CS251: Introduction to Language Processing

Top-Down Parsing

Vishwesh Jatala

Department of CSE

Indian Institute of Technology Bhilai

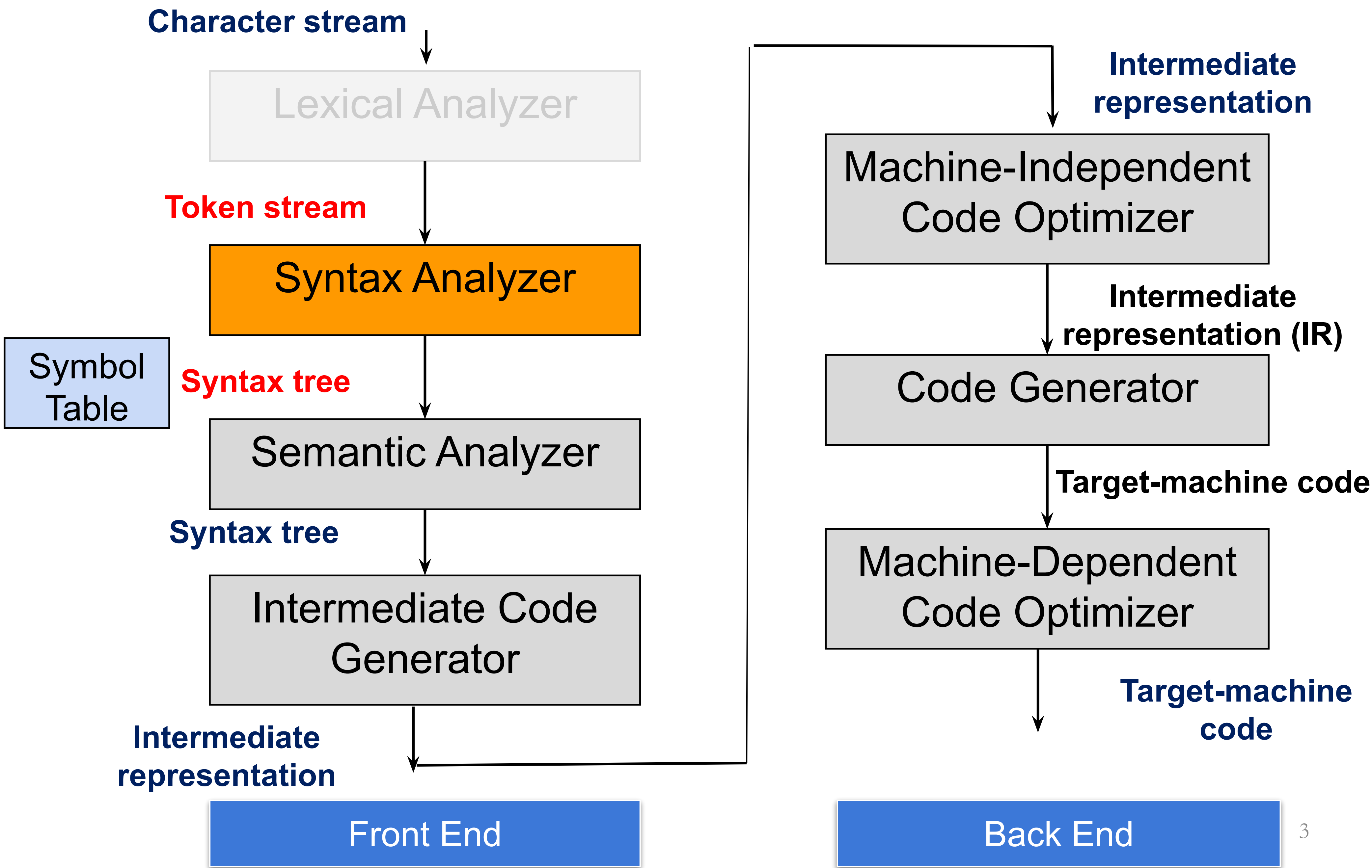
vishwesh@iitbhilai.ac.in

2023-24-M

Acknowledgement

- Today's slides are modified from that of
  - *Stanford University:*
    - *https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/*

Compiler Design



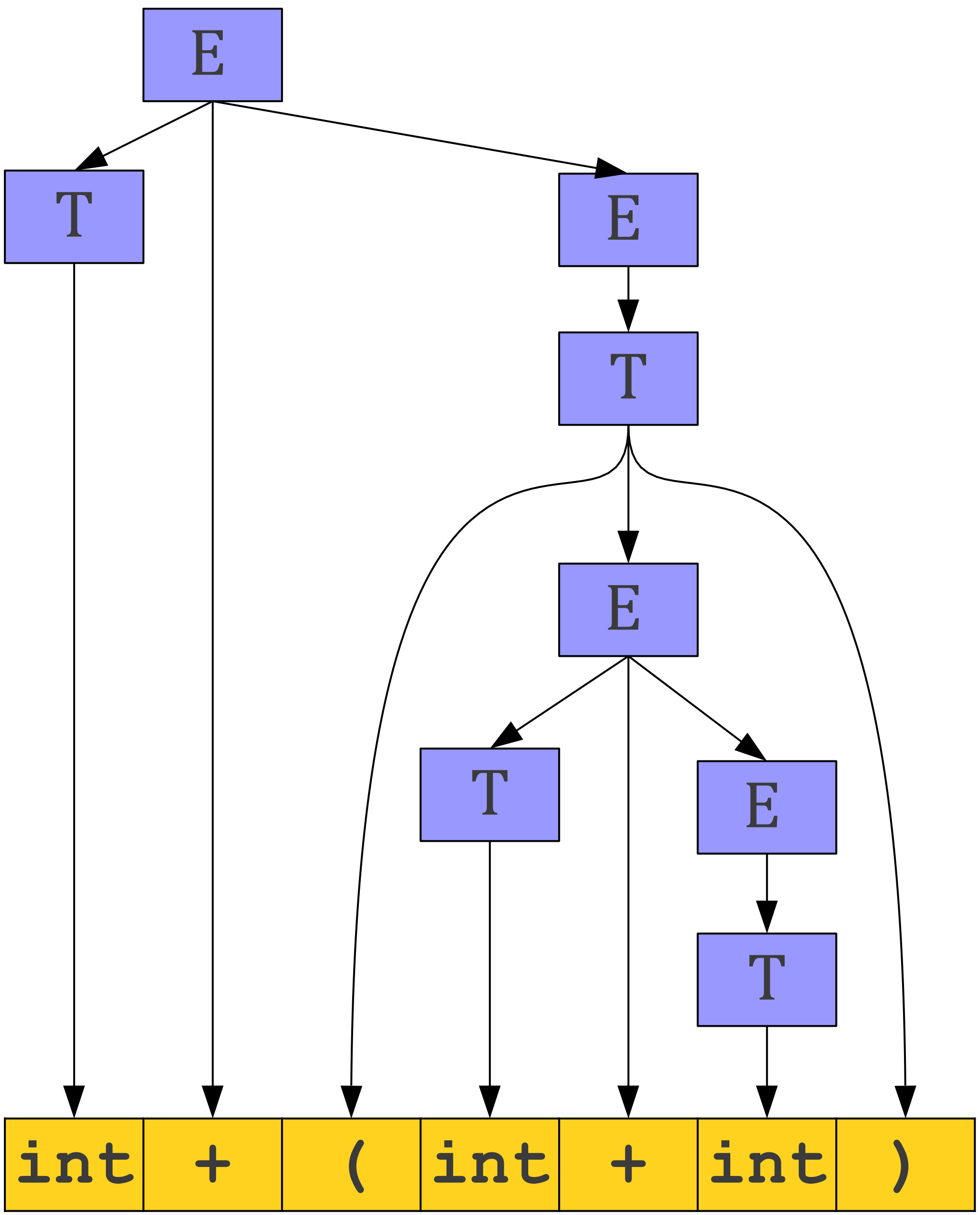
Recap: Parsing

E → T

E → T + E

T → int

T → (E)



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.

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

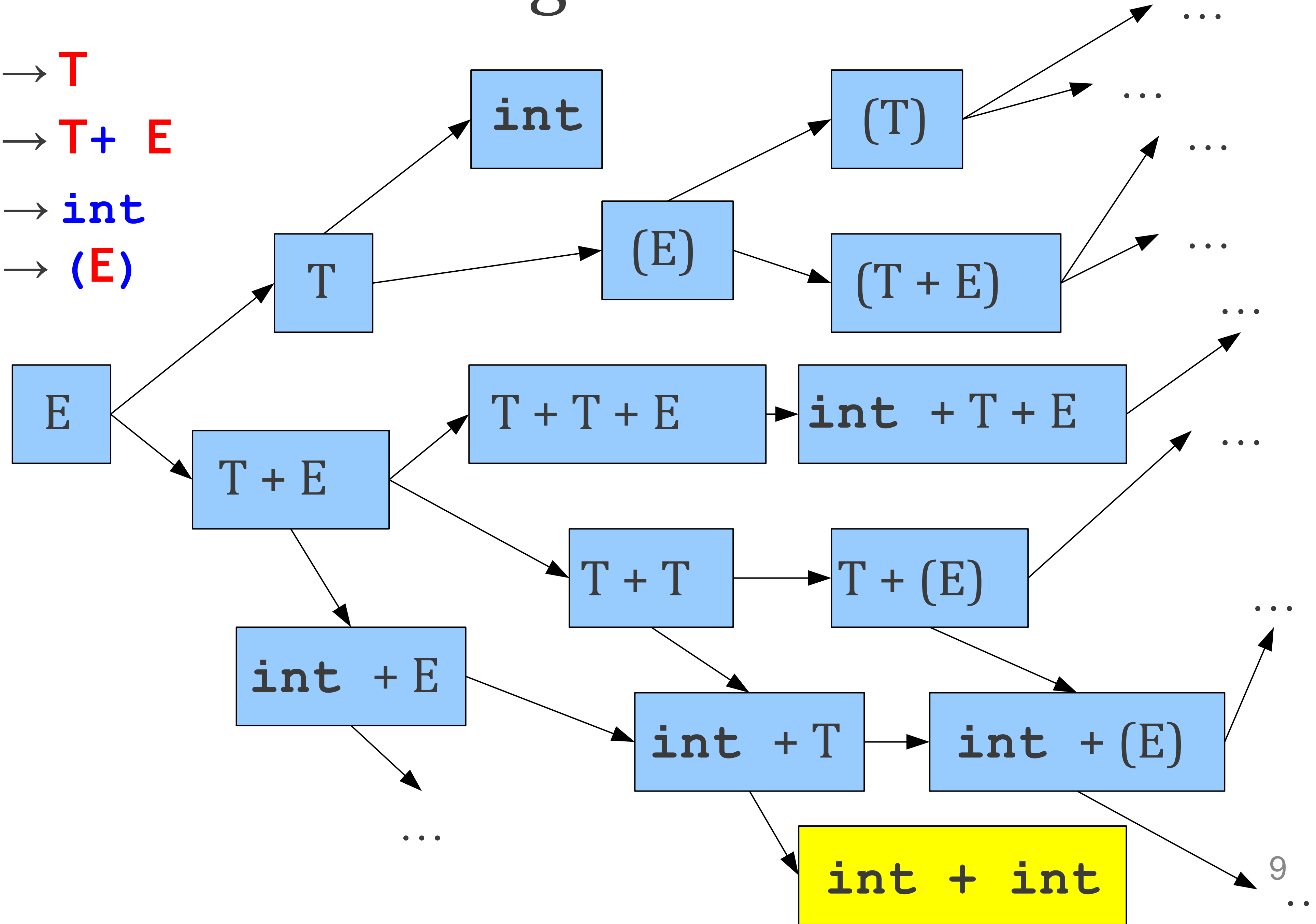
- Top-down parsing begins with virtually no information.
  - Begins with just the start symbol
- How can we know which productions to apply?
- In general, we can't.
  - There are some grammars for which the best we can do is guess and backtrack if we're wrong.

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 α to node β iff $\alpha \Rightarrow \beta$.

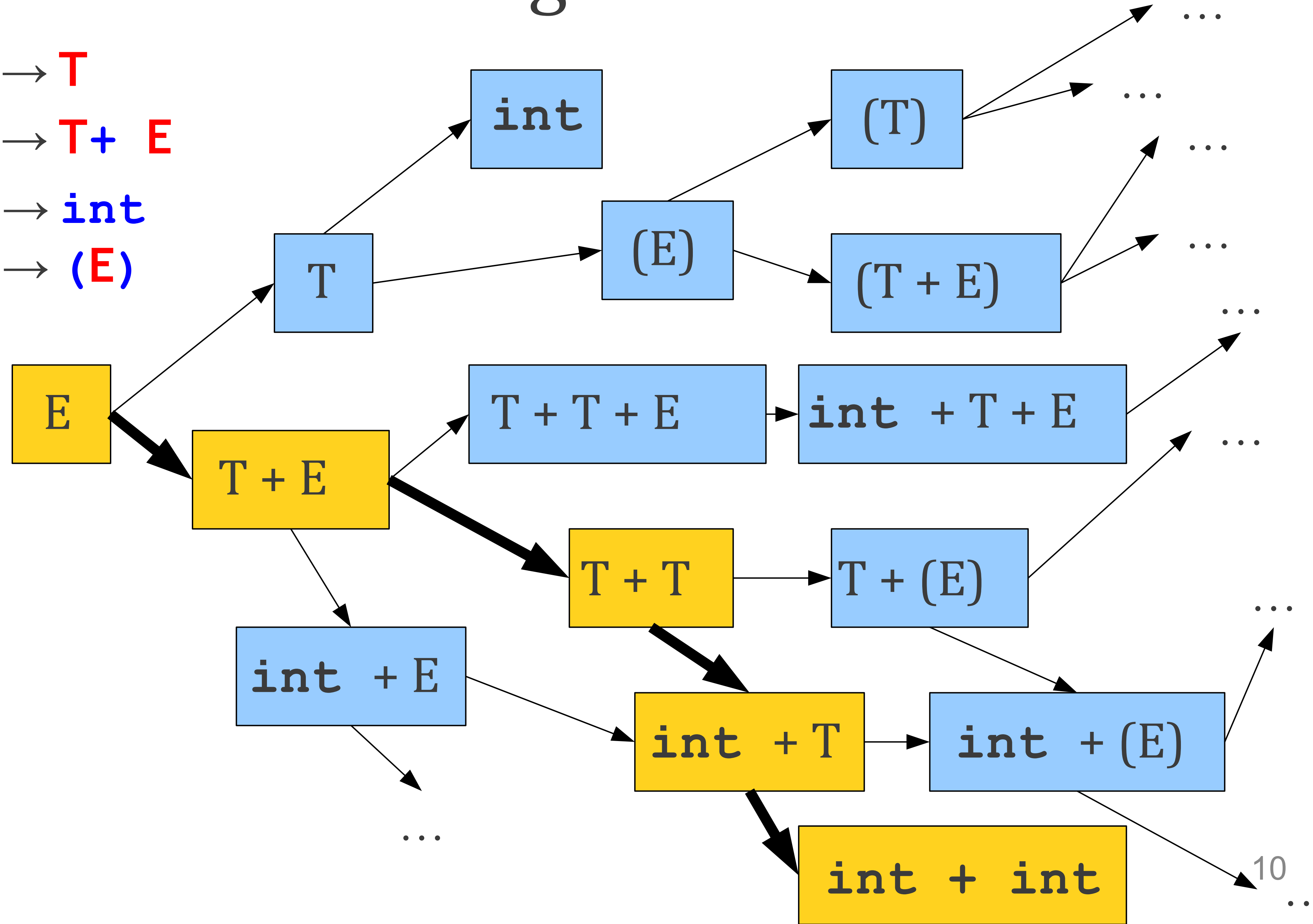
Parsing as a Search

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 → T

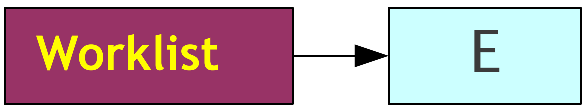
E → T + E

T → int

T → (E)

`int + int`

Breadth-First Search Parsing



$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

`int + int`

Breadth-First Search Parsing

Worklist

E

$E \rightarrow T$

$E \rightarrow T + E$

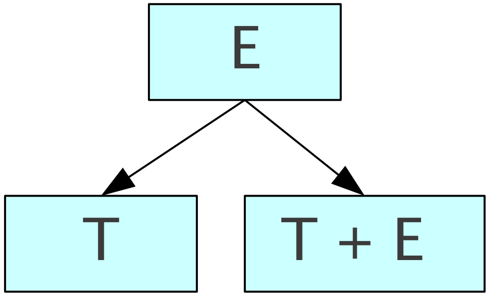
$T \rightarrow \texttt{int}$

$T \rightarrow \texttt{(E)}$

`int + int`

Breadth-First Search Parsing

Worklist



$E \rightarrow T$

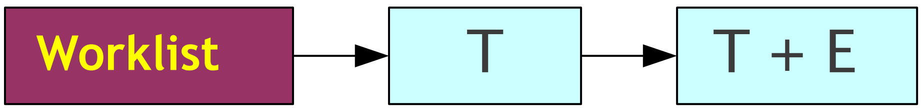
$E \rightarrow T + E$

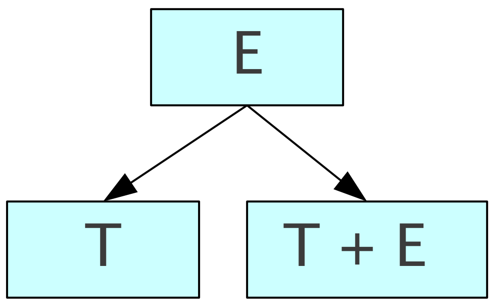
$T \rightarrow$ `int`

$T \rightarrow$ `(E)`

`int + int`

Breadth-First Search Parsing





E → T

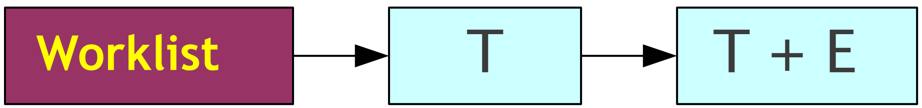
E → T + E

T → int

T → (E)

int + int

Breadth-First Search Parsing



E → T

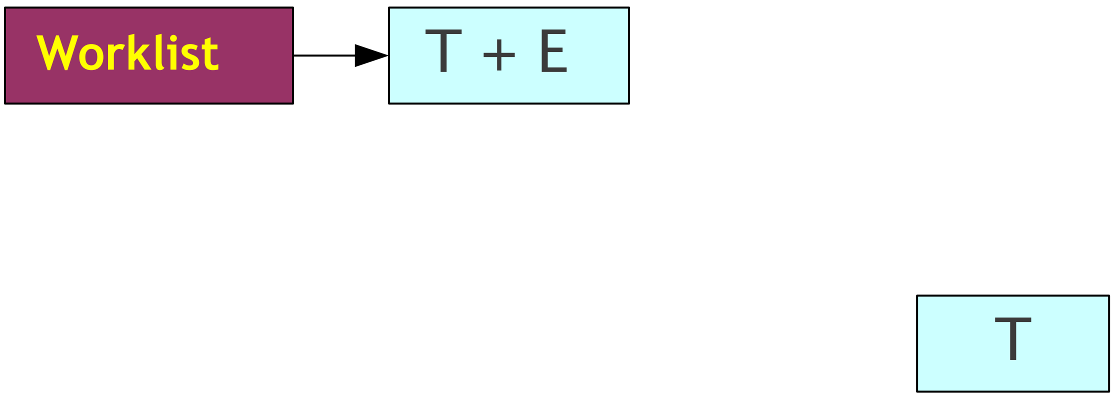
E → T + E

T → int

T → (E)

int + int

Breadth-First Search Parsing



$E \rightarrow T$

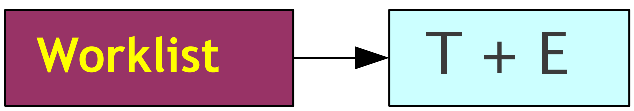
$E \rightarrow T + E$

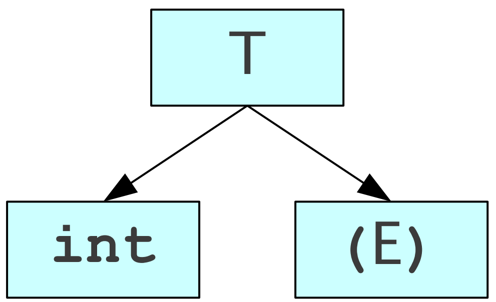
$T \rightarrow$ `int`

$T \rightarrow$ `(E)`

`int + int`

Breadth-First Search Parsing





E → T

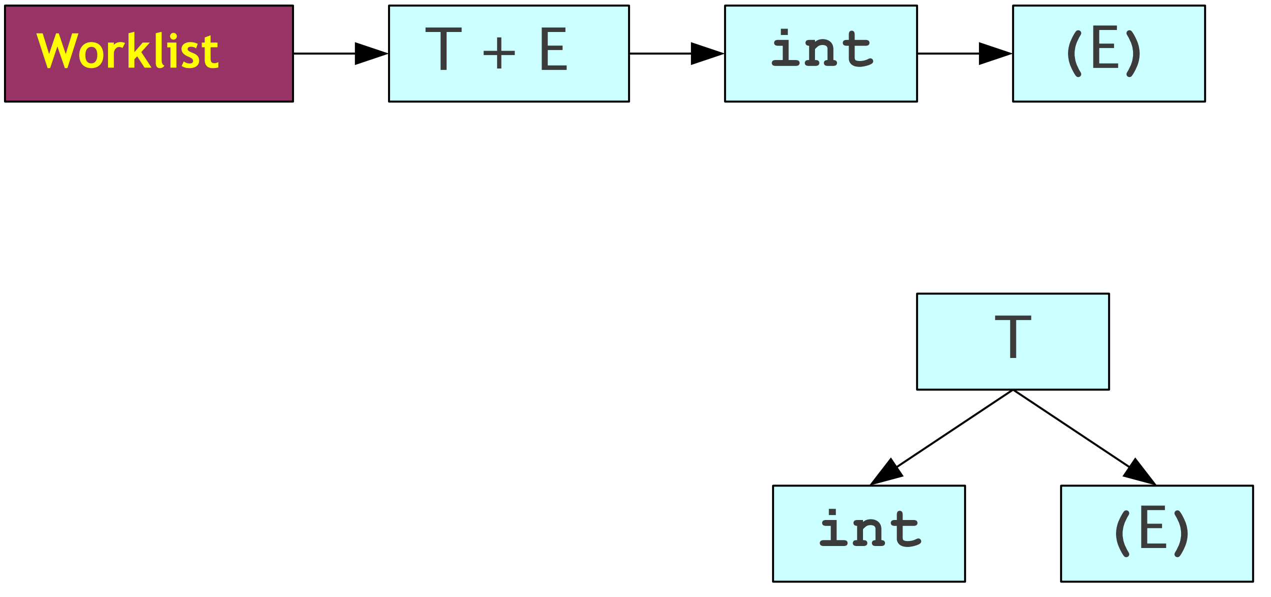
E → T + E

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

Worklist → T + E → `int` → (E)

E → T

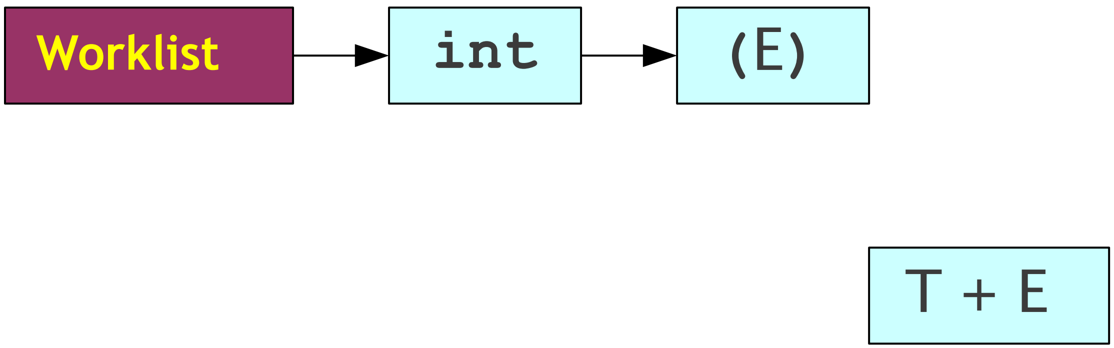
E → T + E

T → `int`

T → (E)

`int + int`

Breadth-First Search Parsing



$E \rightarrow T$

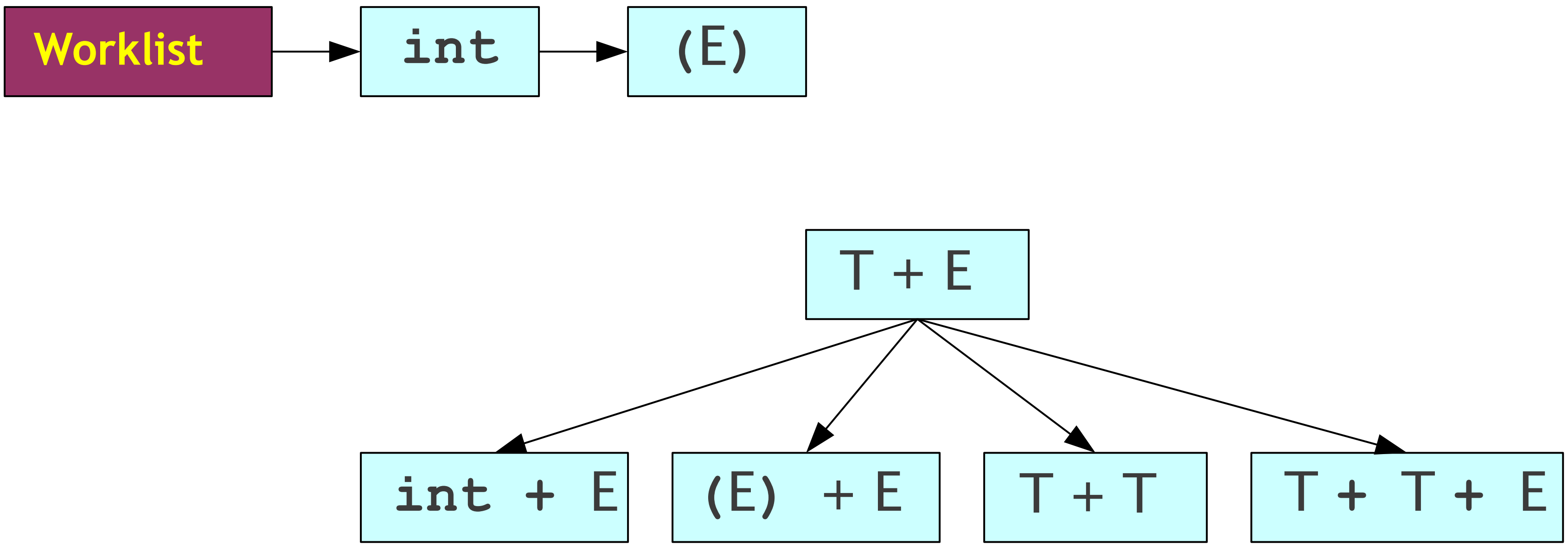
$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

`int + int`

Breadth-First Search Parsing



$E \rightarrow T$

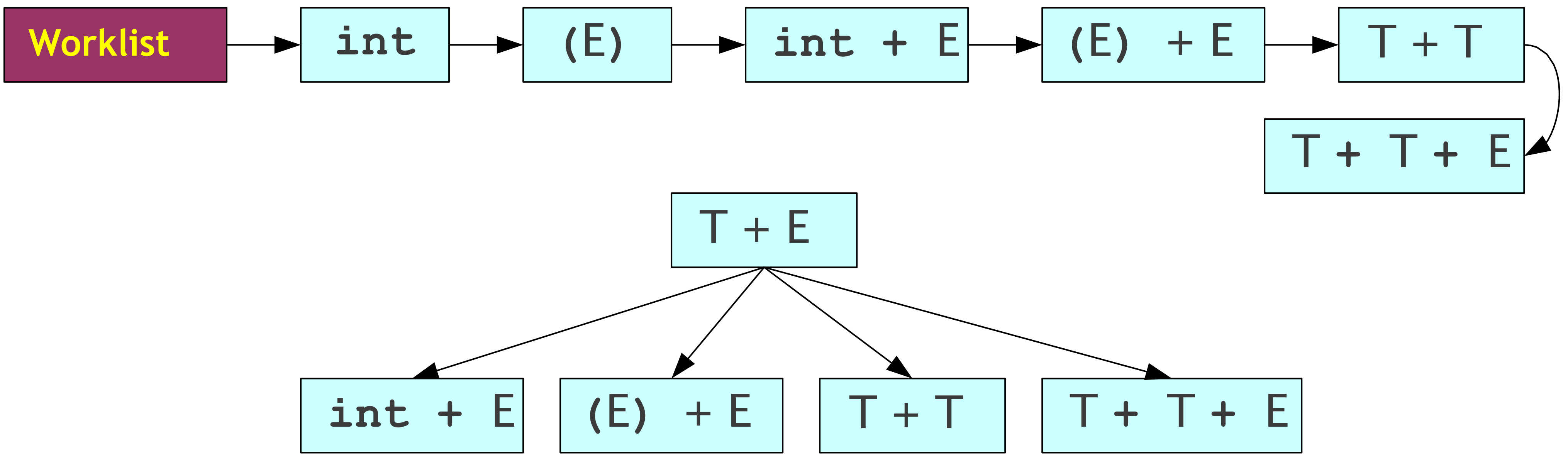
$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow (E)$

`int + int`

Breadth-First Search Parsing



$E \rightarrow T$

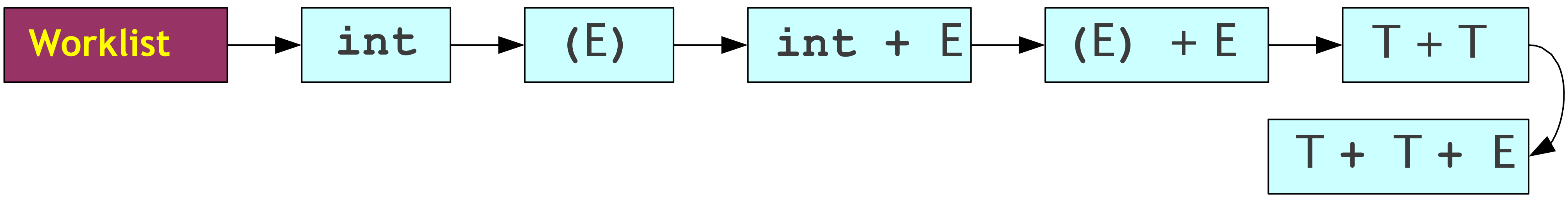
$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$

int + int

Breadth-First Search Parsing



$E \rightarrow T$

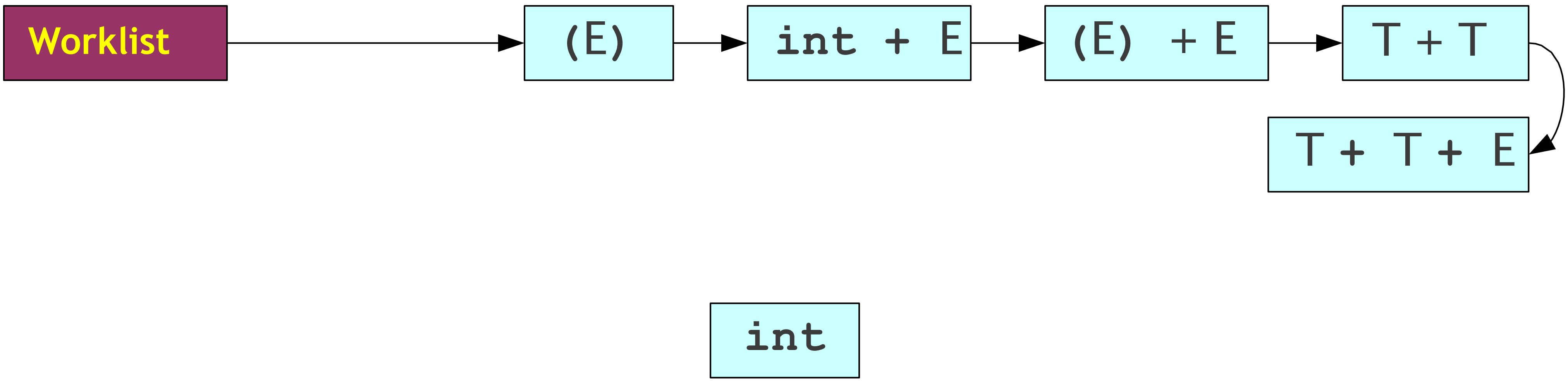
$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$

int + int

Breadth-First Search Parsing



$E \rightarrow T$

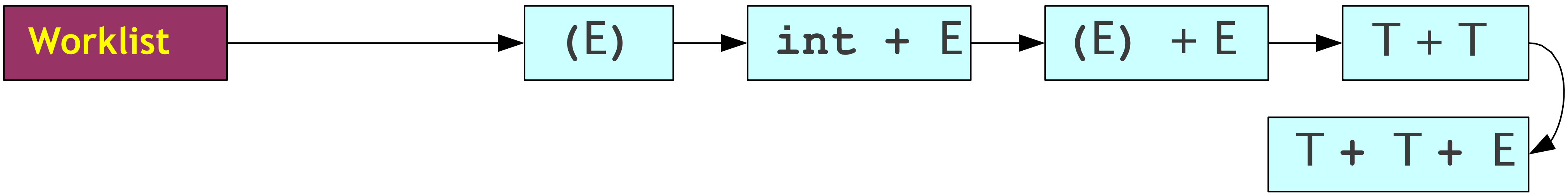
$E \rightarrow T$ **+** E

$T \rightarrow$ **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

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

(E)

E → T

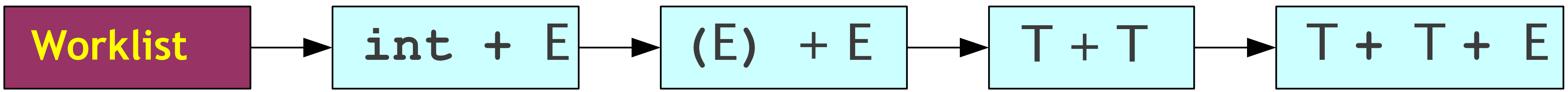
E → T + E

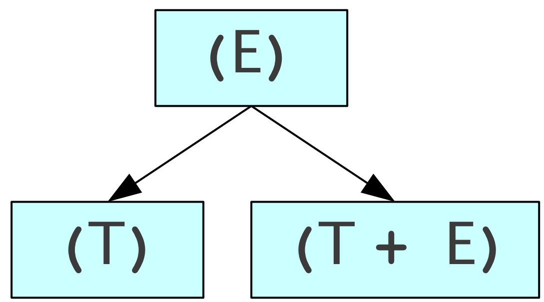
T → int

T → (E)

int + int

Breadth-First Search Parsing





E → T

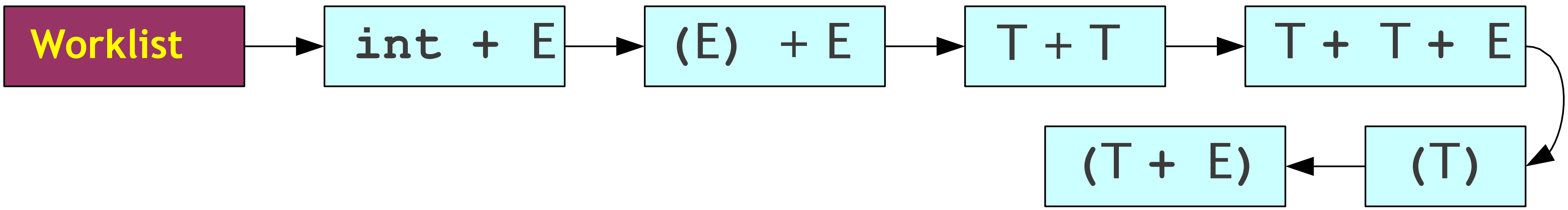
E → T + E

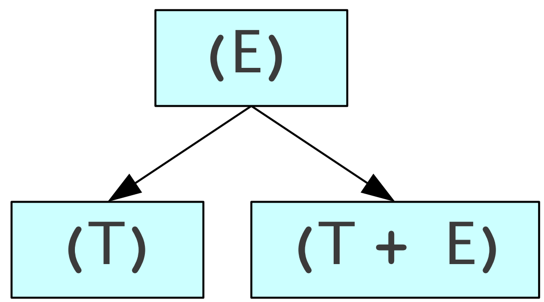
T → int

T → (E)

int + int

Breadth-First Search Parsing





E → T

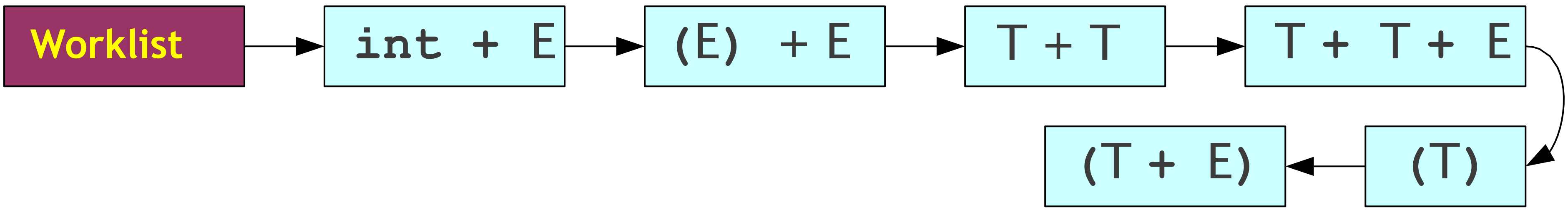
E → T + E

T → int

T → (E)

int + int

Breadth-First Search Parsing



$E \rightarrow T$

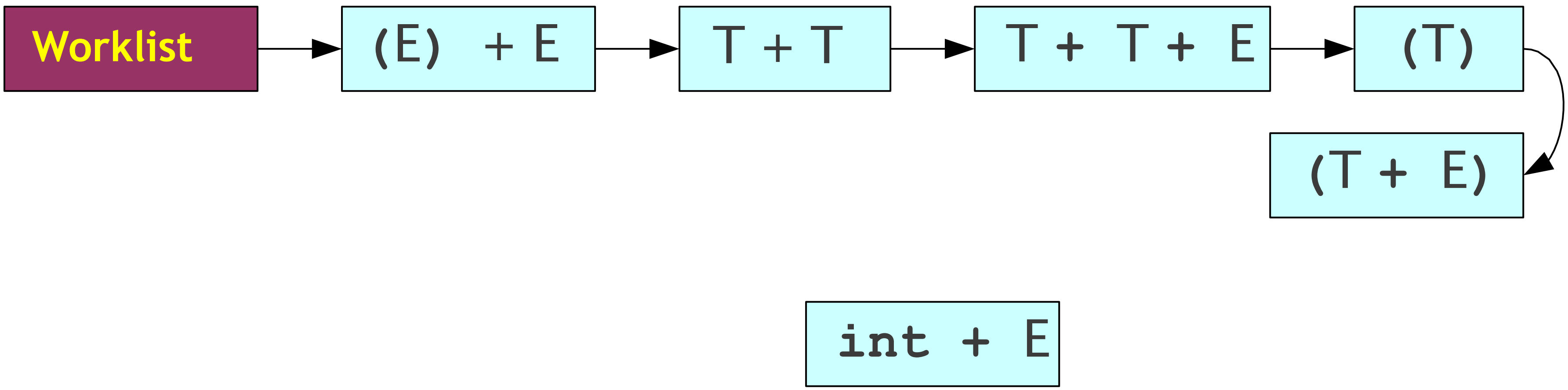
$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

`int + int`

Breadth-First Search Parsing



$E \rightarrow T$

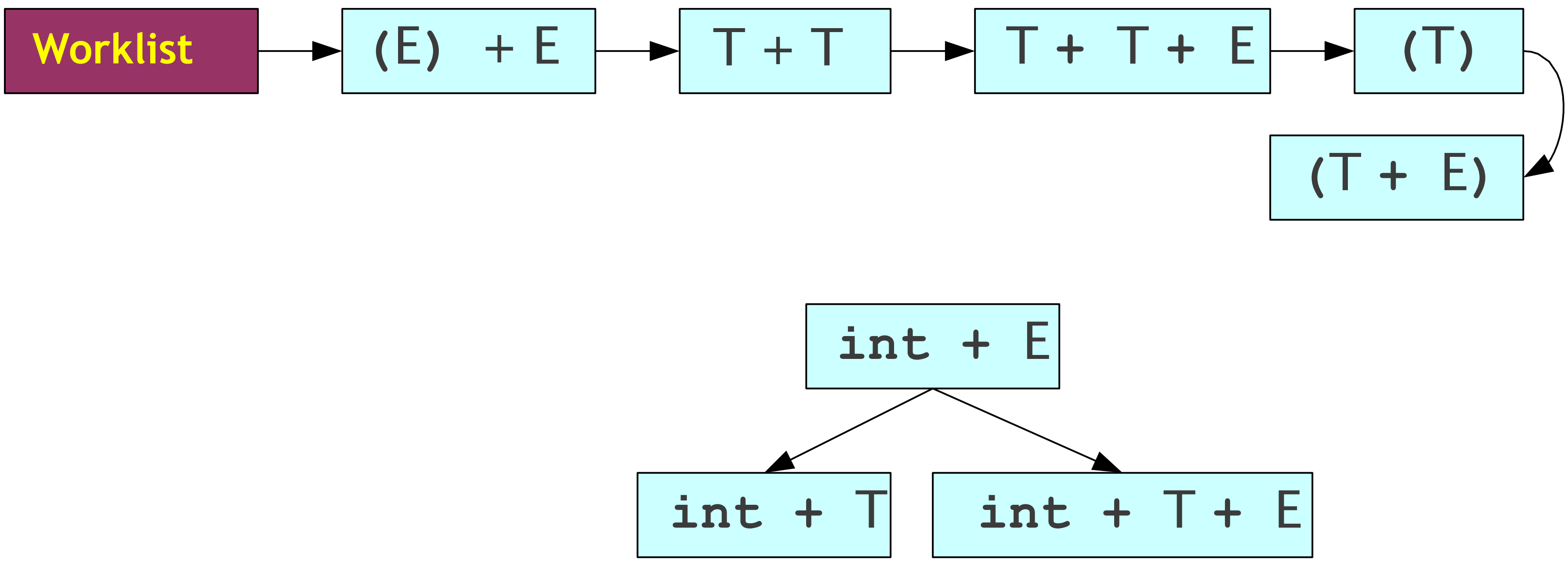
$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

`int + int`

Breadth-First Search Parsing



E → T

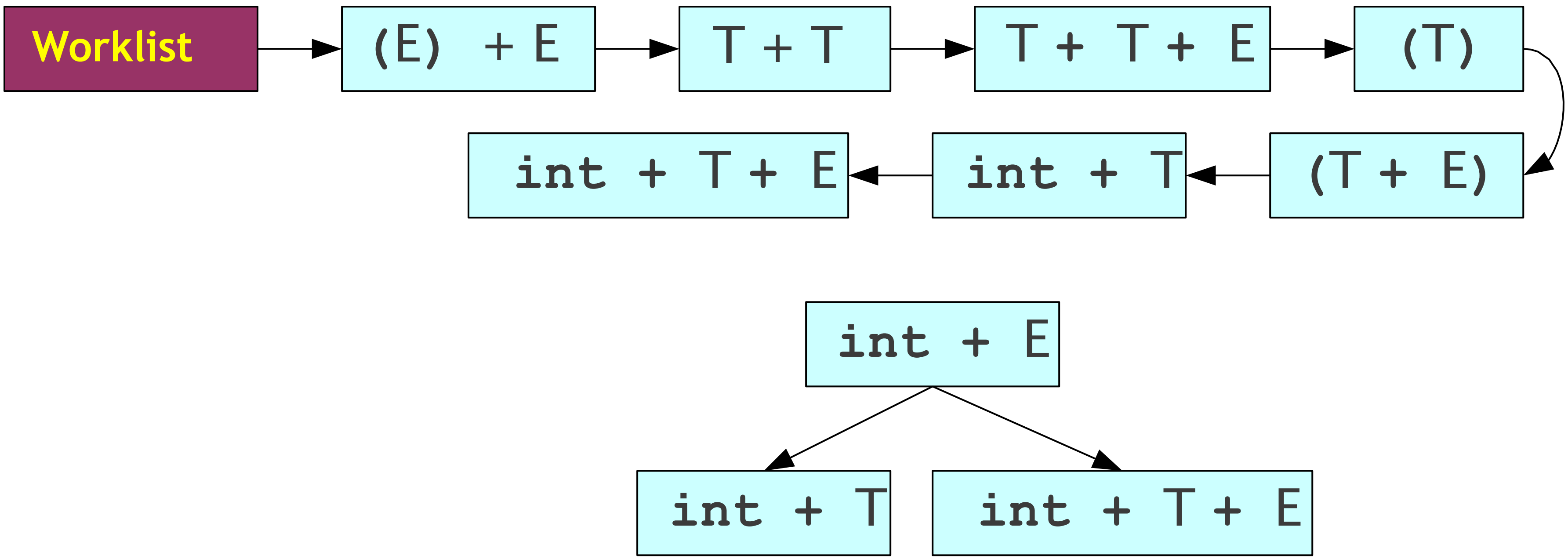
E → T + E

T → int

T → (E)

int + int

Breadth-First Search Parsing



$E \rightarrow T$

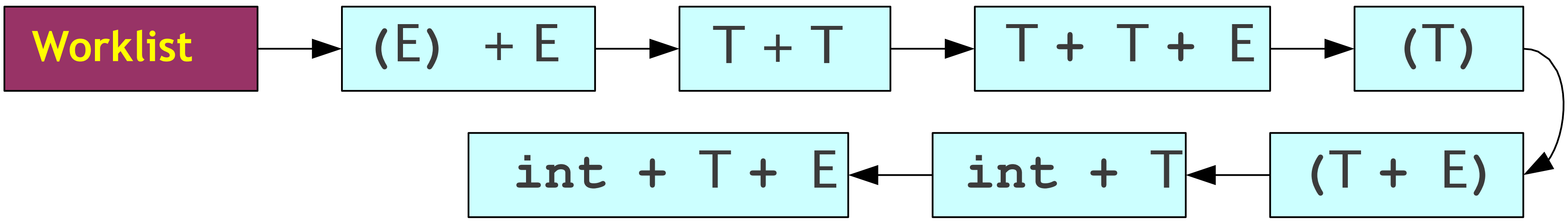
$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$

int + int

Breadth-First Search Parsing



$E \rightarrow T$

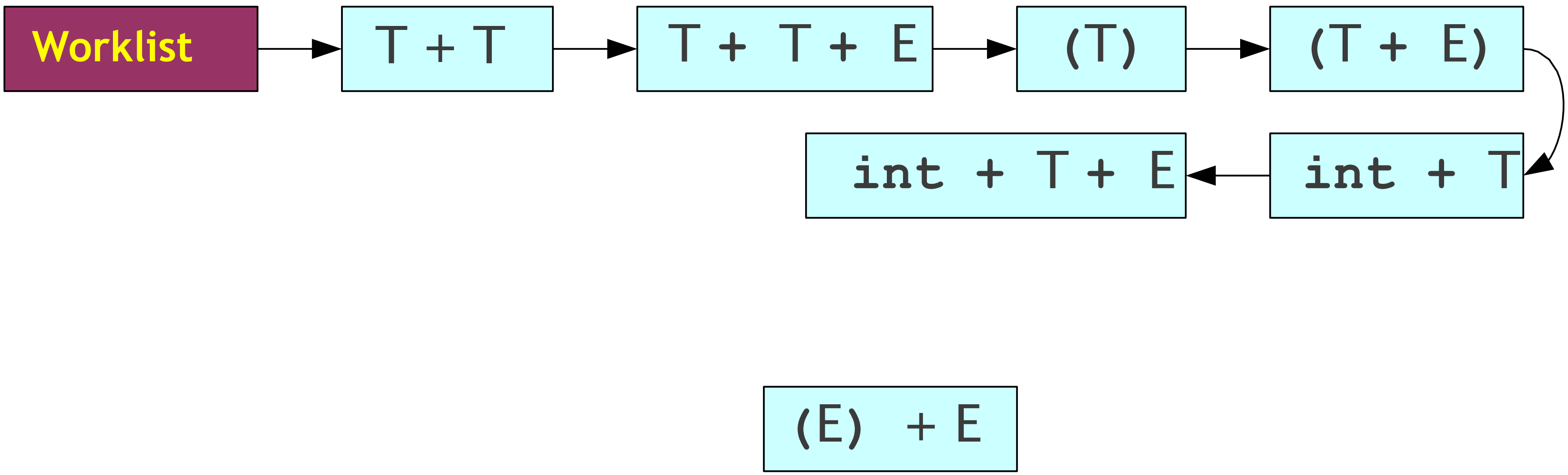
$E \rightarrow T + E$

$T \rightarrow int$

$T \rightarrow (E)$

int + int

Breadth-First Search Parsing



E → T

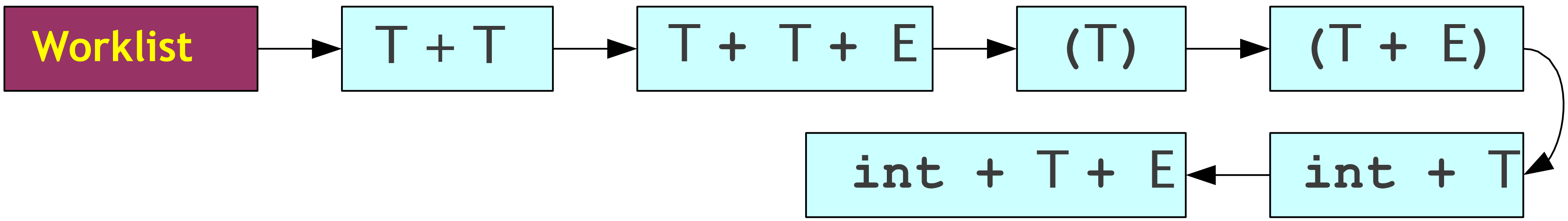
E → T + E

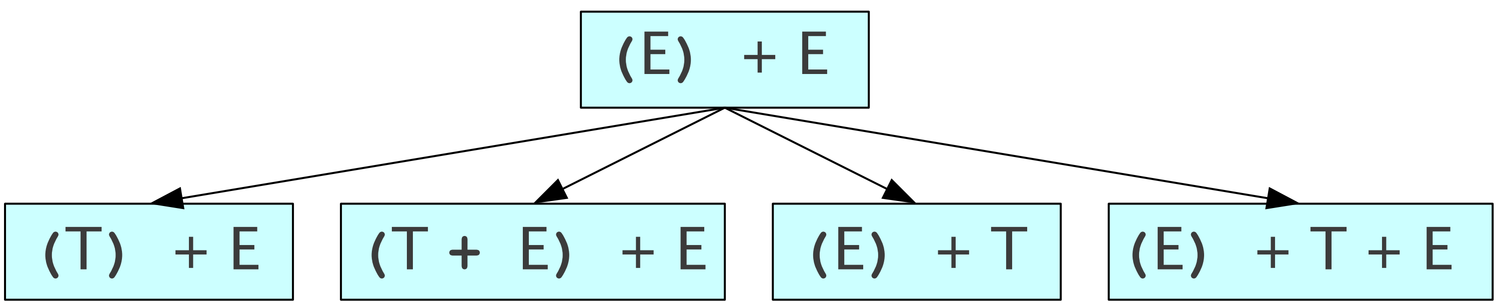
T → int

T → (E)

int + int

Breadth-First Search Parsing





E → T

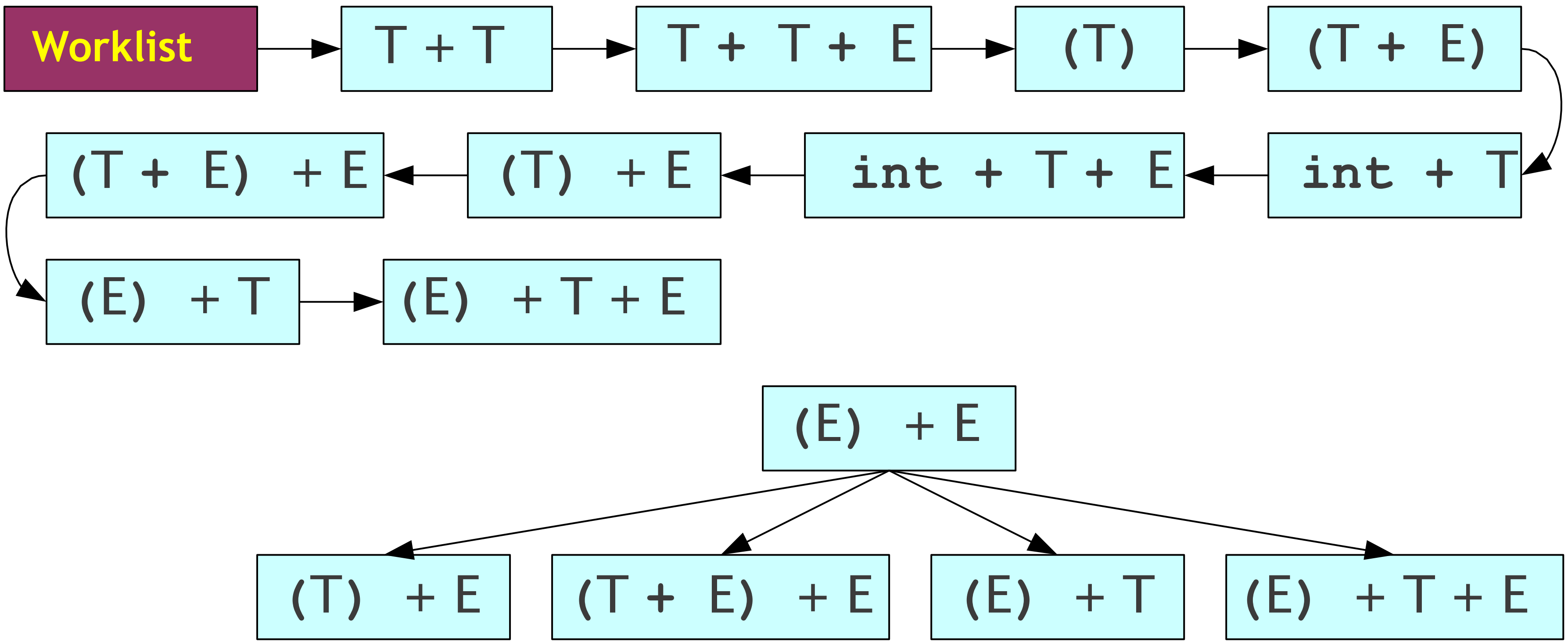
E → T + E

T → int

T → (E)

int + int

Breadth-First Search Parsing



E → T

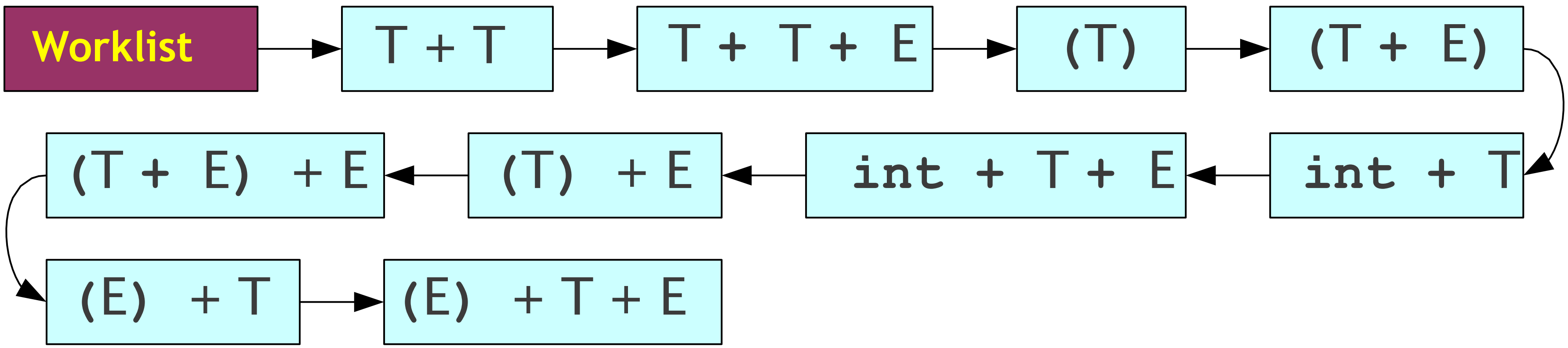
E → T + E

T → int

T → (E)

int + int

Breadth-First Search Parsing



$E \rightarrow T$

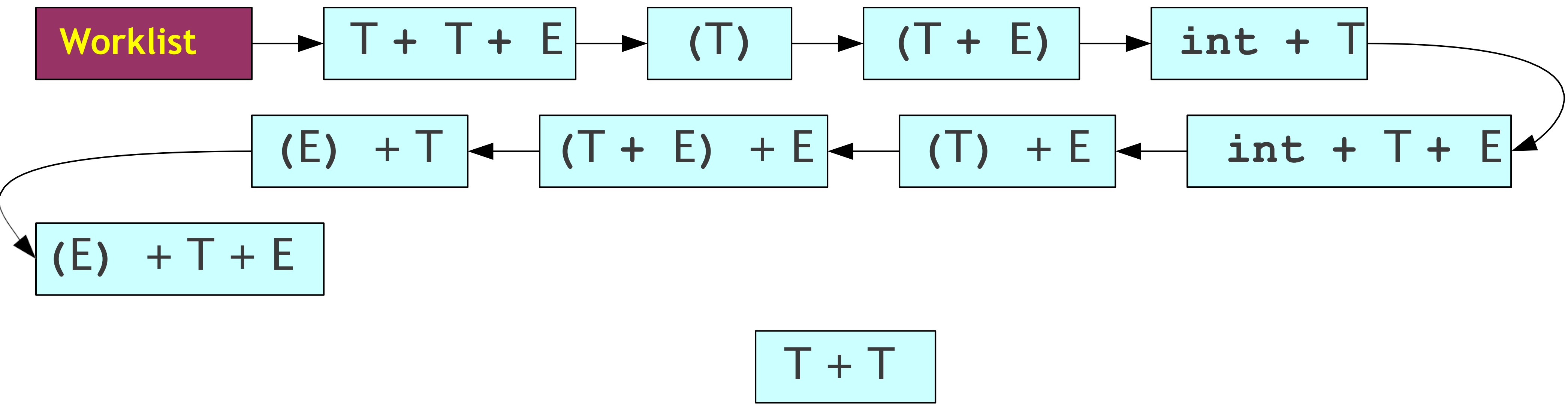
$E \rightarrow T$ **+** E

$T \rightarrow$ **int**

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

int + int

Breadth-First Search Parsing



E → T

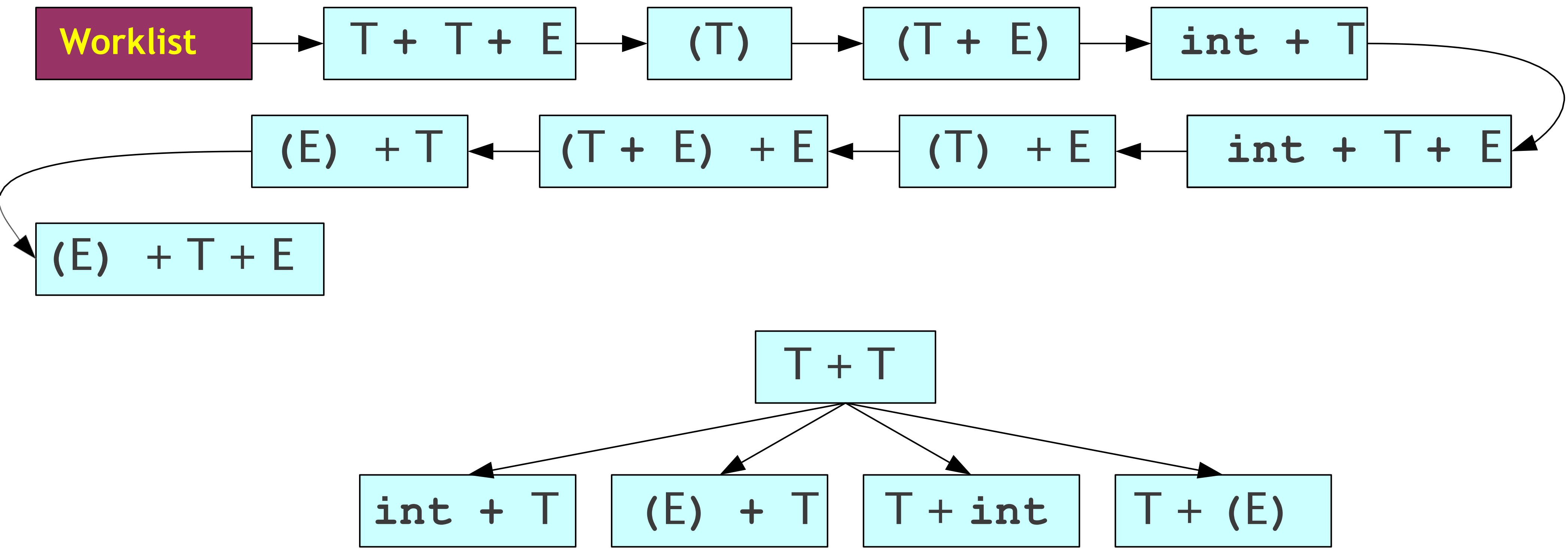
E → T + E

T → int

T → (E)

int + int

Breadth-First Search Parsing



E → T

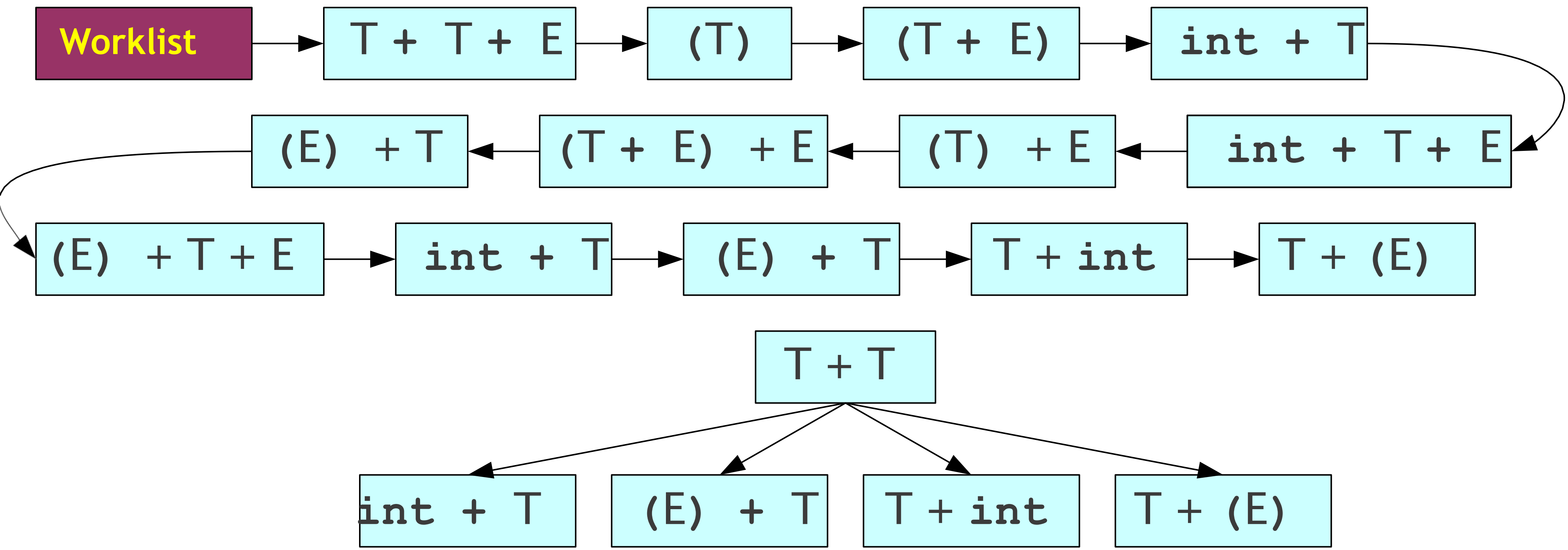
E → T + E

T → int

T → (E)

int + int

Breadth-First Search Parsing



$E \rightarrow T$

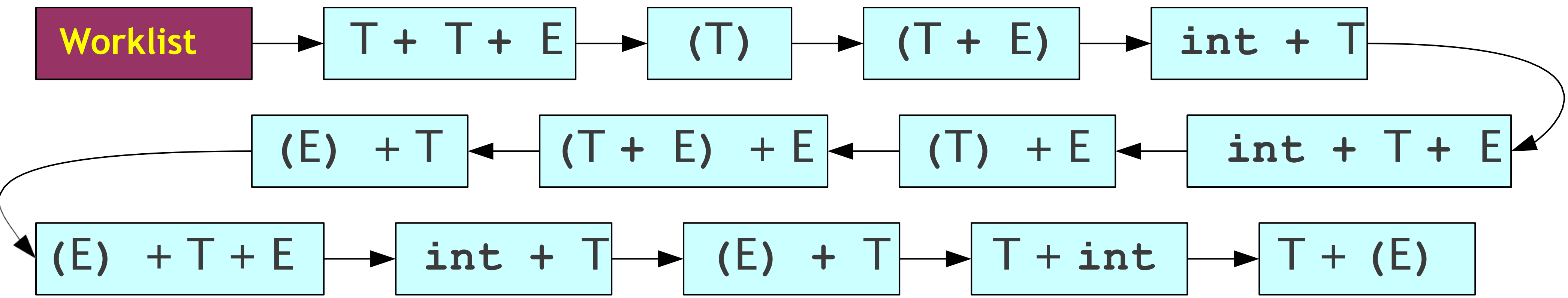
$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$

int + int

Breadth-First Search Parsing



$E \rightarrow T$

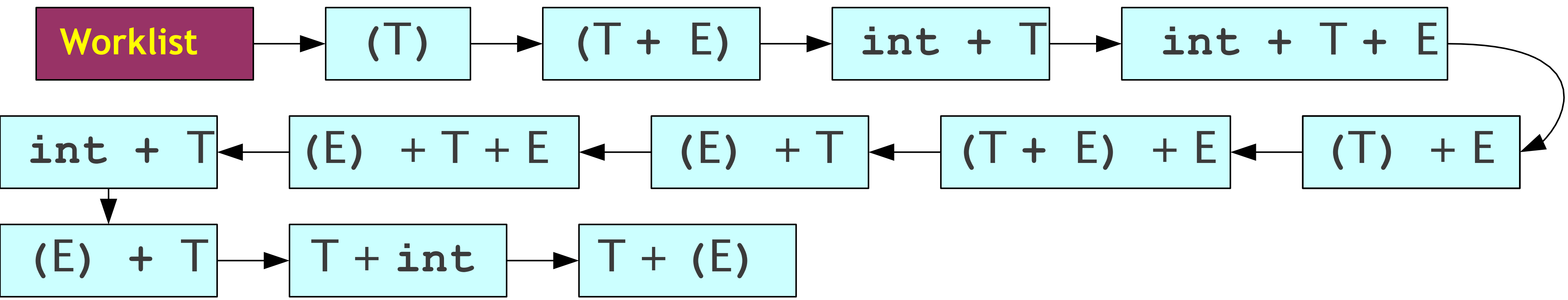
$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$

int + int

Breadth-First Search Parsing



T + T + E

$E \rightarrow T$

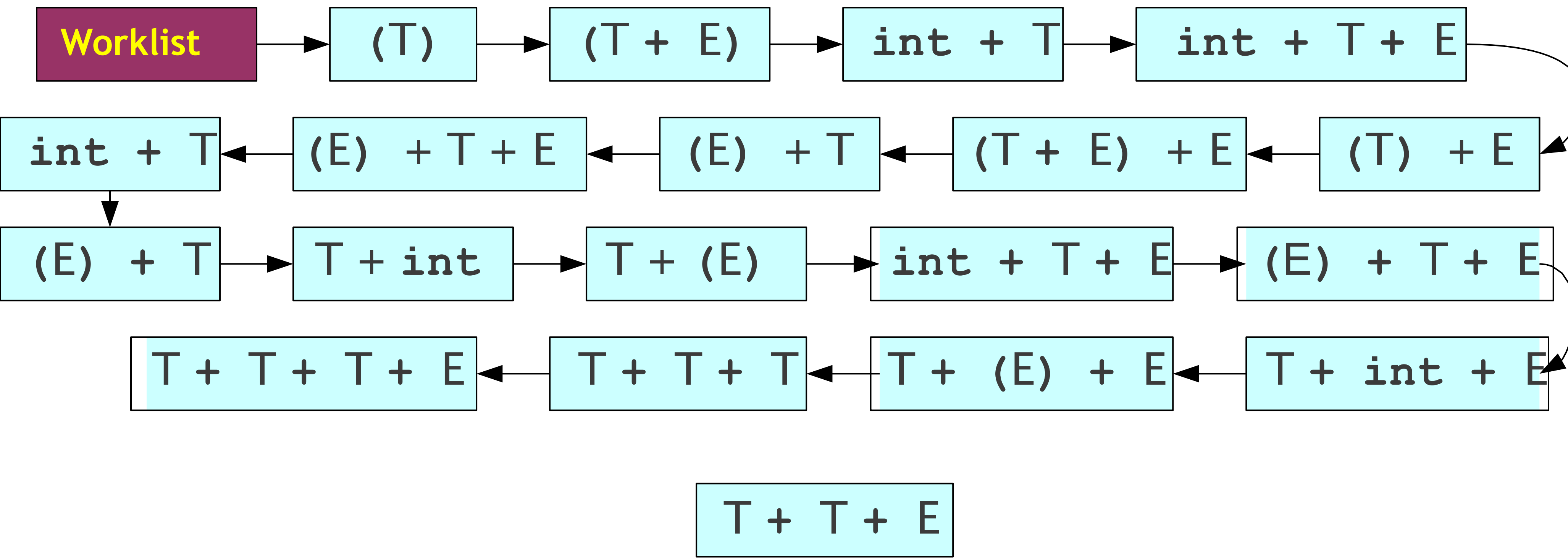
$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow (E)$

`int + int`

Breadth-First Search Parsing



E → T

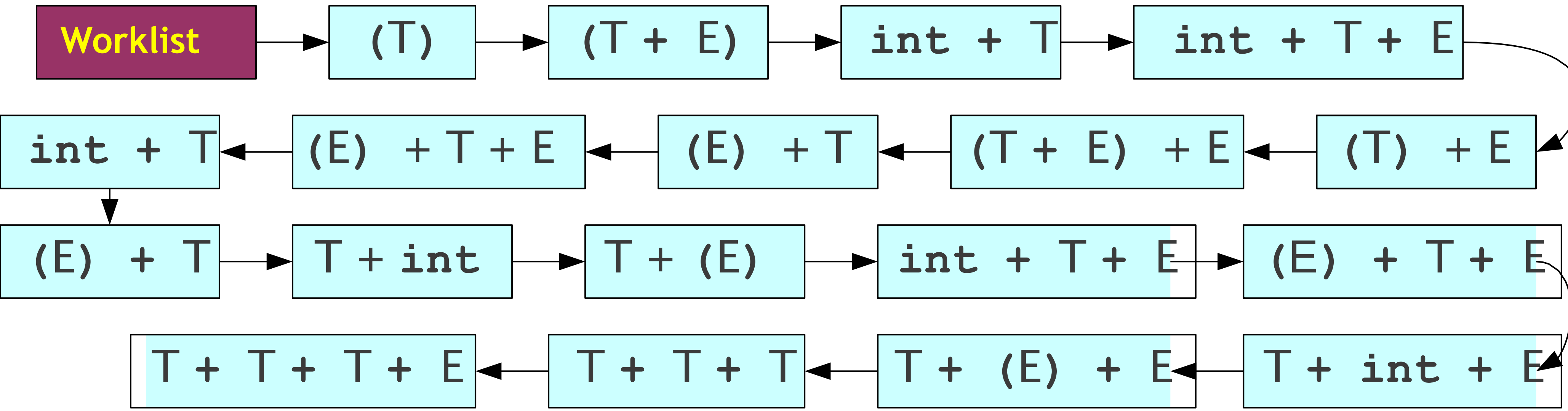
E → T + E

T → int

T → (E)

int + int

Breadth-First Search Parsing



E → T

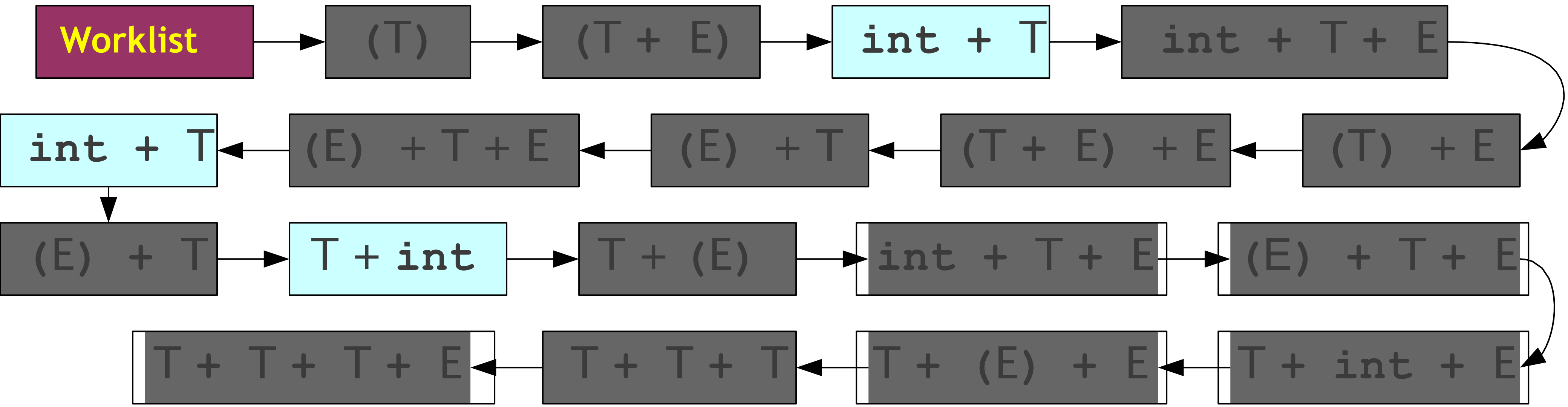
E → T + E

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

BFS is Slow

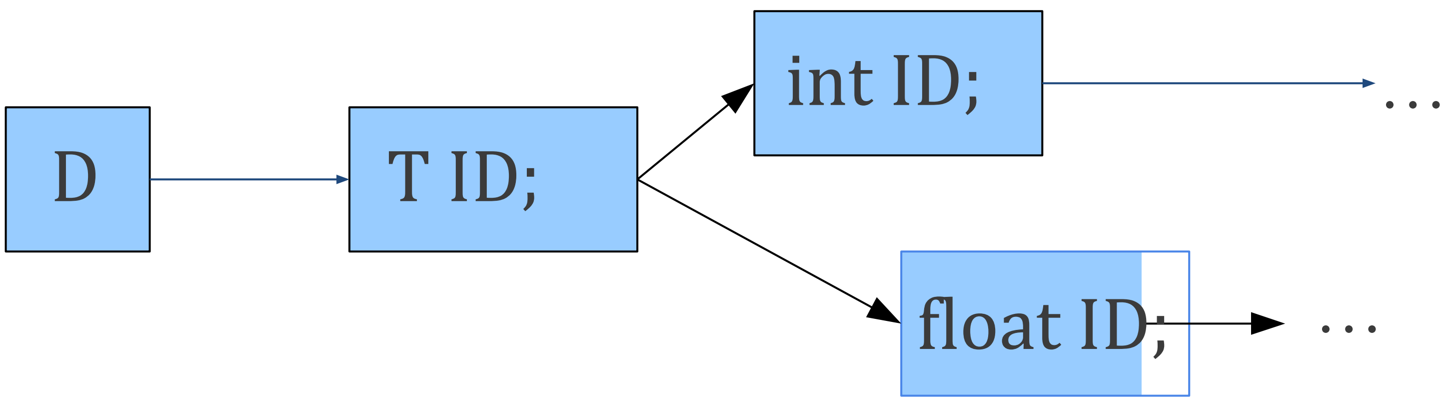
- Enormous time and memory usage:
  - Lots of **wasted effort**:
    - Generates a lot of sentential forms that couldn't possibly match.
    - But in general, extremely hard to tell whether a sentential form can match – that's the job of parsing!
  - High **branching factor**:
    - Each sentential form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

Reducing Wasted Effort

D → T ID;

T -> int|float

float abc;



Reducing Wasted Effort

- Suppose we're trying to match a string γ.
- Suppose we have a sentential form $\tau = \alpha\omega$, where α is a string of terminals and ω is a string of terminals and nonterminals.
- If α isn't a prefix of γ, then no string derived from τ can ever match γ.
- 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

E → T

E → T + E

T → int

T → (E)

`int + int`

Leftmost BFS

Worklist

E

$E \rightarrow T$

$E \rightarrow T + E$

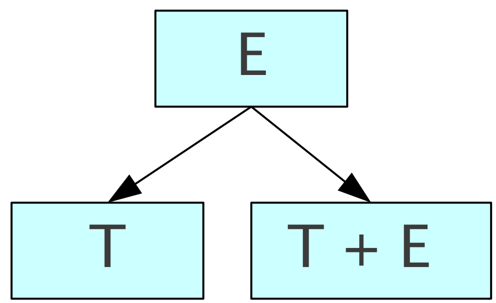
$T \rightarrow$ `int`

$T \rightarrow$ `(E)`

`int + int`

Leftmost BFS

Worklist



$E \rightarrow T$

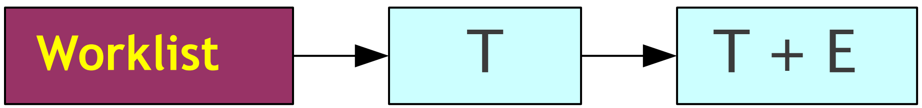
$E \rightarrow T + E$

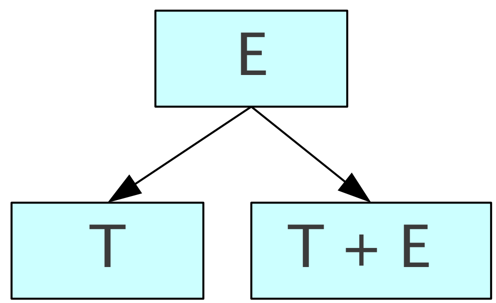
$T \rightarrow$ `int`

$T \rightarrow (E)$

`int + int`

Leftmost BFS





$E \rightarrow T$

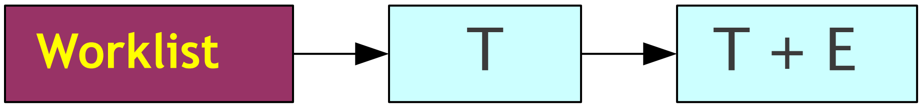
$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow$ `(E)`

`int + int`

Leftmost BFS



E → T

E → T + E

T → int

T → (E)

`int + int`

Leftmost BFS

Worklist → T + E

T

E → T

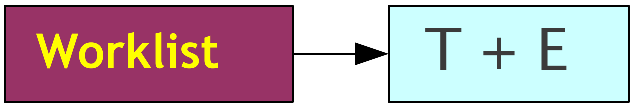
E → T + E

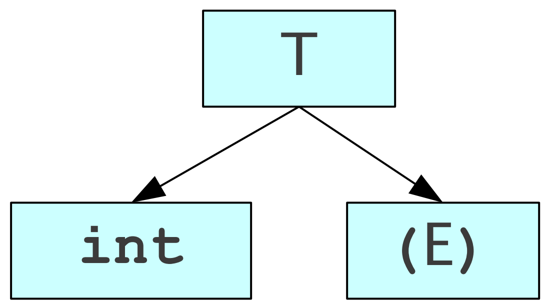
T → int

T → (E)

int + int

Leftmost BFS





$E \rightarrow T$

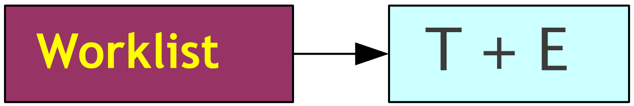
$E \rightarrow T + E$

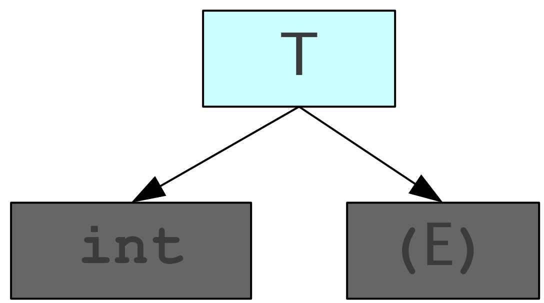
$T \rightarrow$ **int**

$T \rightarrow (E)$

int + int

Leftmost BFS





$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow (E)$

`int + int`

Leftmost BFS

Worklist → T + E

$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

`int + int`

Leftmost BFS

Worklist

T + E

$E \rightarrow T$

$E \rightarrow T + E$

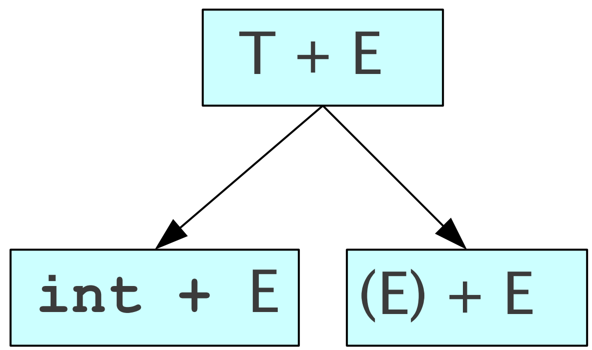
$T \rightarrow$ `int`

$T \rightarrow$ `(E)`

`int + int`

Leftmost BFS

Worklist



$E \rightarrow T$

$E \rightarrow T + E$

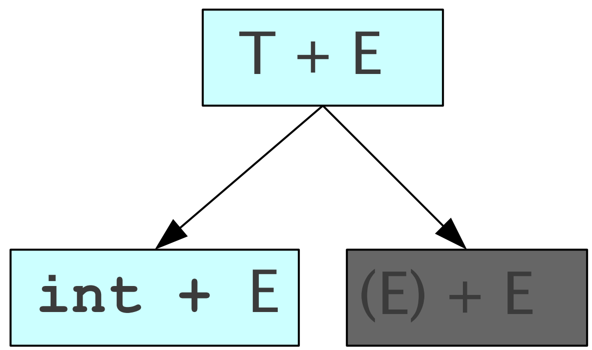
$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

`int + int`

Leftmost BFS

Worklist



$E \rightarrow T$

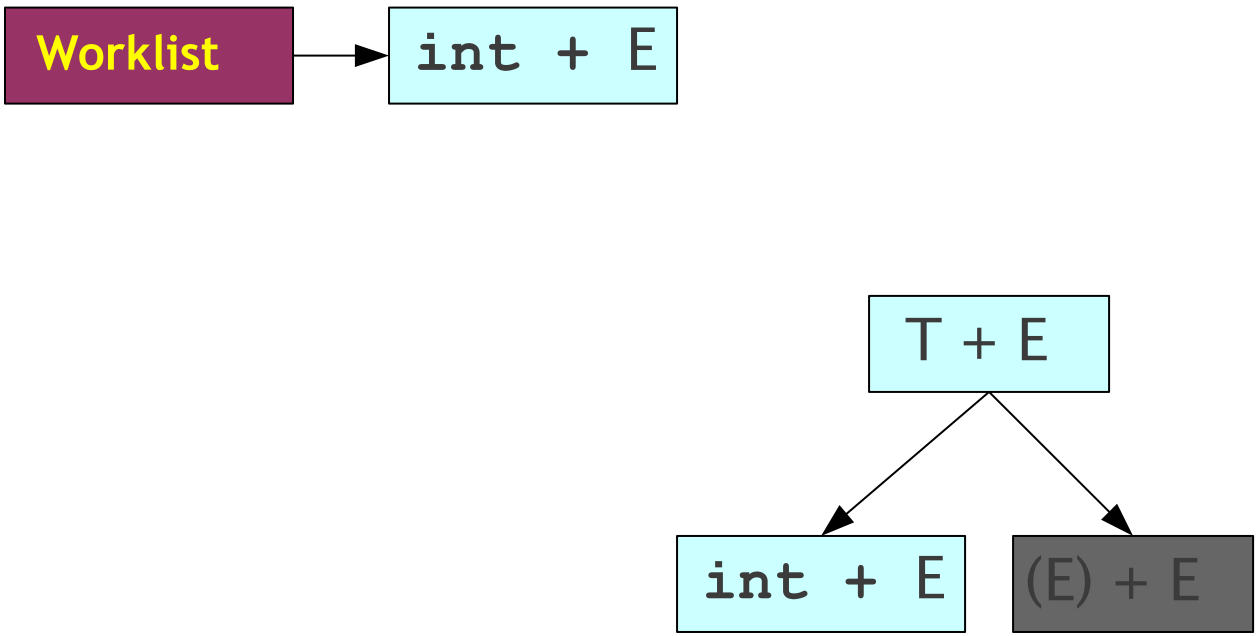
$E \rightarrow T$ **+** E

$T \rightarrow$ **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

Worklist → `int + E`

$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow$ `(E)`

`int + int`

Leftmost BFS

Worklist

`int + E`

$E \rightarrow T$

$E \rightarrow T + E$

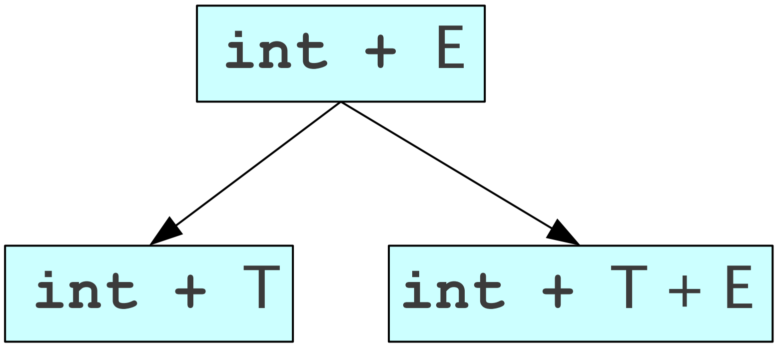
$T \rightarrow$ `int`

$T \rightarrow (E)$

`int + int`

Leftmost BFS

Worklist



E → T

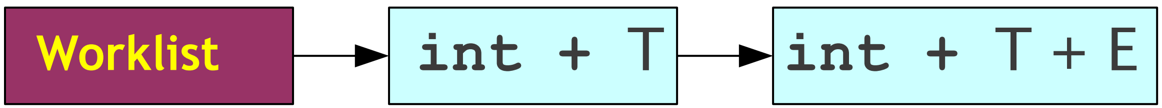
E → T + E

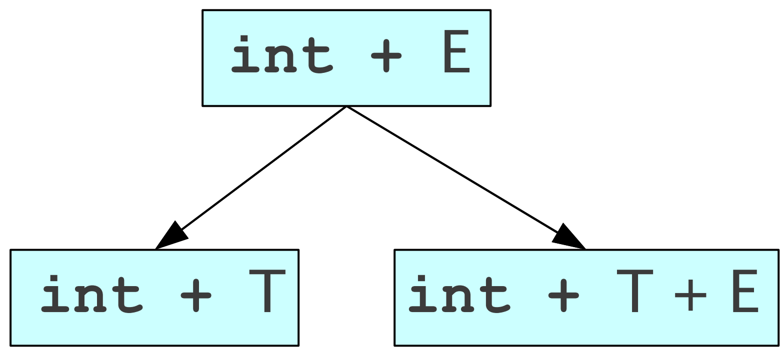
T → int

T → (E)

int + int

Leftmost BFS





E → T

E → T + E

T → int

T → (E)

int + int

Leftmost BFS

Worklist → `int + T` → `int + T + E`

E → T

E → T + E

T → `int`

T → (E)

`int + int`

Leftmost BFS

Worklist → `int + T + E`

`int + T`

$E \rightarrow T$

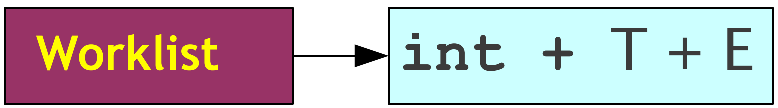
$E \rightarrow T + E$

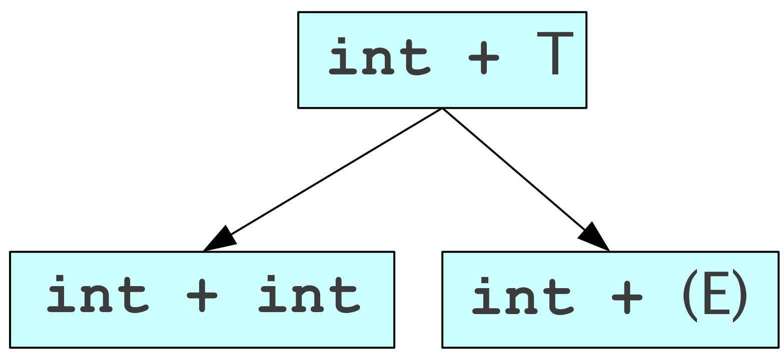
$T \rightarrow$ `int`

$T \rightarrow (E)$

`int + int`

Leftmost BFS





$E \rightarrow T$

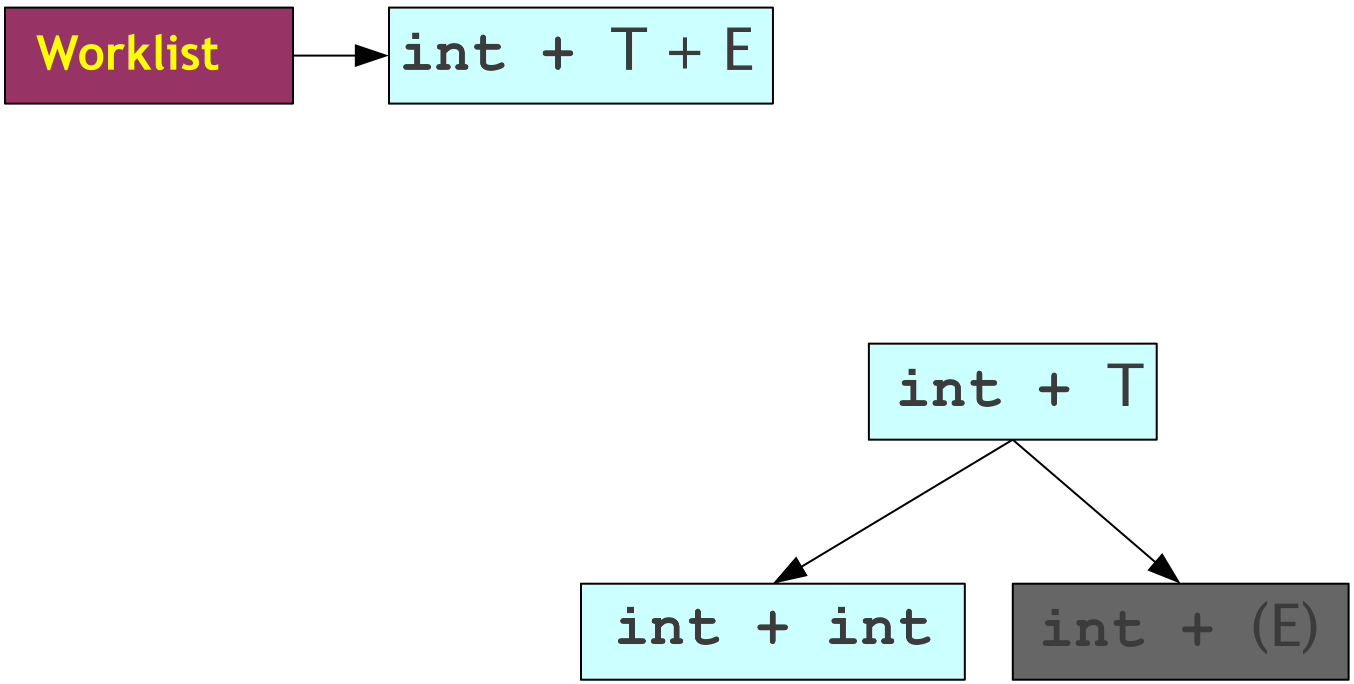
$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$

int + int

Leftmost BFS



$E \rightarrow T$

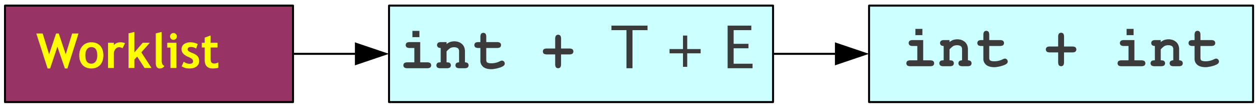
$E \rightarrow T + E$

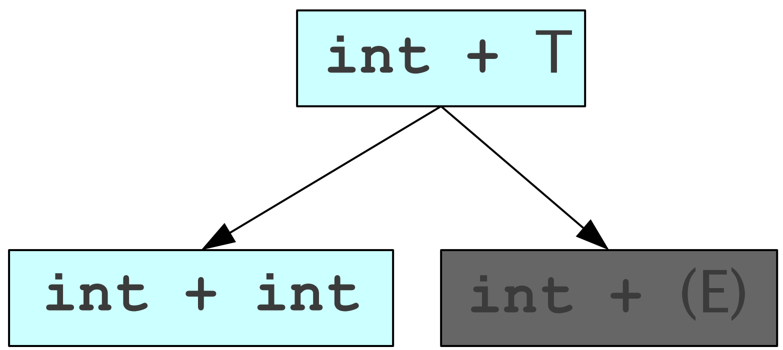
$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

int + int

Leftmost BFS





$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$

int + int

Leftmost BFS

Worklist → `int +` T + E → `int + int`

E → T

E → T **+** E

T → **int**

T → **(**E**)**

`int + int`

Leftmost BFS

Worklist → `int + int`

`int +` T + E

$E \rightarrow T$

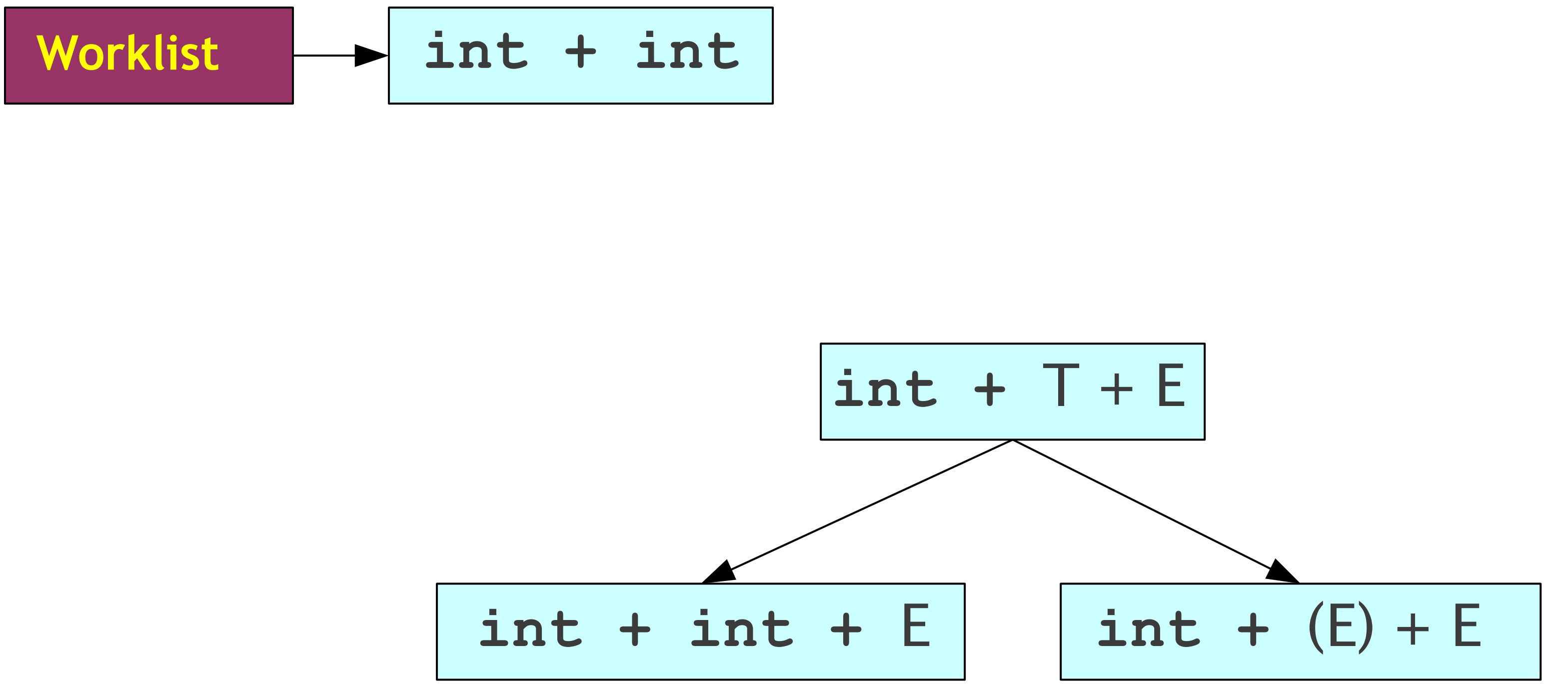
$E \rightarrow T$ **+** E

$T \rightarrow$ **int**

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

`int + int`

Leftmost BFS



$E \rightarrow T$

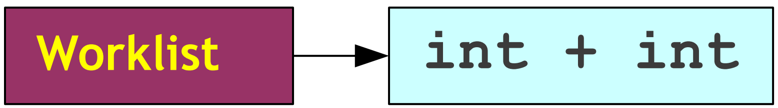
$E \rightarrow T + E$

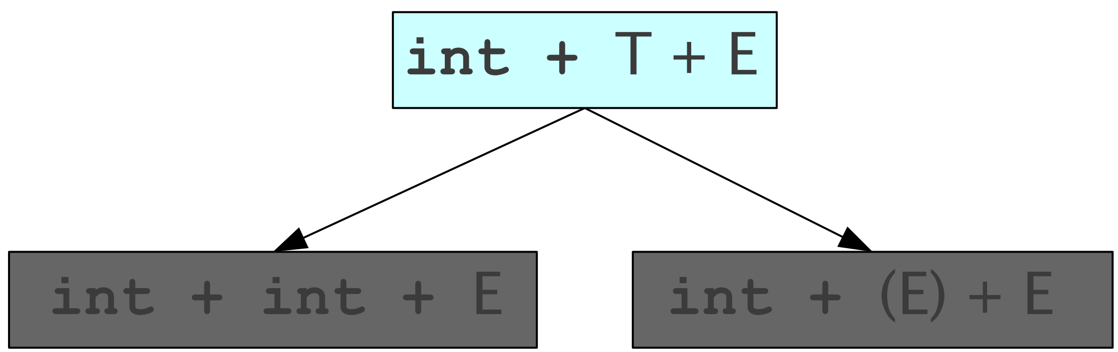
$T \rightarrow$ int

$T \rightarrow (E)$

int + int

Leftmost BFS





$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$

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`

$E \to T$

$E \to T + E$

$T \to$ int

$T \to$ (E)

int + int

Leftmost BFS

- Substantial improvement over naïve algorithm.
- Will always find a valid parse of a program if one exists.
- But, there are still problems.

Leftmost BFS Has Problems

Worklist

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

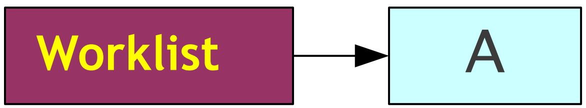
Leftmost BFS Has Problems

Worklist

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

caaaaaaaaaa

Leftmost BFS Has Problems



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

caaaaaaaaaa

Leftmost BFS Has Problems

Worklist

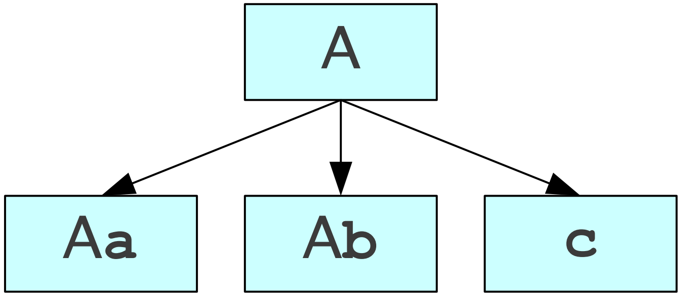
A

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

`caaaaaaaaa`

Leftmost BFS Has Problems

Worklist

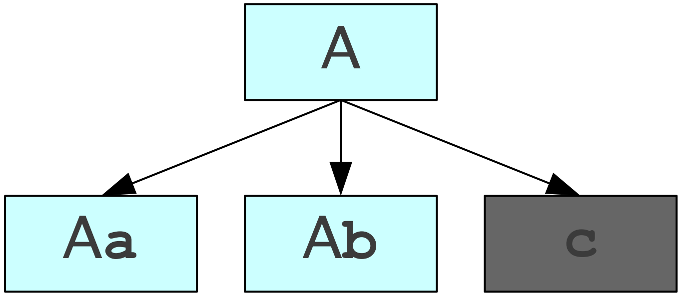


$A \rightarrow Aa \mid Ab \mid$
c

caaaaaaaaa

Leftmost BFS Has Problems

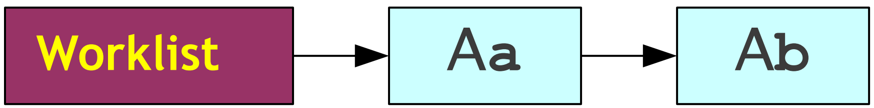
Worklist

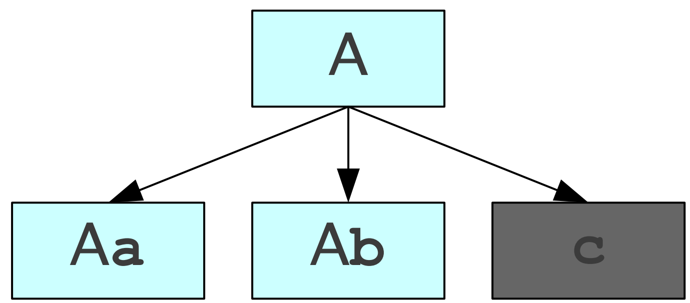


A → Aa | Ab |
c

caaaaaaaaaa

Leftmost BFS Has Problems

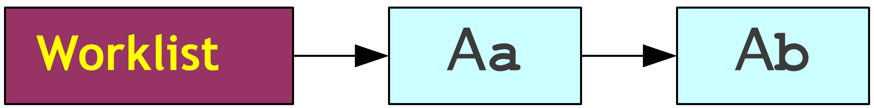




$A \rightarrow$ Aa | Ab |
c

caaaaaaaaaa

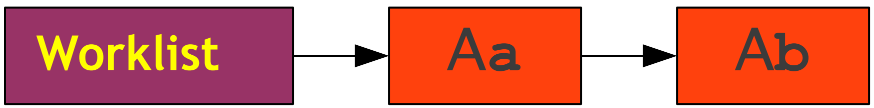
Leftmost BFS Has Problems



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

caaaaaaaaaa

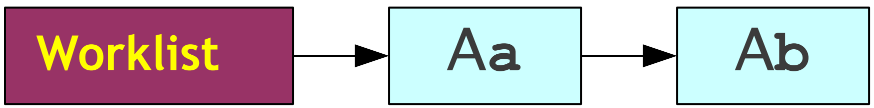
Leftmost BFS Has Problems



A → Aa | Ab |
c

caaaaaaaaaa

Leftmost BFS Has Problems



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

caaaaaaaaaa

Leftmost BFS Has Problems

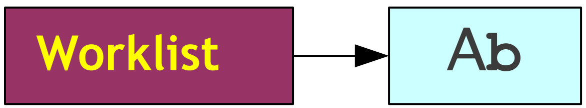
Worklist → Ab

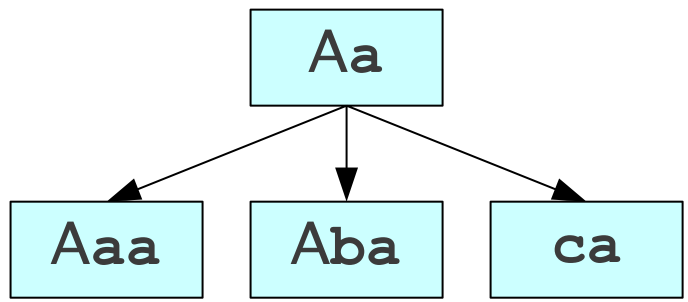
Aa

A → Aa | Ab |
c

caaaaaaaaaa

Leftmost BFS Has Problems

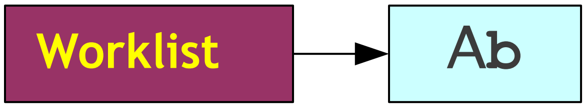


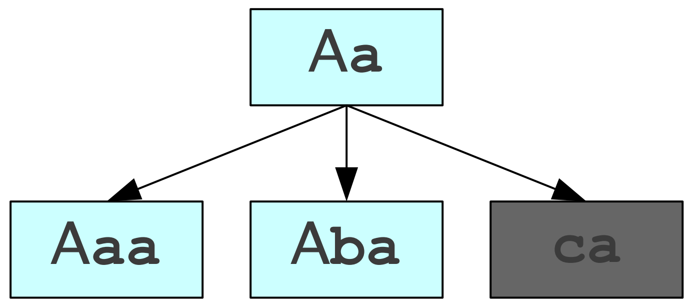


A → Aa | Ab | c

caaaaaaaaaa

Leftmost BFS Has Problems

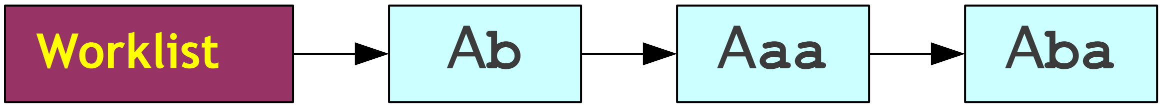


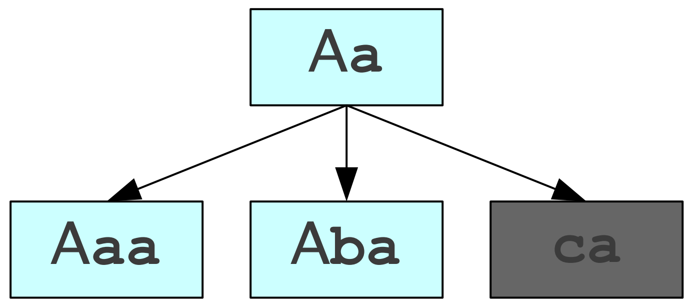


A → Aa | Ab |
c

caaaaaaaaaa

Leftmost BFS Has Problems

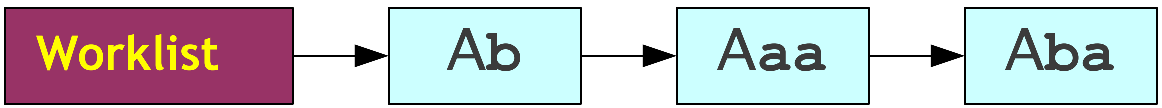




A → Aa | Ab |
c

caaaaaaaaaa

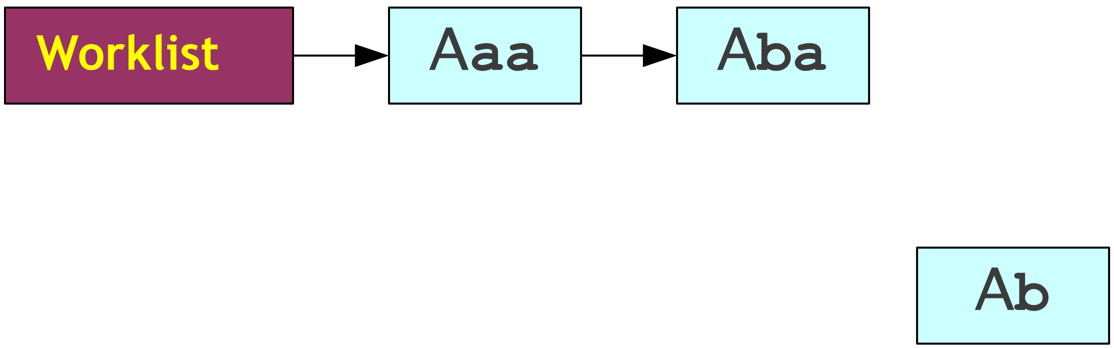
Leftmost BFS Has Problems



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

caaaaaaaaaa

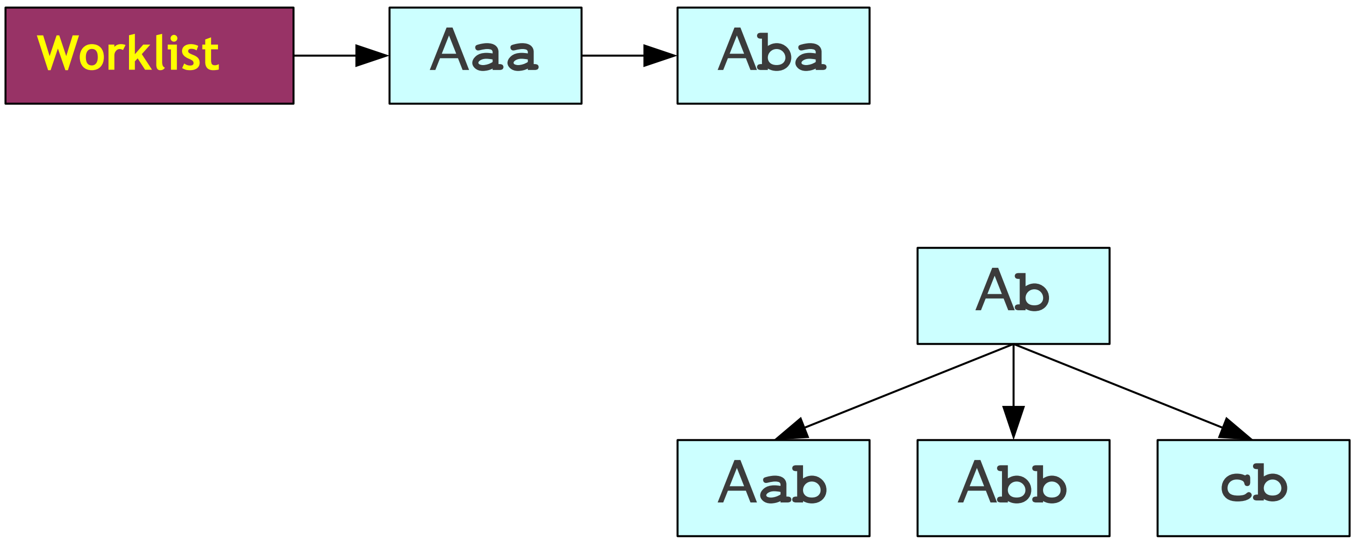
Leftmost BFS Has Problems



$A \rightarrow$ Aa | Ab | c

caaaaaaaaaa

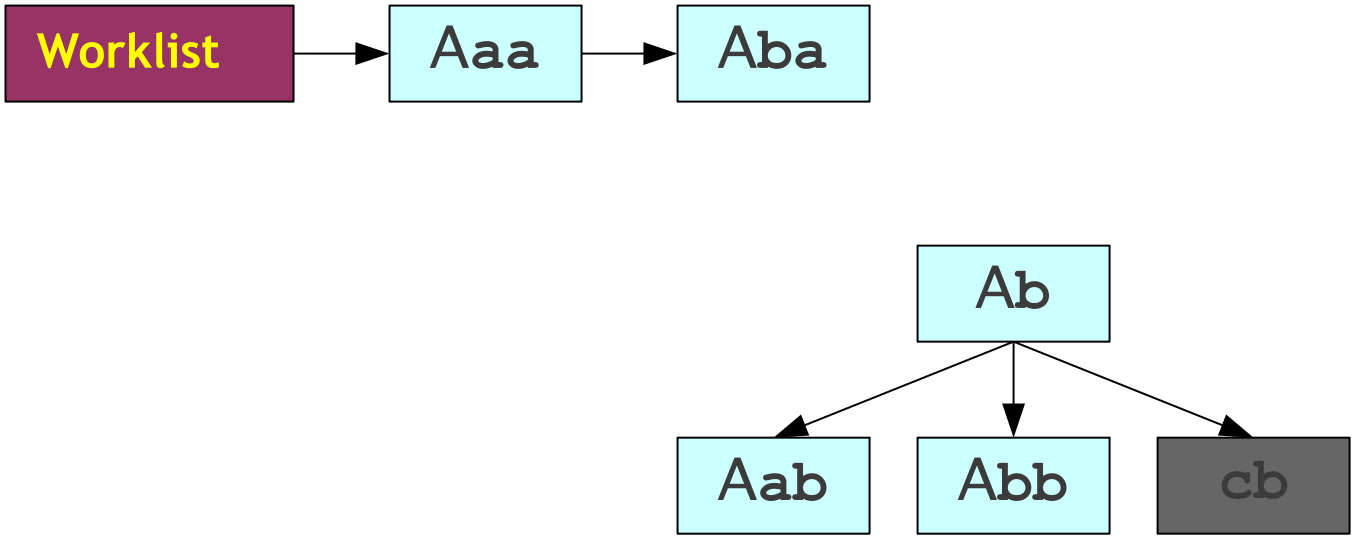
Leftmost BFS Has Problems



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

caaaaaaaaa

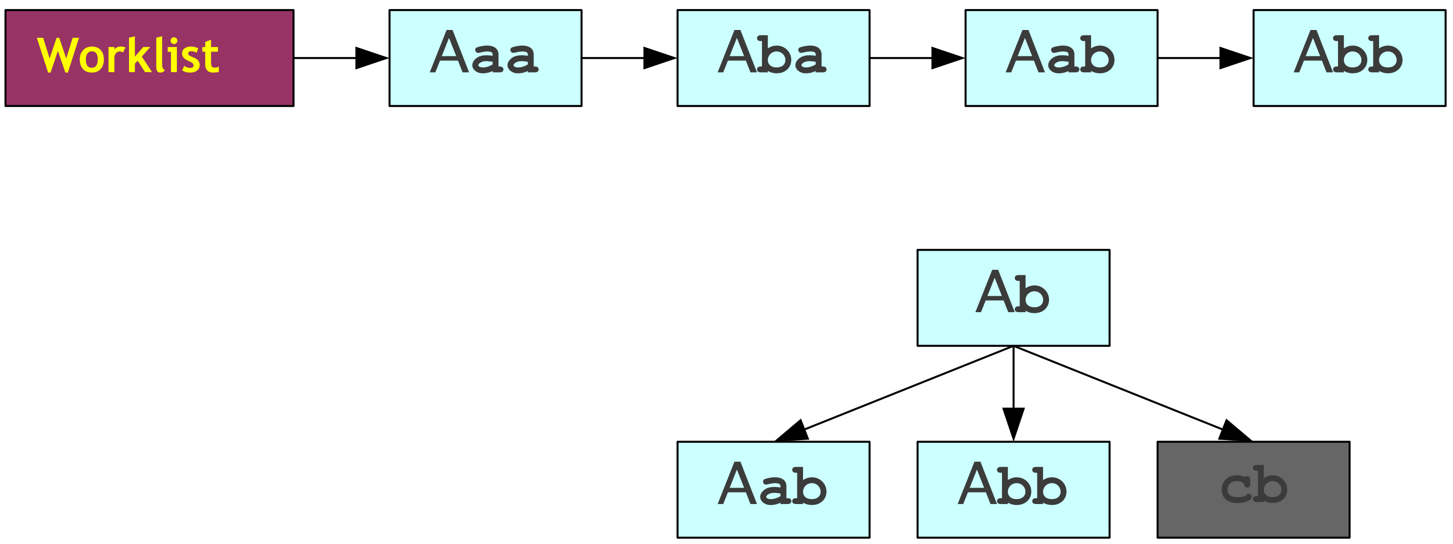
Leftmost BFS Has Problems



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

caaaaaaaaa

Leftmost BFS Has Problems



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

caaaaaaaaa

Leftmost BFS Has Problems

Worklist → Aaa → Aba → Aab → Abb

A → Aa | Ab |
c

caaaaaaaaaa

Leftmost BFS Has Problems

Worklist → Aaa → Aba → Aab → Abb

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

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

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

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

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

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

Leftmost DFS

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

`int + int`

Leftmost DFS

E

$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

`int + int`

Leftmost DFS

E
T

$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

`int + int`

Leftmost DFS

E
T
`int`

$E \rightarrow T$

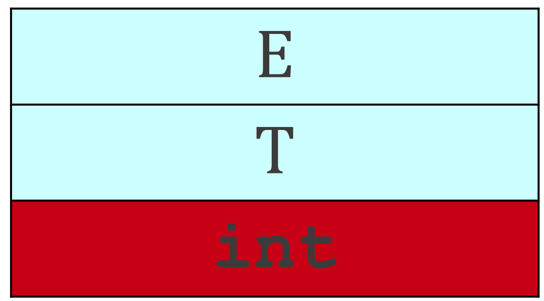
$E \rightarrow T$ `+` E

$T \rightarrow$ `int`

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

`int + int`

Leftmost DFS



E → T

E → T+ E

T → int

T → (E)

int + int

Leftmost DFS

E
T

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

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

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

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

$E \rightarrow T$

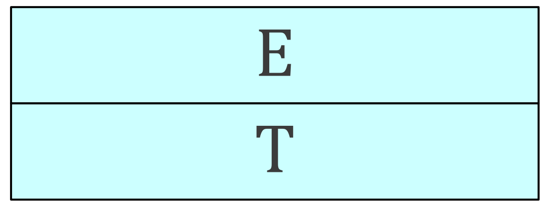
$E \rightarrow T + E$

$T \rightarrow$ int

$T \rightarrow (E)$

int + int

Leftmost DFS



$E \rightarrow T$

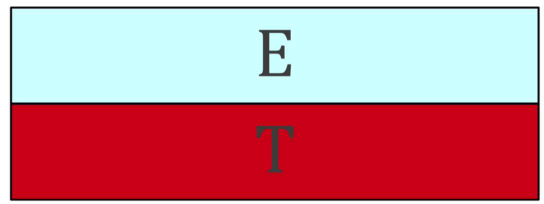
$E \rightarrow T + E$

$T \rightarrow \texttt{int}$

$T \rightarrow (E)$

`int + int`

Leftmost DFS



$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow$ `int`

$T \rightarrow (E)$

`int + int`

Leftmost DFS

E

$E \to T$

$E \to T + E$

$T \to \texttt{int}$

$T \to (E)$

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

$E \to T$

$E \to T + E$

$T \to$ int

$T \to$ (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

Summary of Leftmost BFS/DFS

- Worst-case runtime is exponential.
- Worst-case memory usage is exponential.

- Worst-case runtime is exponential.
- Worst-case memory usage is linear.

Predictive Parsing

Predictive Parsing

- The leftmost DFS/BFS algorithms are **backtracking** algorithms.
  - Guess which production to use, then back up if it doesn't work.
  - Try to match a prefix by sheer dumb luck.
- There is another class of parsing algorithms called **predictive** algorithms.
  - Based on remaining input, predict (*without backtracking*) which production to use.

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.

Predictive Parsing

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

(	int	+	(	int	*	int	)	)

Predictive Parsing

E

$E \rightarrow$ `int`

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

$Op \rightarrow$ `+`

$Op \rightarrow$ `*`

(	int	+	(	int	*	int	)	)

Predictive Parsing

E
(E Op E)

E → int
E → (E Op E)
Op → +
Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

E
(E Op E)
(int Op E)

E → int
E → (E Op E)
Op → +
Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

E
(E Op E)
(int Op E)
(int + E)

E → int
E → (E Op E)
Op → +
Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

E
(E Op E)
(int Op E)
(int + E)
(int + (E Op E))

E → int

E → (E Op E)

Op → +

Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

E
(E Op E)
(int Op E)
(int + E)
(int + (E Op E))
(int + (int Op E))

E → int
E → (E Op E)
Op → +
Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

E → int
E → (E Op E)
Op → +
Op → *

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

(	int	+	(	int	*	int	)	)

Predictive Parsing

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

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

(	int	+	(	int	*	int	)	)

Predictive Parsing

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

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

(	int	+	(	int	*	int	)	)

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.**