

Parsing Algorithms

April 10, 2013

Review from Last Time

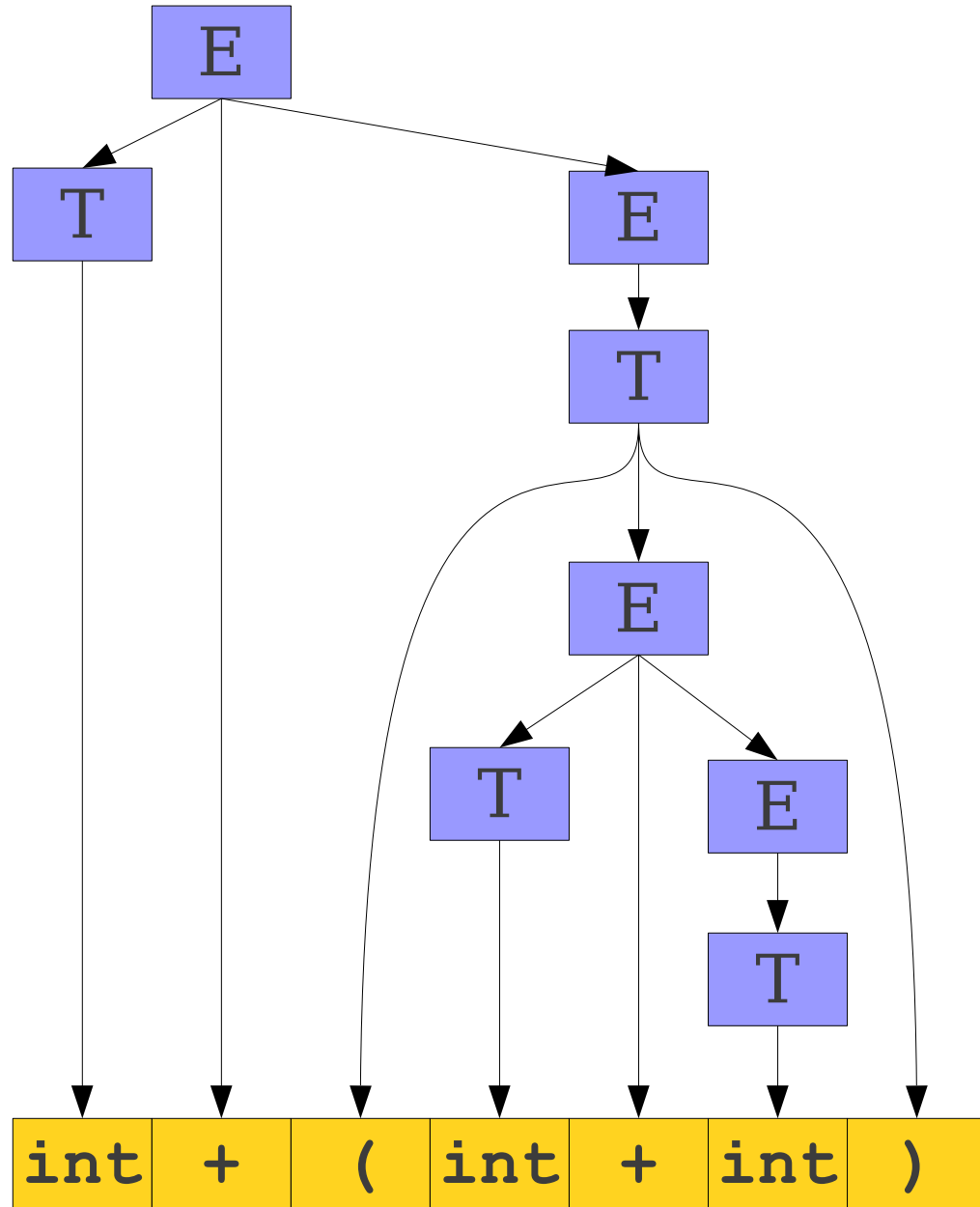
- Goal of syntax analysis: recover the intended structure of the program.
- Idea: Use a **context-free grammar** to describe the programming language.
- Given a sequence of tokens, look for a **parse tree** that generates those tokens.
- Recovering this syntax tree is called **parsing** and is the topic of this week (and part of next!)

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.

Top-Down Parsing

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



Challenges in Top-Down Parsing

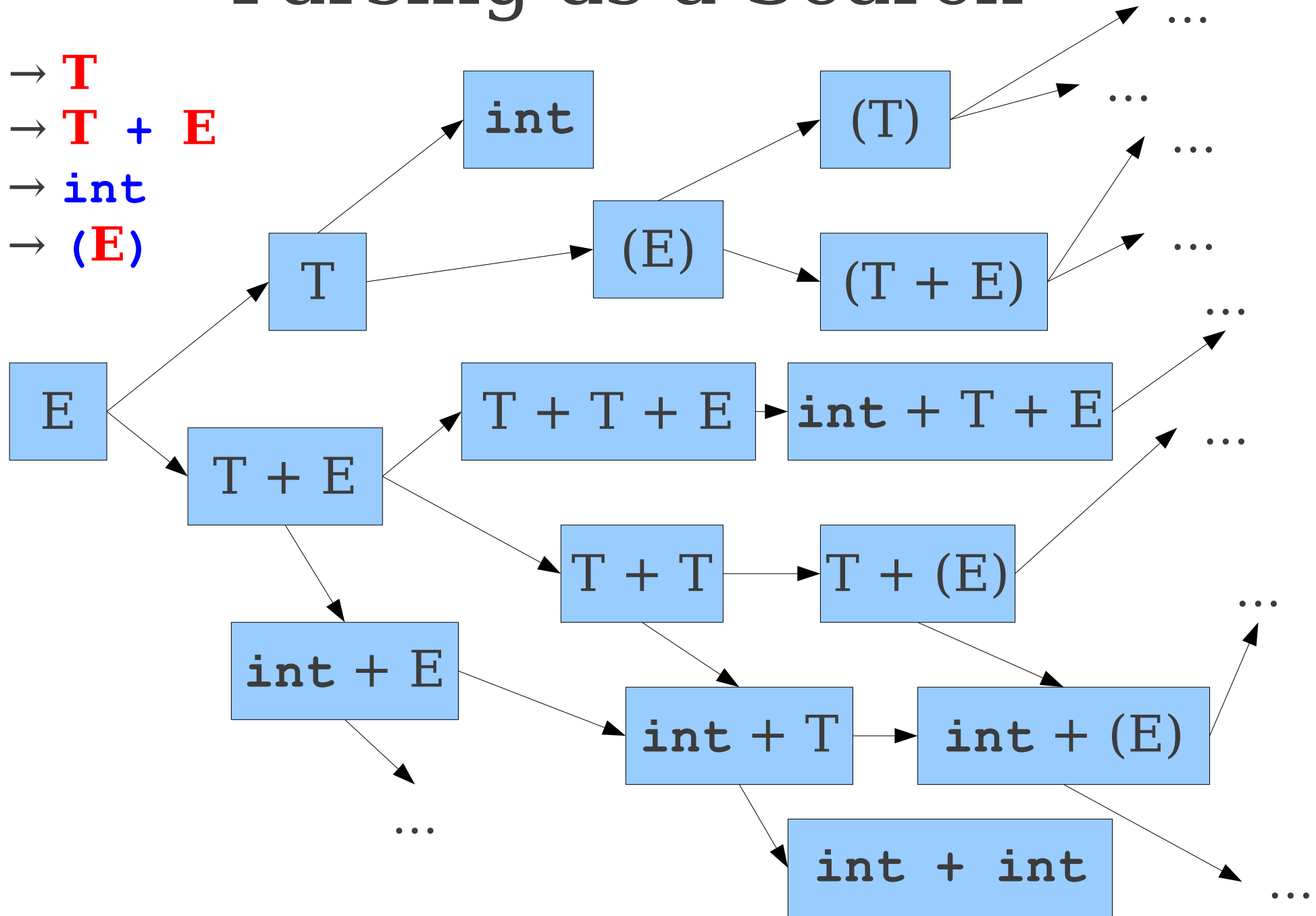
- Top-down parsing begins with virtually no information.
 - Begins with just the start symbol, which matches *every* program.
- How can we know which productions to apply?
- In general, we can't.
 - There are some grammars for which the best we can do is guess and backtrack if we're wrong.
 - If we have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

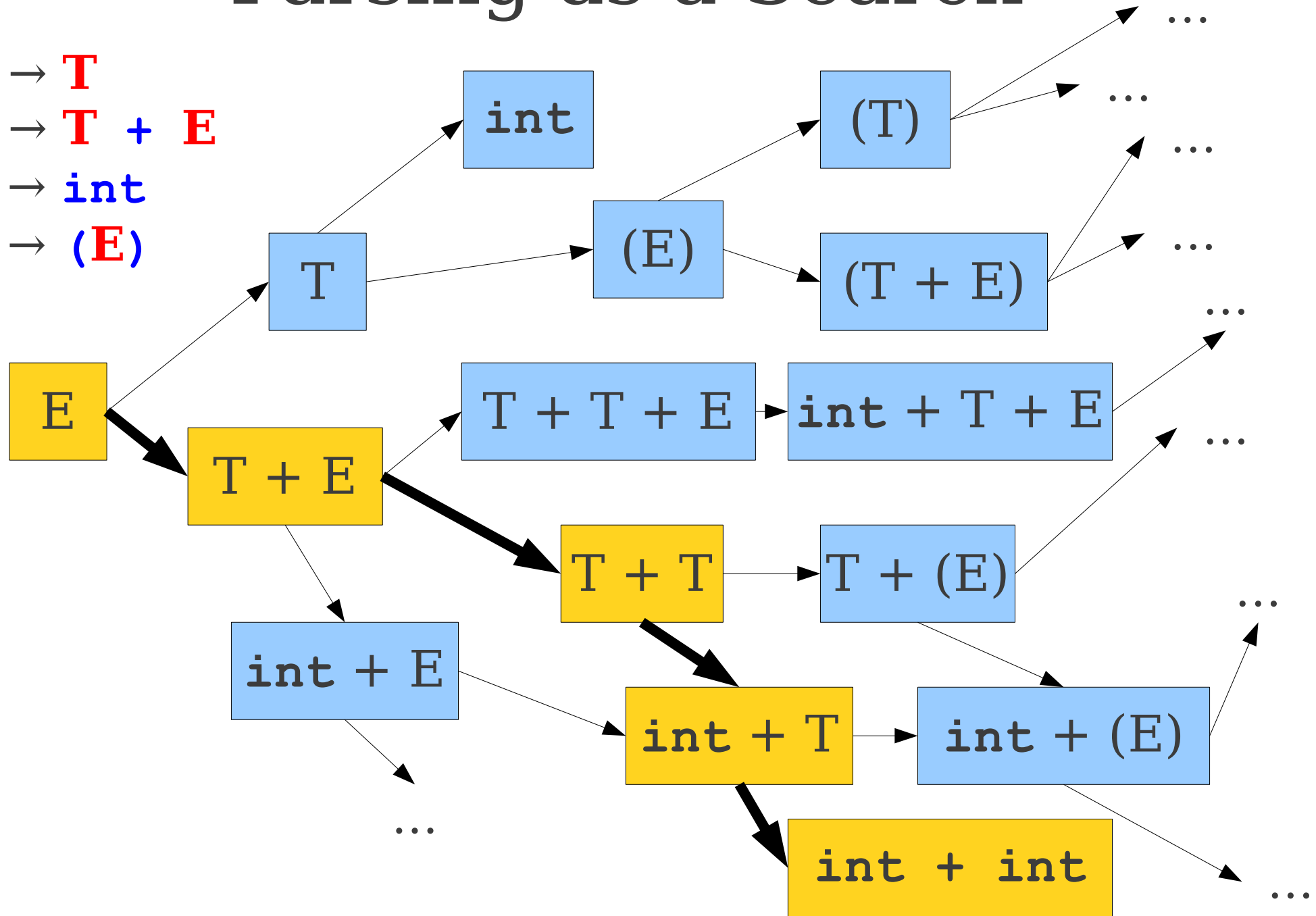
Parsing as a Search

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



Parsing as a Search

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



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 → **T**

E → **T** + **E**

T → **int**

T → (**E**)

`int + int`

Breadth-First Search Parsing

Worklist

E

E → **T**

E → **T** + **E**

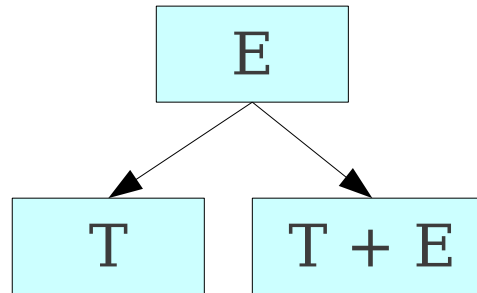
T → **int**

T → (**E**)

`int + int`

Breadth-First Search Parsing

Worklist



E → **T**

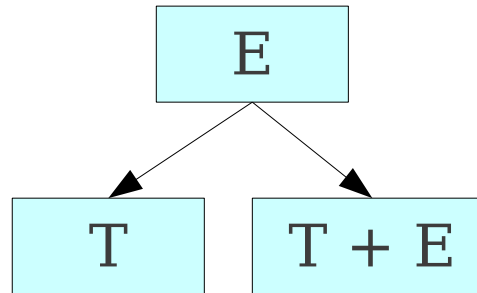
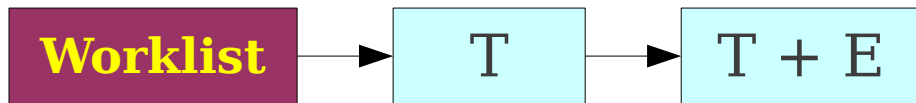
E → **T** + **E**

T → **int**

T → (**E**)

`int + int`

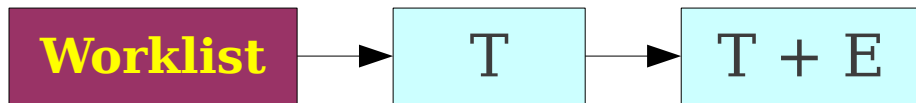
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing



E → **T**

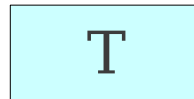
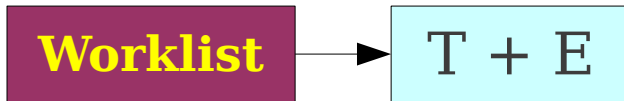
E → **T** + **E**

T → **int**

T → (**E**)

`int + int`

Breadth-First Search Parsing



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

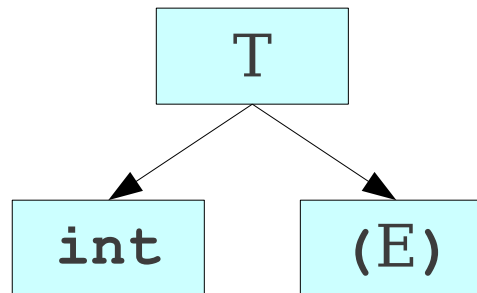
int + int

Breadth-First Search Parsing

Worklist



T + E



E → **T**

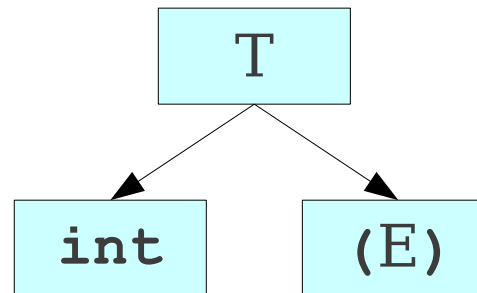
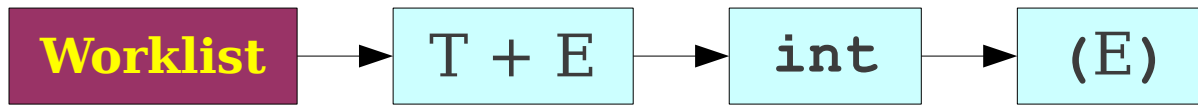
E → **T** + **E**

T → **int**

T → **(E)**

int + int

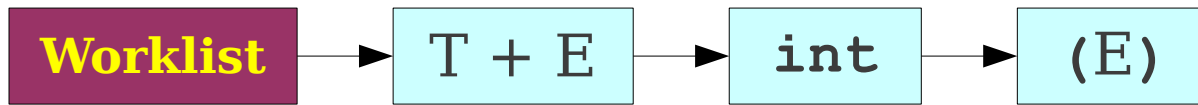
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing



E → **T**

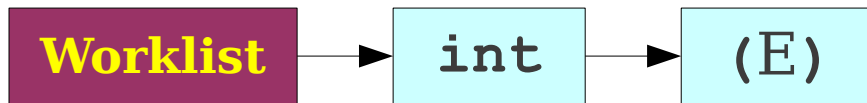
E → **T** + **E**

T → **int**

T → **(E)**

`int + int`

Breadth-First Search Parsing



T + E

E → **T**

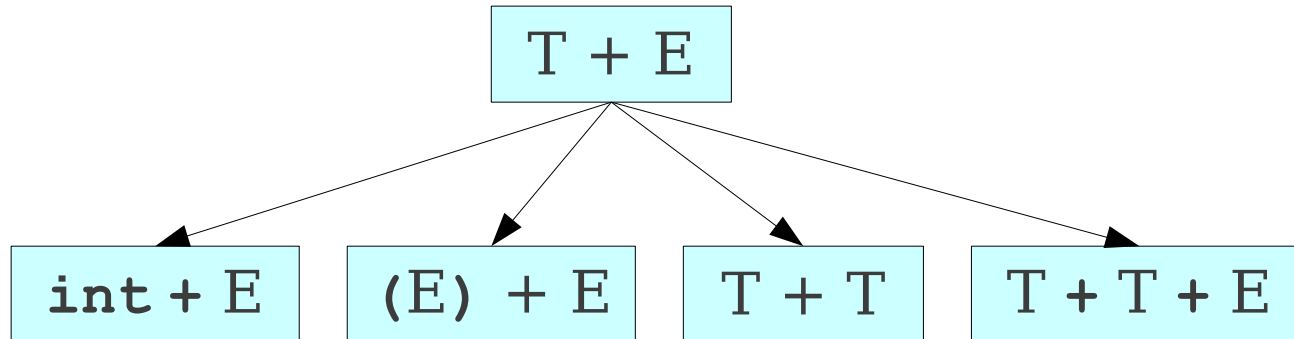
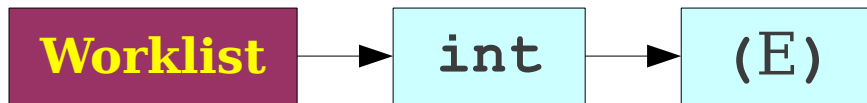
E → **T** + **E**

T → **int**

T → **(E)**

int + int

Breadth-First Search Parsing



E → **T**

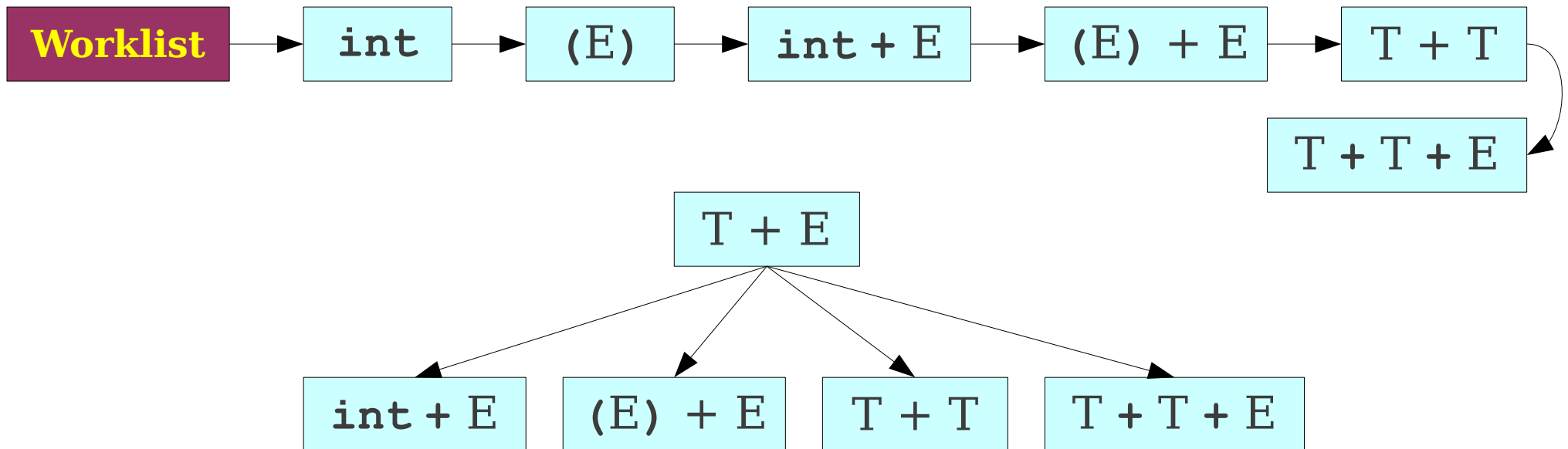
E → **T** + **E**

T → **int**

T → **(E)**

int + int

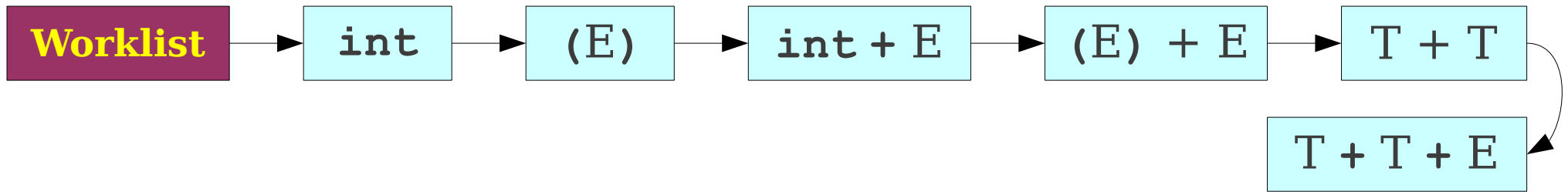
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing



E → **T**

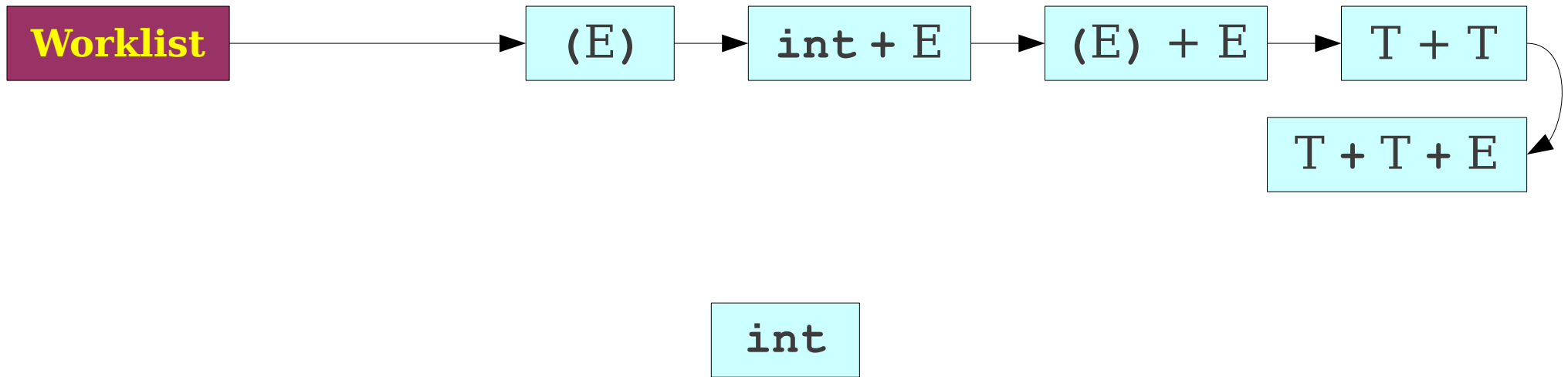
E → **T** + **E**

T → **int**

T → **(E)**

`int + int`

Breadth-First Search Parsing



E → **T**

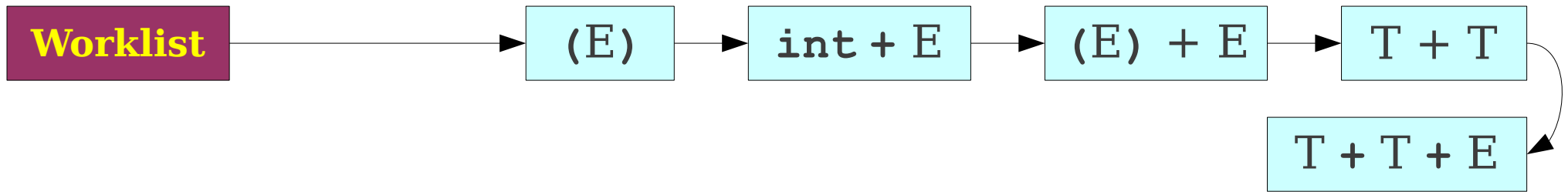
E → **T + E**

T → **int**

T → **(E)**

int + int

Breadth-First Search Parsing



$E \rightarrow T$

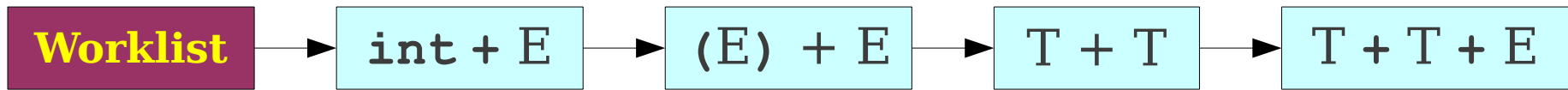
$E \rightarrow T + E$

$T \rightarrow \text{int}$

$T \rightarrow (E)$

$\text{int} + \text{int}$

Breadth-First Search Parsing



(E)

E → **T**

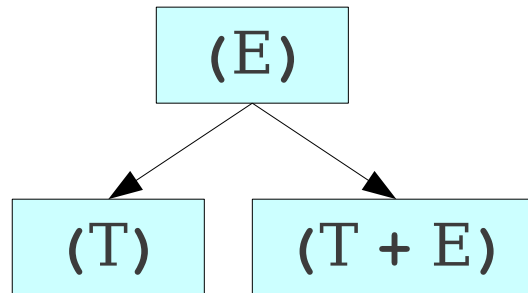
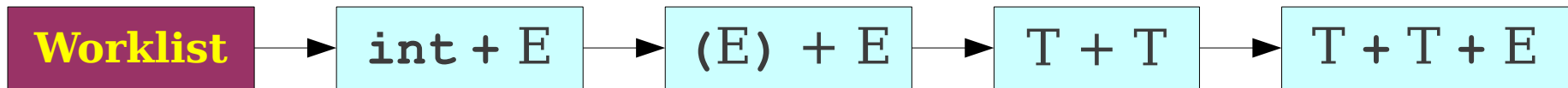
E → **T** + **E**

T → **int**

T → **(E)**

int + int

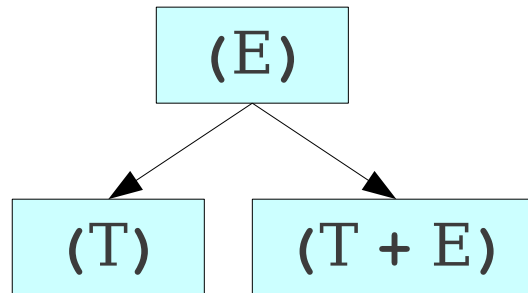
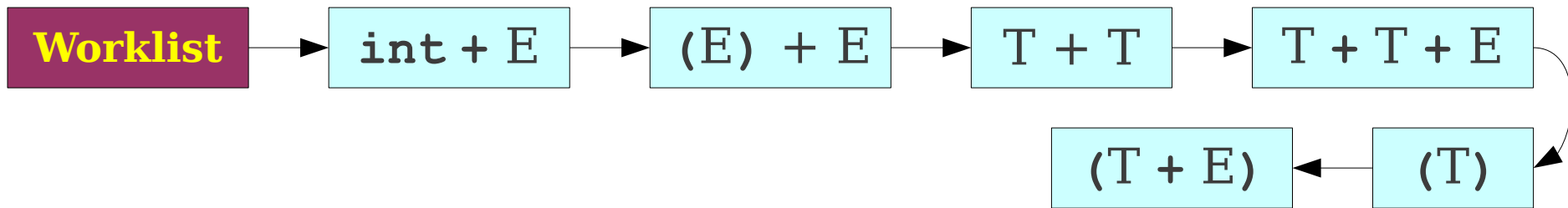
Breadth-First Search Parsing



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

int + int

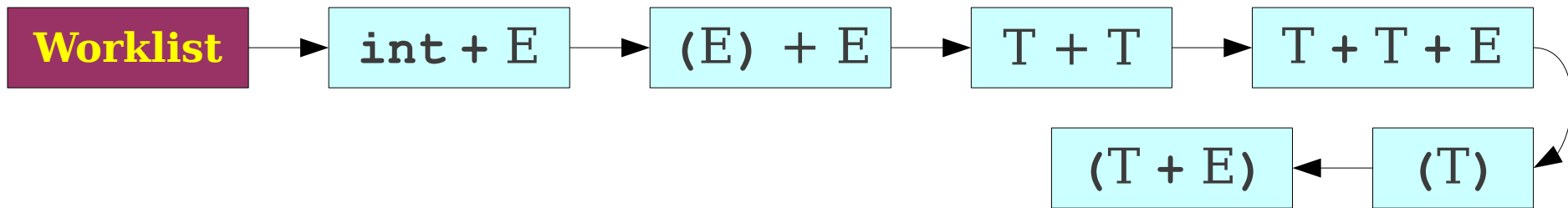
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing



E → **T**

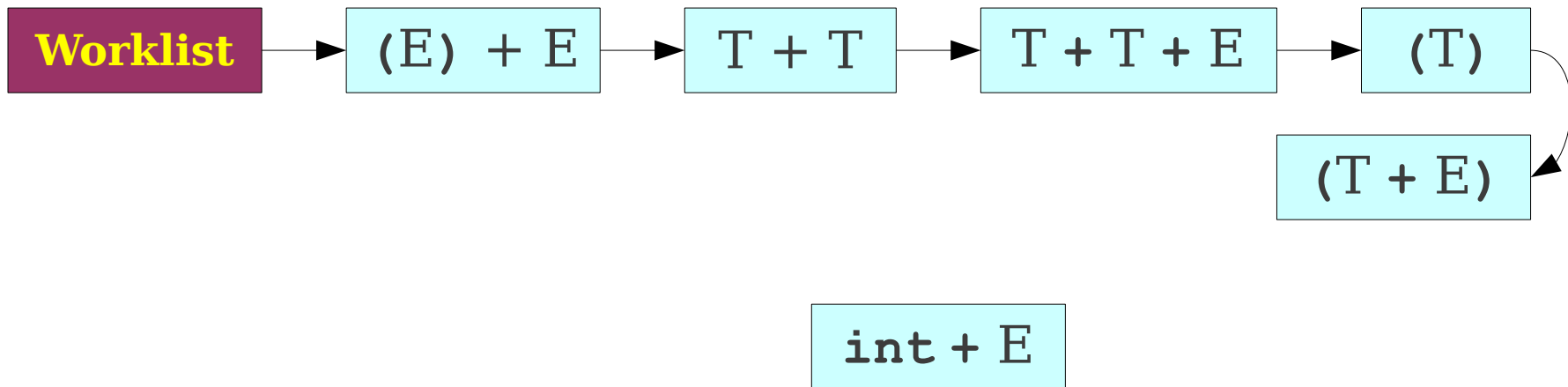
E → **T** + **E**

T → **int**

T → **(E)**

`int + int`

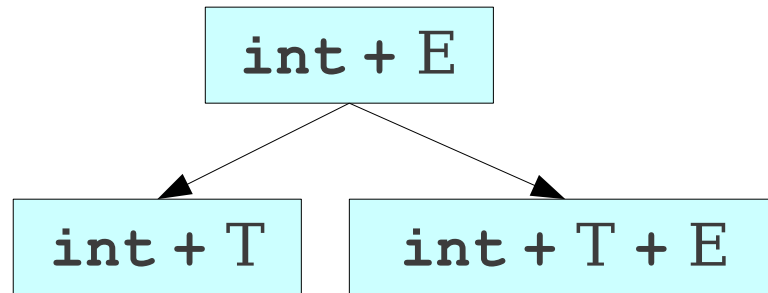
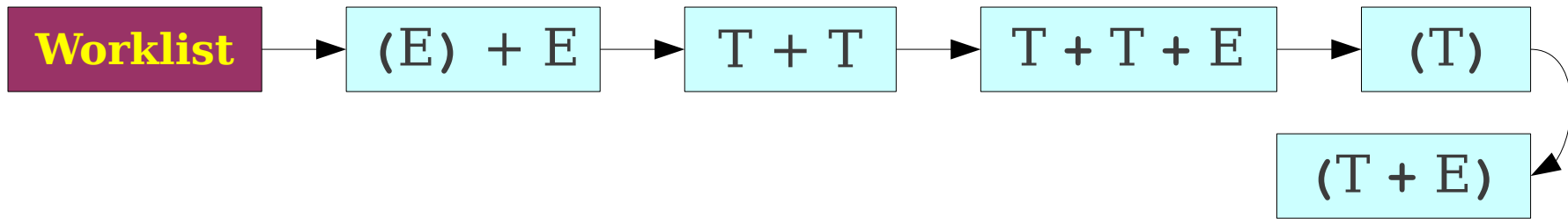
Breadth-First Search Parsing



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

`int + int`

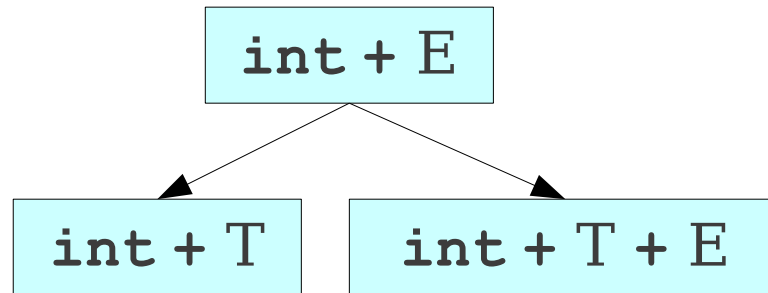
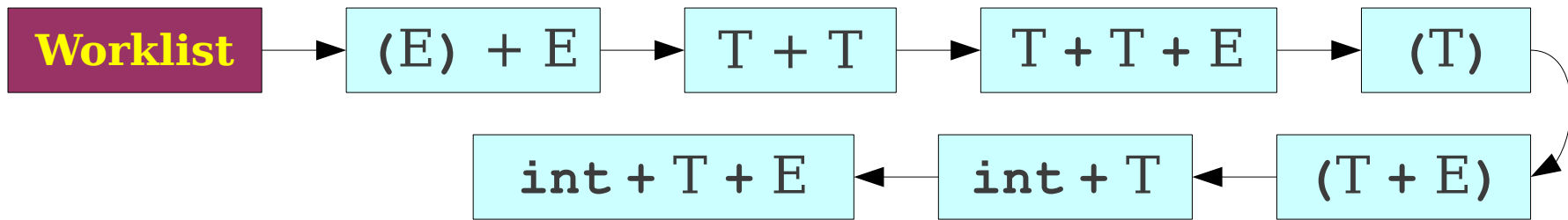
Breadth-First Search Parsing



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

`int + int`

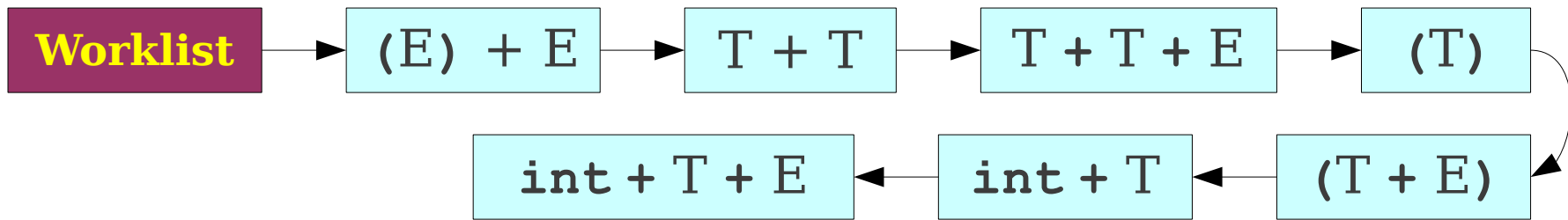
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing



E → **T**

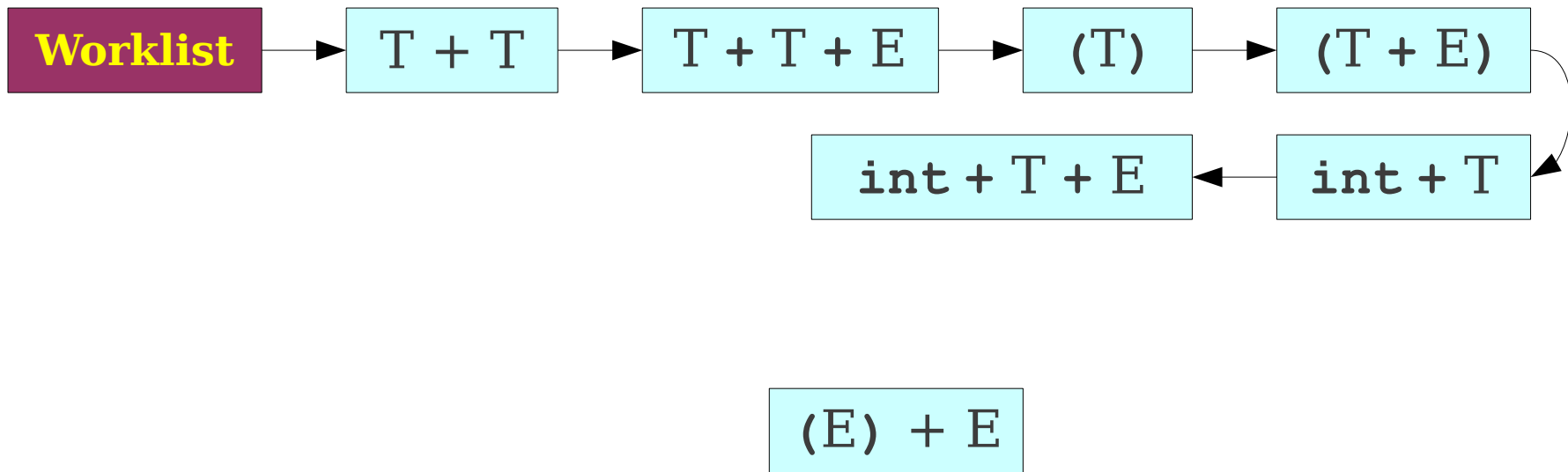
E → **T** + **E**

T → **int**

T → **(E)**

int + int

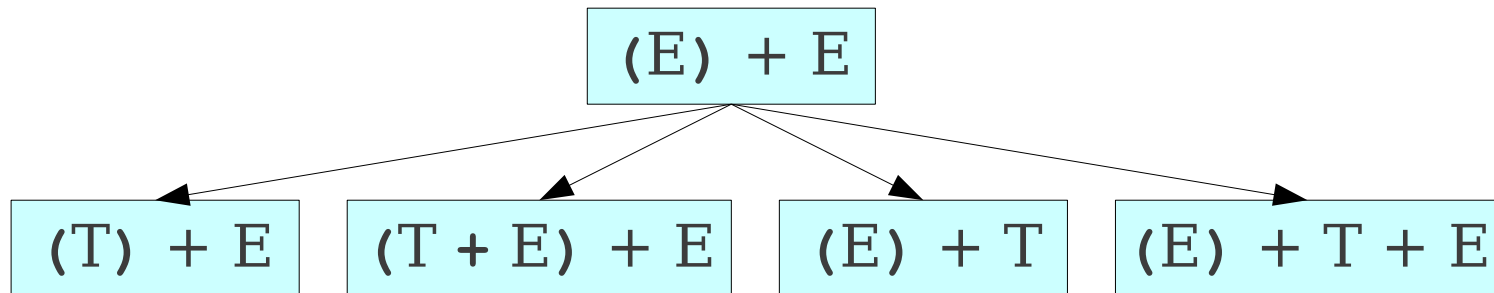
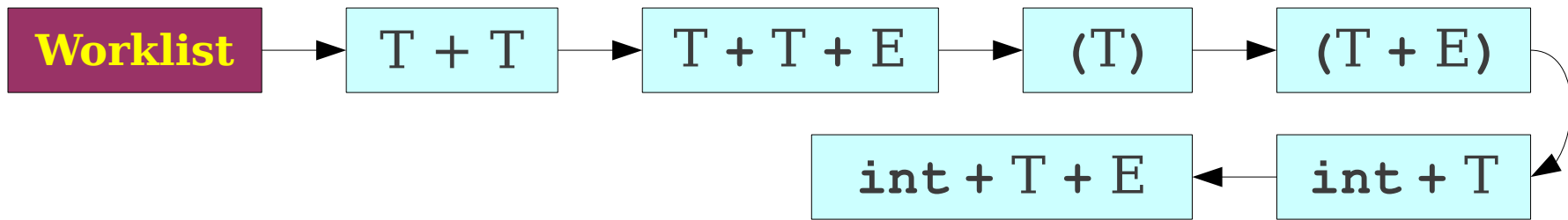
Breadth-First Search Parsing



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

int + int

Breadth-First Search Parsing



E → **T**

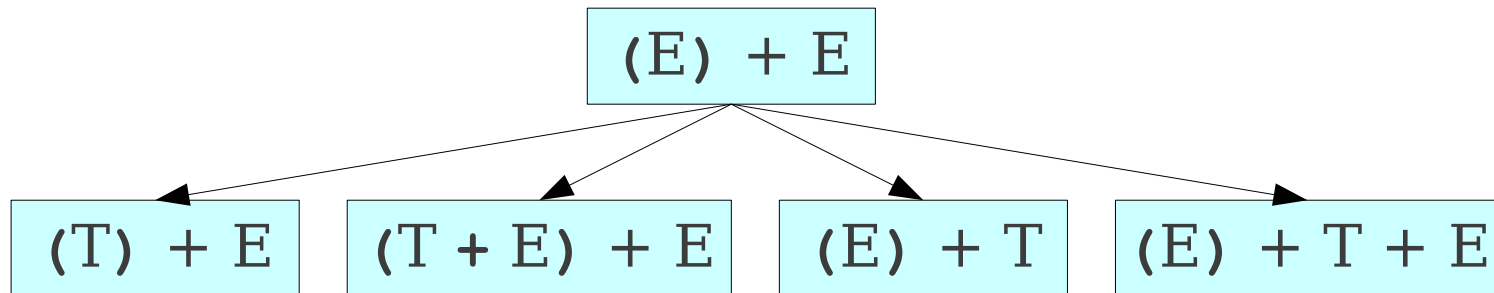
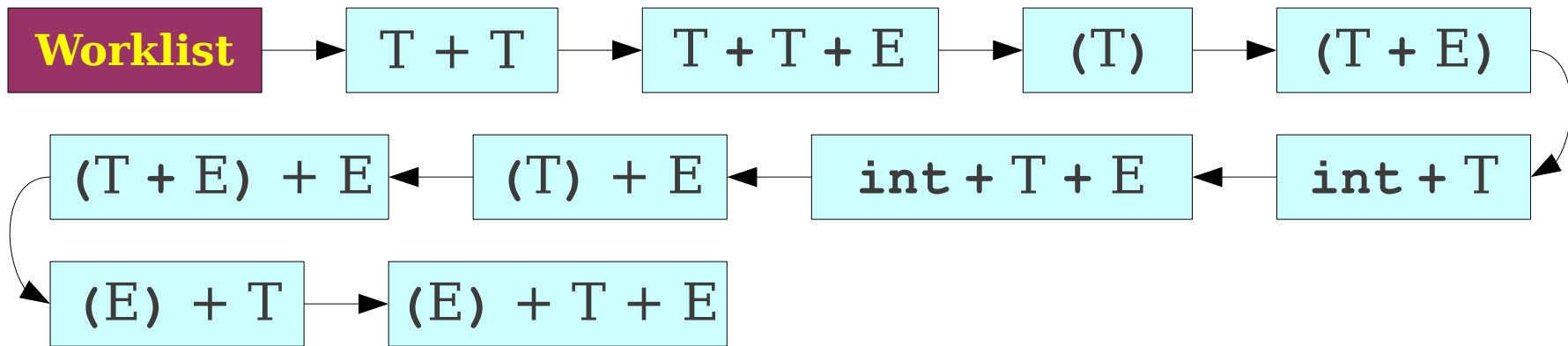
E → **T** + **E**

T → **int**

T → **(E)**

int + **int**

Breadth-First Search Parsing



$E \rightarrow T$

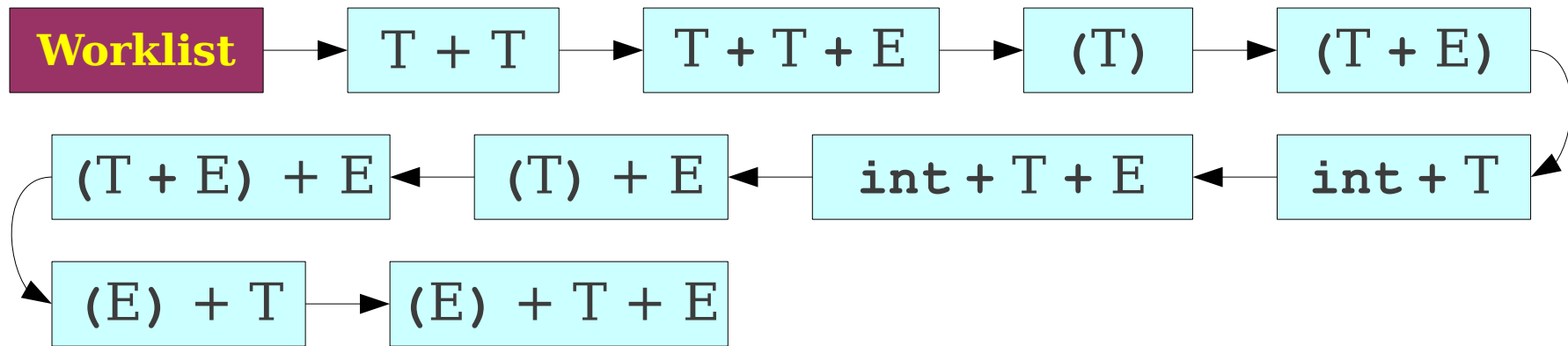
$E \rightarrow T + E$

$T \rightarrow int$

$T \rightarrow (E)$

$int + int$

Breadth-First Search Parsing



E → **T**

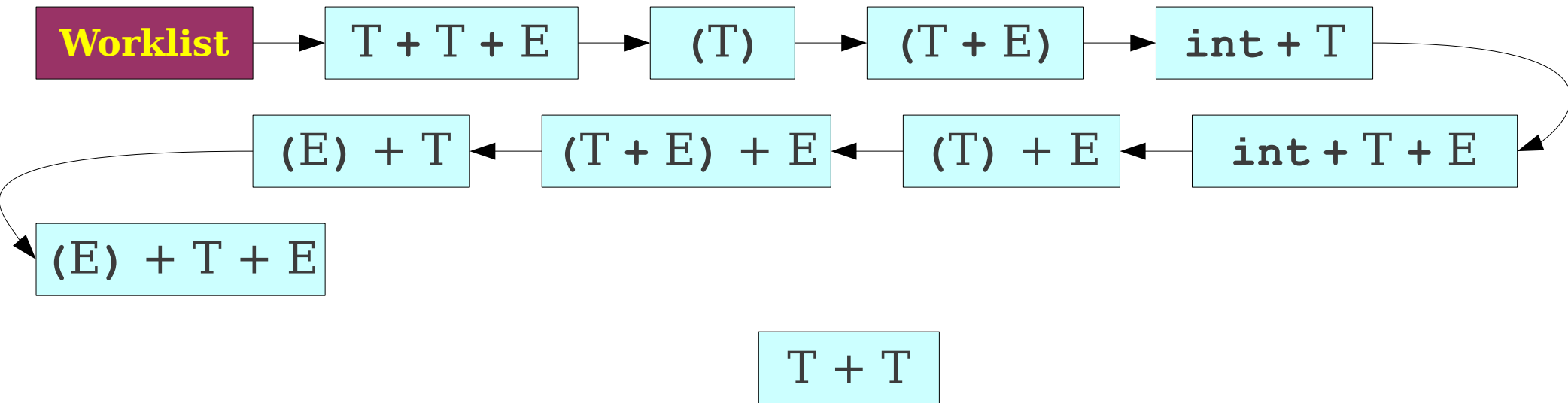
E $\rightarrow$ **T** + **E**

T → **int**

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

int + int

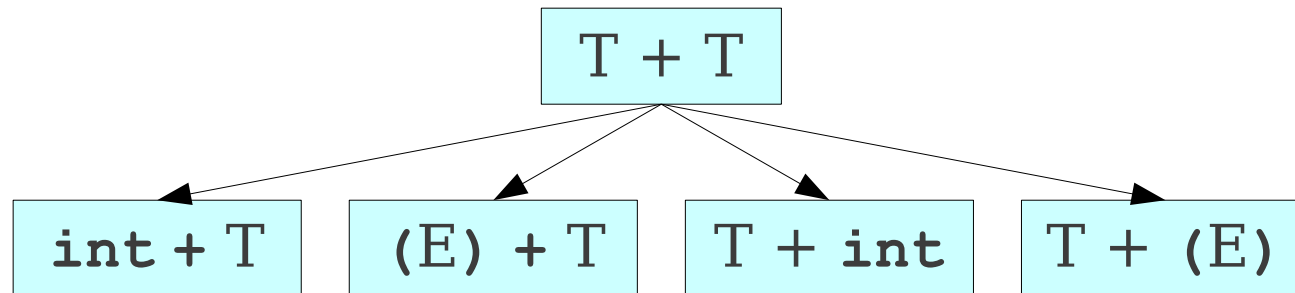
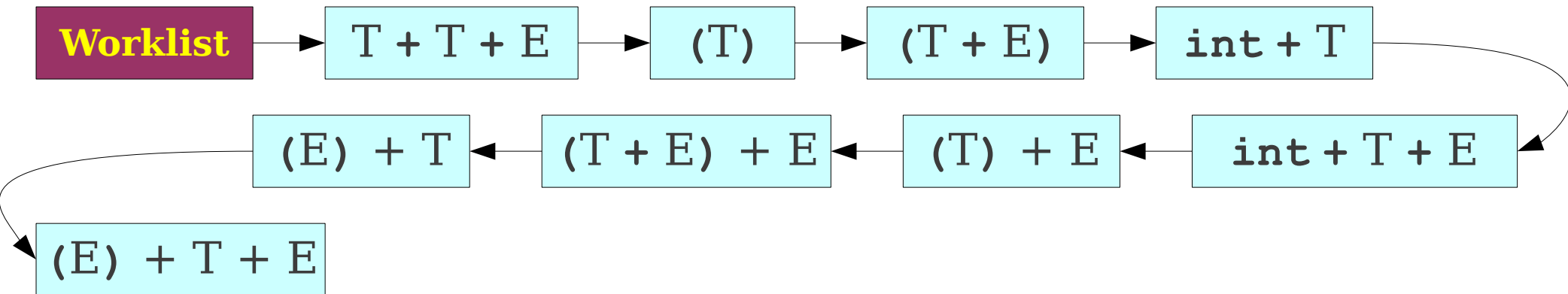
Breadth-First Search Parsing



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

$\text{int} + \text{int}$

Breadth-First Search Parsing



E → **T**

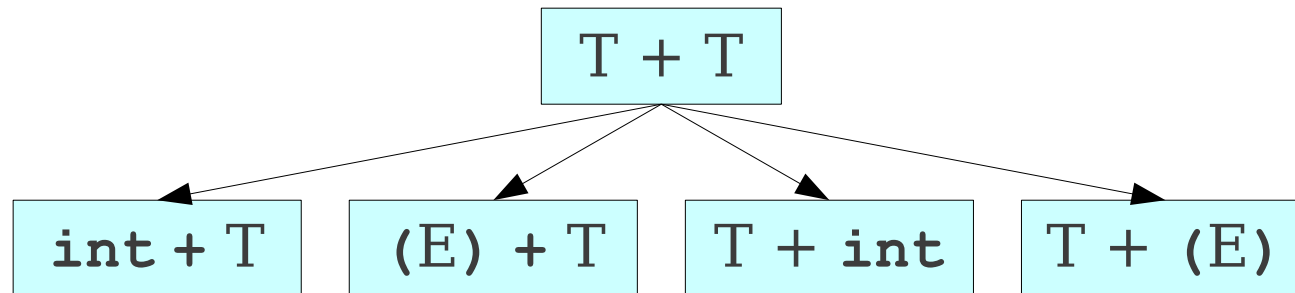
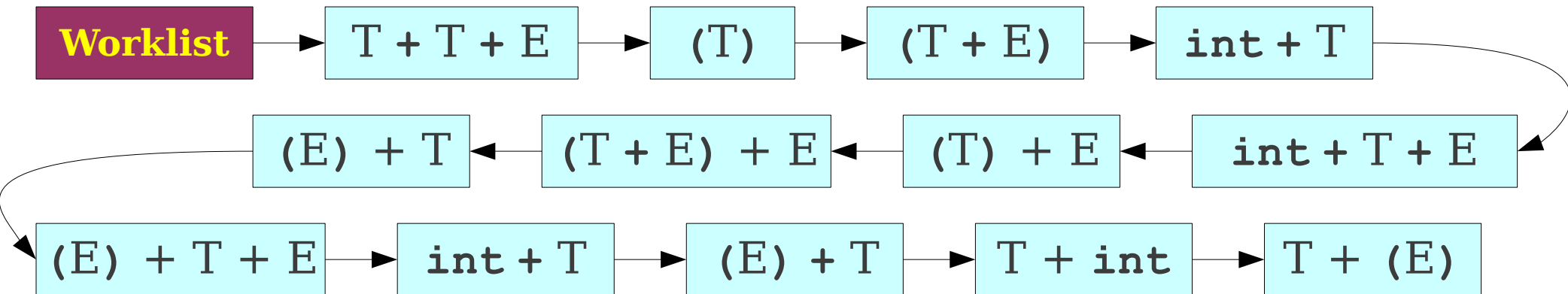
E → **T** + **E**

T → **int**

T → (**E**)

int + int

Breadth-First Search Parsing



E → **T**

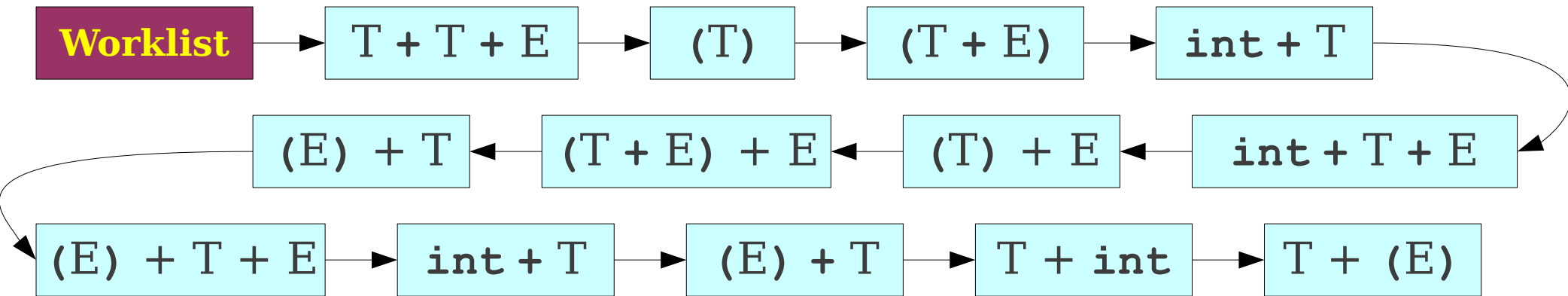
E → **T** + **E**

T → **int**

T → **(E)**

int + int

Breadth-First Search Parsing



$E \rightarrow T$

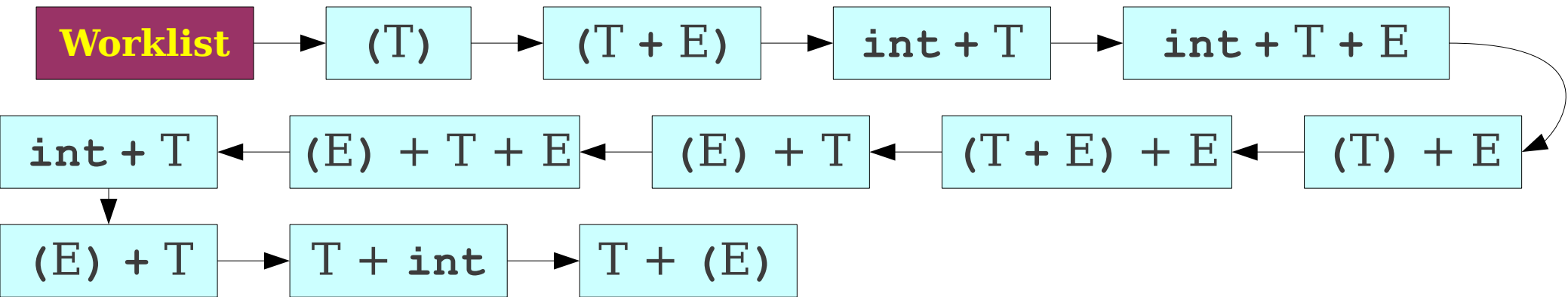
$E \rightarrow T + E$

$T \rightarrow \text{int}$

$T \rightarrow (E)$

$\text{int} + \text{int}$

Breadth-First Search Parsing

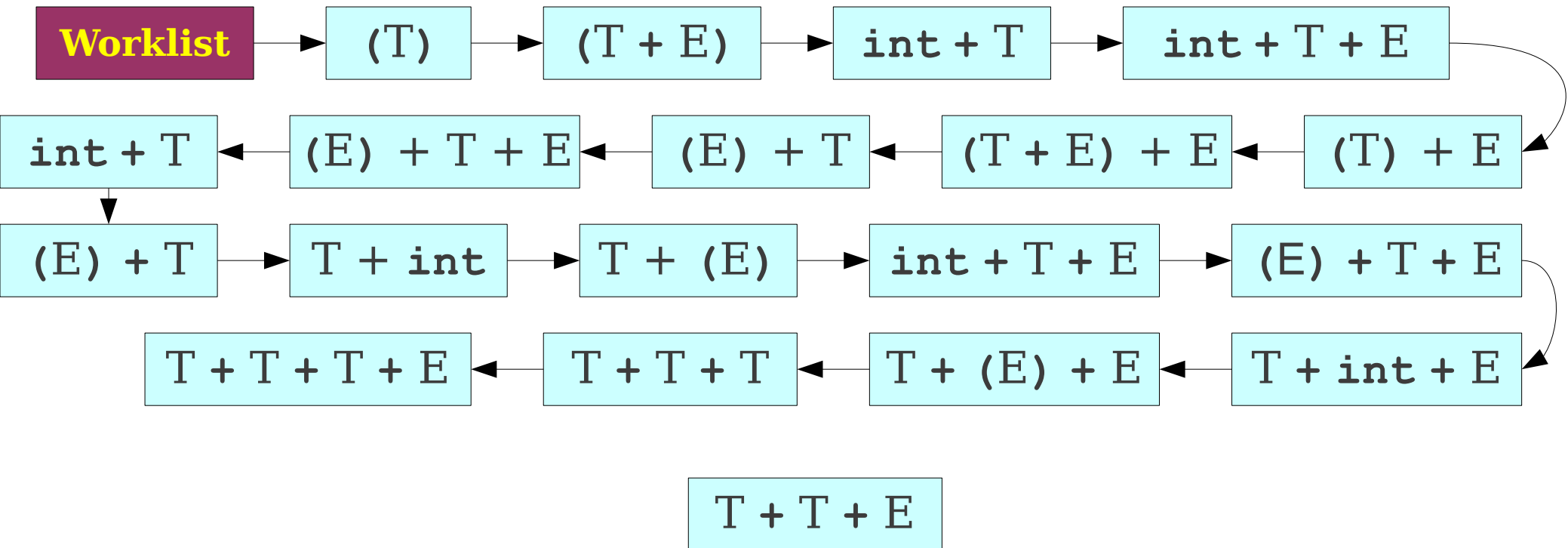


T + T + E

int + int

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

Breadth-First Search Parsing



E → **T**

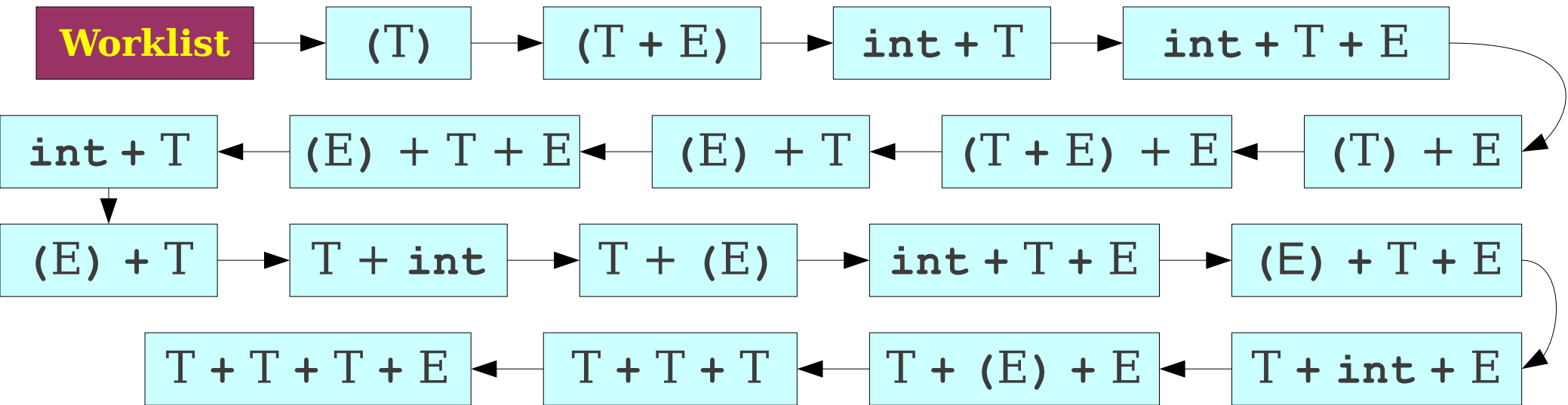
E → **T** + **E**

T → **int**

T → (**E**)

int + int

Breadth-First Search Parsing



E → **T**

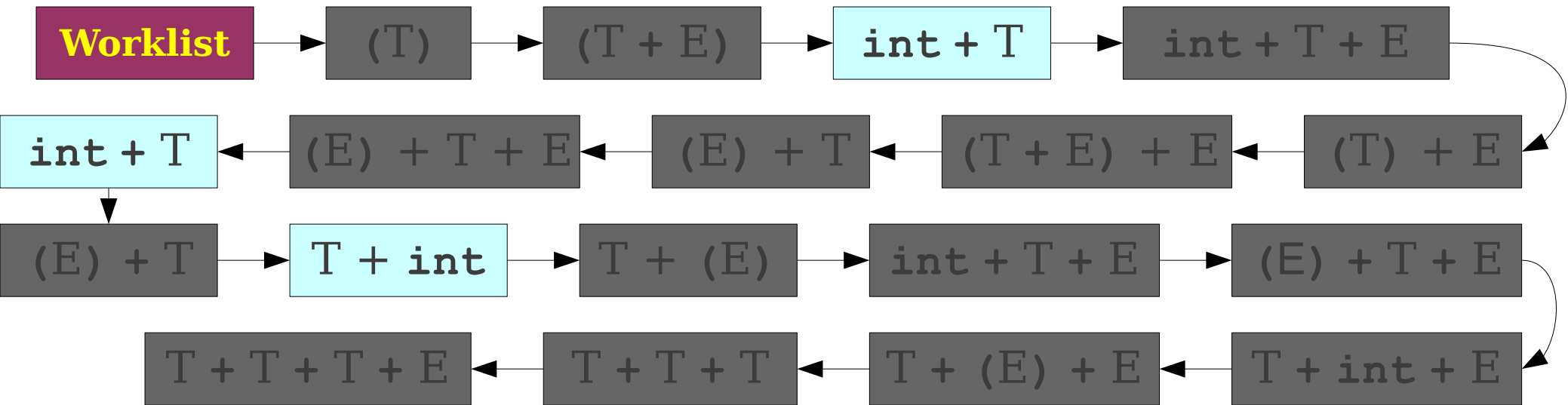
E → **T** + **E**

T → **int**

T → **(E)**

int + int

Breadth-First Search Parsing



E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int + int

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 y .
- 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 y , then no string derived from τ can ever match y .
- 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



E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

`int + int`

Leftmost BFS

Worklist

E

E → **T**

E → **T** + **E**

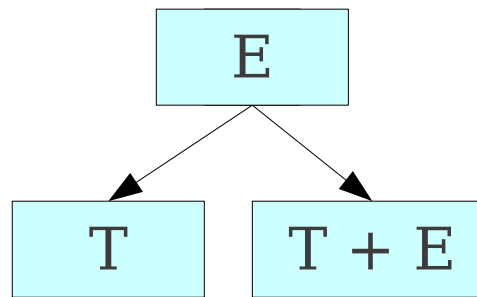
T → **int**

T → (**E**)

`int + int`

Leftmost BFS

Worklist



E → **T**

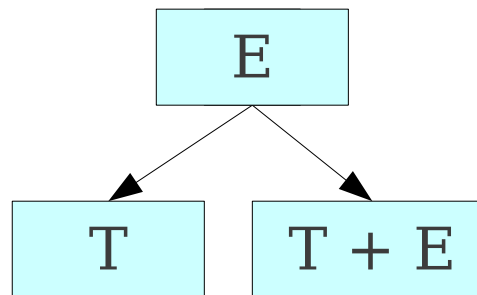
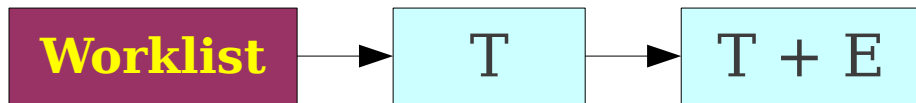
E → **T** + **E**

T → **int**

T → (**E**)

`int + int`

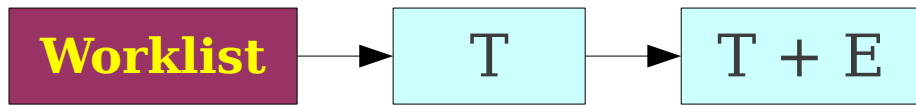
Leftmost BFS



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

`int + int`

Leftmost BFS



E → **T**

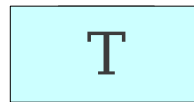
E → **T** + **E**

T → **int**

T → (**E**)

`int + int`

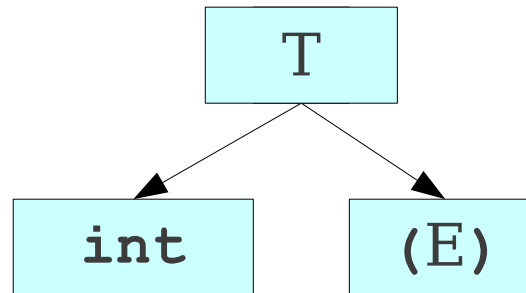
Leftmost BFS



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

`int + int`

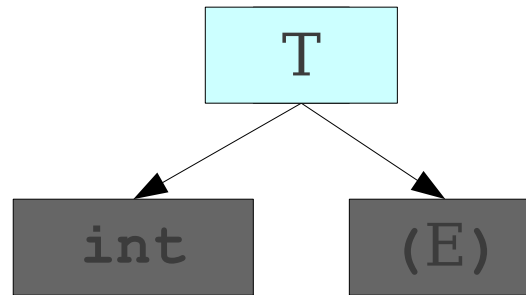
Leftmost BFS



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

`int + int`

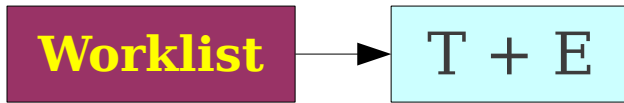
Leftmost BFS



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

`int + int`

Leftmost BFS



E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

`int + int`

Leftmost BFS

Worklist

T + E

E → **T**

E → **T** + **E**

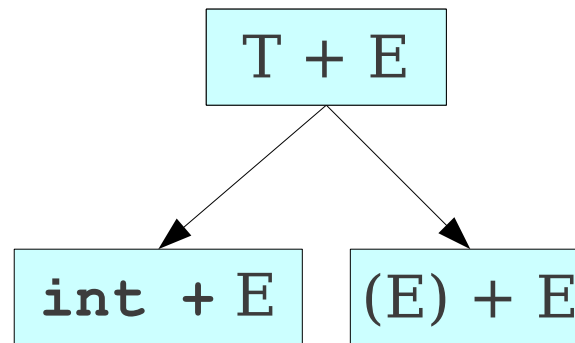
T → **int**

T → (**E**)

int + int

Leftmost BFS

Worklist

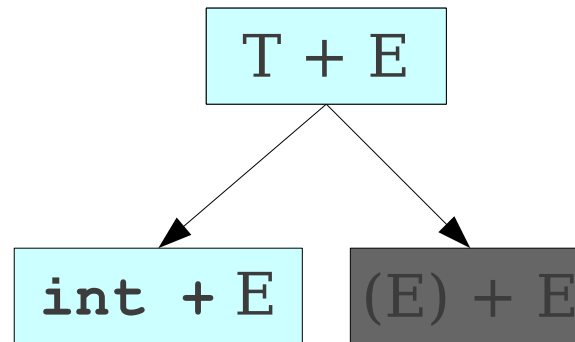


E → **T**
E → **T** + **E**
T → **int**
T → **(E)**

`int + int`

Leftmost BFS

Worklist



E → **T**

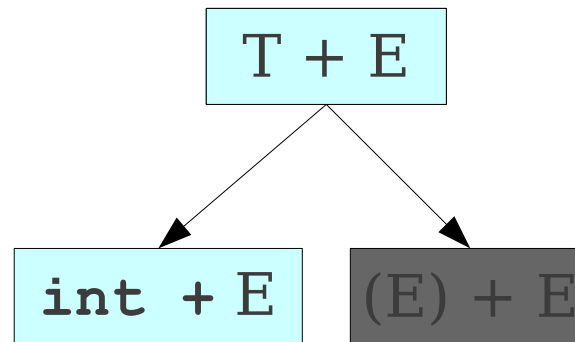
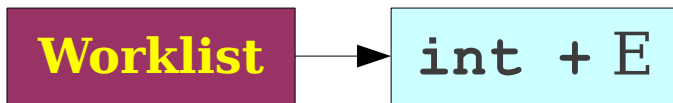
E → **T** + **E**

T → **int**

T → (**E**)

int + int

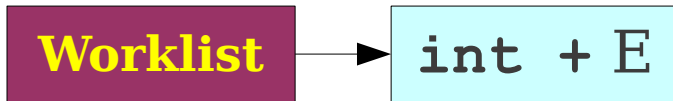
Leftmost BFS



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

`int + int`

Leftmost BFS



E → **T**

E → **T** + **E**

T → **int**

T → **(E)**

`int + int`

Leftmost BFS

Worklist

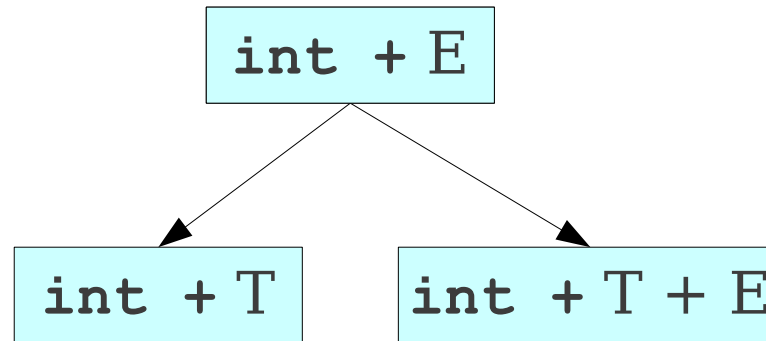
int + E

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

int + int

Leftmost BFS

Worklist



E → **T**

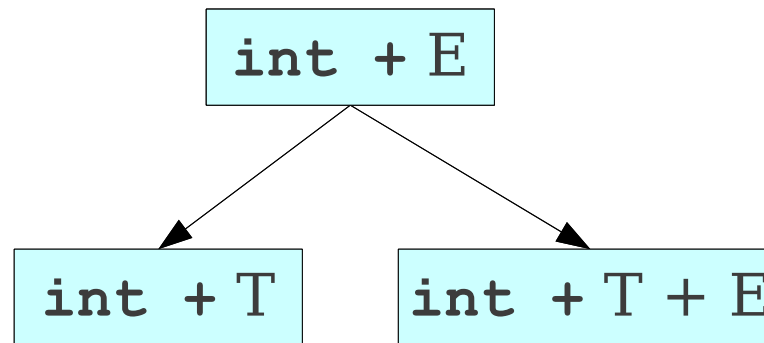
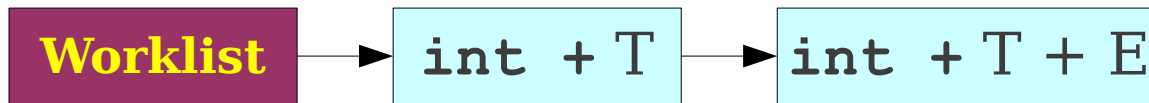
E → **T** + **E**

T → **int**

T → (**E**)

int + int

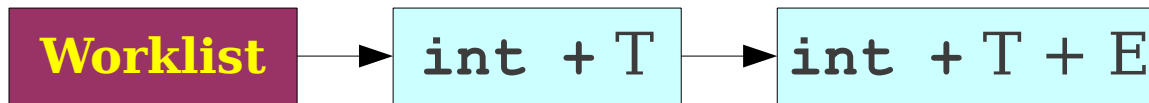
Leftmost BFS



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

$\text{int} + \text{int}$

Leftmost BFS



E → **T**

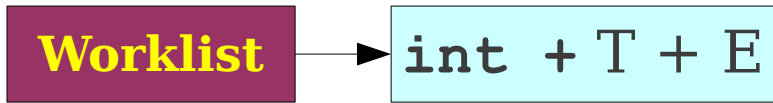
E → **T** + **E**

T → **int**

T → **(E)**

`int + int`

Leftmost BFS

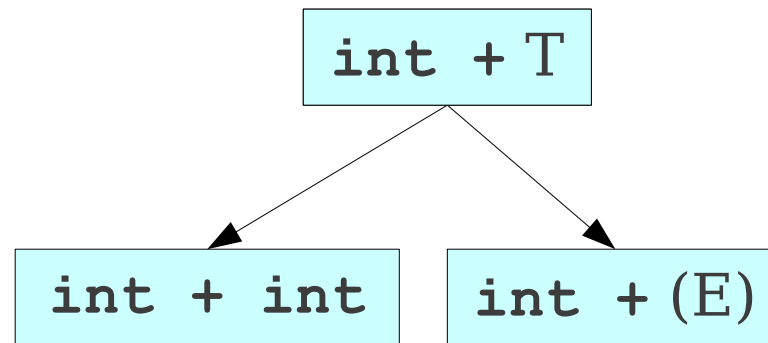
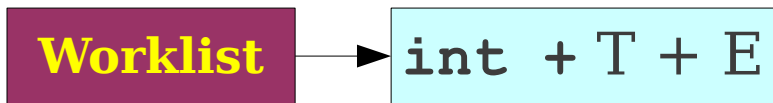


int + T

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

int + int

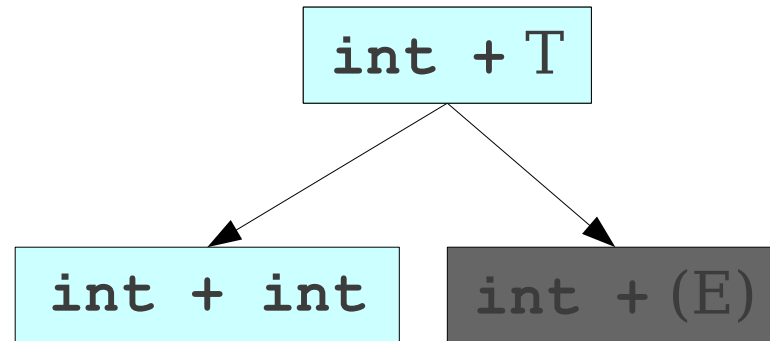
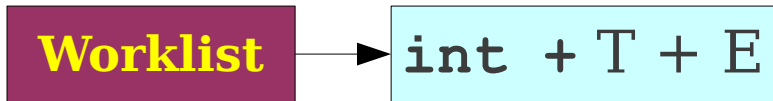
Leftmost BFS



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

int + int

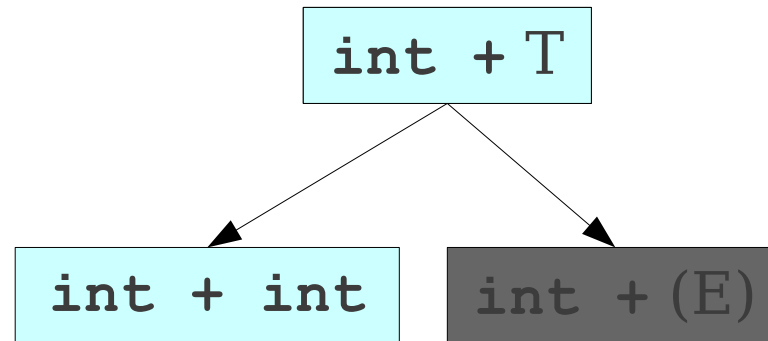
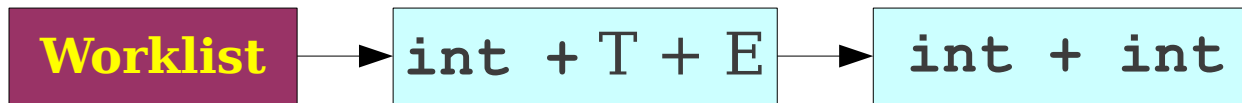
Leftmost BFS



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

int + int

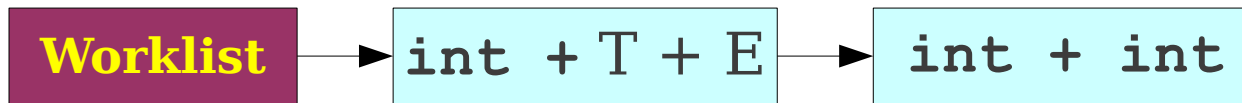
Leftmost BFS



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

`int + int`

Leftmost BFS



E → **T**

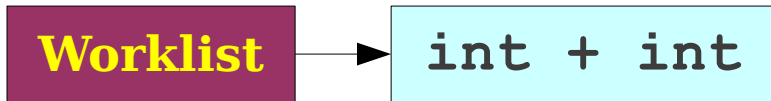
E → **T** + **E**

T → **int**

T → **(E)**

int + int

Leftmost BFS

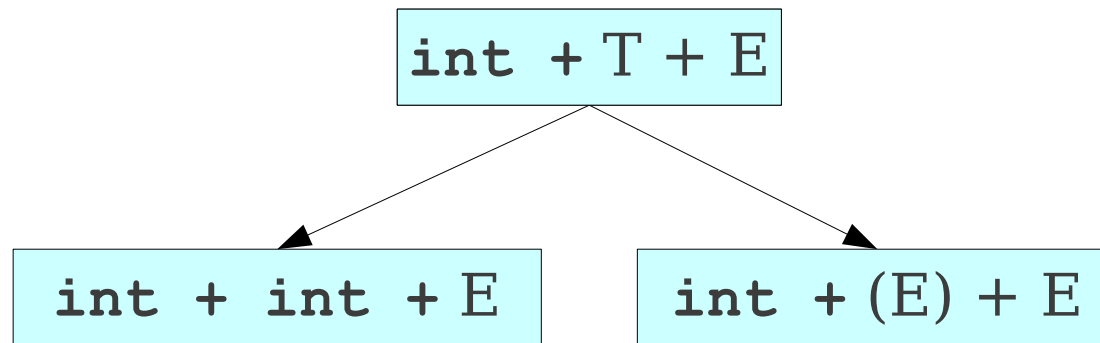
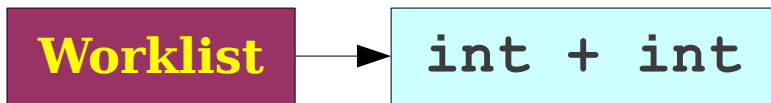


int + T + E

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

int + int

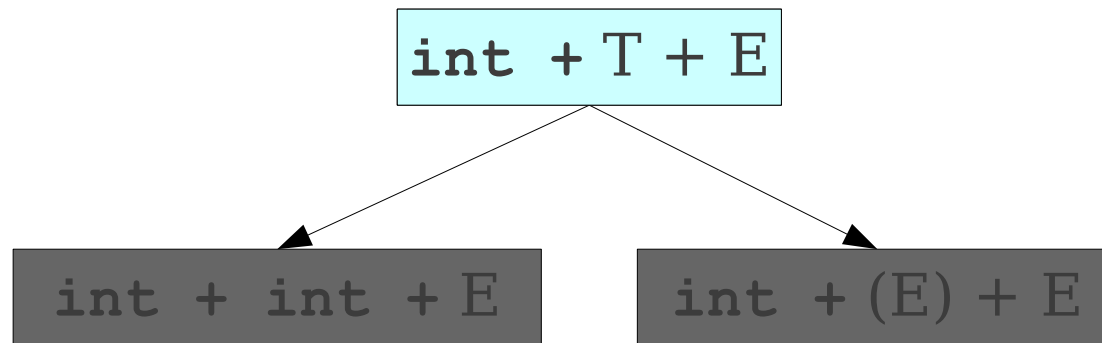
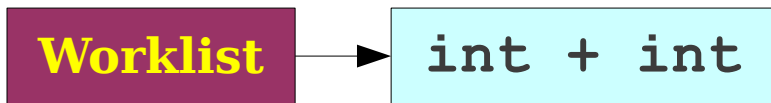
Leftmost BFS



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

int + int

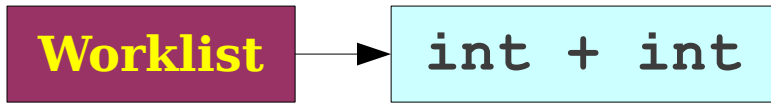
Leftmost BFS



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

int + int

Leftmost BFS



E → **T**

E → **T** + **E**

T → **int**

T → **(E)**

`int + int`

Leftmost BFS

Worklist

`int + int`

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

`int + int`

Leftmost BFS

Worklist



`int + int`

E → **T**

E → **T** + **E**

T → **int**

T → (**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

A → **A****a** | **A****b** | **c**

Leftmost BFS Has Problems

Worklist

A → **A****a** | **A****b** | **c**

caaaaaaaaaa

Leftmost BFS Has Problems



A → **A***a* | **A***b* | *c*

caaaaaaaaaa

Leftmost BFS Has Problems

Worklist

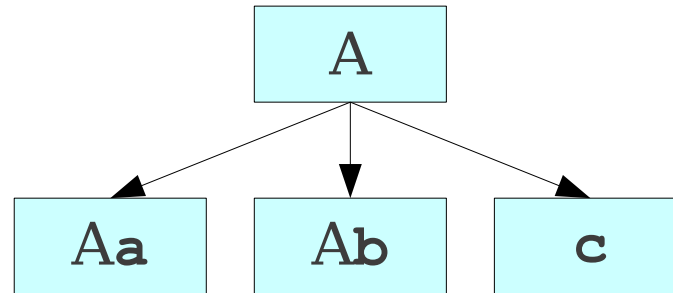
A

A → **A****a** | **A****b** | **c**

caaaaaaaaaa

Leftmost BFS Has Problems

Worklist

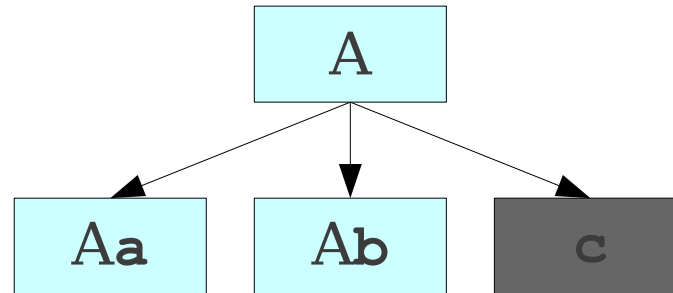


A → **Aa** | **Ab** | **c**

caaaaaaaaaa

Leftmost BFS Has Problems

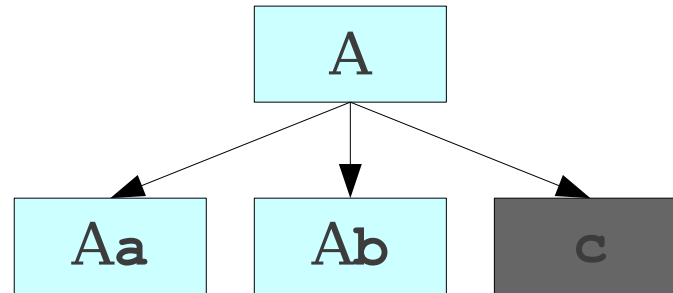
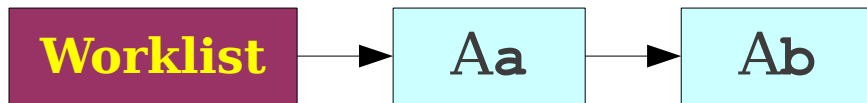
Worklist



A → **A****a** | **A****b** | **c**

caaaaaaaaaa

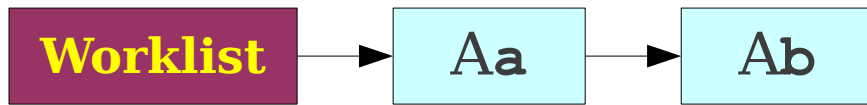
Leftmost BFS Has Problems



A → **A****a** | **A****b** | **c**

caaaaaaaaaa

Leftmost BFS Has Problems



A → **A****a** | **A****b** | **c**

caaaaaaaaaa

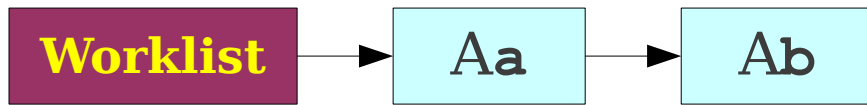
Leftmost BFS Has Problems



A → **A****a** | **A****b** | **c**

caaaaaaaaaa

Leftmost BFS Has Problems



A → **A****a** | **A****b** | **c**

caaaaaaaaaa

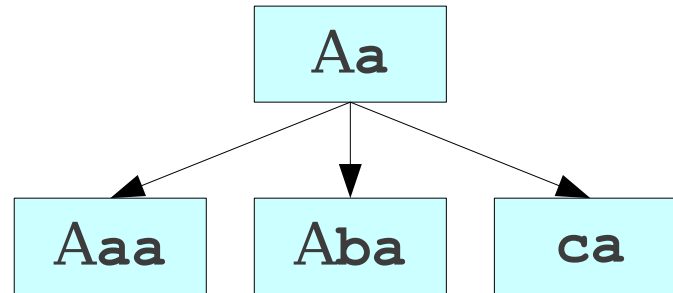
Leftmost BFS Has Problems



A → **A****a** | **A****b** | **c**

caaaaaaaaaa

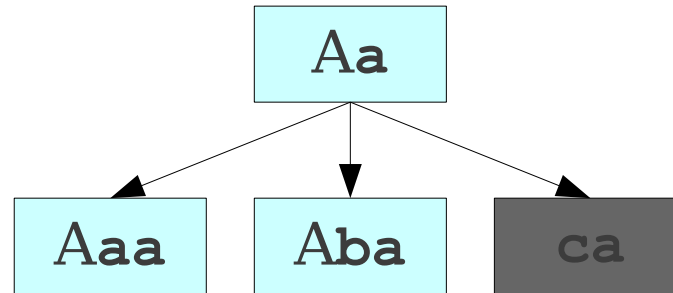
Leftmost BFS Has Problems



A → **Aa** | **Ab** | **c**

caaaaaaaaaa

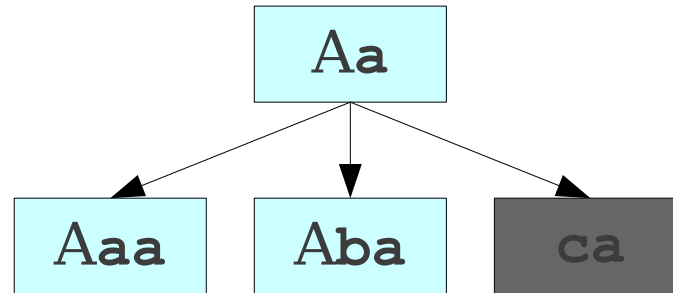
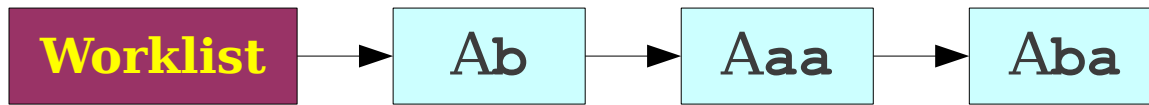
Leftmost BFS Has Problems



A → **A****a** | **A****b** | **c**

caaaaaaaaaa

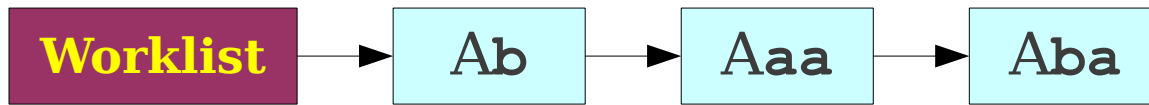
Leftmost BFS Has Problems



A → **Aa** | **Ab** | **c**

caaaaaaaaaa

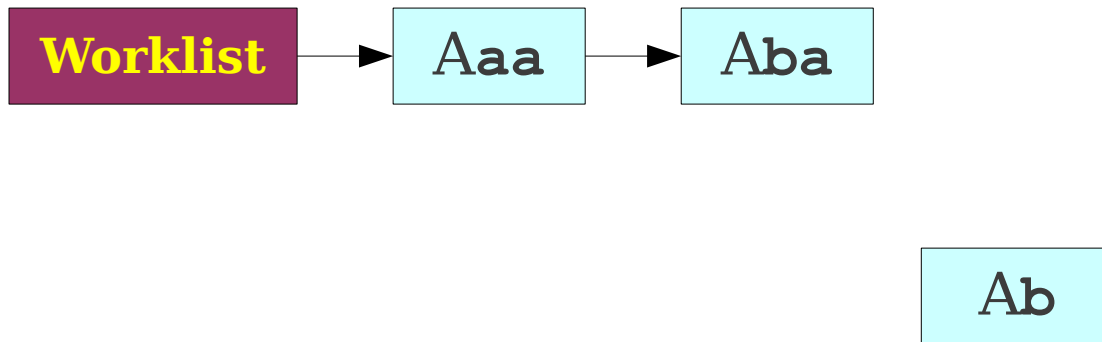
Leftmost BFS Has Problems



A → **Aa** | **Ab** | **c**

caaaaaaaaaa

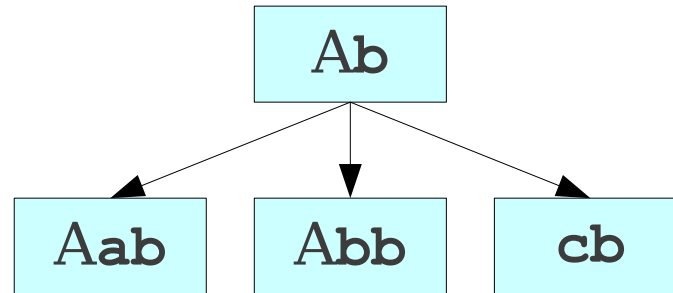
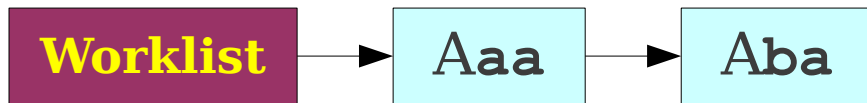
Leftmost BFS Has Problems



A → **A****a** | **A****b** | **c**

caaaaaaaaaa

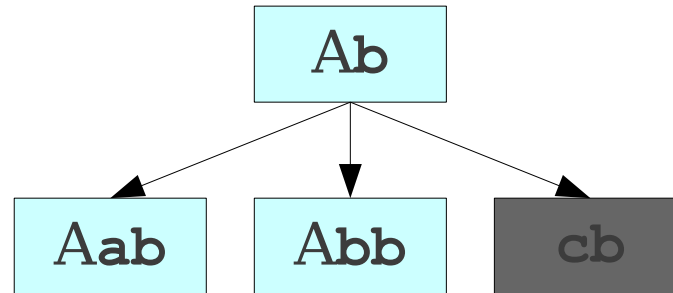
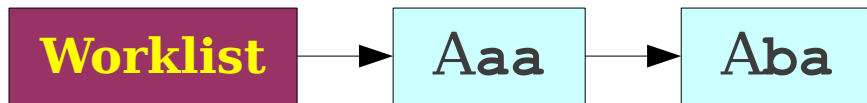
Leftmost BFS Has Problems



A → **Aa** | **Ab** | **c**

caaaaaaaaaa

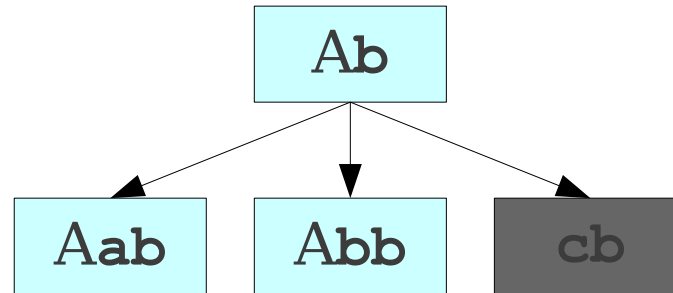
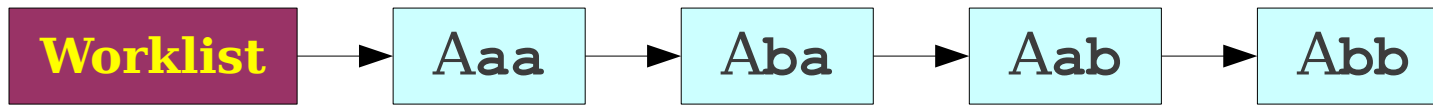
Leftmost BFS Has Problems



A → **Aa** | **Ab** | **c**

caaaaaaaaaa

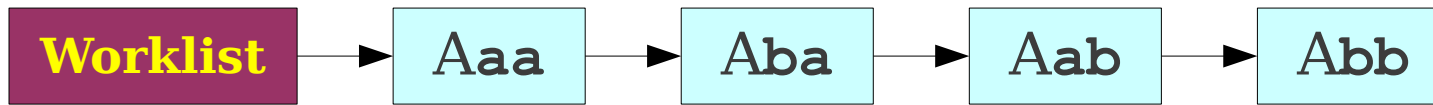
Leftmost BFS Has Problems



A → **Aa** | **Ab** | **c**

caaaaaaaaaa

Leftmost BFS Has Problems



A → **Aa** | **Ab** | **c**

caaaaaaaaaa

Leftmost BFS Has Problems



A → **Aa** | **Ab** | **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

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

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

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

E
T

`int + int`

Leftmost DFS

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

E
T
int

`int + int`

Leftmost DFS

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

E
T
int

int + int

Leftmost DFS

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

E
T

`int + int`

Leftmost DFS

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

E
T
(E)

`int + int`

Leftmost DFS

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

E
T
(E)

`int + int`

Leftmost DFS

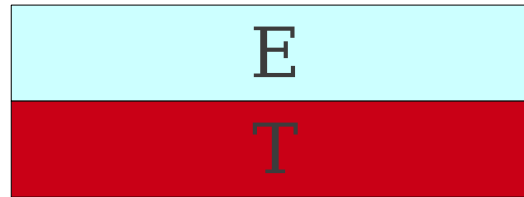
$E \rightarrow T$
 $E \rightarrow T + E$
 $T \rightarrow \text{int}$
 $T \rightarrow (E)$

E
T

`int + int`

Leftmost DFS

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



`int + int`

Leftmost DFS

E

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

`int + int`

Leftmost DFS

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

E
T + E

`int + int`

Leftmost DFS

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

E
T + E
int + E

int + int

Leftmost DFS

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

E
T + E
int + E
int + T

int + int

Leftmost DFS

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

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

int + int

Leftmost DFS

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

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

int + int



Problems with Leftmost DFS

A → **A****a** | **c**

A
Aa
Aaa
Aaaa
Aaaaa



c

Left Recursion

- A nonterminal **A** is said to be **left-recursive** iff

$$A \Rightarrow^* A\omega$$

for some string ω .

- Leftmost DFS may fail on left-recursive grammars.
- Fortunately, in many cases it is possible to eliminate left recursion

Summary of Leftmost BFS/DFS

- Leftmost BFS works on all grammars.
- Worst-case runtime is exponential.
- Worst-case memory usage is exponential.
- Rarely used in practice.
- Leftmost DFS works on grammars without left recursion.
- Worst-case runtime is exponential.
- Worst-case memory usage is linear.
- Often used in a limited form as **recursive descent**.

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.

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.

Predictive Parsing

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

E

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

E

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

E
T + E

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
------------	----------	----------	------------	----------	------------	----------

Predictive Parsing

E
T + E

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
------------	----------	----------	------------	----------	------------	----------

Predictive Parsing

E
T + E

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
------------	---	---	-----	---	-----	---

Predictive Parsing

E
T + E
int + E

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
------------	---	---	-----	---	-----	---

Predictive Parsing

E
T + E
int + E

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
------------	----------	----------	------------	----------	------------	----------

Predictive Parsing

E
T + E
int + E

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
------------	----------	----------	------------	----------	------------	----------

Predictive Parsing

E
T + E
int + E

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

E
T + E
int + E
int + T

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

E
T + E
int + E
int + T

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

E
T + E
int + E
int + T

E → **T**

E → **T** + **E**

T → **int**

T → (**E**)

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$
$\text{int} + (\text{int} + T)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$
$\text{int} + (\text{int} + T)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$
$\text{int} + (\text{int} + T)$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$
$\text{int} + (\text{int} + T)$
$\text{int} + (\text{int} + \text{int})$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$
$\text{int} + (\text{int} + T)$
$\text{int} + (\text{int} + \text{int})$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$
$\text{int} + (\text{int} + T)$
$\text{int} + (\text{int} + \text{int})$

int	+	(	int	+	int	)
-----	---	---	-----	---	-----	---

Predictive Parsing

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

E
$T + E$
$\text{int} + E$
$\text{int} + T$
$\text{int} + (E)$
$\text{int} + (T + E)$
$\text{int} + (\text{int} + E)$
$\text{int} + (\text{int} + T)$
$\text{int} + (\text{int} + \text{int})$



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.**

LL(1) Parse Tables

LL(1) Parse Tables

E $\rightarrow$ **int**

E $\rightarrow$ **(E Op E)**

Op $\rightarrow$ **+**

Op $\rightarrow$ *****

LL(1) Parse Tables

E $\rightarrow$ **int**

E $\rightarrow$ **(E Op E)**

Op $\rightarrow$ **+**

Op $\rightarrow$ *****

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

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** $\rightarrow$ int

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ +

(4) **Op** $\rightarrow$ *

	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** $\rightarrow$ int

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ +

(4) **Op** $\rightarrow$ *

	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

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

- (1) **E** $\rightarrow$ **int**
- (2) **E** $\rightarrow$ (**E Op E**)
- (3) **Op** $\rightarrow$ **+**
- (4) **Op** $\rightarrow$ *****

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

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.

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

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** → *****

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** → *****

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

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))\$

	int	(	)	+	*
E	1	2			
Op				3	4

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

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))\$

	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))\$
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** → *****

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	(	)	+	*
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))\$
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) Parsing

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

	int	(	)
E	1	2	
Op			



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

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

E\$	int + int\$
-----	-------------

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

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 \$	int + int\$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

E\$	int + int\$
int \$	int + int\$
\$	+ int\$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

E\$	int + int\$
int \$	int + int\$
\$	+ int\$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

(1) **E** $\rightarrow$ **int**

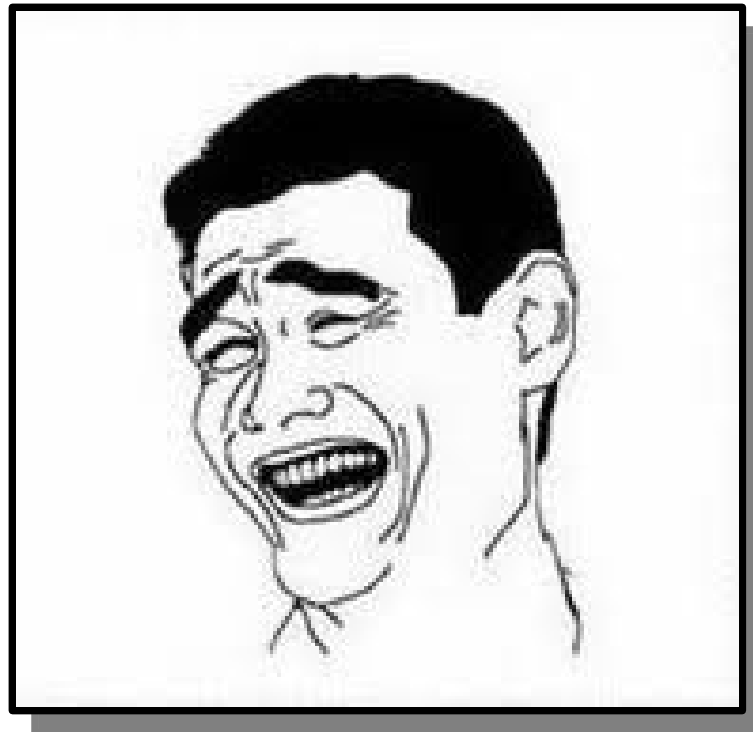
(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

E\$	int + int\$
int \$	int + int\$
\$	+ int\$

	int	(	)	+	*
E	1	2			
Op				3	4



LL(1) Error Detection, Part II

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

(int (int))\$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

E\$	(int (int))\$
-----	---------------

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

E\$	(int (int))\$
-----	---------------

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

E\$	(int (int))\$
(E Op E)\$	(int (int))\$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ **(E Op E)**

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

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** $\rightarrow$ **int**

(2) **E** $\rightarrow$ **(E Op E)**

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

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** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

E\$	(int (int))\$
(E Op E)\$	(int (int))\$
E Op E)\$	int (int))\$
int 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))\$
E Op E)\$	int (int))\$
int Op E)\$	int (int))\$
Op E)\$	(int))\$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) **E** $\rightarrow$ **int**

(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

E\$	(int (int))\$
(E Op E)\$	(int (int))\$
E Op E)\$	int (int))\$
int Op E)\$	int (int))\$
Op E)\$	(int))\$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

(1) **E** $\rightarrow$ **int**

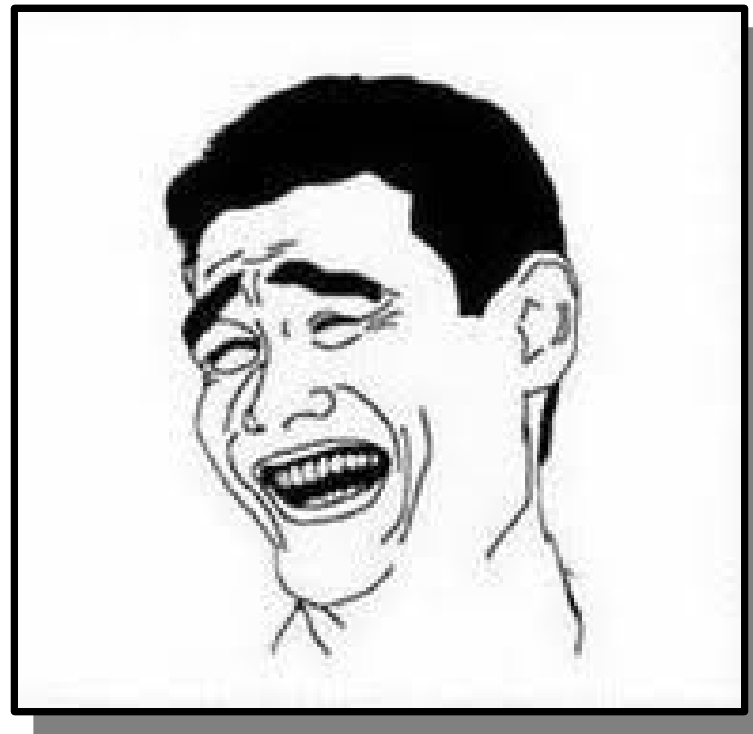
(2) **E** $\rightarrow$ (**E Op E**)

(3) **Op** $\rightarrow$ **+**

(4) **Op** $\rightarrow$ *****

E\$	(int (int))\$
(E Op E)\$	(int (int))\$
E Op E)\$	int (int))\$
int Op E)\$	int (int))\$
Op E)\$	(int))\$

	int	(	)	+	*
E	1	2			
Op				3	4



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 $T[\mathbf{A}, \mathbf{t}]$ is undefined, report an error.
 - Replace the top of the stack with $T[\mathbf{A}, \mathbf{t}]$.

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** **id** → **id**;
| **zero?** **TERM** **while** **not** **zero?** **id**
| **not** **EXPR** **do** **--id**;
| **++ id** **if** **not** **zero?** **id** **then**
| **-- id** **if** **not** **zero?** **id** **then**
 constant → **id**;

TERM → **id**
| **constant**

Constructing LL(1) Parse Tables

STMT $\rightarrow$ **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)

$$\begin{array}{lll} \text{TERM} & \rightarrow & \text{id} & (9) \\ & | & \text{constant} & (10) \end{array}$$
[illegible]

Constructing LL(1) Parse Tables

STMT $\rightarrow$ **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 $\rightarrow$ **id** (9)
 $\mid$ **constant** (10)

[illegible]

Constructing LL(1) Parse Tables

STMT $\rightarrow$ **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

|

constant

(9)

(10)

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

Constructing LL(1) Parse Tables

STMT $\rightarrow$ **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)

$$\begin{array}{lll} \text{TERM} & \rightarrow & \text{id} & (9) \\ & | & \text{constant} & (10) \end{array}$$

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

Constructing LL(1) Parse Tables

STMT $\rightarrow$ **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)

$$\begin{array}{lll} \text{TERM} & \rightarrow & \text{id} & (9) \\ & | & \text{constant} & (10) \end{array}$$
[illegible]

Constructing LL(1) Parse Tables

STMT $\rightarrow$ **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 $\rightarrow$ **id** (9)
 $\mid$ **constant** (10)

[illegible]

Constructing LL(1) Parse Tables

STMT $\rightarrow$ **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 $\rightarrow$ **id** (9)
 $\mid$ **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 $\rightarrow$ **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)

$$\begin{array}{lcl} \text{TERM} & \rightarrow & \text{id} & (9) \\ & | & \text{constant} & (10) \end{array}$$
[illegible]

Constructing LL(1) Parse Tables

STMT $\rightarrow$ **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)

$$\begin{array}{lll} \text{TERM} & \rightarrow & \text{id} & (9) \\ & | & \text{constant} & (10) \end{array}$$
[illegible]

Constructing LL(1) Parse Tables

$$\text{STMT} \rightarrow \text{if } \text{EXPR} \text{ then } \text{STMT} \quad (1)$$

while **EXPR** **do** **STMT** **(2)**

$$| \text{EXPR} ; \quad (3)$$
$$\text{EXPR} \rightarrow \text{TERM} \rightarrow \text{id} \quad (4)$$
$$\text{zero? TERM} \quad (5)$$

$$\text{not } \text{EXPR} \quad (6)$$

$$| \quad ++ \text{ id} \quad (7)$$
$$| \quad -- id \qquad \qquad \qquad (8)$$
$$\text{TERM} \rightarrow \text{id} \quad (9)$$
$$| \text{constant} \quad (10)$$
[illegible]

Constructing LL(1) Parse Tables

$$\text{STMT} \rightarrow \text{if } \text{EXPR} \text{ then } \text{STMT} \quad (1)$$

while **EXPR** **do** **STMT** **(2)**

$$| \text{EXPR} ; \quad (3)$$
$$\mathbf{EXPR} \rightarrow \mathbf{TERM} \rightarrow \text{id} \quad (4)$$
$$\text{zero? TERM} \quad (5)$$

$$\text{not } \text{EXPR} \quad (6)$$

$$| \quad ++ \text{ id} \quad (7)$$
$$| \quad -- id \qquad \qquad \qquad (8)$$
$$\text{TERM} \rightarrow \text{id} \quad (9)$$
$$| \text{constant} \quad (10)$$
[illegible]

Constructing LL(1) Parse Tables

STMT $\rightarrow$ **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

|

constant

(9)

(10)

[illegible]

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)

[illegible]

Constructing LL(1) Parse Tables

STMT $\rightarrow$ **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)

$$\begin{array}{lll} \text{TERM} & \rightarrow & \text{id} & (9) \\ & | & \text{constant} & (10) \end{array}$$
[illegible]

Can we find an algorithm for constructing LL(1) parse tables?

Filling in Table Entries

- Intuition: The next character should uniquely identify a production, so we should pick a production that ultimately starts with that character.
- $T[A, t]$ should be a production $A \rightarrow \omega$ iff ω derives something starting with t .
- More rigorously:
$$T[A, t] = B\omega \text{ iff } A \rightarrow \omega \text{ and } \omega \Rightarrow^* t\omega'$$

- Algorithm of constructing the parse table for LL(1) in the book

The Limits of LL(1)

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

$$A \rightarrow Ab \mid c$$

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

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

$$A \rightarrow Ab \mid c$$

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

	b	c
A		$A \rightarrow Ab$ $A \rightarrow c$

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

$$A \rightarrow Ab \mid c$$

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

	b	c
A		$A \rightarrow Ab$ $A \rightarrow c$

- 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

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

- This will produce α followed by some number of ω 's.
- Can rewrite the grammar as

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

$$\mathbf{B} \rightarrow \epsilon \mid \omega \mathbf{B}$$

LL(1) Straightforward and Fast

- Cant 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

Summary

- Top-down parsing tries to derive the user's program from the start symbol.
- Leftmost BFS is one approach to top-down parsing; it is mostly of theoretical interest.
- Leftmost DFS is another approach to top-down parsing that is uncommon in practice.
- LL(1) parsing scans from left-to-right, using one token of lookahead to find a leftmost derivation.
-