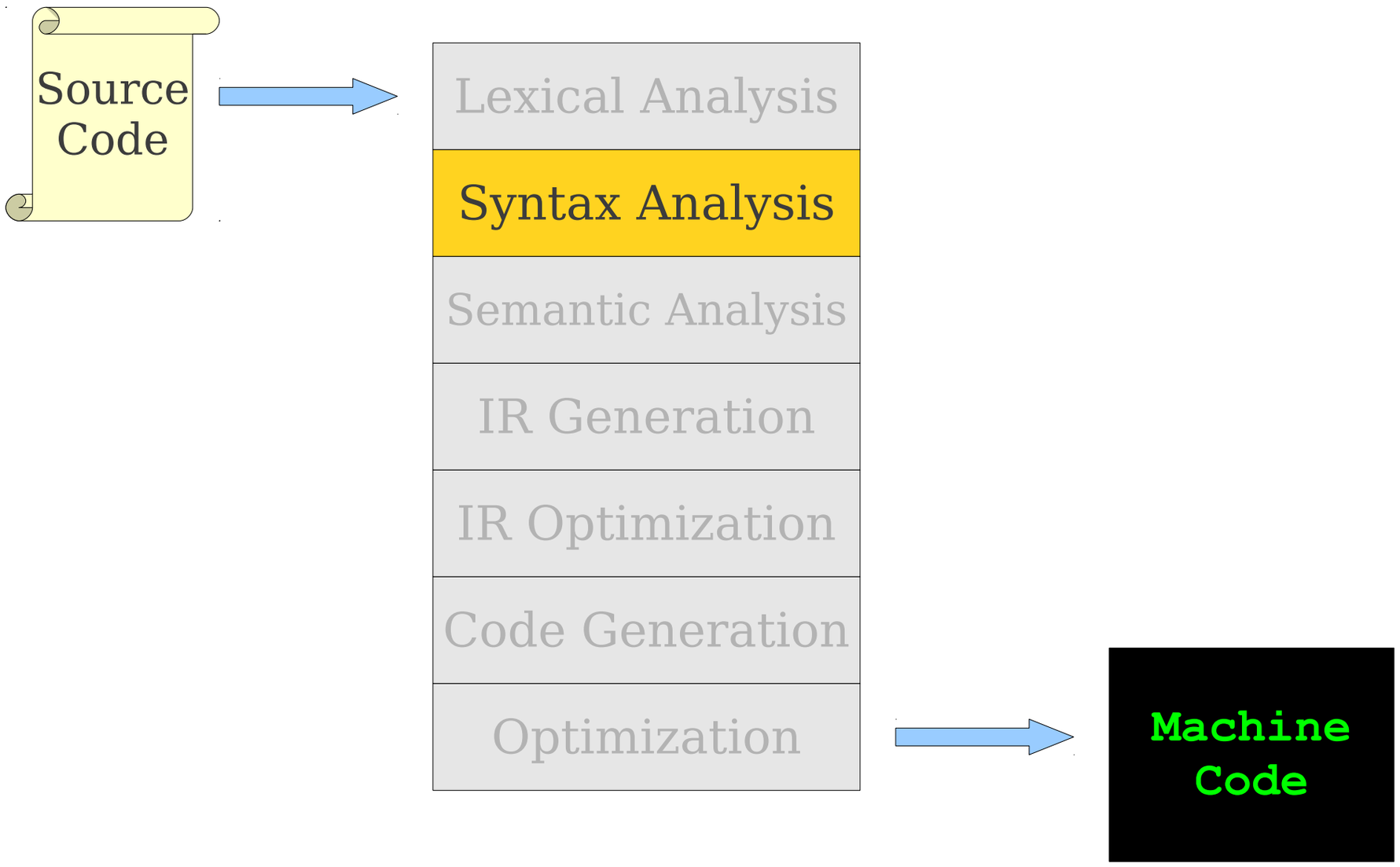
Syntax Analysis

- Introduction to parsing

Feb 1, 2021

Where We Are


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

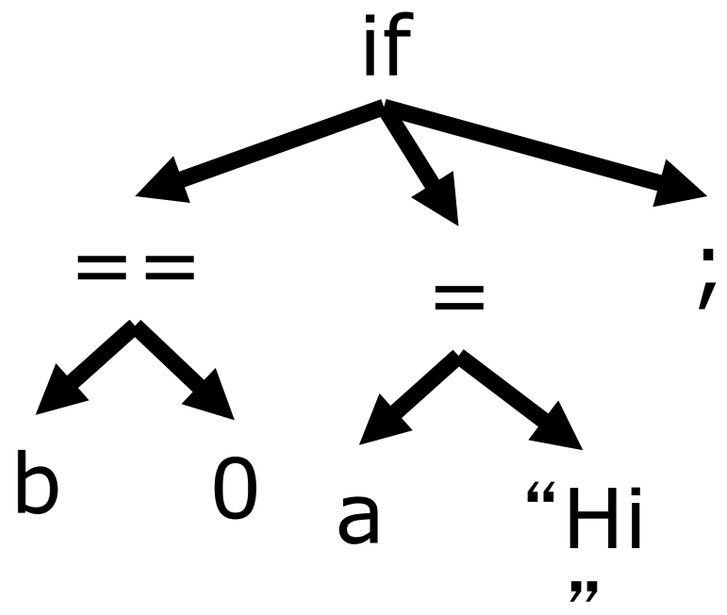
Overview of Syntax Analysis

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

Source code: if (b==0) a = “Hi”;

- Report errors if the tokens do not properly encode a structure

AST:



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

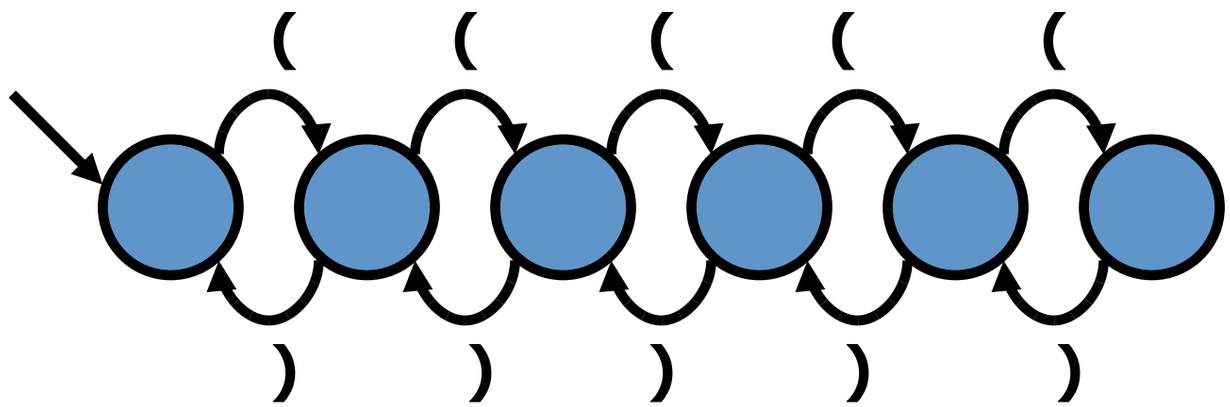
- Today: Formalism for syntax analysis
  - Grammars
  - Derivation
  - Ambiguity
  - Top down parsing algorithms

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 → (S) S
  - S → ε
- 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 → (S) S
- S → ε

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 → `int`

E → **E Op E**

E → `(`**E**`)`

Op → `+`

Op → `-`

Op → `*`

Op → `/`

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

- 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. α, γ, ω

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

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

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

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

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

- 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 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 → `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 Op E**

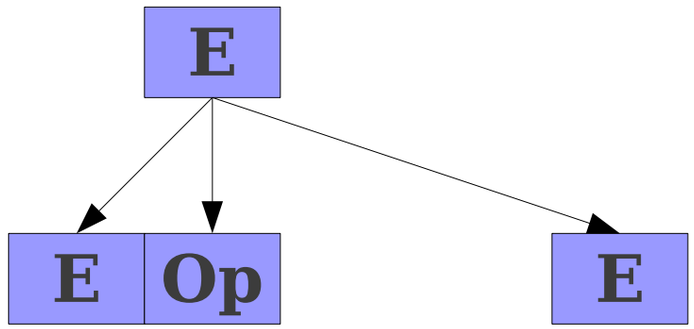
Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**



Parse Trees

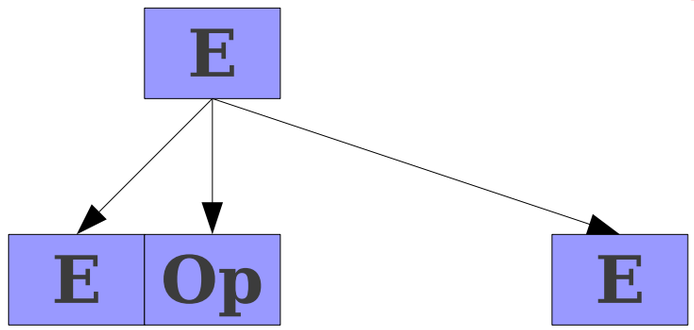
E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ `int` **Op E**



Parse Trees

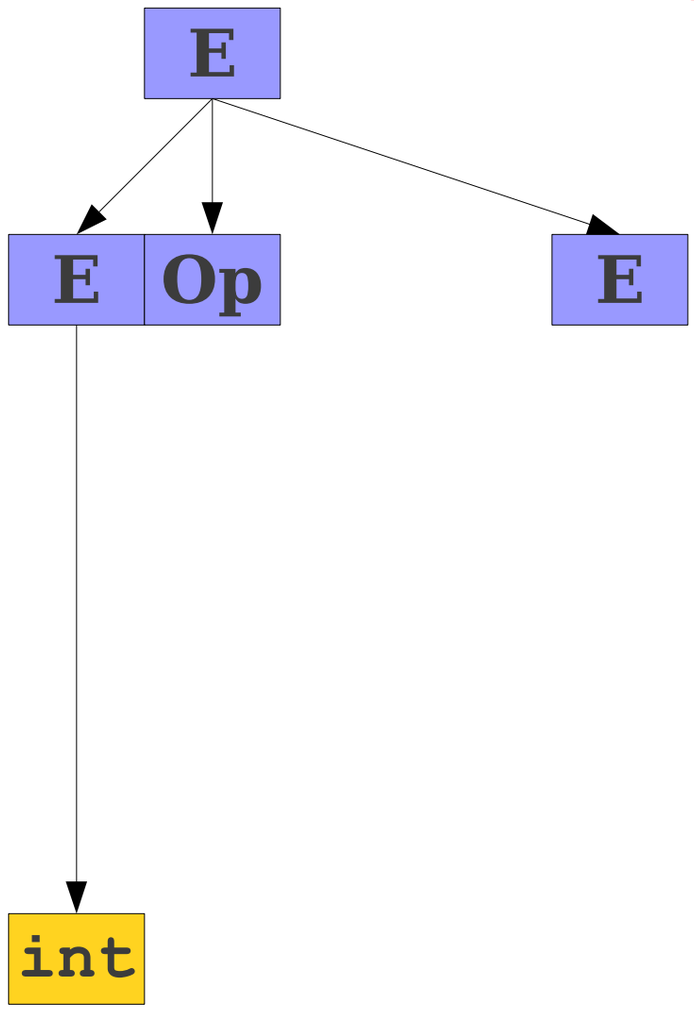
E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ `int` **Op E**



E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

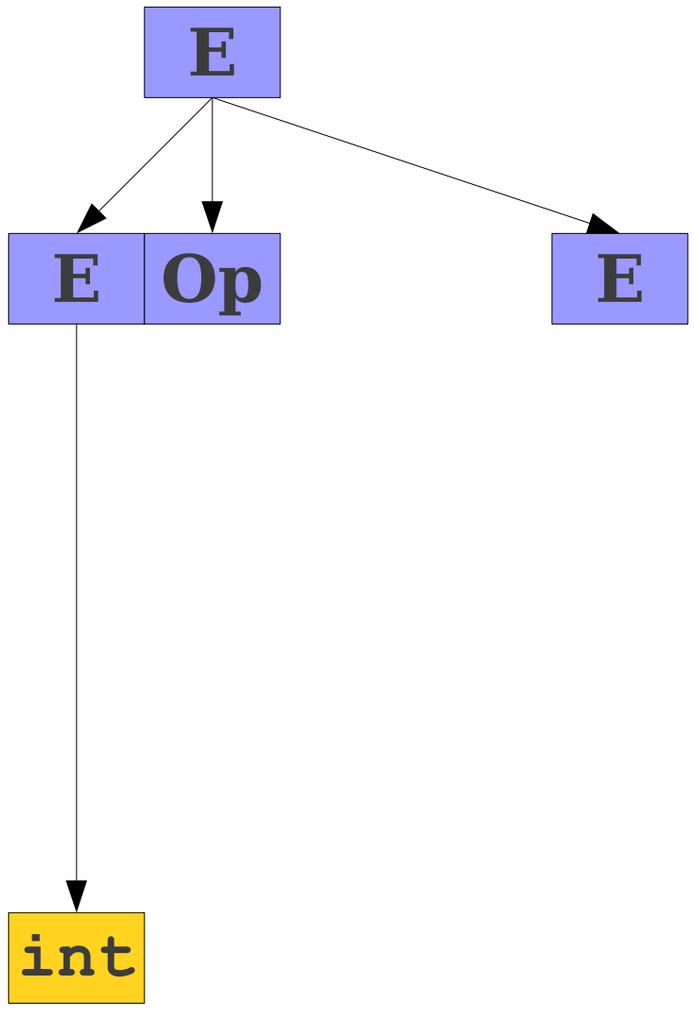
Parse Trees

E

⇒ **E Op E**

⇒ `int` **Op E**

⇒ `int *` **E**



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

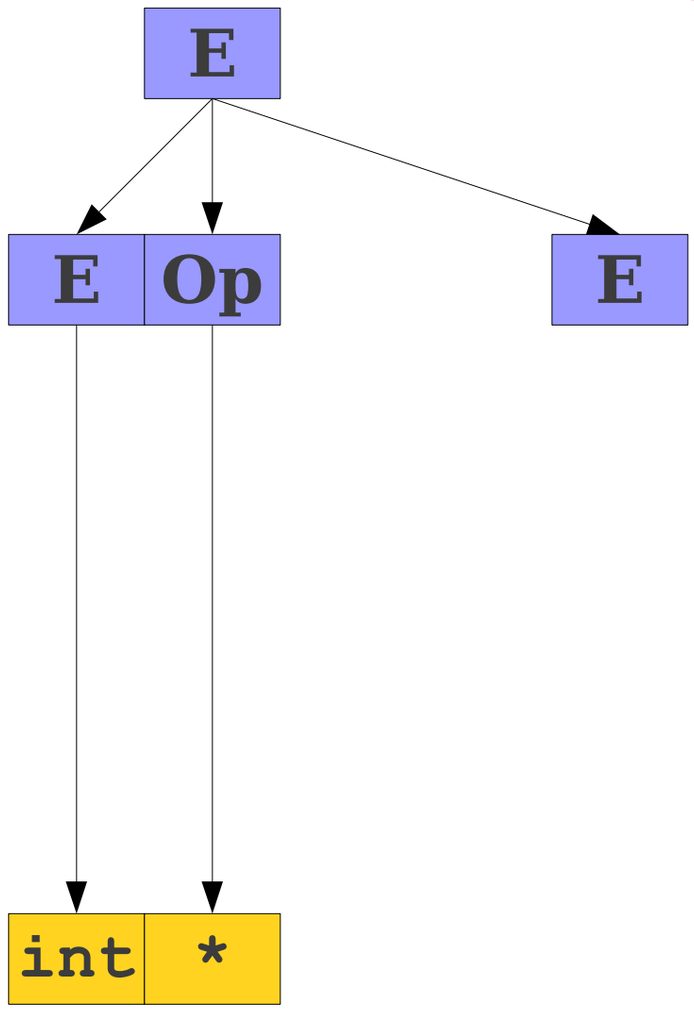
Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ `int` **Op E**

⇒ `int` `*` **E**



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

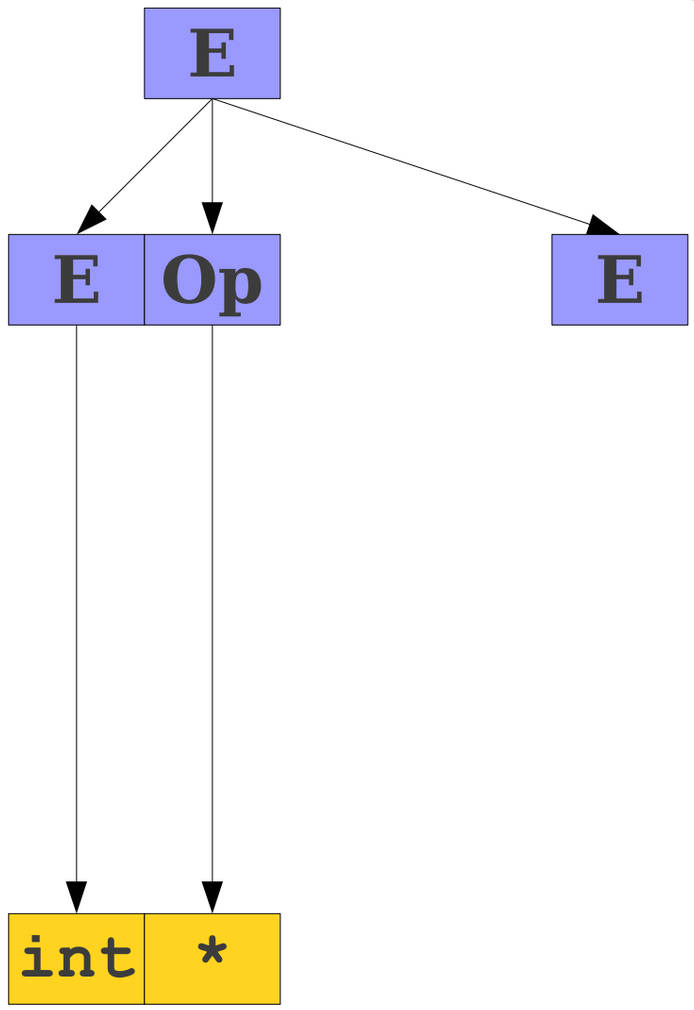
E

⇒ **E Op E**

⇒ `int` **Op E**

⇒ `int *` **E**

⇒ `int * (`**E**`)`



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

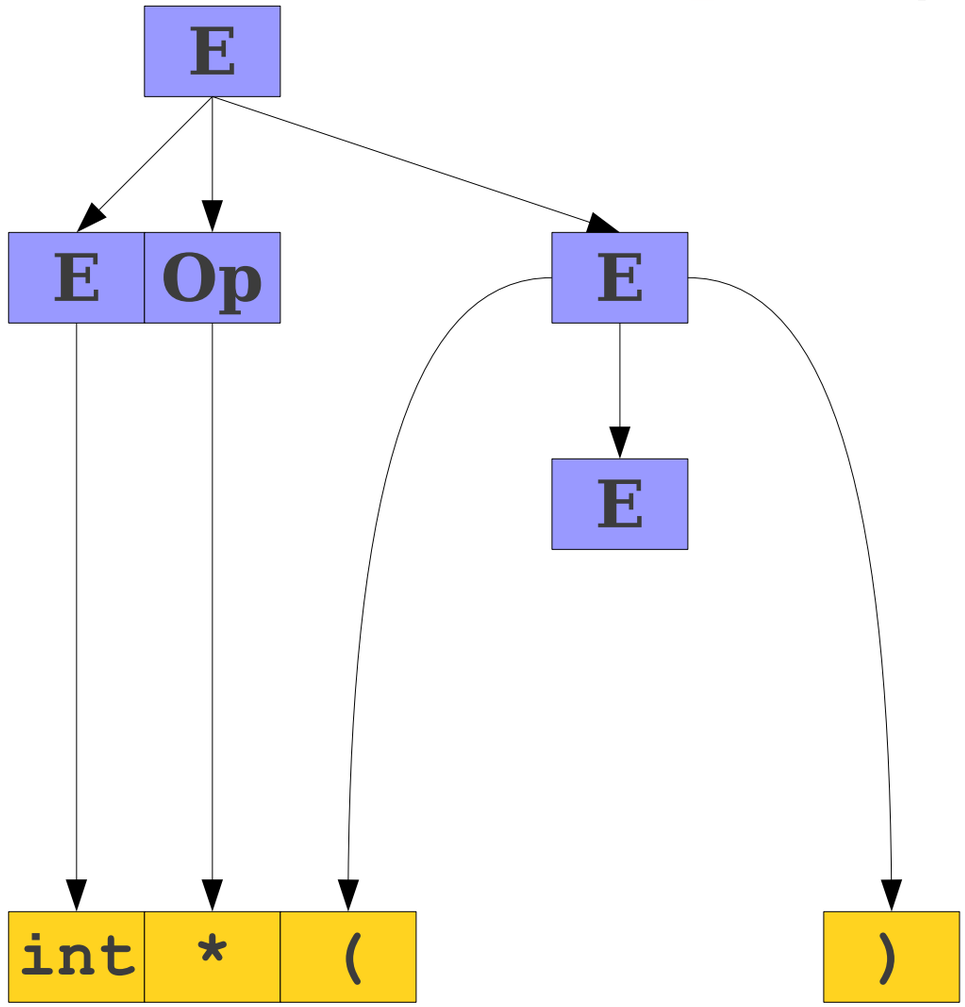
E

⇒ **E Op E**

⇒ `int` **Op E**

⇒ `int *` **E**

⇒ `int * (`**E**`)`

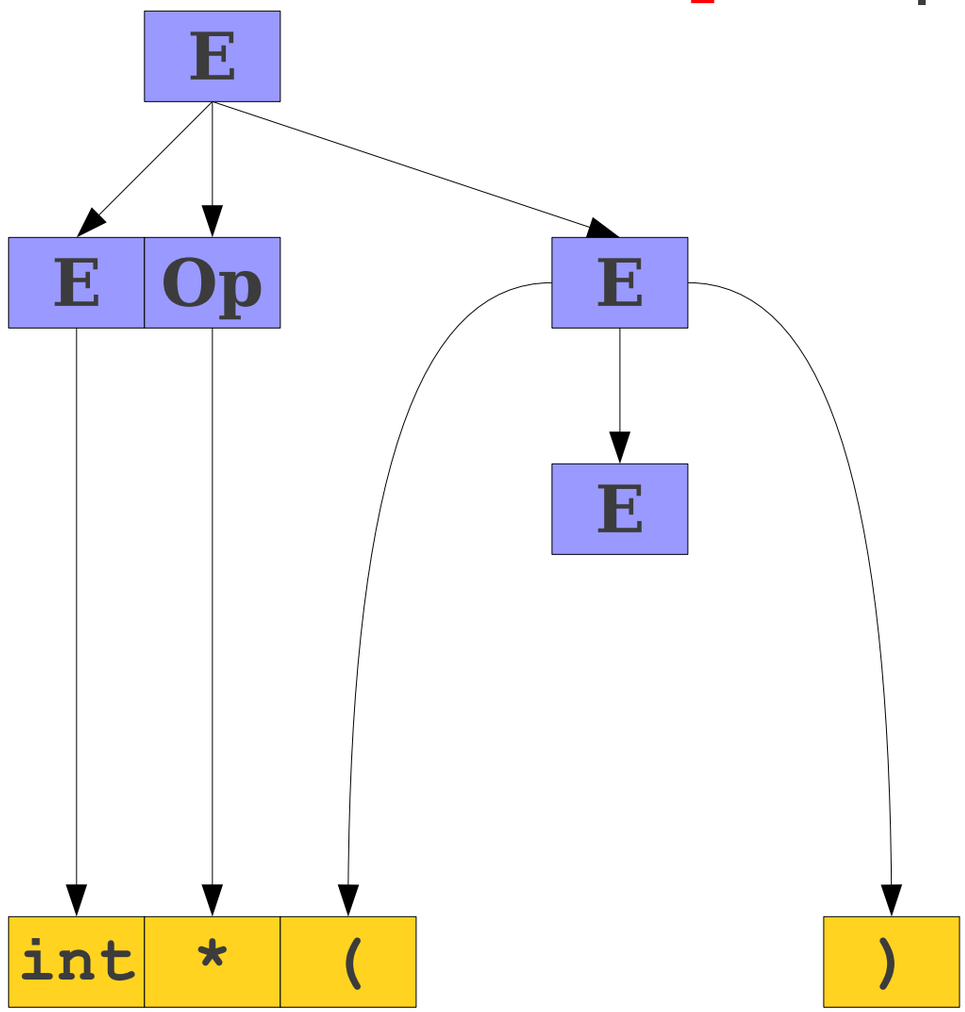


Parse Trees

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

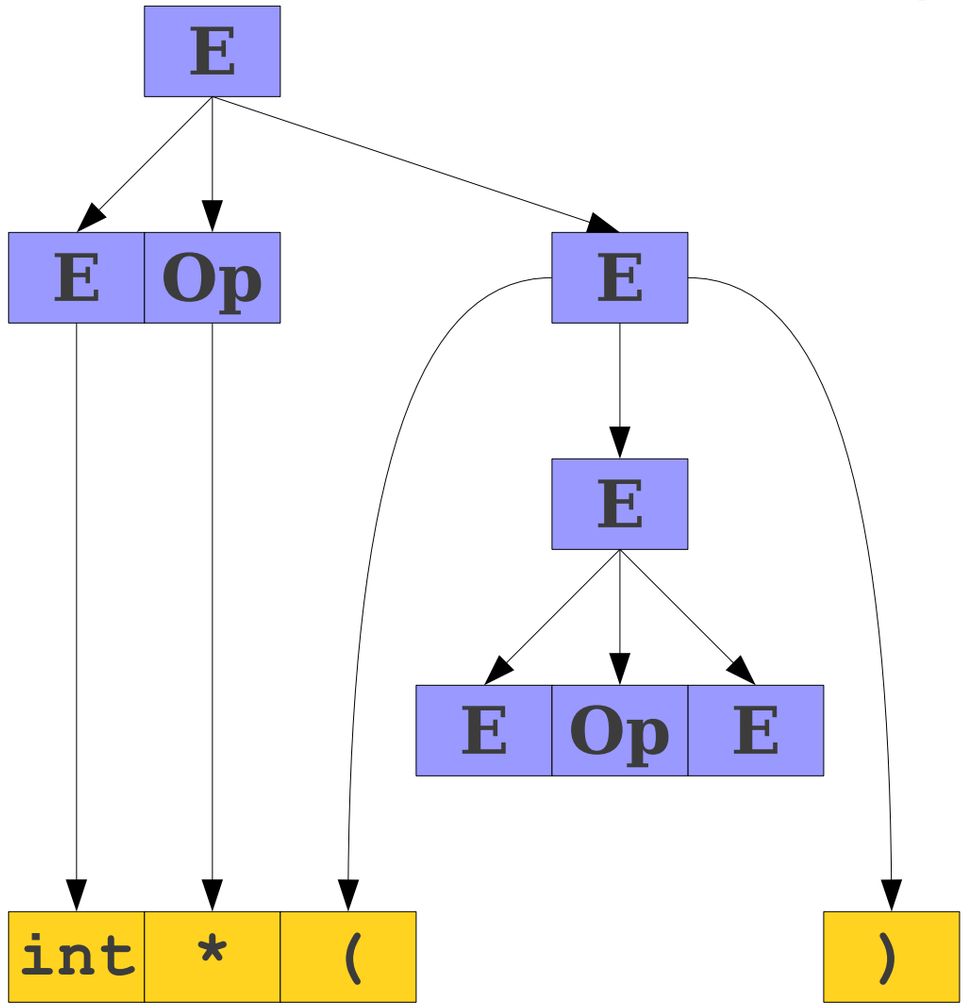
⇒ **E Op E**

⇒ `int` **Op E**

⇒ `int *` **E**

⇒ `int * (`**E**`)`

⇒ `int * (`**E Op E**`)`



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

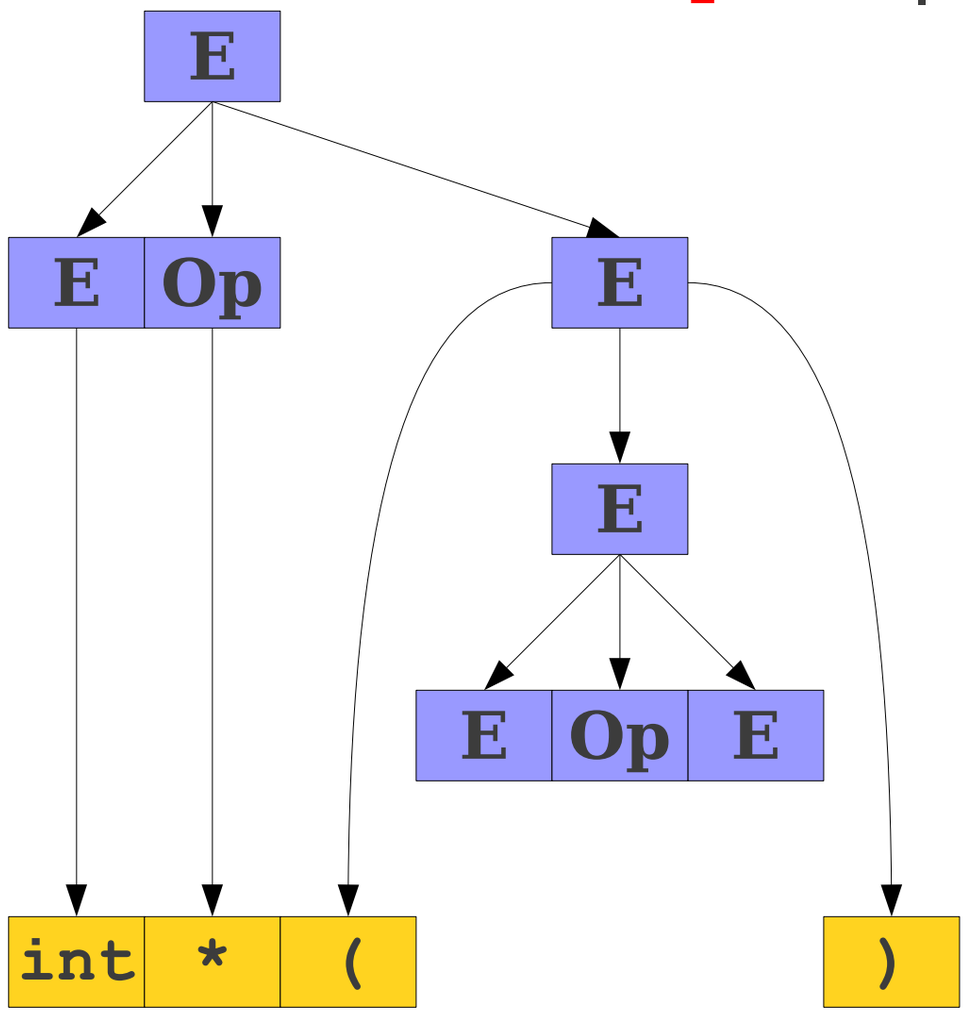
⇒ `int` **Op E**

⇒ `int *` **E**

⇒ `int * (`**E**`)`

⇒ `int * (`**E Op E**`)`

⇒ `int * (int` **Op E**`)`

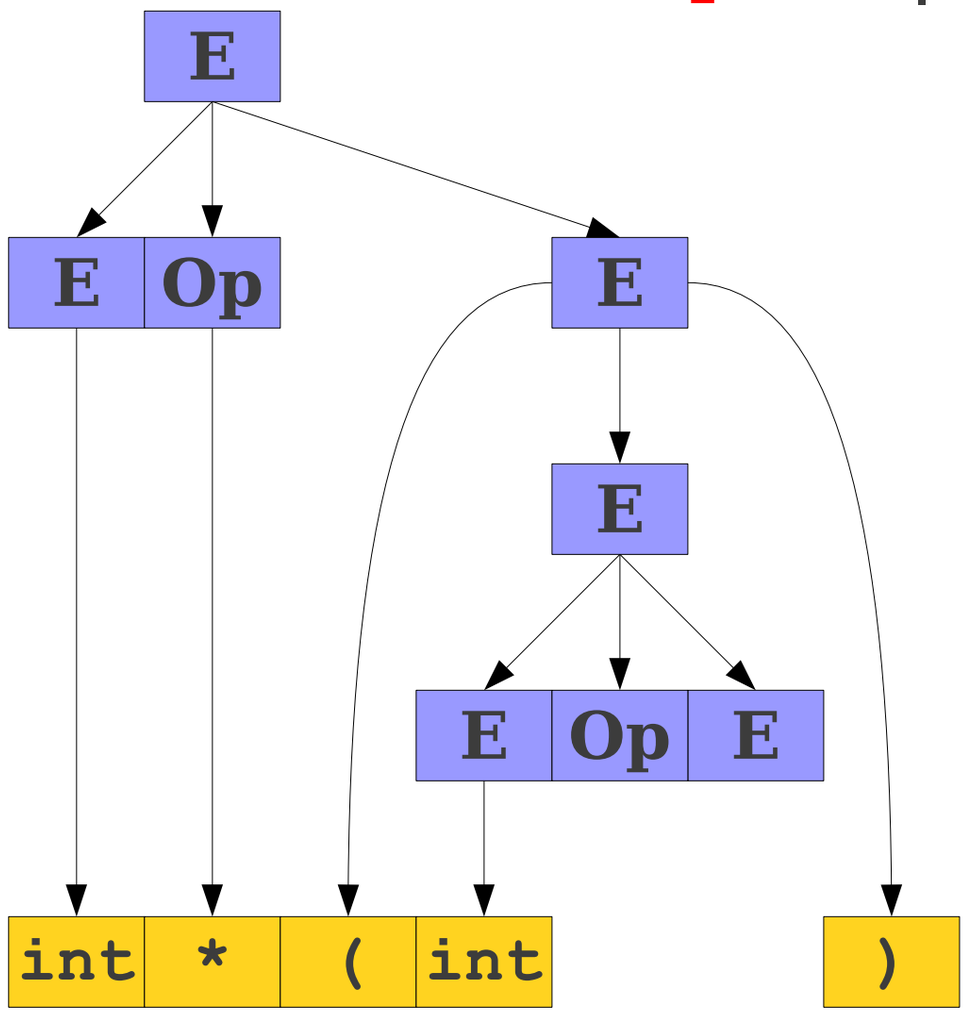


Parse Trees

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**`)`

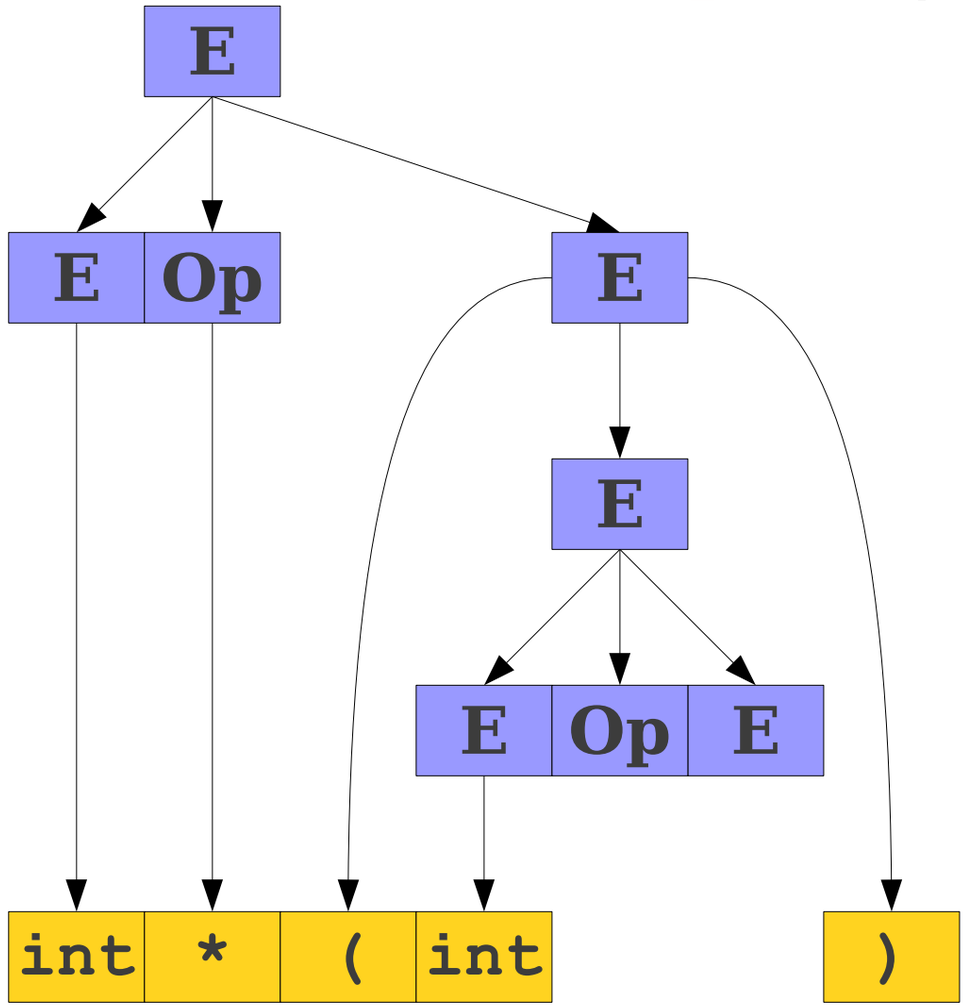


Parse Trees

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**`)`

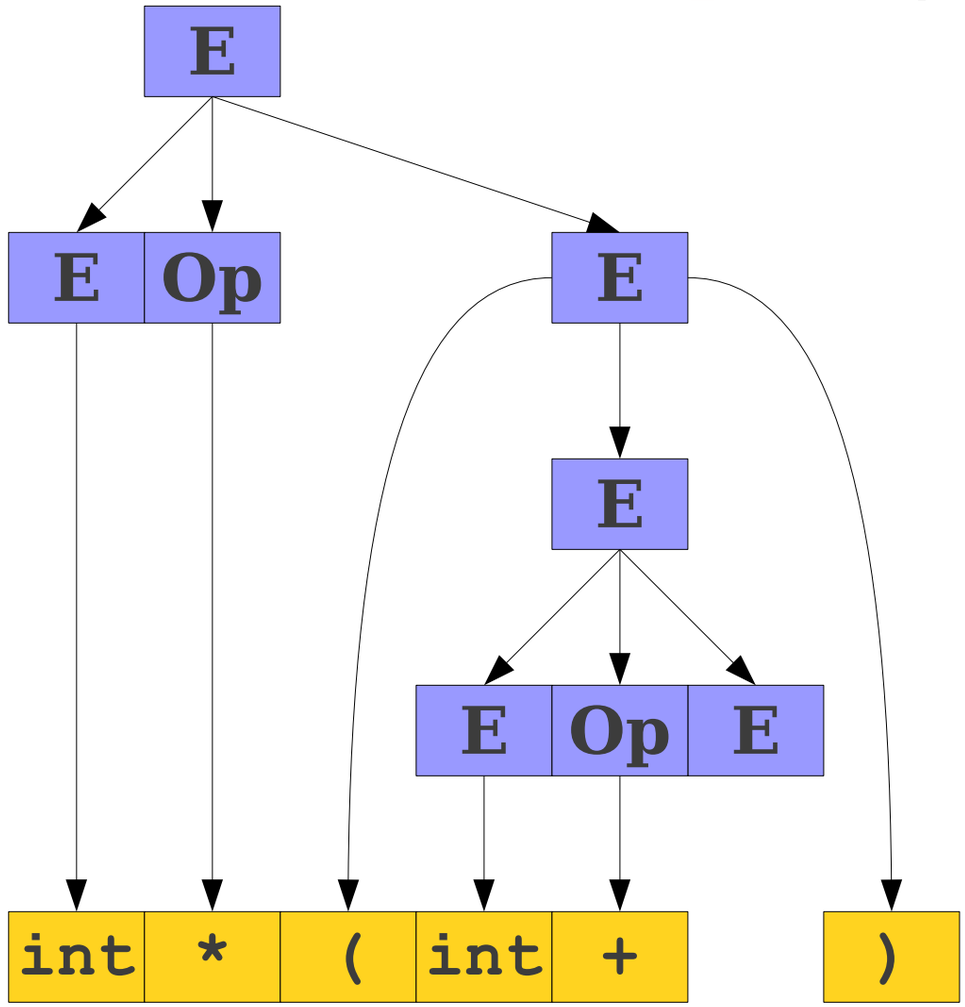


Parse Trees

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**`)`

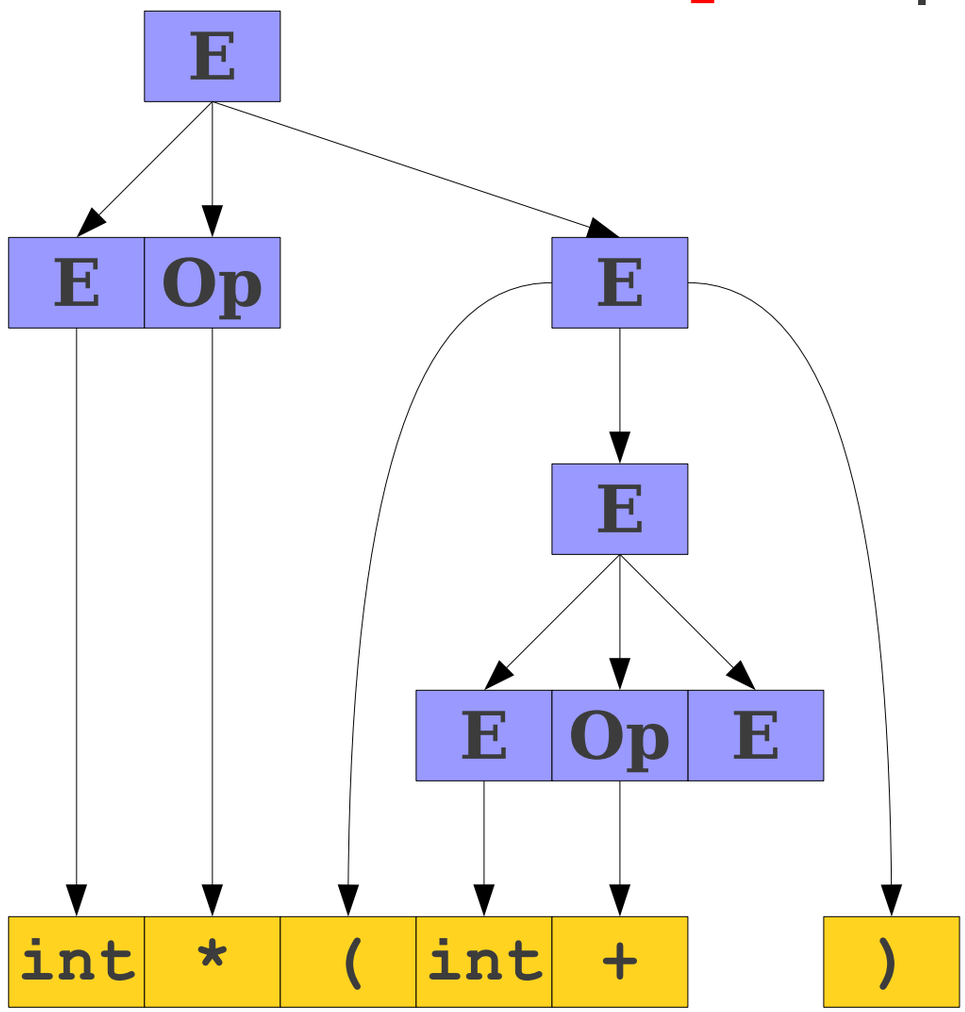


Parse Trees

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)`

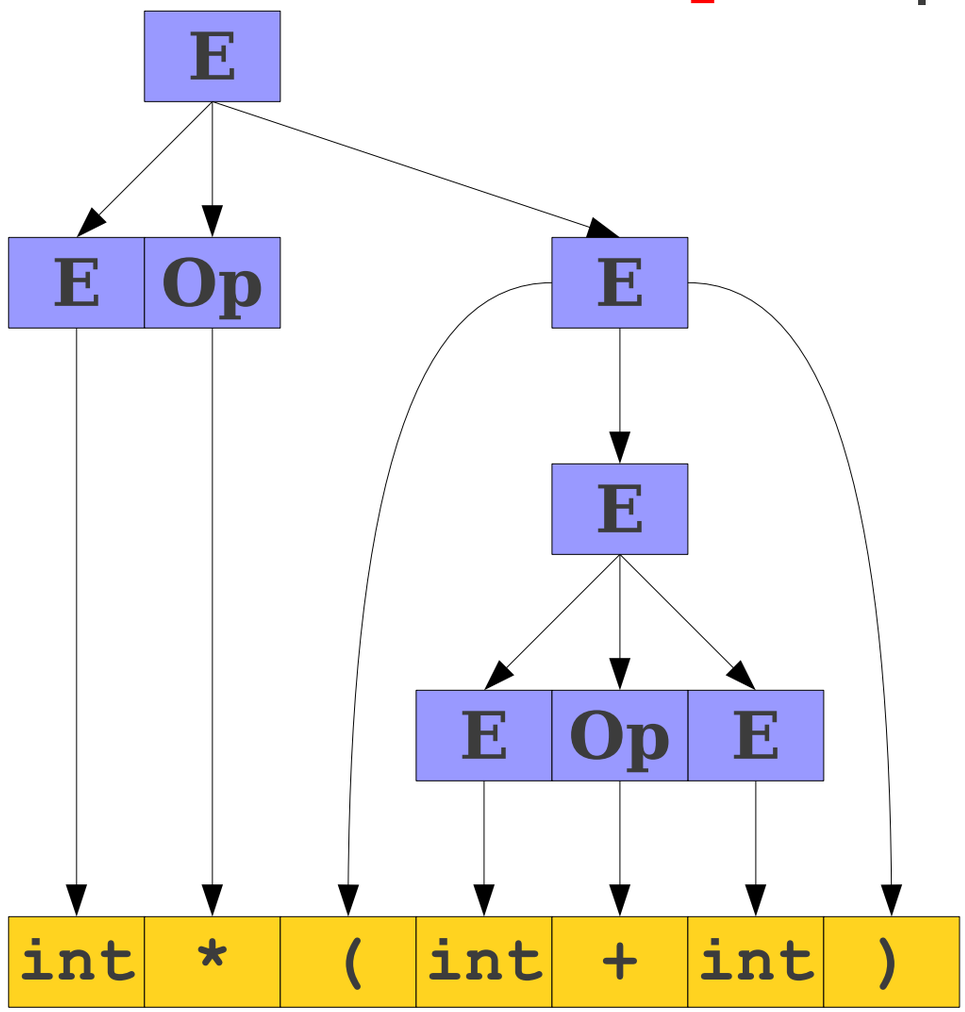


Parse Trees

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 Op E**

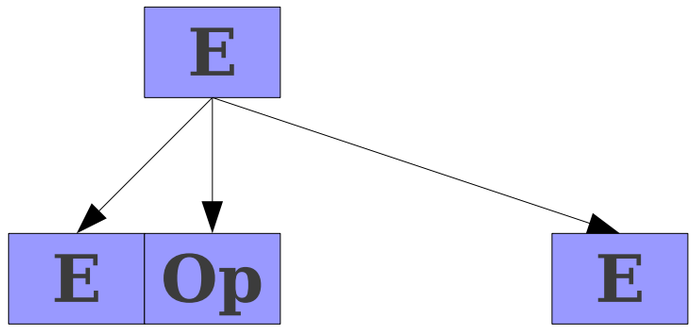
Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**



Parse Trees

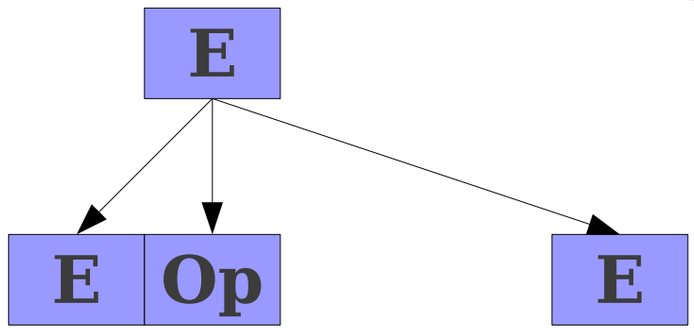
E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`



Parse Trees

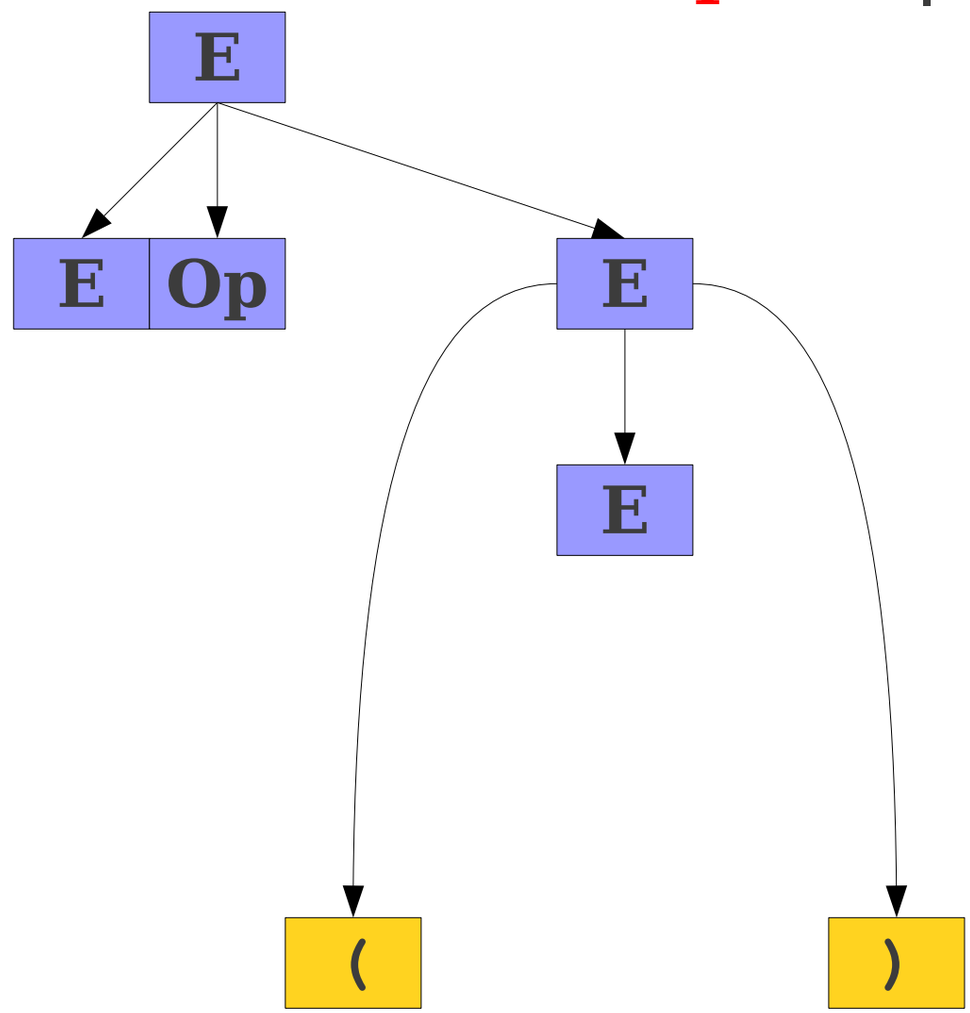
E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

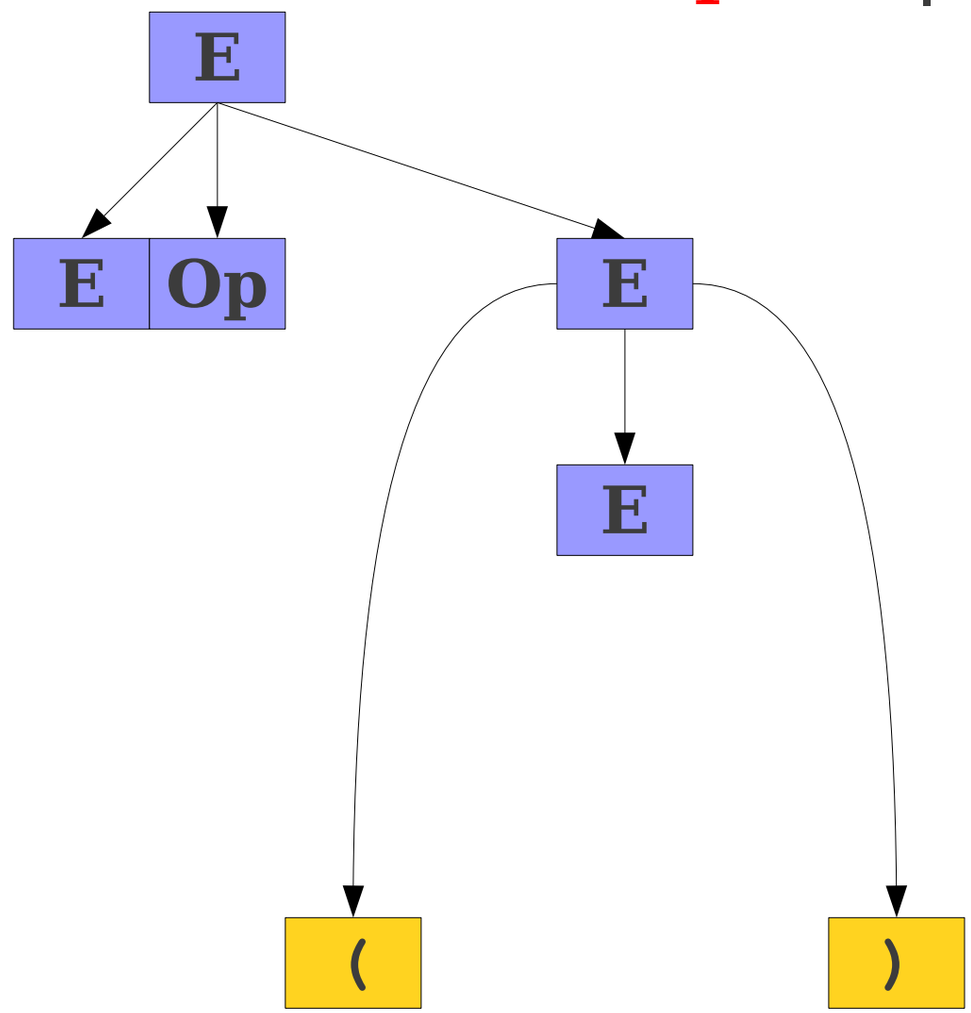
Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

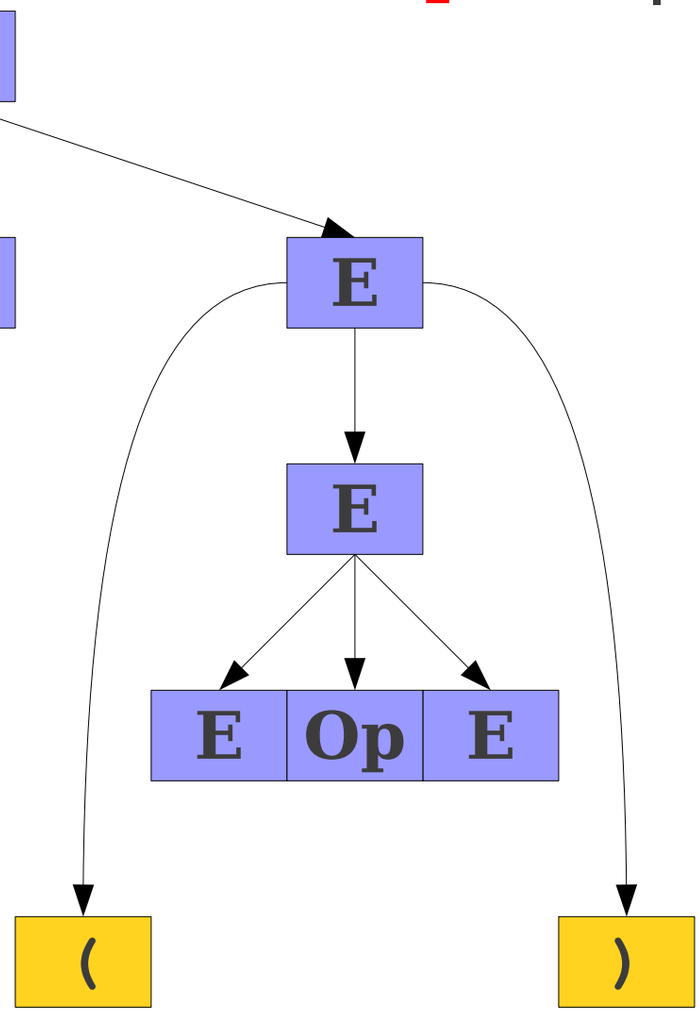
Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

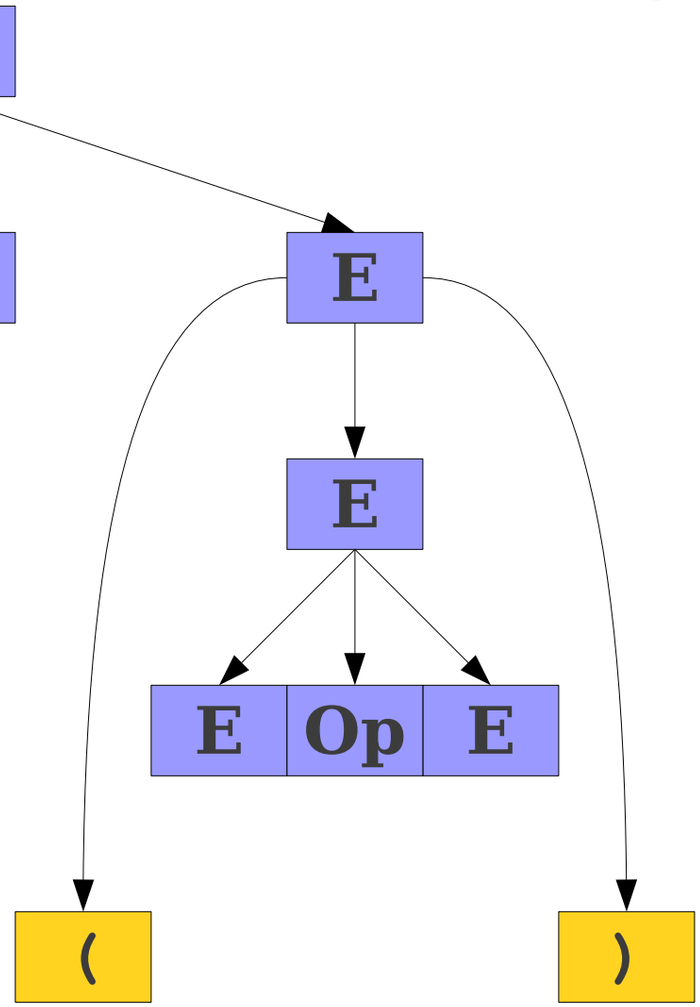
E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

⇒ **E Op** `(`**E Op** `int`**E**`)`



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

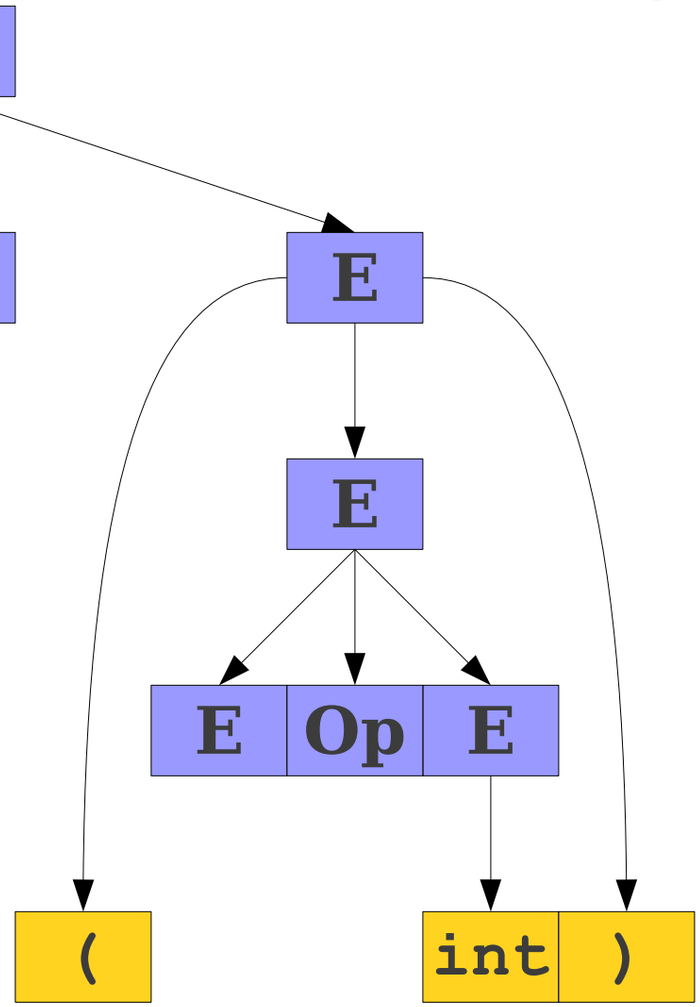
E

⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

⇒ **E Op** `(`**E Op** `int`**E**`)`



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

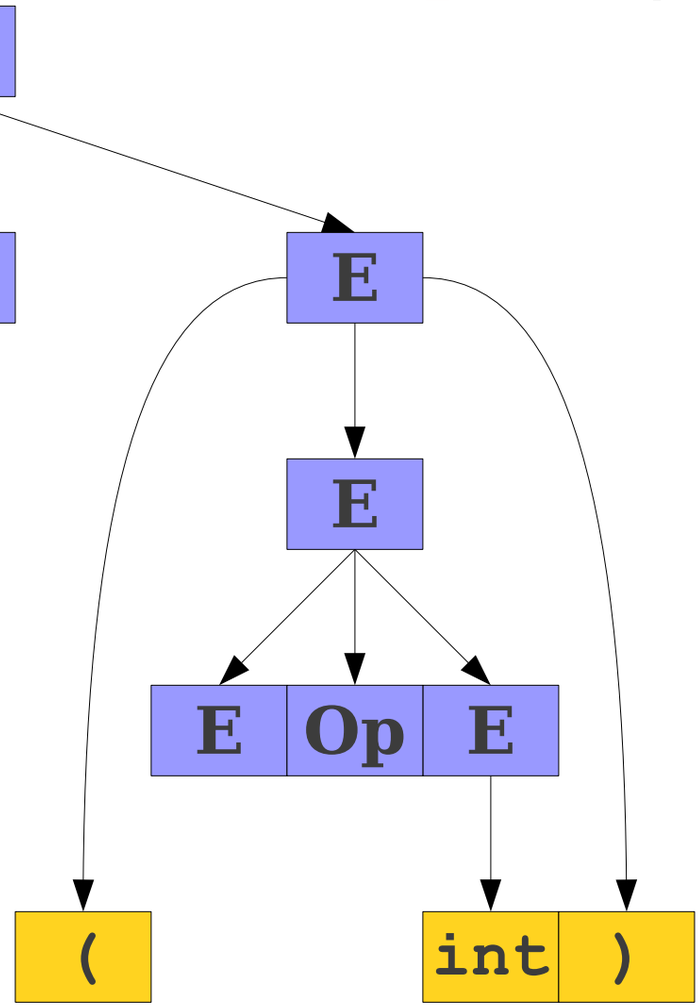
⇒ **E Op E**

⇒ **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

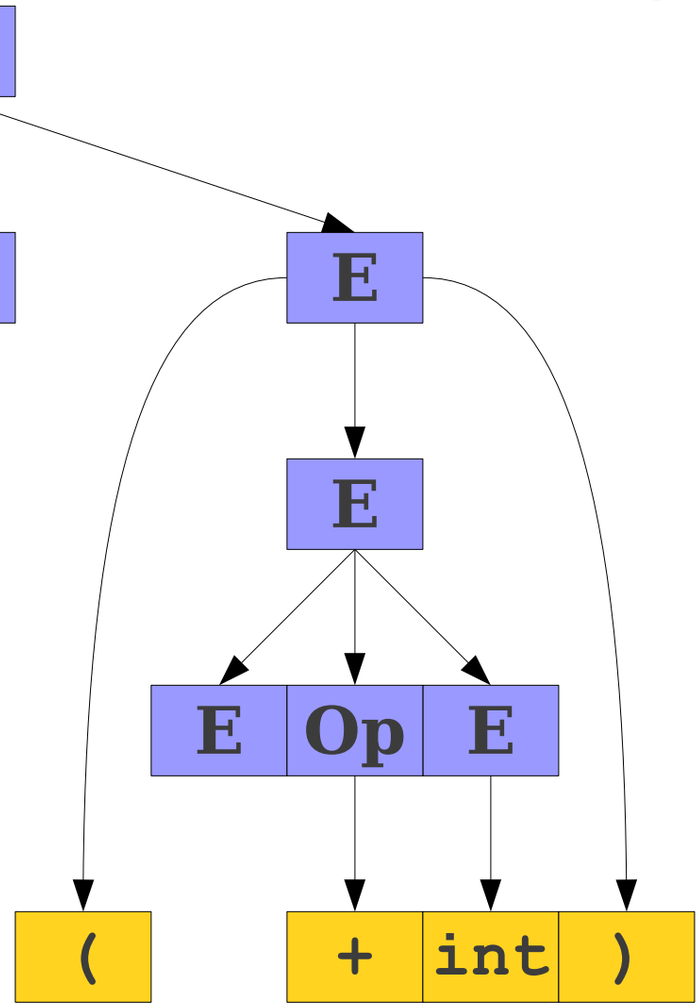
⇒ **E Op E**

⇒ **E Op** `(`**E**`)`

⇒ **E Op** `(`**E Op E**`)`

⇒ **E Op** `(`**E Op** `int`**E**`)`

⇒ **E Op** `(`**E** `+ int`**E**`)`



Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

E

⇒ **E Op E**

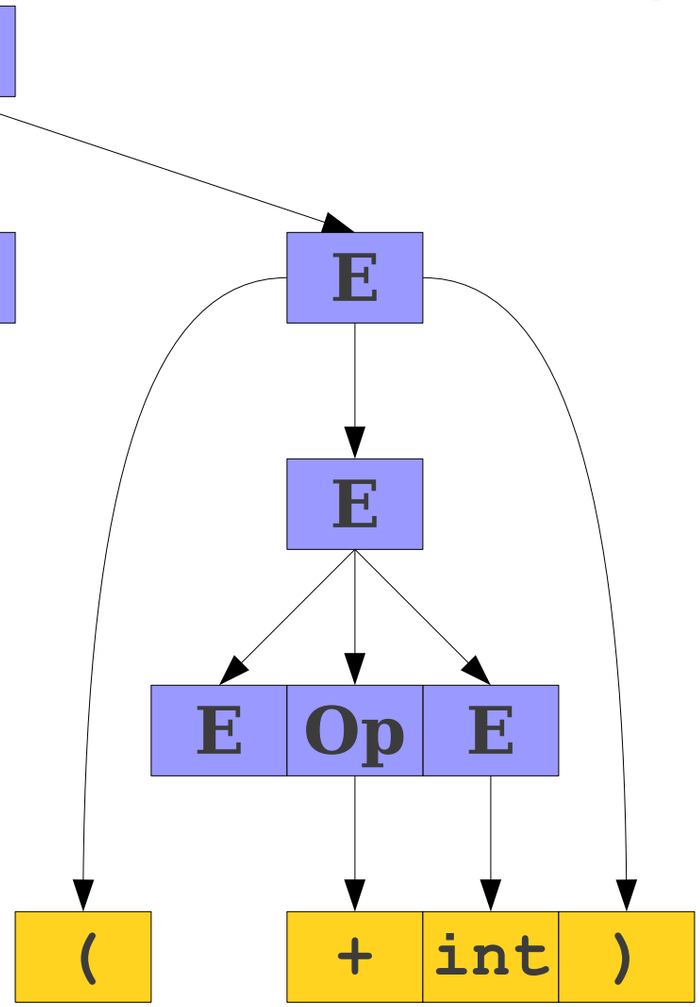
⇒ **E Op** `(`**E**`)`

⇒ **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**

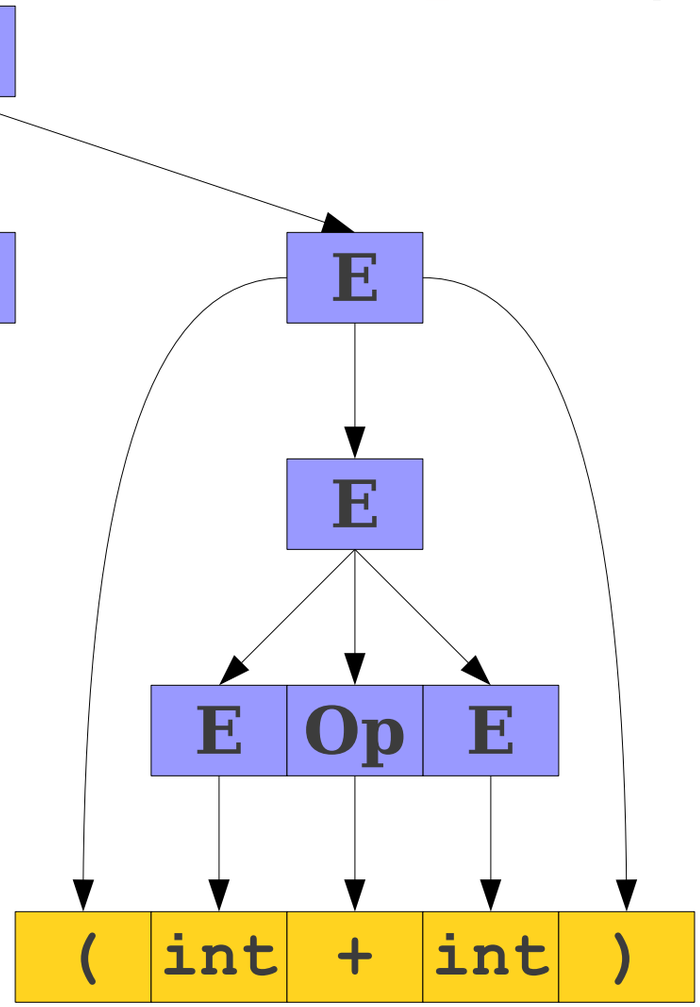
⇒ **E Op** `(`**E**`)`

⇒ **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**`)`

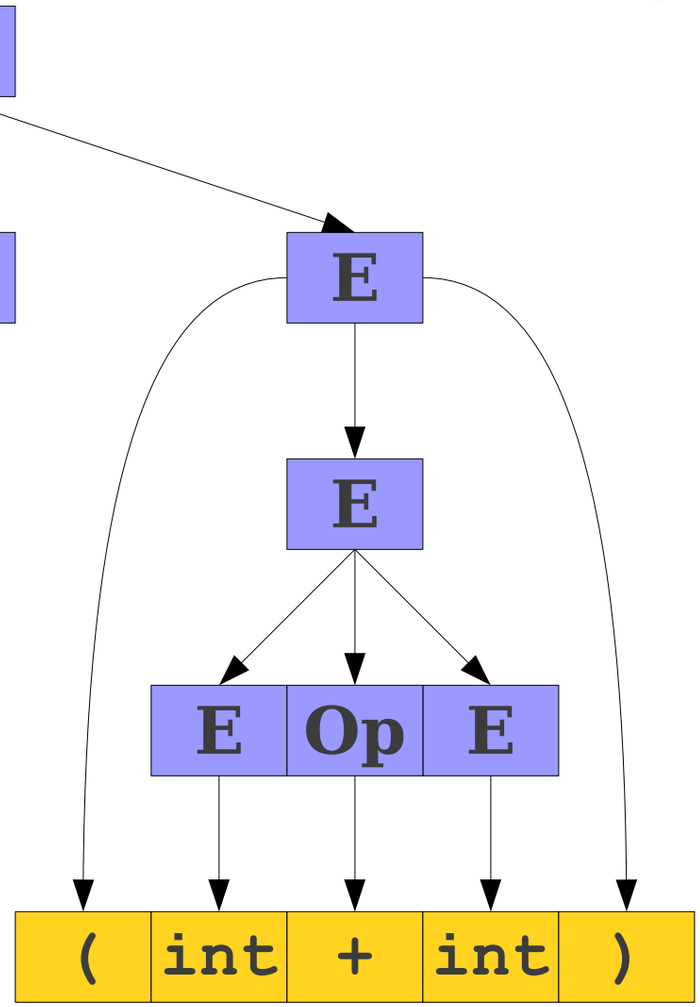
⇒ **E Op** `(`**E Op E**`)`

⇒ **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**`)`

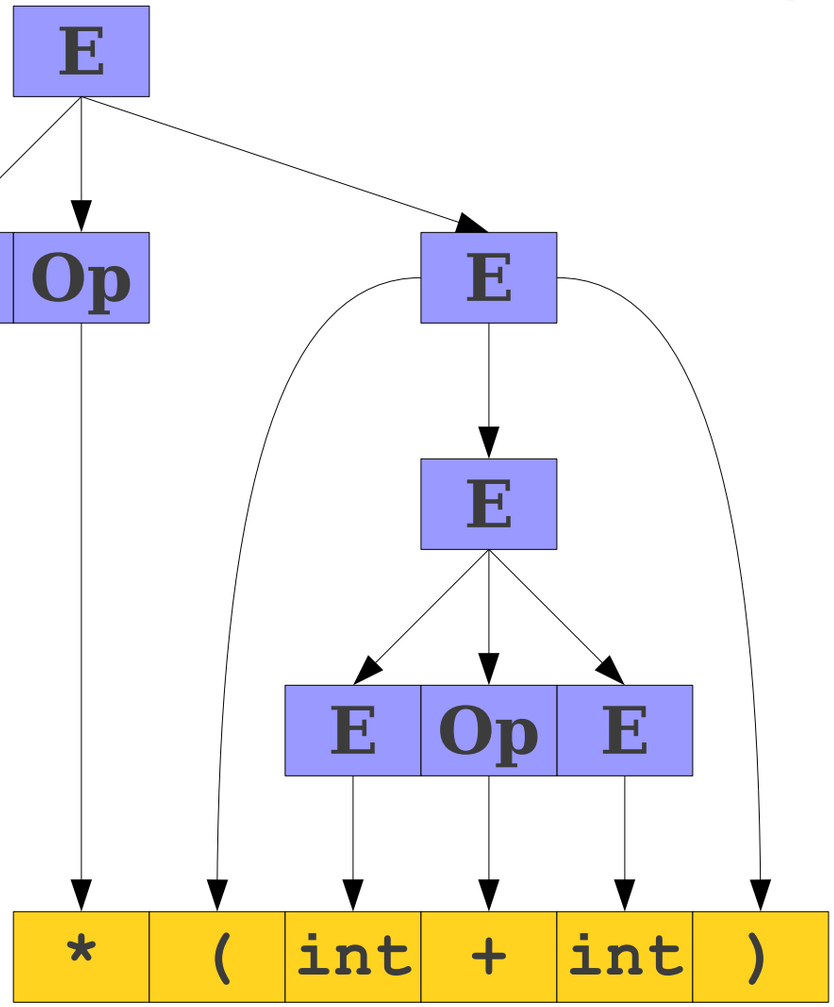
⇒ **E Op** `(`**E Op E**`)`

⇒ **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**`)`

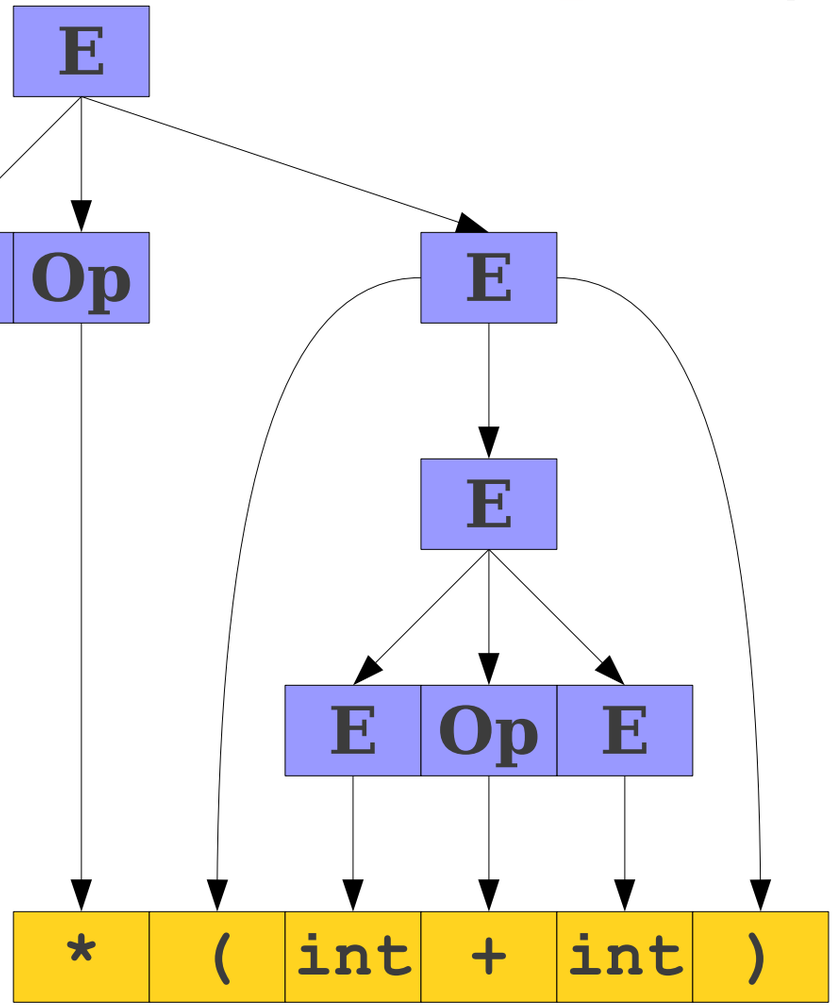
⇒ **E Op** `(`**E Op** `int`**)**

⇒ **E Op** `(`**E** `+ int`**)**

⇒ **E Op** `(int + int)`

⇒ **E** `* (int + int)`

⇒ `int * (int + int)`

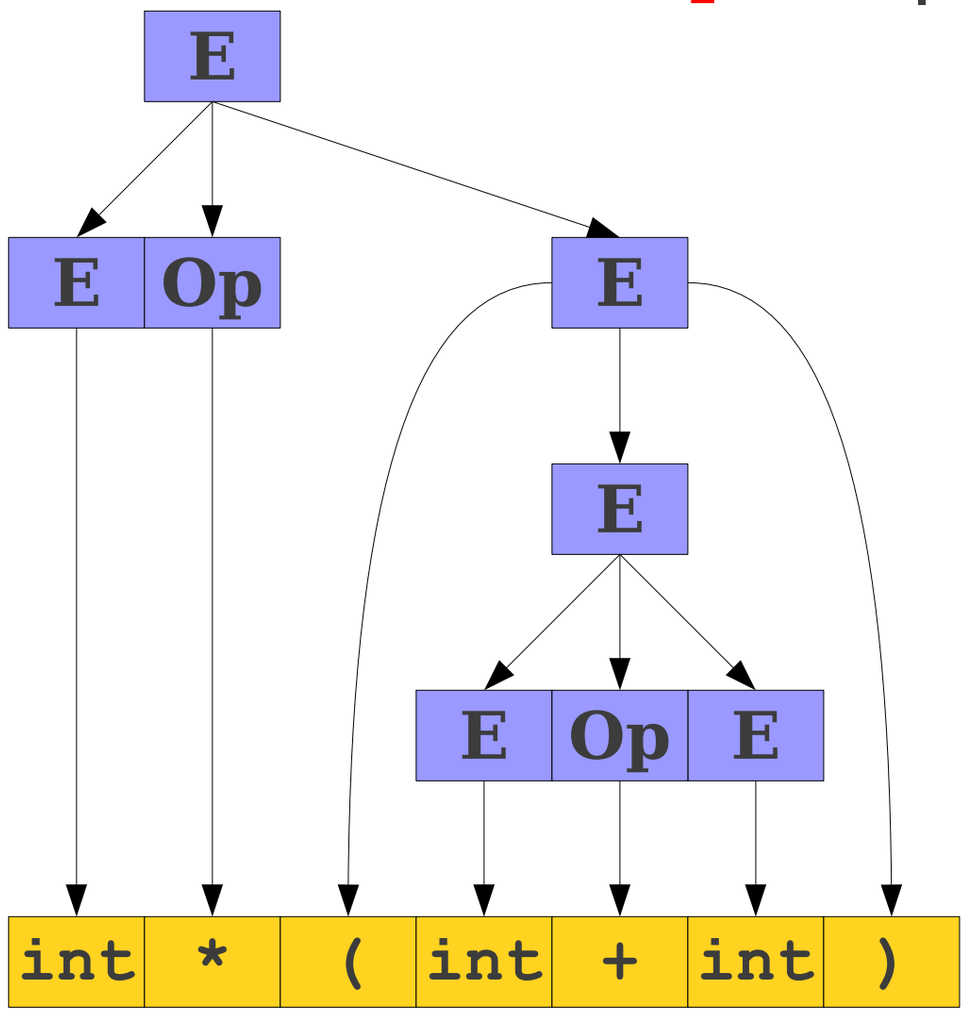


Parse Trees

E → `int` | **E Op E** | `(`**E**`)`

Op → `+` | `-` | `*` | `/`

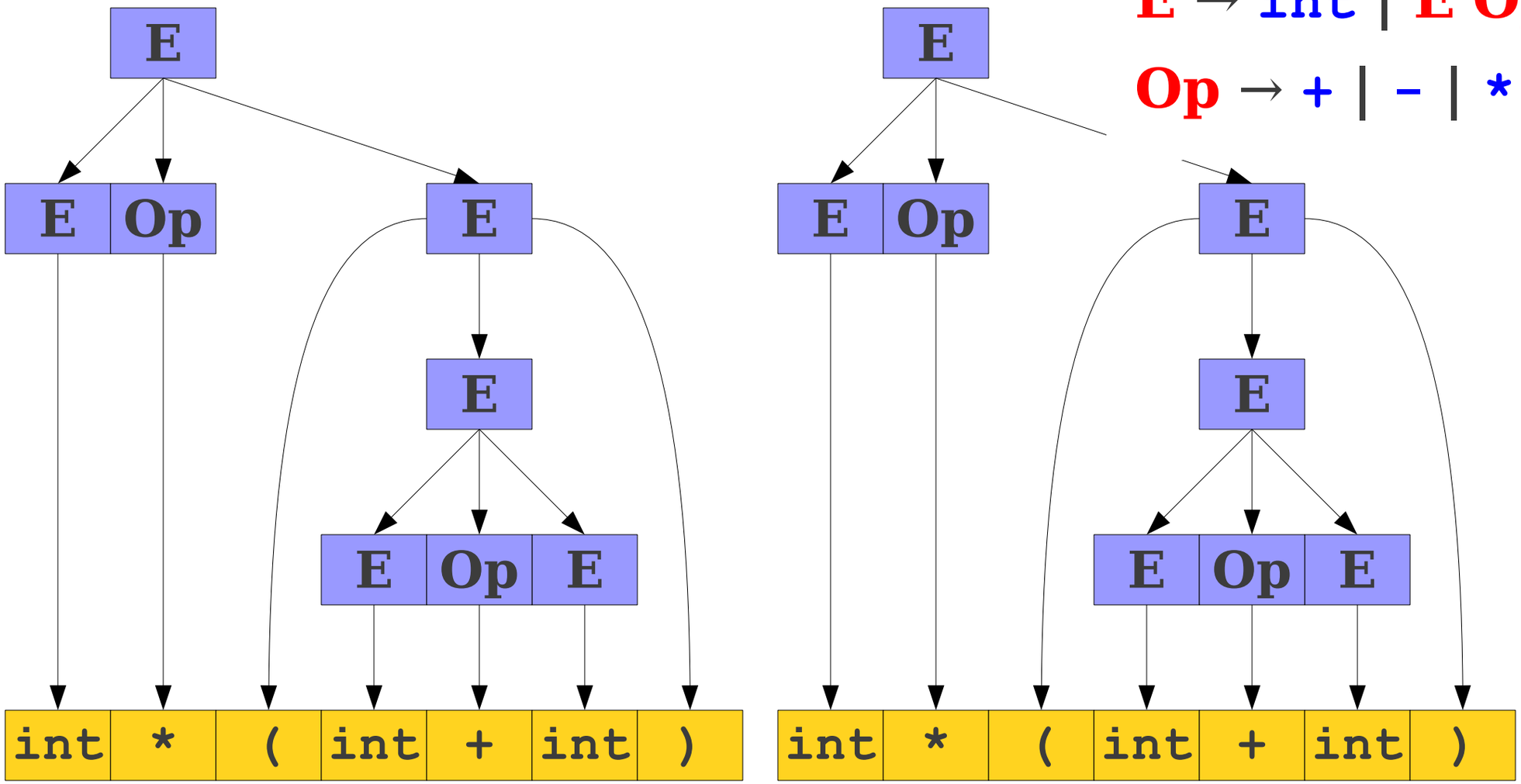
E
⇒ **E Op E**
⇒ **E Op** `(`**E**`)`
⇒ **E Op** `(`**E Op E**`)`
⇒ **E Op** `(`**E Op** `int``)`
⇒ **E Op** `(`**E** `+ int)`
⇒ **E Op** `(int + int)`
⇒ **E** `* (int + int)`
⇒ `int * (int + int)`



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

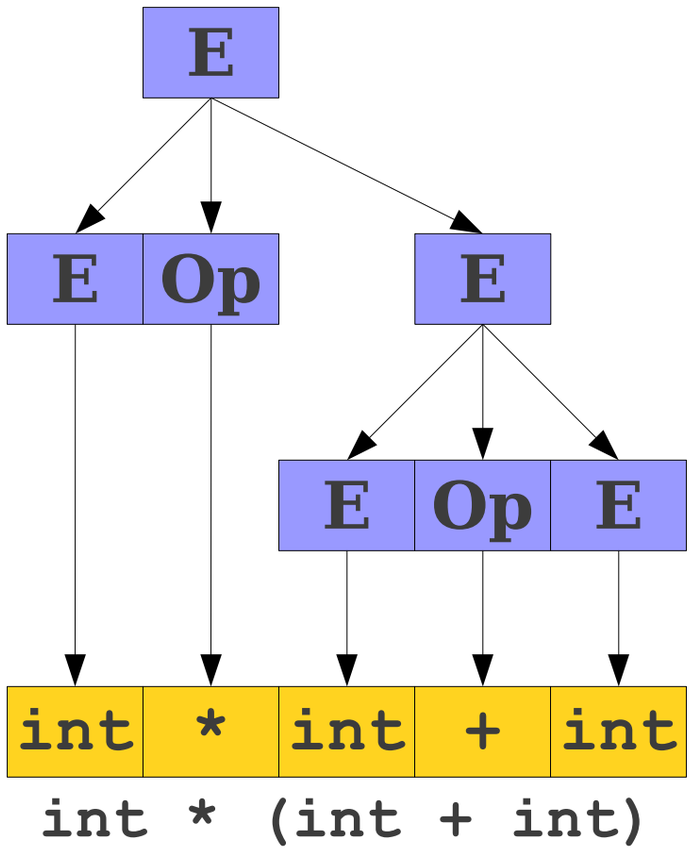
- A **parse tree** is a tree encoding the steps in a derivation.
- Internal nodes represent nonterminal symbols used in the production.
- Inorder walk of the leaves contains the generated string.
- Encodes what productions are used, not the order in which those productions are applied.

The Goal of Parsing

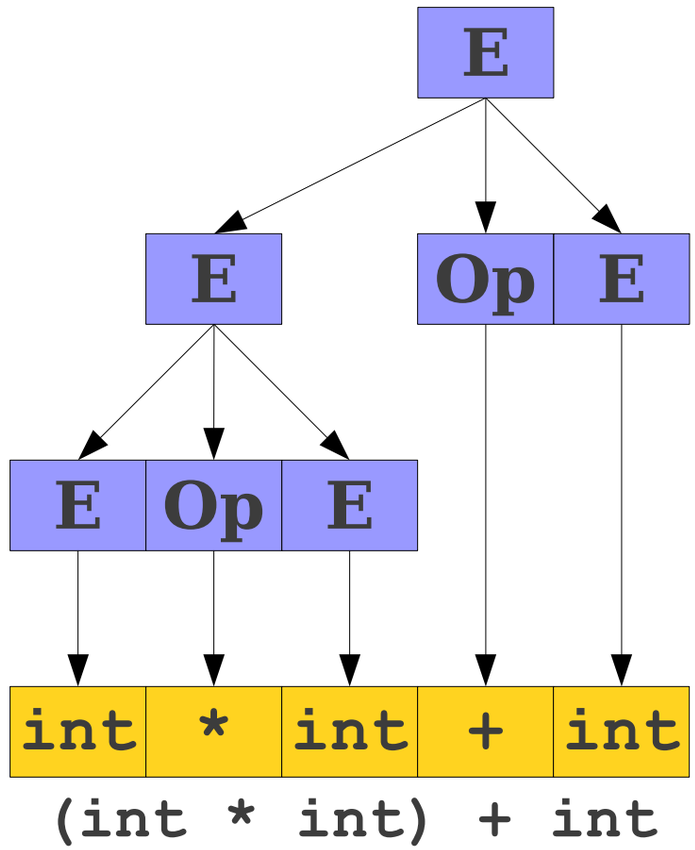
- Goal of syntax analysis: Recover the **structure** described by a series of tokens.
- If language is described as a CFG, goal is to recover a parse tree for the the input string.
  - Usually we do some simplifications on the tree; more on that later.

Challenges in Parsing

A Serious Problem




```
int * (int + int)
```




```
(int * int) + int
```

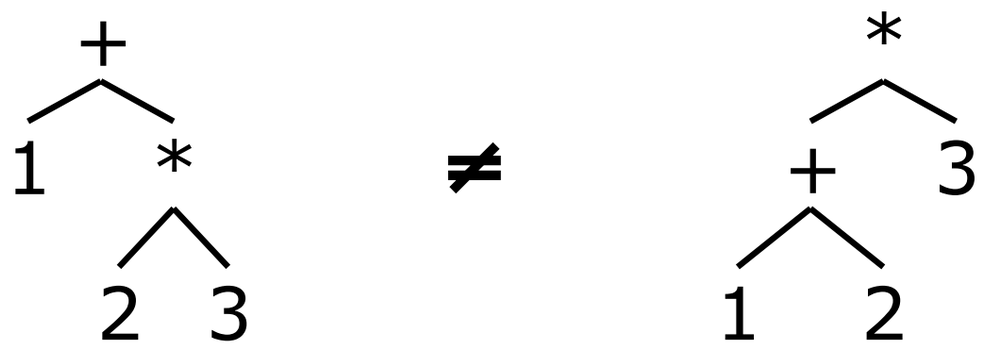
Ambiguity

- 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*.

Different Parse Trees

$S \rightarrow S + S \mid S * S \mid$ number

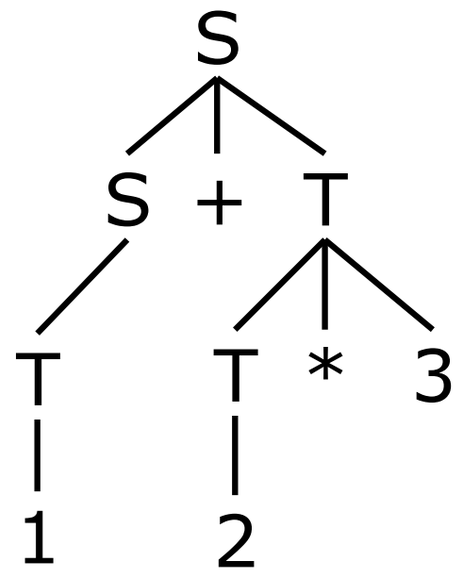
- Consider expression 1 + 2 * 3
- Derivation 1: $S \rightarrow S + S \rightarrow 1 + S \rightarrow 1 + S * S \rightarrow 1 + 2 * S \rightarrow 1 + 2 * 3$
- Derivation 2: $S \rightarrow S * S \rightarrow S * 3 \rightarrow S + S * 3 \rightarrow S + 2 * 3 \rightarrow 1 + 2 * 3$



Eliminating Ambiguity

- • Often can eliminate ambiguity by adding non-terminals & allowing recursion only on right or left

- $S \rightarrow S + T \mid T$
- $T \rightarrow T$ * 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
- $S \rightarrow X = E$

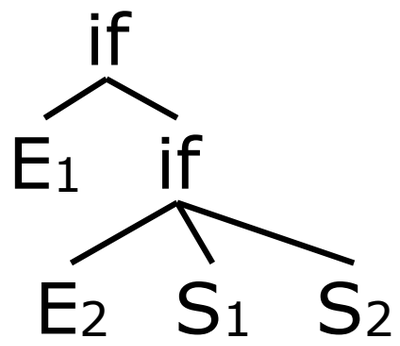
- Is this grammar OK?

No! Ambiguous

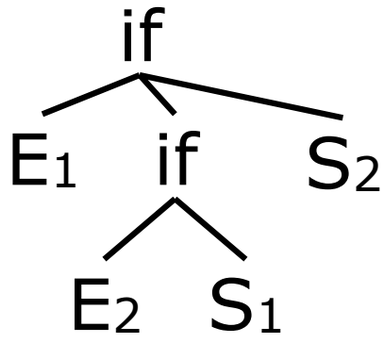
- if (*E*1) if (*E*2) *S*1 else *S*2

$S \rightarrow$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**

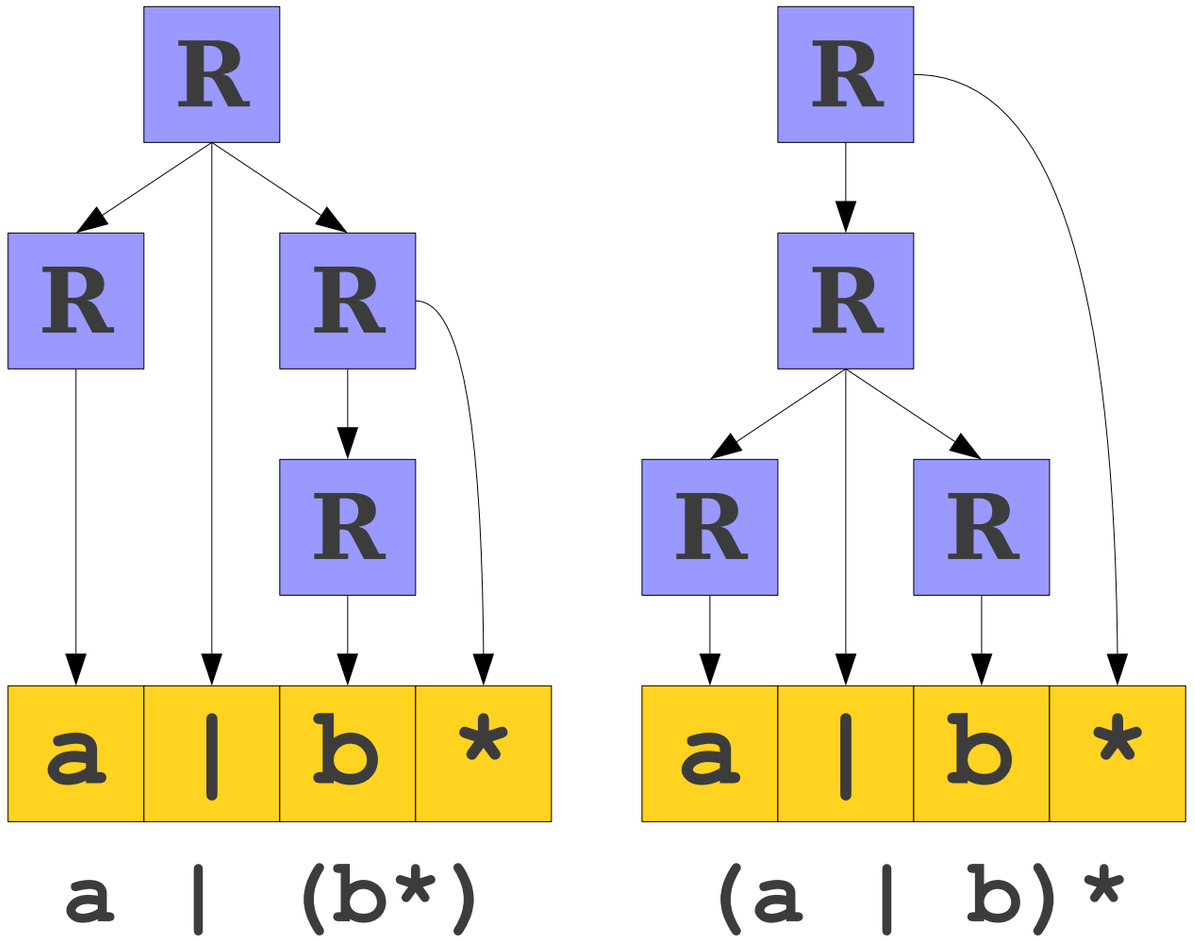
  R → **R** "**|**" **R**

  R → **R***

  R → **(R)**

An Ambiguous Grammar

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



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 | …

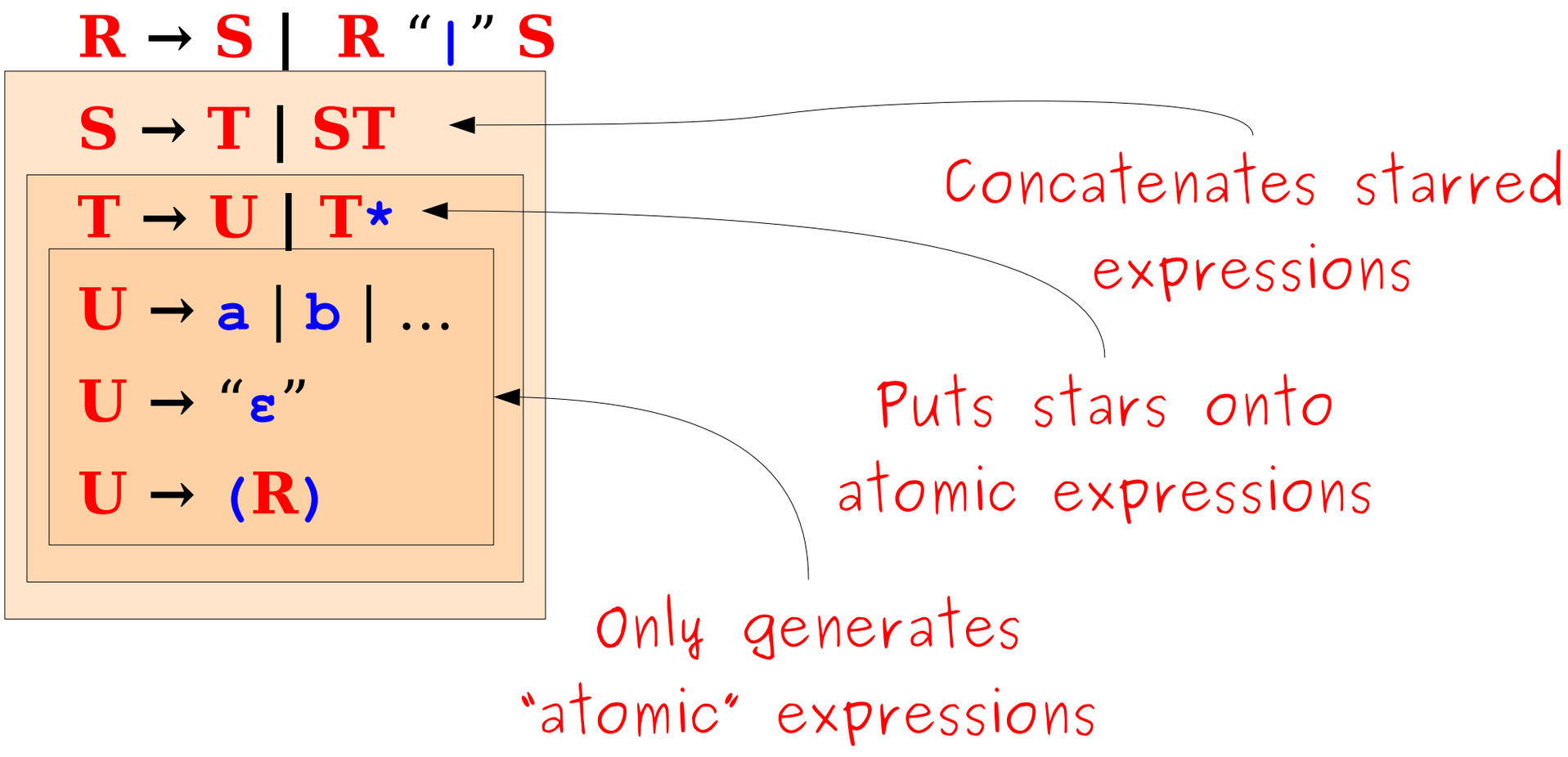
U → “ε”

U → (R)

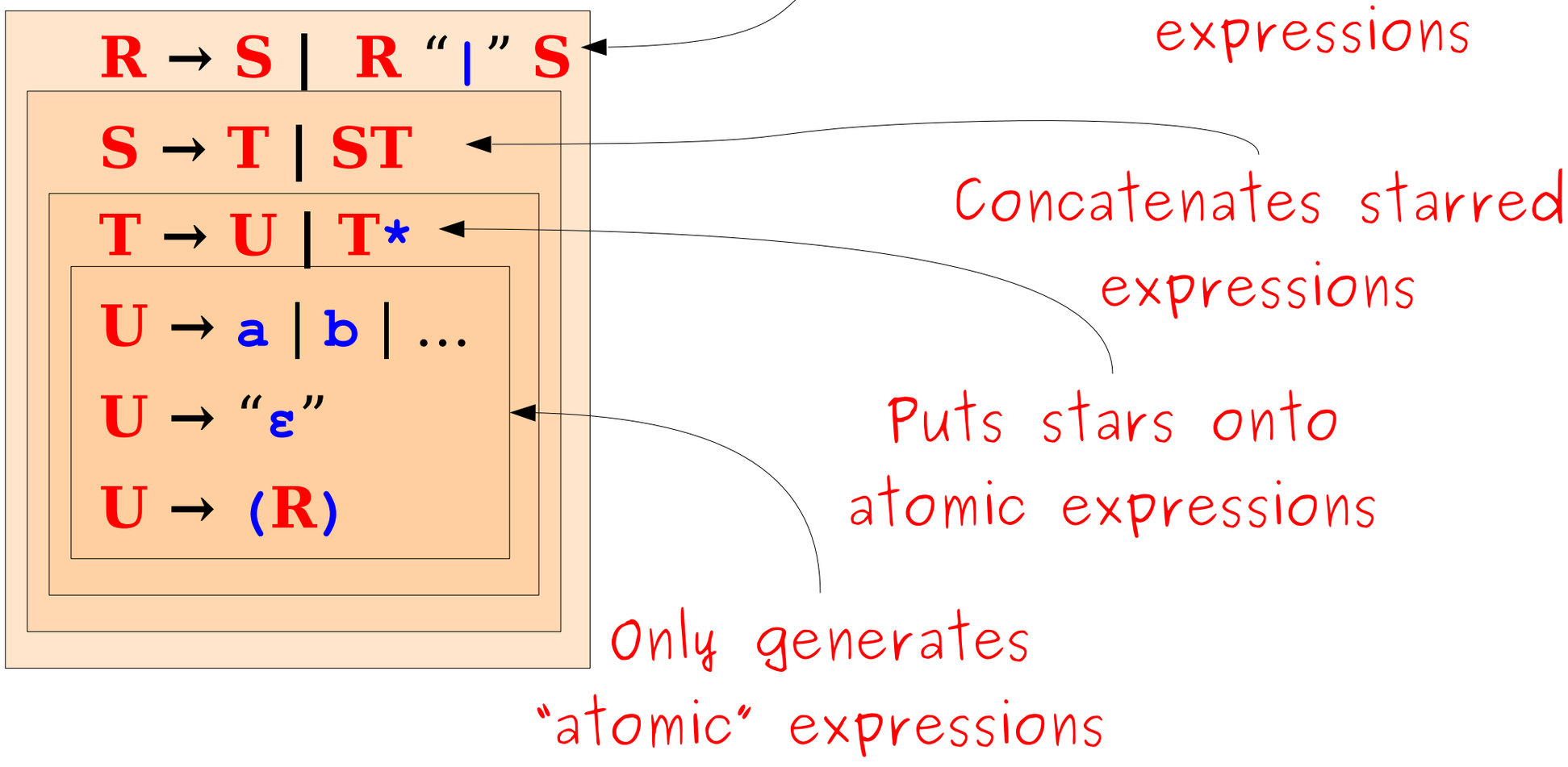
Puts stars onto atomic expressions

Only generates “atomic” expressions

Why is this unambiguous?



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

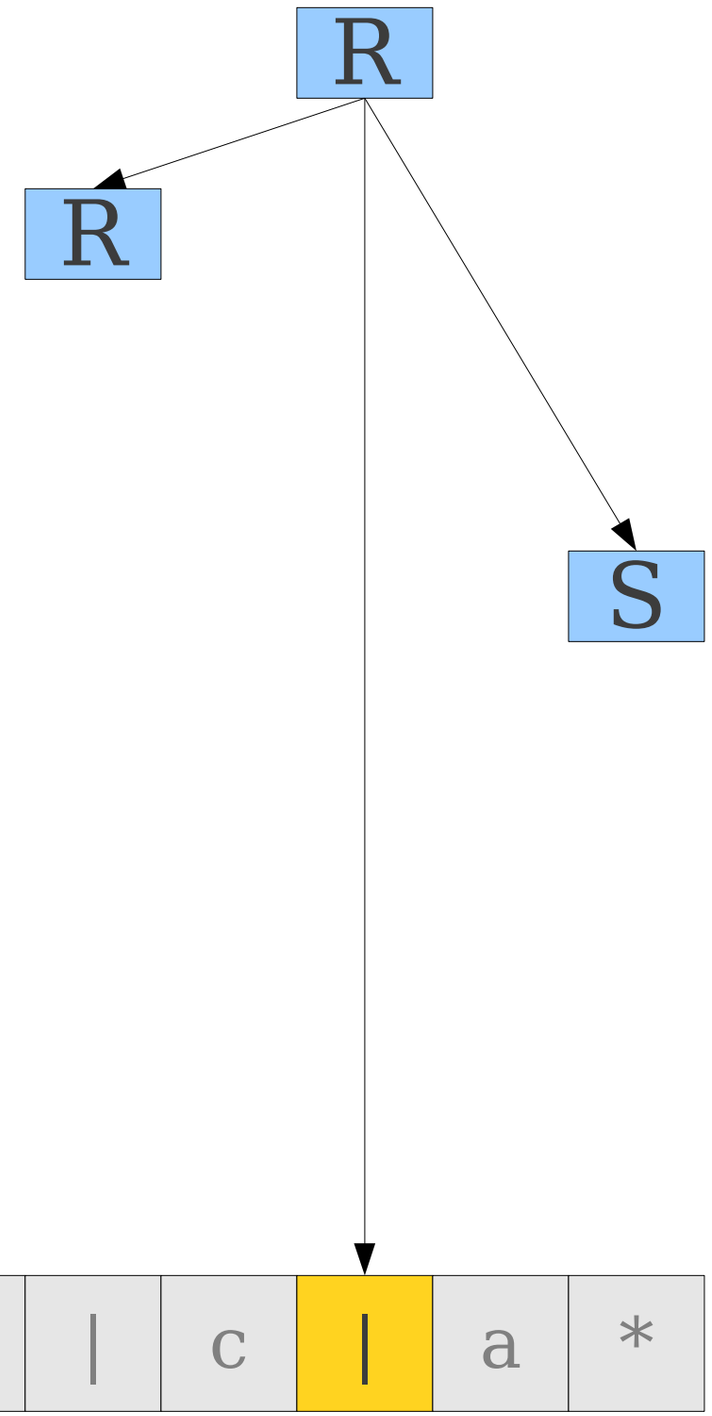
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

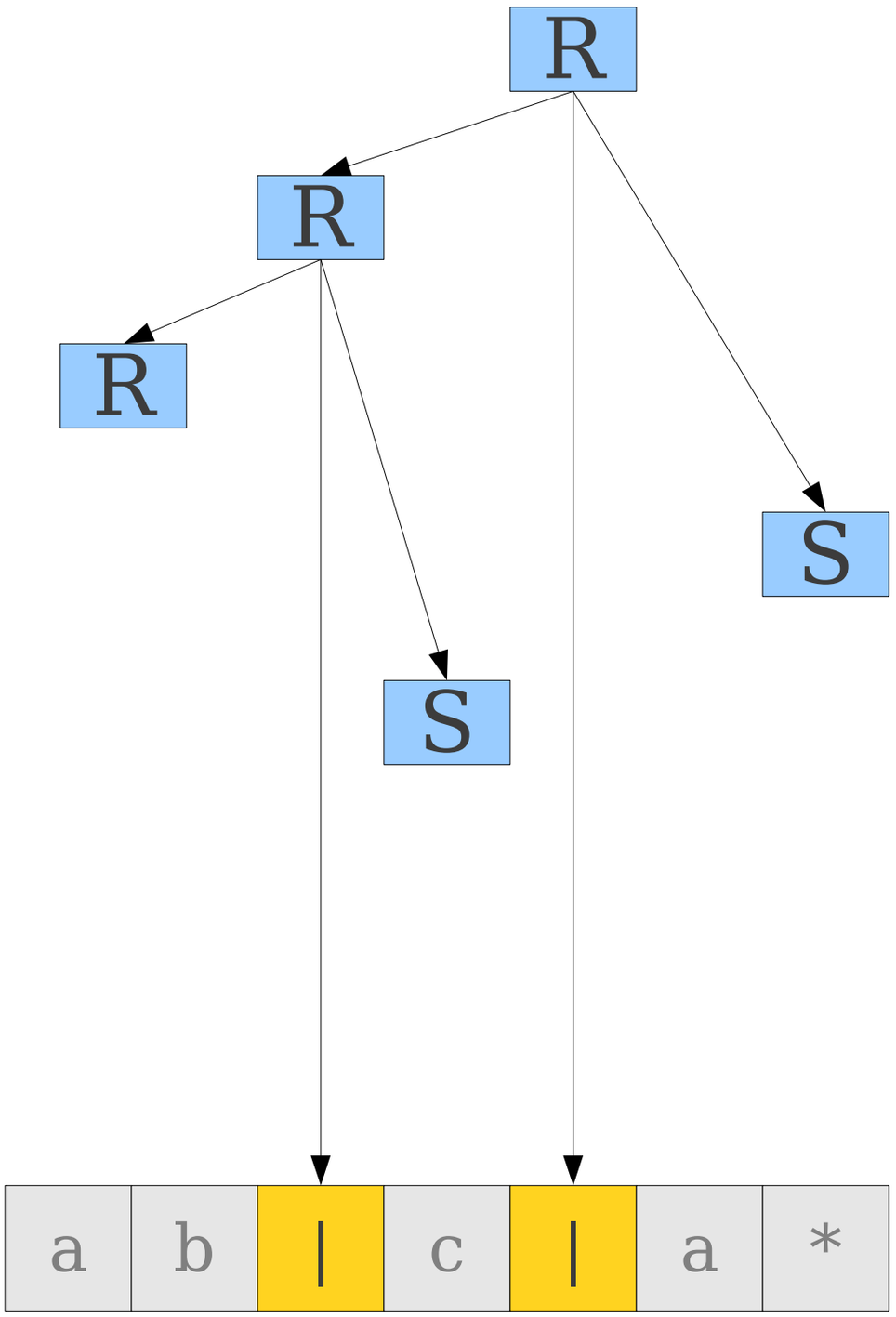
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

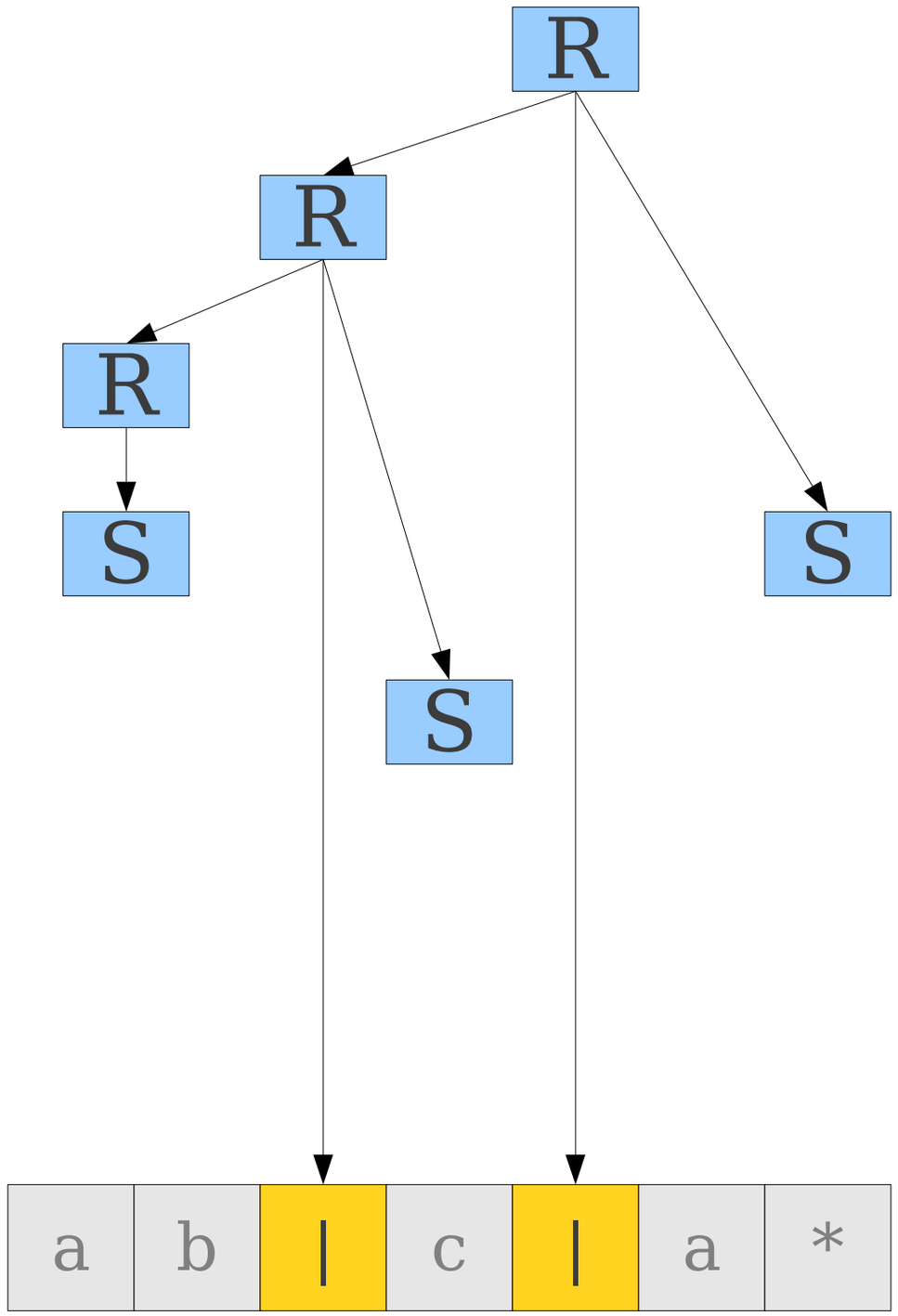
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

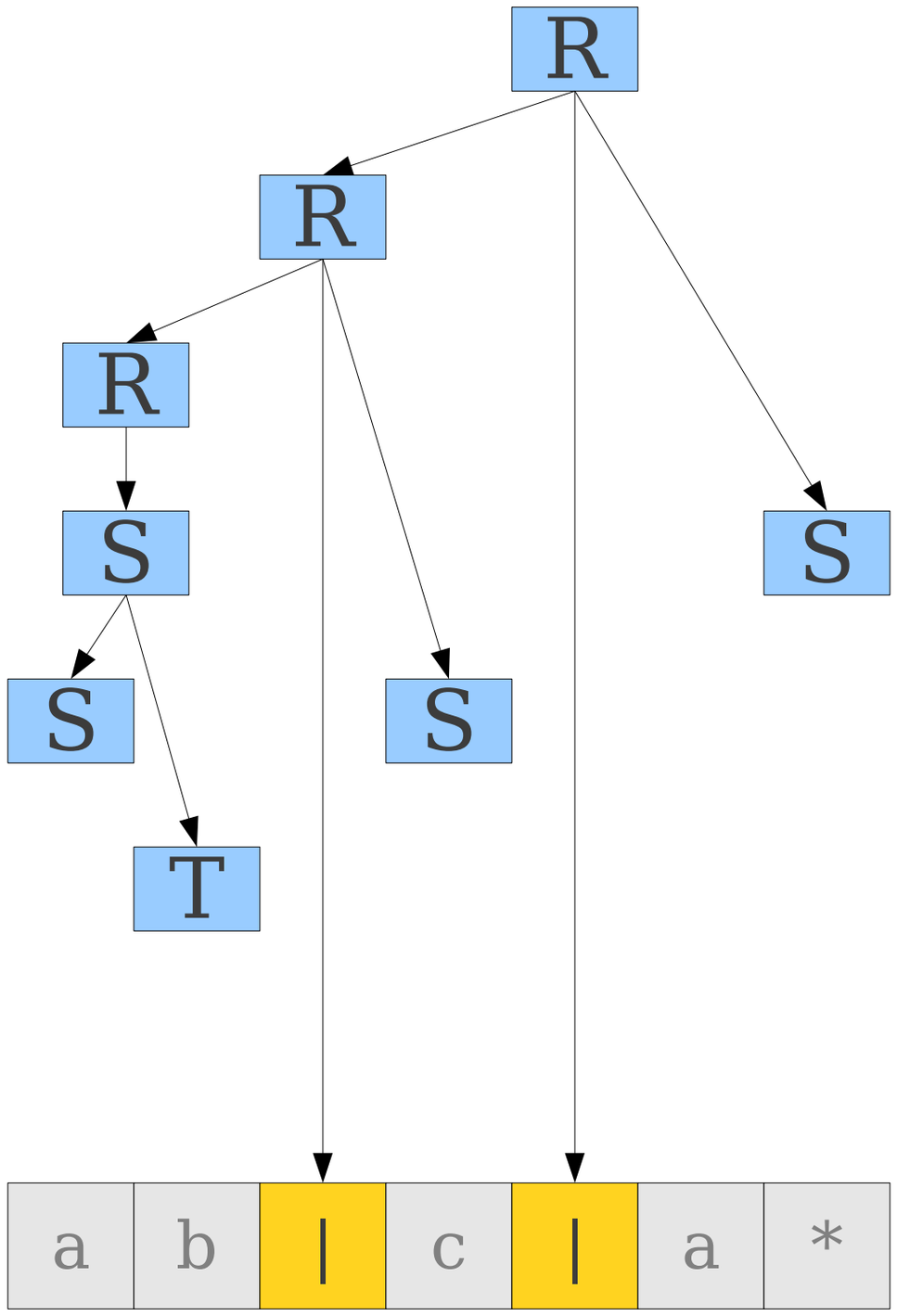
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

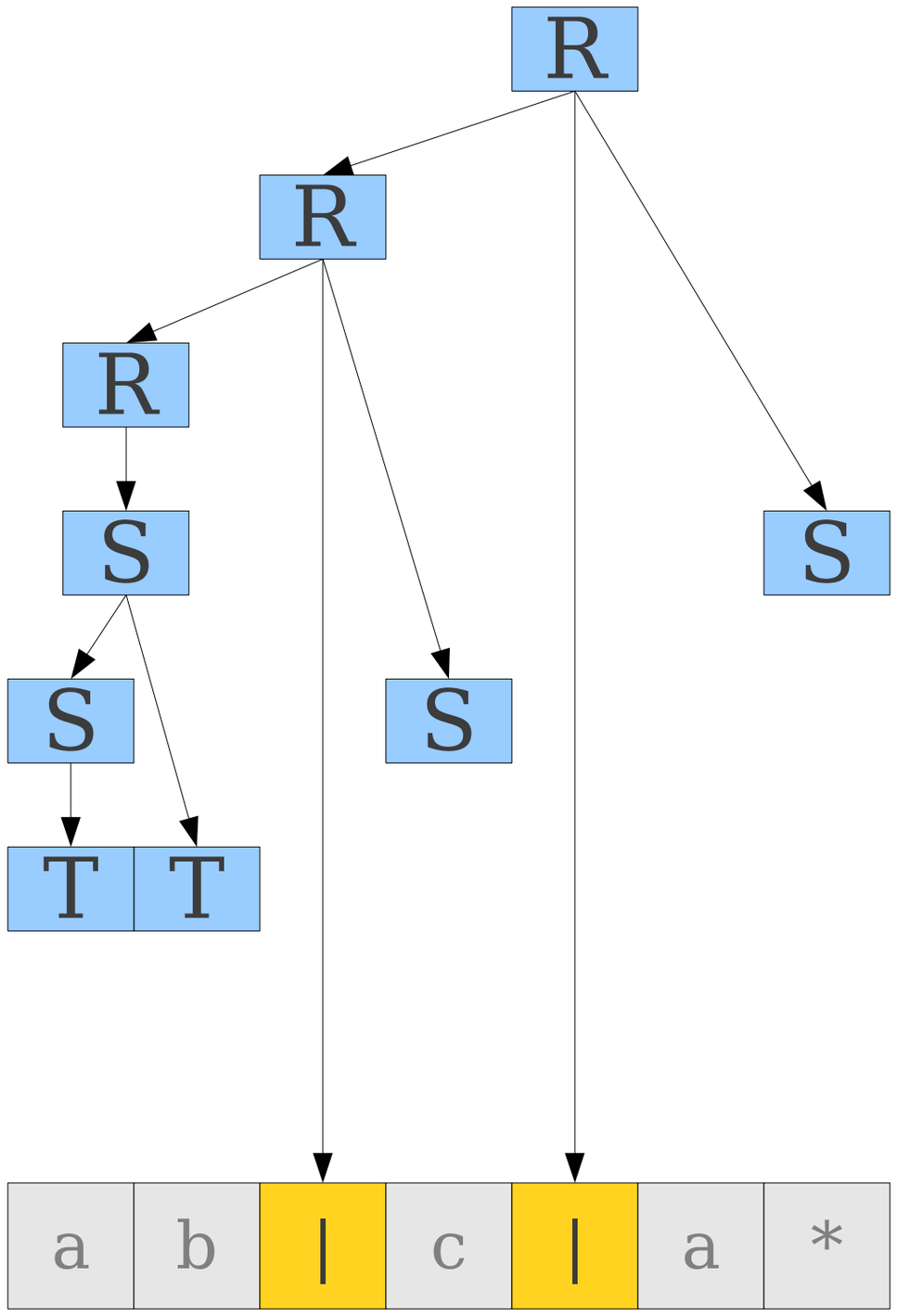
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

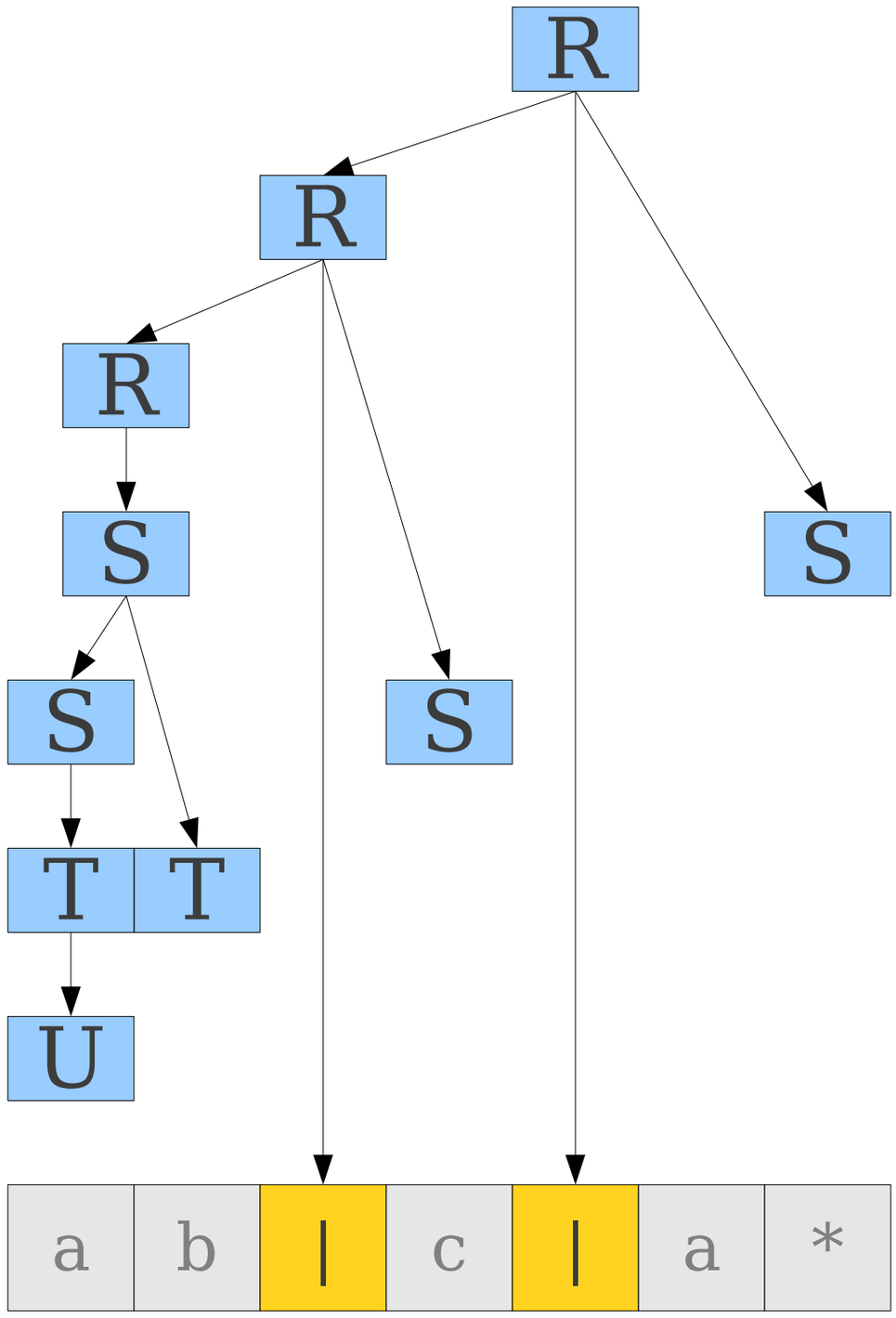
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

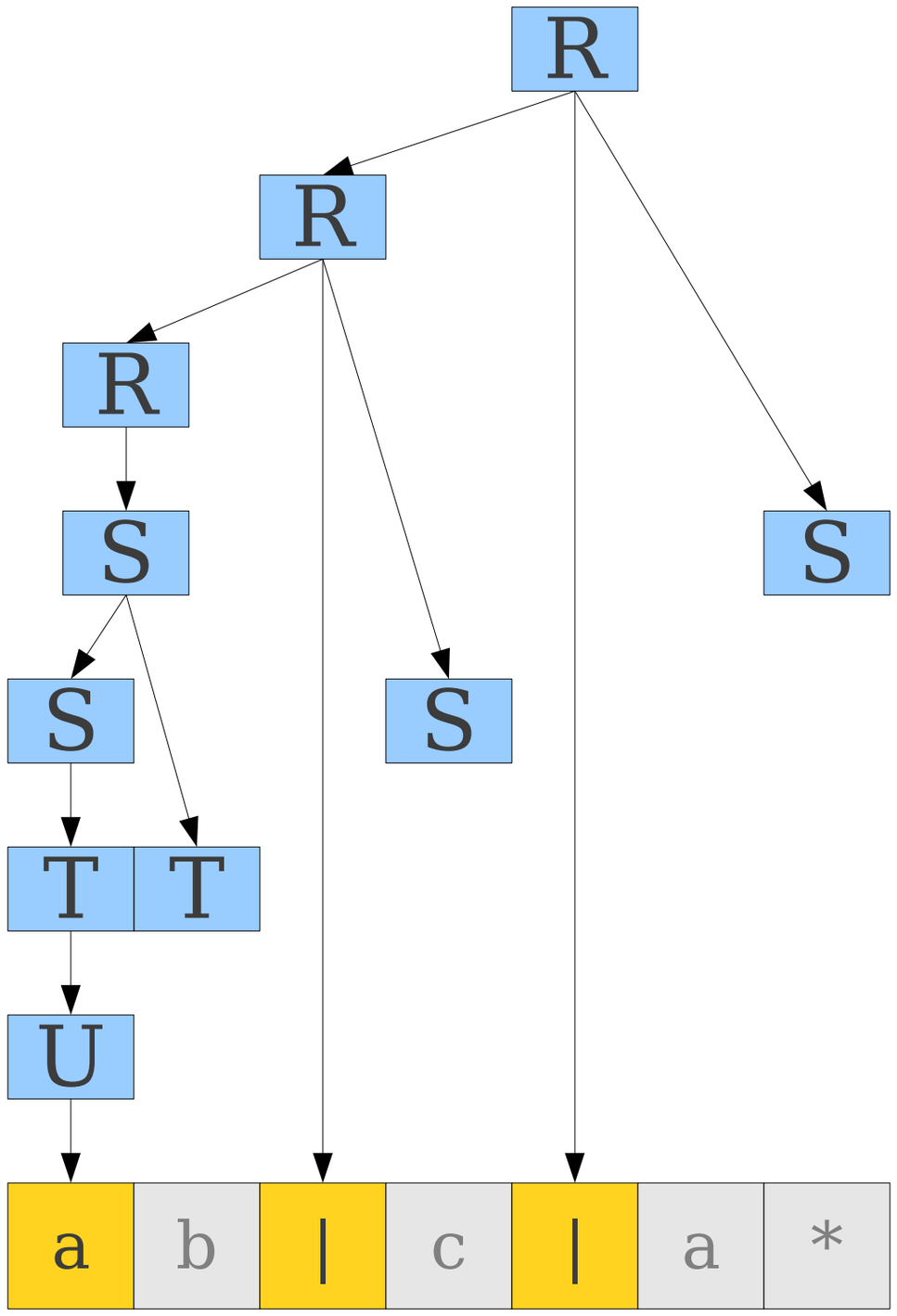
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

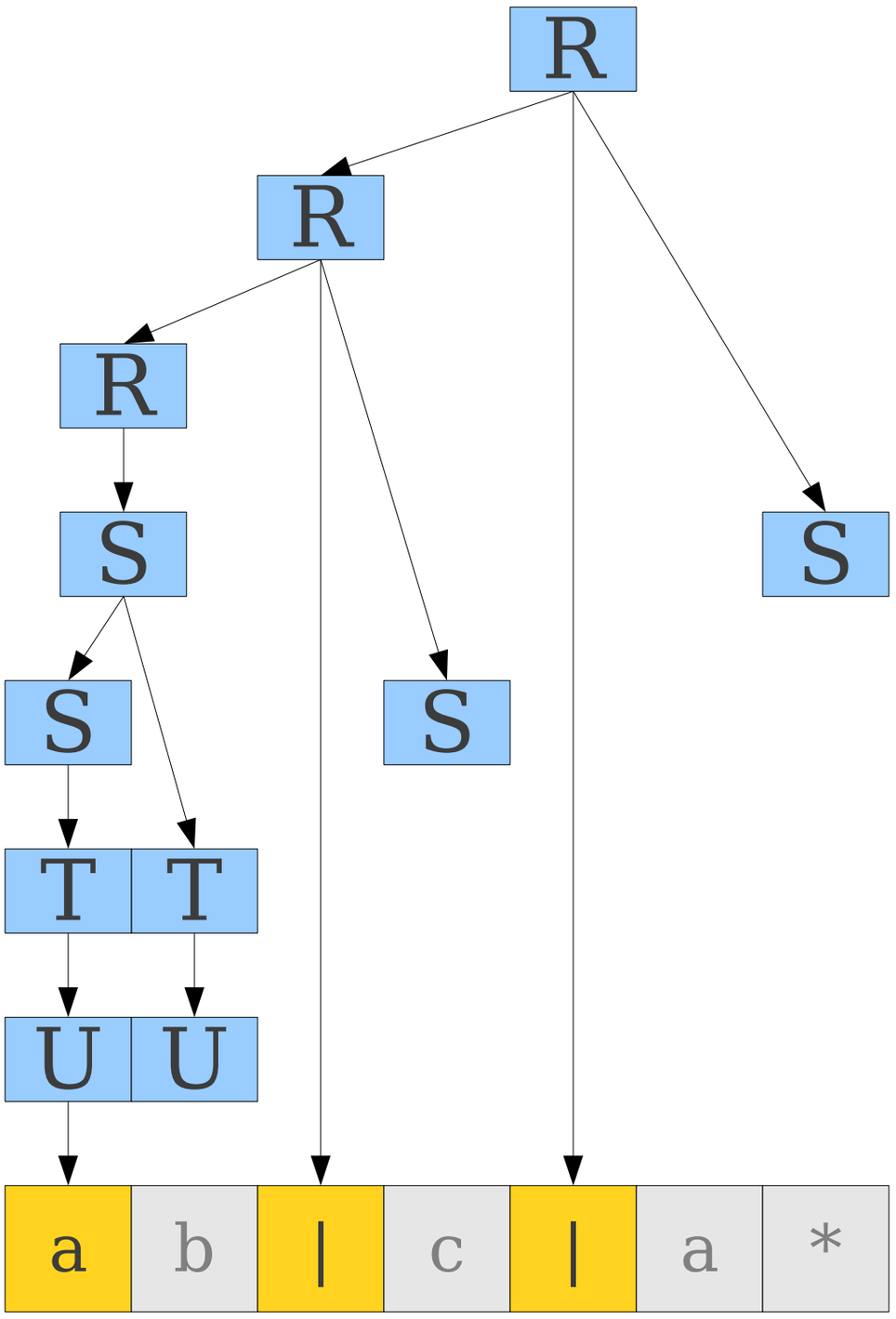
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

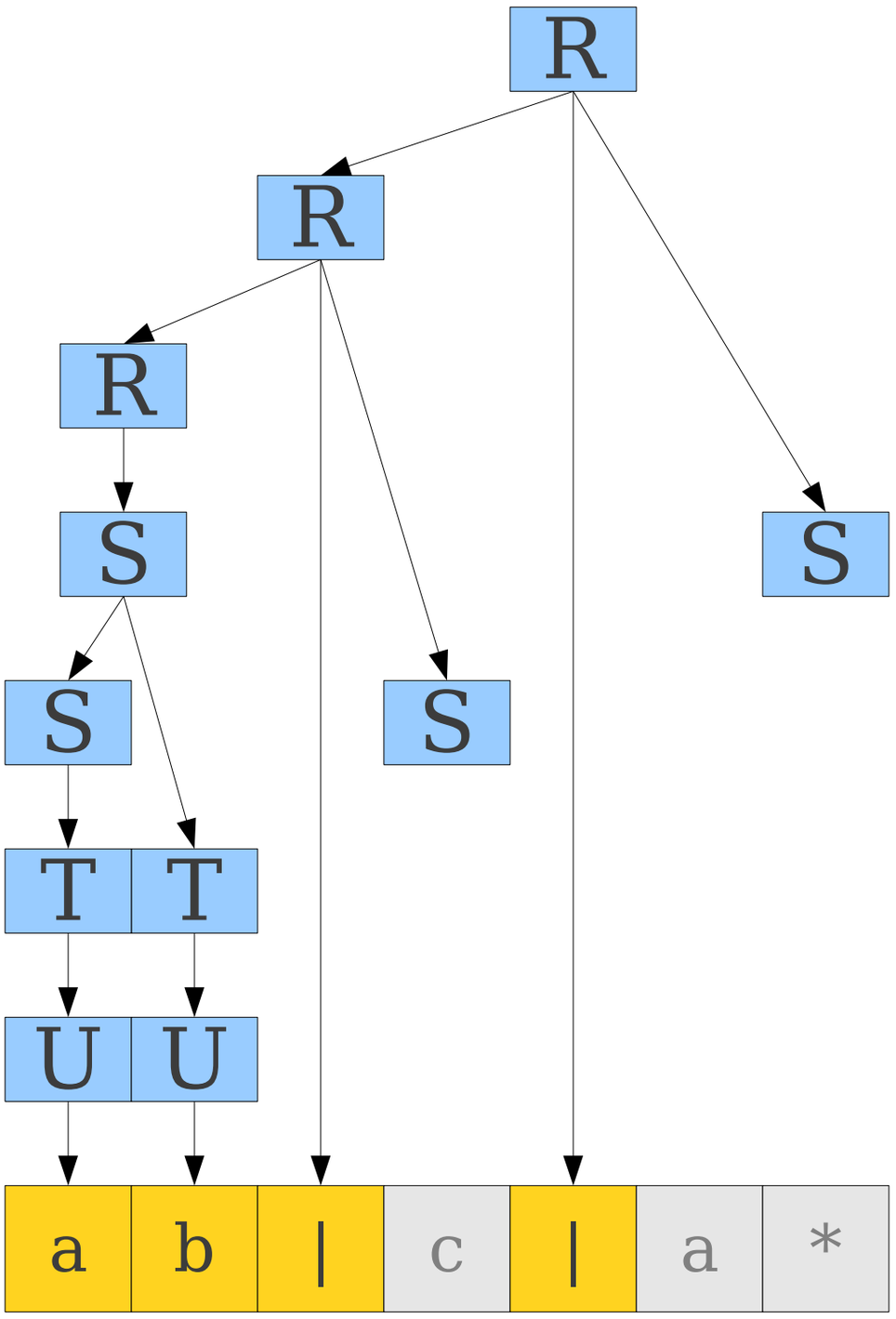
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

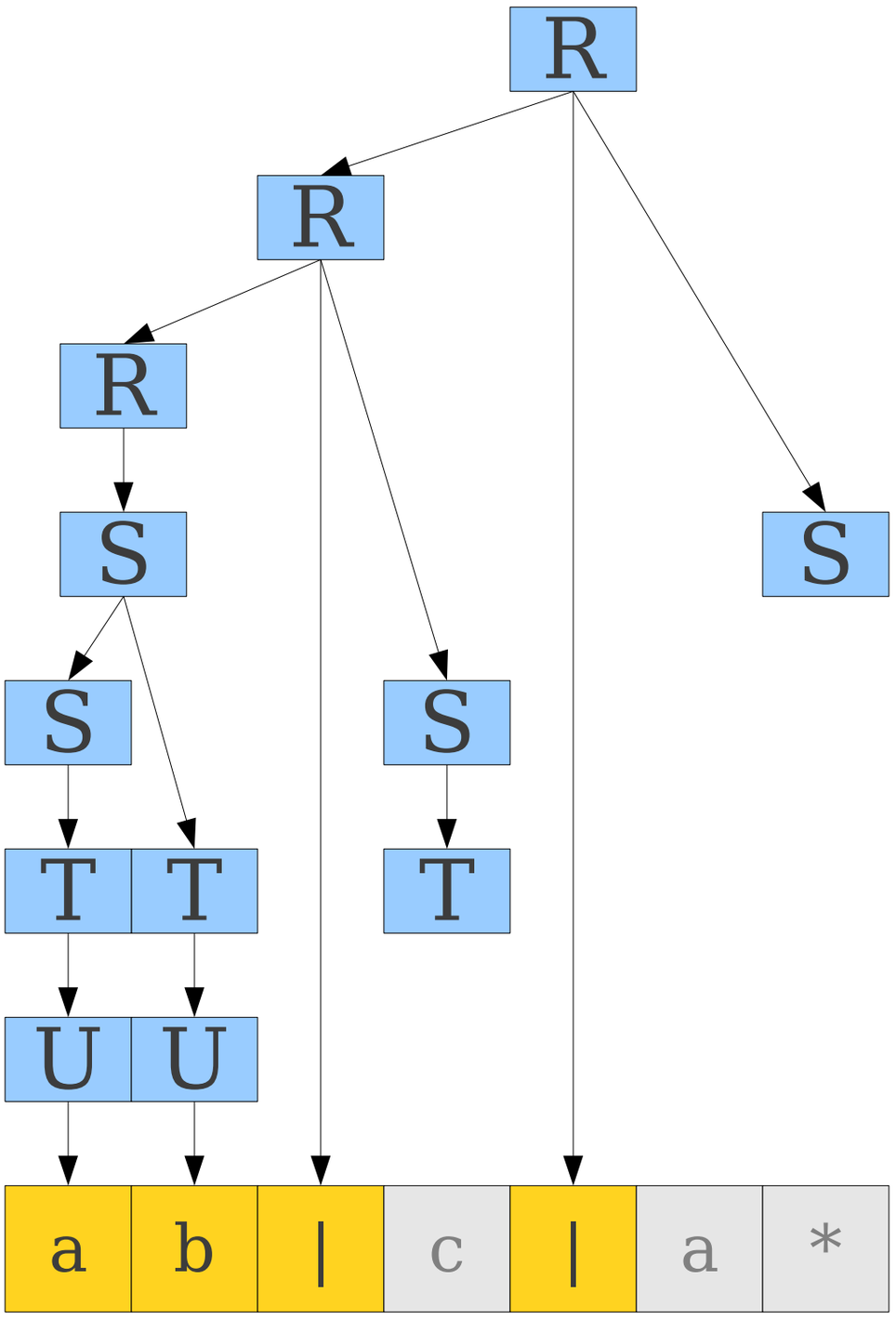
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

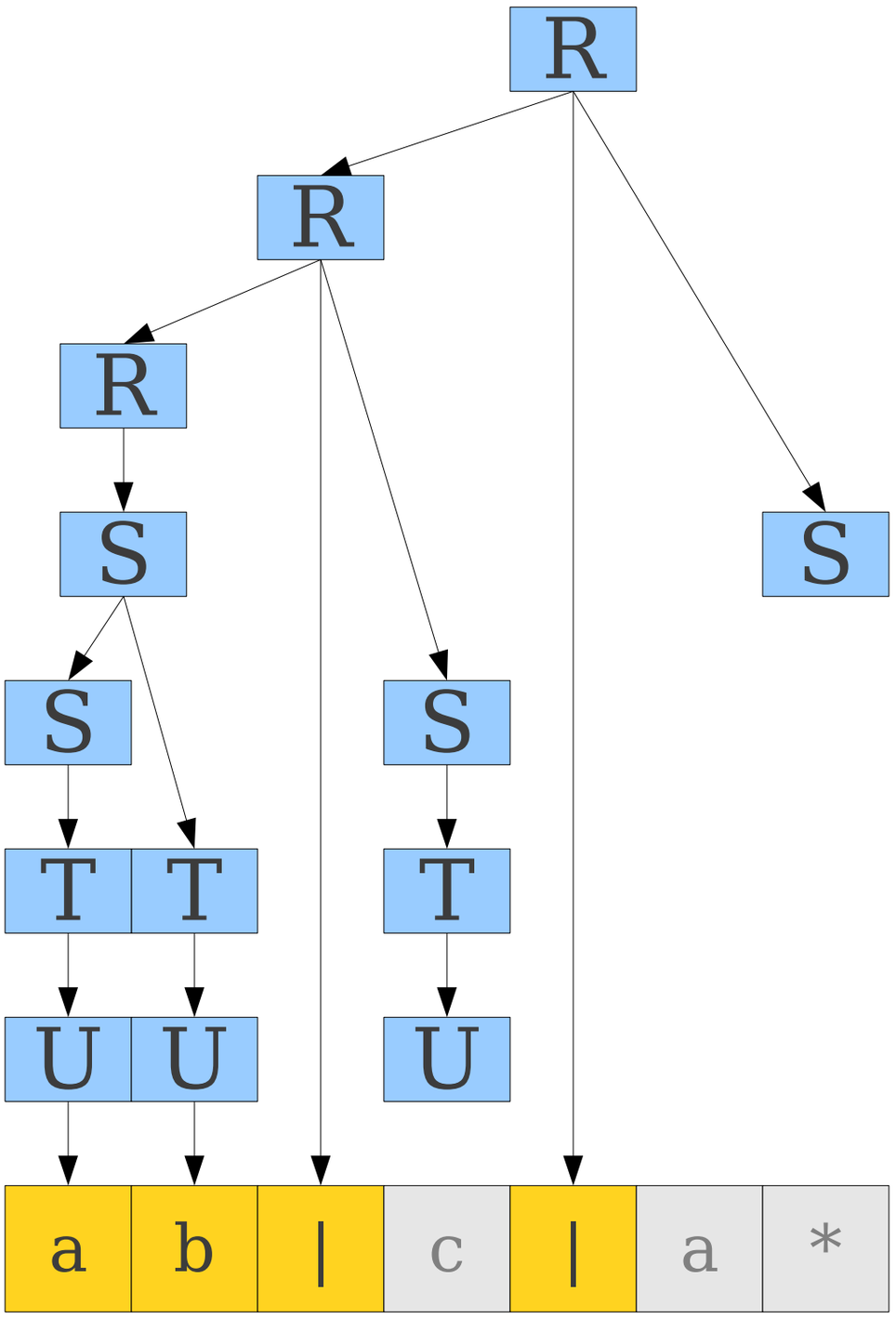
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

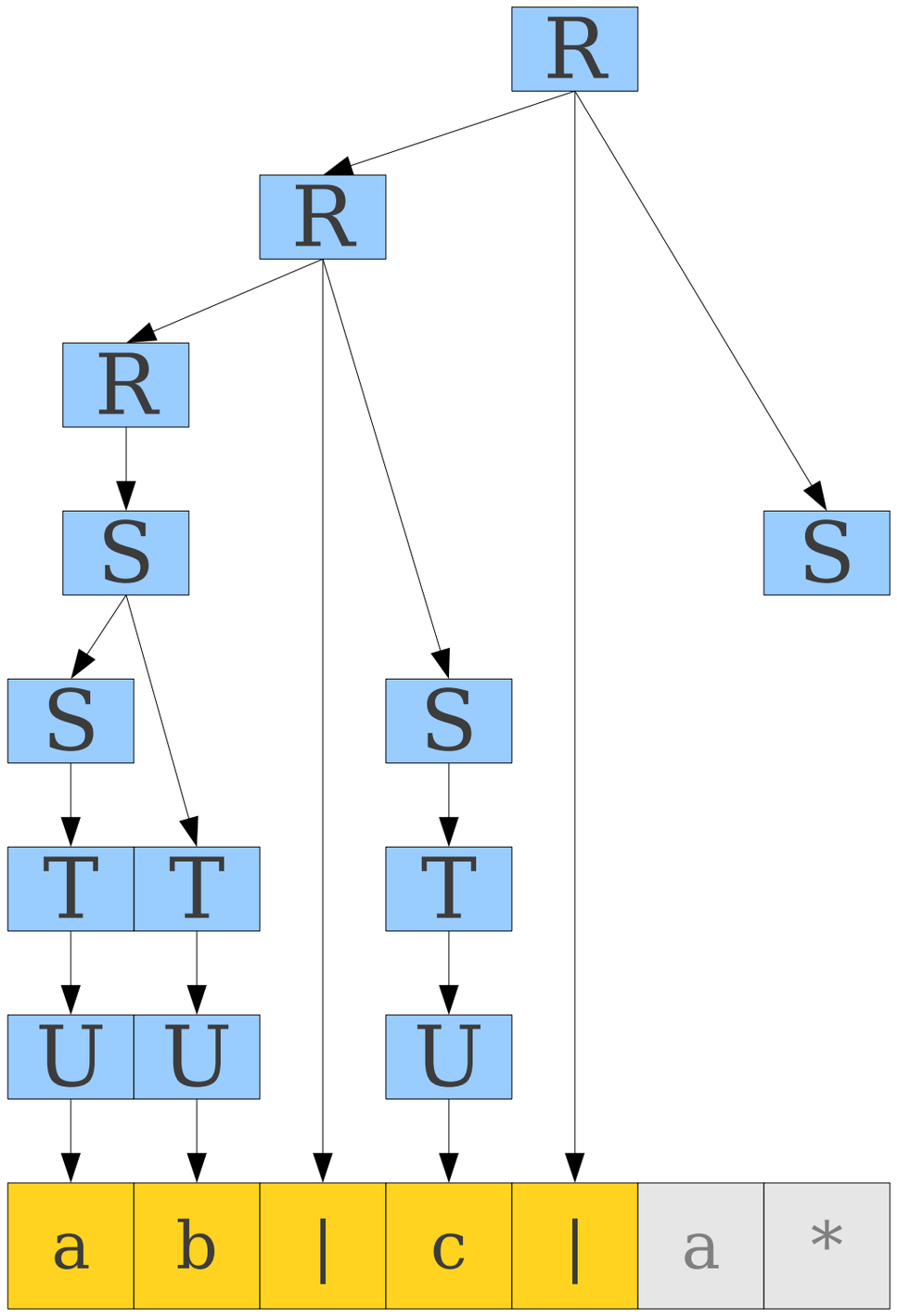
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

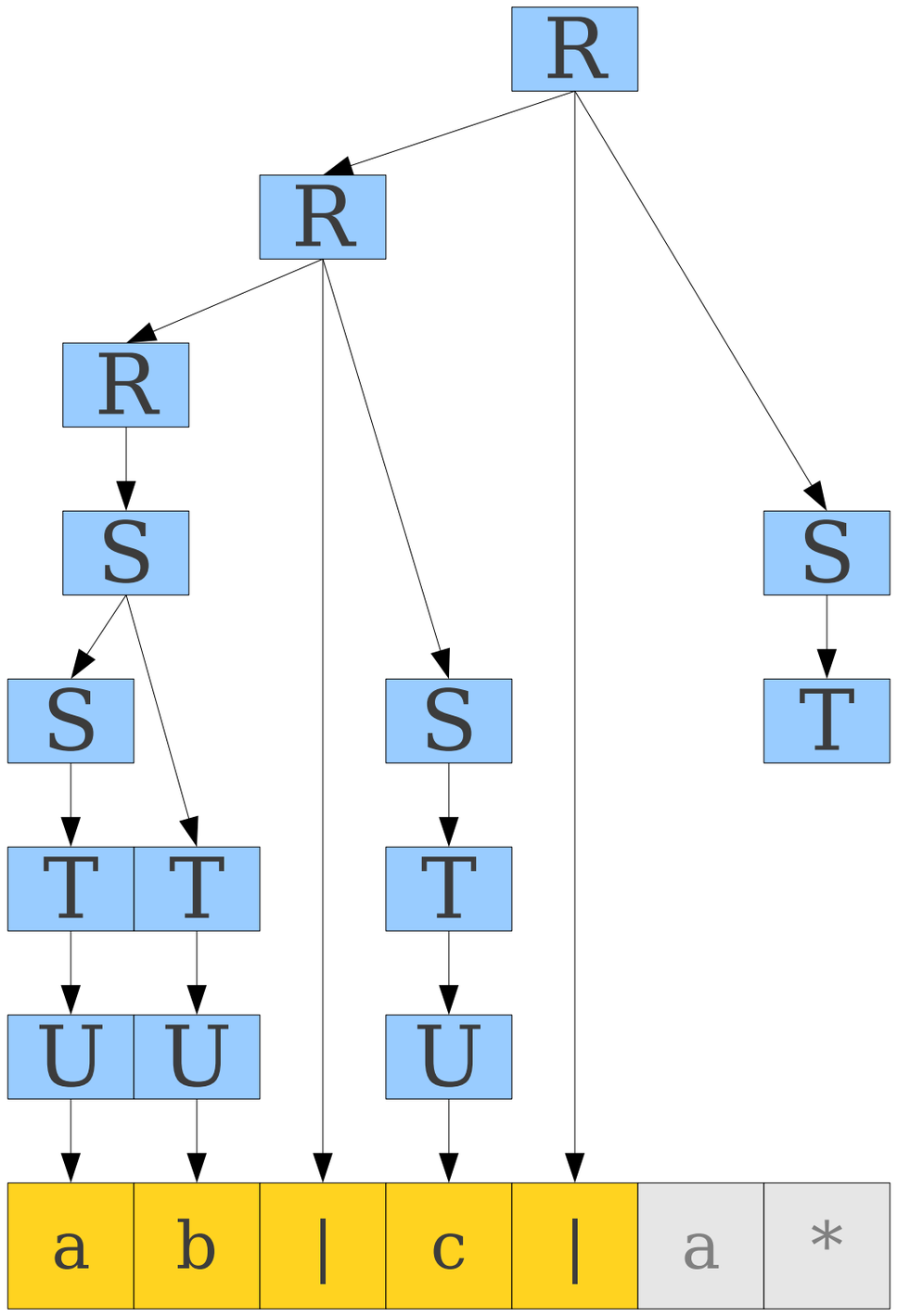
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

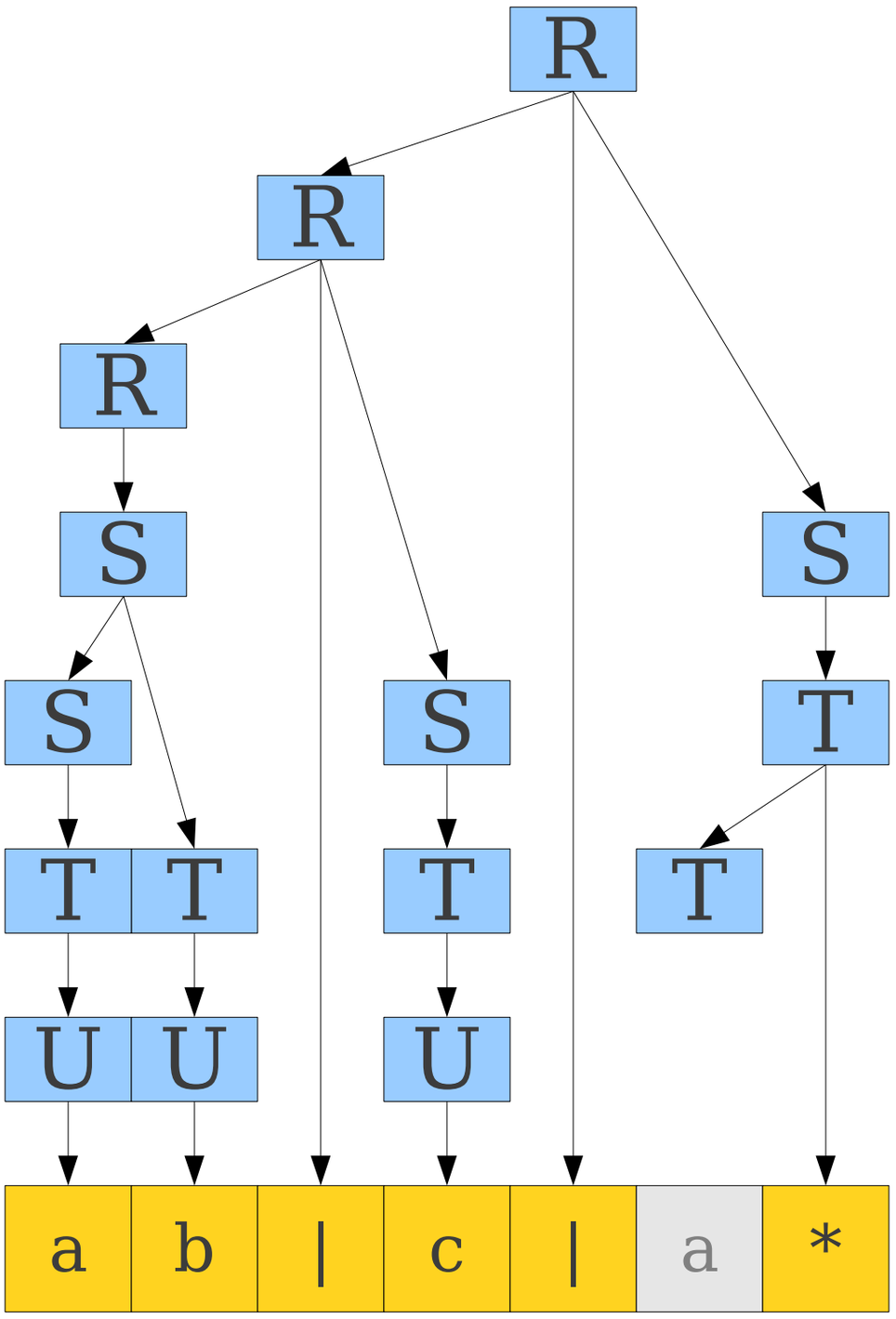
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

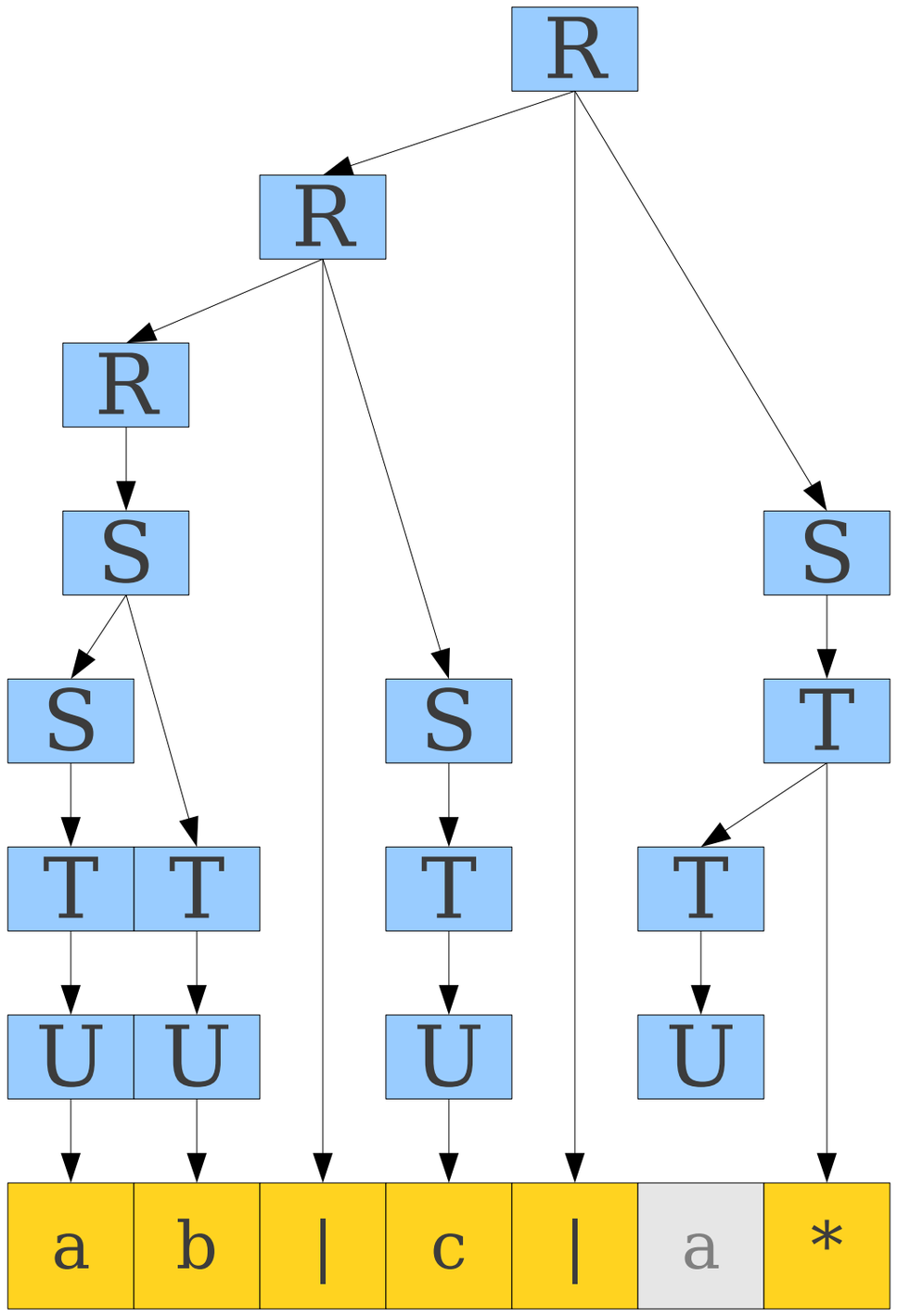
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



R → S | R “|” S

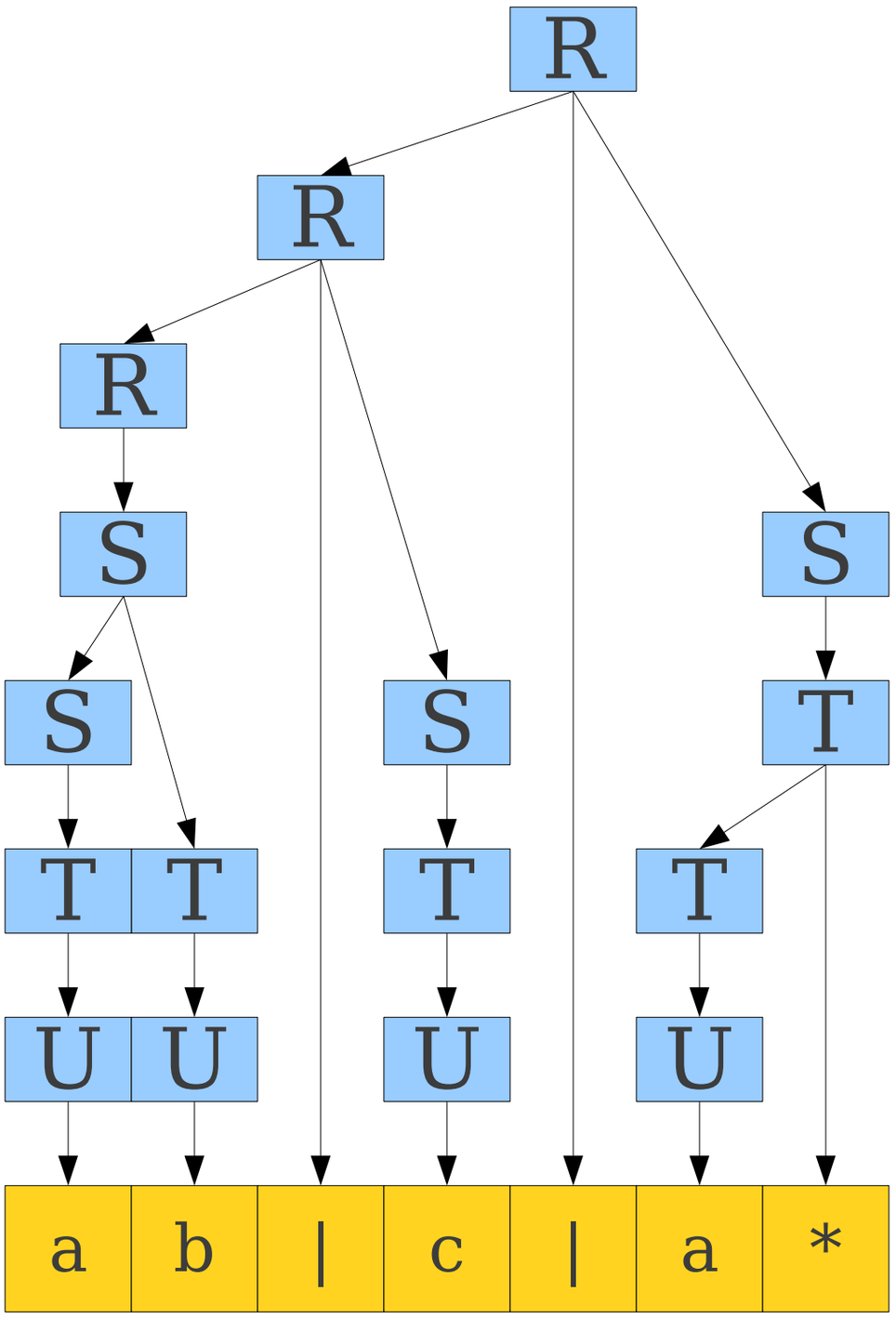
S → T | ST

T → U | T*

U → a | b | c | …

U → “ε”

U → (R)



Precedence Declarations

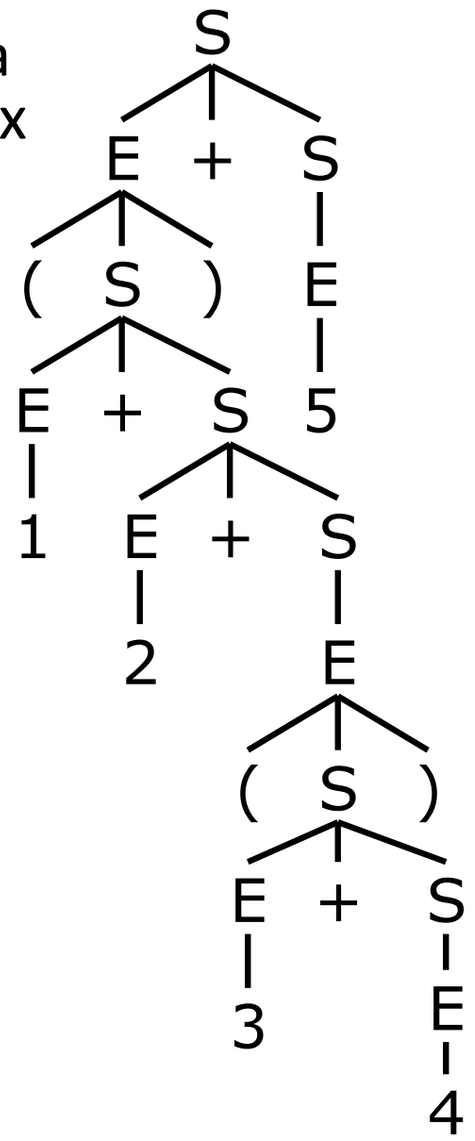
- If we leave the world of pure CFGs, we can often resolve ambiguities through **precedence declarations**.
  - e.g. multiplication has higher precedence than addition, but lower precedence than exponentiation.
- Allows for unambiguous parsing of ambiguous grammars.

Abstract Syntax Trees (ASTs)

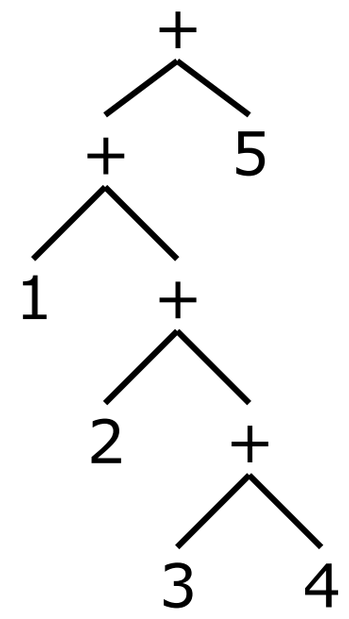
- A parse tree is a **concrete syntax tree**; it shows exactly how the text was derived.
- A more useful structure is an **abstract syntax tree**, which retains only the essential structure of the input.

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**

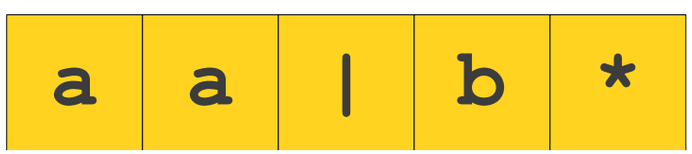
S → **T** | **ST**

T → **U** | **T***

U → **a** | **b** | …

U → “**ε**”

U → **(R)**



The Structure of a Parse Tree

R → S | R “|” S

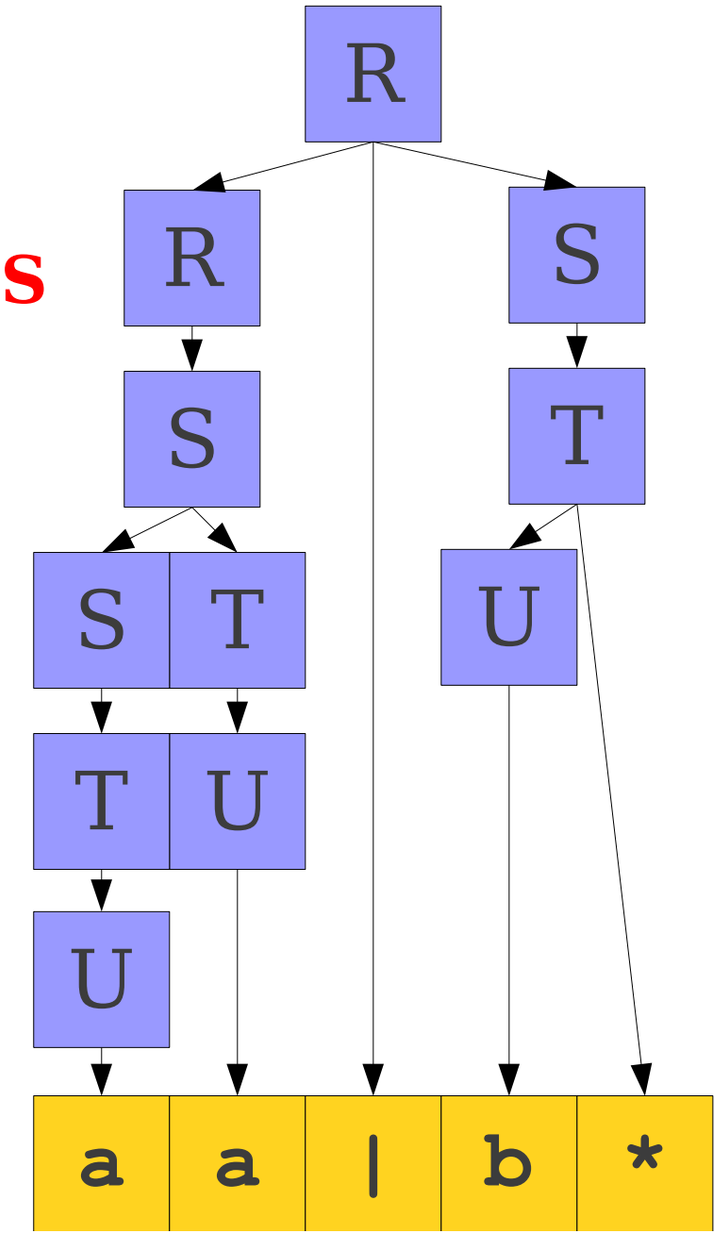
S → T | ST

T → U | T*

U → a | b | …

U → “ε”

U → (R)



The Structure of a Parse Tree

R → S | R “|” S

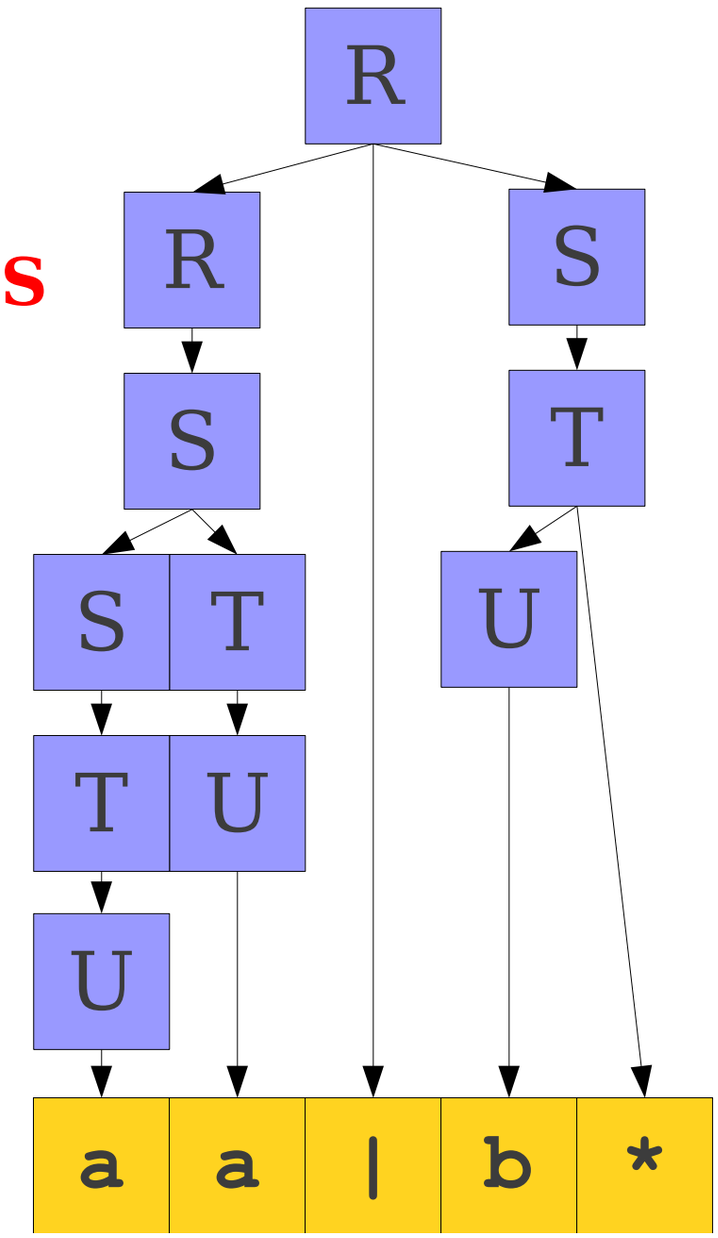
S → T | ST

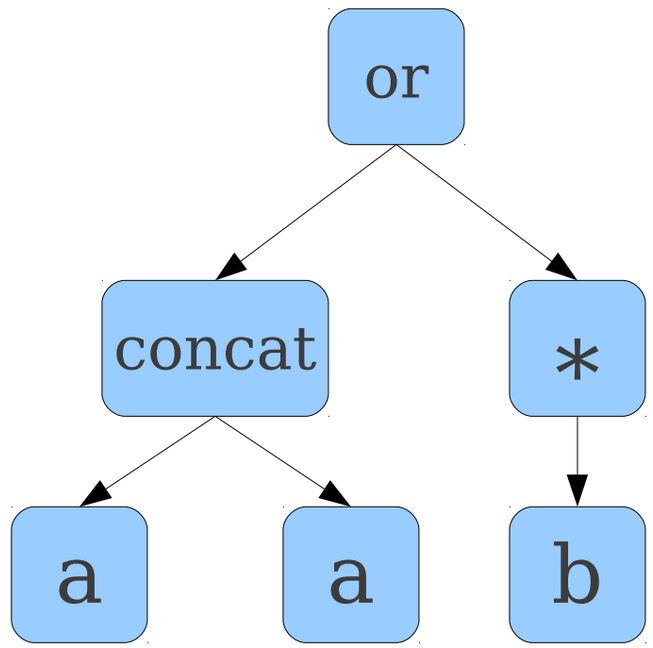
T → U | T*

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

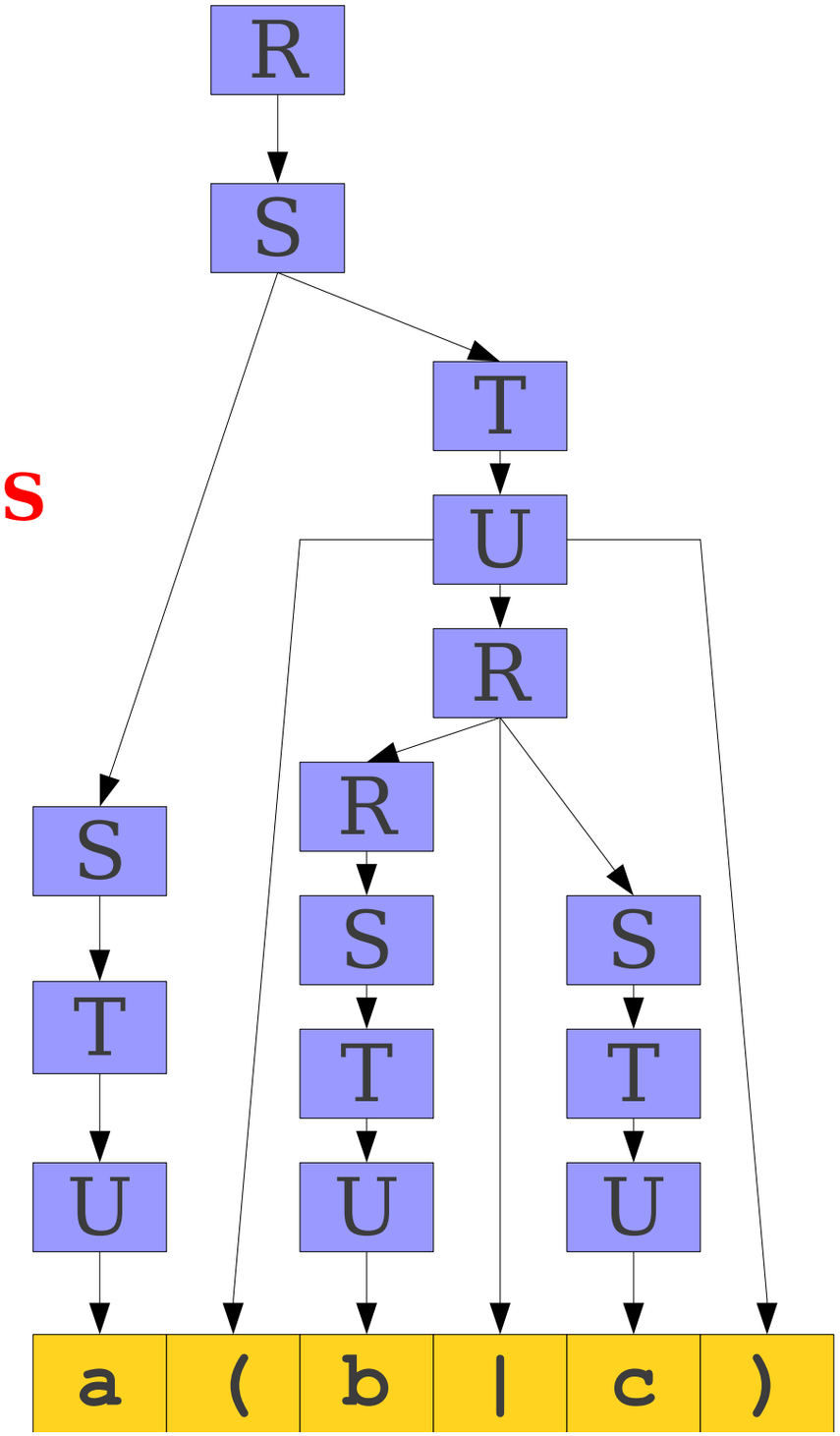
S → T | ST

T → U | T*

U → a | b | …

U → “ε”

U → (R)



R → S | R “|” S

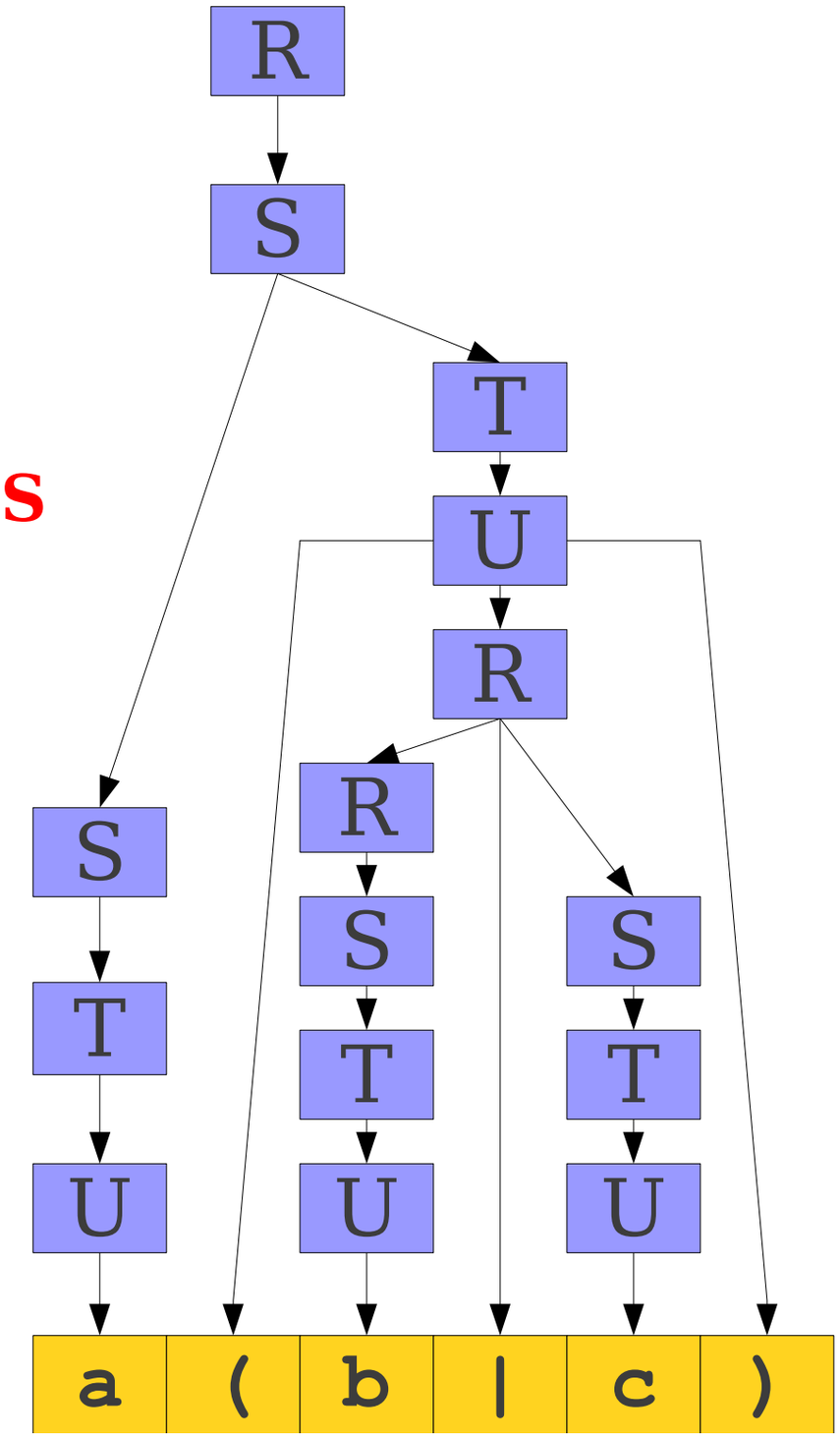
S → T | ST

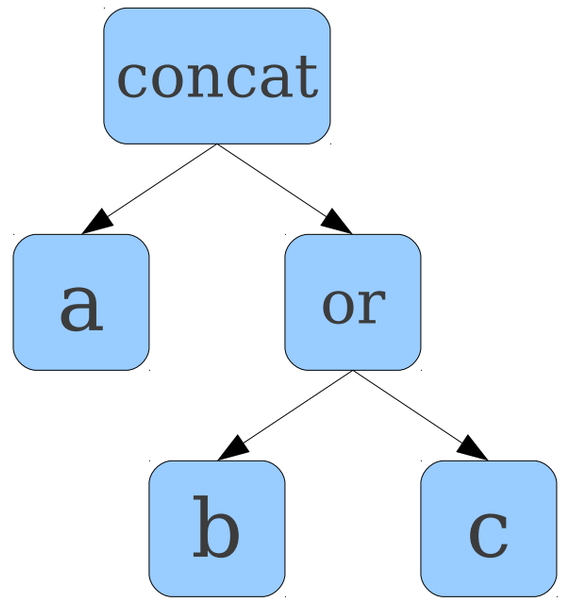
T → U | T*

U → a | b | …

U → “ε”

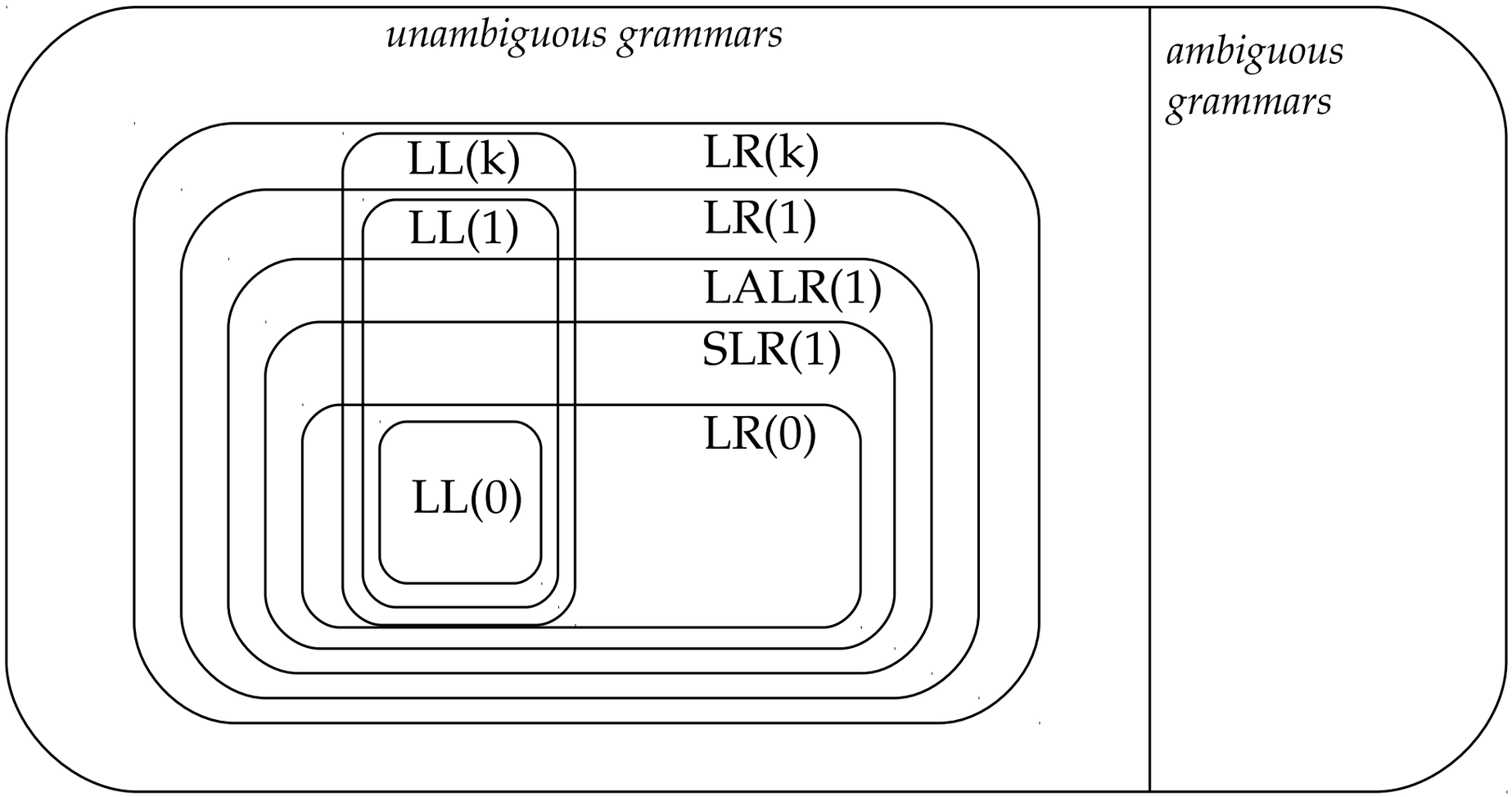
U → (R)





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.


unambiguous grammars
ambiguous grammars
LL(k)
LR(k)
LL(1)
LR(1)
LALR(1)
SLR(1)
LR(0)
LL(0)

Different Types of Parsing

- Top Down Parsing
  - Beginning with the start symbol, try to guess the productions to apply to end up at the user's program.
- Bottom-Up Parsing
  - Beginning with the user's program, try to apply productions in reverse to convert the program back into the start symbol.

Predictive Parsing

Tradeoffs in Prediction

- Predictive parsers are *fast*.
  - Many predictive algorithms can be made to run in linear time.
  - Often can be table-driven for extra performance.
- Predictive parsers are *weak*.
  - Not all grammars can be accepted by predictive parsers.
- Trade *expressiveness* for *speed*.

Exploiting Lookahead

- Given just the start symbol, how do you know which productions to use to get to the input program?
- Idea: Use **lookahead tokens**.
- When trying to decide which production to use, look at some number of tokens of the input to help make the decision.

Implementing Predictive Parsing

- Predictive parsing is only possible if we can predict which production to use given some number of lookahead tokens.
- Increasing the number of lookahead tokens increases the number of grammars we can parse, but complicates the parser.
- Decreasing the number of lookahead tokens decreases the number of grammars we can parse, but simplifies the parser.

A Simple Predictive Parser: **LL(1)**

- Top-down, predictive parsing:
  - **L**: Left-to-right scan of the tokens
  - **L**: Leftmost derivation.
  - **(1)**: One token of lookahead
- Construct a leftmost derivation for the sequence of tokens.
- When expanding a nonterminal, we predict the production to use by looking at the next token of the input. **The decision is forced.**

LL(1) Parse Tables

LL(1) Parse Tables

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

LL(1) Parse Tables

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

Next Input Token

Leftmost NonTerminal

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

rhs of the production rule to use

LL(1) Parsing

```
(int + (int * int))
```

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

LL(1) Parsing

E	(int + (int * int))

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

LL(1) Parsing

E	(int + (int * int))

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

LL(1) Parsing

E$	(int + (int * int))$

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

E$	(int + (int * int))$

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

The `$` symbol is the end-of-input marker and is used by the parser to detect when we have reached the end of the input. It is not a part of the grammar.

LL(1) Parsing

E$	(int + (int * int))$

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

E$	(int + (int * int))$

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

	int	(	)	+	*
E	1	2			
Op				3	4

The first symbol of our guess is a nonterminal. We then look at our parsing table to see what production to use.

This is called a **predict** step.

LL(1) Parsing

E$	(int + (int * int))$

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

	int	(	)	+	*
E	1	2			
Op				3	4

The first symbol of our guess is now a terminal symbol. We thus match it against the first symbol of the string to parse.

This is called a match step.

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

	int	(	)	+	*
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

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

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$
int))$	int))$

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$
int))$	int))$
))$	))$

LL(1) Parsing

(1) E → int
(2) E → (E Op E)
(3) Op → +
(4) Op → *

	int	(	)	+	*
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$
int))$	int))$
))$	))$
)$	)$

LL(1) Parsing

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E$	(int + (int * int))$
(E Op E)$	(int + (int * int))$
E Op E)$	int + (int * int))$
int Op E)$	int + (int * int))$
Op E)$	+ (int * int))$
+ E)$	+ (int * int))$
E)$	(int * int))$
(E Op E))$	(int * int))$
E Op E))$	int * int))$
int Op E))$	int * int))$
Op E))$	* int))$
* E))$	* int))$
E))$	int))$
int))$	int))$
))$	))$
)$	)$
$	$

LL(1) Error Detection

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

```
int + int$
```

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

E$	int + int$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

E$	int + int$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E$	`int + int$`
`int $`	`int + int$`

LL(1) Error Detection

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E$	`int + int$`
`int $`	`int + int$`
`$`	`+ int$`

LL(1) Error Detection

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E$	`int + int$`
`int $`	`int + int$`
`$`	`+ int$`

LL(1) Error Detection, Part II

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

```
(int (int))$
```

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

E$	(int (int))$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

E$	(int (int))$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

E$	`(int (int))$`
`(`E Op E`)`$	`(int (int))$`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E`$`	`(int (int))$`
`(`E Op E`)$`	`(int (int))$`
E Op E`)$`	`int (int))$`

LL(1) Error Detection, Part II

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

E$	`(int (int))$`
(E Op E)$	`(int (int))$`
E Op E)$	`int (int))$`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E`$`	`(int (int))$`
`(`E Op E`)$`	`(int (int))$`
E Op E`)$`	`int (int))$`
`int` Op E`)$`	`int (int))$`

LL(1) Error Detection, Part II

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E`$`	`(int (int))$`
`(`E Op E`)$`	`(int (int))$`
E Op E`)$`	`int (int))$`
`int` Op E`)$`	`int (int))$`
Op E`)$`	`(int))$`

LL(1) Error Detection, Part II

(1) E → `int`
(2) E → `(`E Op E`)`
(3) Op → `+`
(4) Op → `*`

	`int`	`(`	`)`	`+`	`*`
E	1	2			
Op				3	4

E`$`	`(int (int))$`
`(`E Op E`)$`	`(int (int))$`
E Op E`)$`	`int (int))$`
`int` Op E`)$`	`int (int))$`
Op E`)$`	`(int))$`

The LL(1) Algorithm

- Suppose a grammar has start symbol **S** and LL(1) parsing table T. We want to parse string ω
- Initialize a stack containing **S**`$`.
- Repeat until the stack is empty:
  - Let the next character of ω be `t`.
  - If the top of the stack is a terminal `r`:
    - If `r` and `t` don't match, report an error.
    - Otherwise consume the character `t` and pop `r` from the stack.
  - Otherwise, the top of the stack is a nonterminal **A**:
    - If T[**A**, `t`] is undefined, report an error.
    - Replace the top of the stack with T[**A**, `t`].

A Simple LL(1) Grammar

```
STMT  → if EXPR then STMT
      | while EXPR do STMT
      | EXPR ;

EXPR  → TERM -> id
      | zero? TERM
      | not EXPR
      | ++ id
      | -- id

TERM  → id
      | constant
```

A Simple LL(1) Grammar

```
STMT  → if EXPR then STMT
      | while EXPR do STMT
      | EXPR ;

EXPR  → TERM -> id
      | zero? TERM
      | not EXPR
      | ++ id
      | -- id

TERM  → id
      | constant
```

```
id -> id;

while not zero? id
   do --id;

if not zero? id then
   if not zero? id then
        constant -> id;
```

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT												
EXPR												
TERM												

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT												
EXPR												
TERM												

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT												
EXPR												
TERM										9	10	

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT												
EXPR												
TERM										9	10	

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT												
EXPR					5	6	7	8				
TERM										9	10	

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT												
EXPR					5	6	7	8				
TERM										9	10	

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT												
EXPR					5	6	7	8		4	4	
TERM										9	10	

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT												
EXPR					5	6	7	8		4	4	
TERM										9	10	

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2									
EXPR					5	6	7	8		4	4	
TERM										9	10	

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2									
EXPR					5	6	7	8		4	4	
TERM										9	10	

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2		3	3	3	3				
EXPR					5	6	7	8		4	4	
TERM										9	10	

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2		3	3	3	3				
EXPR					5	6	7	8		4	4	
TERM										9	10	

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2		3	3	3	3		3	3	
EXPR					5	6	7	8		4	4	
TERM										9	10	

Constructing LL(1) Parse Tables

STMT	→	**if EXPR then STMT**	**(1)**
	|	**while EXPR do STMT**	**(2)**
	|	**EXPR ;**	**(3)**
EXPR	→	**TERM -> id**	**(4)**
	|	**zero? TERM**	**(5)**
	|	**not EXPR**	**(6)**
	|	**++ id**	**(7)**
	|	**-- id**	**(8)**
TERM	→	**id**	**(9)**
	|	**constant**	**(10)**

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2		3	3	3	3		3	3	
EXPR					5	6	7	8		4	4	
TERM										9	10	

The Limits of LL(1)

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → **A**`b` | `c`

- FIRST(**A**) = {`c`}
- However, we cannot build an LL(1) parse table.
- **Why?**

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → **A**`b` | `c`

- FIRST(**A**) = {`c`}
- However, we cannot build an LL(1) parse table.
- **Why?**

	b	c
A		**A** → **A**`b` **A** → `c`

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → **A**`b` | `c`

- FIRST(**A**) = {`c`}
- However, we cannot build an LL(1) parse table.
- **Why?**

	b	c
A		**A** → **A**`b` **A** → `c`

- **Cannot uniquely predict production!**
- This is called a **FIRST/FIRST conflict**.

Eliminating Left Recursion

- In general, left recursion can be converted into **right recursion** by a mechanical transformation.
- Consider the grammar

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

- This will produce $\boldsymbol{\alpha}$ followed by some number of $\boldsymbol{\omega}$'s.
- Can rewrite the grammar as

$$\mathbf{A} \rightarrow \boldsymbol{\alpha}\mathbf{B}$$

$$\mathbf{B} \rightarrow \boldsymbol{\varepsilon} \mid \boldsymbol{\omega}\mathbf{B}$$

LL(1) Straightforward and Fast

- Can be implemented quickly with a table-driven design
- Can be implemented by recursive descent:
  - Define a function for each nonterminal
  - Have these functions call each other based on the lookahead token