بسم الله الرحمن الرحيم

Parsing:

Context-Free Grammars, Parsing and Programming Languages (2)

Comparison with Lexical Analysis

Phase	Input	Output
Lexer	String of characters	String of tokens
Parser	String of tokens	Parse tree (may be implicit)

Example

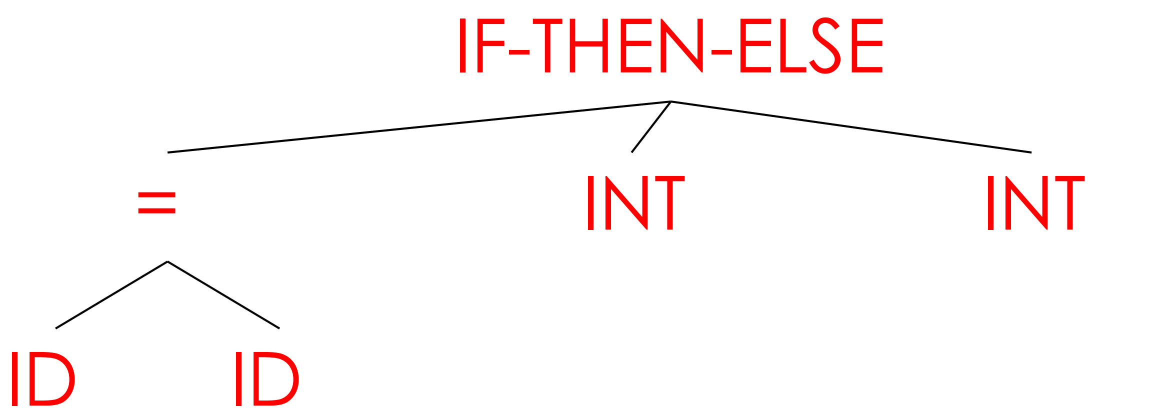
- Cool

  if x = y then 1 else 2 fi

- Parser input

  IF ID = ID THEN INT ELSE INT FI

- Parser output

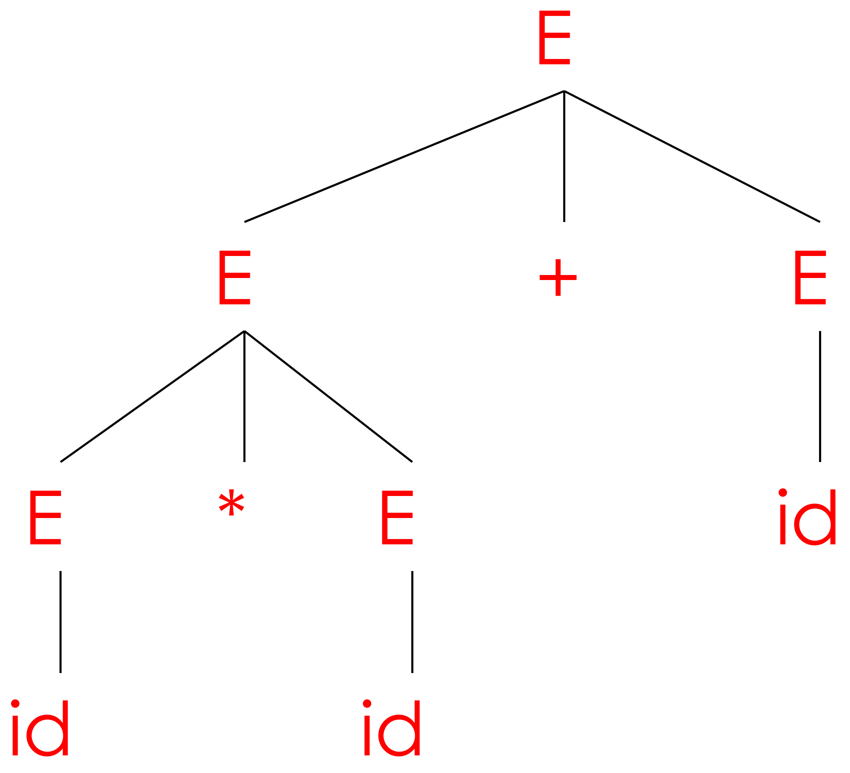


Ambiguity

- Grammar $E \rightarrow E + E \mid E * E \mid (E) \mid id$
- String $id * id + id$

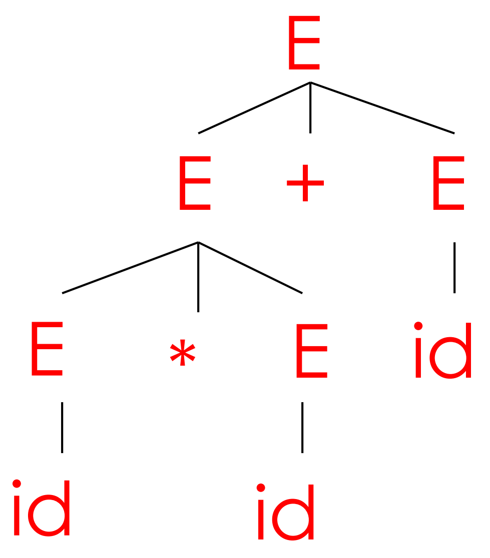
Derivation Example (Cont.)

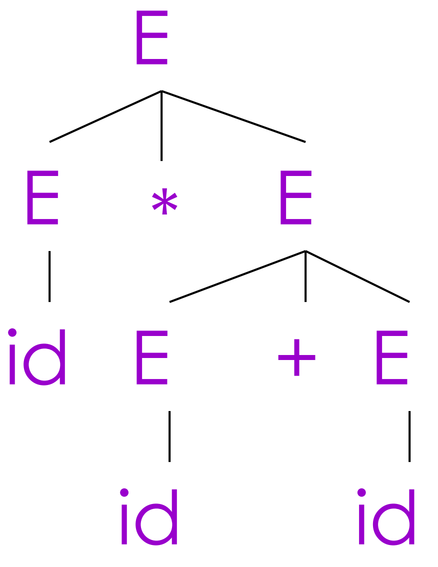
$$
\begin{aligned}
& E \\
\rightarrow\ & E+E \\
\rightarrow\ & E * E+E \\
\rightarrow\ & id * E + E \\
\rightarrow\ & id * id + E \\
\rightarrow\ & id * id + id
\end{aligned}
$$



Ambiguity (Cont.)

This string has two parse trees



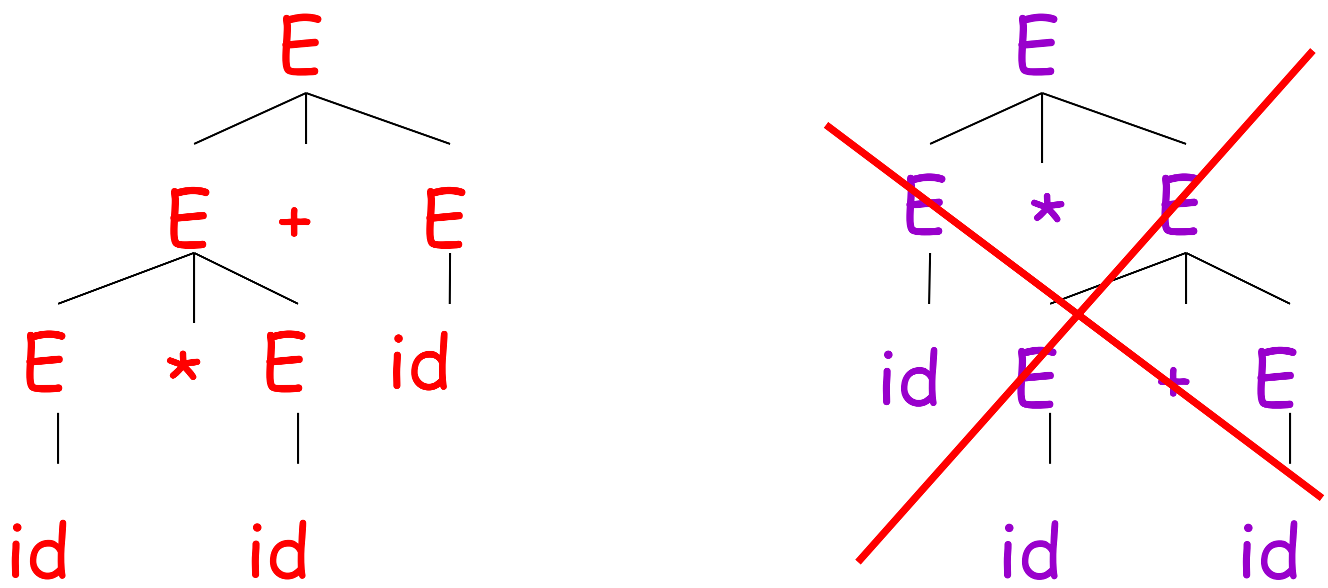


Ambiguity in Arithmetic Expressions

- Recall the grammar

  $E \rightarrow E + E \mid E * E \mid (E) \mid id$

- The string $id * id + id$ has two parse trees:



Dealing with Ambiguity

- There are several ways to handle ambiguity
- Most direct method is to rewrite grammar unambiguously

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

- Enforces precedence of * over +

Ambiguity: The Dangling Else

- Consider the grammar

$$S \rightarrow \text{if } E \text{ then } S$$
$$\mid \text{if } E \text{ then } S \text{ else } S$$
$$\mid \text{OTHER}$$

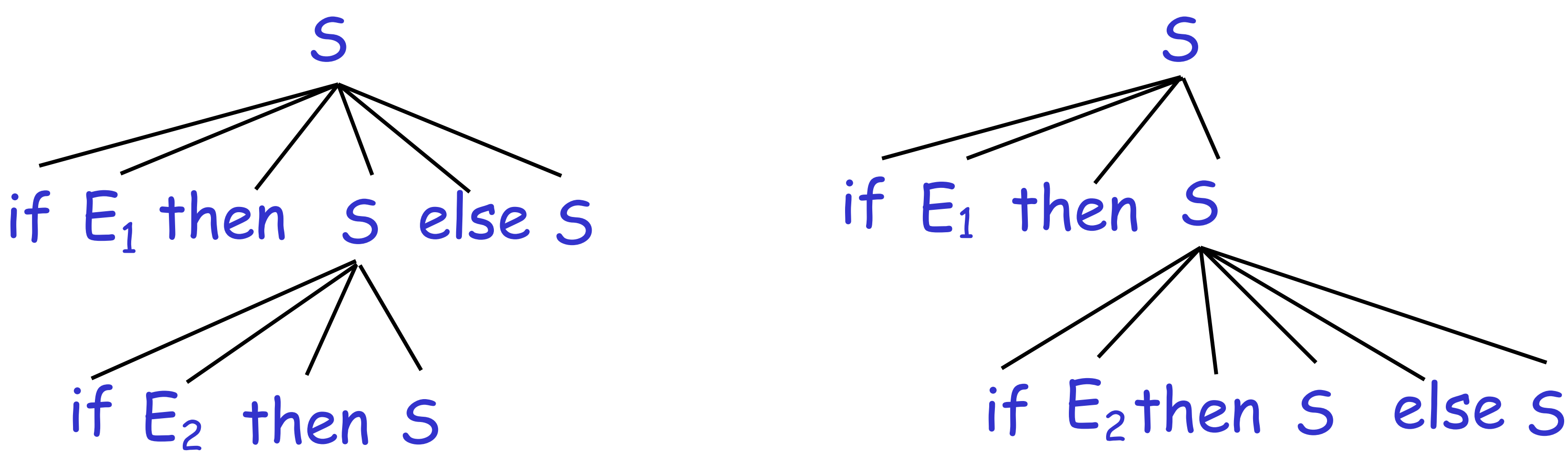
- This grammar is also ambiguous

The Dangling Else: Example

- The expression

  if E_1 then if E_2 then S else S

  has two parse trees



- Typically we want the second form

The Dangling Else: A Fix

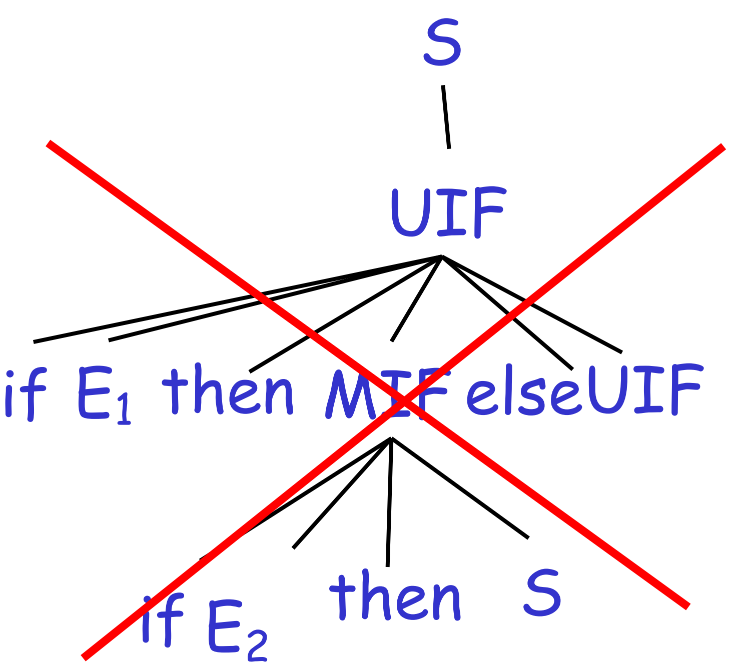
- else matches the closest unmatched then
- We can describe this in the grammar

```
S → MIF                      /* all then are matched */
  | UIF                      /* some then is unmatched */
MIF → if E then MIF else MIF
    | OTHER
UIF → if E then S
    | if E then MIF else UIF
```

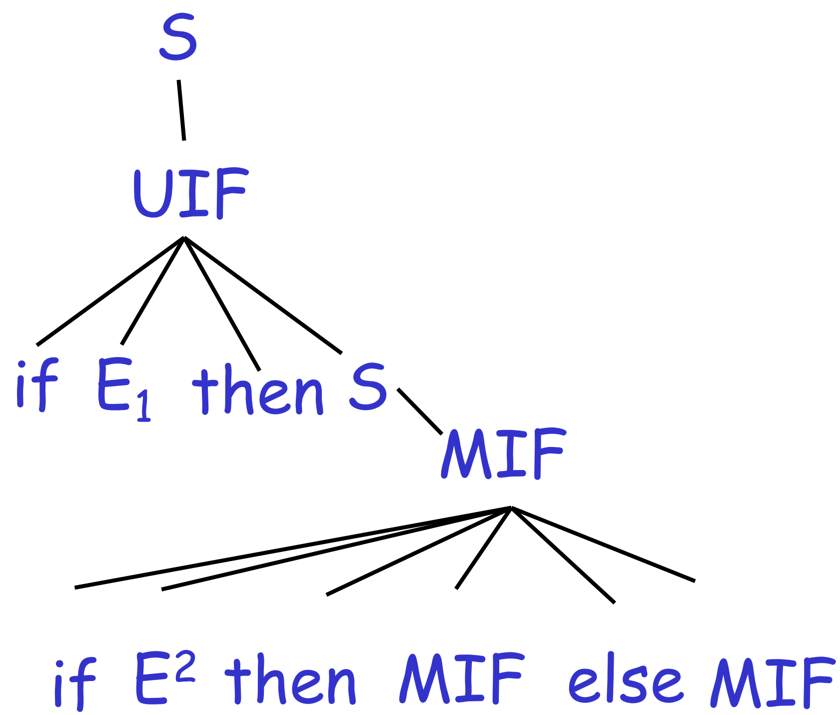
- Describes the same set of strings

The Dangling Else: Example Revisited

- The expression if E_1 then if E_2 then S else S



- Not valid because the then expression is not a MIF



- A valid parse tree (for a UIF)

Ambiguity

- No general techniques for handling ambiguity
- Impossible to convert automatically an ambiguous grammar to an unambiguous one
- Used with care, ambiguity can simplify the grammar
  - Sometimes allows more natural definitions
  - We need disambiguation mechanisms

Resolving Ambiguity

- If a grammar can be made unambiguous, 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:

$$P \rightarrow \varepsilon \mid PP \mid (P)$$

Balanced Parentheses

- Given the grammar $\mathbf{P} \to \boldsymbol{\varepsilon} \mid \mathbf{PP} \mid \mathbf{(P)}$
- How might we generate the string `(()())`?

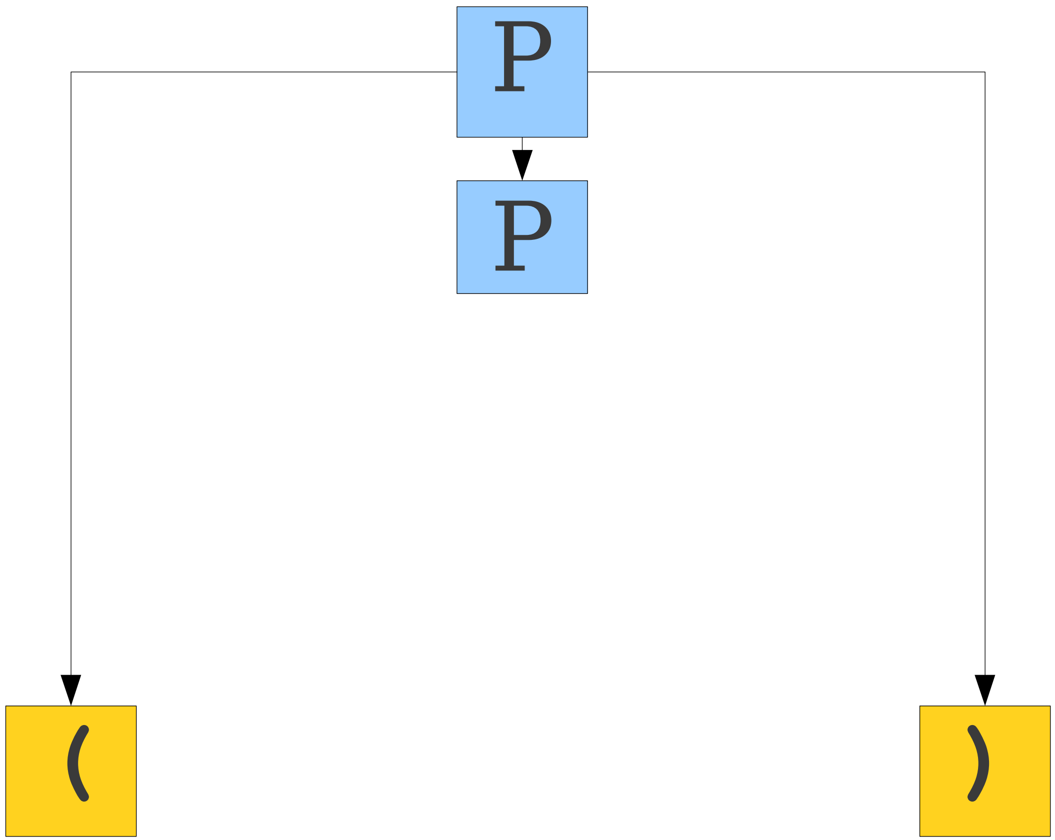
Balanced Parentheses

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

P

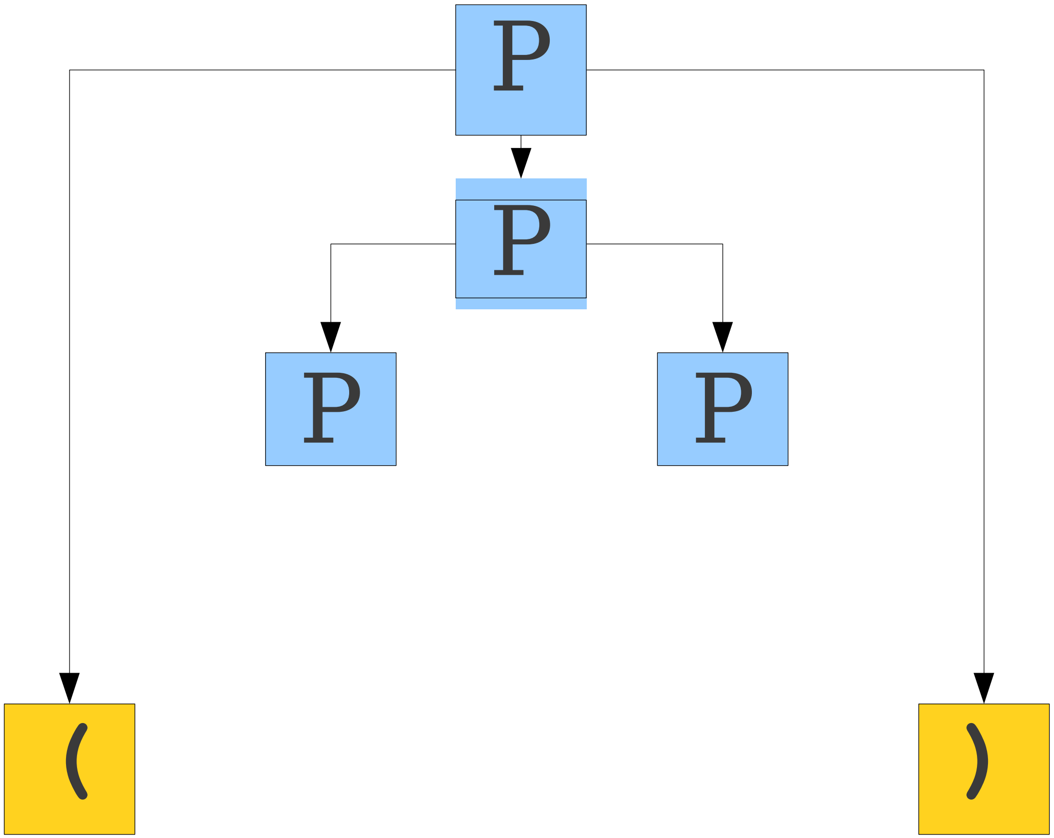
Balanced Parentheses

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



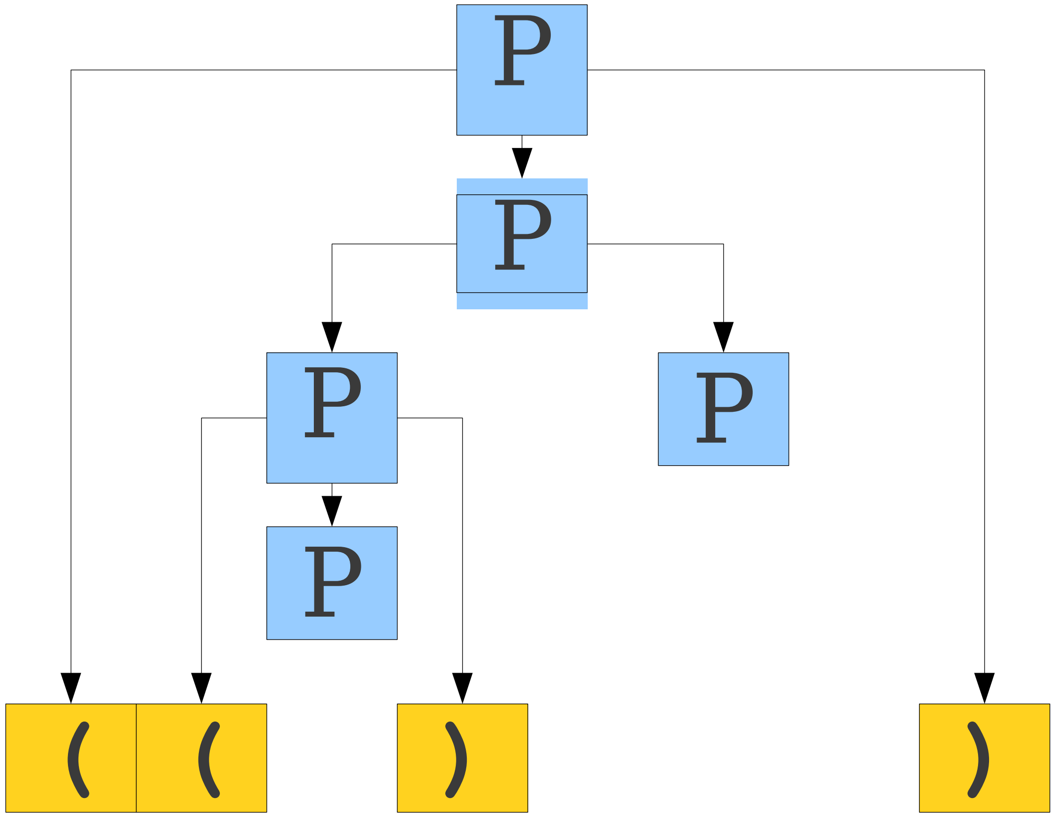
Balanced Parentheses

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



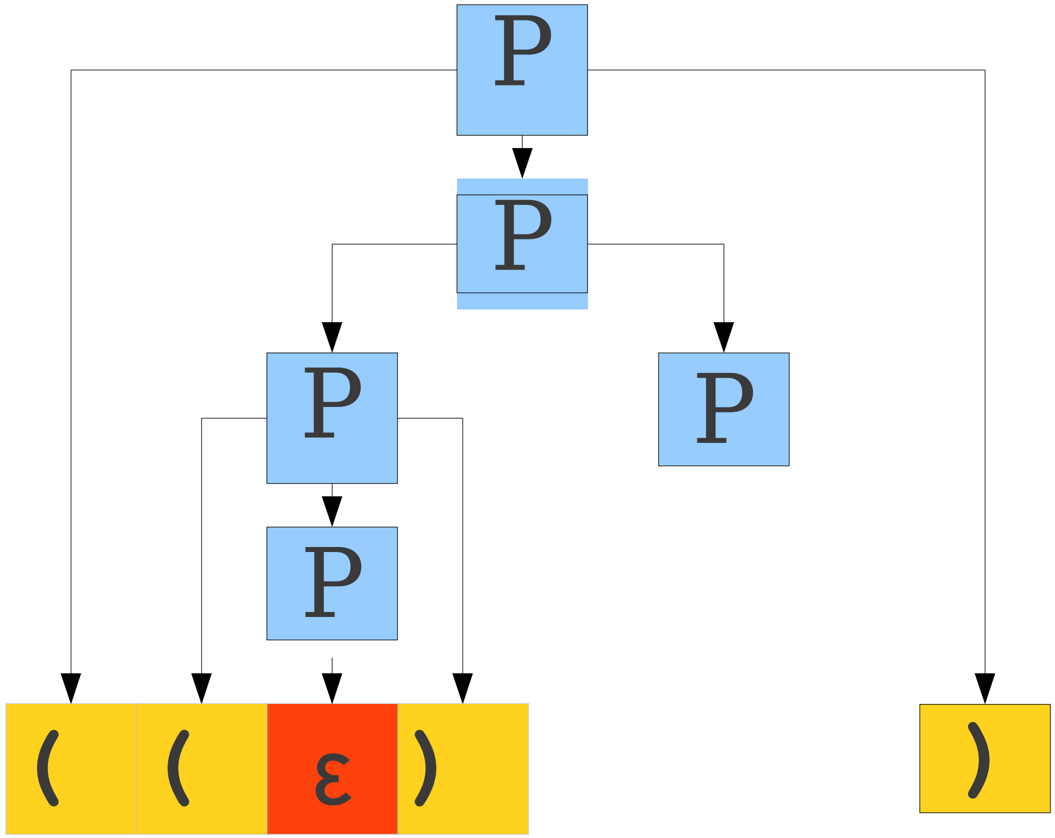
Balanced Parentheses

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



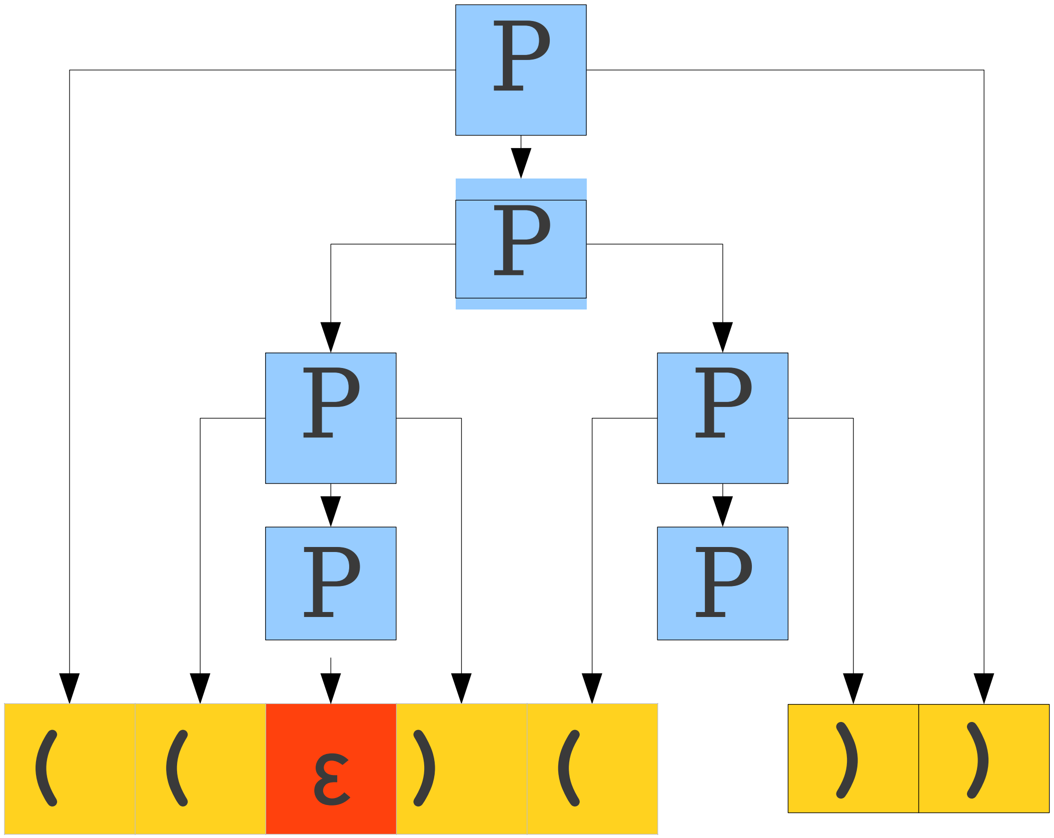
Balanced Parentheses

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



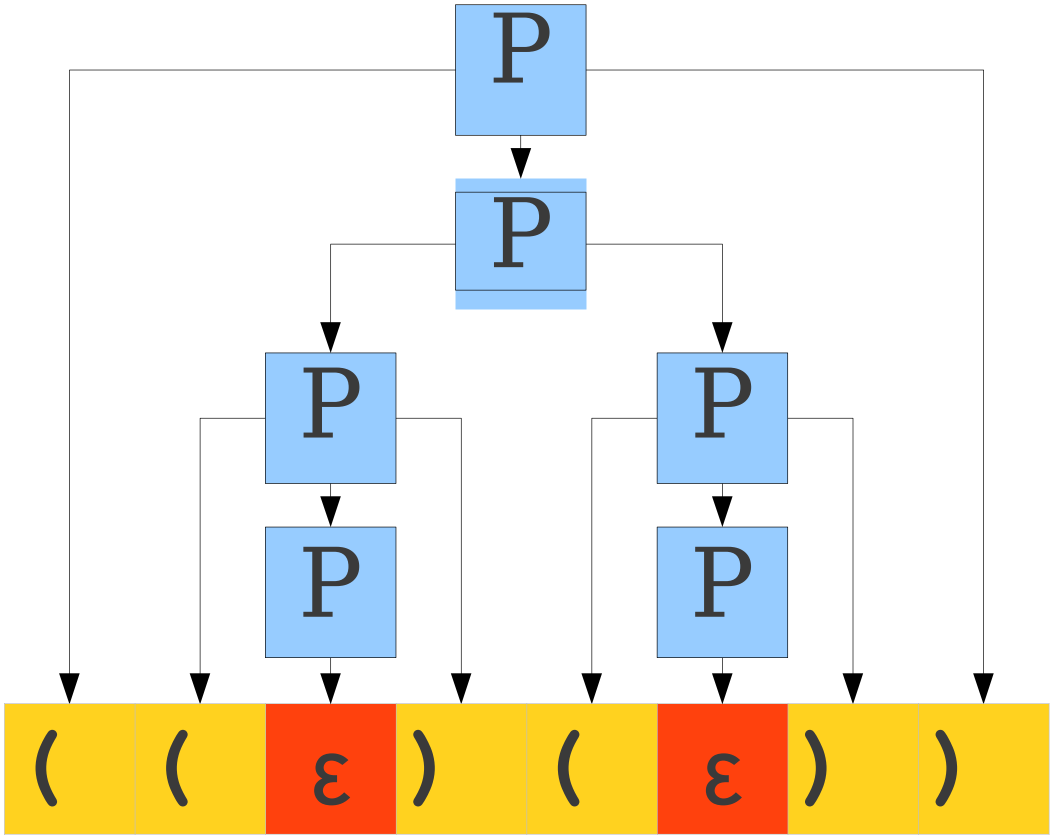
Balanced Parentheses

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

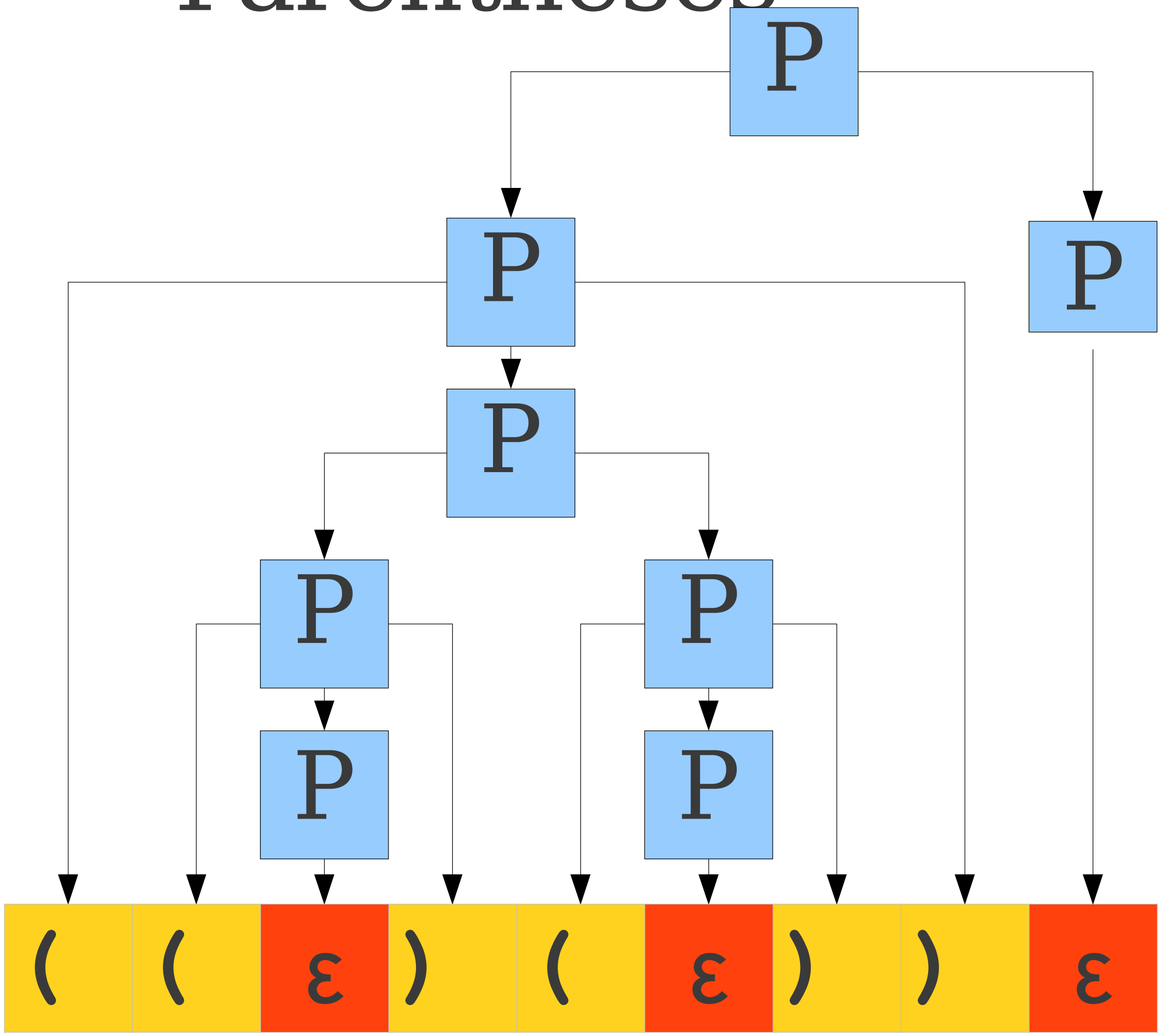


Balanced Parentheses

- 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.

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.

$$\mathbf{S} \rightarrow \mathbf{P}\ \mathbf{S} \mid \varepsilon$$

$$\mathbf{P} \rightarrow (\ \mathbf{S}\ )$$

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:

$\mathbf{R} \rightarrow$ `a` | `b` | `c` | …

$\mathbf{R} \rightarrow$ "`ε`"

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

$\mathbf{R} \rightarrow \mathbf{R}$ "`|`" $\mathbf{R}$

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

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

An Ambiguous Grammar

R → **a** | **b** | **c** | …

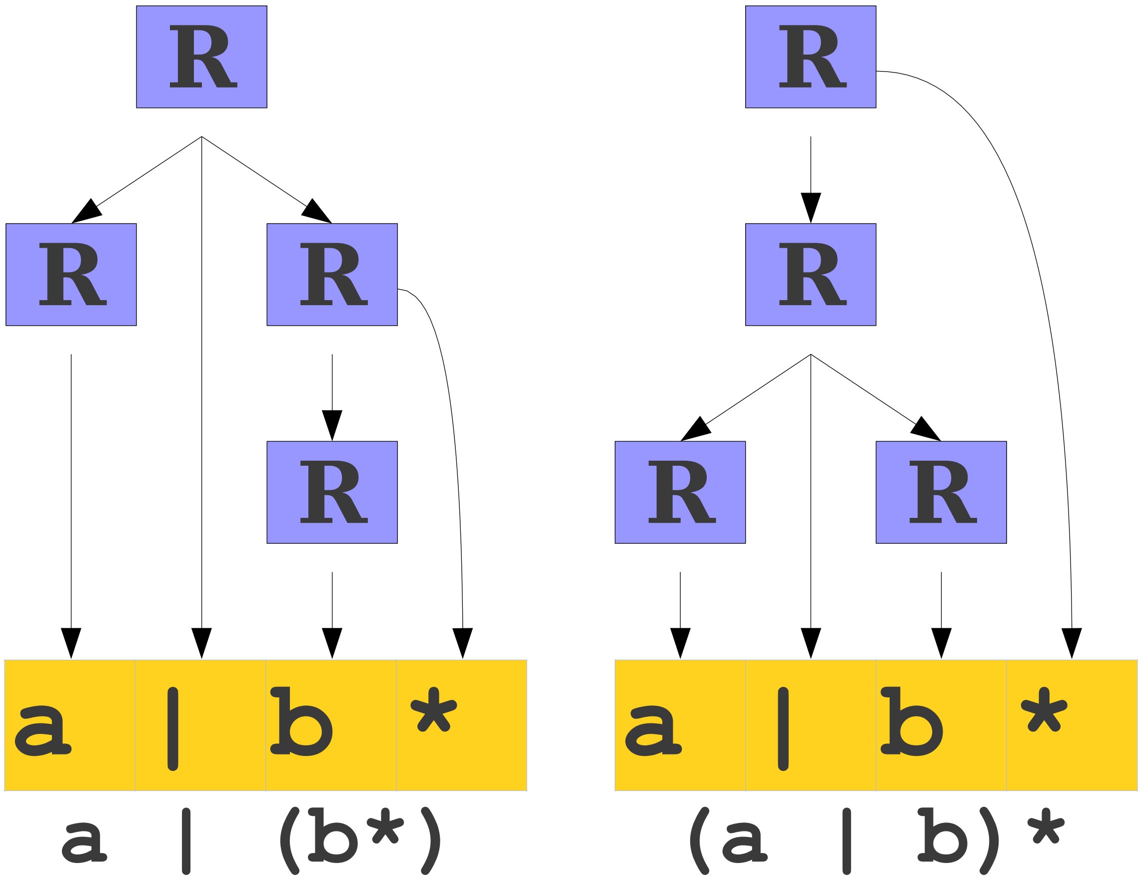
R → “**ε**”

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` | …

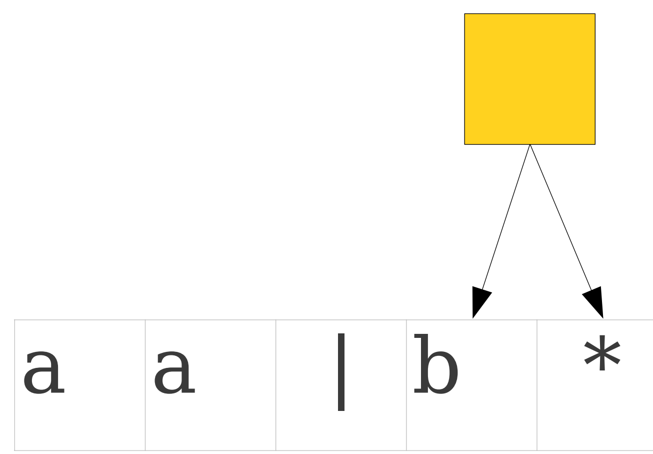
  R → “`ε`”

  R → **RR**

  R → **R** “`|`” **R**

  R → **R**`*`

  R → `(`**R**`)`



Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

  R → `a` | `b` | `c` | …

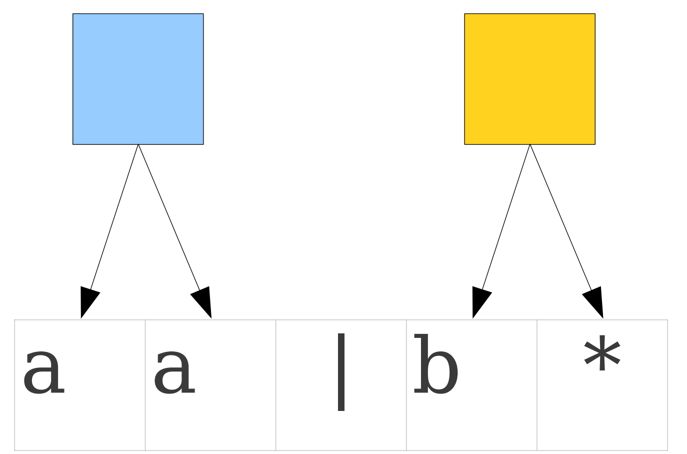
  R → "`ε`"

  R → **RR**

  R → **R** "`|`" **R**

  R → **R**`*`

  R → `(`**R**`)`



Resolving Ambiguity

- We can try to resolve the ambiguity via layering:

$\mathbf{R} \rightarrow$ `a` | `b` | `c` | …

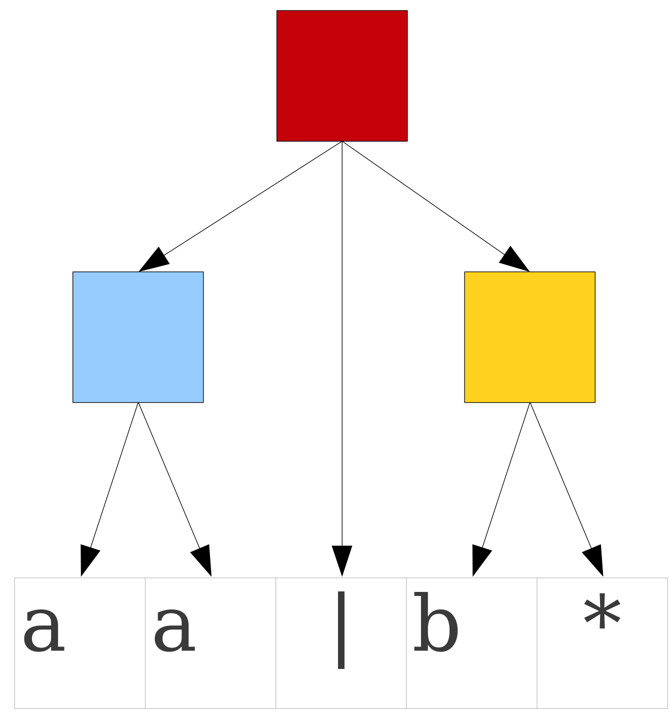
$\mathbf{R} \rightarrow$ "`ε`"

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

$\mathbf{R} \rightarrow \mathbf{R}$ "`|`" $\mathbf{R}$

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

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



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?

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

$$S \rightarrow T \mid ST$$

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

$$U \rightarrow a \mid b \mid \ldots$$

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

$$U \rightarrow (R)$$

Unions concatenated expressions

Concatenates starred expressions

Puts stars onto atomic expressions

Only generates “atomic” expressions

R

$R \rightarrow S \mid R$ "|" S

$S \rightarrow T \mid ST$

$T \rightarrow U \mid T$*

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

$U \rightarrow$ "ε"

$U \rightarrow$ (R)

a	b	|	c	|	a	*

R → **S** | **R** “`|`” **S**

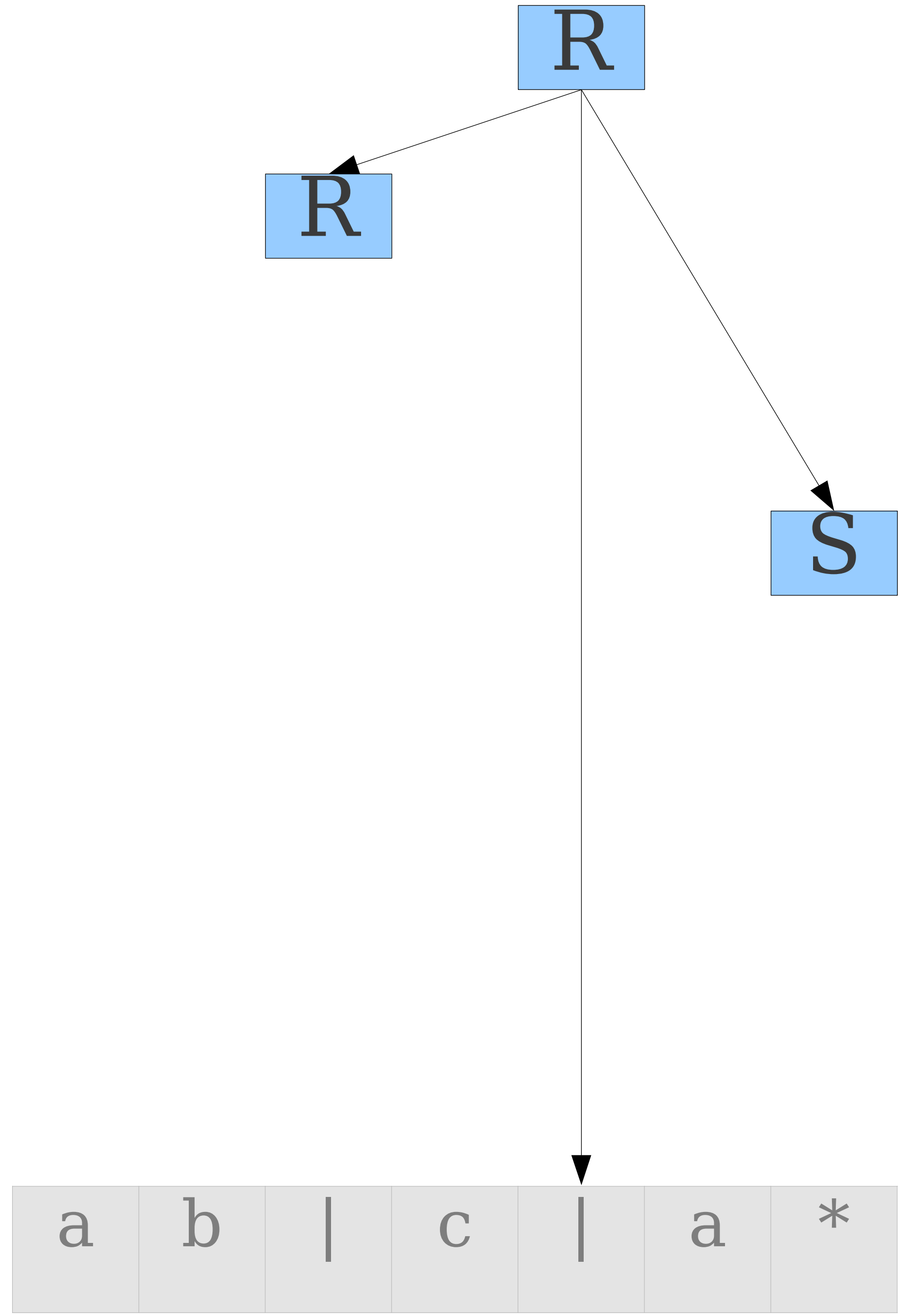
S → **T** | **ST**

T → **U** | **T**`*`

U → `a` | `b` | `c` | ...

U → “`ε`”

U → `(`**R**`)`



R → **S** | **R** "|" **S**

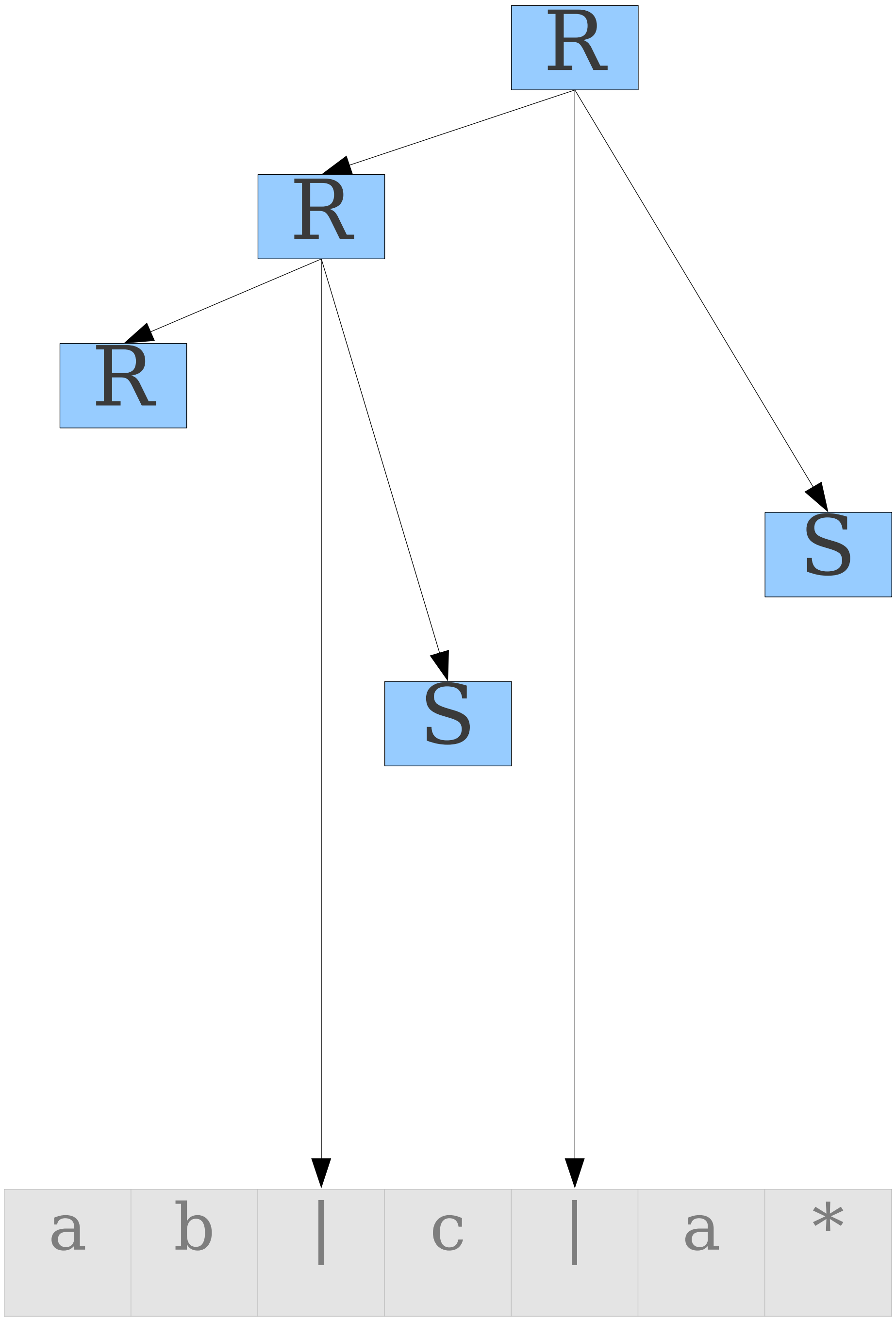
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → **S** | **R** "|" **S**

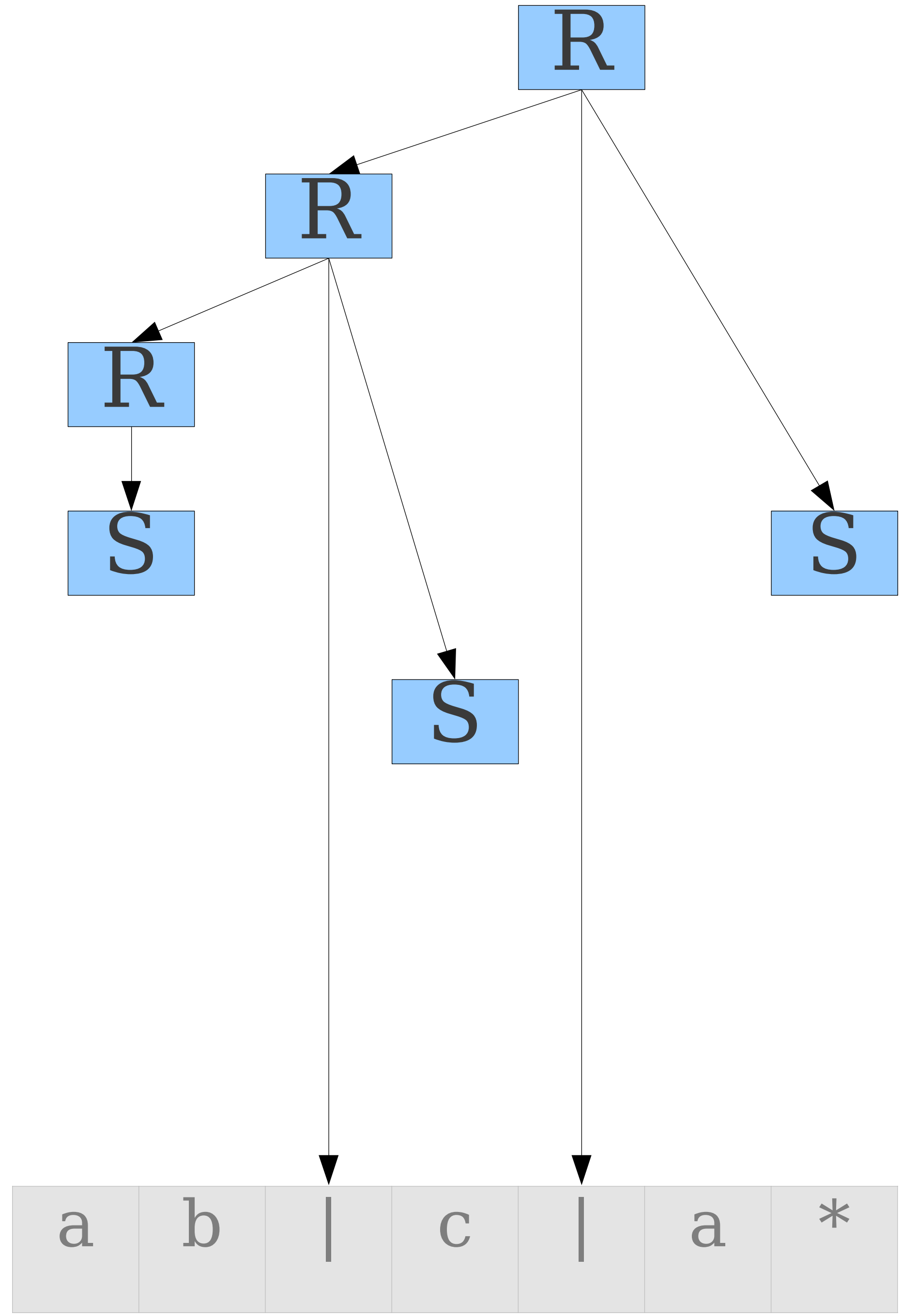
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → **S** | **R** “|” **S**

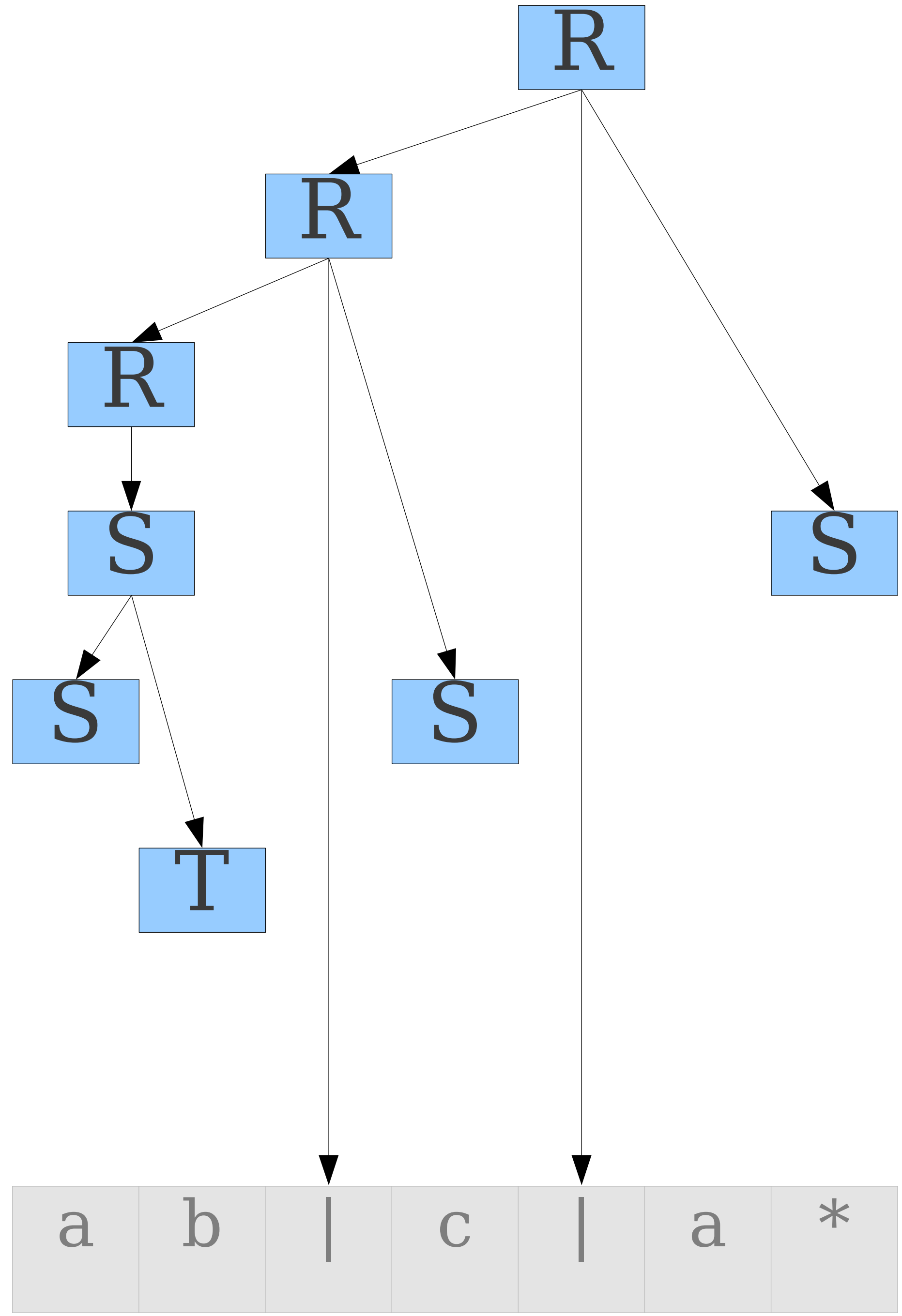
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | …

U → “ε”

U → (**R**)



R → **S** | **R** “|” **S**

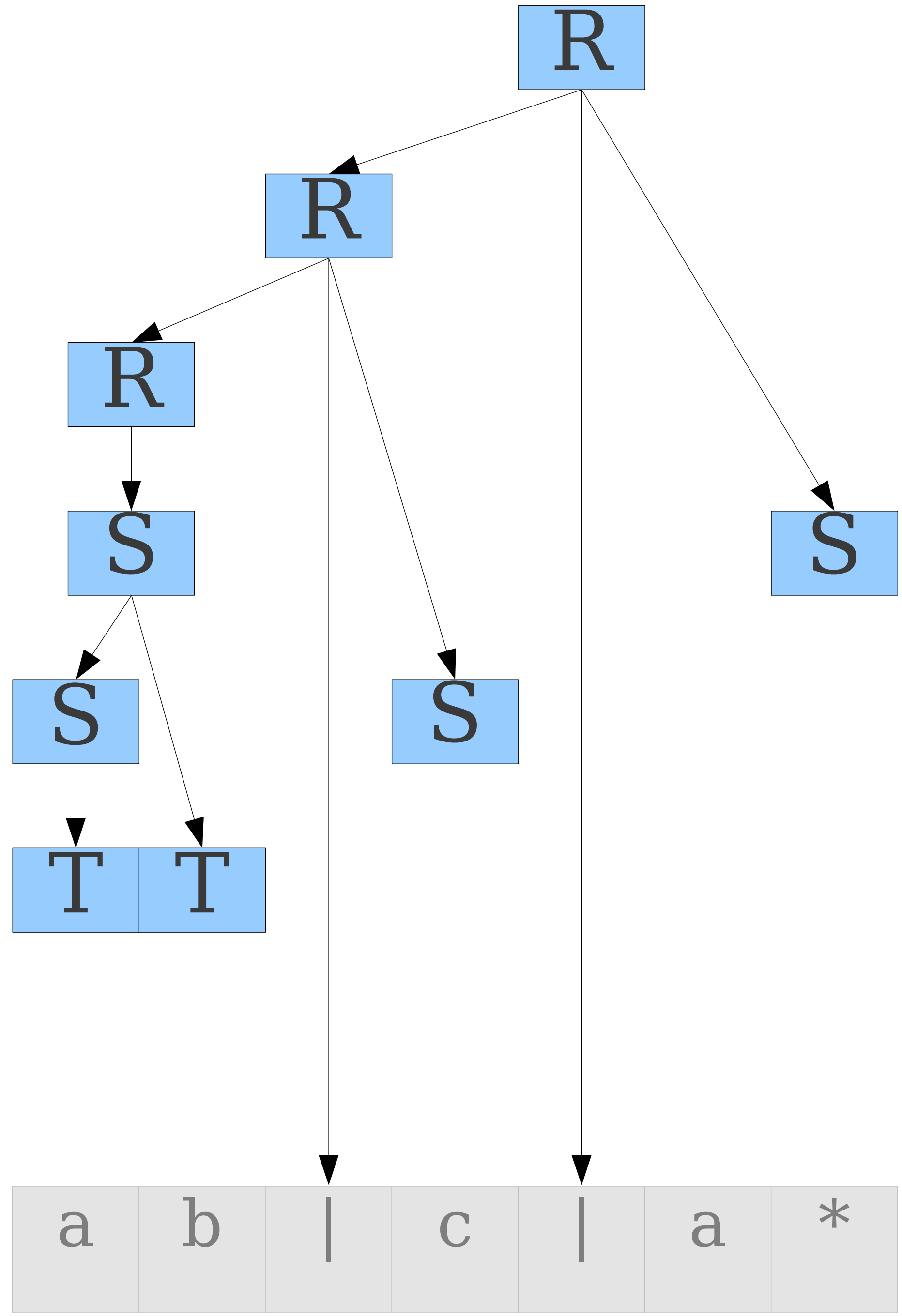
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → “ε”

U → (**R**)



R → S | R "|" S

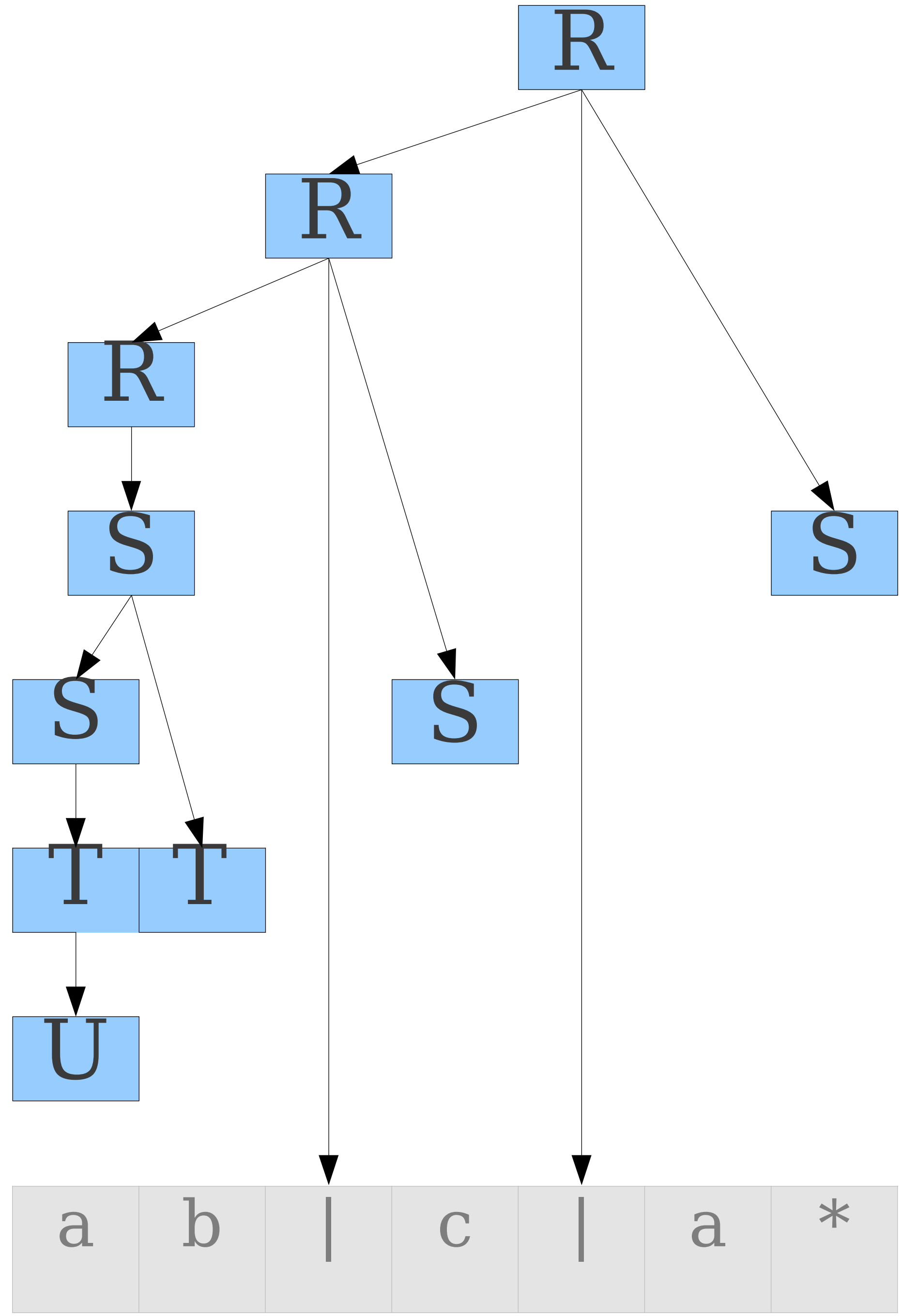
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



$R \rightarrow S \mid R$ "|" S

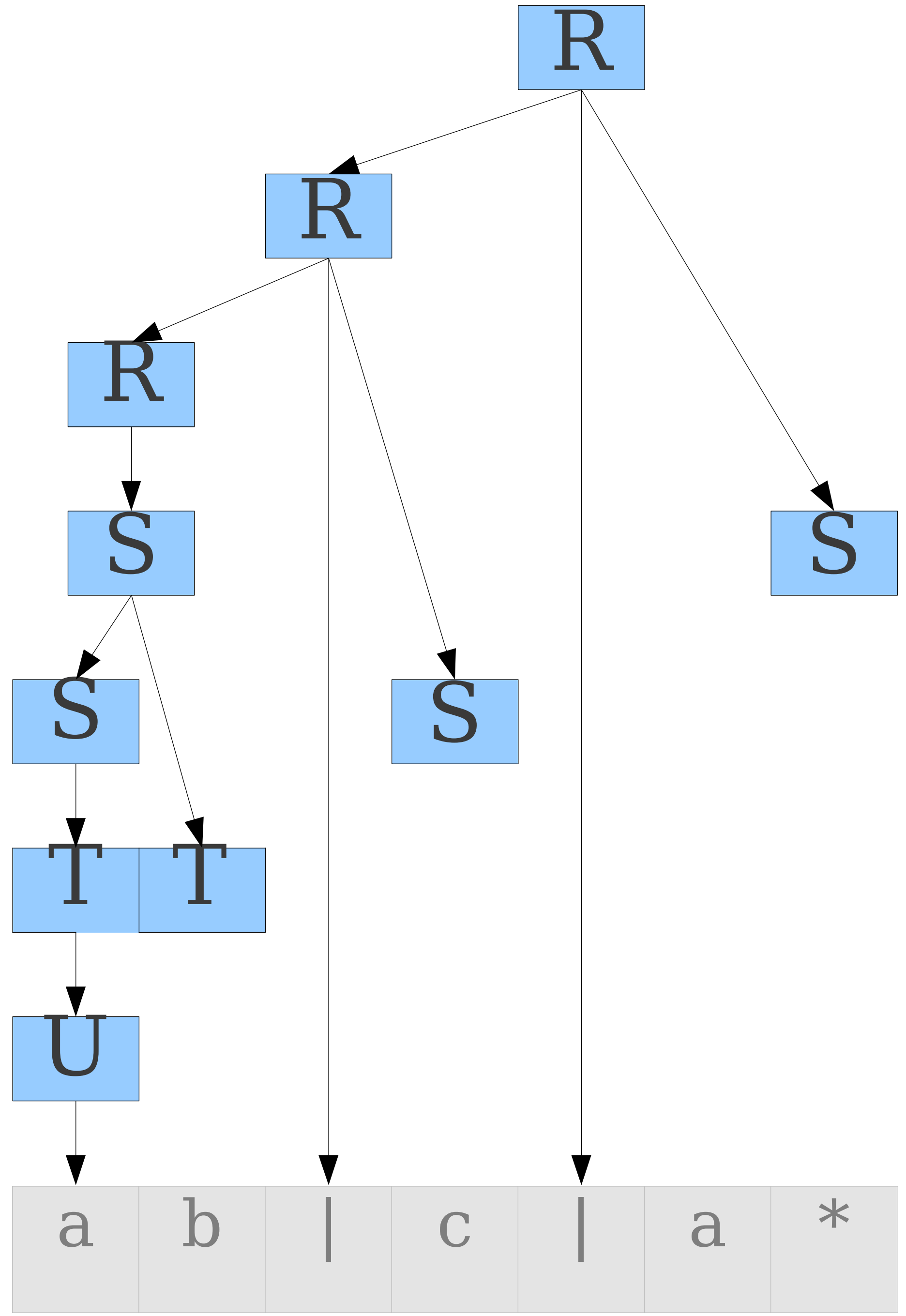
$S \rightarrow T \mid ST$

$T \rightarrow U \mid T$*

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

$U \rightarrow$ "ε"

$U \rightarrow$ (R)



R → **S** | **R** "|" **S**

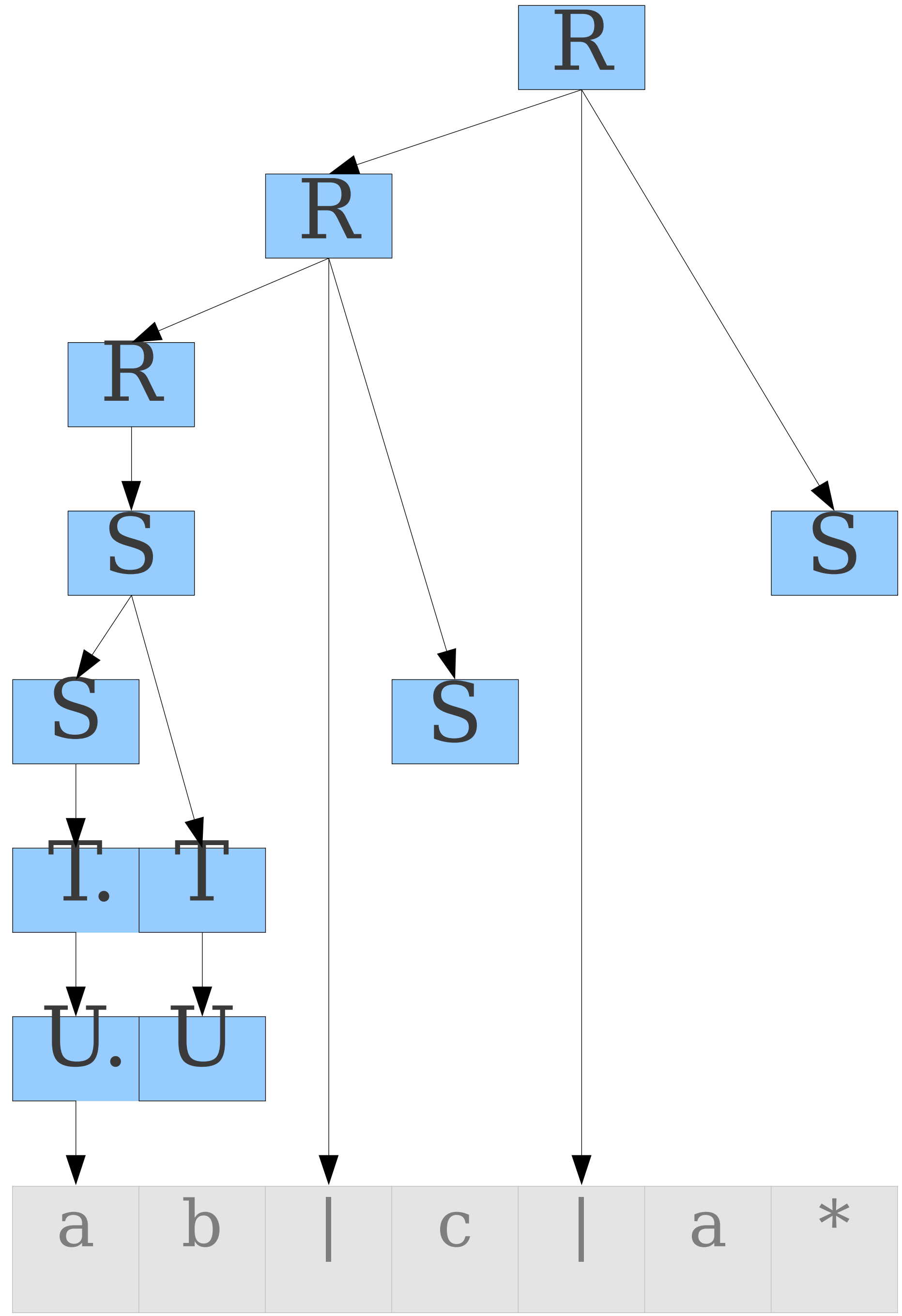
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → S | R "|" S

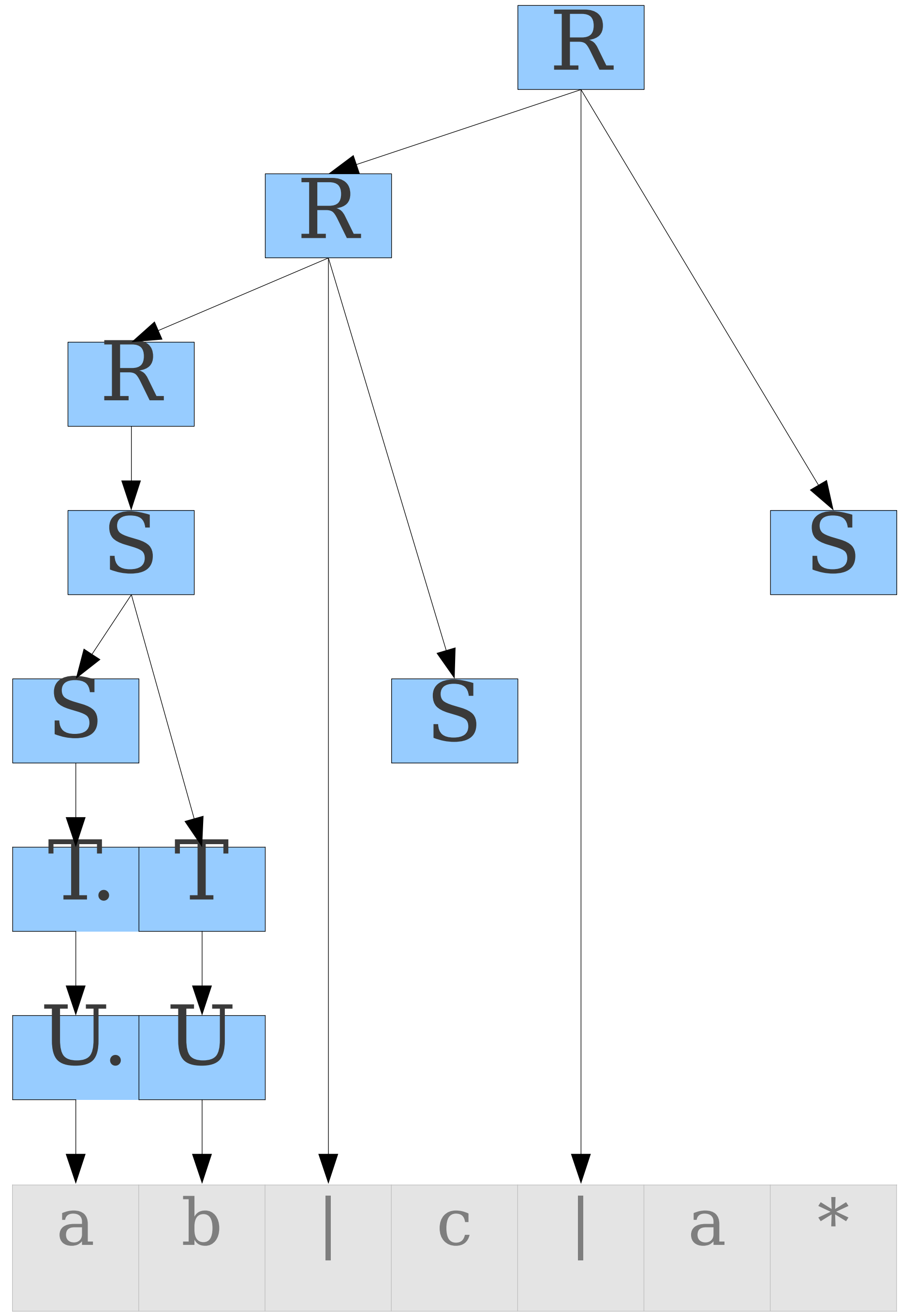
S → T | ST

T → U | T*

U → a | b | c | ...

U → "ε"

U → (R)



R → **S** | **R** “|” **S**

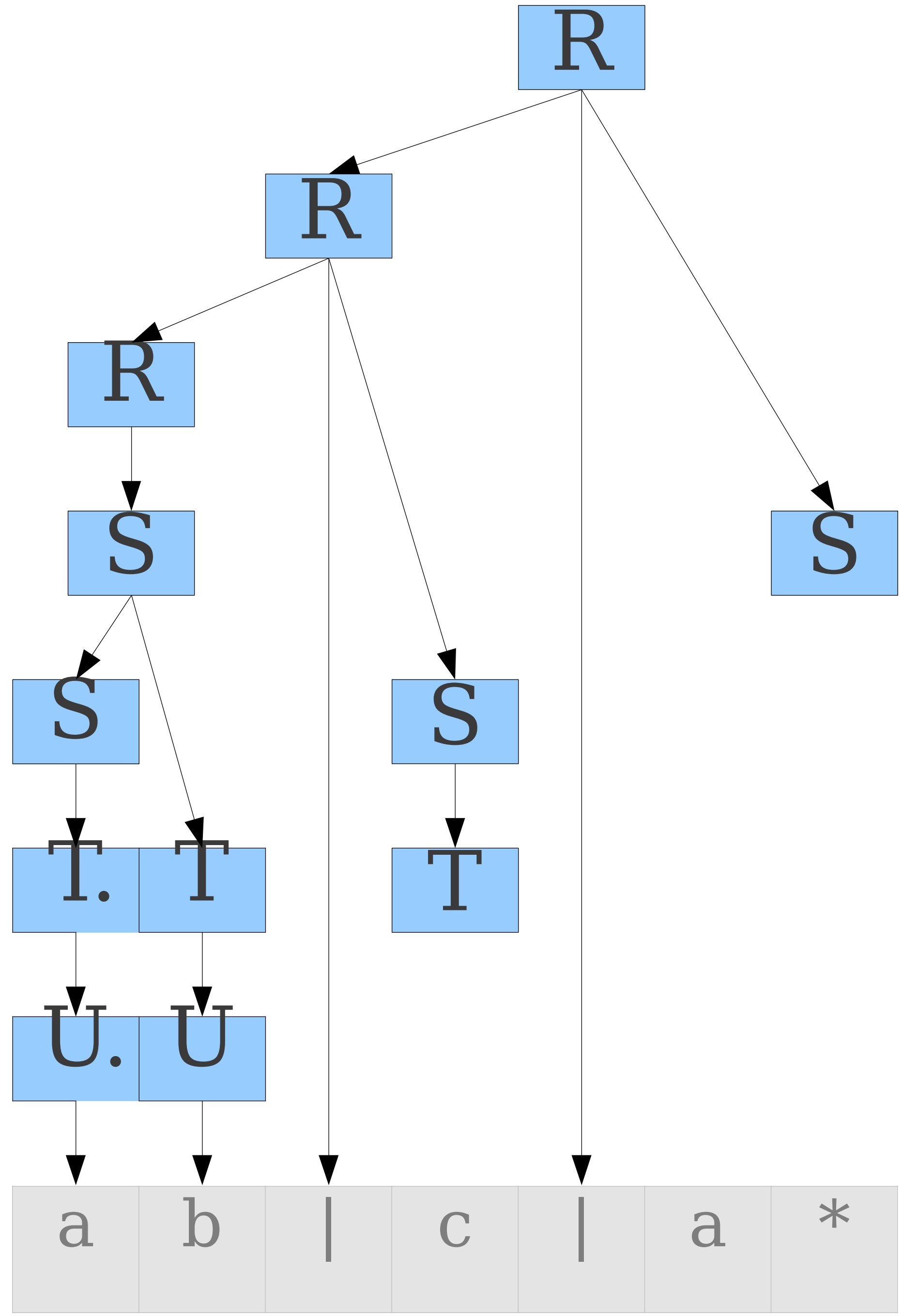
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → “ε”

U → (**R**)



R → **S** | **R** "|" **S**

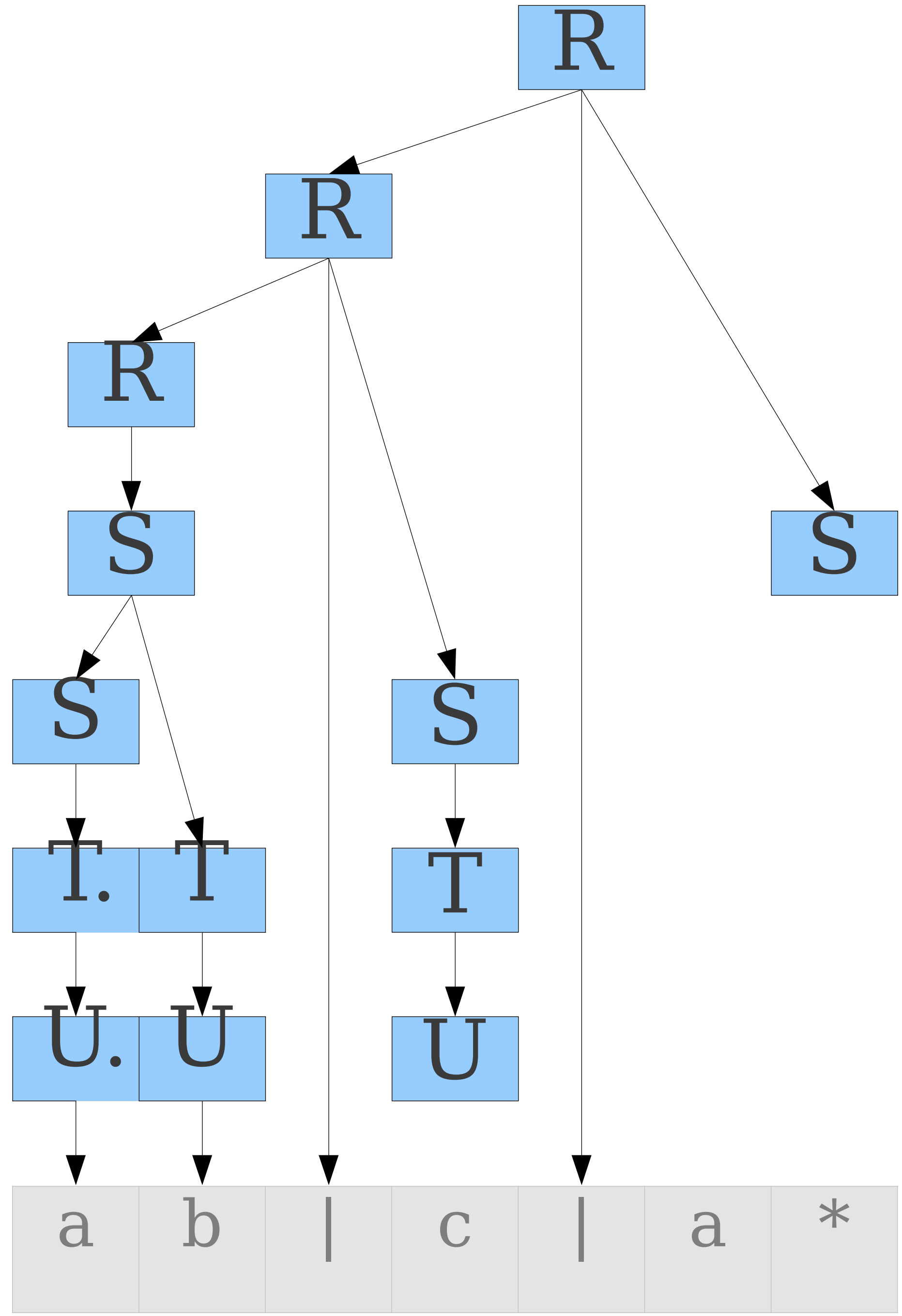
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → **S** | **R** "|" **S**

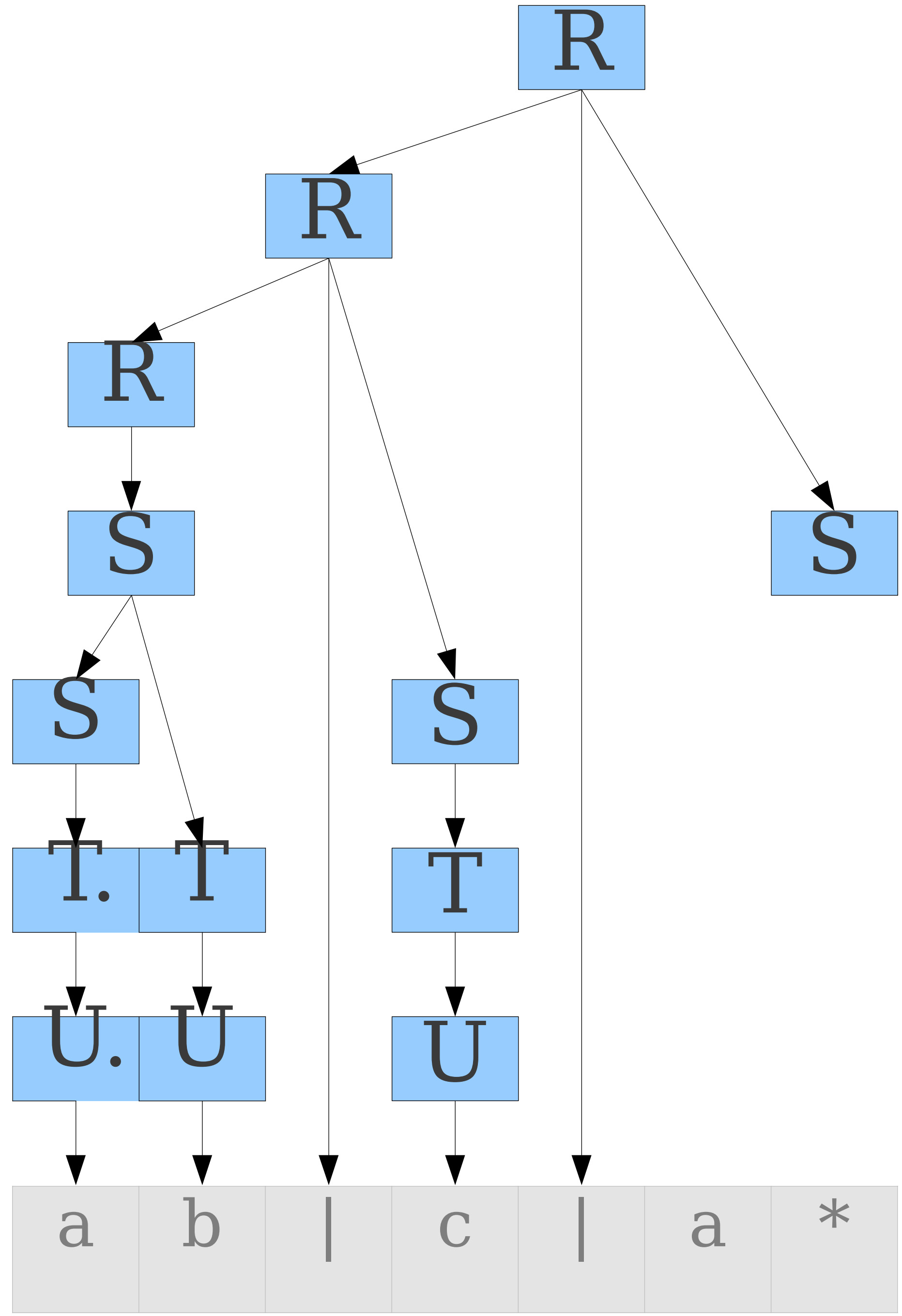
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → "ε"

U → (**R**)



R → **S** | **R** “|” **S**

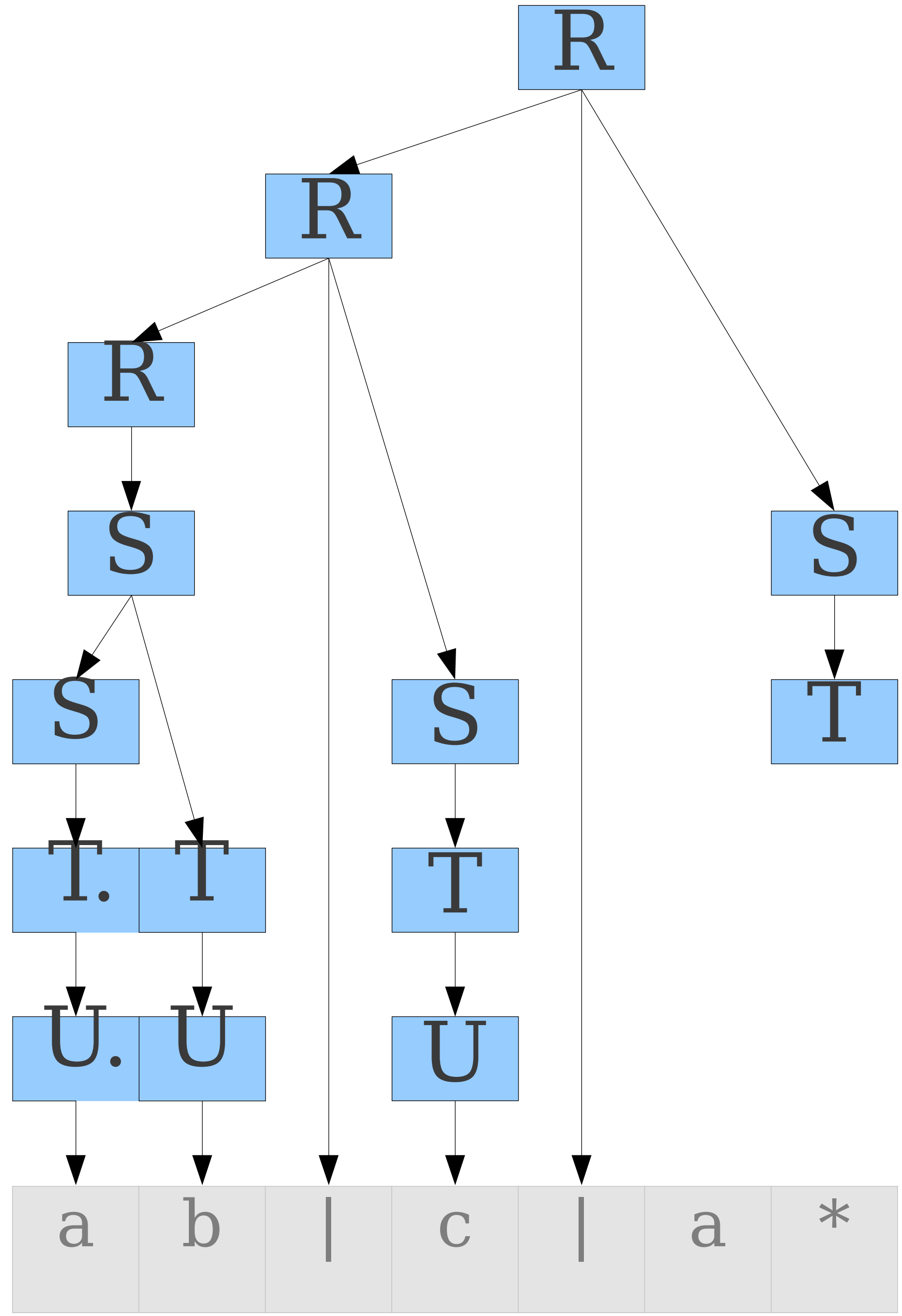
S → **T** | **ST**

T → **U** | **T***

U → a | b | c | ...

U → “ε”

U → (**R**)



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

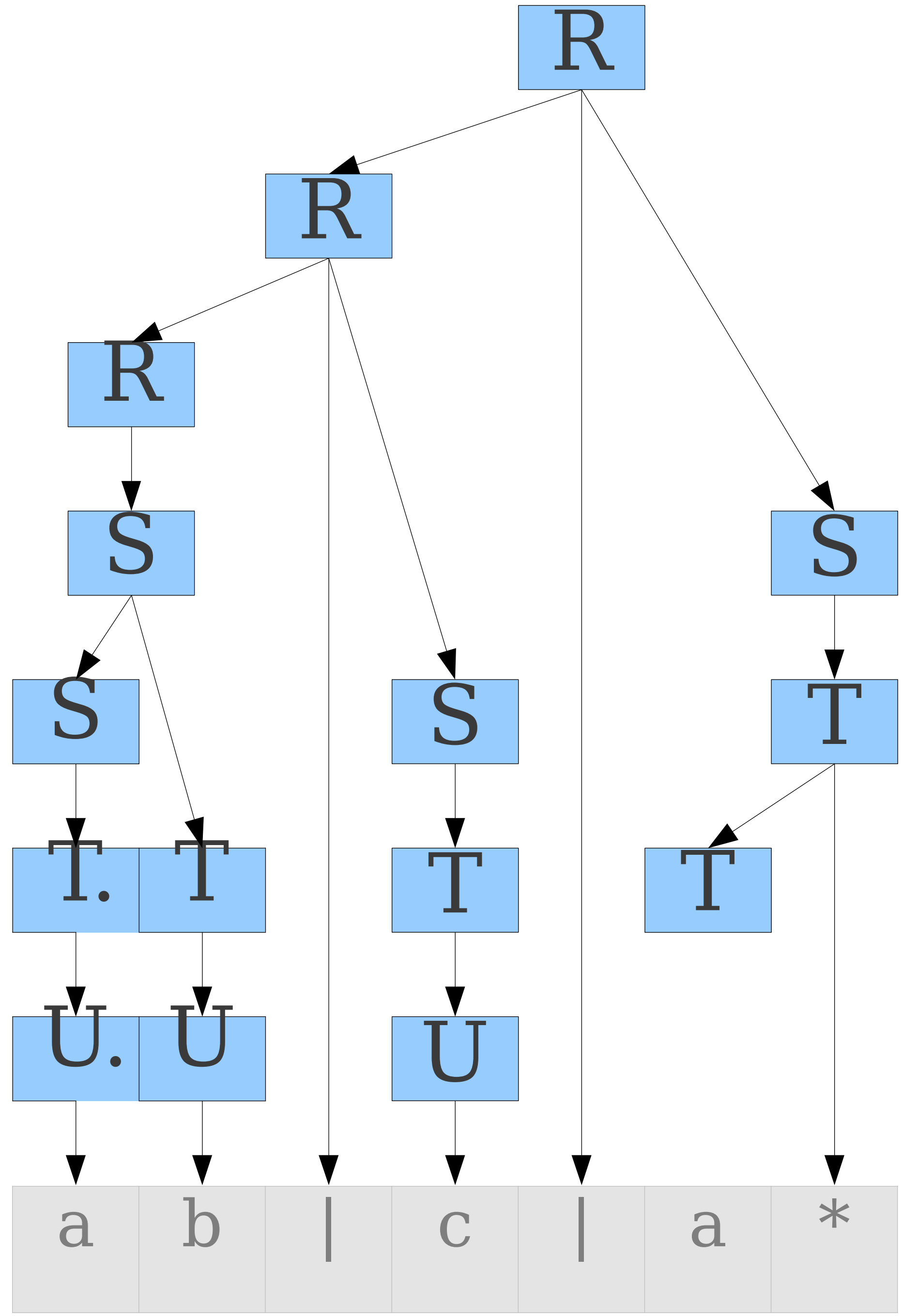
$\mathbf{S} \rightarrow \mathbf{T} \mid \mathbf{ST}$

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

$\mathbf{U} \rightarrow$ `a` | `b` | `c` | …

$\mathbf{U} \rightarrow$ “`ε`”

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



R → **S** | **R** "`|`" **S**

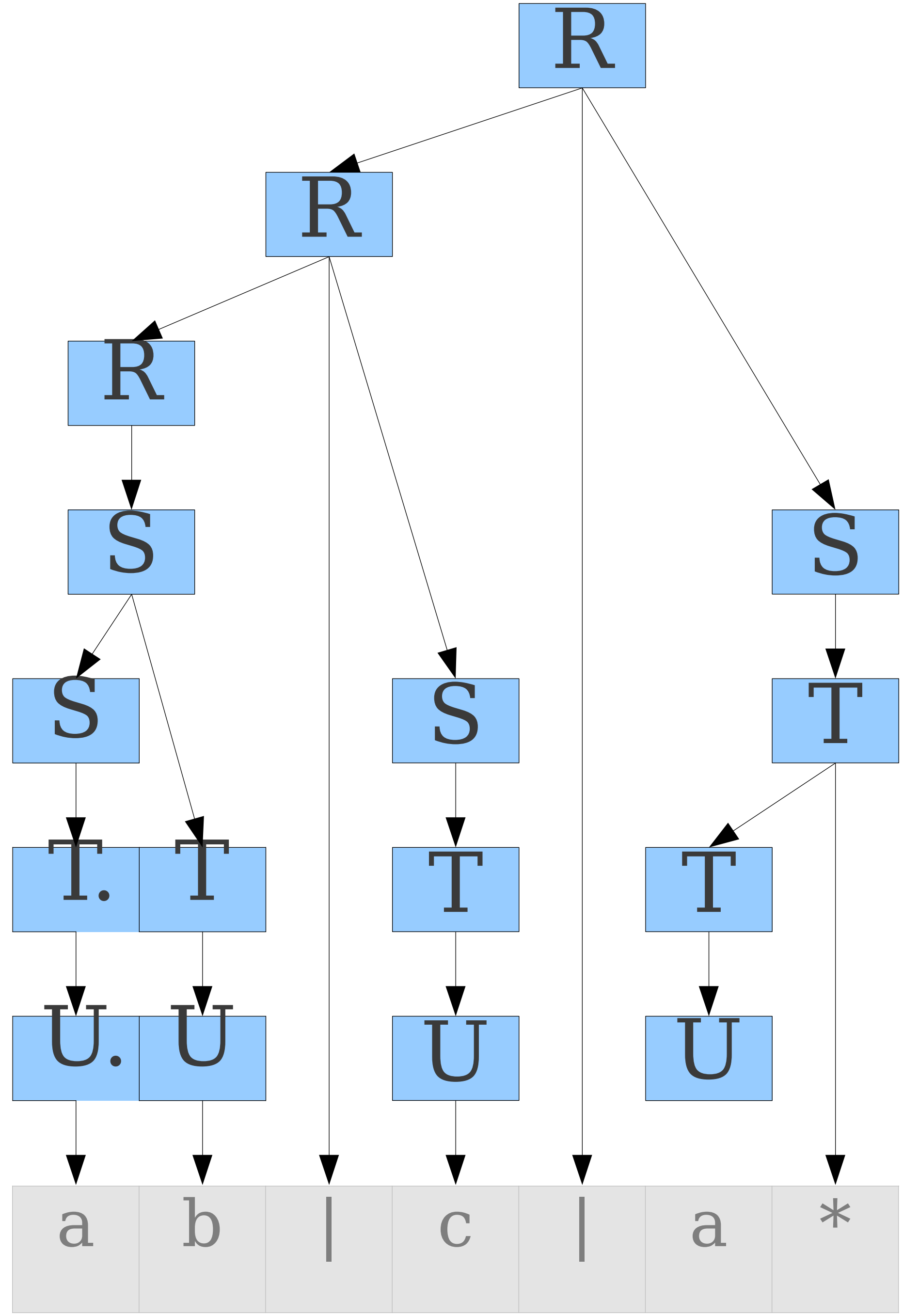
S → **T** | **ST**

T → **U** | **T**`*`

U → `a` | `b` | `c` | ...

U → "`ε`"

U → `(`**R**`)`



R → **S** | **R** "`|`" **S**

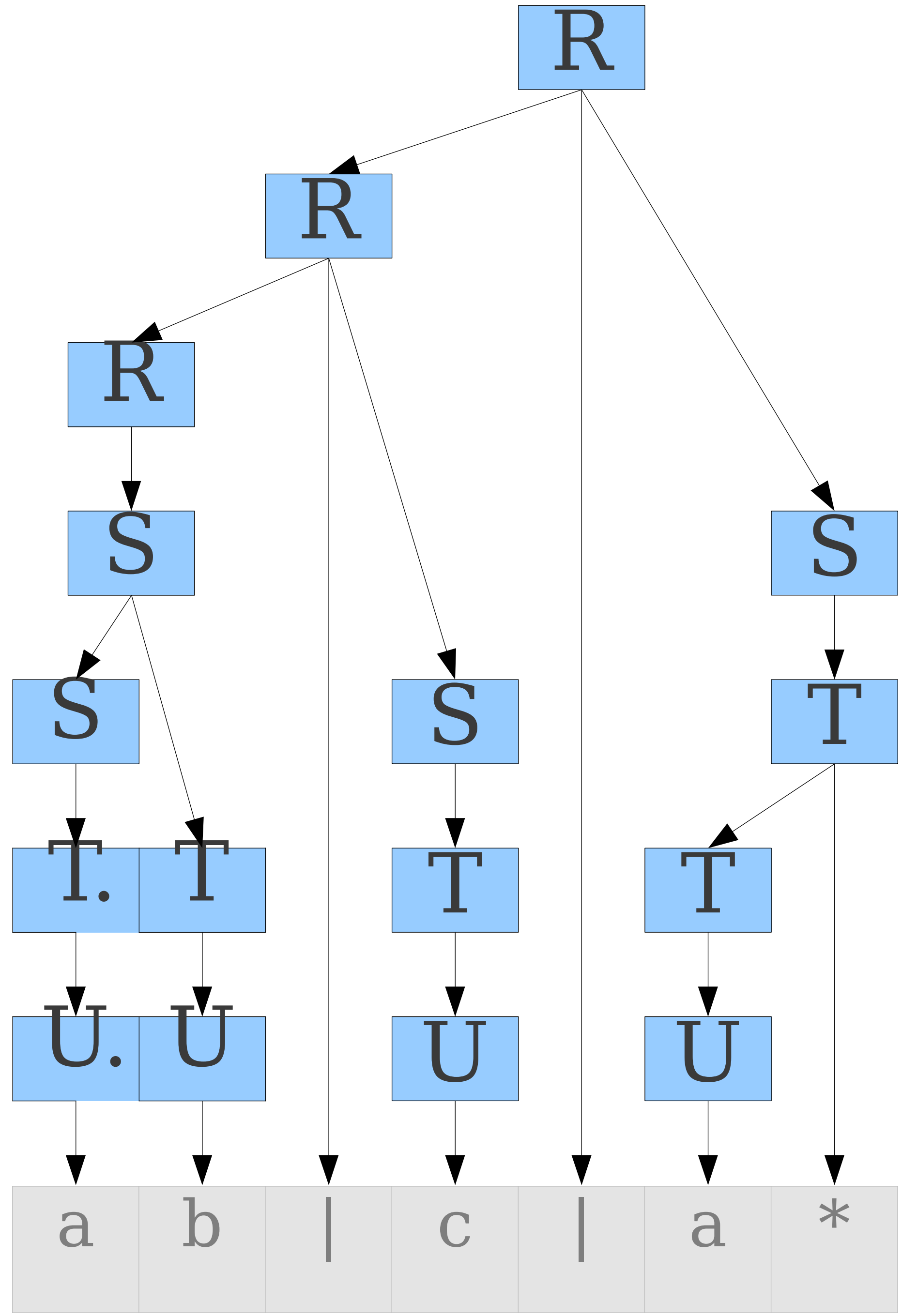
S → **T** | **ST**

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**

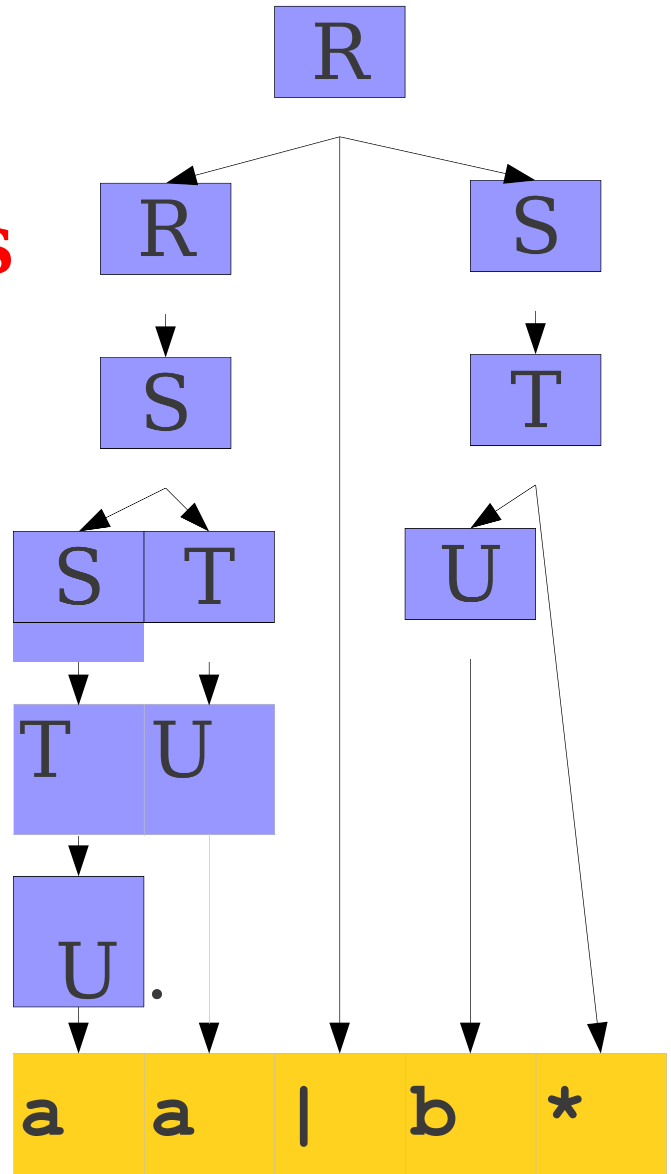
S → **T** | **ST**

T → **U** | **T***

U → a | b | …

U → "ε"

U → (**R**)



The Structure of a Parse Tree

R → **S** | **R** "|" **S**

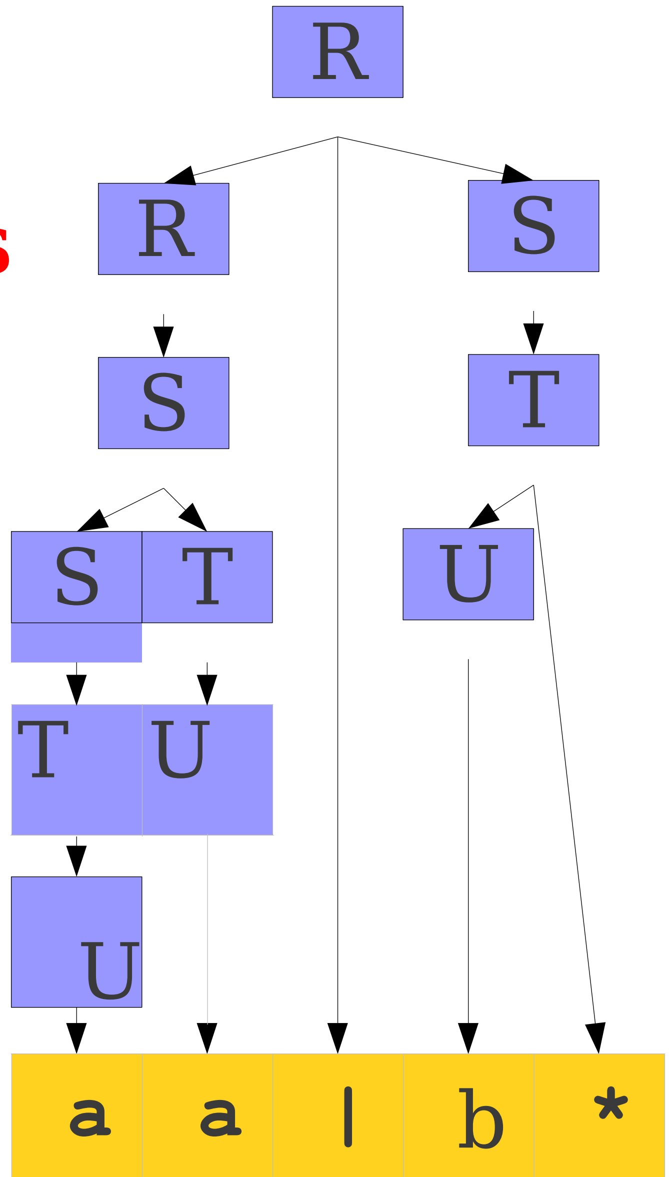
S → **T** | **ST**

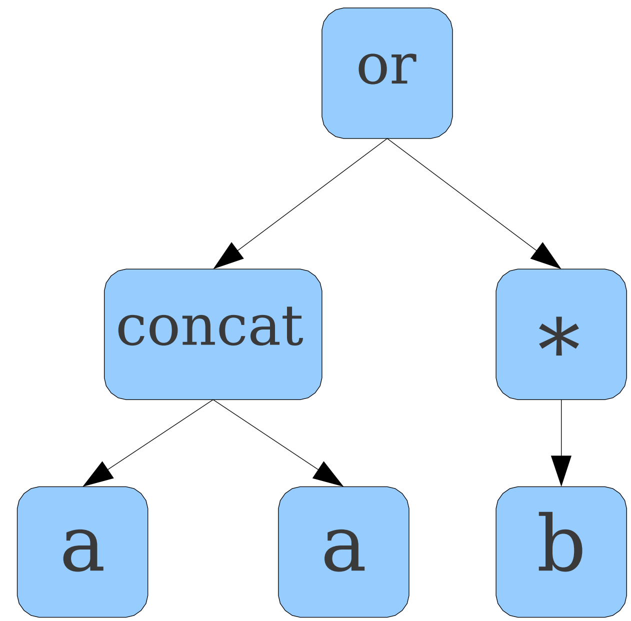
T → **U** | **T***

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

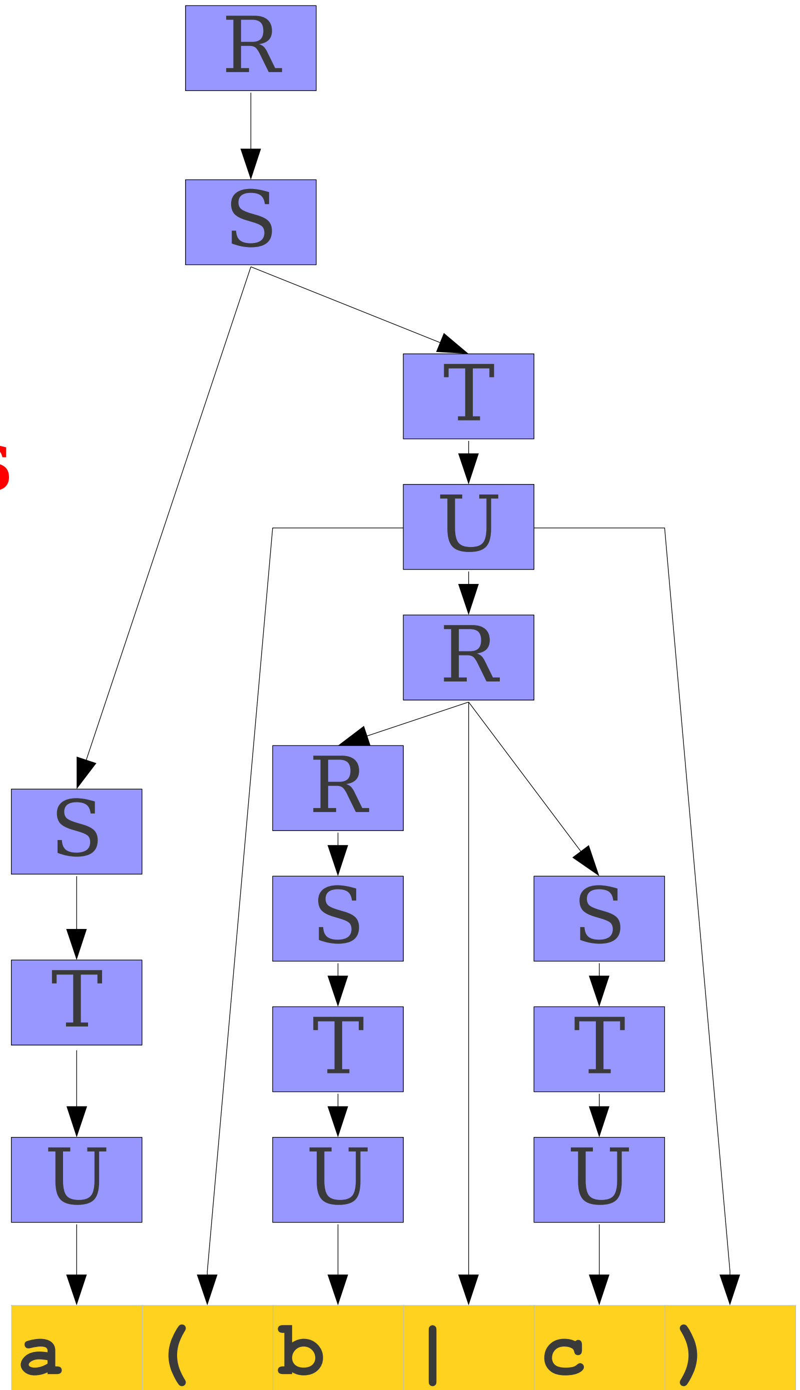
S → T | ST

T → U | T*

U → a | b | …

U → "ε"

U → (R)



R → S | R "|" S

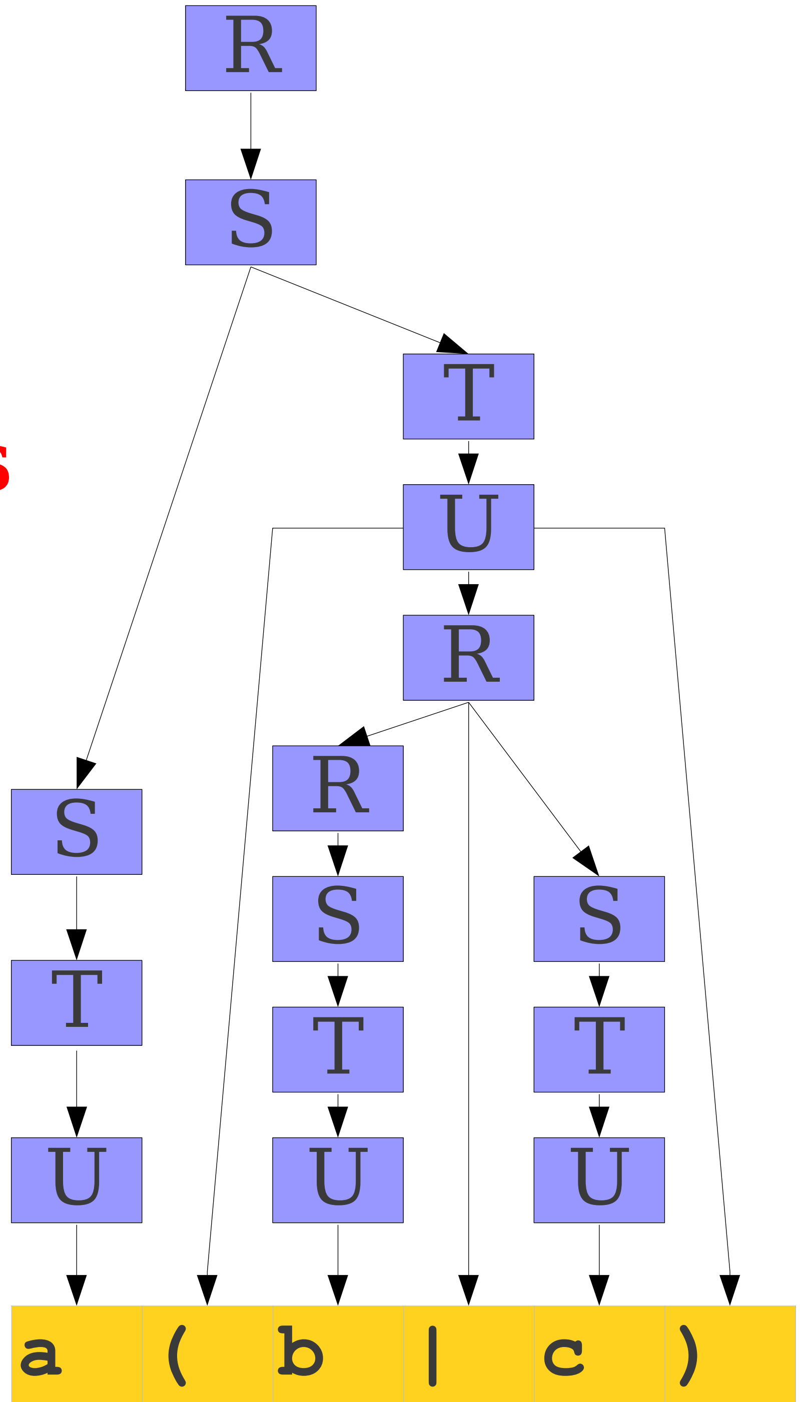
S → T | ST

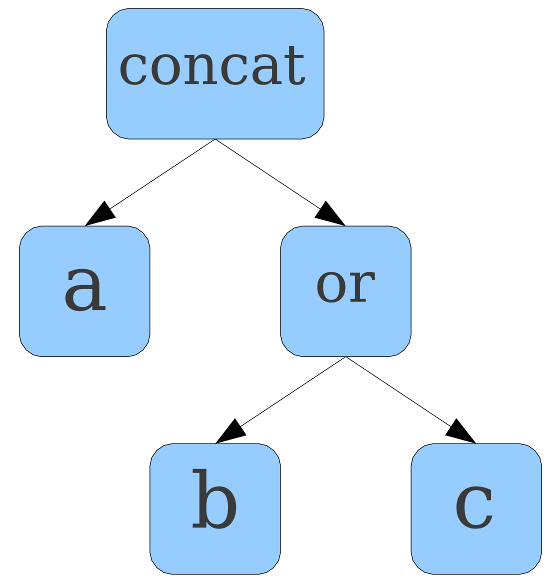
T → U | T*

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**.

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.

Challenges in Top-Down Parsing

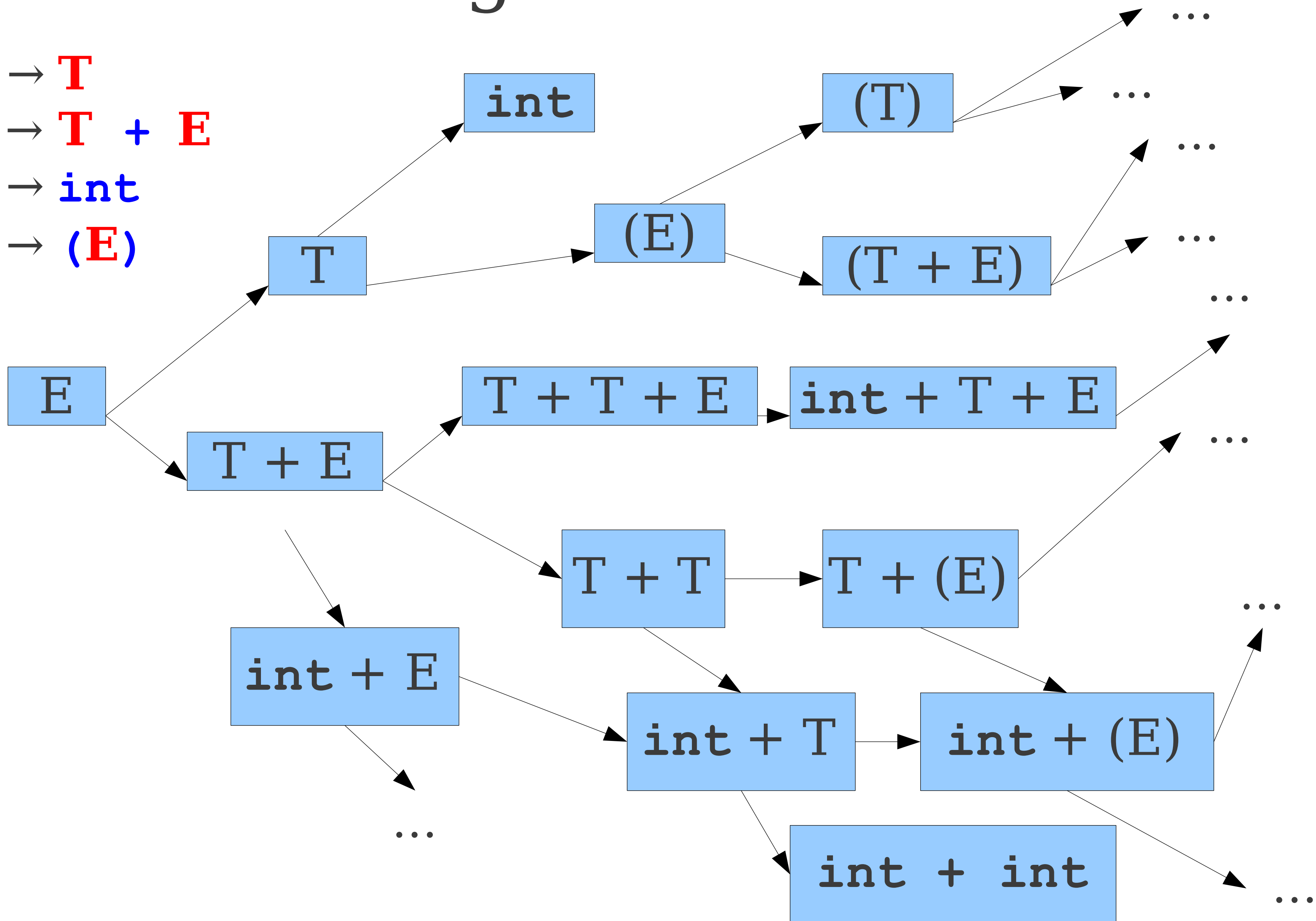
- Top-down parsing begins with virtually no information.
  - Begins with just the **start symbol**, which matches *every* program.
- How can we know which productions to apply?
- In general, we can't.
  - There are some grammars for which the best we can do is guess and backtrack if we're wrong.
  - If we have to guess, how do we do it?

Parsing as a Search

- An idea: **treat parsing as a graph search**.
- Each node is a **sentential form** (a string of terminals and nonterminals derivable from the start symbol).
- There is an edge from node $\boldsymbol{\alpha}$ to node $\boldsymbol{\beta}$ iff $\boldsymbol{\alpha} \Rightarrow \boldsymbol{\beta}$.

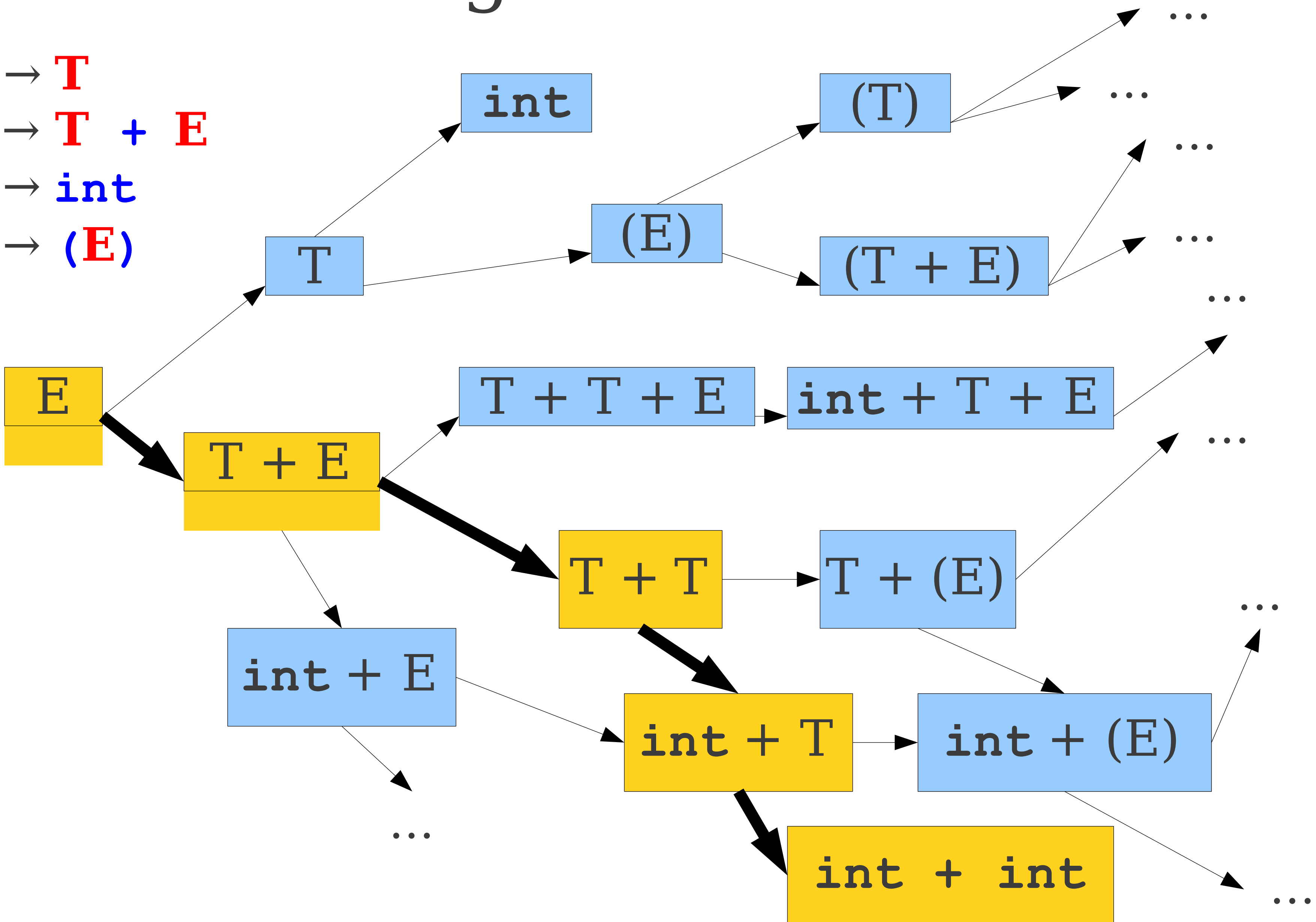
Parsing as a Search

E → **T**
E → **T + E**
T → **int**
T → **(E)**



Parsing as a Search

E → **T**
E → **T** **+** **E**
T → **int**
T → **(E)**



Our First Top-Down Algorithm

- **Breadth-First Search**
- Maintain a worklist of sentential forms, initially just the start symbol **S**.
- While the worklist isn't empty:
  - Remove an element from the worklist.
  - If it matches the target string, you're done.
  - Otherwise, for each possible string that can be derived in one step, add that string to the worklist.
- Can recover a parse tree by tracking what productions we applied at each step.

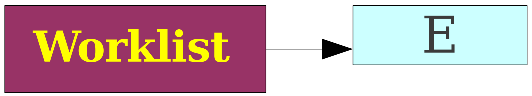
Breadth-First Search Parsing

Worklist

$E \rightarrow T$
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

`int + int`

Breadth-First Search Parsing



$\mathbf{E} \rightarrow \mathbf{T}$

$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

Breadth-First Search Parsing

Worklist

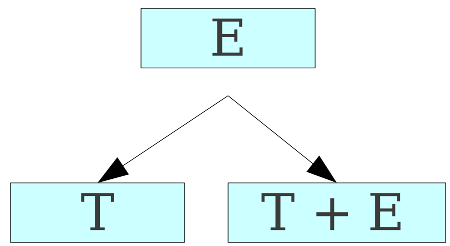
E

$E \rightarrow T$
$E \rightarrow T$ **+** E
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**

int + int

Breadth-First Search Parsing

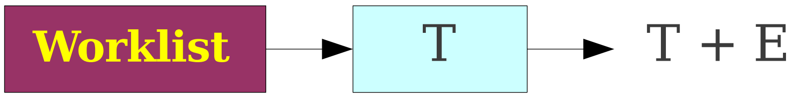
Worklist

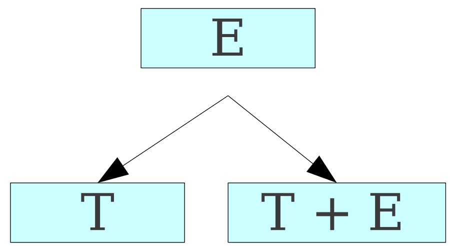


E → **T**
E → **T** **+** **E**
T → **int**
T → **(E)**

int + int

Breadth-First Search Parsing

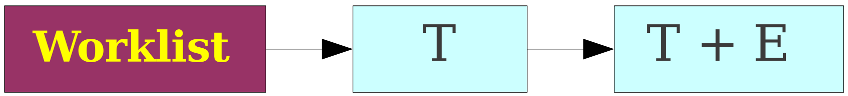




E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

`int + int`

Breadth-First Search Parsing



E → **T**

E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

Breadth-First Search Parsing

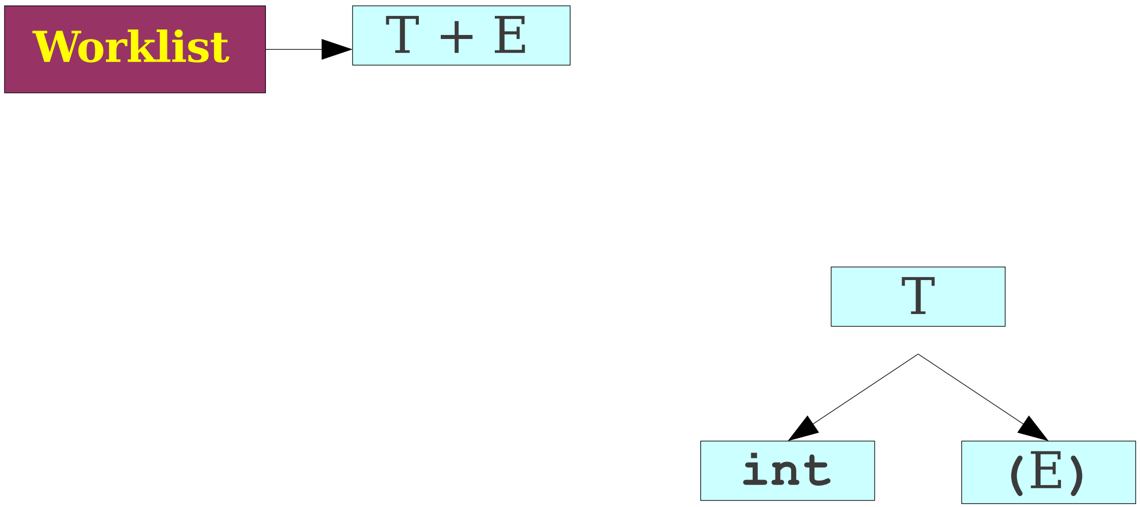
Worklist → T + E

T

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

`int + int`

Breadth-First Search Parsing



$E \rightarrow T$

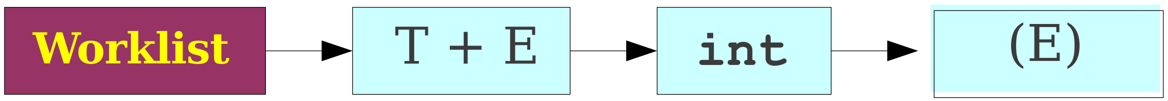
$E \rightarrow T$ **+** E

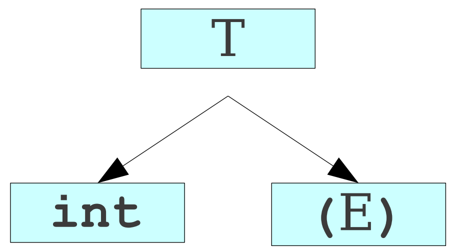
$T \rightarrow$ **int**

$T \rightarrow$ **(**E**)**

int + int

Breadth-First Search Parsing





E → T
E → T + E
T → int
T → (E)

int + int

Breadth-First Search Parsing

Worklist → T + E → `int` → (E)

E → **T**

E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

Breadth-First Search Parsing

Worklist → `int` → `(E)`

T + E

$\mathbf{E} \to \mathbf{T}$

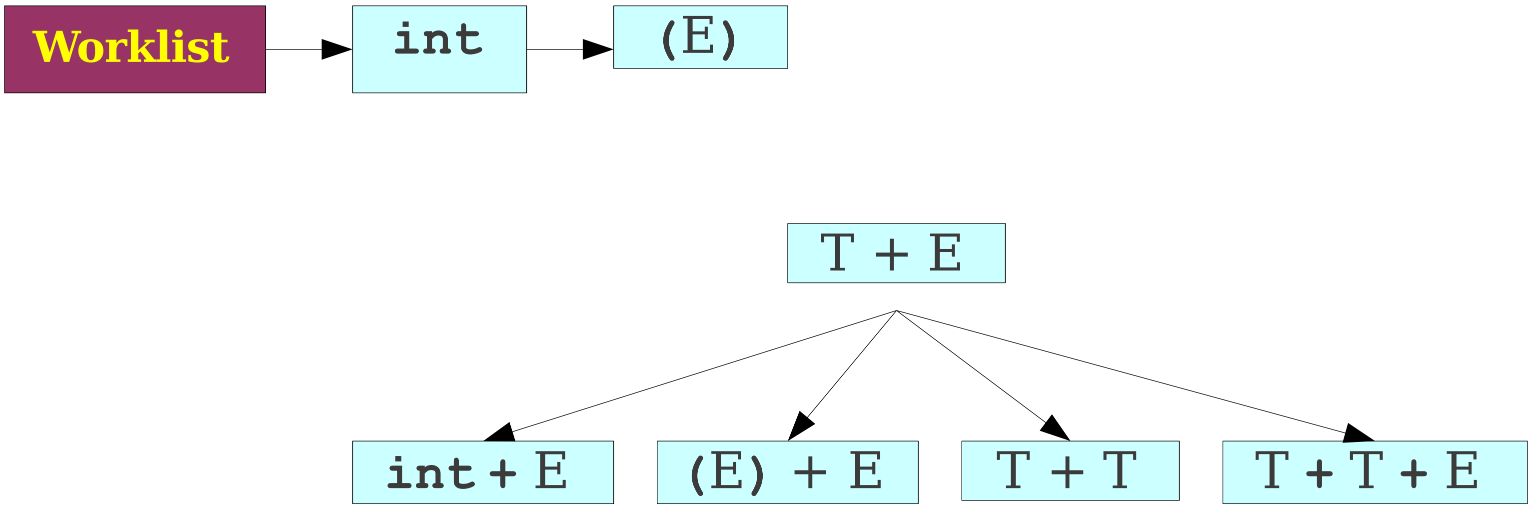
$\mathbf{E} \to \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \to$ `int`

$\mathbf{T} \to$ `(`$\mathbf{E}$`)`

`int + int`

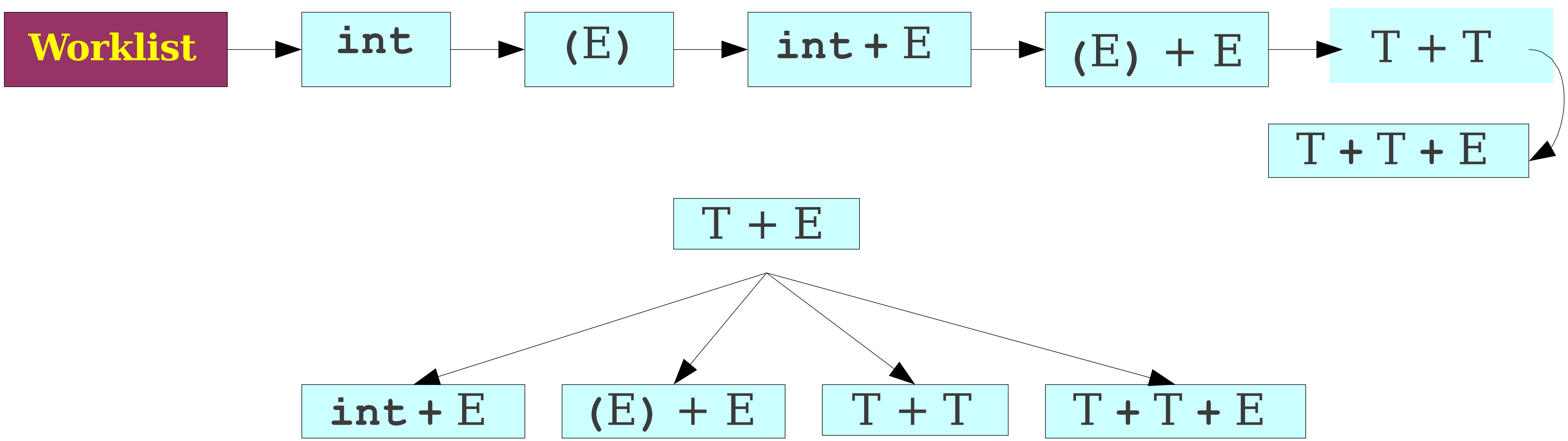
Breadth-First Search Parsing



$E \rightarrow T$
$E \rightarrow T$ **+** E
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**

int + int

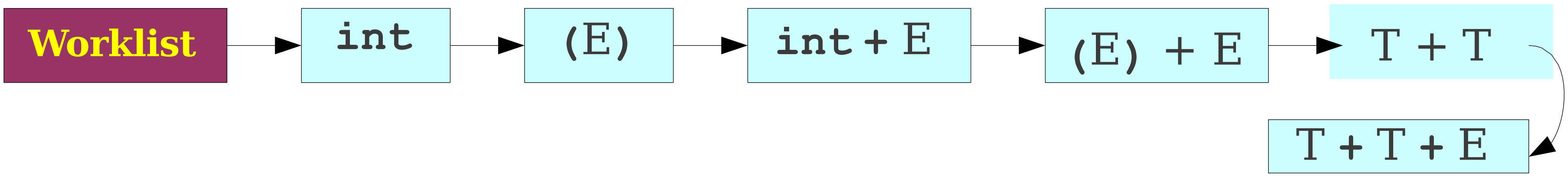
Breadth-First Search Parsing



E → **T**
E → **T** **+** **E**
T → **int**
T → **(E)**

int + int

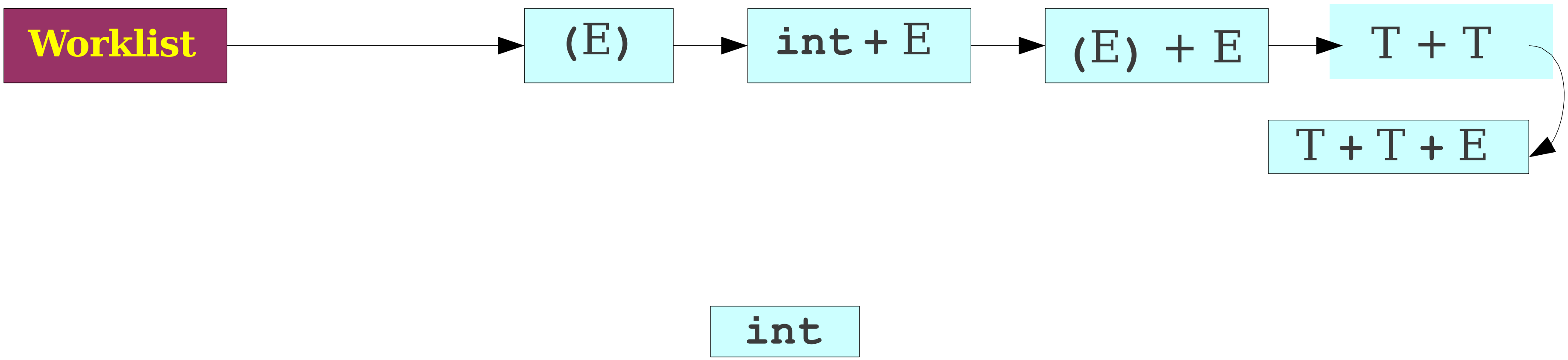
Breadth-First Search Parsing



E → **T**
E → **T** **+** **E**
T → **int**
T → **(E)**

`int + int`

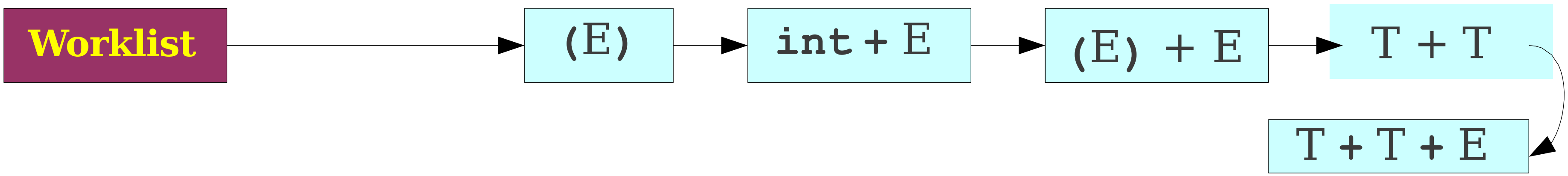
Breadth-First Search Parsing



$\mathbf{E} \to \mathbf{T}$
$\mathbf{E} \to \mathbf{T}$ **+** $\mathbf{E}$
$\mathbf{T} \to$ **int**
$\mathbf{T} \to$ **(E)**

int + int

Breadth-First Search Parsing



$\mathbf{E} \to \mathbf{T}$

$\mathbf{E} \to \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \to$ `int`

$\mathbf{T} \to$ `(`$\mathbf{E}$`)`

`int + int`

Breadth-First Search Parsing

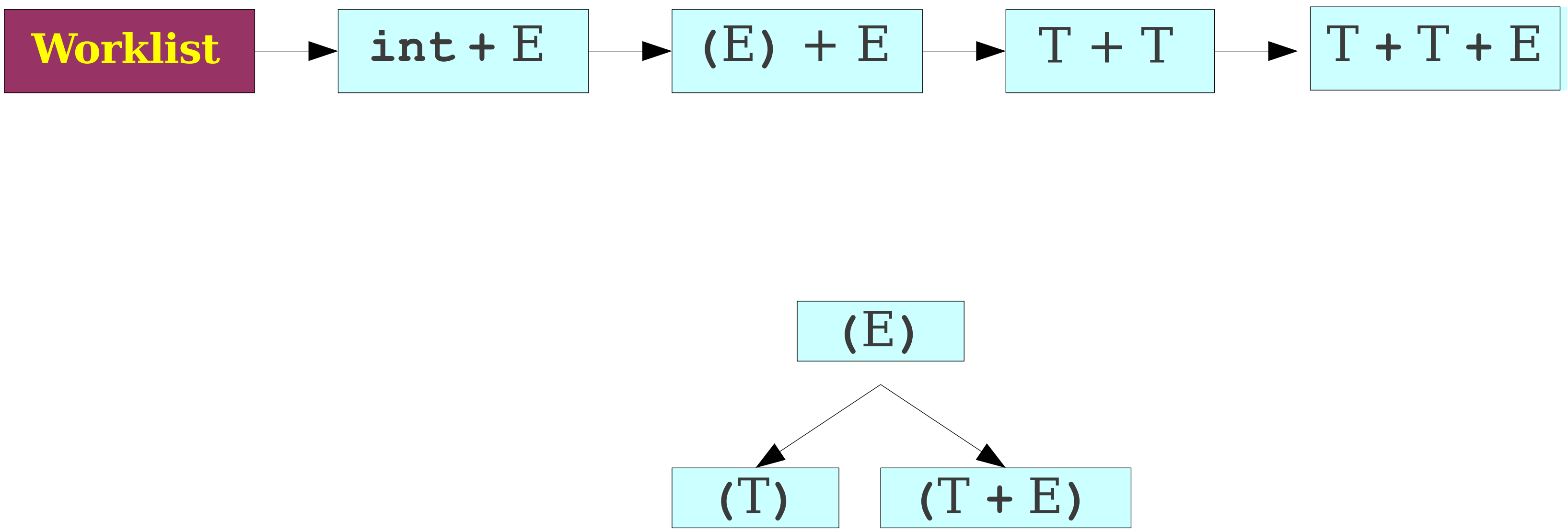
Worklist → int + E → (E) + E → T + T → T + T + E

(E)

$E \rightarrow T$
$E \rightarrow T$ **+** E
$T \rightarrow$ **int**
$T \rightarrow$ **(E)**

int + int

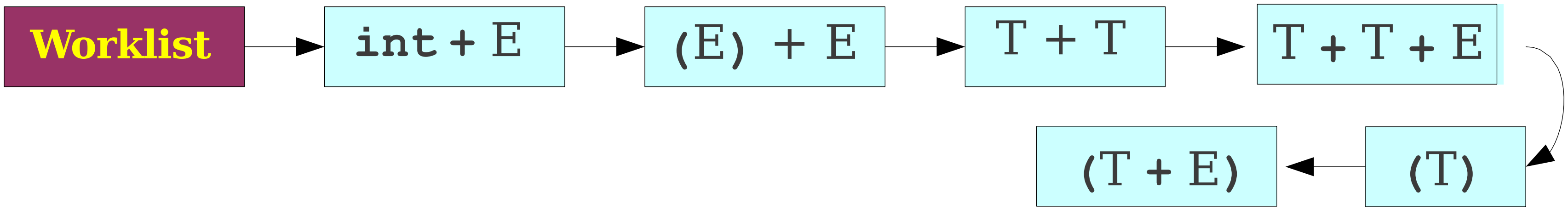
Breadth-First Search Parsing

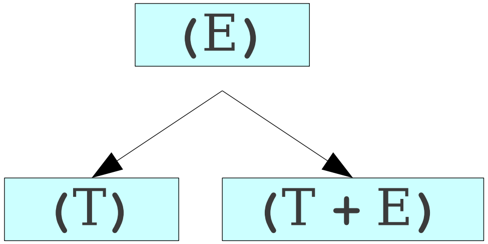


E → T
E → T + E
T → int
T → (E)

int + int

Breadth-First Search Parsing

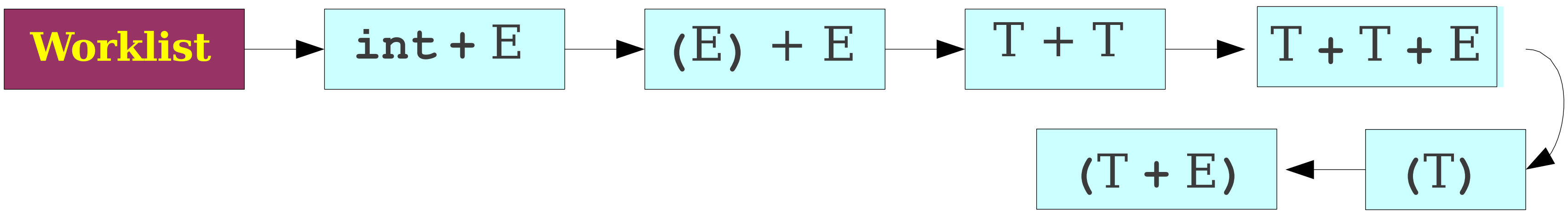




E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

`int + int`

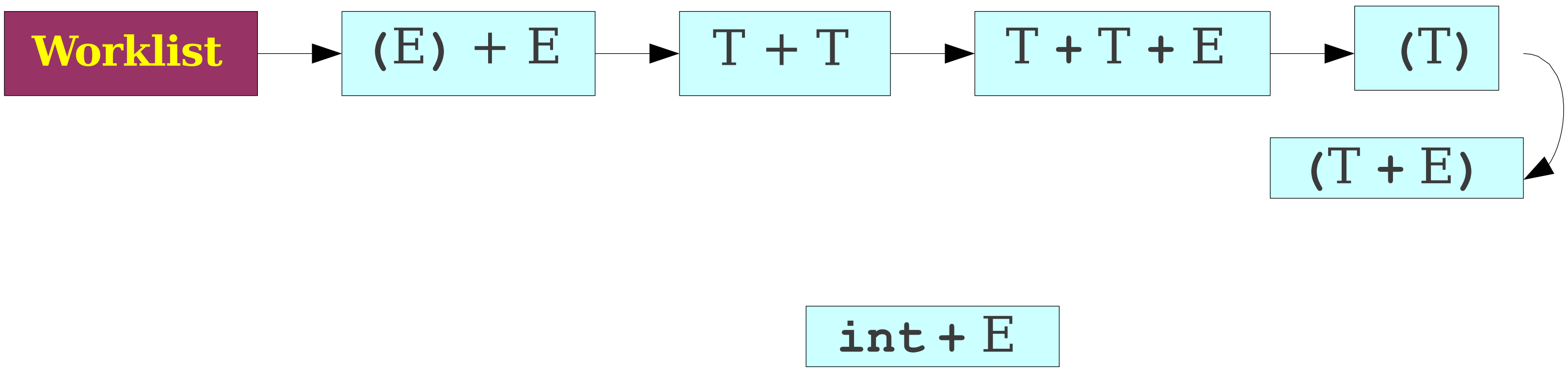
Breadth-First Search Parsing



$E \rightarrow T$
$E \rightarrow T$ **+** E
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**

int + int

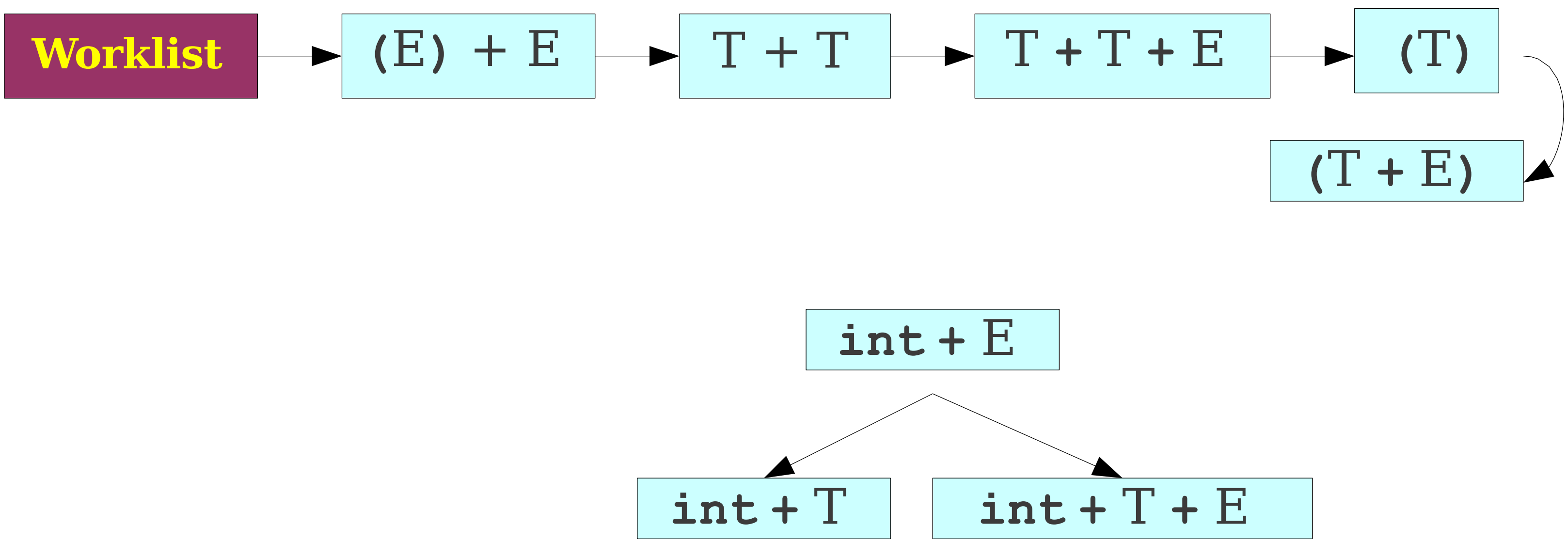
Breadth-First Search Parsing



$E \rightarrow T$
$E \rightarrow T$ + E
$T \rightarrow$ int
$T \rightarrow$ (E)

int + int

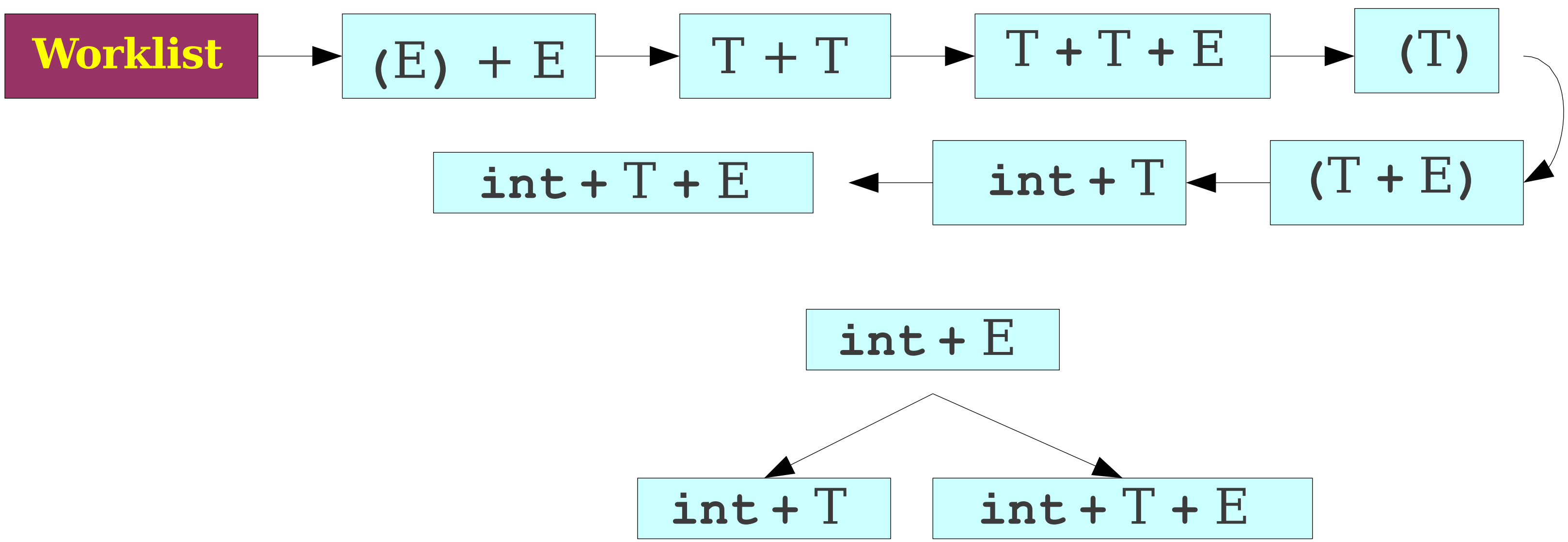
Breadth-First Search Parsing



$E \rightarrow T$
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(E)`

`int + int`

Breadth-First Search Parsing



$\mathbf{E} \rightarrow \mathbf{T}$

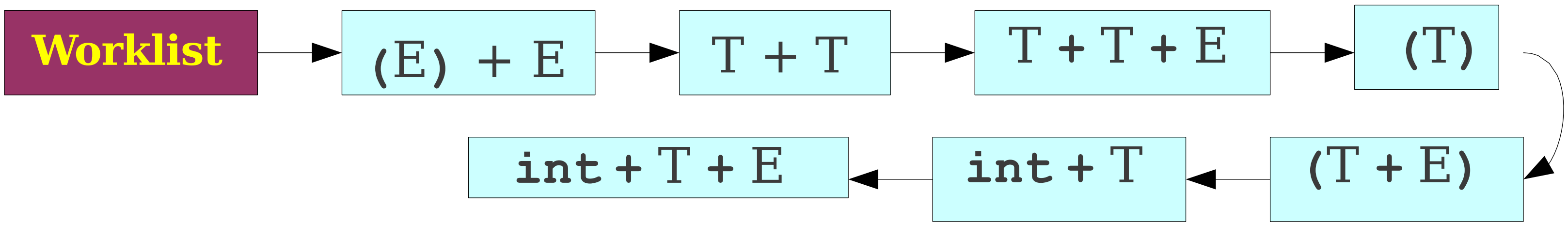
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

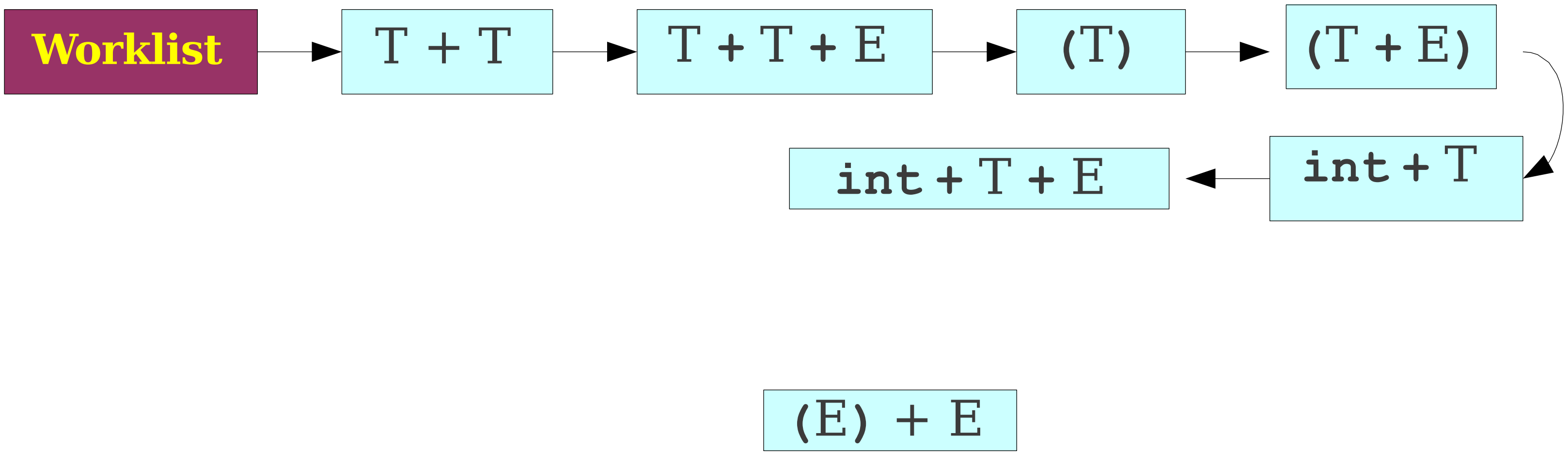
Breadth-First Search Parsing



$E \rightarrow T$
$E \rightarrow T$ + E
$T \rightarrow$ int
$T \rightarrow$ (E)

int + int

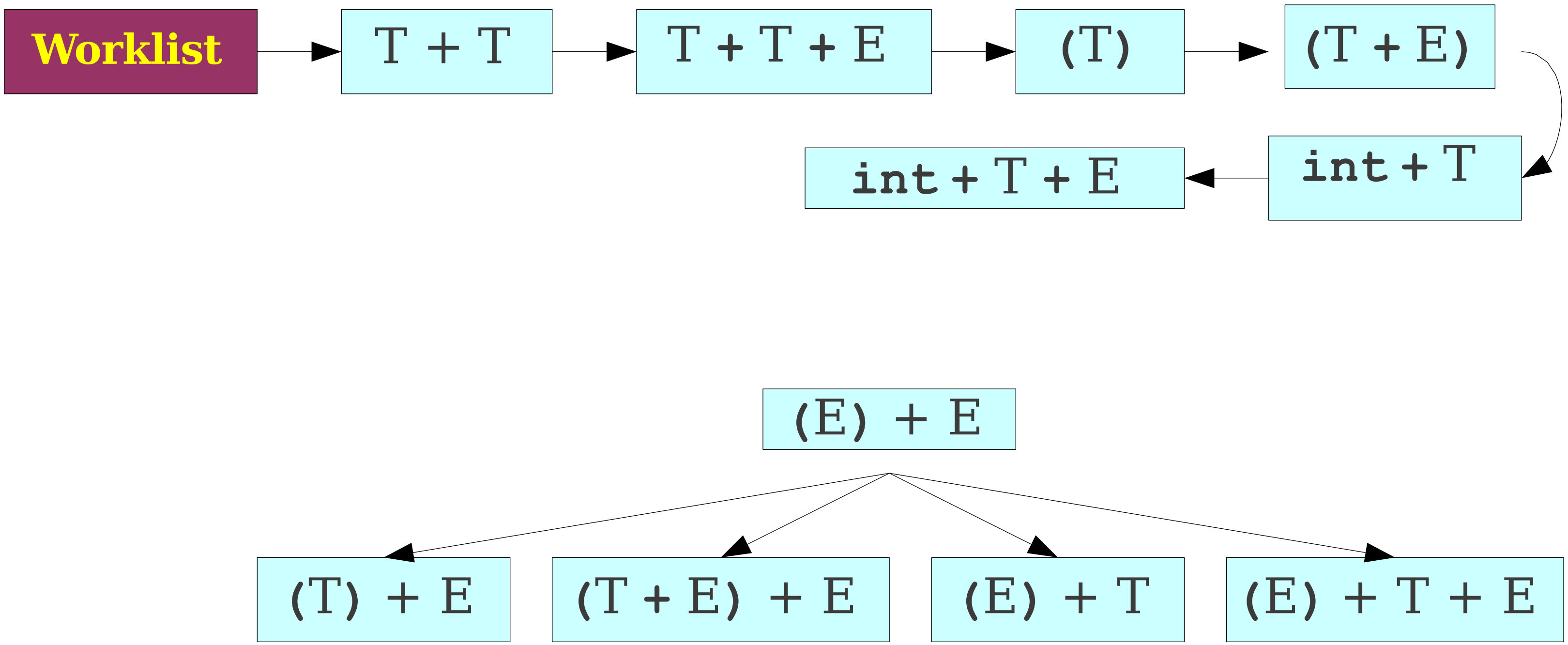
Breadth-First Search Parsing



$E \to T$
$E \to T + E$
$T \to$ int
$T \to$ (E)

int + int

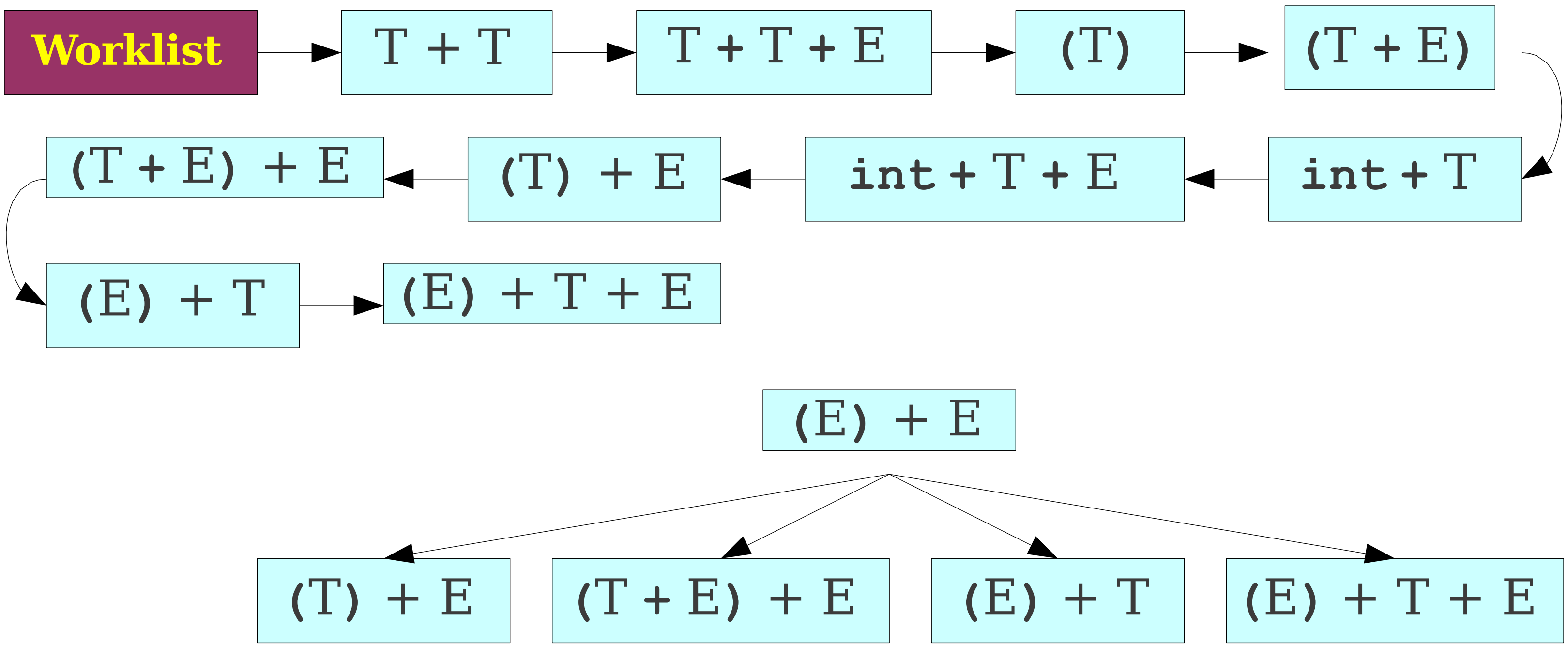
Breadth-First Search Parsing



$E \rightarrow T$
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

`int + int`

Breadth-First Search Parsing



E → **T**
E → **T** **+** **E**
T → **int**
T → **(E)**

int + int

Breadth-First Search Parsing

Worklist → T + T → T + T + E → (T) → (T + E) → int + T → int + T + E → (T) + E → (T + E) + E → (E) + T → (E) + T + E

$E \rightarrow T$

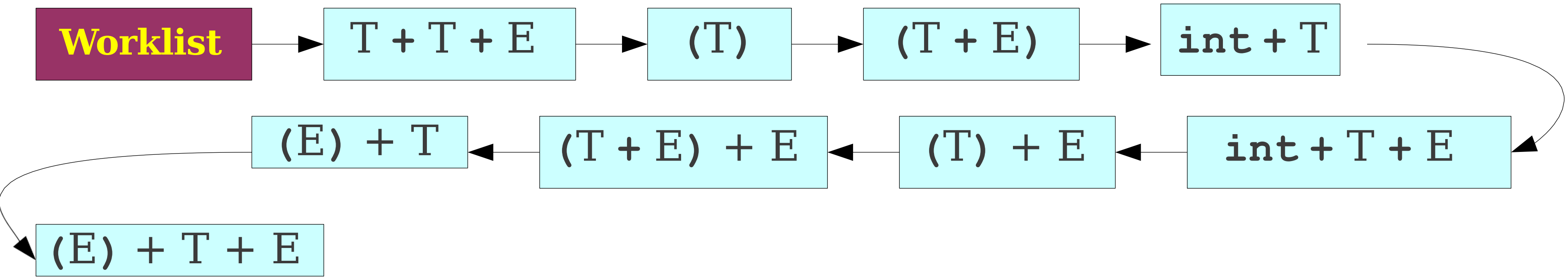
$E \rightarrow T$ `+` E

$T \rightarrow$ `int`

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

`int + int`

Breadth-First Search Parsing

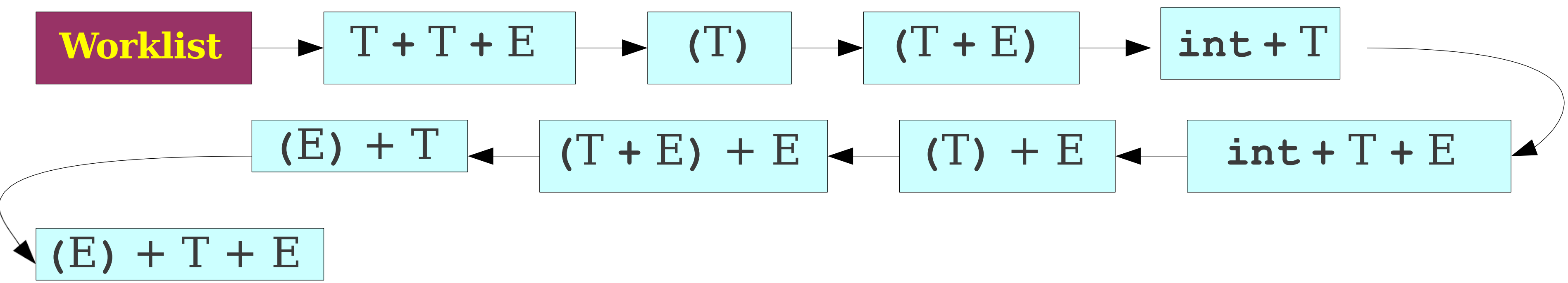


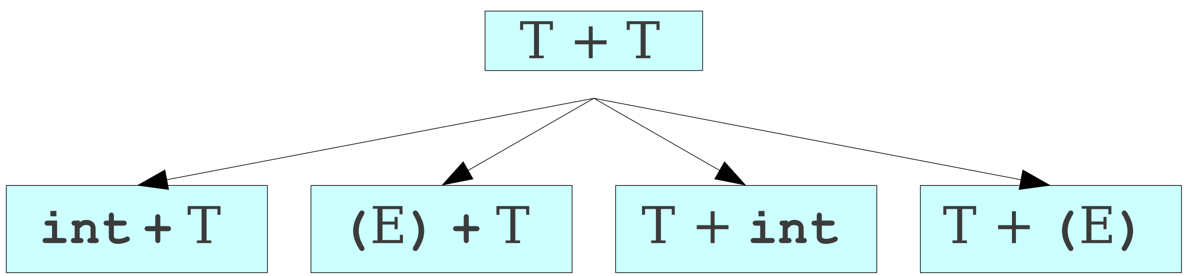
T + T

E → **T**
E → **T** **+** **E**
T → **int**
T → **(E)**

int + int

Breadth-First Search Parsing





E → **T**

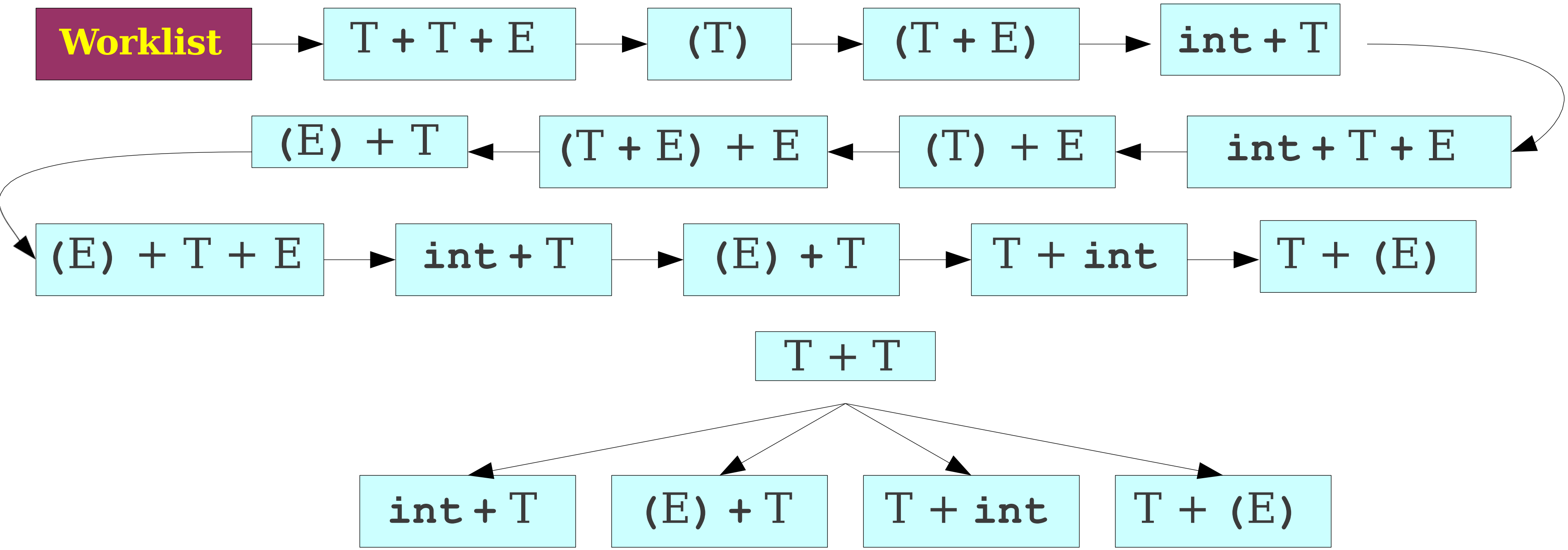
E → **T** `+` **E**

T → `int`

T → `(E)`

`int + int`

Breadth-First Search Parsing



E → **T**

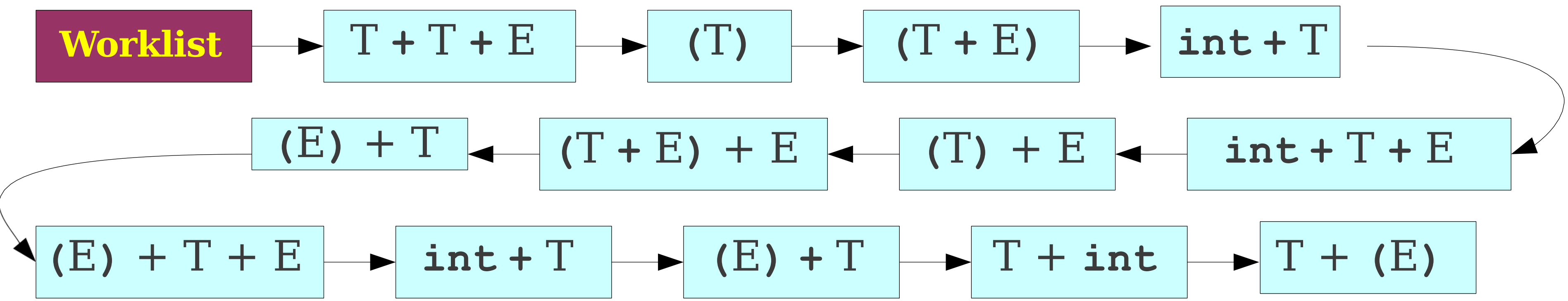
E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

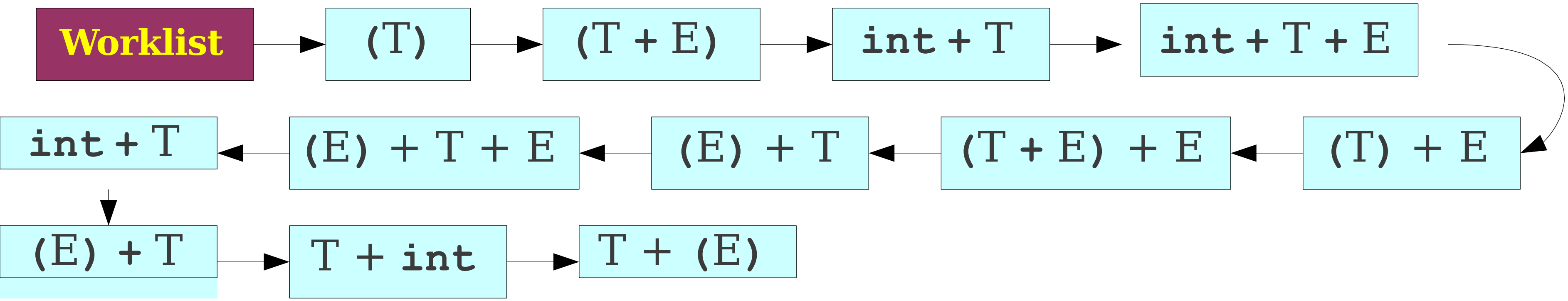
Breadth-First Search Parsing



$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

Breadth-First Search Parsing



T + T + E

$E \rightarrow T$

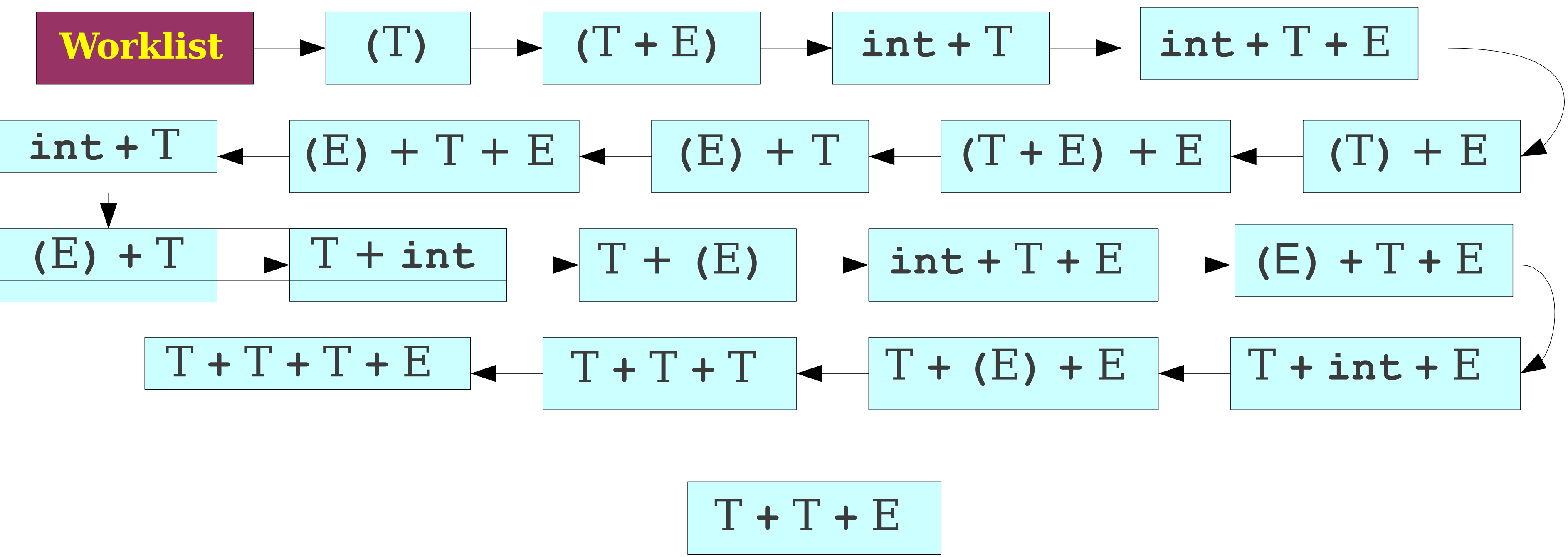
$E \rightarrow T$ **+** E

$T \rightarrow$ **int**

$T \rightarrow$ **(E)**

int + int

Breadth-First Search Parsing



$\mathbf{E} \rightarrow \mathbf{T}$

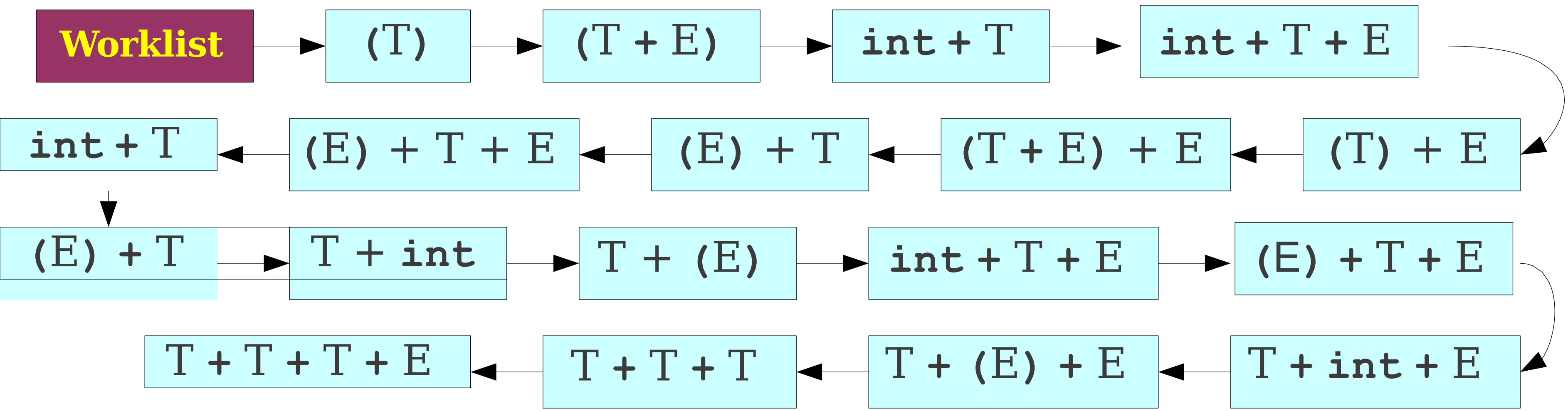
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(E)`

`int + int`

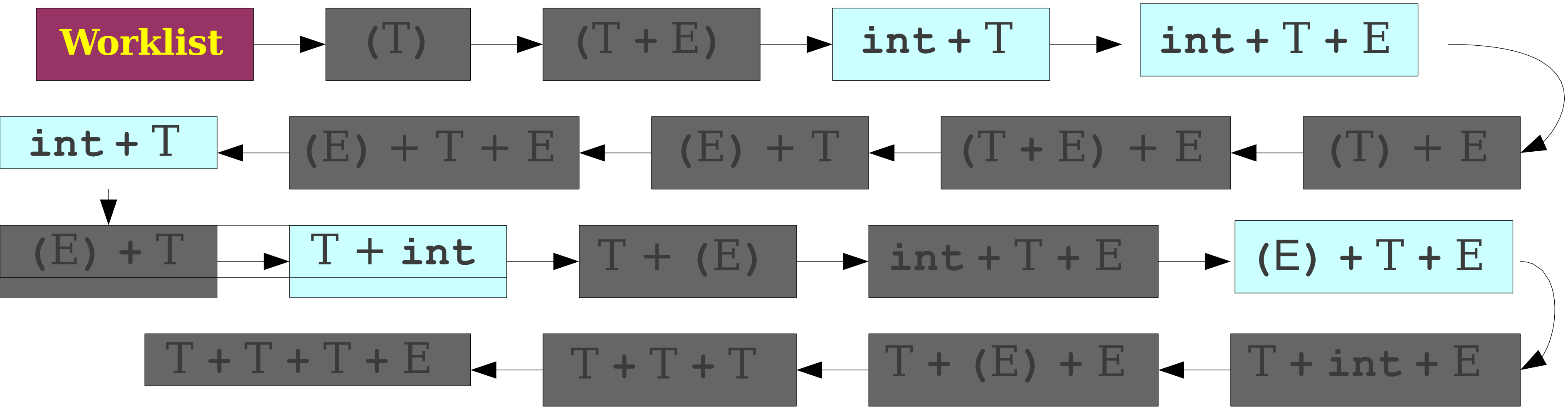
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing



$\mathbf{E} \rightarrow \mathbf{T}$

$\mathbf{E} \rightarrow \mathbf{T}$ **+** $\mathbf{E}$

$\mathbf{T} \rightarrow$ **int**

$\mathbf{T} \rightarrow$ **(E)**

int + int

BFS is Slow

- Enormous time and memory usage:
  - Lots of **wasted effort**:
    - Generates a lot of sentential forms that couldn't possibly match.
    - But in general, extremely hard to tell whether a sentential form can match – that's the job of parsing!
  - High **branching factor**:
    - Each sentential form can expand in (potentially) many ways for each nonterminal it contains.

Reducing Wasted Effort

- Suppose we're trying to match a string $\boldsymbol{\gamma}$.
- Suppose we have a sentential form (derived from CFG) $\boldsymbol{\tau} = \boldsymbol{\alpha\omega}$, where $\boldsymbol{\alpha}$ is a string of terminals and $\boldsymbol{\omega}$ is a string of terminals and nonterminals.
- If $\boldsymbol{\alpha}$ isn't a prefix of $\boldsymbol{\gamma}$, then no string derived from $\boldsymbol{\tau}$ can ever match $\boldsymbol{\gamma}$.
- If we can find a way to try to get a prefix of terminals at the front of our sentential forms, then we can start pruning out impossible options.

Reducing the Branching Factor

- If a string has many nonterminals in it, the branching factor can be high.
  - Sum of the number of productions of each nonterminal involved.
- If we can restrict which productions we apply, we can keep the branching factor lower.

Leftmost Derivations

- Recall: A **leftmost derivation** is one where we always expand the leftmost symbol first.
- Updated algorithm:
  - Do a breadth-first search, **only considering leftmost derivations**.
    - Dramatically drops branching factor.
    - Increases likelihood that we get a prefix of nonterminals.
  - Prune sentential forms that can't possibly match.
    - Avoids wasted effort.

Leftmost BFS

Worklist → E

$\mathbf{E} \rightarrow \mathbf{T}$

$\mathbf{E} \rightarrow \mathbf{T}$ **`+`** $\mathbf{E}$

$\mathbf{T} \rightarrow$ **`int`**

$\mathbf{T} \rightarrow$ **`(`**$\mathbf{E}$**`)`**

`int + int`

Leftmost BFS

Worklist

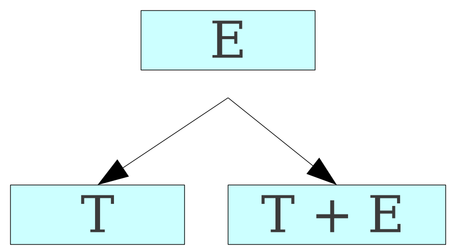
E

$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

Leftmost BFS

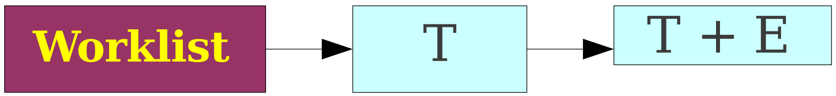
Worklist

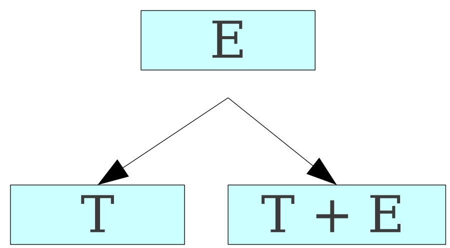


$E \rightarrow T$
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

`int + int`

Leftmost BFS





$\mathbf{E} \rightarrow \mathbf{T}$

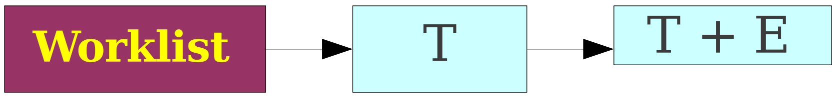
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

Leftmost BFS



$\mathbf{E} \rightarrow \mathbf{T}$

$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

Leftmost BFS

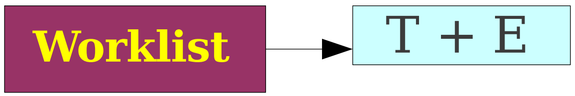
Worklist → T + E

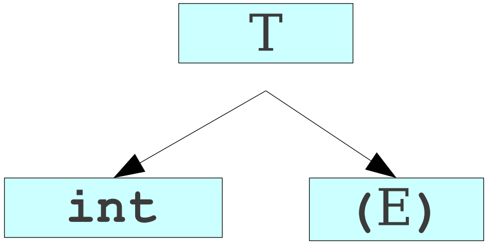
T

$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

Leftmost BFS

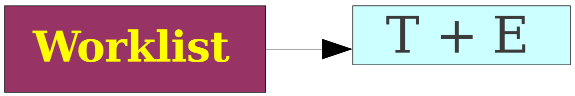


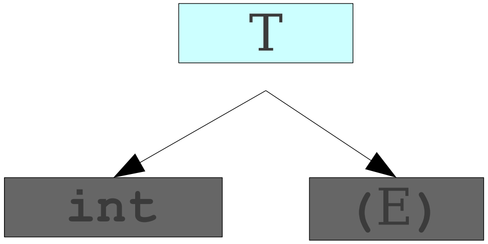


$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(E)`

`int + int`

Leftmost BFS





E → T
E → T + E
T → int
T → (E)

int + int

Leftmost BFS

Worklist → T + E

$E \rightarrow T$
$E \rightarrow T$ **+** E
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**

int + int

Leftmost BFS

Worklist

T + E

E → **T**

E → **T** `+` **E**

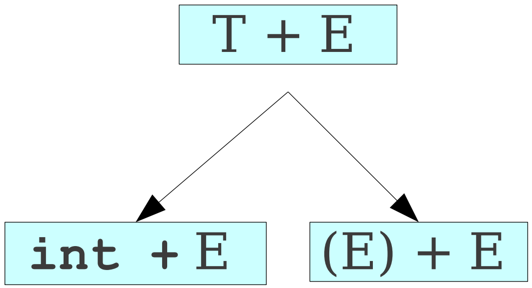
T → `int`

T → `(`**E**`)`

`int + int`

Leftmost BFS

Worklist

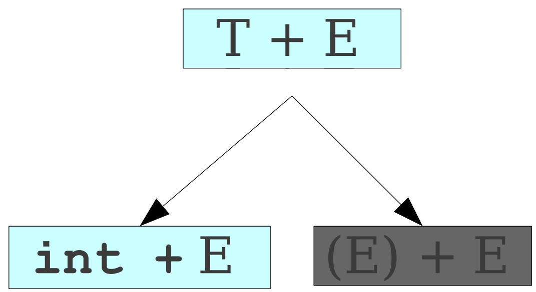


$E \rightarrow T$
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

`int + int`

Leftmost BFS

Worklist



$\mathbf{E} \rightarrow \mathbf{T}$

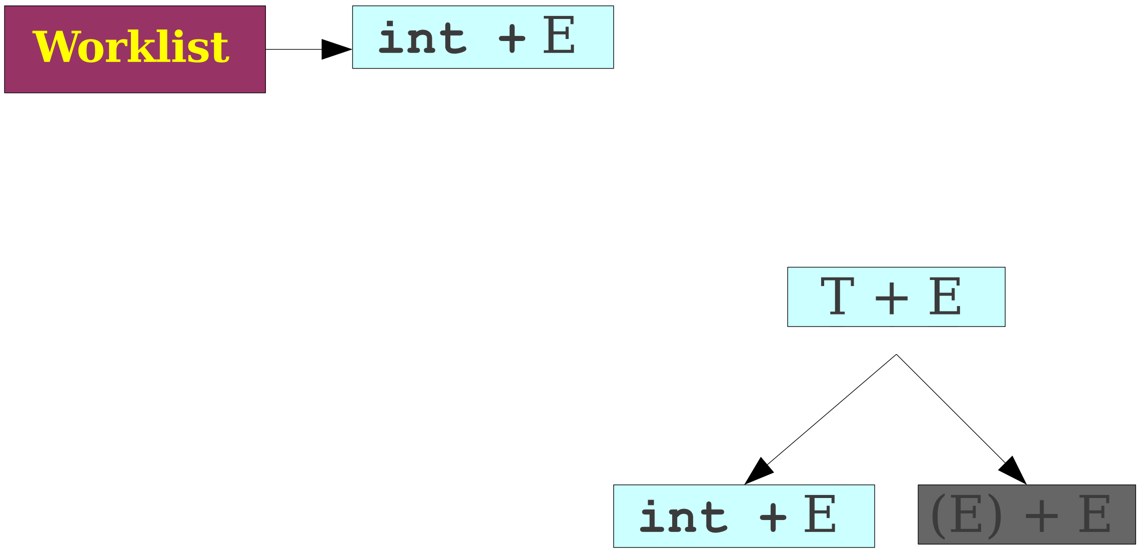
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

Leftmost BFS



$E \rightarrow T$

$E \rightarrow T$ **+** E

$T \rightarrow$ **int**

$T \rightarrow$ **(** E **)**

int + int

Leftmost BFS

Worklist	→	`int + E`

$$E \rightarrow T$$
$$E \rightarrow T \; \texttt{+} \; E$$
$$T \rightarrow \texttt{int}$$
$$T \rightarrow \texttt{(}E\texttt{)}$$

`int + int`

Leftmost BFS

Worklist

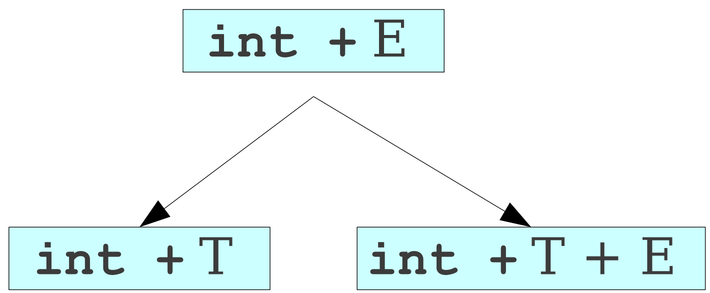
`int +` E

E → T
E → T `+` E
T → `int`
T → `(`E`)`

`int + int`

Leftmost BFS

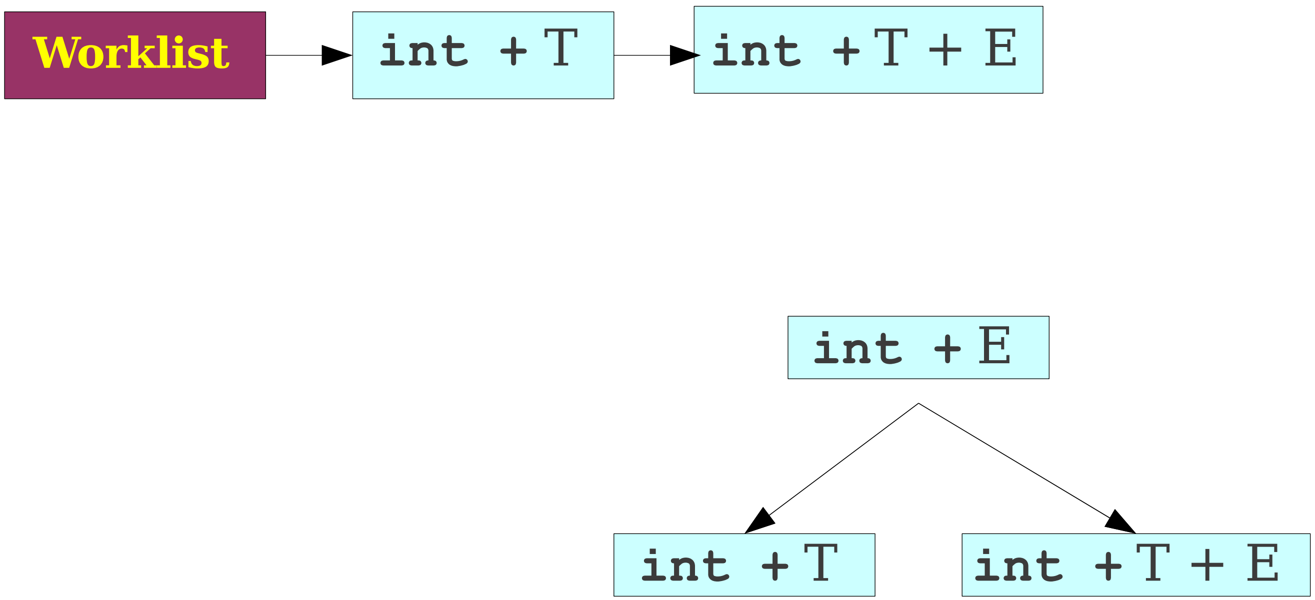
Worklist



E → T
E → T + E
T → int
T → (E)

int + int

Leftmost BFS



$E \to T$
$E \to T$ **+** E
$T \to$ **int**
$T \to$ **(E)**

int + int

Leftmost BFS

Worklist → `int` + T → `int` + T + E

$\mathbf{E} \rightarrow \mathbf{T}$

$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

Leftmost BFS

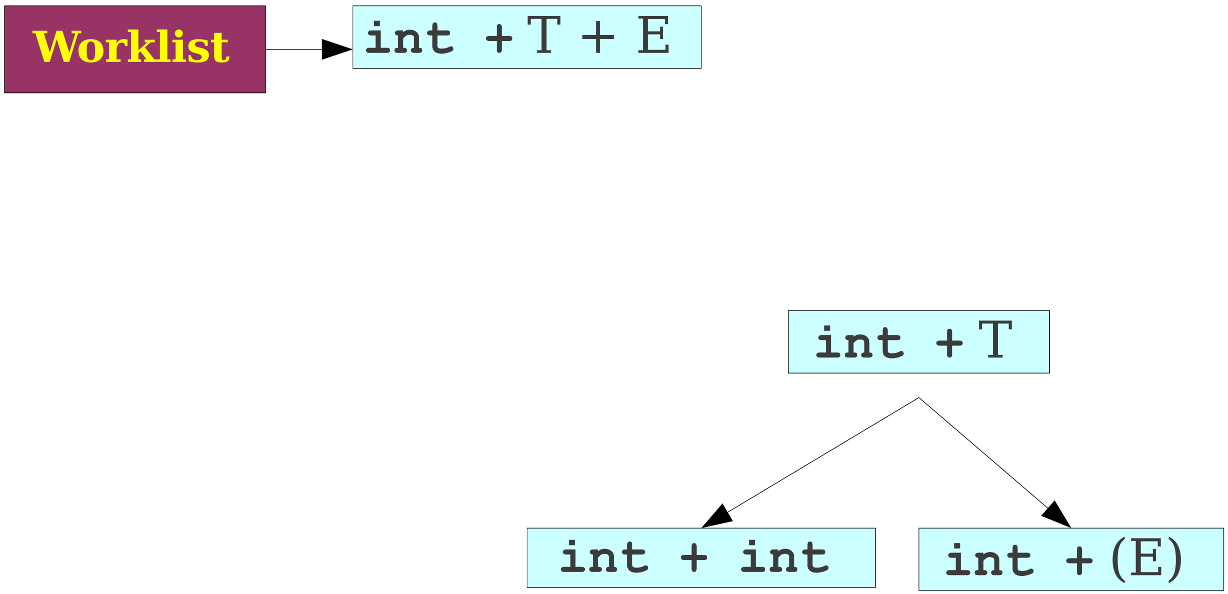
Worklist → int + T + E

int + T

$E \rightarrow T$
$E \rightarrow T$ + E
$T \rightarrow$ int
$T \rightarrow$ (E)

int + int

Leftmost BFS



$E \rightarrow T'$

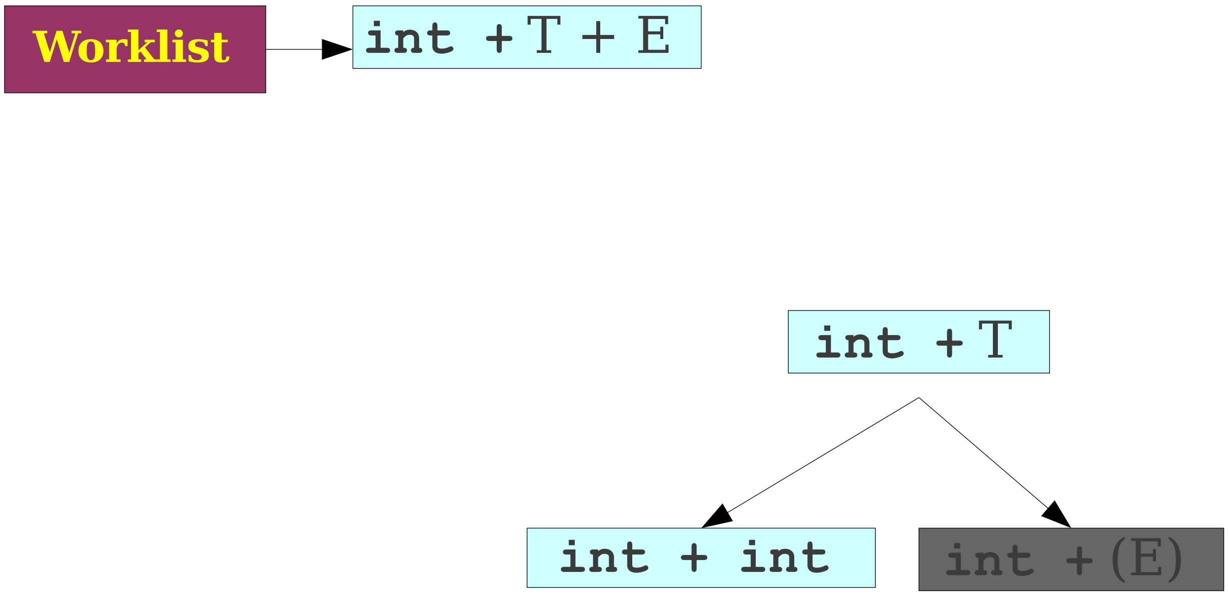
$E \rightarrow T$ **+** E

$T \rightarrow$ **int**

$T \rightarrow$ **(**E**)**

int + int

Leftmost BFS



$\mathbf{E} \rightarrow \mathbf{T}$

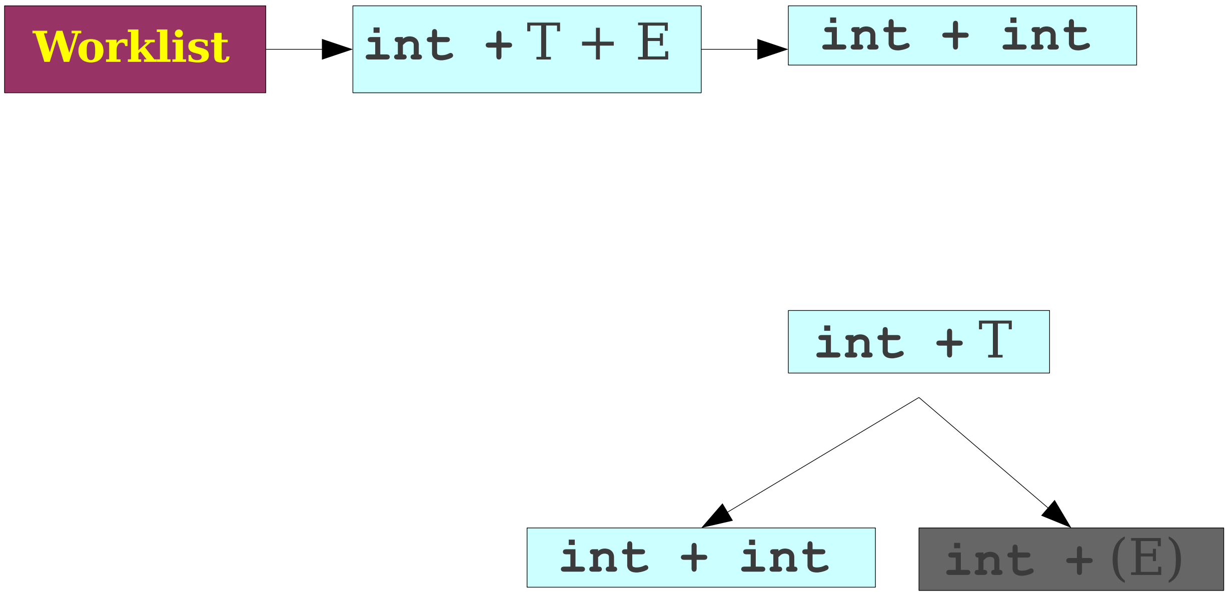
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

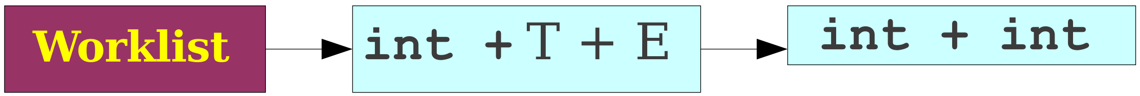
Leftmost BFS



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

`int + int`

Leftmost BFS



$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

Leftmost BFS

Worklist → `int + int`

`int +` T + E

$\mathbf{E} \rightarrow \mathbf{T}$

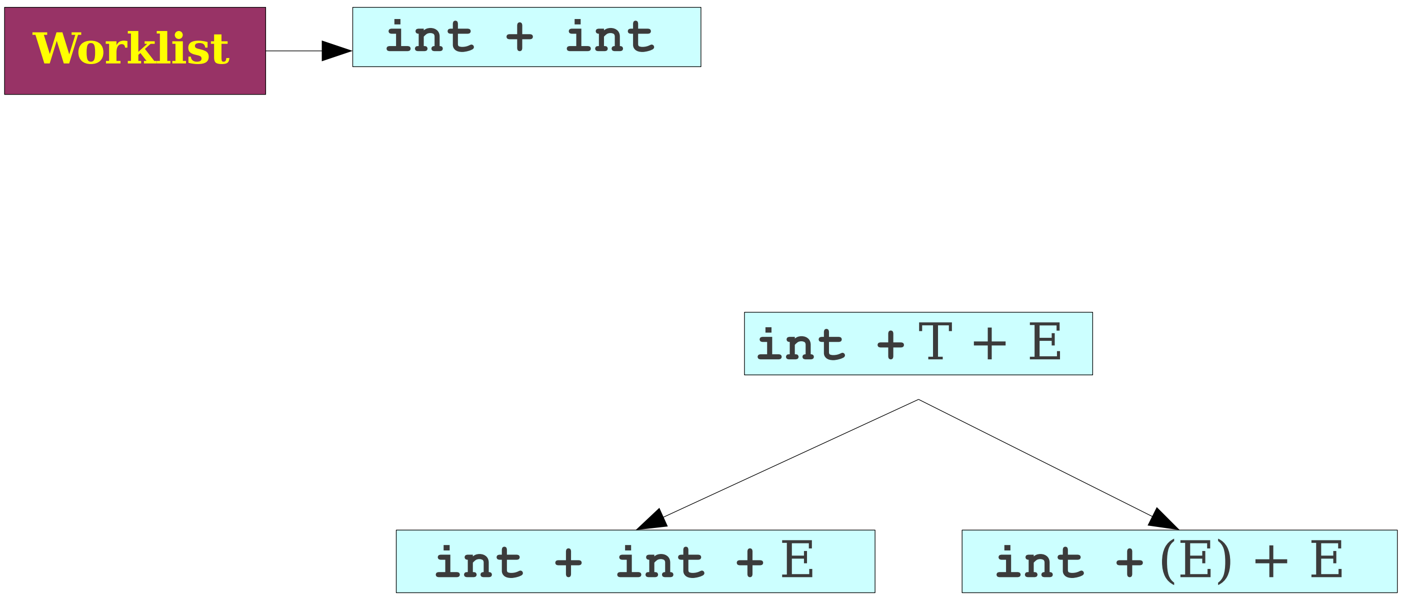
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

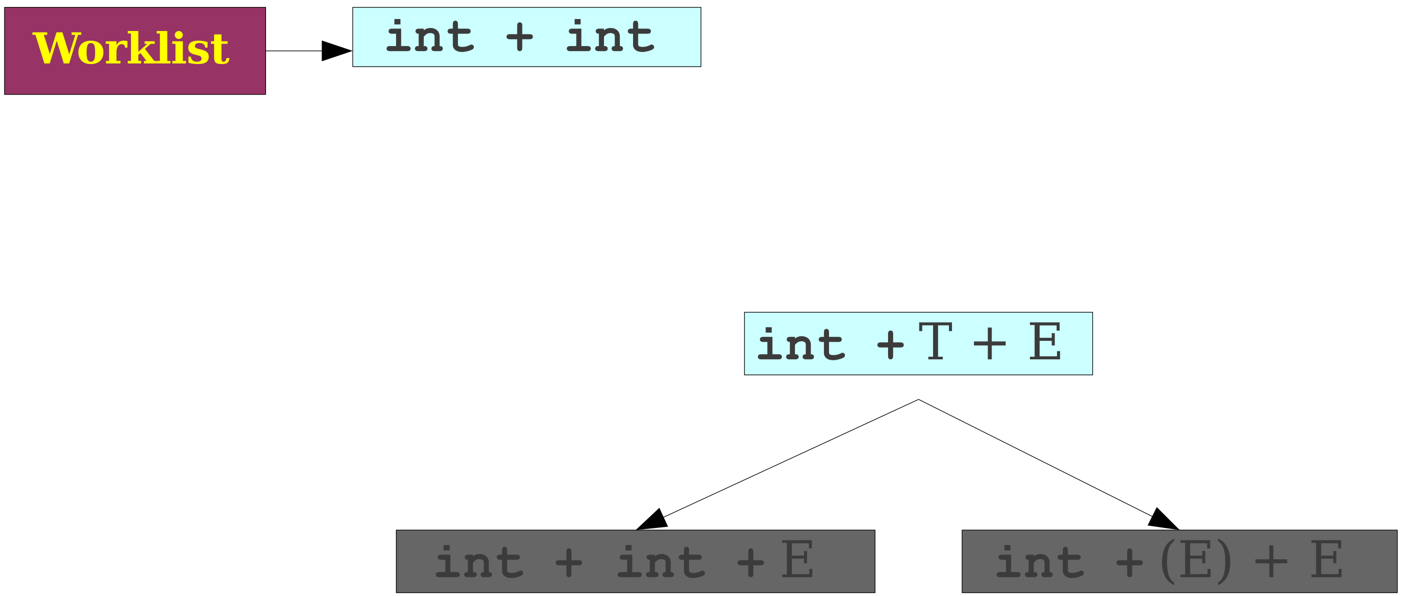
Leftmost BFS



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

`int + int`

Leftmost BFS



$\mathbf{E} \rightarrow \mathbf{T}$

$\mathbf{E} \rightarrow \mathbf{T}$ **`+`** $\mathbf{E}$

$\mathbf{T} \rightarrow$ **`int`**

$\mathbf{T} \rightarrow$ **`(`**$\mathbf{E}$**`)`**

`int + int`

Leftmost BFS

Worklist → `int + int`

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

`int + int`

Leftmost BFS

Worklist

`int + int`

$$E \rightarrow T$$
$$E \rightarrow T \; + \; E$$
$$T \rightarrow \texttt{int}$$
$$T \rightarrow (E)$$

`int + int`

Leftmost BFS

Worklist

`int + int`

$\mathbf{E} \rightarrow \mathbf{T}$

$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

$\mathbf{T} \rightarrow$ `int`

$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

Leftmost BFS

- Substantial improvement over naïve algorithm.
- Will always find a valid parse of a program if one exists.
- Can easily be modified to find if a program can't be parsed.
- But, there are still problems.

Leftmost BFS Has Problems

Worklist

$\mathbf{A} \rightarrow \mathbf{A}\mathtt{a} \mid \mathbf{A}\mathtt{b} \mid \mathtt{c}$

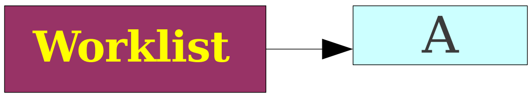
Leftmost BFS Has Problems

Worklist

$\mathbf{A} \rightarrow \mathbf{A}\mathtt{a} \mid \mathbf{A}\mathtt{b} \mid \mathtt{c}$

`caaaaaaaaaa`

Leftmost BFS Has Problems



$A \to Aa \mid Ab \mid c$

`caaaaaaaaaa`

Leftmost BFS Has Problems

Worklist

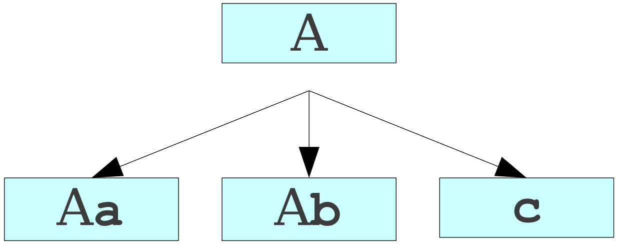
A

$\mathbf{A} \rightarrow \mathbf{A}\texttt{a} \mid \mathbf{A}\texttt{b} \mid \texttt{c}$

`caaaaaaaaaa`

Leftmost BFS Has Problems

Worklist

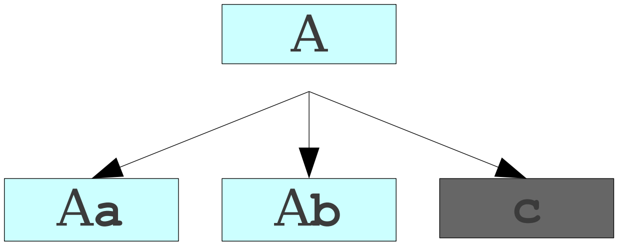


$A \rightarrow Aa \mid Ab \mid c$

caaaaaaaaa

Leftmost BFS Has Problems

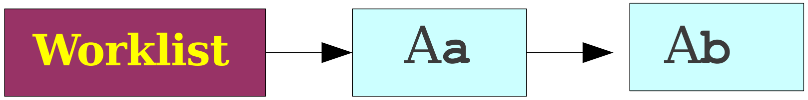
Worklist

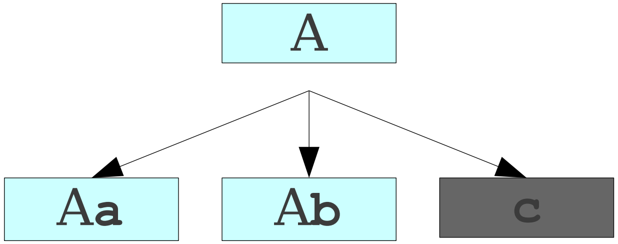


$\mathbf{A} \rightarrow \mathbf{A}\mathtt{a} \mid \mathbf{A}\mathtt{b} \mid \mathtt{c}$

`caaaaaaaaaa`

Leftmost BFS Has Problems

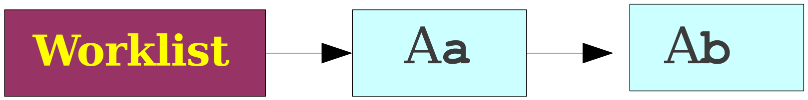




A → **A**a | **A**b | c

caaaaaaaaaa

Leftmost BFS Has Problems



$A \rightarrow Aa \mid Ab \mid c$

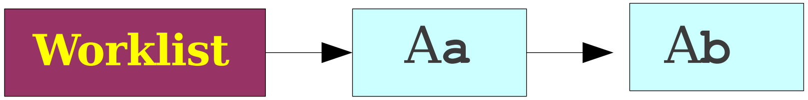
caaaaaaaaaa

Leftmost BFS Has Problems

$A \rightarrow Aa \mid Ab \mid c$

caaaaaaaaaa

Leftmost BFS Has Problems



A → **A**a | **A**b | c

caaaaaaaaaa

Leftmost BFS Has Problems

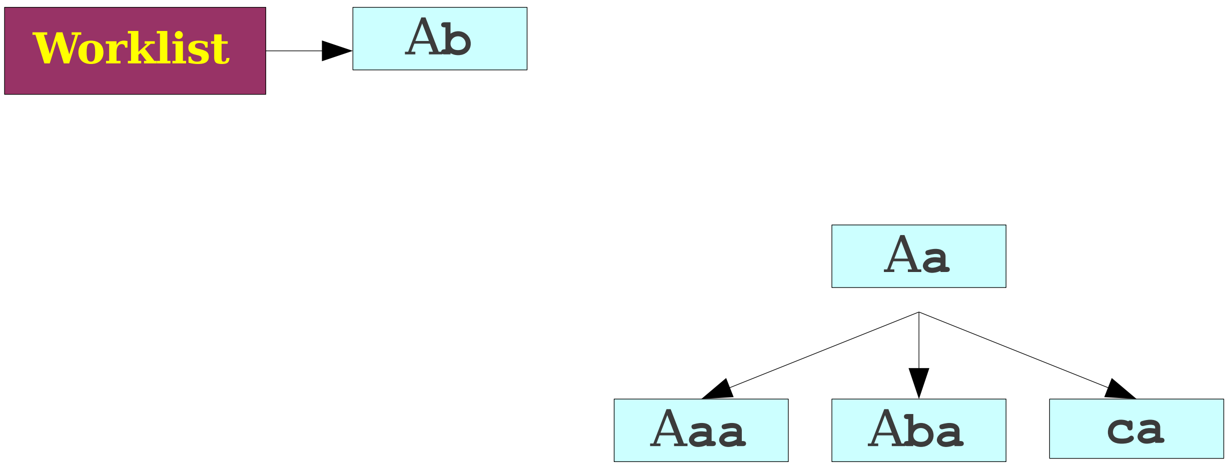
Worklist → Ab

Aa

A → **Aa** | **Ab** | **c**

caaaaaaaaaa

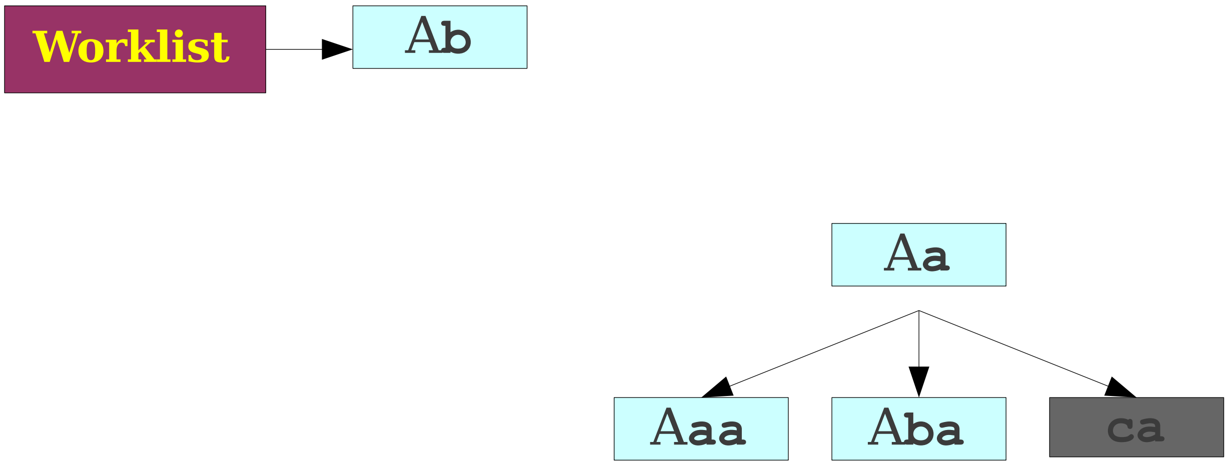
Leftmost BFS Has Problems



$A \rightarrow Aa \mid Ab \mid c$

caaaaaaaaaa

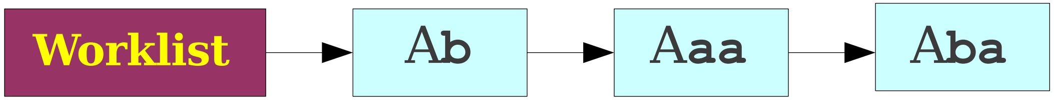
Leftmost BFS Has Problems

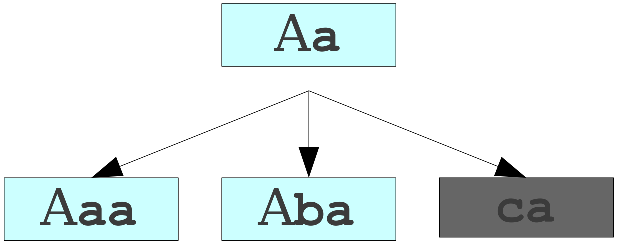


A → **Aa** | **Ab** | **c**

caaaaaaaaaa

Leftmost BFS Has Problems





A → **A**`a` | **A**`b` | `c`

`caaaaaaaaaa`

Leftmost BFS Has Problems

Worklist → Ab → Aaa → Aba

$A \to Aa \mid Ab \mid c$

caaaaaaaaaa

Leftmost BFS Has Problems

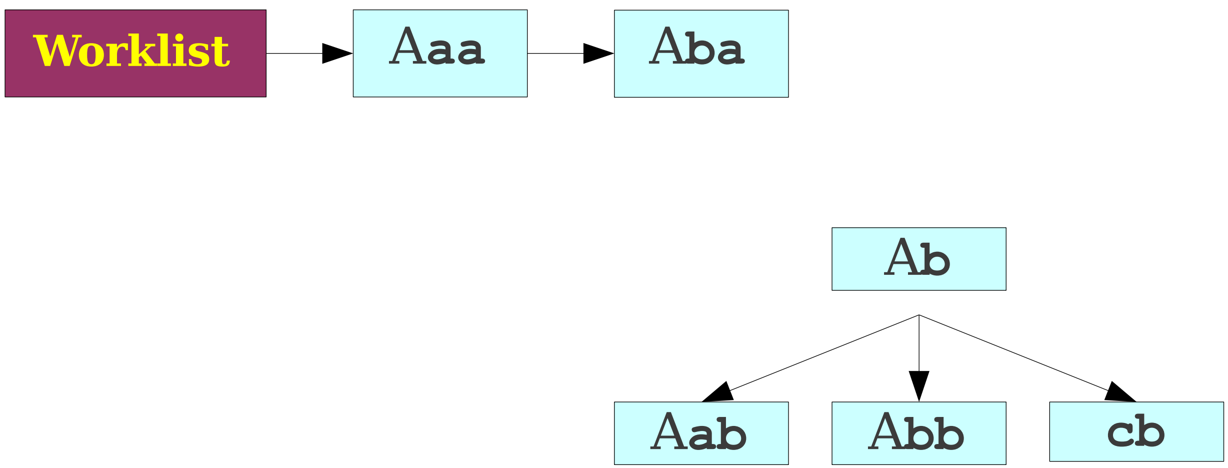
Worklist → Aaa → Aba

Ab

A → Aa | Ab | c

caaaaaaaaaa

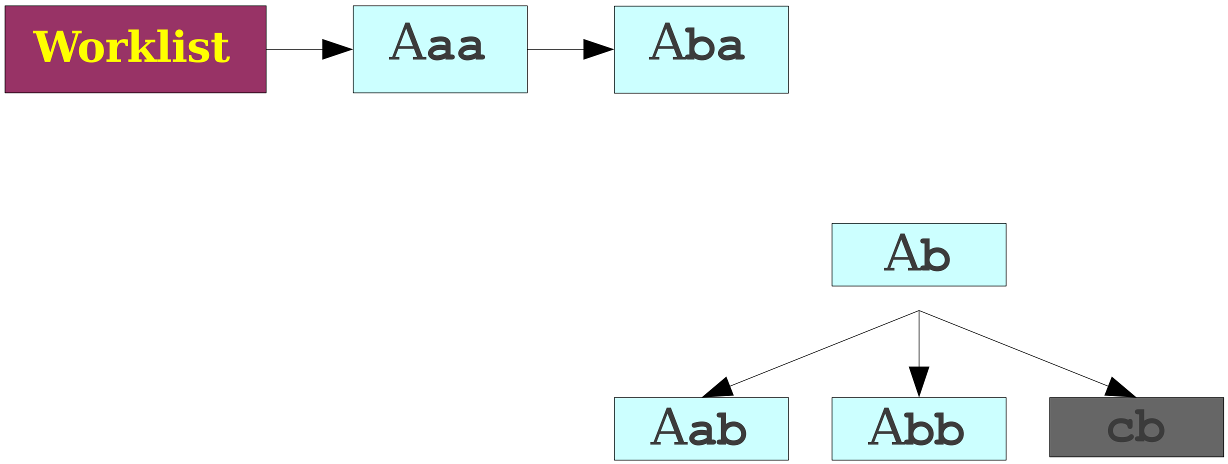
Leftmost BFS Has Problems



$A \rightarrow Aa \mid Ab \mid c$

caaaaaaaaaa

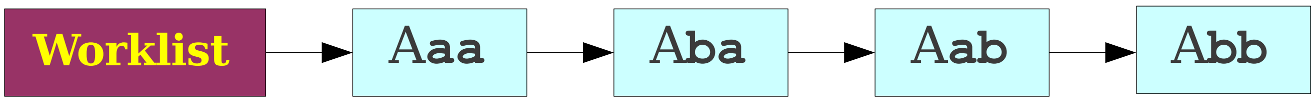
Leftmost BFS Has Problems

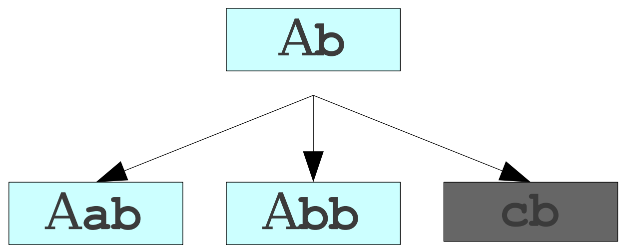


$A \rightarrow Aa \mid Ab \mid c$

caaaaaaaaa

Leftmost BFS Has Problems





$\mathbf{A} \rightarrow \mathbf{A}\mathtt{a} \mid \mathbf{A}\mathtt{b} \mid \mathtt{c}$

caaaaaaaaaa

Leftmost BFS Has Problems

Worklist → `Aaa` → `Aba` → `Aab` → `Abb`

$\mathbf{A} \rightarrow \mathbf{A}\mathtt{a} \mid \mathbf{A}\mathtt{b} \mid \mathtt{c}$

`caaaaaaaaaa`

Leftmost BFS Has Problems

Worklist → `Aaa` → `Aba` → `Aab` → `Abb`

$A \rightarrow A$`a` | A`b` | `c`

`caaaaaaaaaa`

Problems with Leftmost BFS

- Grammars like this can make parsing take exponential time.
- Also uses exponential memory.
- What if we search the graph with a different algorithm?

Leftmost DFS

- Idea: Use **depth-first** search.
- Advantages:
  - Lower memory usage: Only considers one branch at a time.
  - High performance: On many grammars, runs very quickly.
  - Easy to implement: Can be written as a set of mutually recursive functions.

Leftmost DFS

$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

Leftmost DFS

$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{T}$ **`+`** $\mathbf{E}$
$\mathbf{T} \rightarrow$ **`int`**
$\mathbf{T} \rightarrow$ **`(`**$\mathbf{E}$**`)`**

`int + int`

Leftmost DFS

E

$E \rightarrow T$
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

`int + int`

Leftmost DFS

E
T

$E \rightarrow T$
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

`int + int`

Leftmost DFS

E
T
`int`

$E \rightarrow T$
$E \rightarrow T$ **`+`** E
$T \rightarrow$ **`int`**
$T \rightarrow$ **`(`**E**`)`**

`int + int`

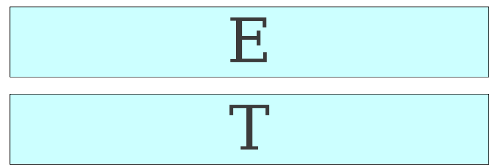
Leftmost DFS

E
T
`int`

$E \rightarrow T$
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

`int + int`

Leftmost DFS



$E \rightarrow T$

$E \rightarrow T$ **`+`** E

$T \rightarrow$ **`int`**

$T \rightarrow$ **`(`**E**`)`**

`int + int`

Leftmost DFS

E
T
(E)

$E \rightarrow T$
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

`int + int`

Leftmost DFS

E
T
(E)

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

`int + int`

Leftmost DFS

E

T

$E \rightarrow T$

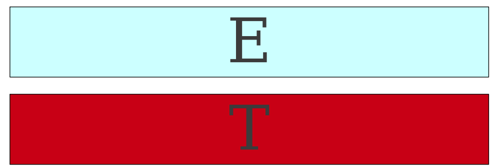
$E \rightarrow T$ **+** E

$T \rightarrow$ **int**

$T \rightarrow$ **(**E**)**

int + int

Leftmost DFS



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

`int + int`

Leftmost DFS

E

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

`int + int`

Leftmost DFS

E

T + E

$E \rightarrow T$

$E \rightarrow T$ + E

$T \rightarrow$ int

$T \rightarrow$ (E)

int + int

Leftmost DFS

E
T **+** E
int + E

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

`int + int`

Leftmost DFS

$E \rightarrow T$
$E \rightarrow T$ `+` E
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

E
T + E
int + E
int + T

`int + int`

Leftmost DFS

$E \rightarrow T$
$E \rightarrow T$ **+** E
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**

E
T **+** E
int + E
int + T
int + int

int + int

Leftmost DFS

$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow$ `int`
$T \rightarrow$ `(`E`)`

E
T + E
int + E
int + T
int + int

`int + int`

Problems with Leftmost DFS

$A \rightarrow Aa \mid c$

A
Aa
Aaa
Aaaa
Aaaaa

c