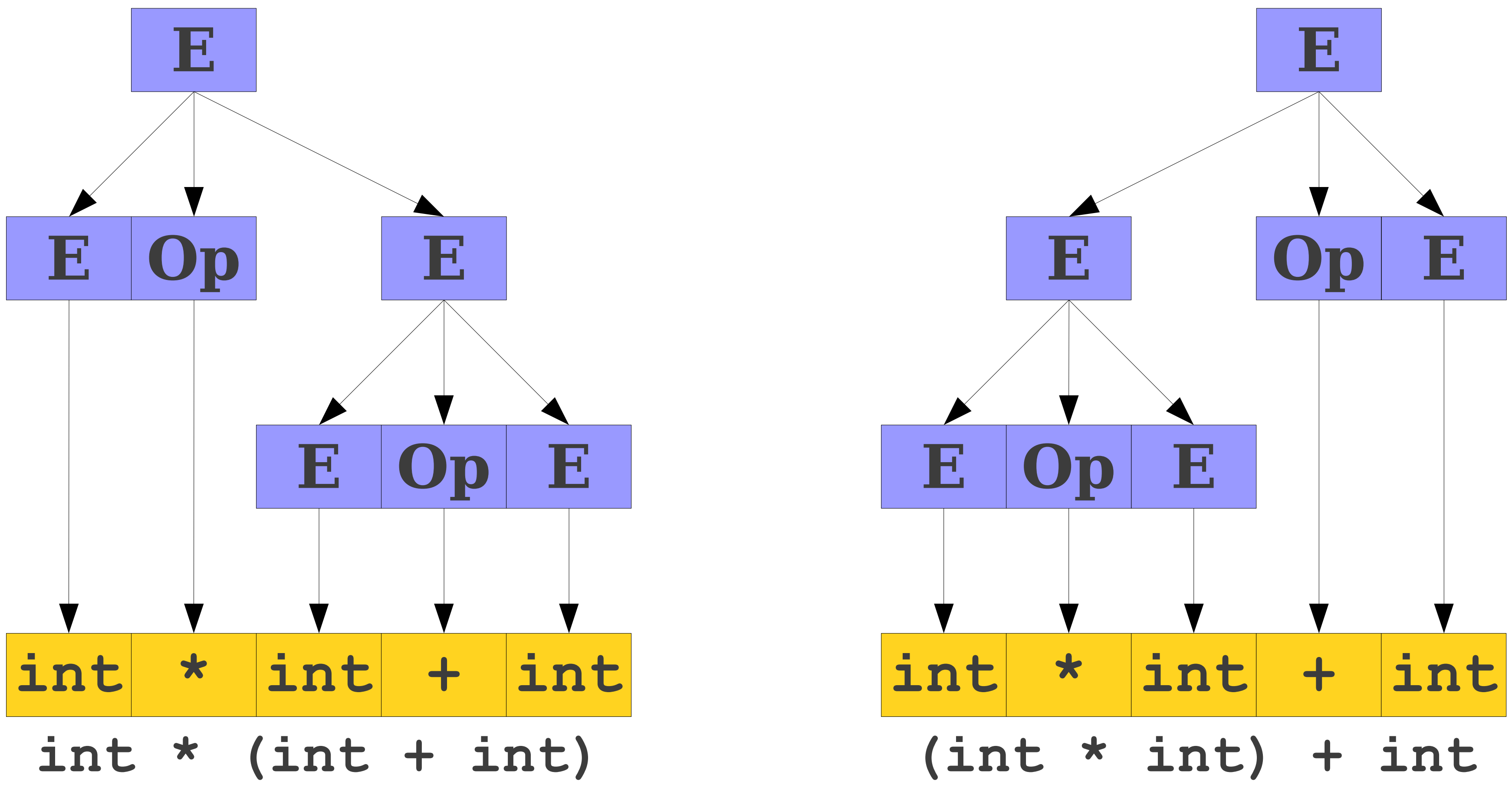
- A **parse tree** is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Inorder walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

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.

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?

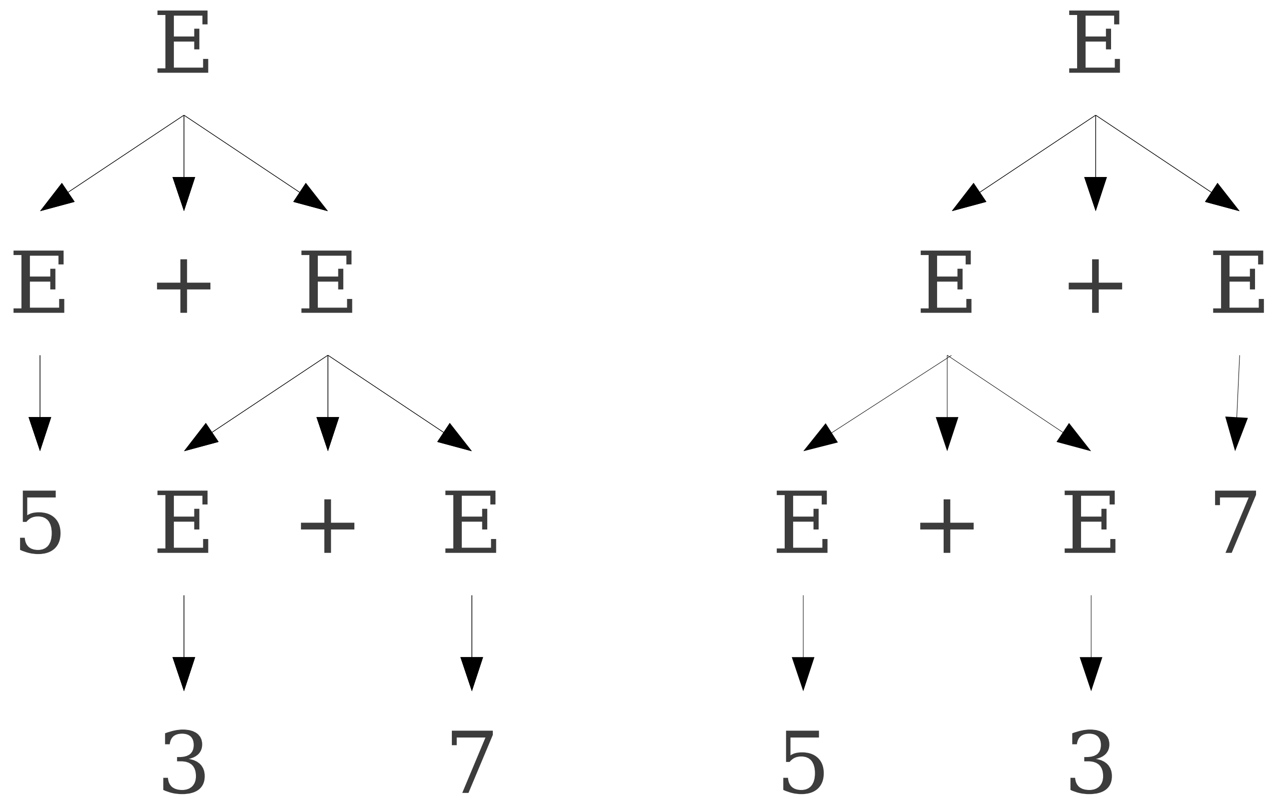
- Depends on **semantics**.

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

Is Ambiguity a Problem?

- Depends on **semantics**.

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



Is Ambiguity a Problem?

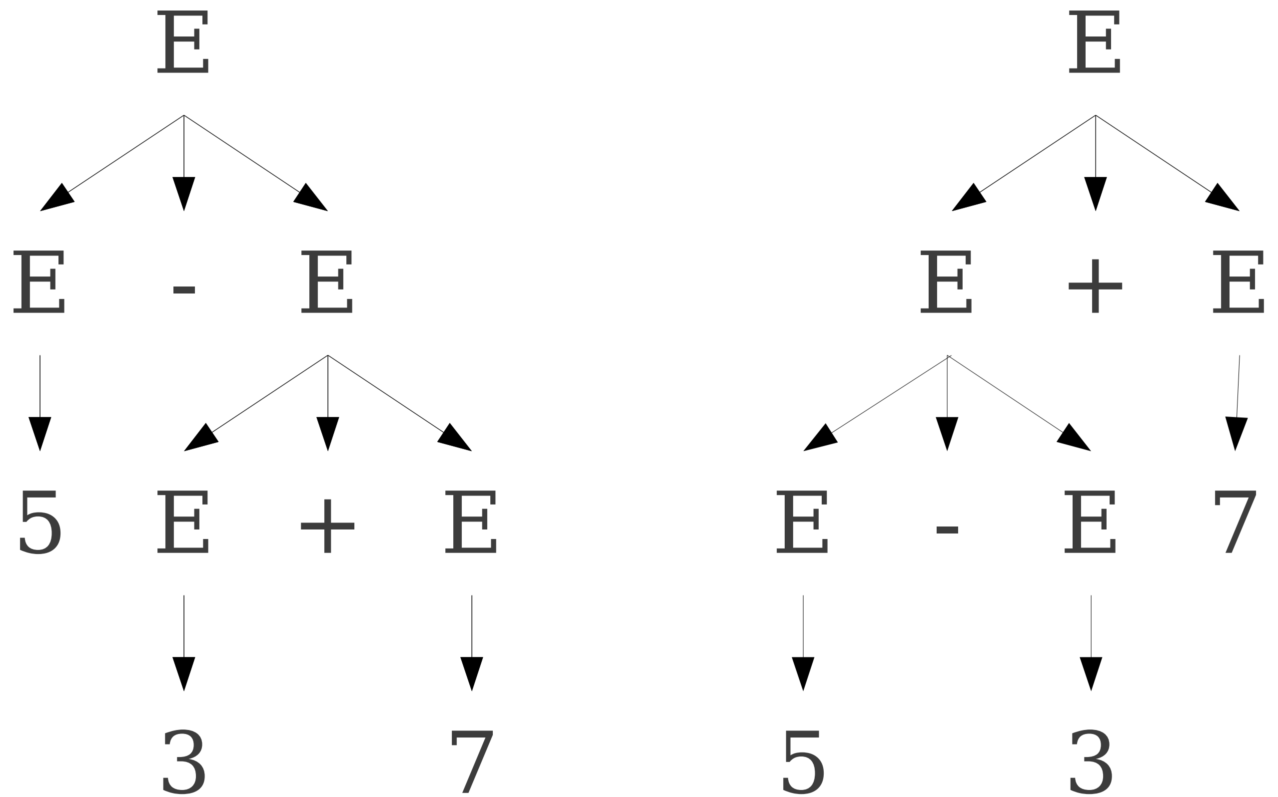
- Depends on **semantics**.

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

Is Ambiguity a Problem?

- Depends on **semantics**.

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



Resolving Ambiguity

- If a grammar can be made unambiguous at all, it is usually made unambiguous through **layering**.
- Have exactly one way to build each piece of the string.
- Have exactly one way of combining those pieces back together.

Example: Balanced Parentheses

- Consider the language of all strings of balanced parentheses.
- Examples:
  - ε
  - `()`
  - `(()())`
  - `((()))(())()`
- Here is one possible grammar for balanced parentheses:

$$\mathbf{P} \rightarrow \boldsymbol{\varepsilon} \mid \mathbf{PP} \mid \mathtt{(}\mathbf{P}\mathtt{)}$$

Balanced Parentheses

- Given the grammar **P** → **ε** | **PP** | **(P)**
- How might we generate the string `(()())`?

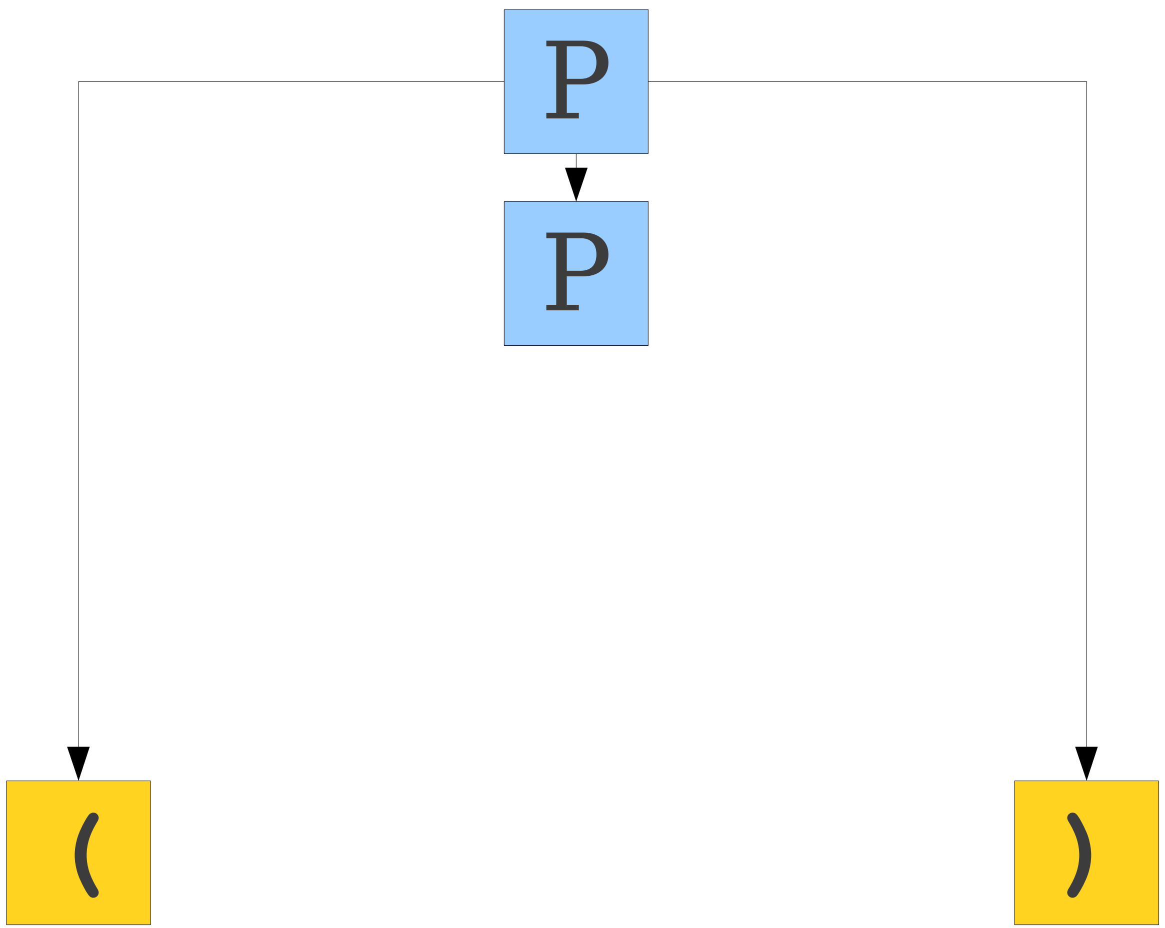
Balanced Parentheses

- Given the grammar **P** → **ε** | **PP** | **(P)**
- How might we generate the string `(()())`?

P

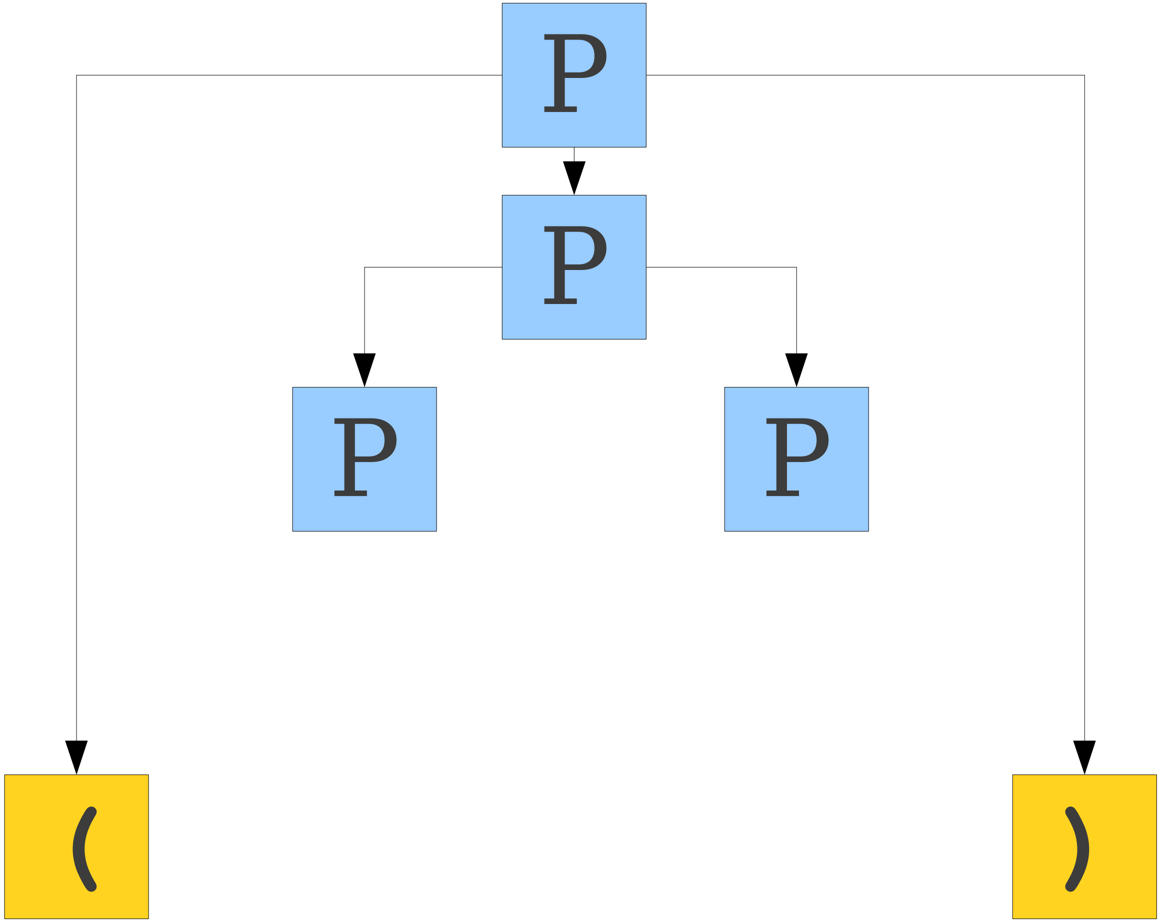
Balanced Parentheses

- Given the grammar **P** → **ε** | **PP** | **(P)**
- How might we generate the string (()())?



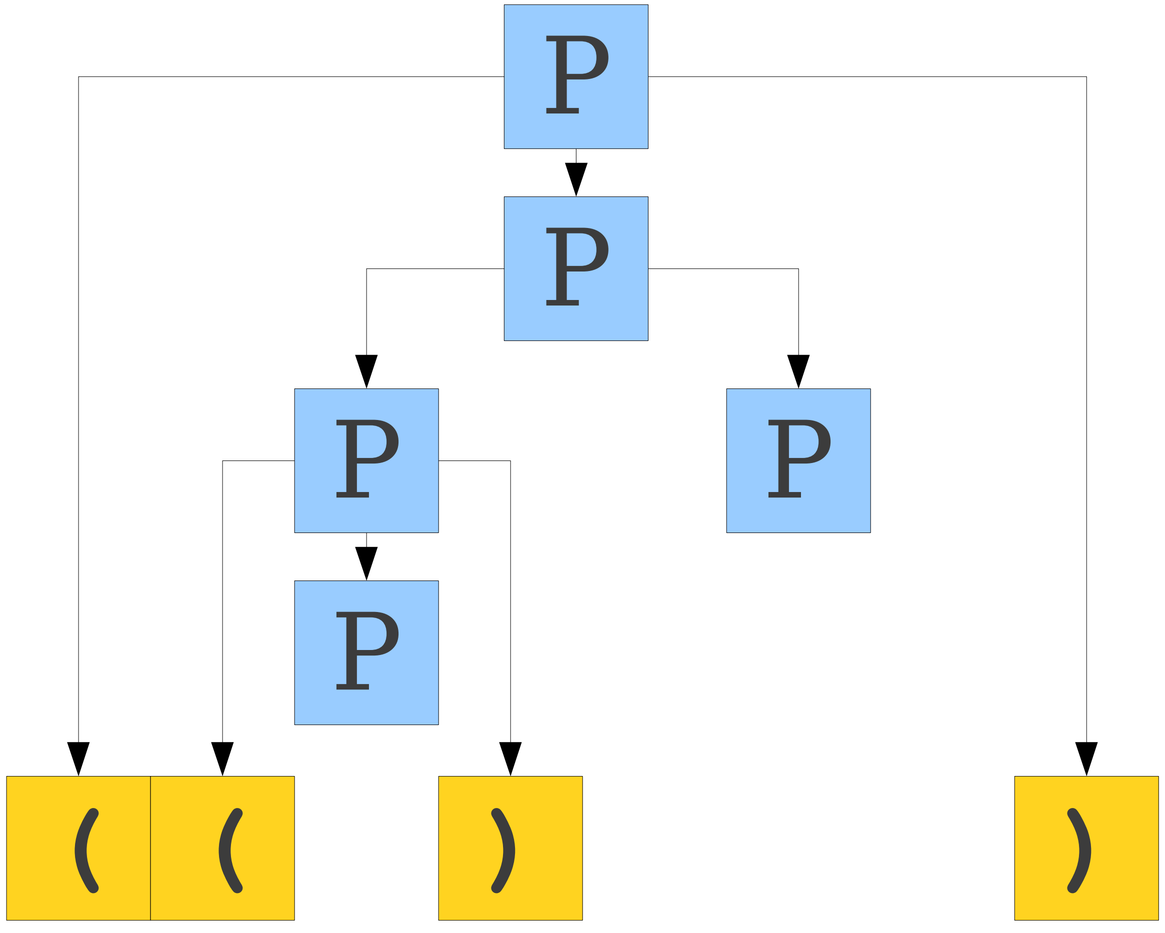
Balanced Parentheses

- Given the grammar **P** → **ε** | **PP** | **(P)**
- How might we generate the string `(()())`?



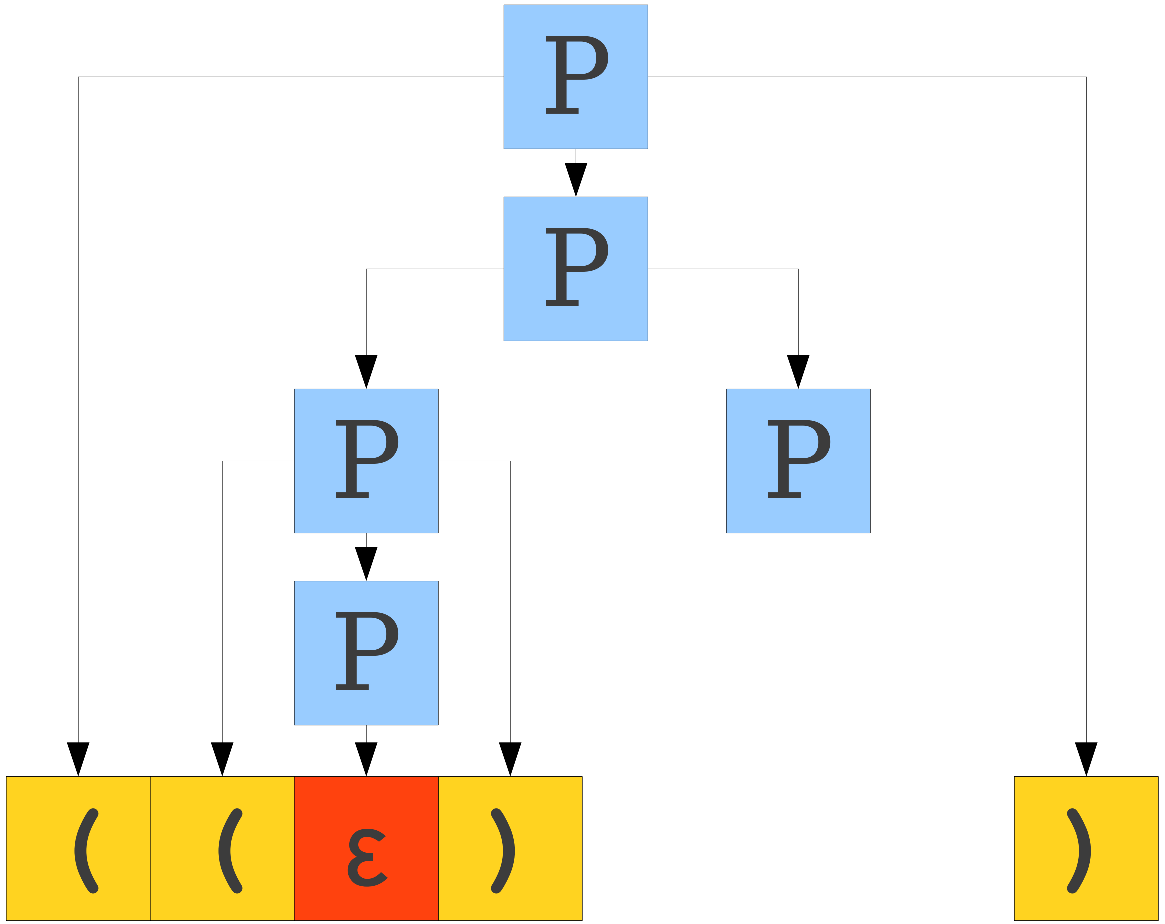
Balanced Parentheses

- Given the grammar **P** → **ε** | **PP** | **(P)**
- How might we generate the string `(()())`?



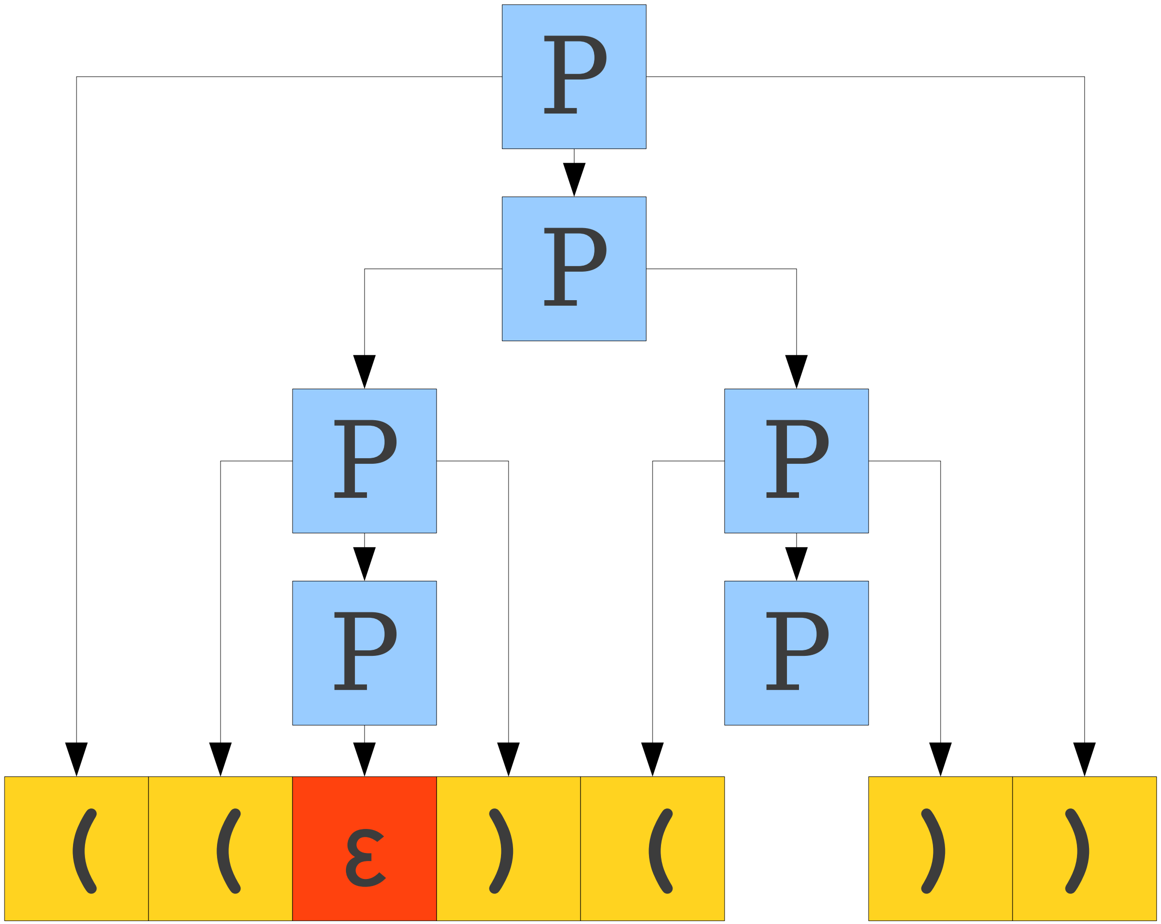
Balanced Parentheses

- Given the grammar **P** → **ε** | **PP** | **(P)**
- How might we generate the string `(()())`?



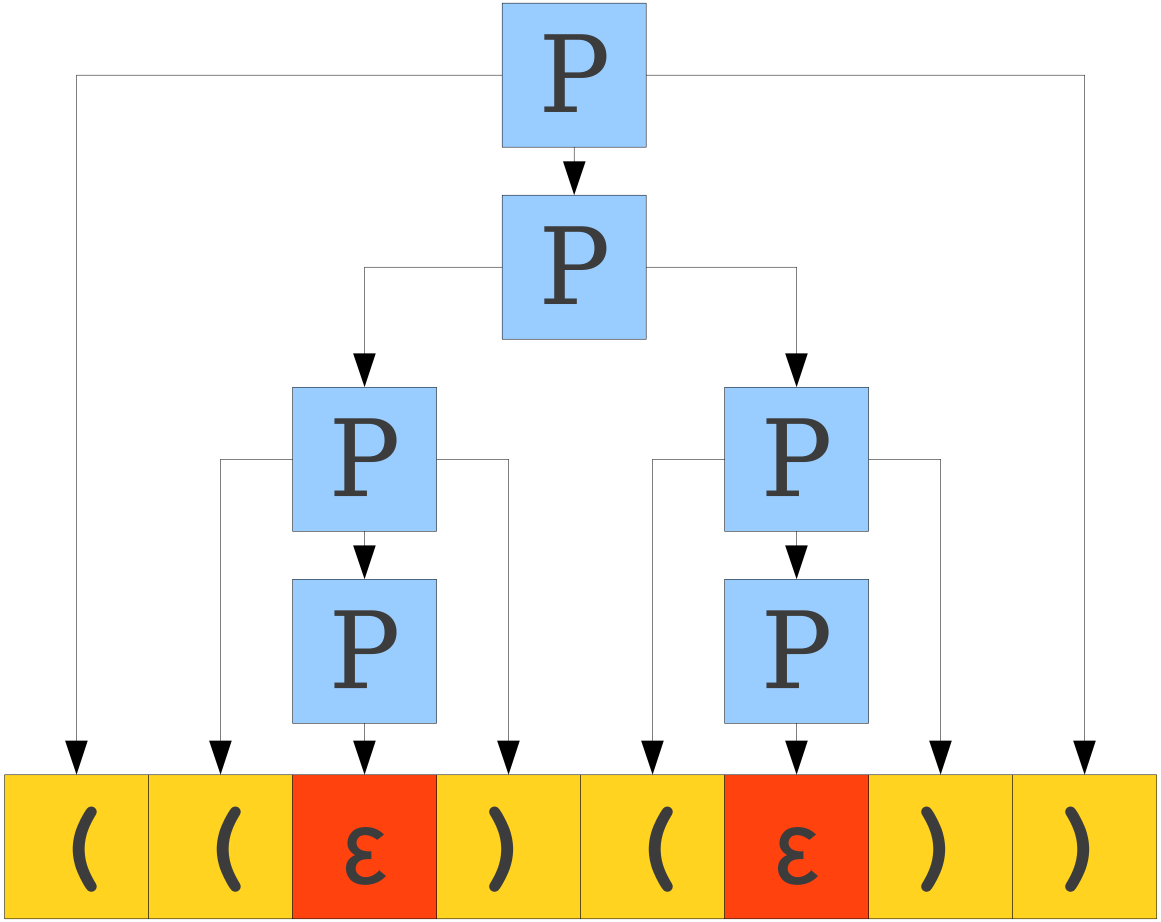
Balanced Parentheses

- Given the grammar **P** → **ε** | **PP** | **(P)**
- How might we generate the string (()())?

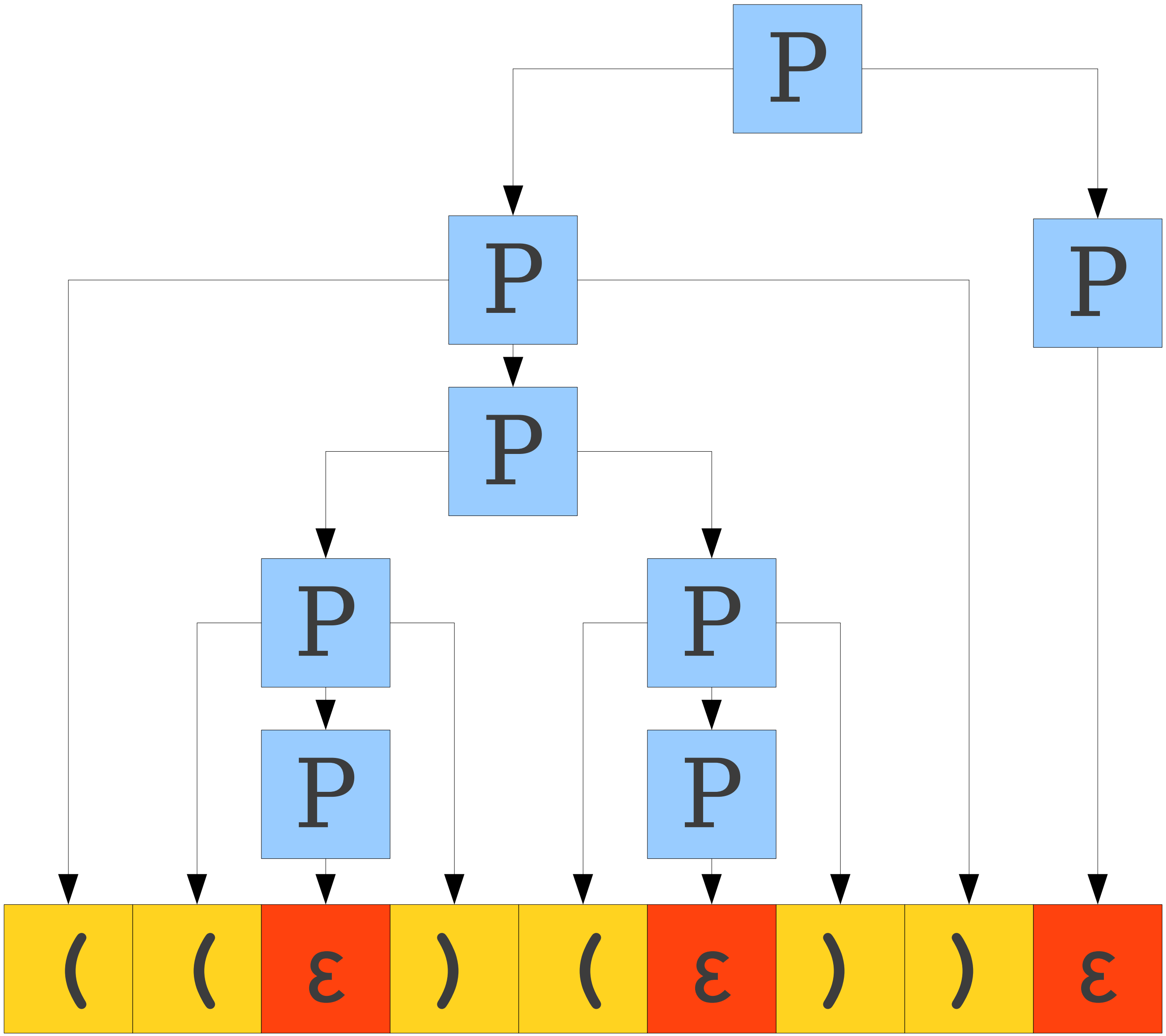


Balanced Parentheses

- Given the grammar **P** → **ε** | **PP** | **(P)**
- How might we generate the string `(()())`?

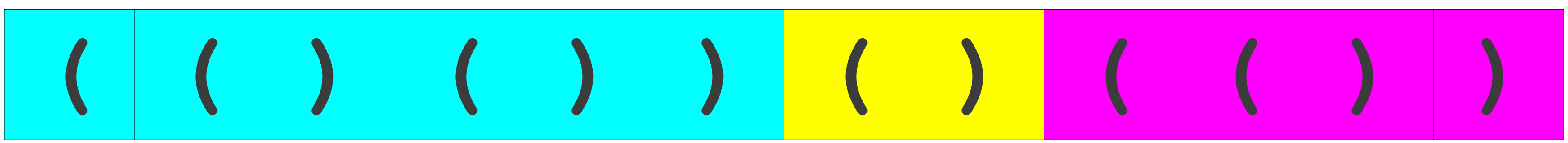


Balanced Parentheses



How to resolve this ambiguity?

(	(	)	(	)	)	(	)	(	(	)	)



Rethinking Parentheses

- A string of balanced parentheses is a sequence of strings that are themselves balanced parentheses.
- To avoid ambiguity, we can build the string in two steps:
  - Decide how many different substrings we will glue together.
  - Build each substring independently.

Let's ask the Internet for help!

Um... what?

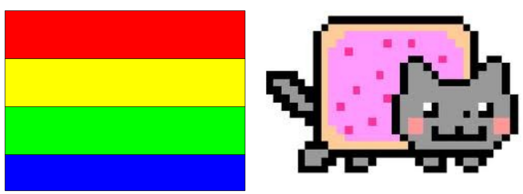
- The way the nyancat flies across the sky is similar to how we can build up strings of balanced parentheses.

Um... what?

- The way the nyancat flies across the sky is similar to how we can build up strings of balanced parentheses.

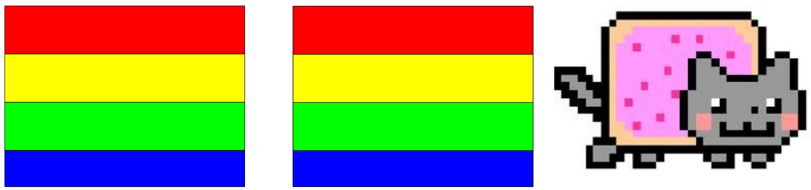
Um... what?

- The way the nyancat flies across the sky is similar to how we can build up strings of balanced parentheses.



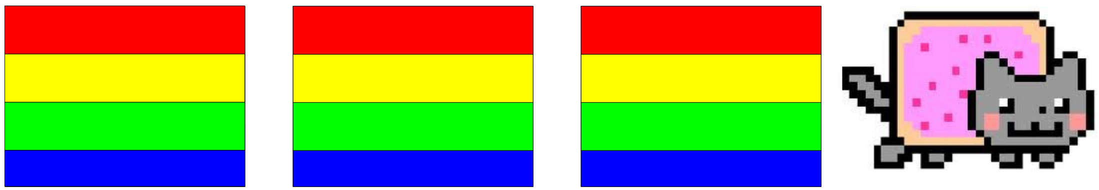
Um... what?

- The way the nyancat flies across the sky is similar to how we can build up strings of balanced parentheses.



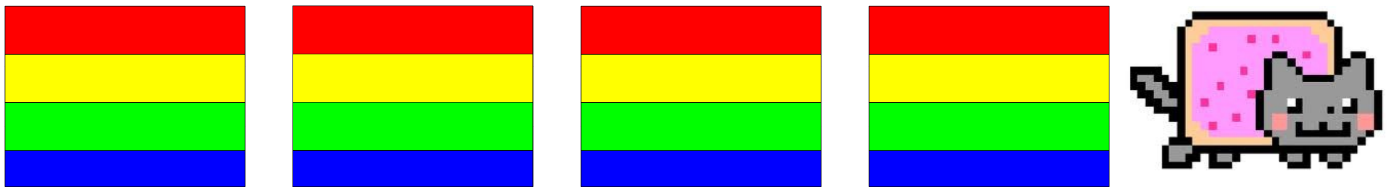
Um... what?

- The way the nyancat flies across the sky is similar to how we can build up strings of balanced parentheses.



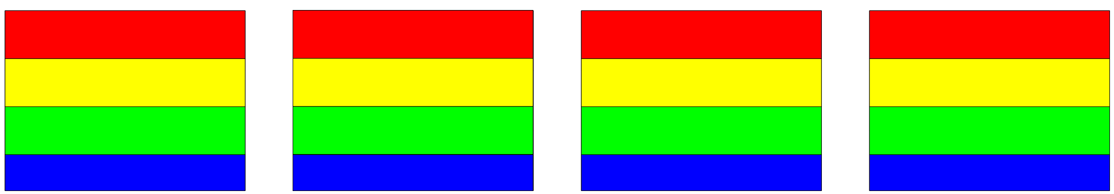
Um... what?

- The way the nyancat flies across the sky is similar to how we can build up strings of balanced parentheses.



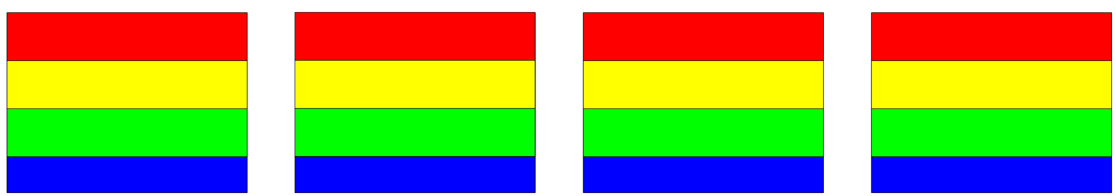
Um… what?

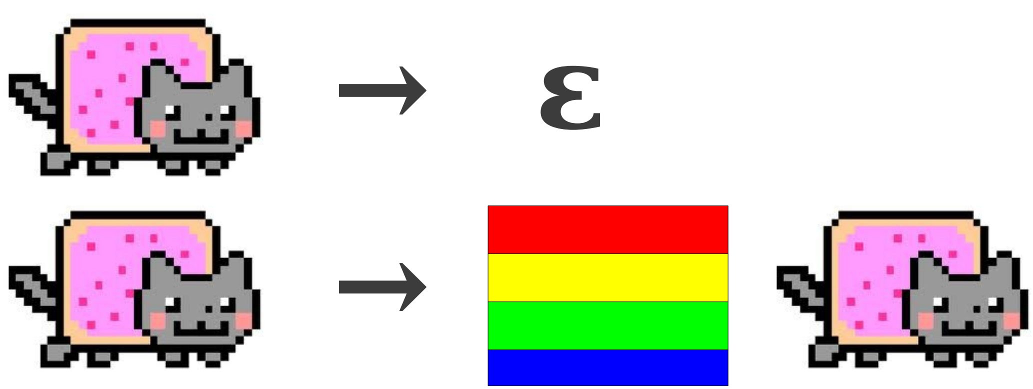
- The way the nyancat flies across the sky is similar to how we can build up strings of balanced parentheses.



Um… what?

- The way the nyancat flies across the sky is similar to how we can build up strings of balanced parentheses.





Building Parentheses

- Spread a string of parentheses across the string. There is exactly one way to do this for any number of parentheses.
- Expand out each substring by adding in parentheses and repeating.

$$S \rightarrow P\ S \mid \varepsilon$$

$$P \rightarrow (\ S\ )$$

Building Parentheses

$$S \rightarrow P\,S \mid \varepsilon$$

$$P \rightarrow (\,S\,)$$

$$
\begin{aligned}
& S \\
\Rightarrow\; & PS \\
\Rightarrow\; & PPS \\
\Rightarrow\; & PP \\
\Rightarrow\; & (S)P \\
\Rightarrow\; & (S)(S) \\
\Rightarrow\; & (PS)(S) \\
\Rightarrow\; & (P)(S) \\
\Rightarrow\; & ((S))(S) \\
\Rightarrow\; & (())(S) \\
\Rightarrow\; & (())()
\end{aligned}
$$

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**

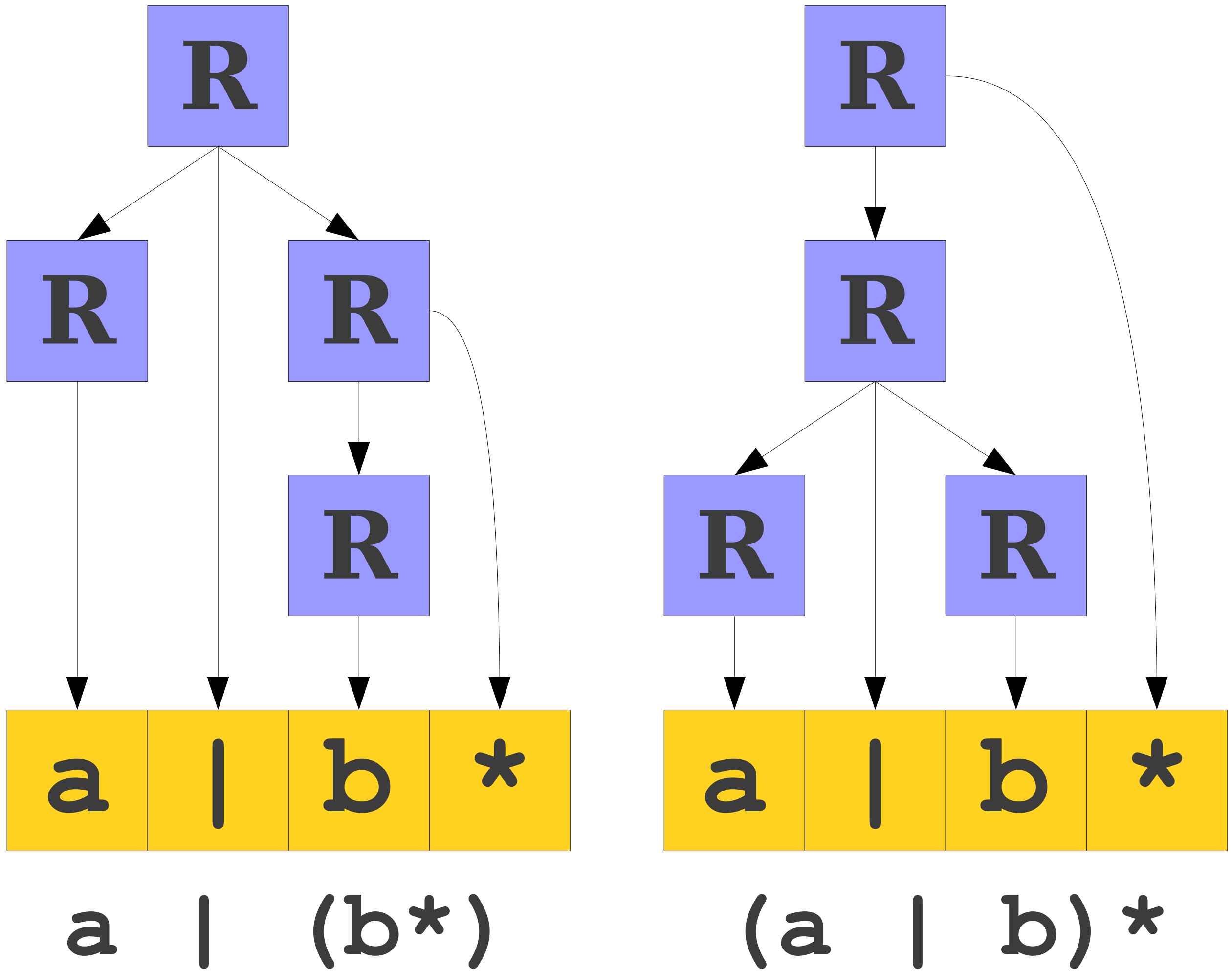
  R → **R** "`|`" **R**

  R → **R**`*`

  R → `(`**R**`)`

An Ambiguous Grammar

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



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)**

a 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)**

a	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)**

a	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**`)`

a	a	|	b	*

Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

R → `a` | `b` | `c` | …

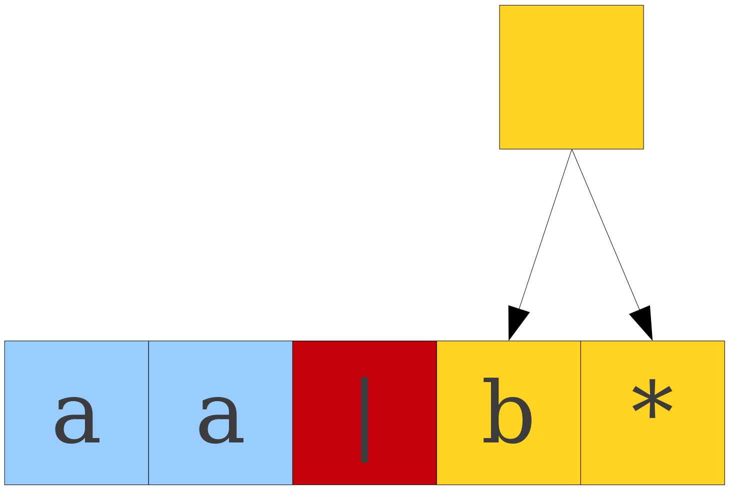
R → "`ε`"

R → **RR**

R → **R** "`|`" **R**

R → **R**`*`

R → `(`**R**`)`



Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

  R → `a` | `b` | `c` | …

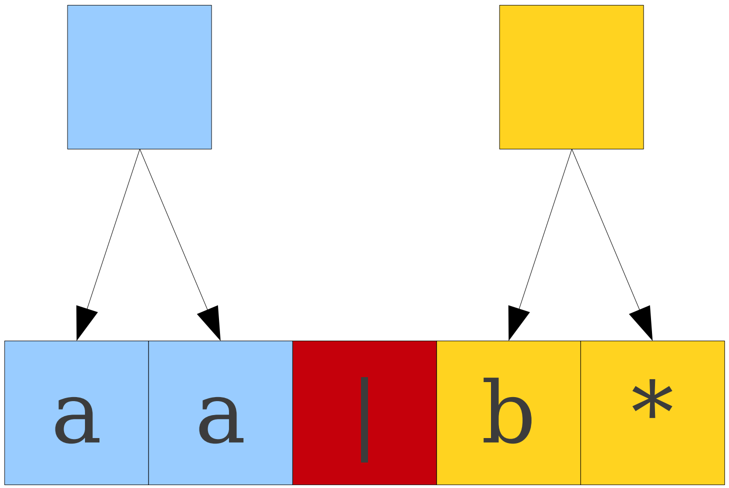
  R → "`ε`"

  R → **RR**

  R → **R** "`|`" **R**

  R → **R**`*`

  R → `(`**R**`)`



Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

R → `a` | `b` | `c` | …

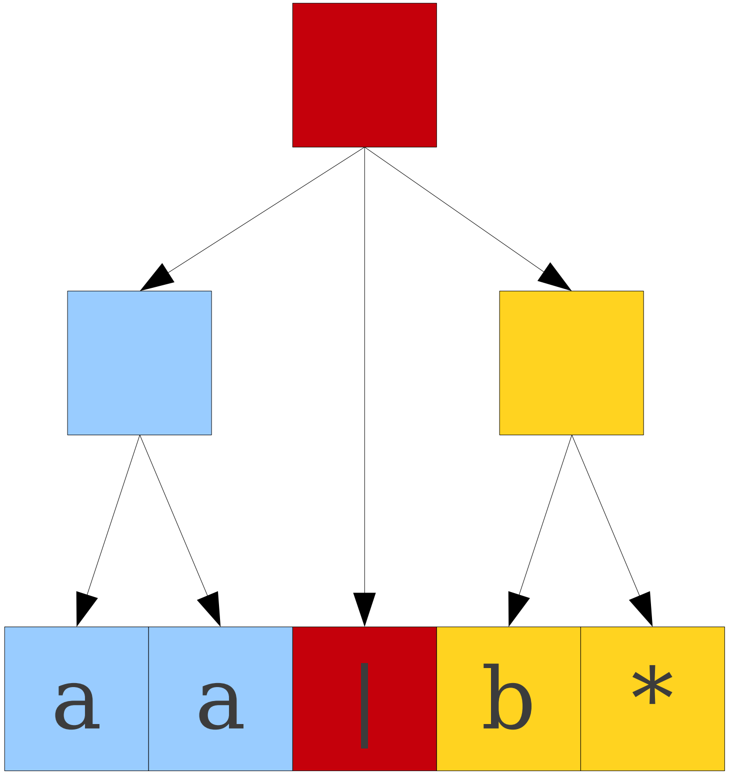
R → "`ε`"

R → **RR**

R → **R** "`|`" **R**

R → **R**`*`

R → `(`**R**`)`



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)**

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"

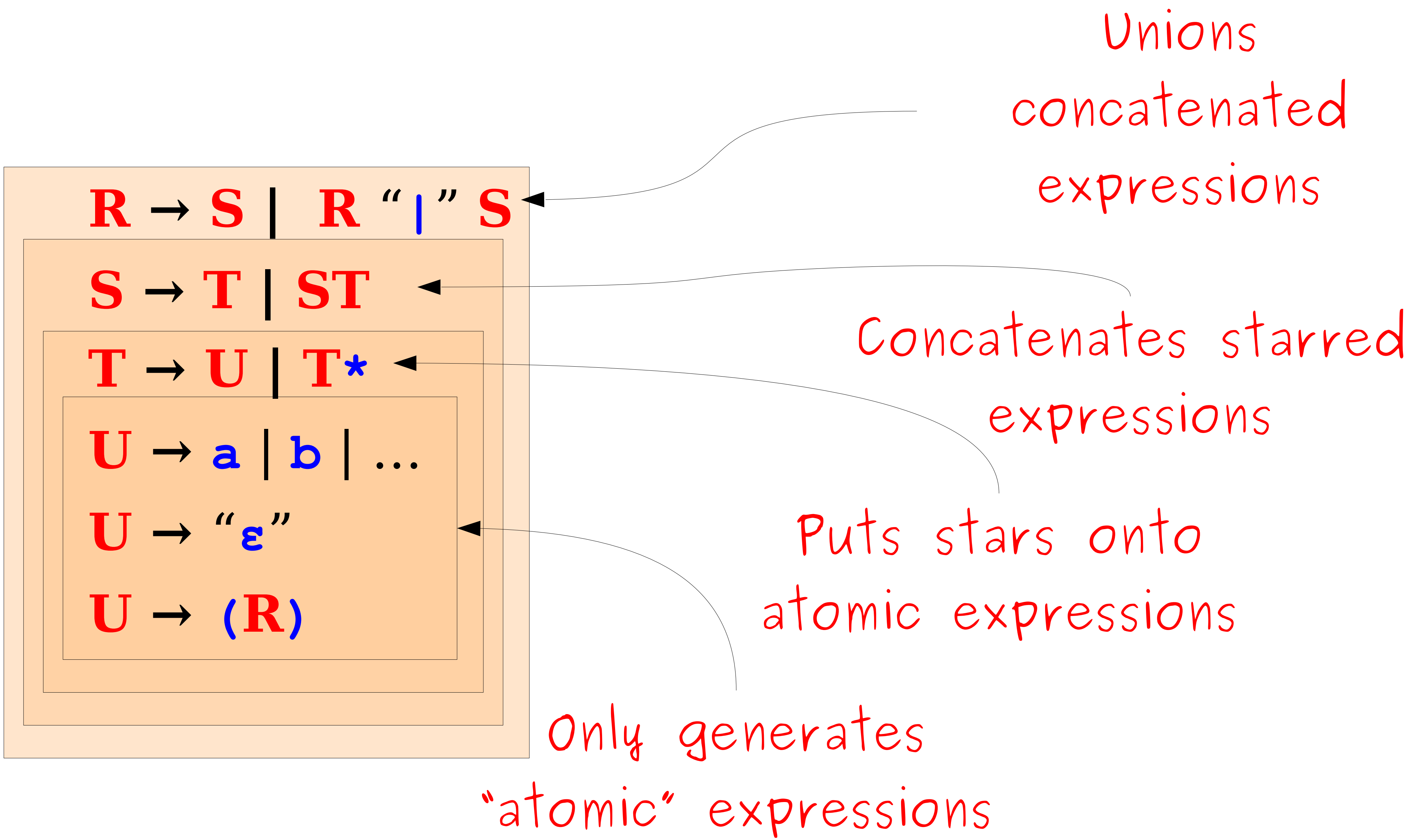
U → (R)

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**

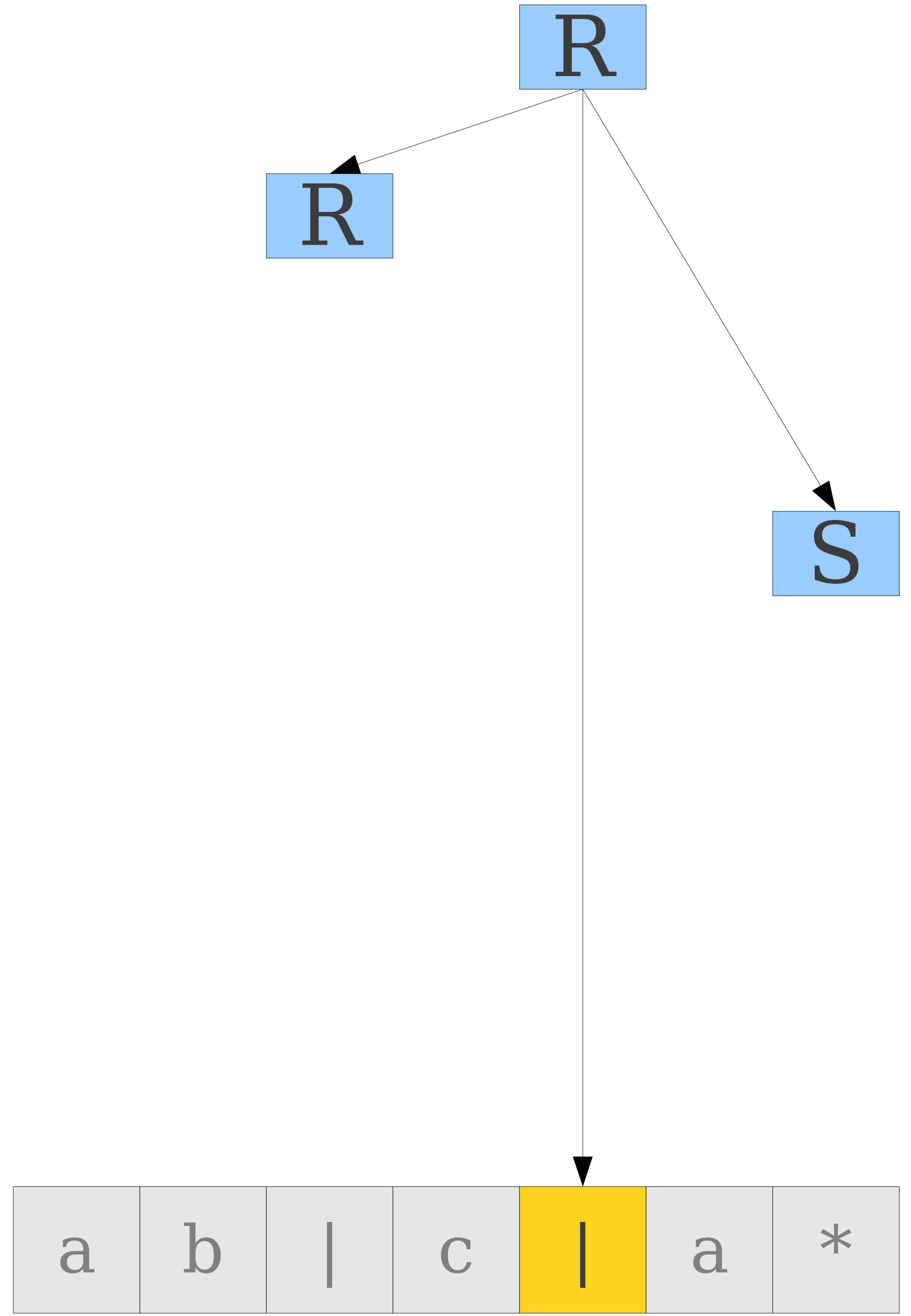
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → “ε”

U → (**R**)



R → **S** | **R** "|" **S**

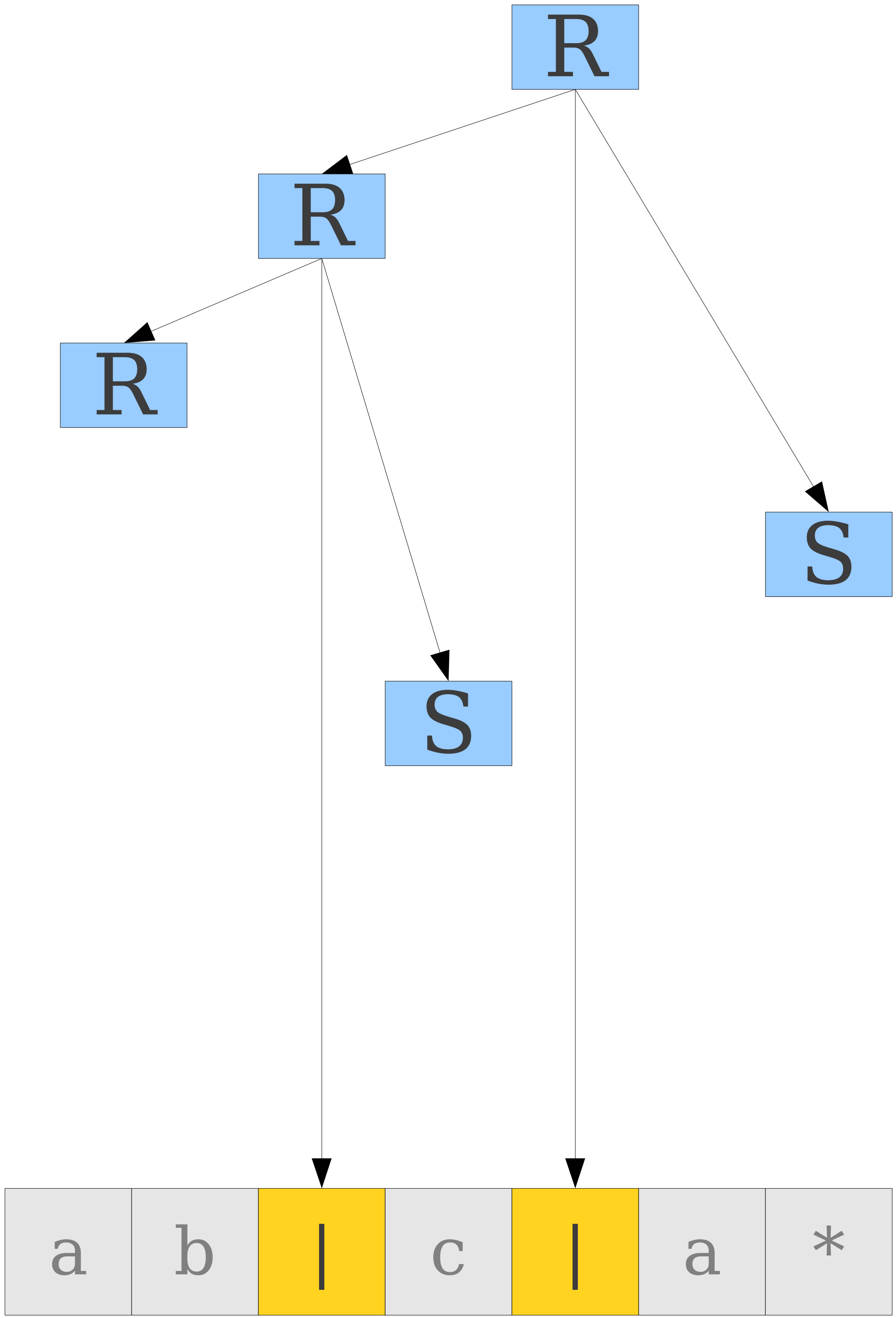
S → **T** | **ST**

T → **U** | **T***

U → **a** | **b** | **c** | ...

U → "ε"

U → **(R)**



R → **S** | **R** “|” **S**

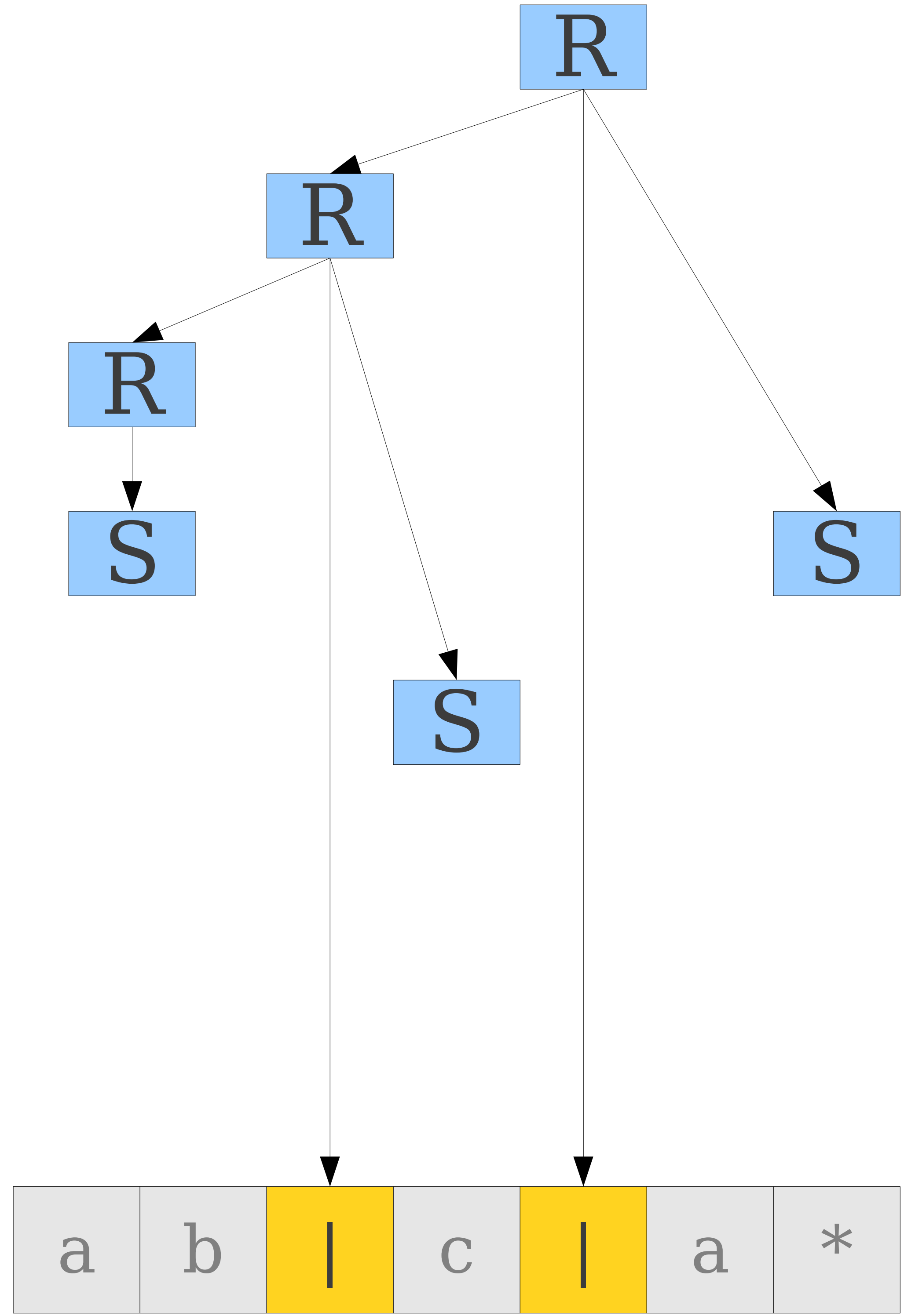
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | …

U → “ε”

U → (**R**)



R → **S** | **R** "|" **S**

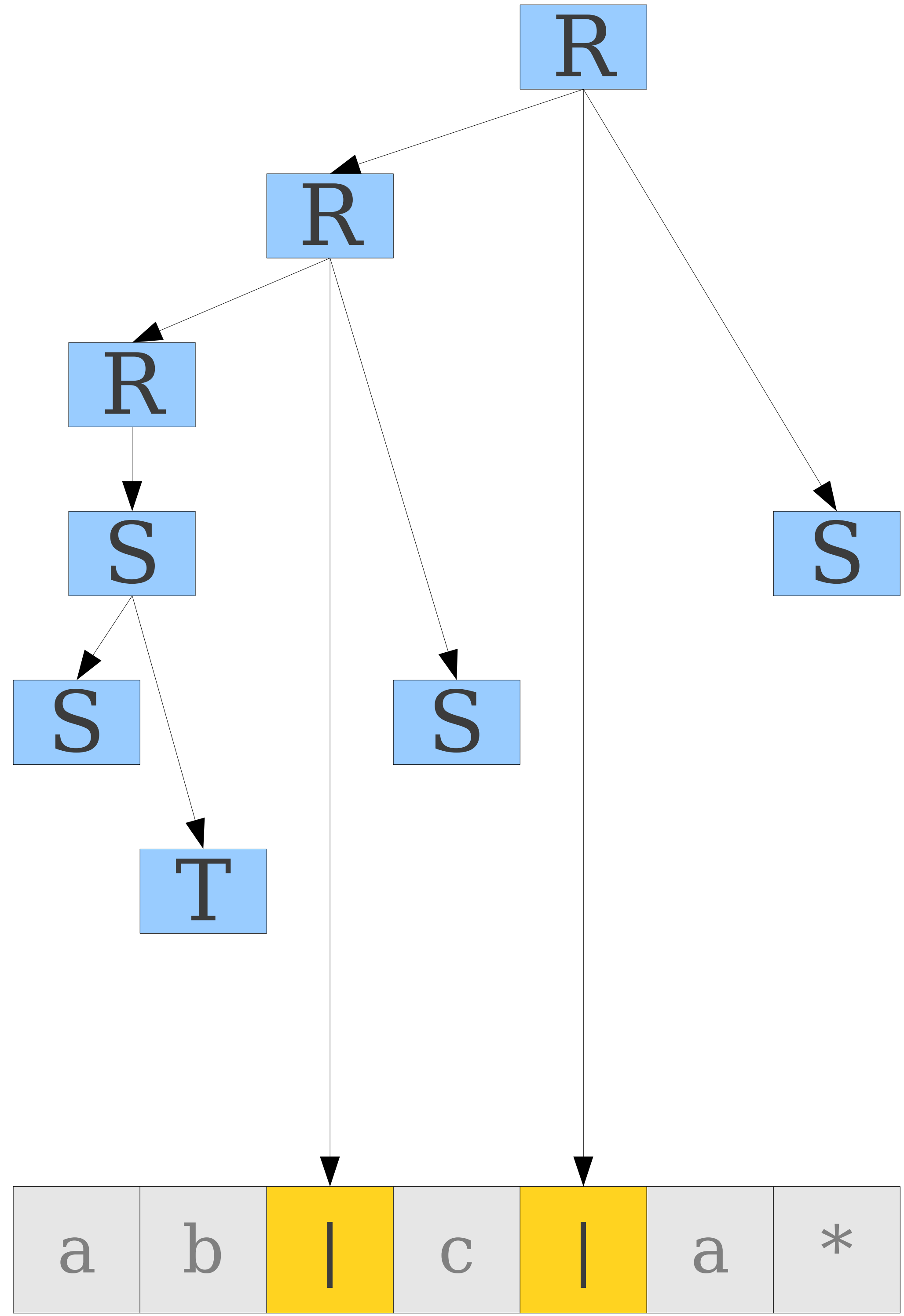
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → S | R “|” S

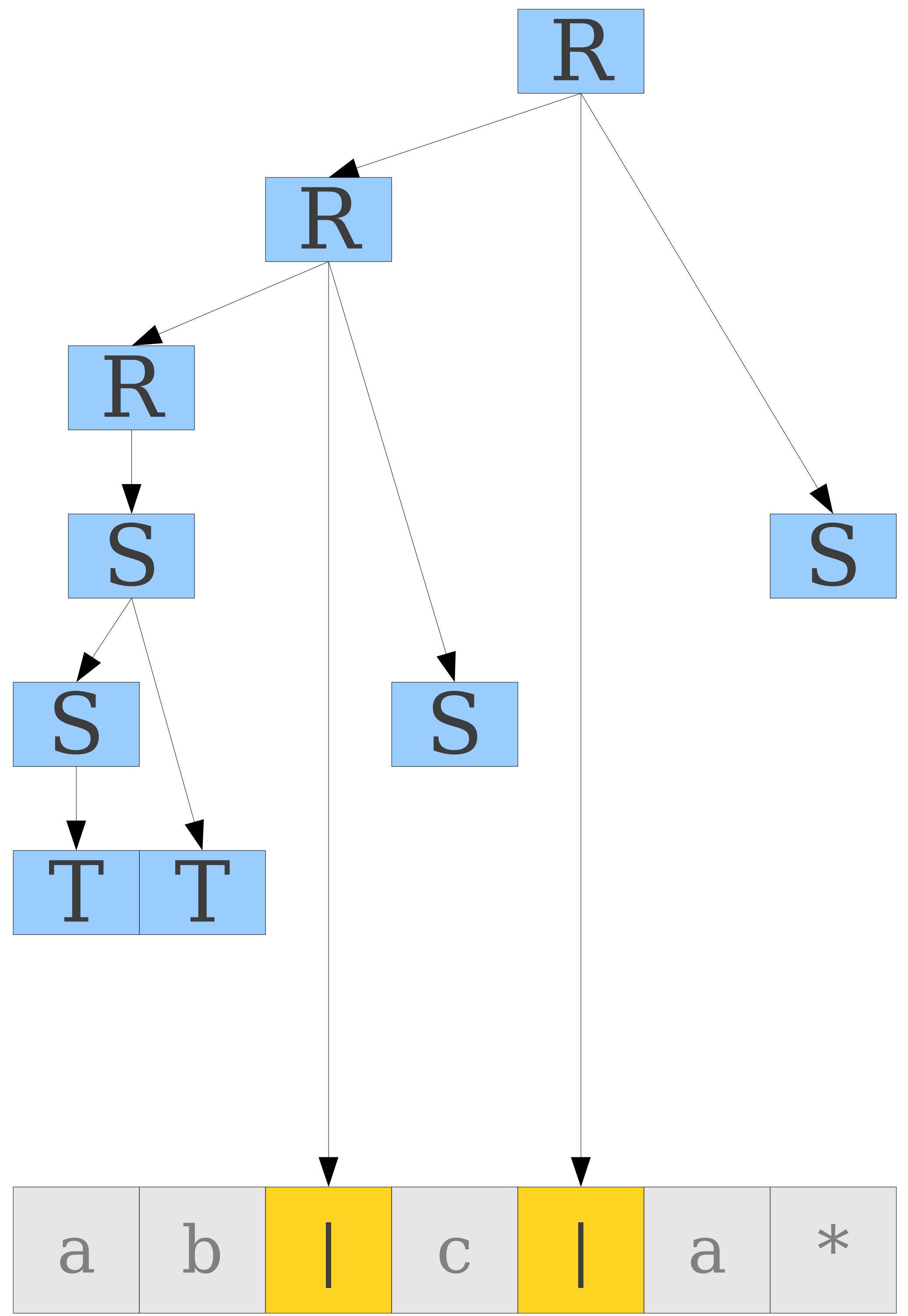
S → T | ST

T → U | T*

U → a | b | c | ...

U → “ε”

U → (R)



R → **S** | **R** "|" **S**

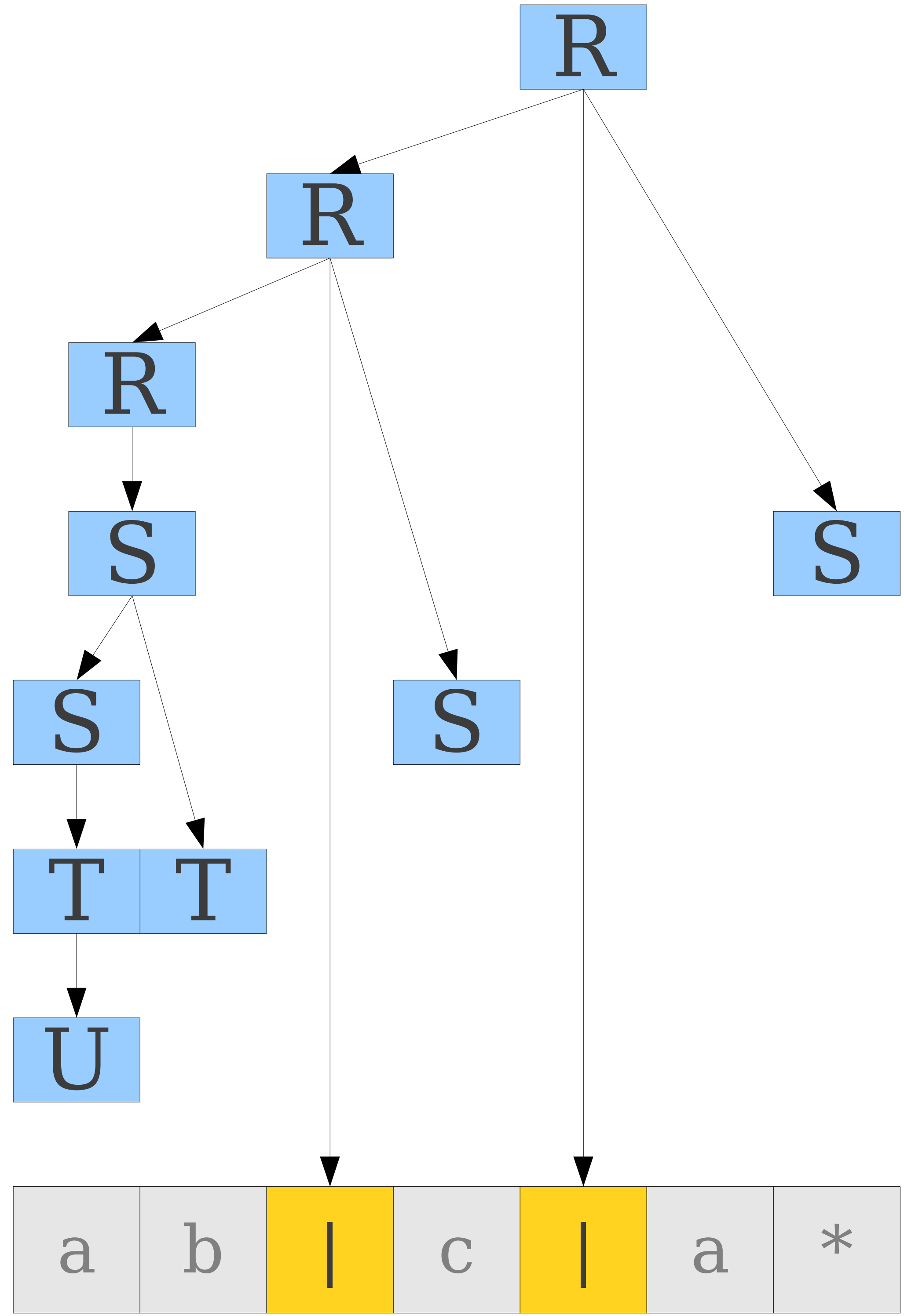
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → **S** | **R** “|” **S**

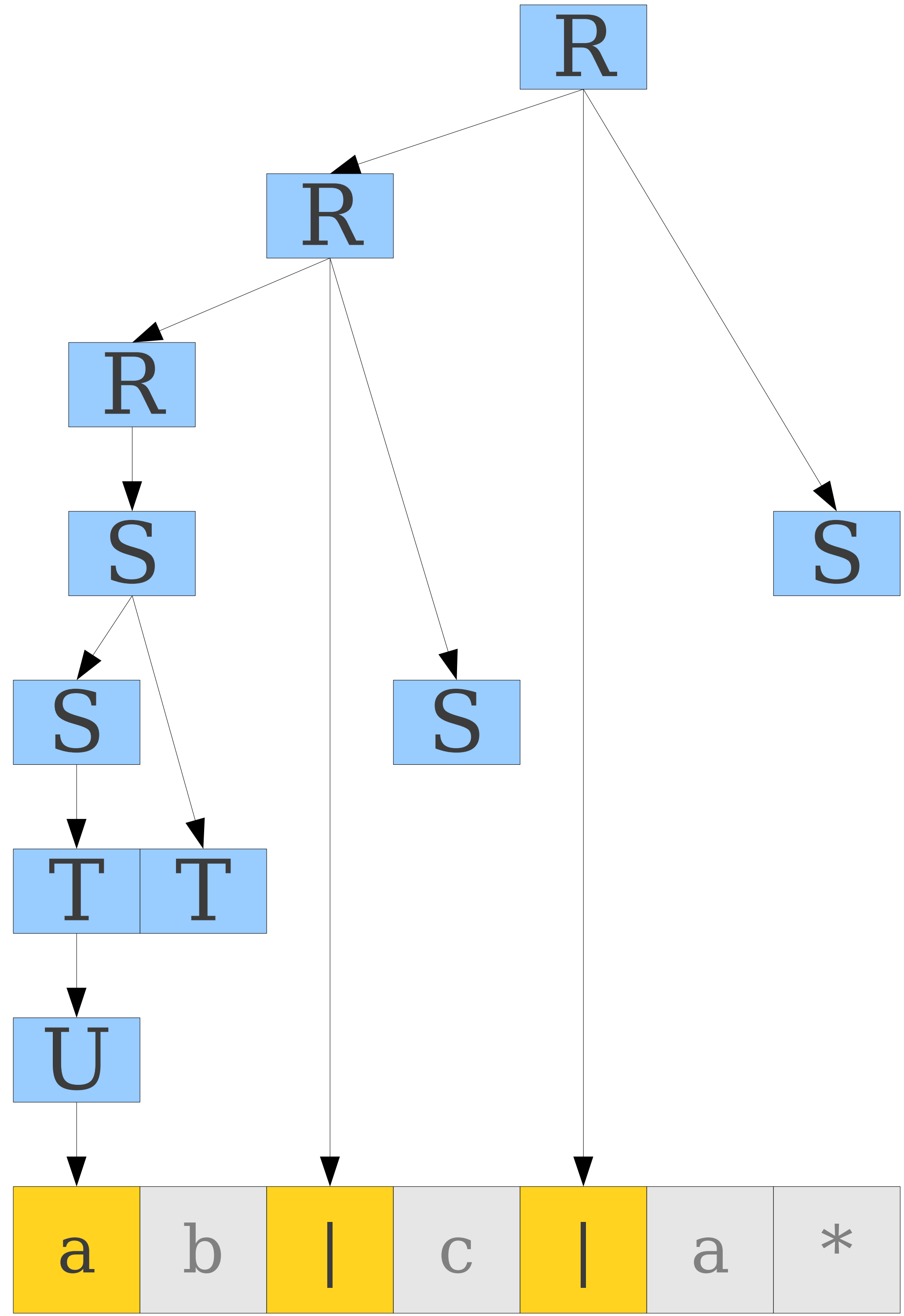
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → “ε”

U → (**R**)



R → **S** | **R** "|" **S**

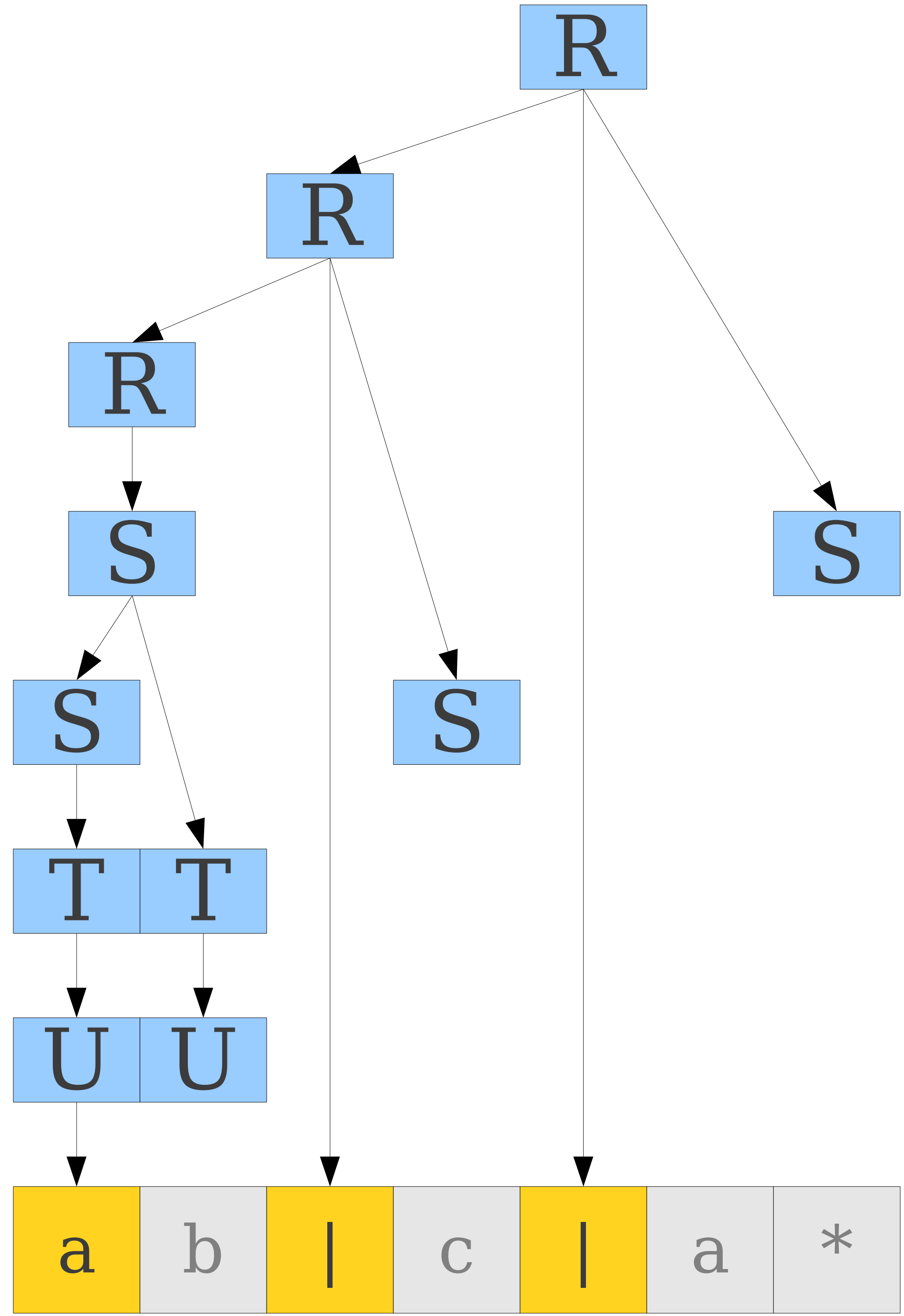
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → **S** | **R** "|" **S**

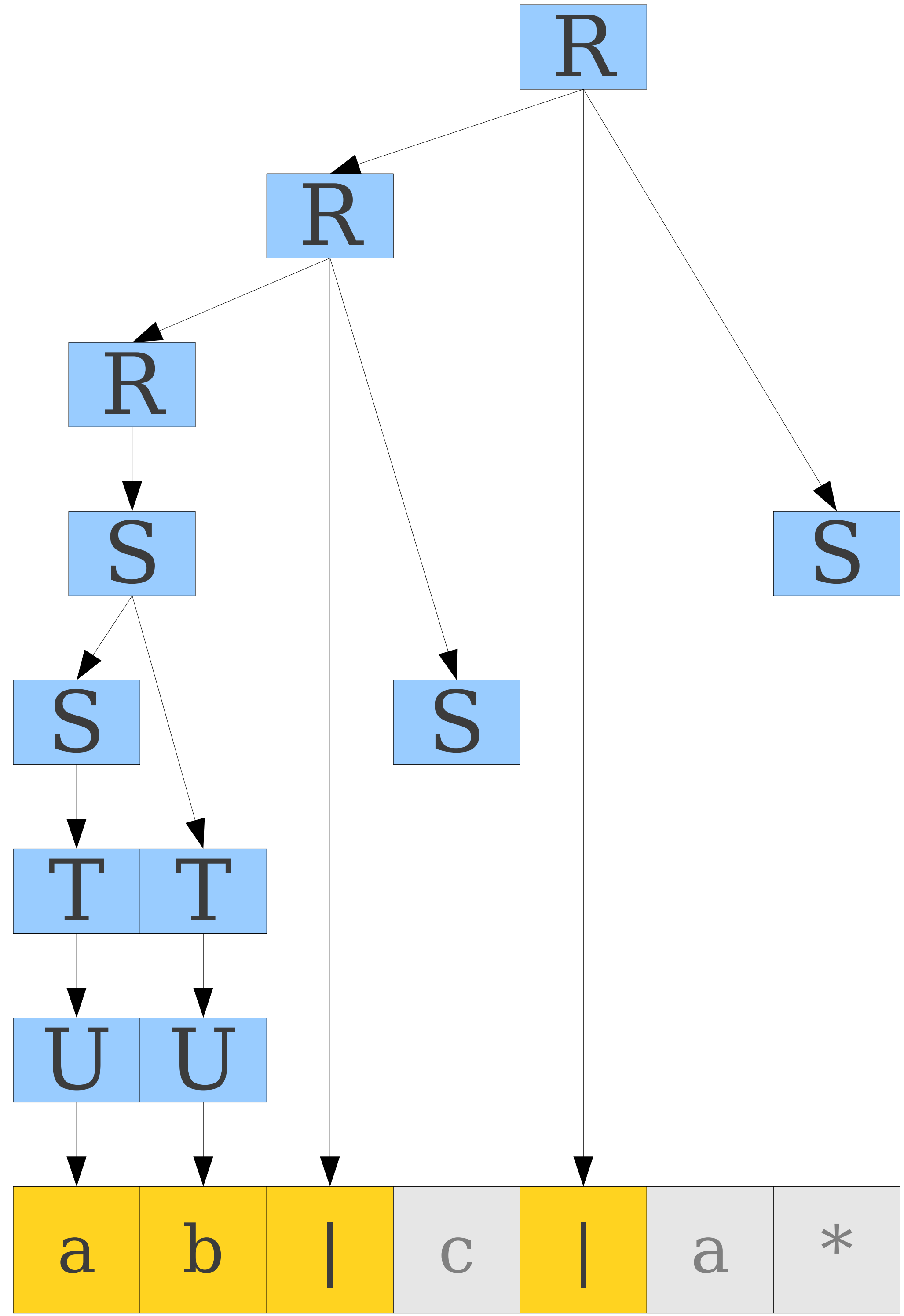
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → **S** | **R** “|” **S**

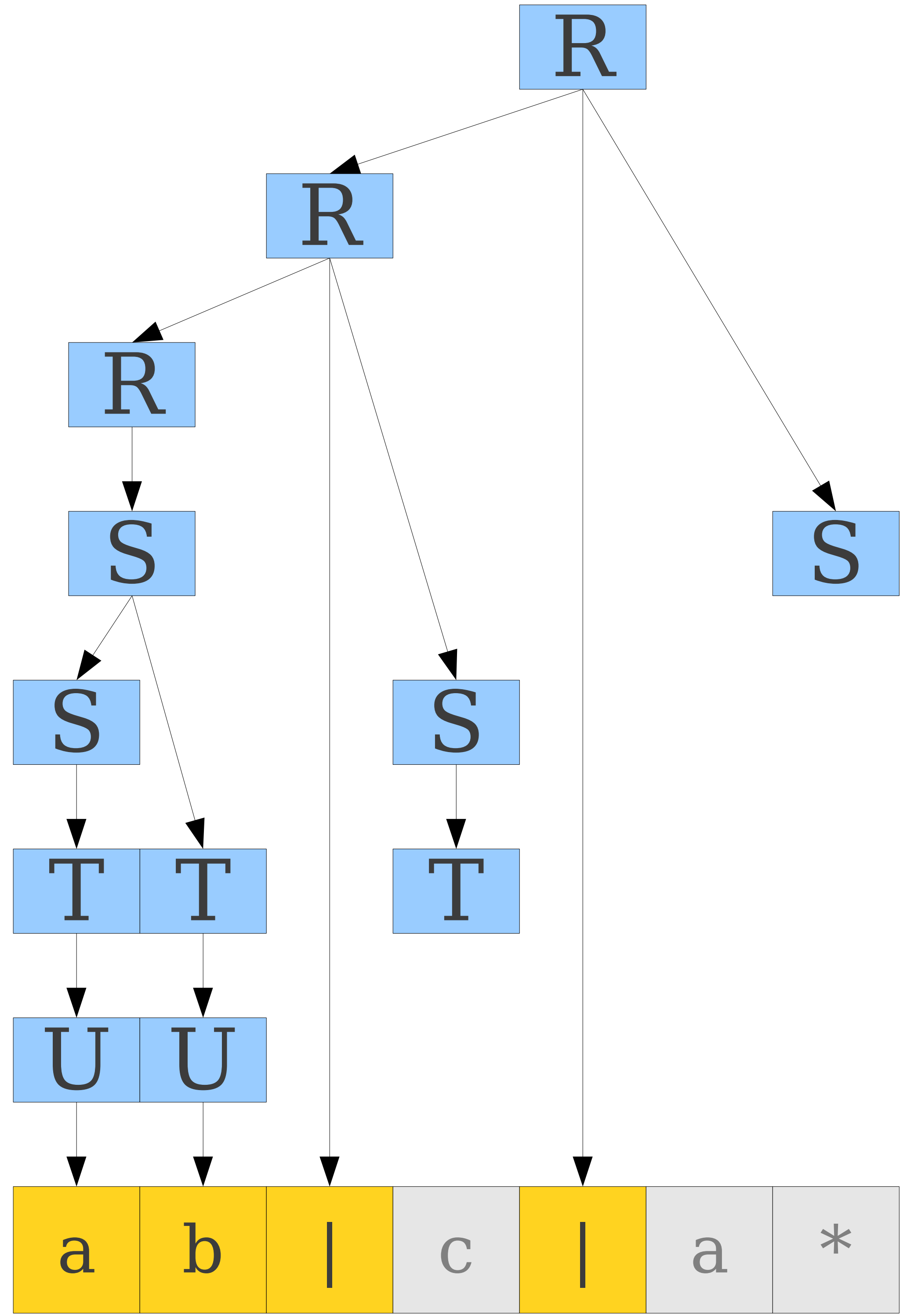
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → “ε”

U → (**R**)



R → **S** | **R** "|" **S**

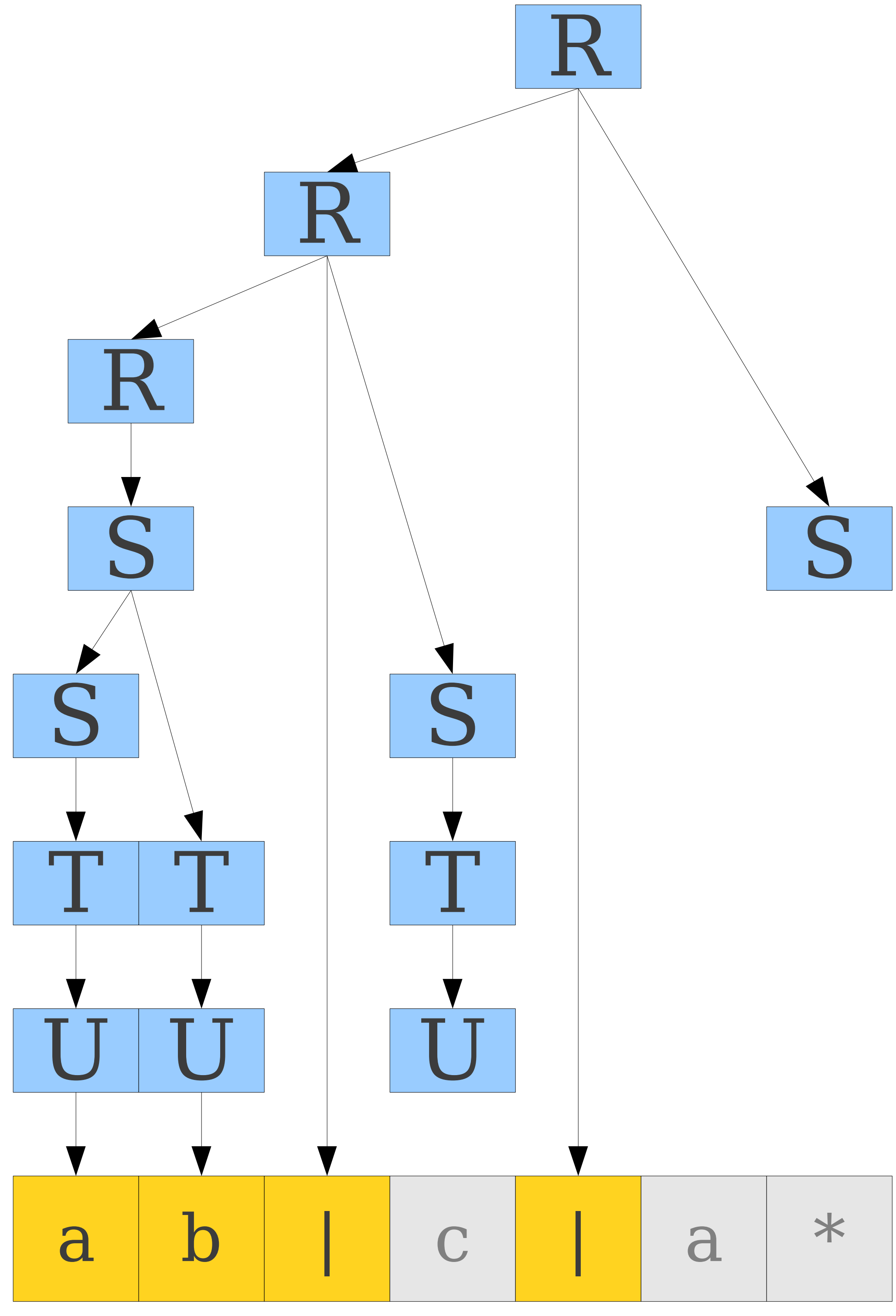
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → S | R “|” S

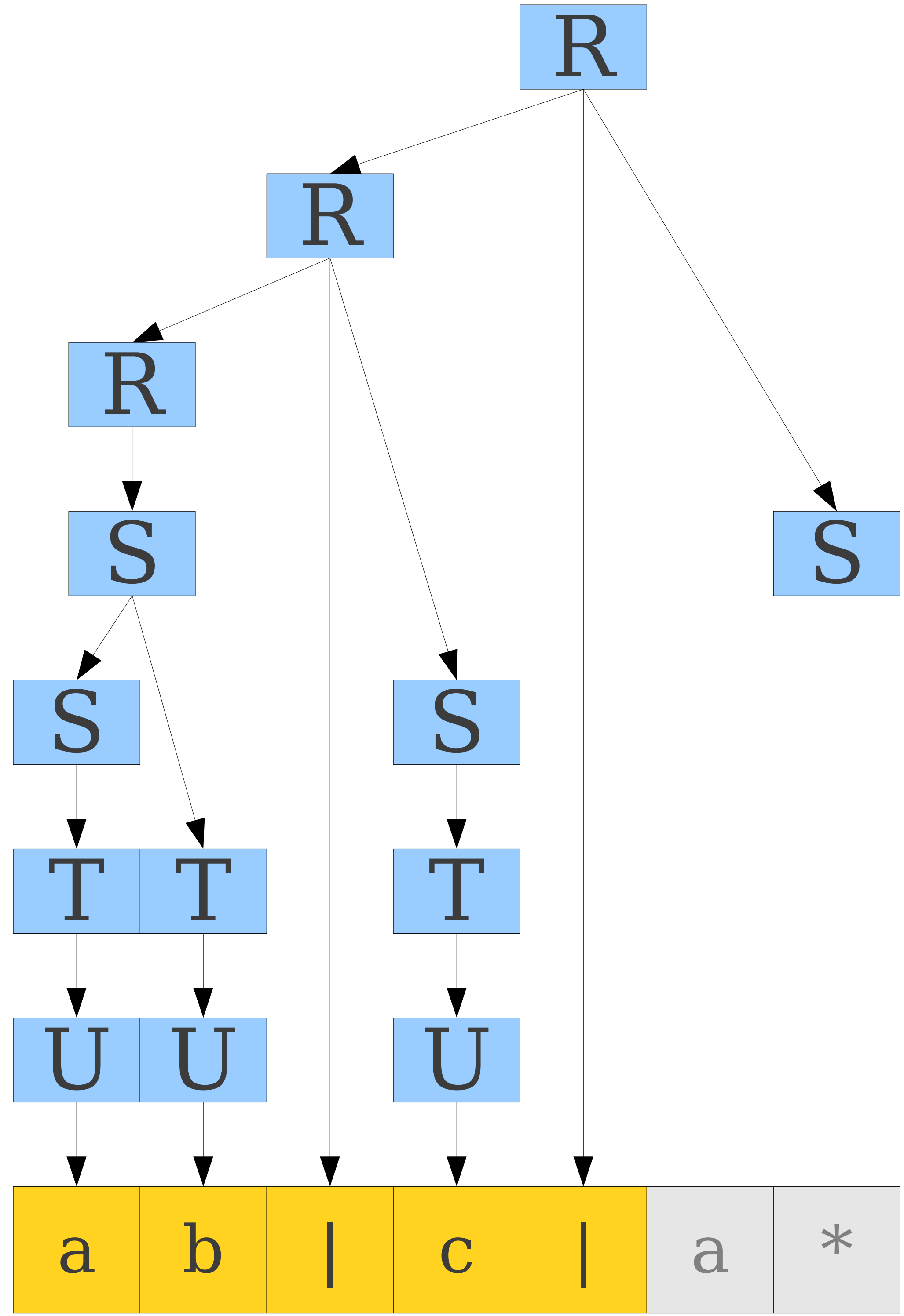
S → T | ST

T → U | T*

U → a | b | c | ...

U → “ε”

U → (R)



R → S | R "|" S

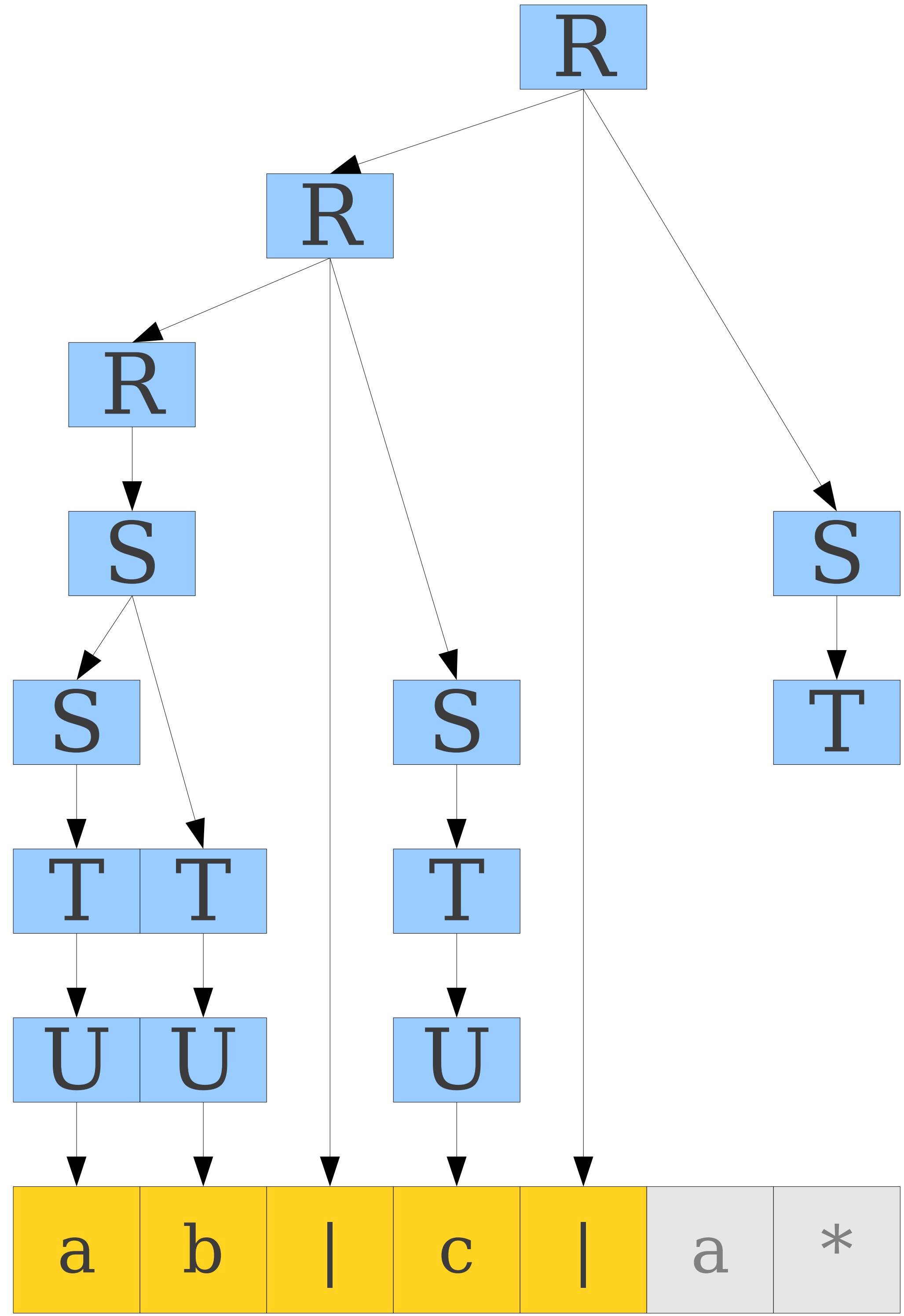
S → T | ST

T → U | T*

U → a | b | c | …

U → "ε"

U → (R)



R → S | R "|" S

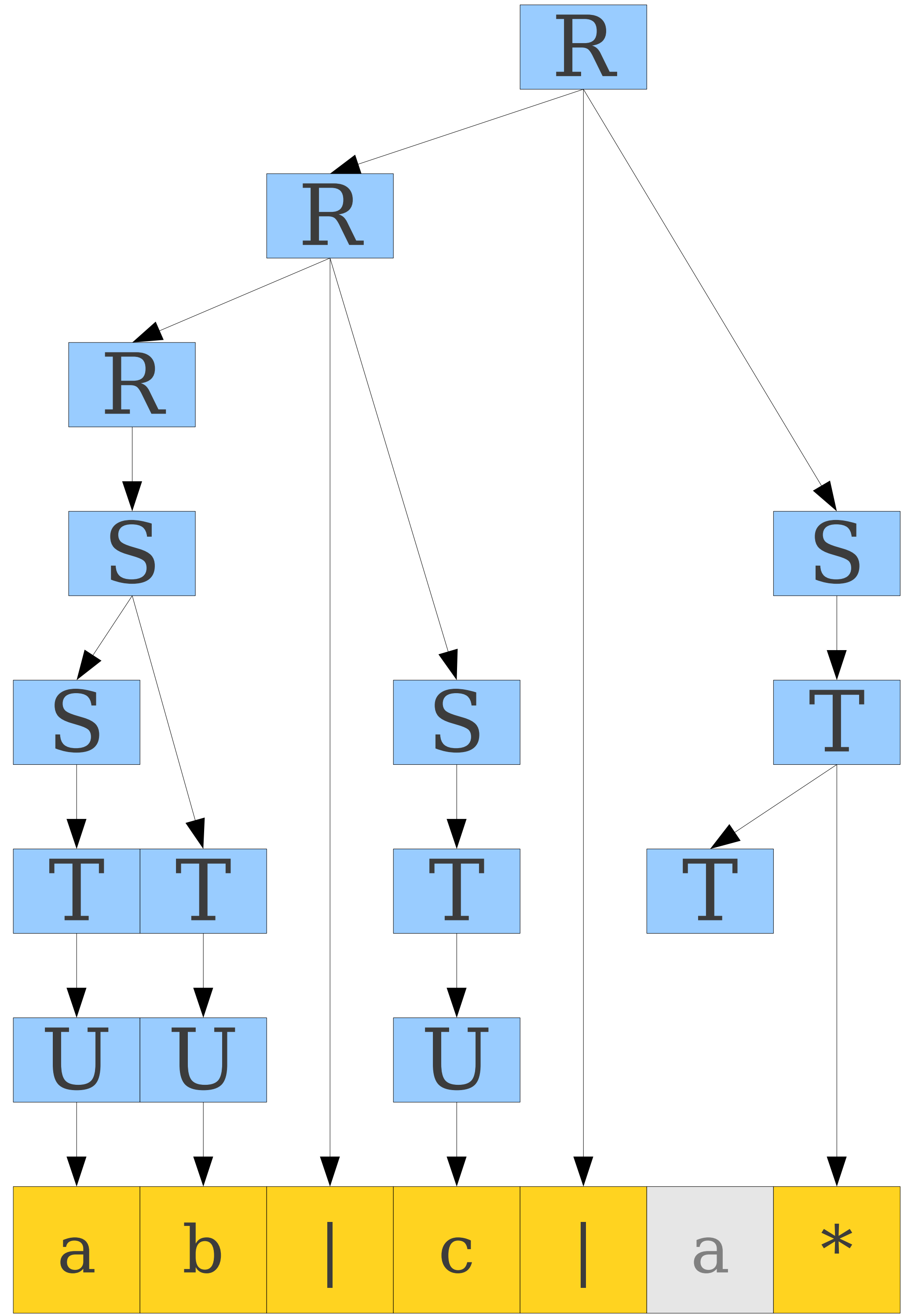
S → T | ST

T → U | T*

U → a | b | c | …

U → "ε"

U → (R)



R → **S** | **R** “|” **S**

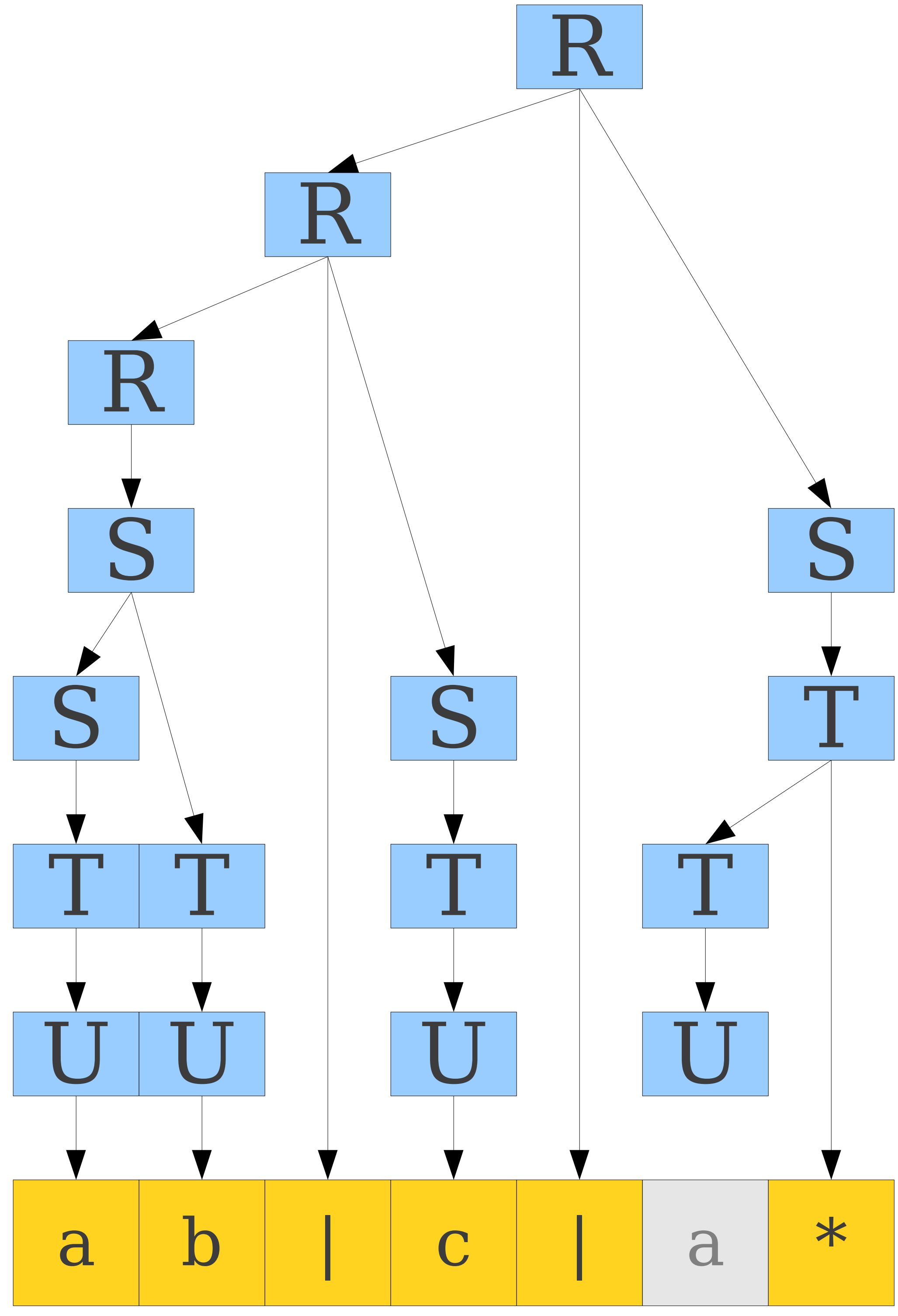
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | …

U → “ε”

U → (**R**)



R → **S** | **R** "|" **S**

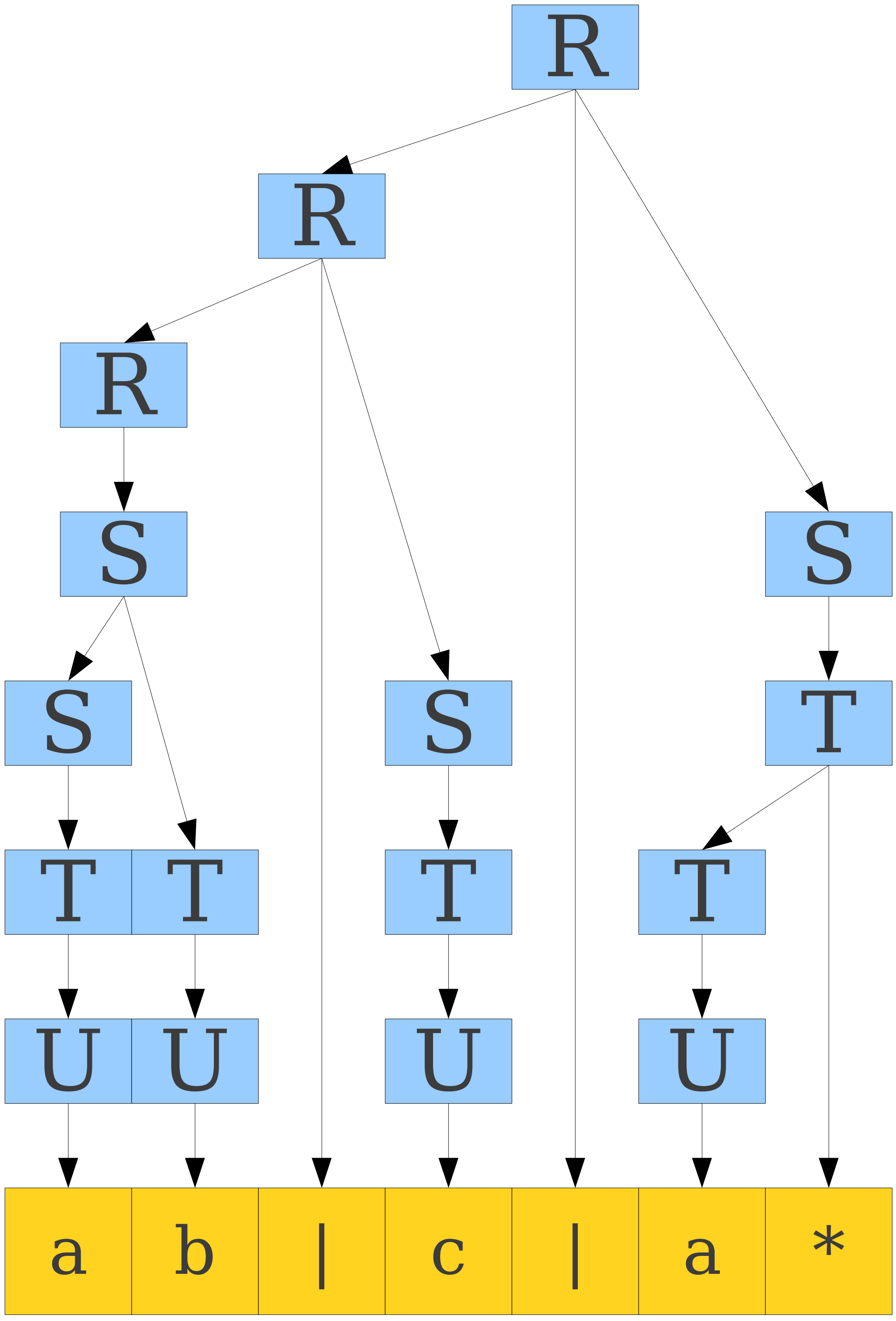
S → **T** | **ST**

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.

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**)

a	a	|	b	*

The Structure of a Parse Tree

R → **S** | **R** “`|`” **S**

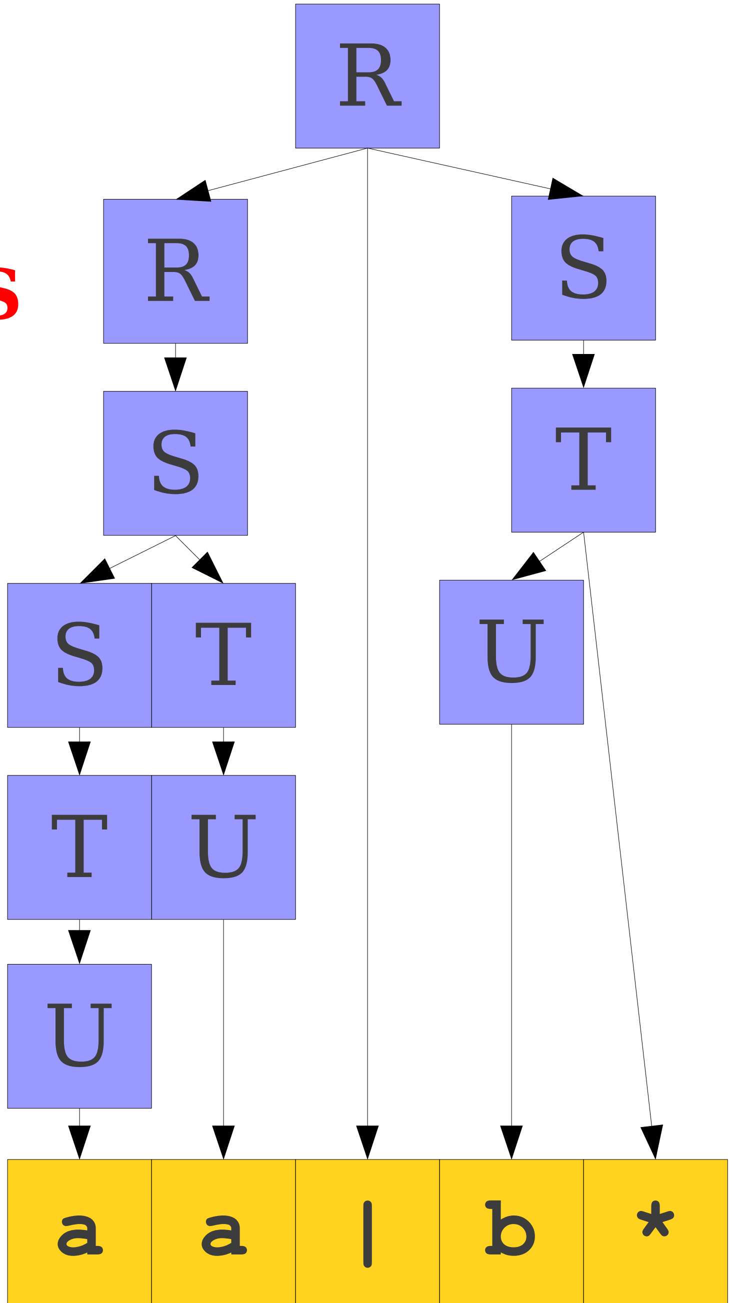
S → **T** | **ST**

T → **U** | **T**`*`

U → `a` | `b` | …

U → “`ε`”

U → `(`**R**`)`



The Structure of a Parse Tree

R → **S** | **R** "|" **S**

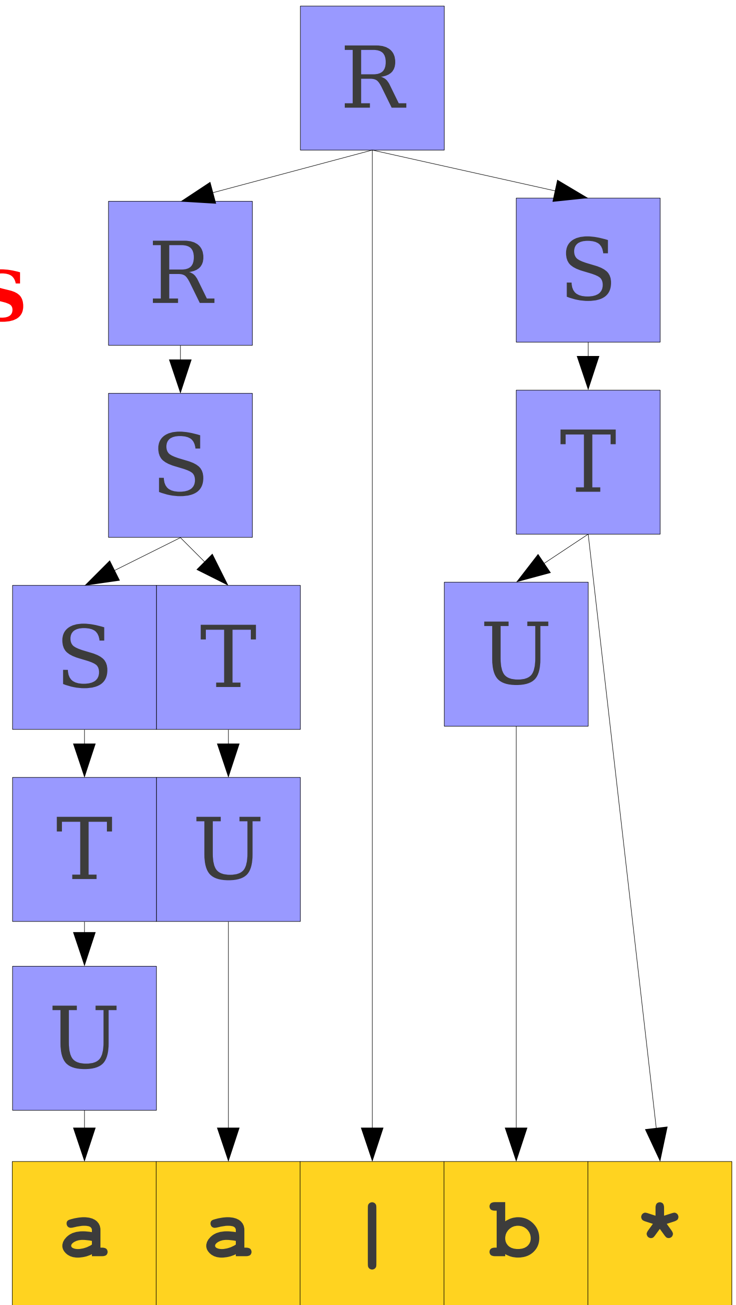
S → **T** | **ST**

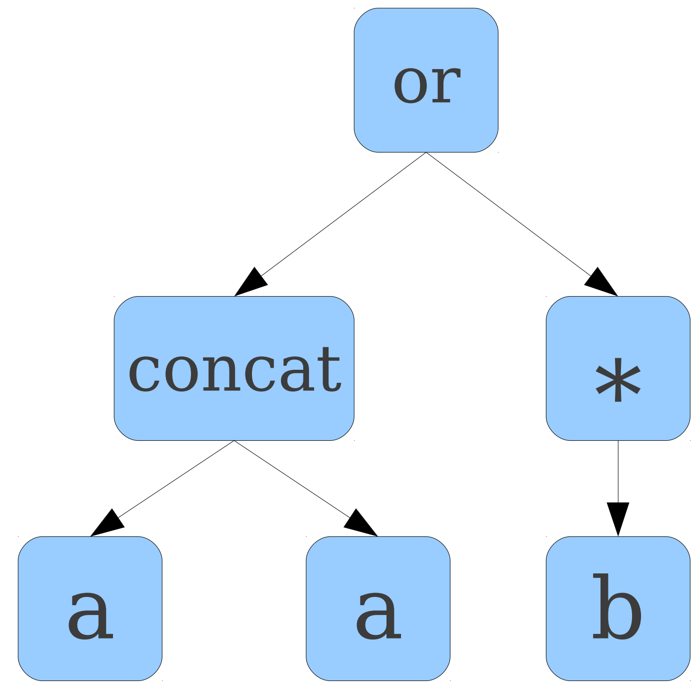
T → **U** | **T***

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**

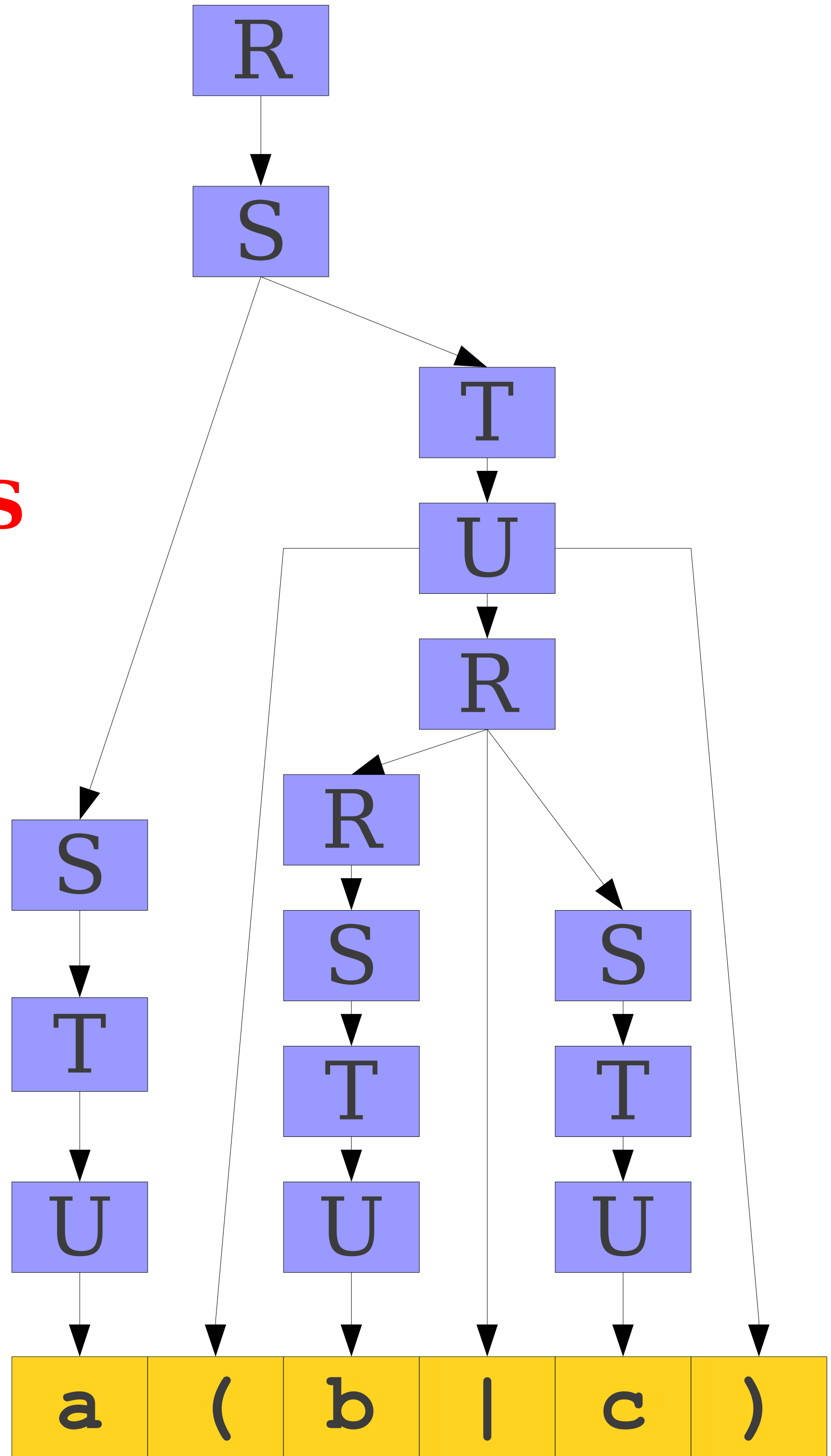
S → **T** | **ST**

T → **U** | **T***

U → a | b | …

U → "ε"

U → (**R**)



R → S | R "|" S

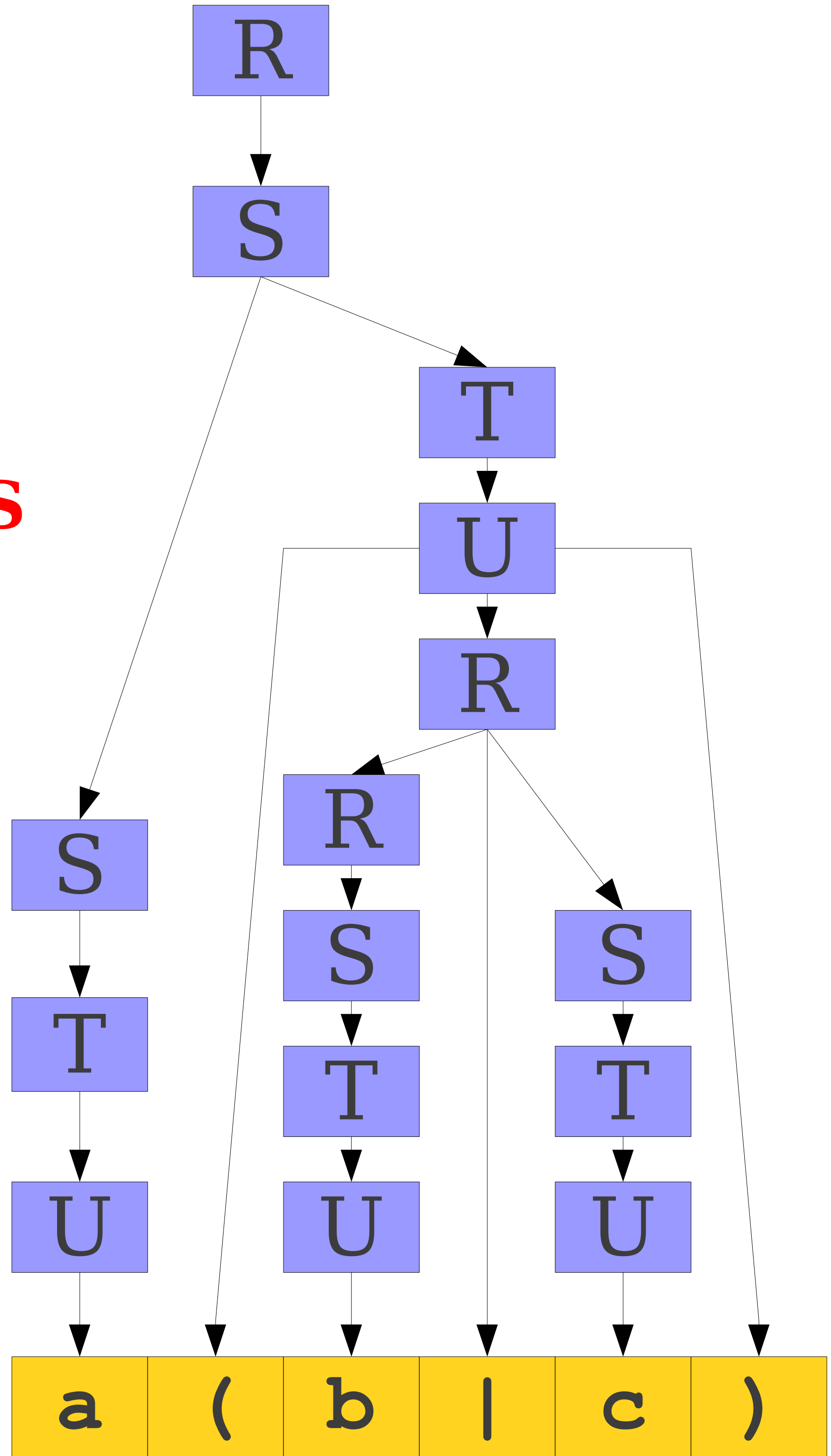
S → T | ST

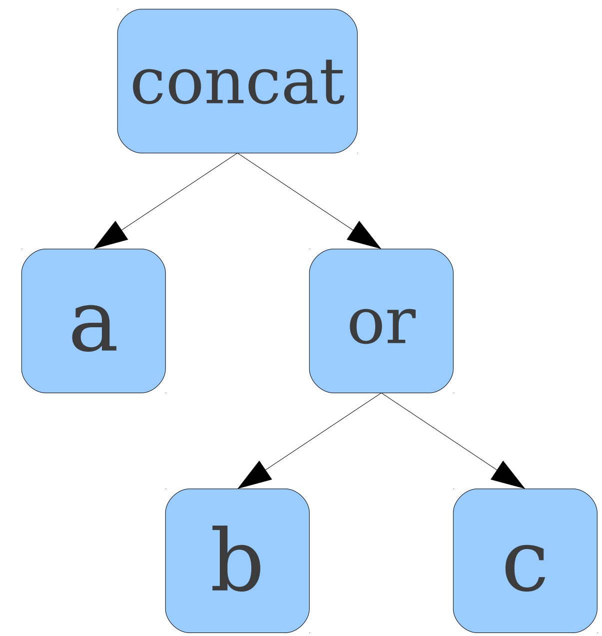
T → U | T*

U → a | b | …

U → "ε"

U → (R)





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.

How to build an AST?

- Typically done through **semantic actions**.
- Associate a piece of code to execute with each production.
- As the input is parsed, execute this code to build the AST.
  - Exact order of code execution depends on the parsing method used.
- This is called a **syntax-directed translation**.

Program fragments embedded within production bodies are called semantic actions. The position at which an action is to be executed is shown by enclosing it between curly braces and writing it within the production body, as in

rest —> + term {print('+')} rest1

Simple Semantic Actions

$E \rightarrow T + E$	E_1.val = T.val + E_2.val
$E \rightarrow T$	E.val = T.val
$T \rightarrow$ **int**	T.val = int.val
$T \rightarrow$ **int** * T	T.val = int.val * T.val
$T \rightarrow$ **(**E**)**	T.val = E.val

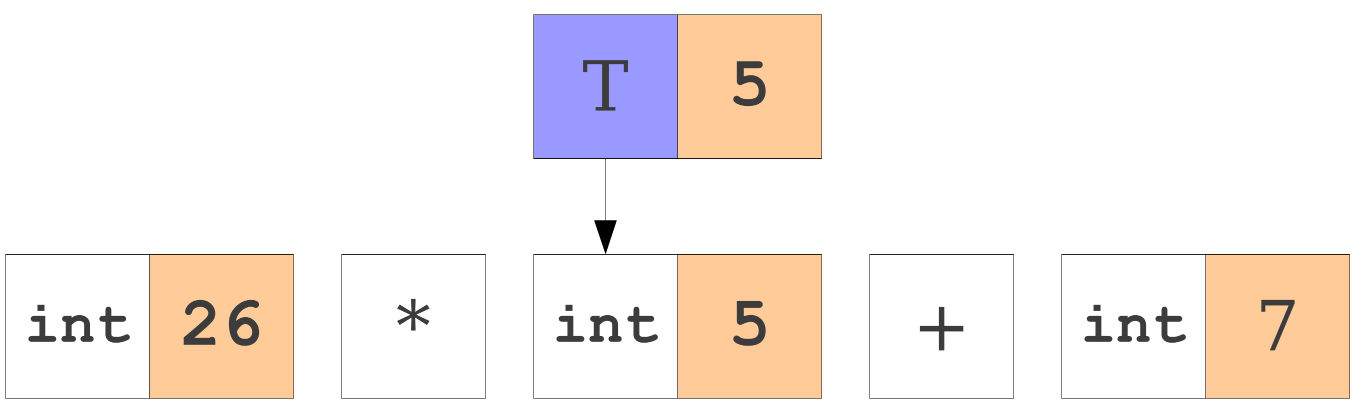
Simple Semantic Actions

E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → int	T.val = int.val
T → int * T	T.val = int.val * T.val
T → (E)	T.val = E.val

int	26	*	int	5	+	int	7

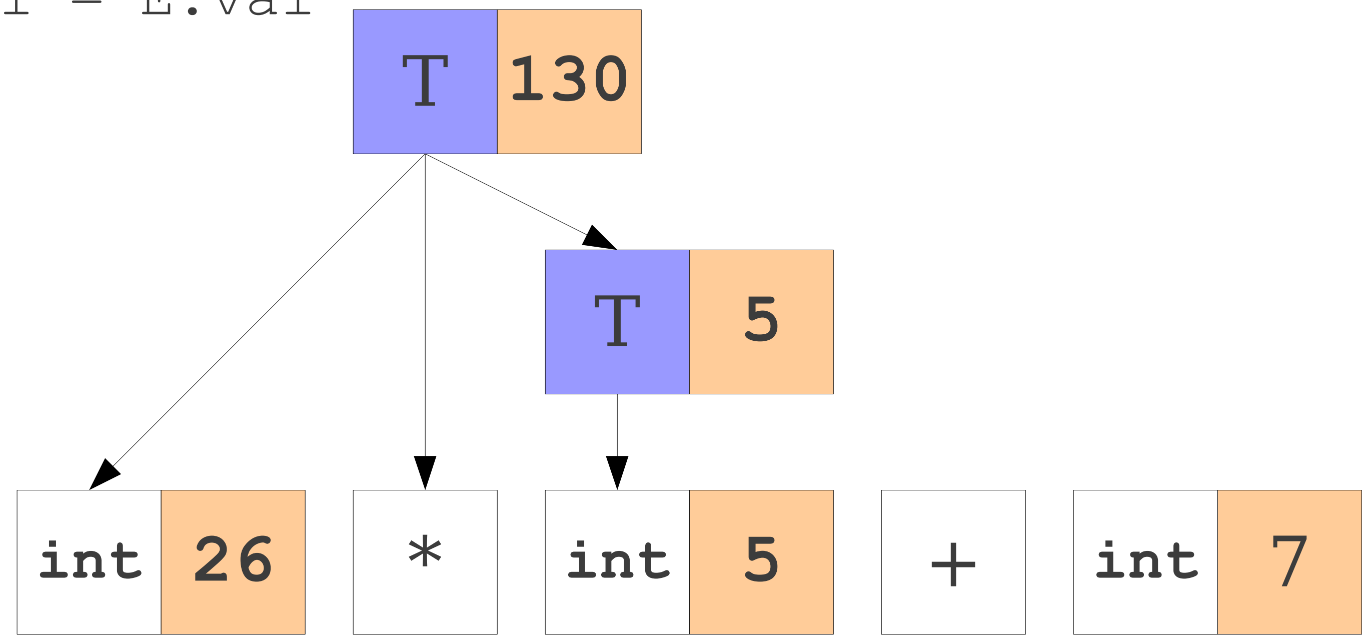
Simple Semantic Actions

E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → int	T.val = int.val
T → int * T	T.val = int.val * T.val
T → (E)	T.val = E.val



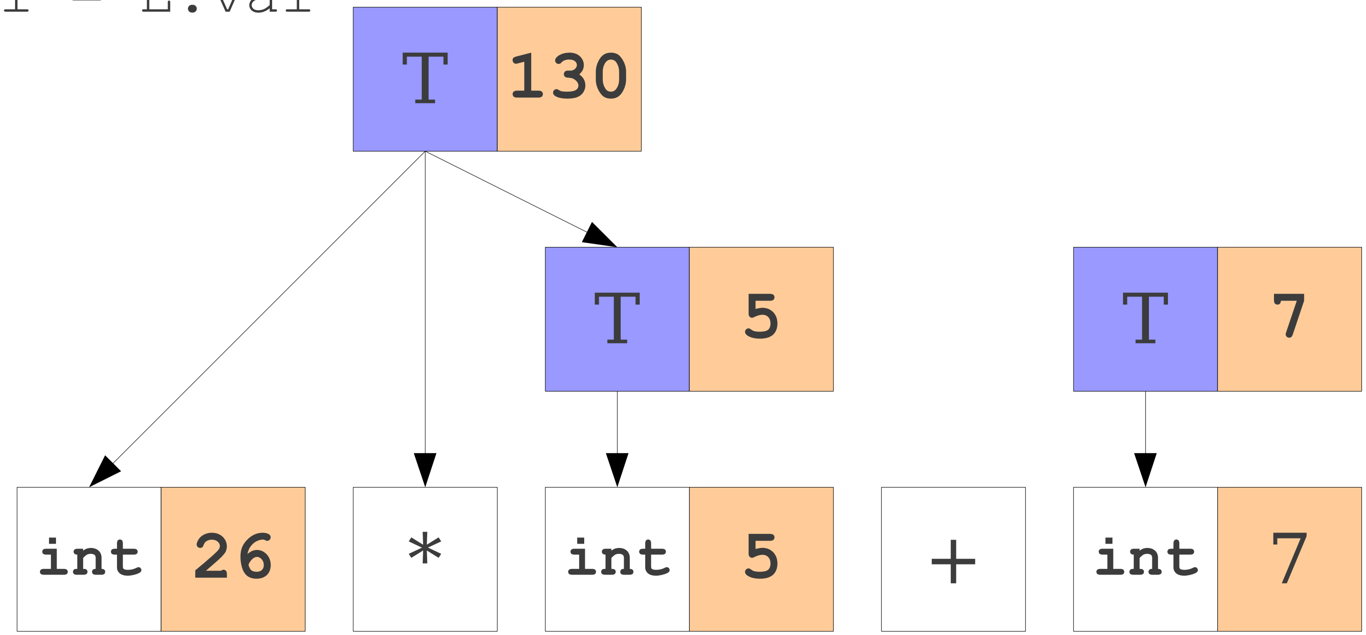
Simple Semantic Actions

E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → int	T.val = int.val
T → int * T	T.val = int.val * T.val
T → (E)	T.val = E.val



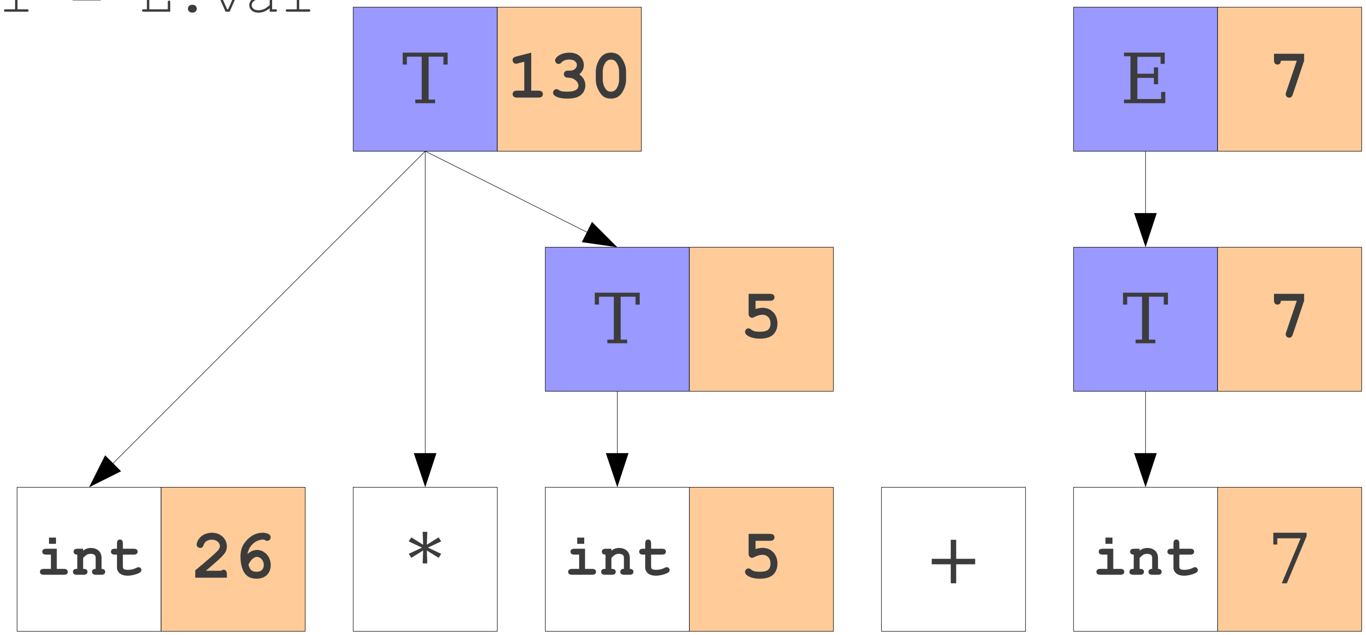
Simple Semantic Actions

E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → int	T.val = int.val
T → int * T	T.val = int.val * T.val
T → (E)	T.val = E.val



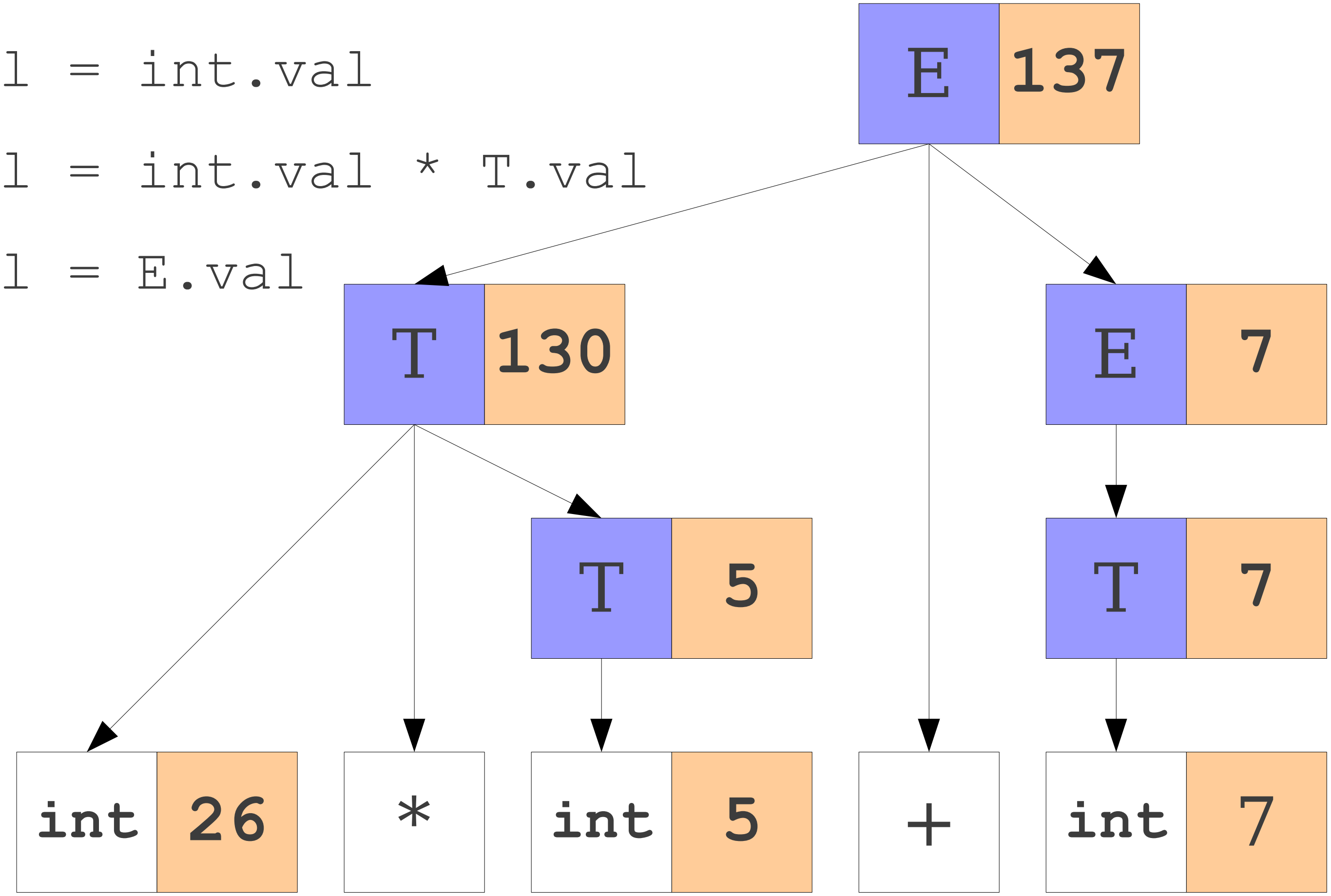
Simple Semantic Actions

E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → int	T.val = int.val
T → int * T	T.val = int.val * T.val
T → (E)	T.val = E.val



Simple Semantic Actions

E → T + E	E_1.val = T.val + E_2.val
E → T	E.val = T.val
T → int	T.val = int.val
T → int * T	T.val = int.val * T.val
T → (E)	T.val = E.val



Semantic Actions to Build ASTs

$R \rightarrow S$	`R.ast = S.ast;`
$R \rightarrow R$ "|" S	R_1`.ast = new Or(`R_2`.ast, S.ast);`
$S \rightarrow T$	`S.ast = T.ast;`
$S \rightarrow ST$	S_1`.ast = new Concat(`S_2`.ast, T.ast);`
$T \rightarrow U$	`T.ast = U.ast;`
$T \rightarrow T*$	T_1`.ast = new Star(`T_2`.ast);`
$U \rightarrow a$	`U.ast = new SingleChar('a');`
$U \rightarrow$ "ε"	`U.ast = new Epsilon();`
$U \rightarrow (R)$	`U.ast = R.ast;`

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.

Next Time

- **Top-Down Parsing**
  - Parsing as a Search
  - Backtracking Parsers
  - Predictive Parsers
  - LL(1)