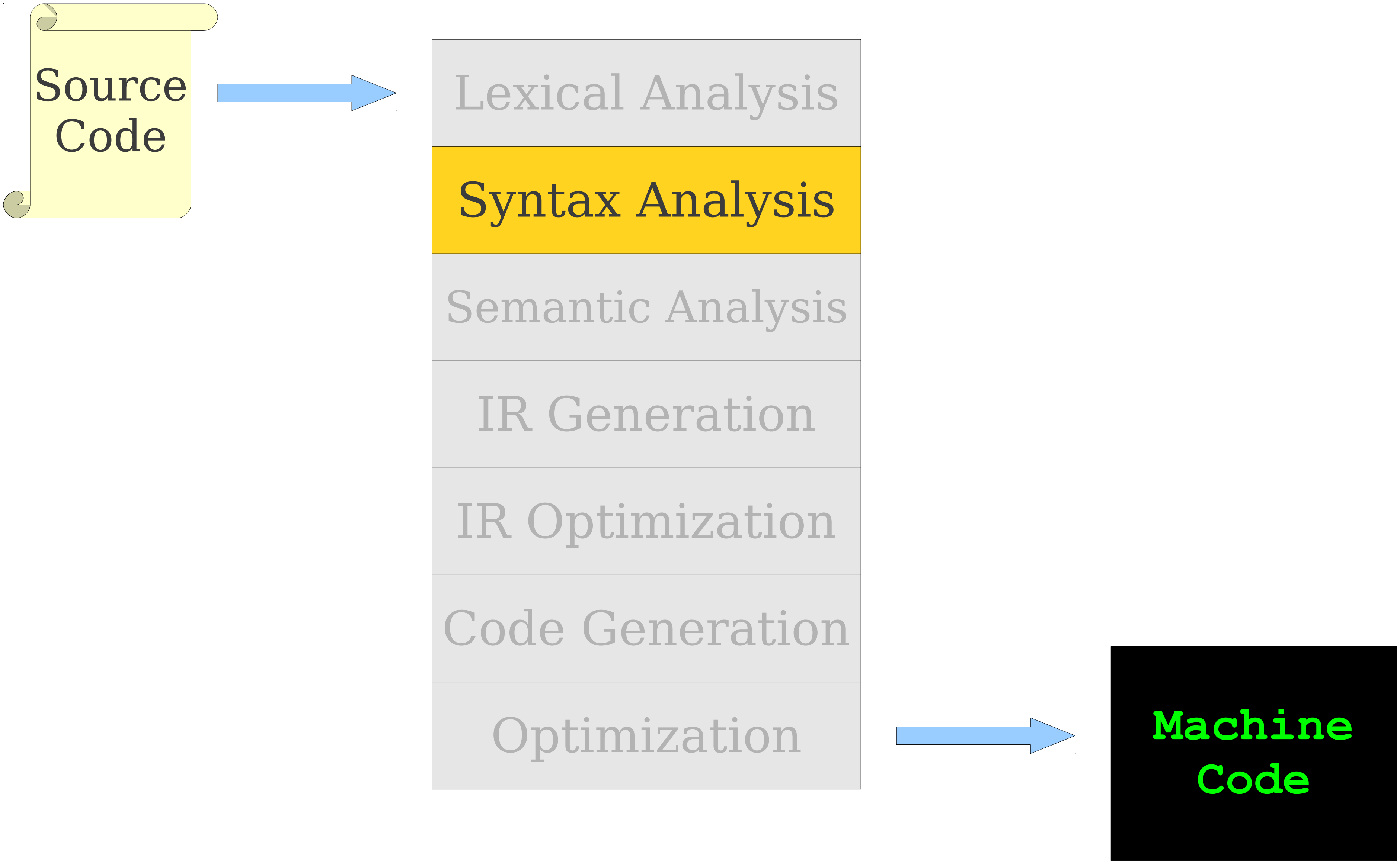
Top-Down Parsing

Where We Are


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

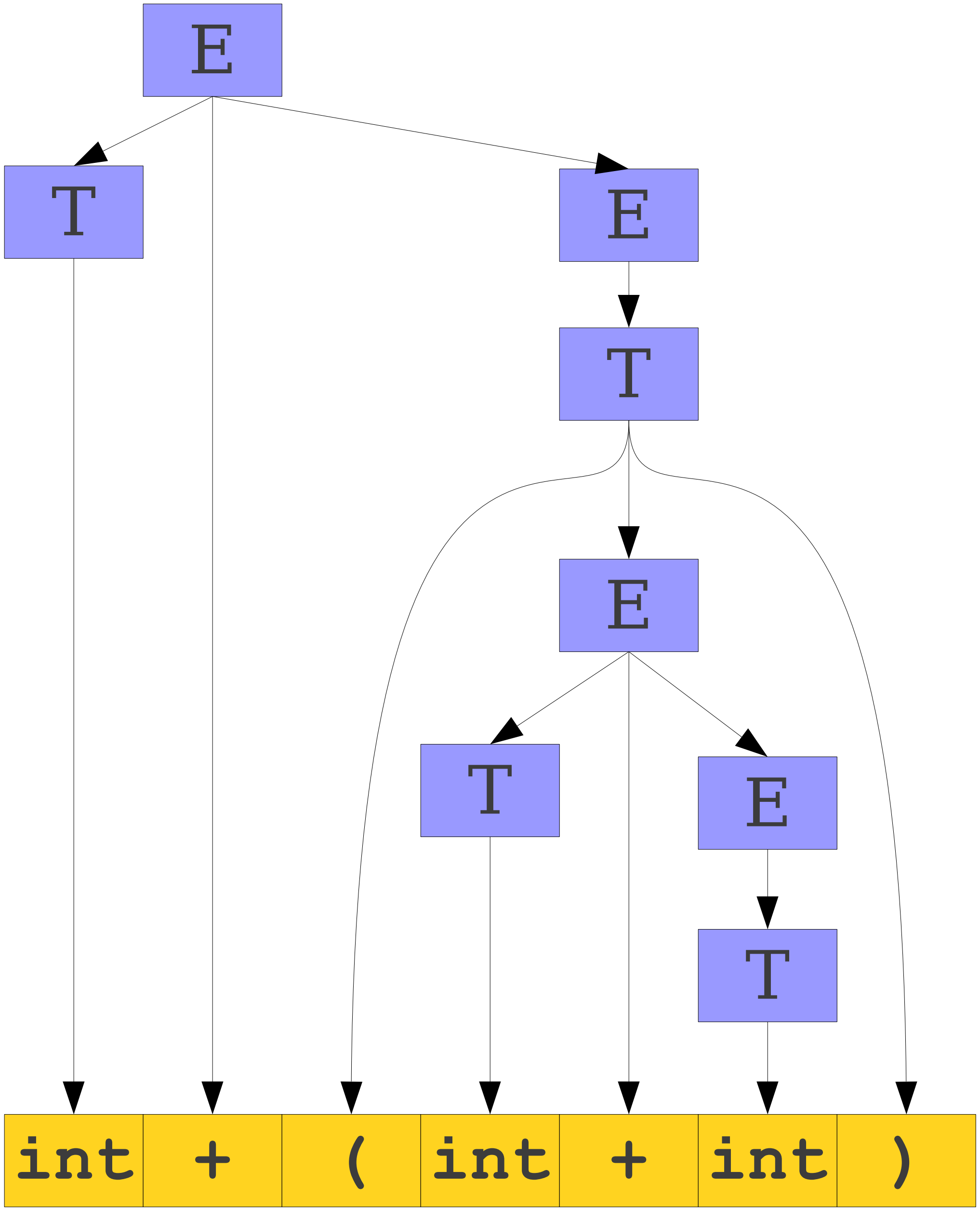
Top-Down Parsing

$E \rightarrow T$

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

$T \rightarrow$ **int**

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



Challenges in Top-Down Parsing

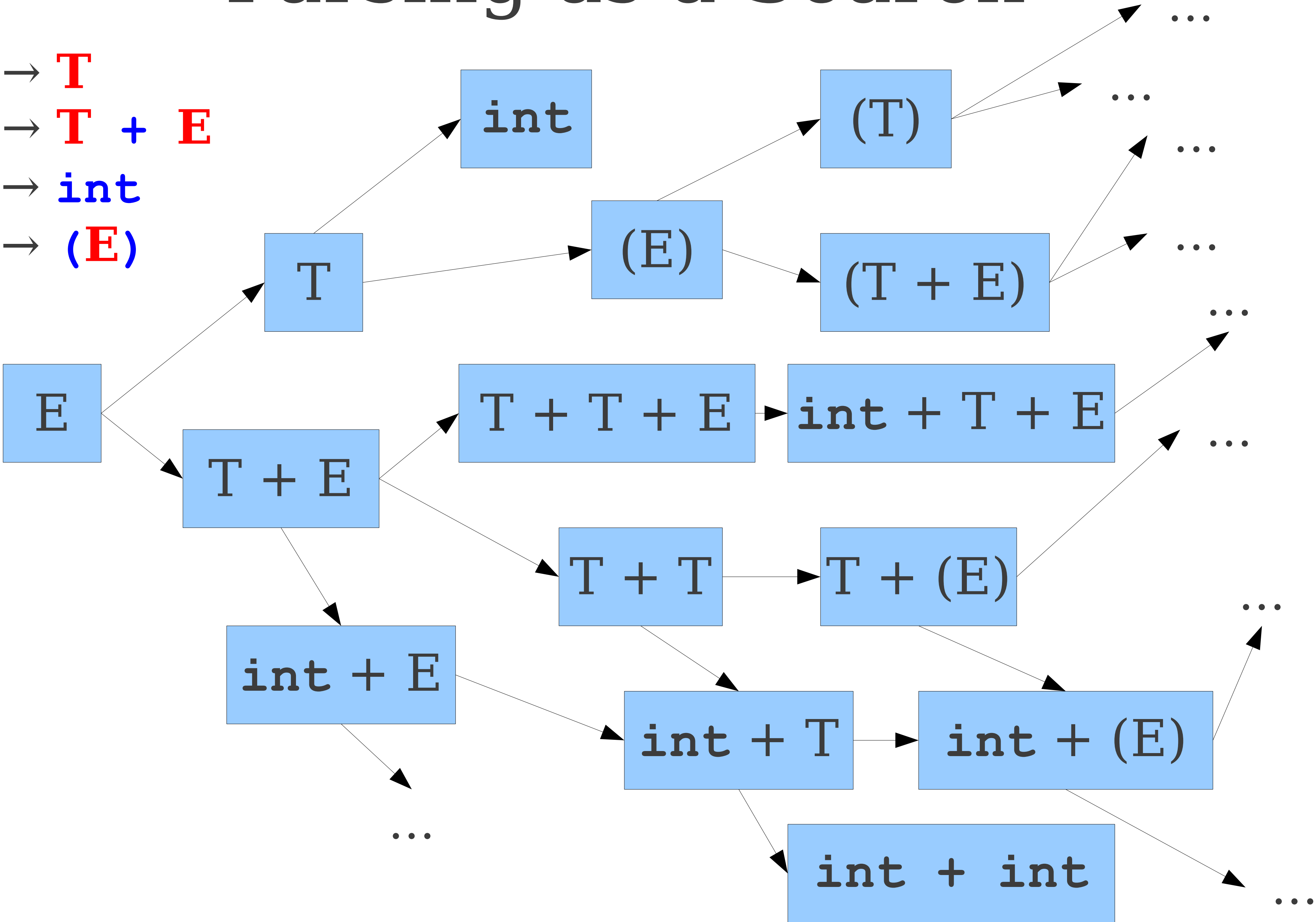
- Top-down parsing begins with virtually no information.
  - Begins with just the start symbol, which matches *every* program.
- How can we know which productions to apply?
- In general, we can't.
  - There are some grammars for which the best we can do is guess and backtrack if we're wrong.
  - If we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

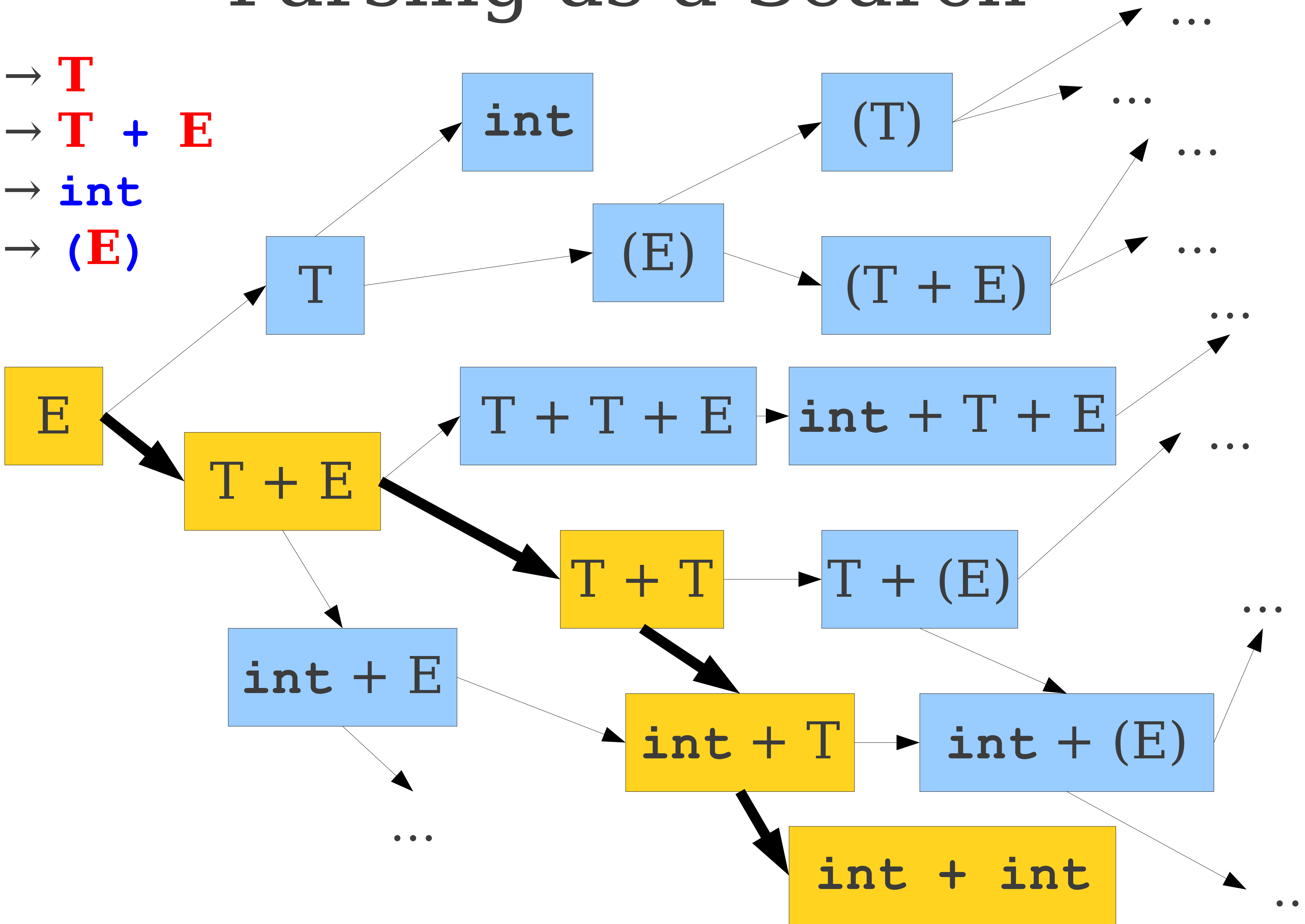
Parsing as a Search

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



Parsing as a Search

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



Our First Top-Down Algorithm

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

Breadth-First Search Parsing

Worklist

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

`int + int`

Breadth-First Search Parsing

Worklist → E

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

`int + int`

Breadth-First Search Parsing

Worklist

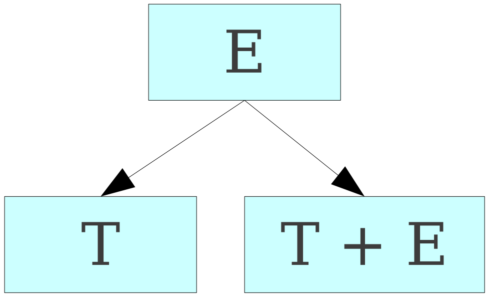
E

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

int + int

Breadth-First Search Parsing

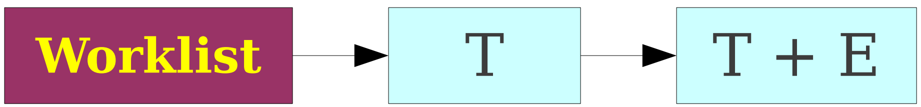
Worklist

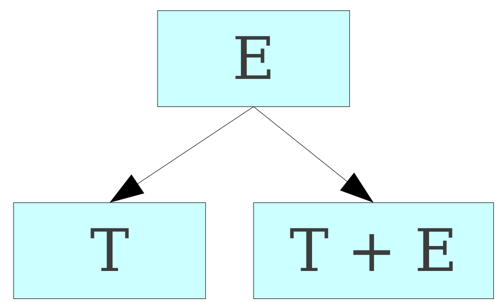


$E \rightarrow T$
$E \rightarrow T$ **+** E
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**

int + int

Breadth-First Search Parsing

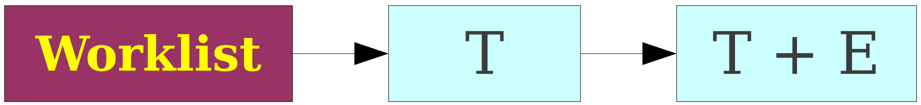




E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

`int + int`

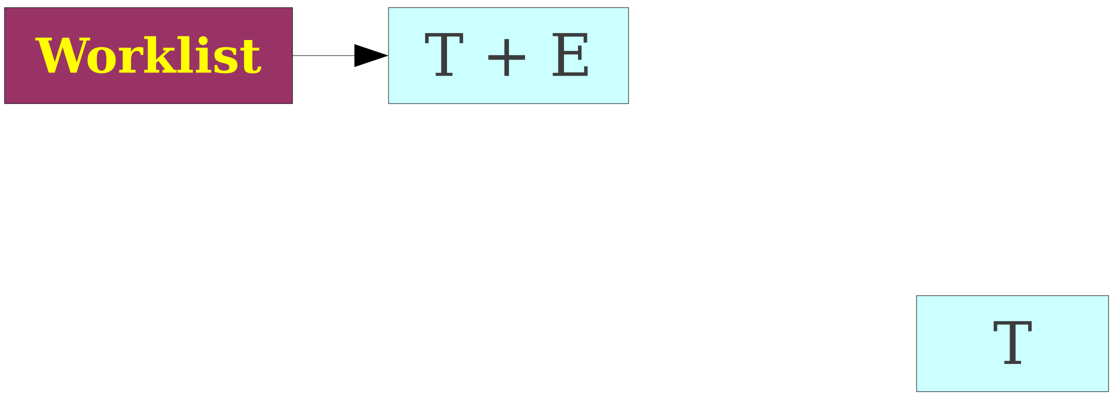
Breadth-First Search Parsing



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

`int + int`

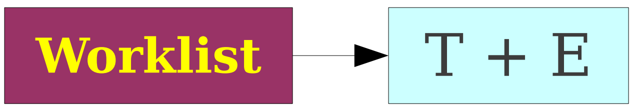
Breadth-First Search Parsing

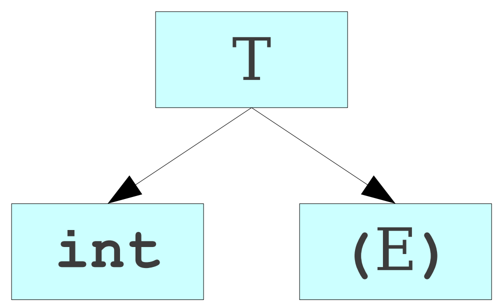


$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**

int + int

Breadth-First Search Parsing

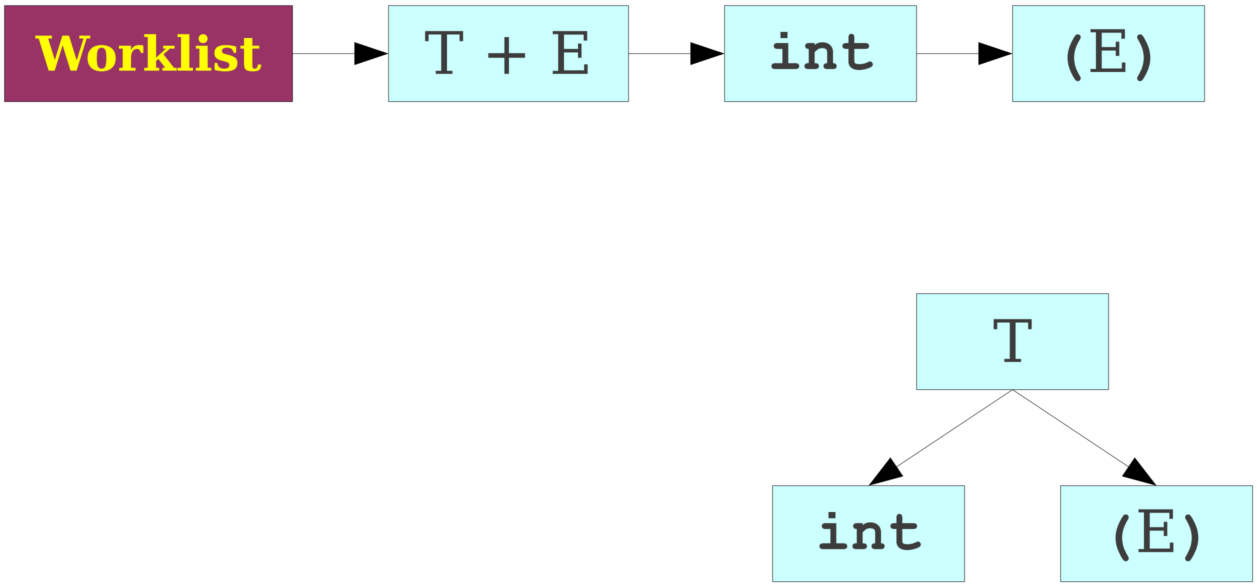




E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

`int + int`

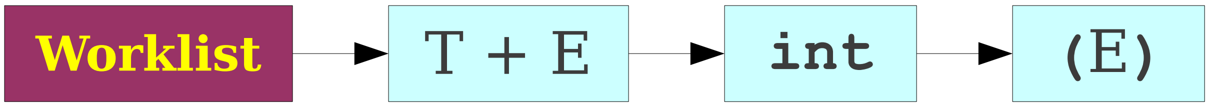
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing



E → **T**

E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

Breadth-First Search Parsing

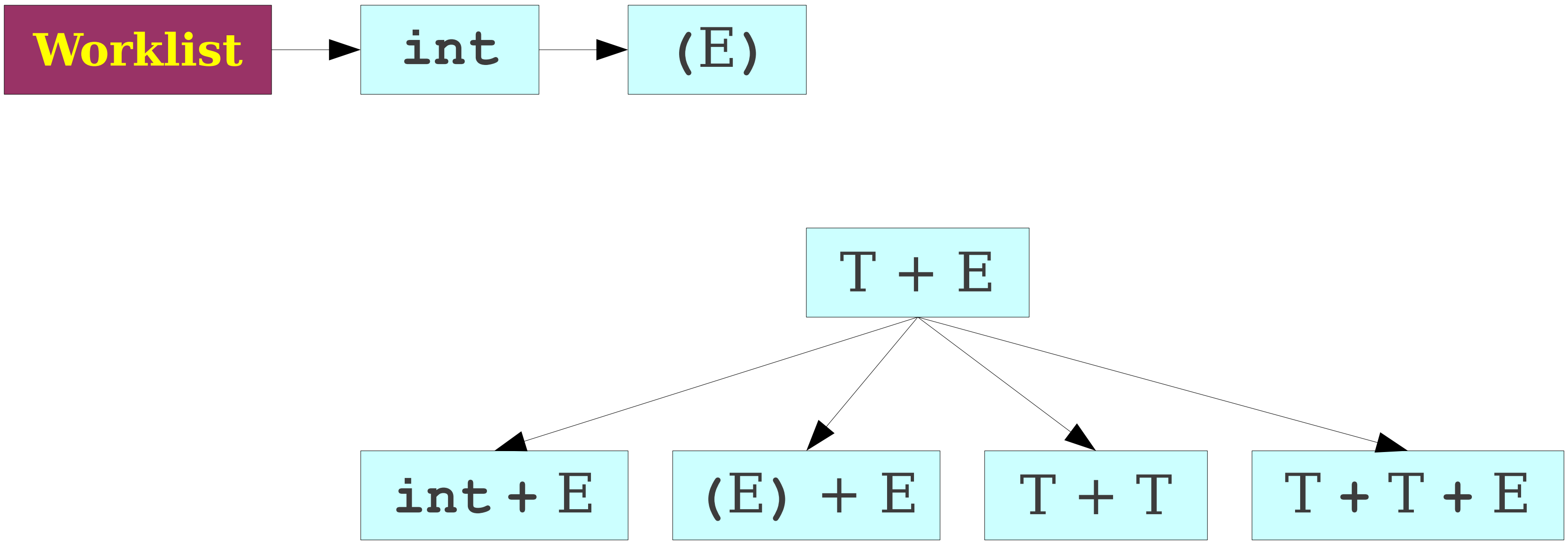
Worklist	→	`int`	→	(E)

T + E

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

`int + int`

Breadth-First Search Parsing



E → **T**

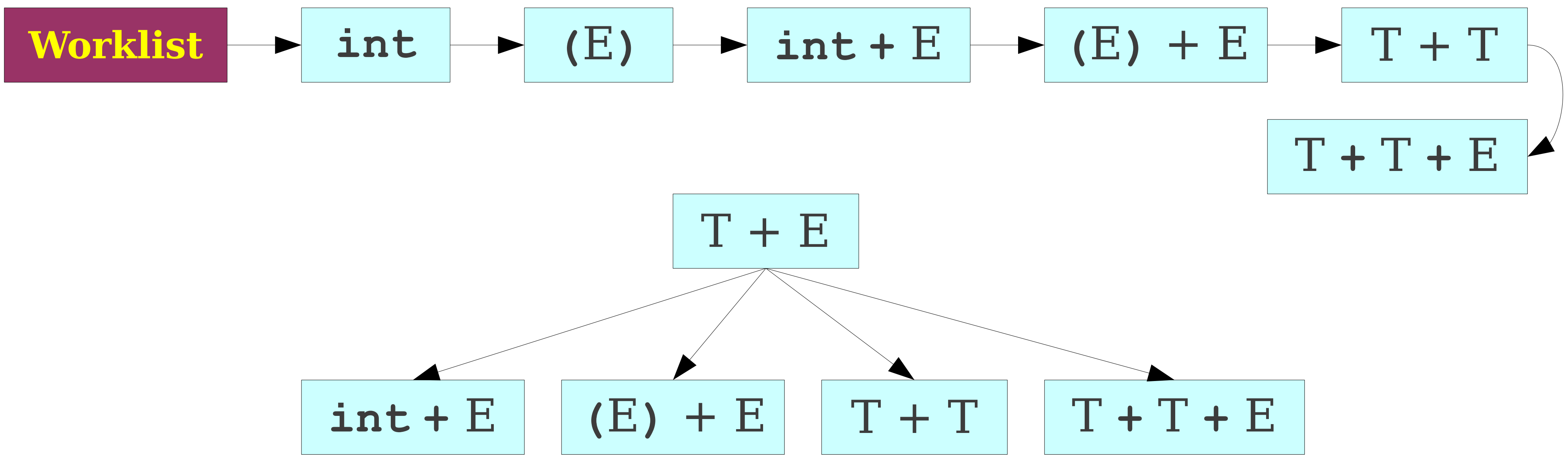
E → **T** + **E**

T → int

T → (**E**)

int + int

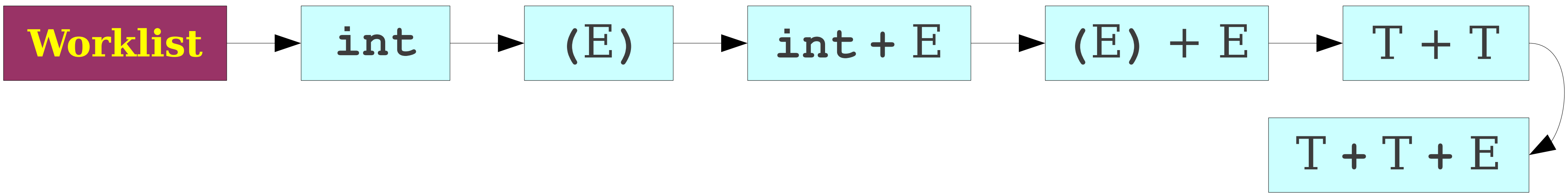
Breadth-First Search Parsing



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

int + int

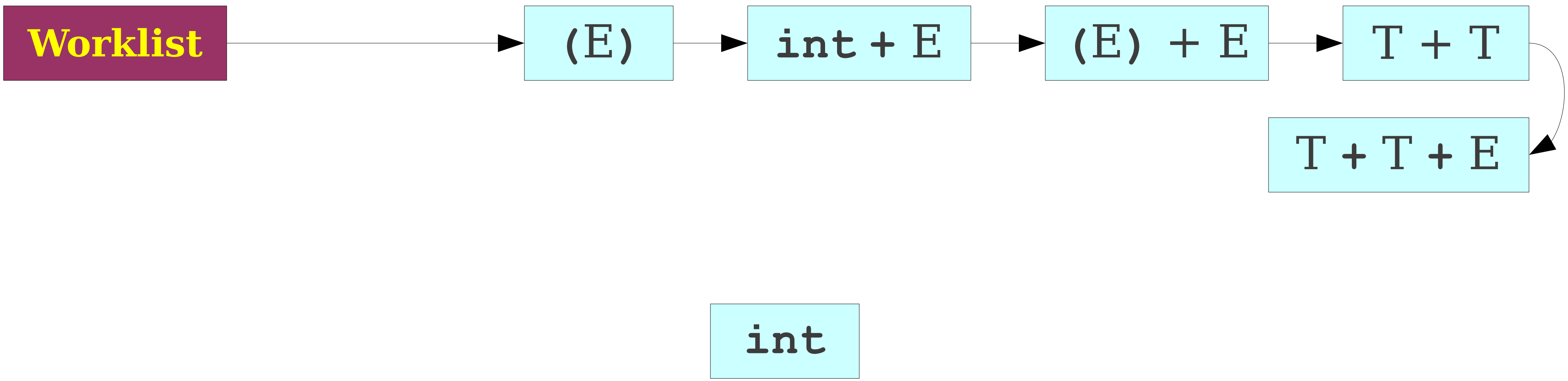
Breadth-First Search Parsing



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

int + int

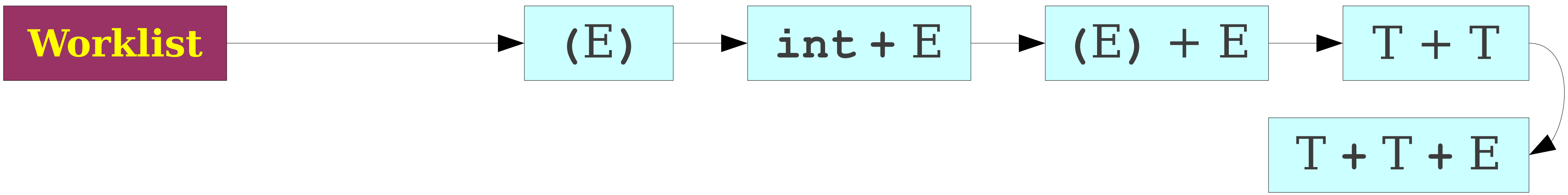
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing

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

(E)

$E \to T$

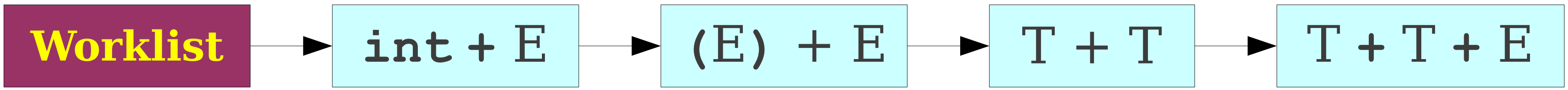
$E \to T \texttt{ + } E$

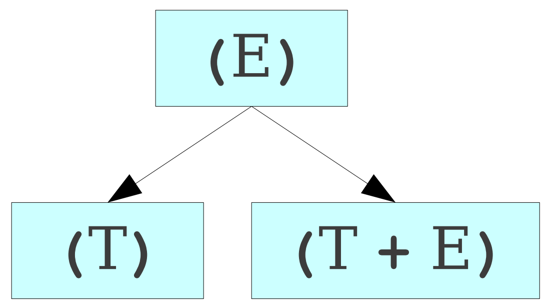
$T \to \texttt{int}$

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

`int + int`

Breadth-First Search Parsing

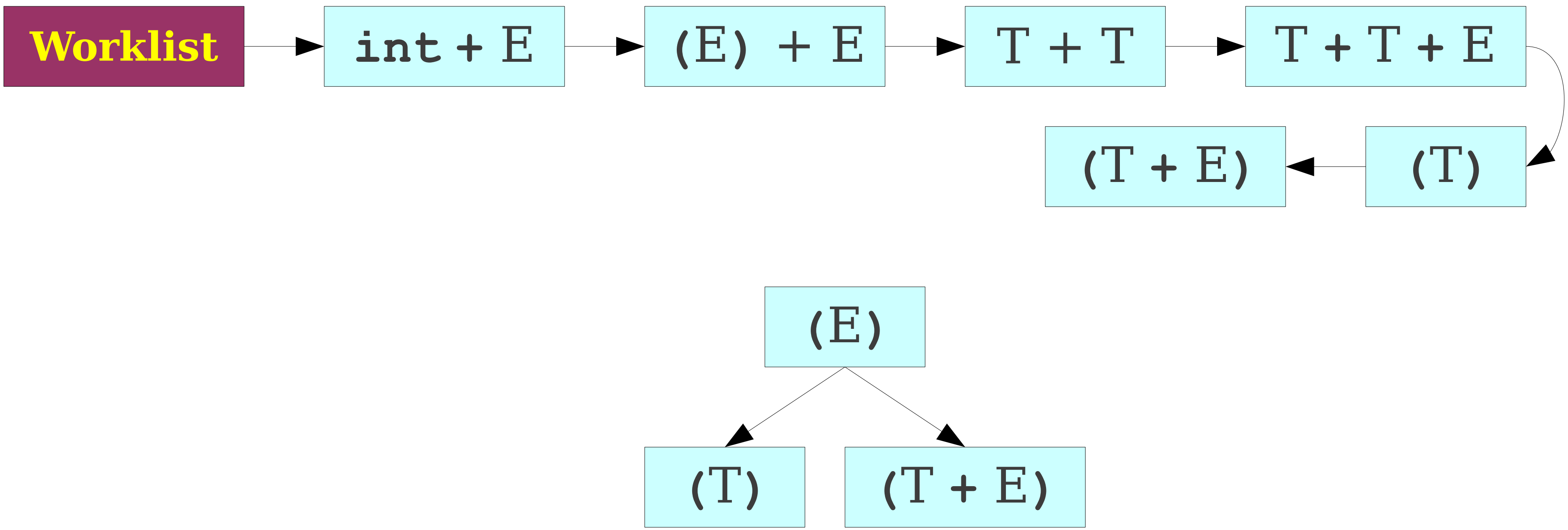




E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

`int + int`

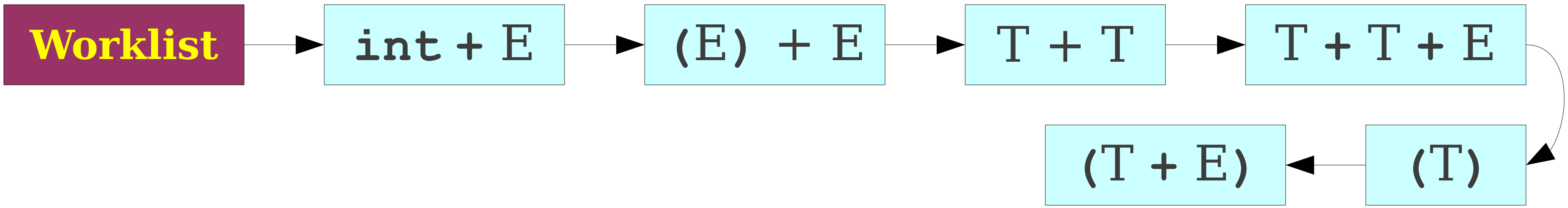
Breadth-First Search Parsing



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

`int + int`

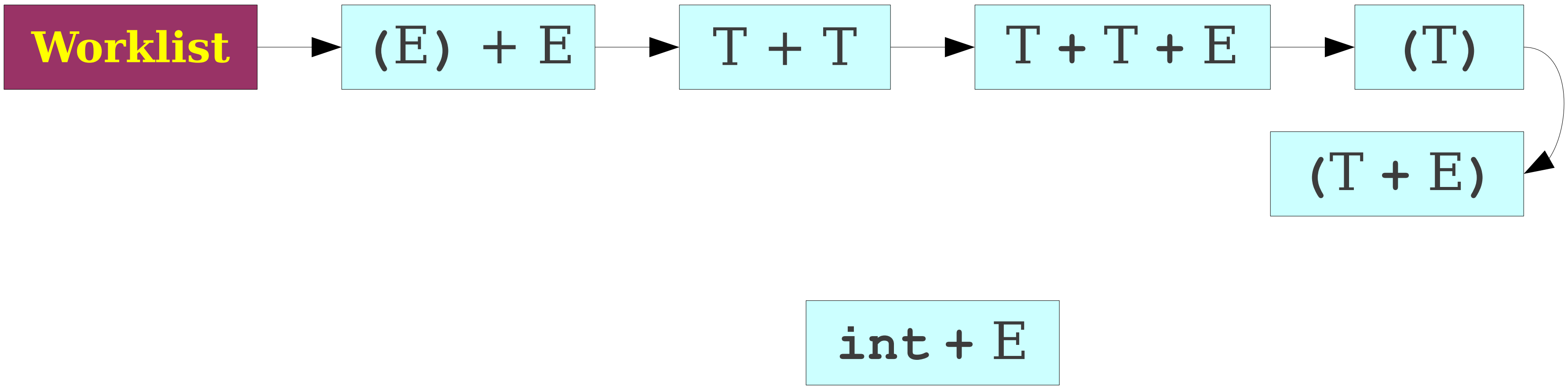
Breadth-First Search Parsing



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

int + int

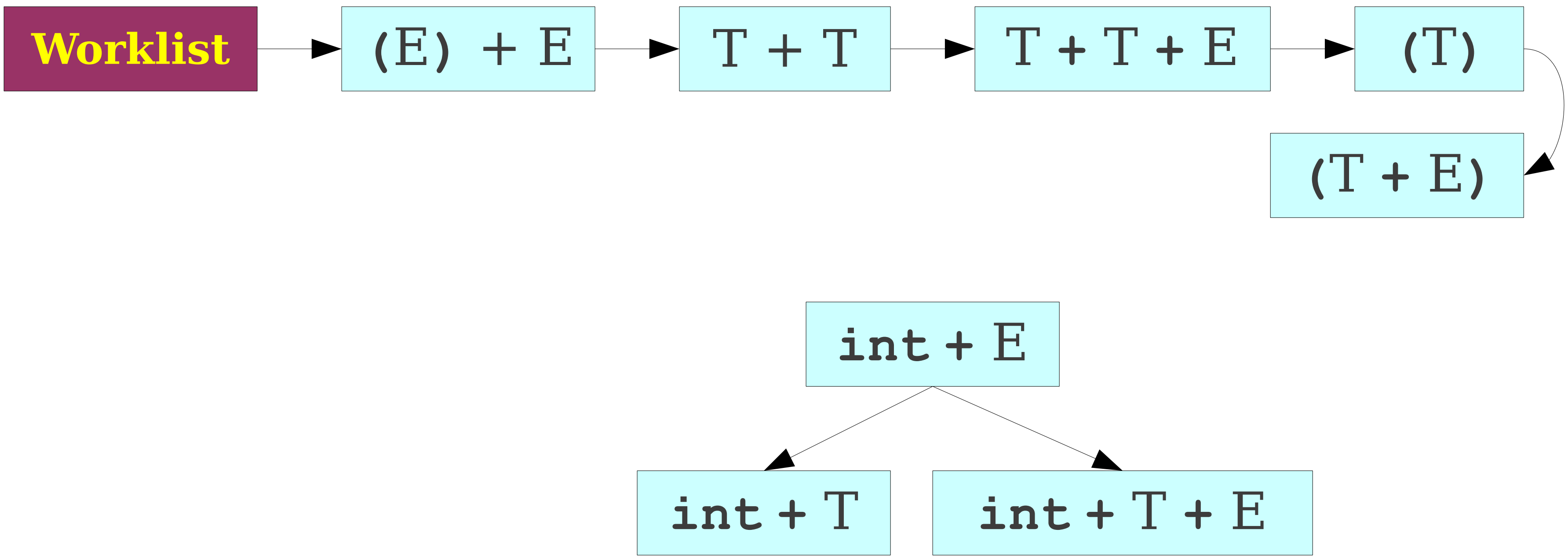
Breadth-First Search Parsing



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

int + int

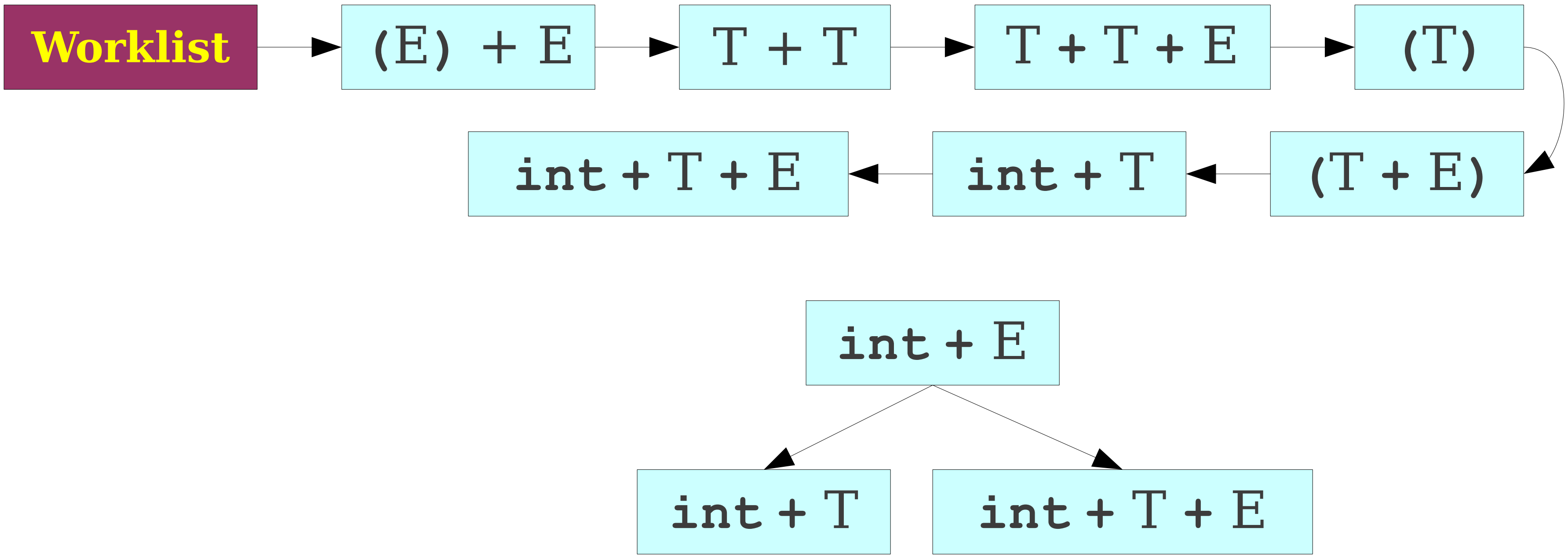
Breadth-First Search Parsing



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

int + int

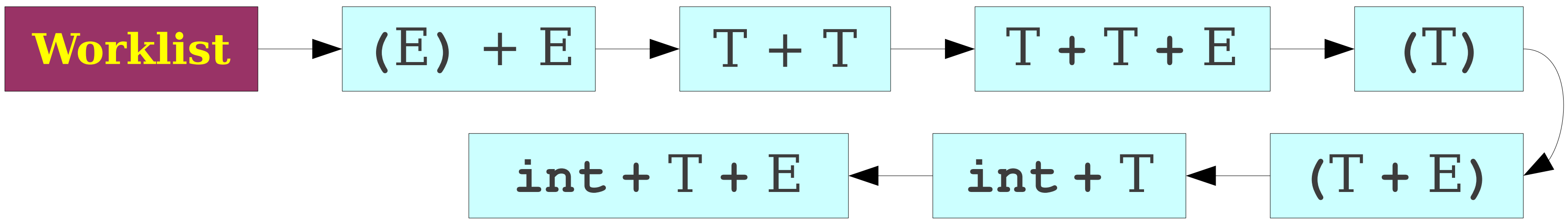
Breadth-First Search Parsing



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

int + int

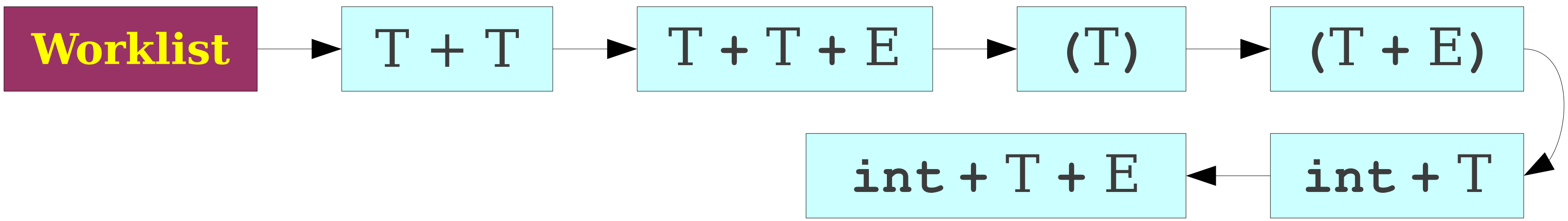
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing

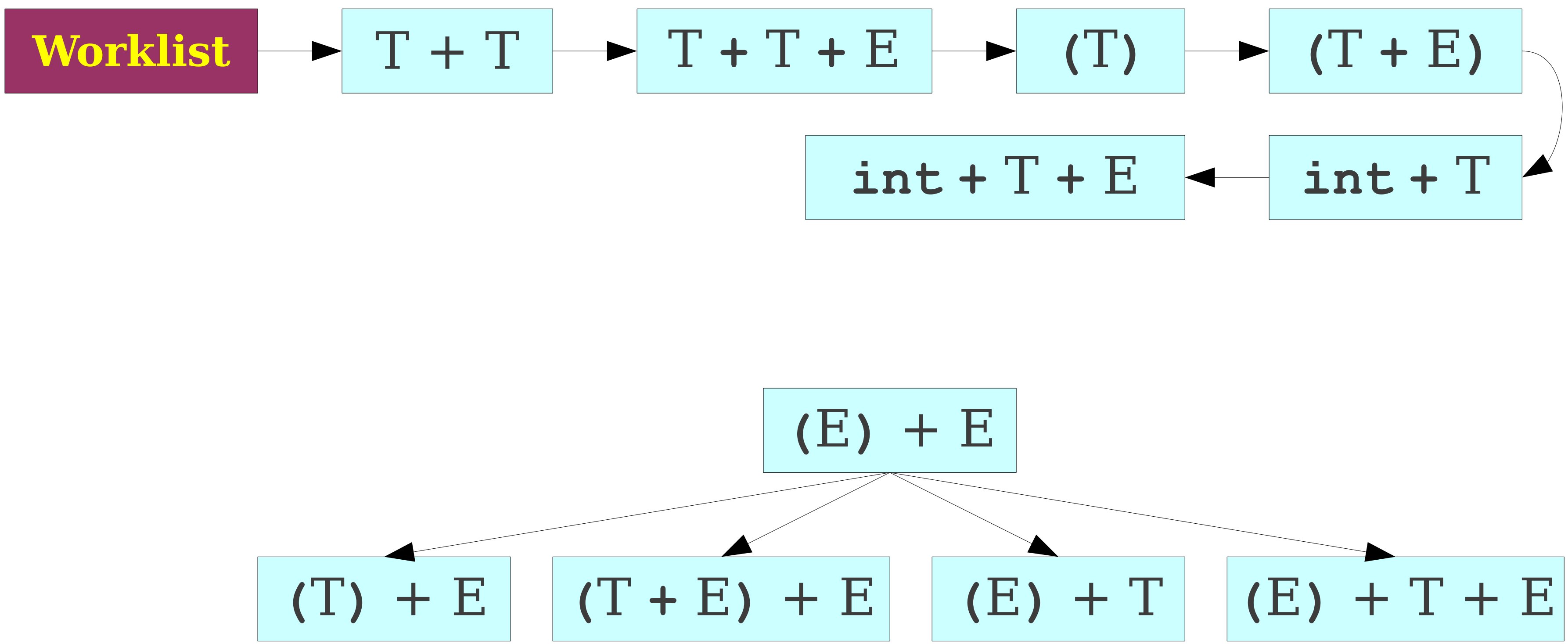


(E) + E

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

`int + int`

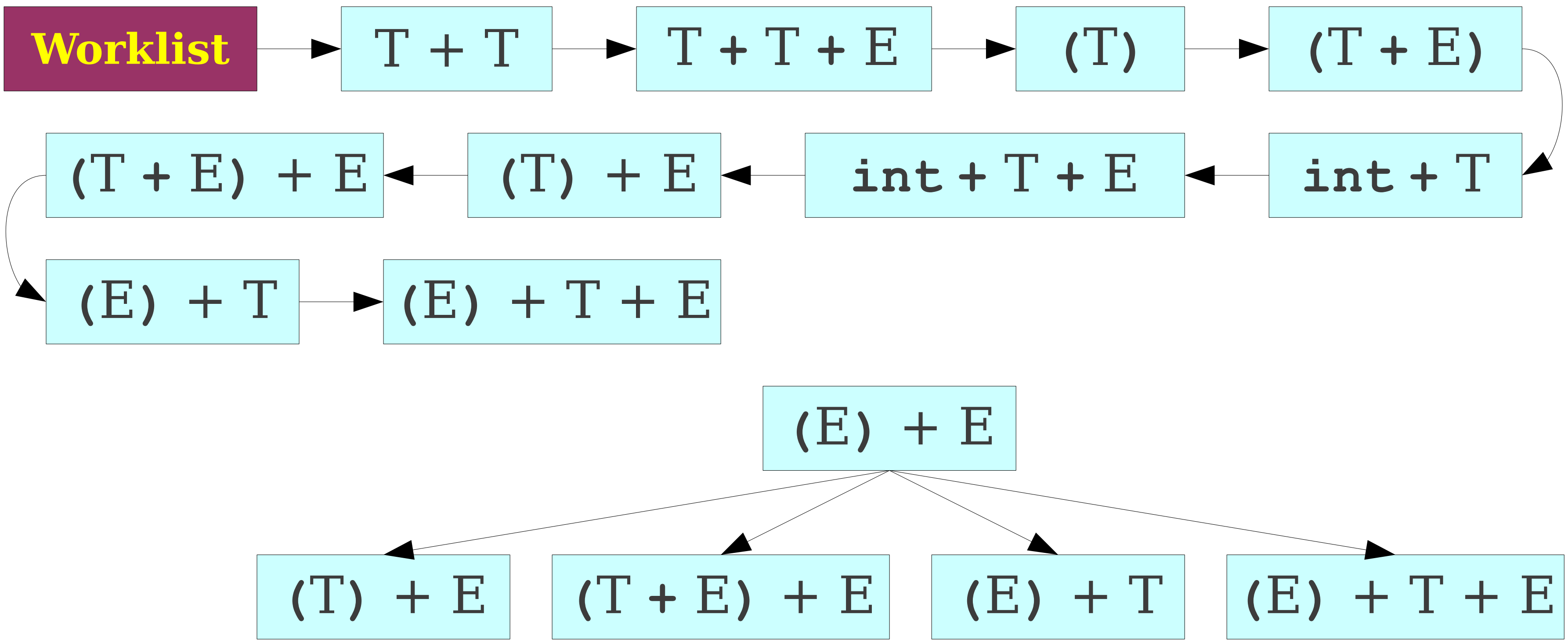
Breadth-First Search Parsing



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

`int + int`

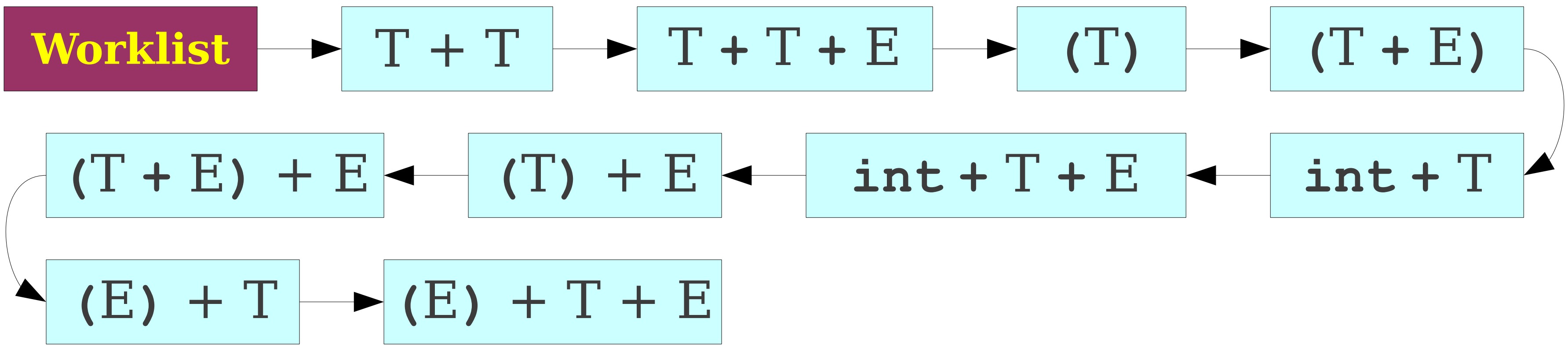
Breadth-First Search Parsing



E → T
E → T + E
T → int
T → (E)

int + int

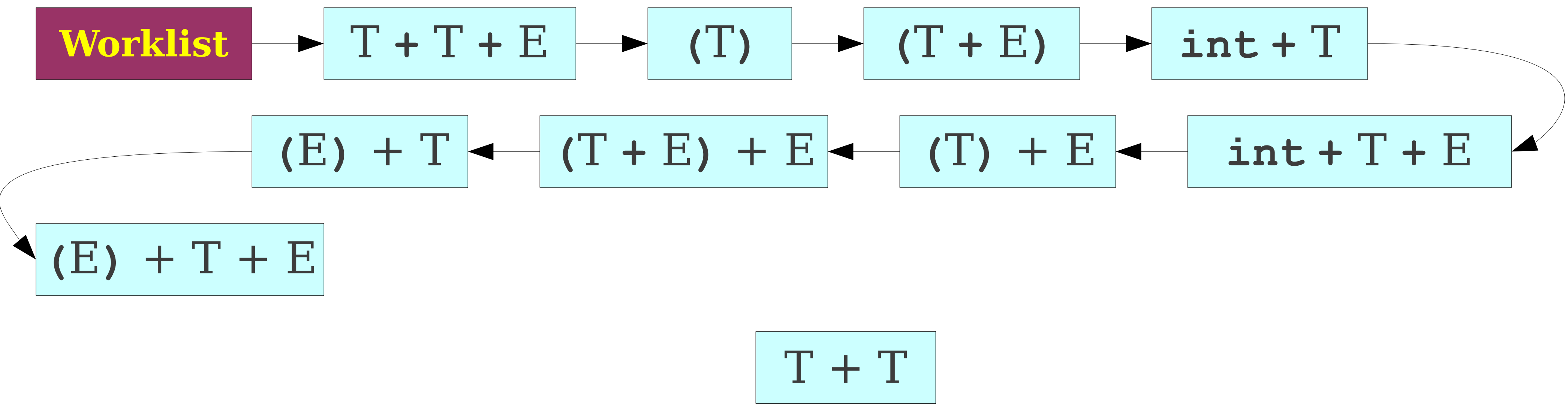
Breadth-First Search Parsing



E → T
E → T + E
T → int
T → (E)

int + int

Breadth-First Search Parsing



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

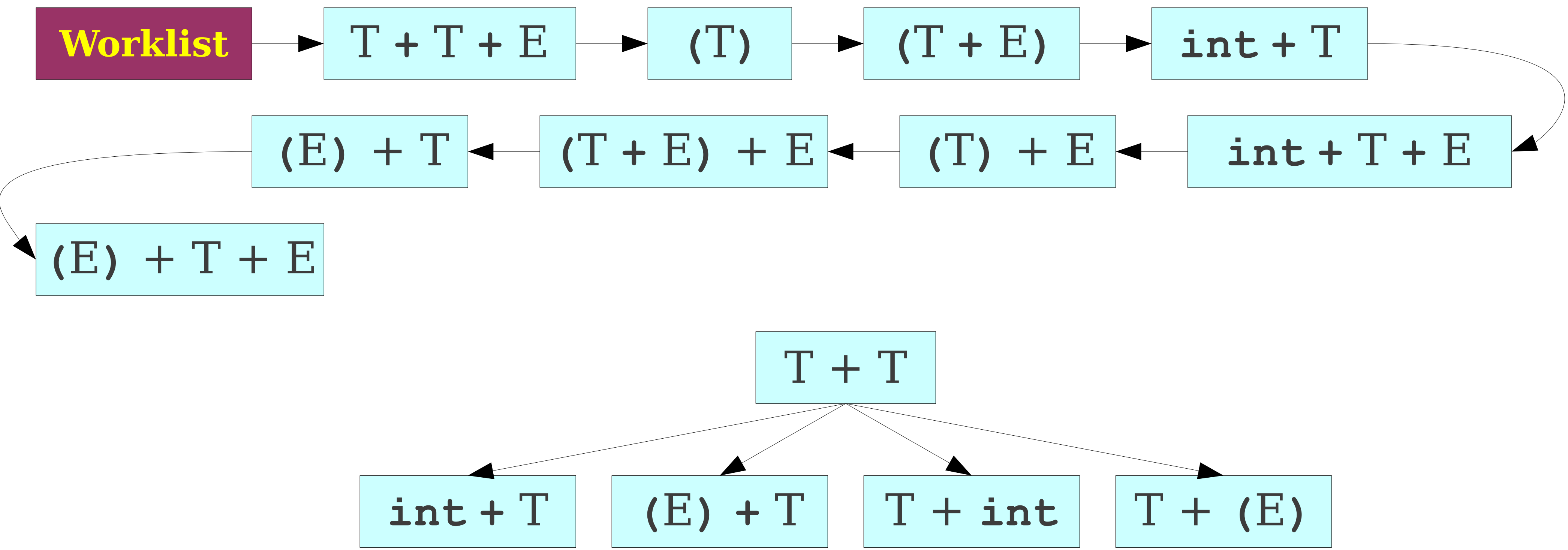
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

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

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

`int + int`

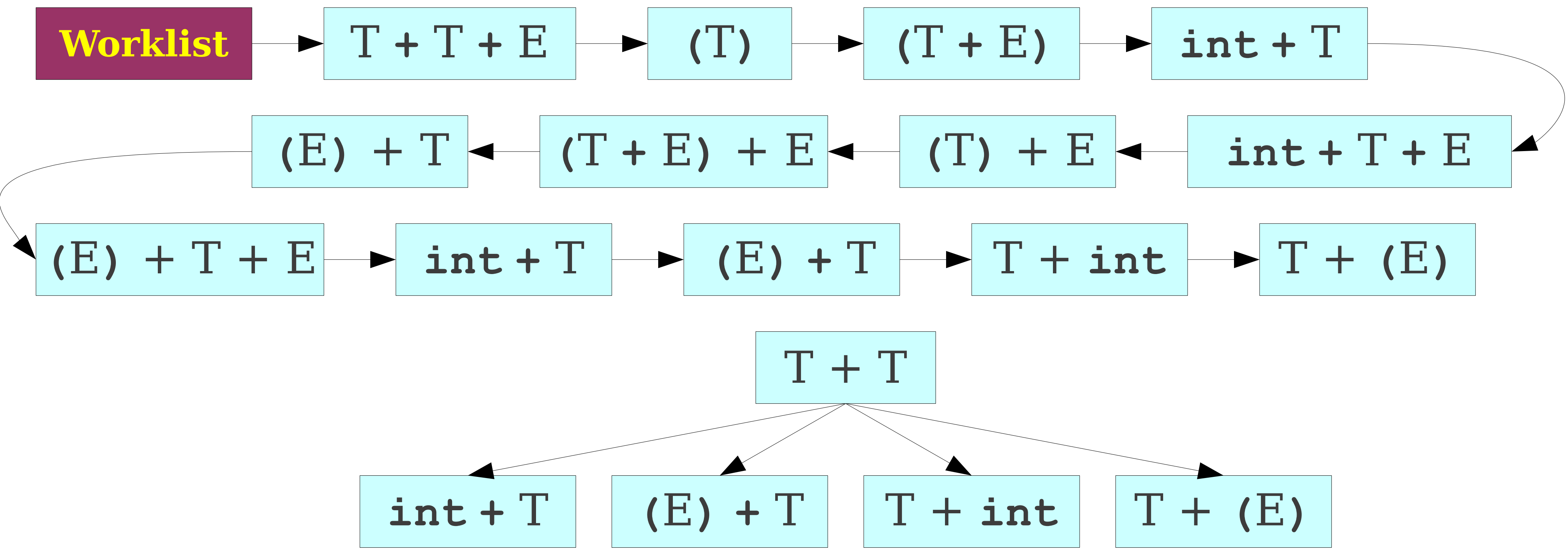
Breadth-First Search Parsing



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

int + int

Breadth-First Search Parsing



$E \rightarrow T$

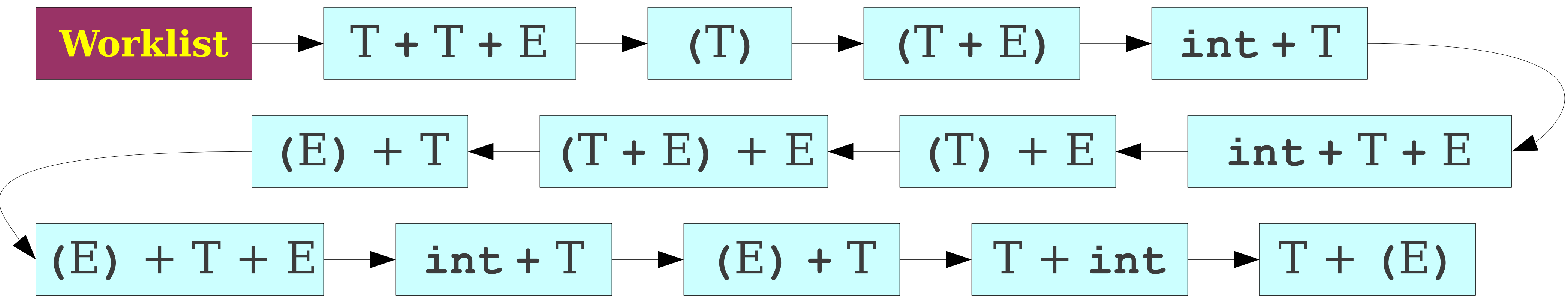
$E \rightarrow T$ + E

$T \rightarrow$ int

$T \rightarrow$ (E)

int + int

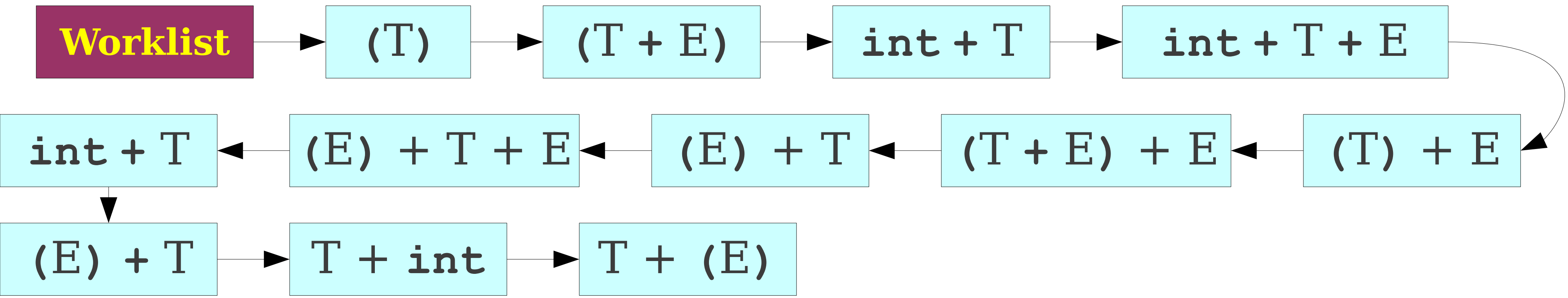
Breadth-First Search Parsing



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

int + int

Breadth-First Search Parsing

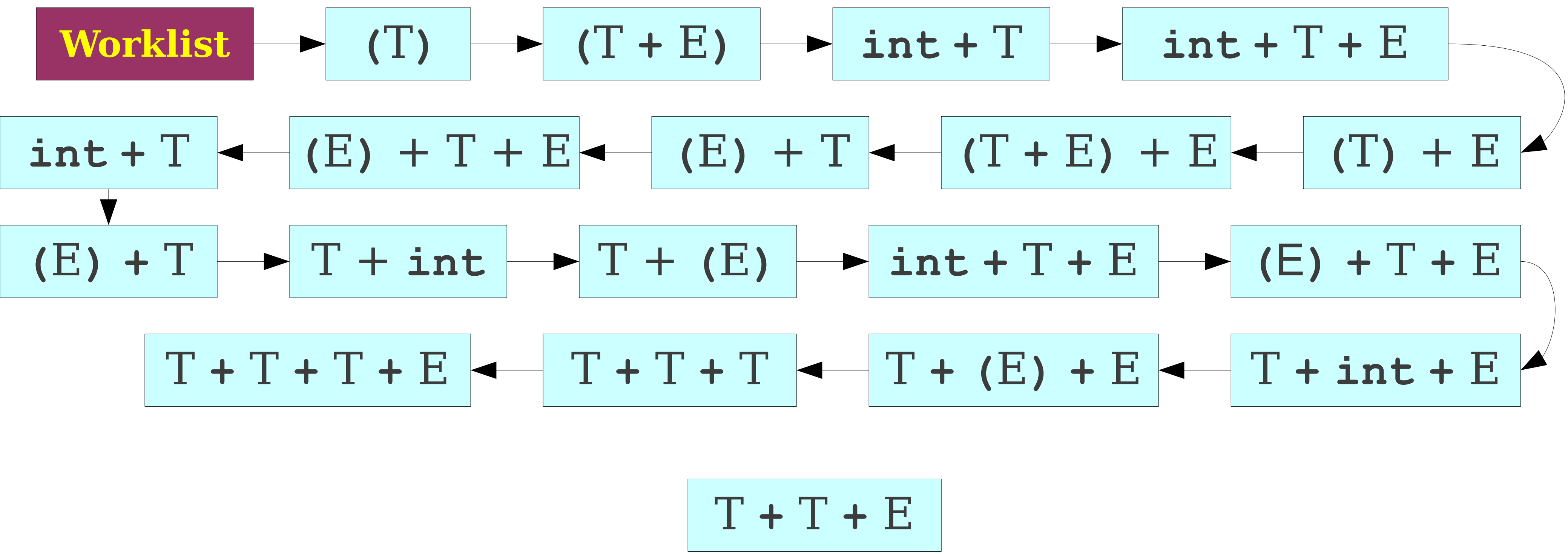


T + T + E

E → T
E → T + E
T → int
T → (E)

int + int

Breadth-First Search Parsing



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

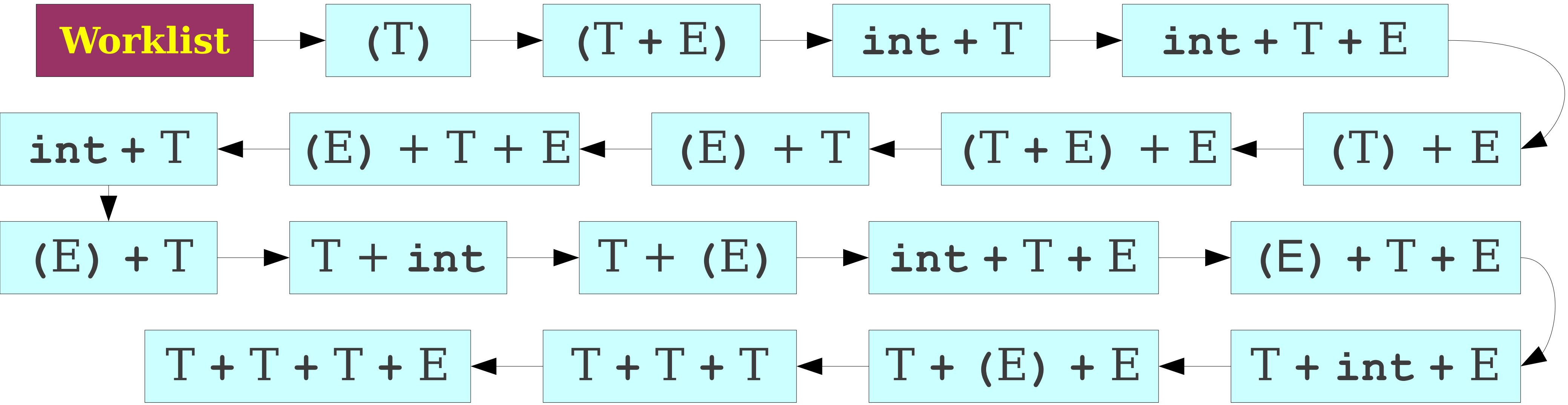
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

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

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

`int + int`

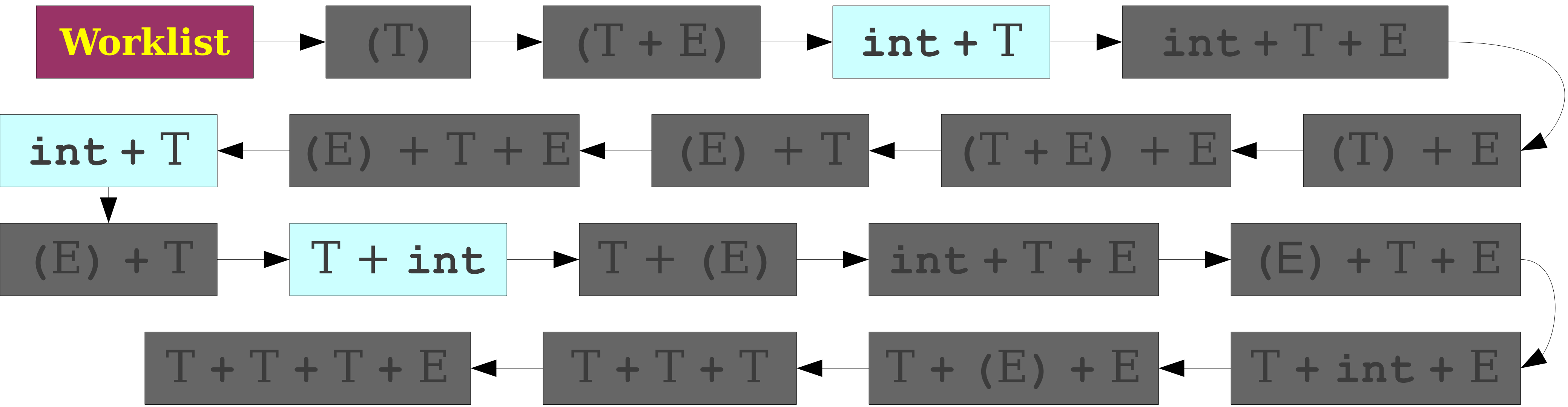
Breadth-First Search Parsing



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

Reducing the Branching Factor

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

Leftmost Derivations

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

Leftmost BFS

Worklist → E

$E \rightarrow T$

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

$T \rightarrow$ **int**

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

int + int

Leftmost BFS

Worklist

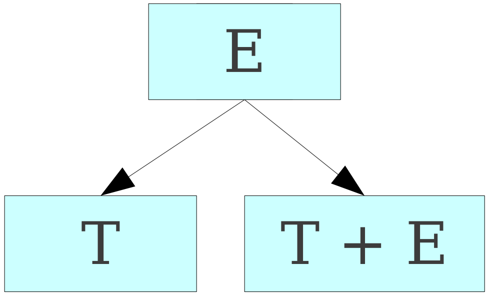
E

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

`int + int`

Leftmost BFS

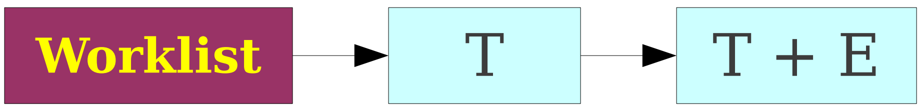
Worklist

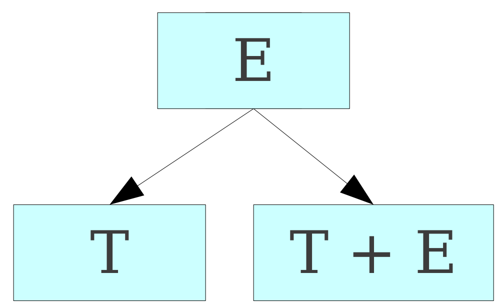


$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$
$\mathbf{T} \rightarrow$ `int`
$\mathbf{T} \rightarrow$ `(`$\mathbf{E}$`)`

`int + int`

Leftmost BFS





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

int + int

Leftmost BFS

Worklist → T → T + E

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

`int + int`

Leftmost BFS

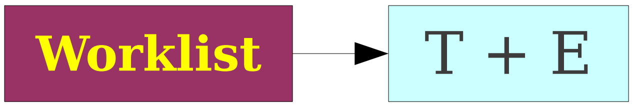
Worklist → T + E

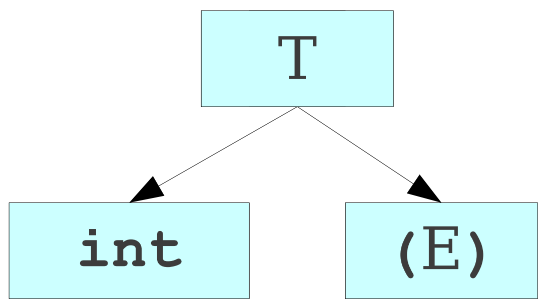
T

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

int + int

Leftmost BFS

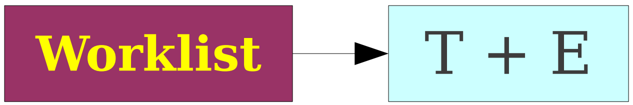


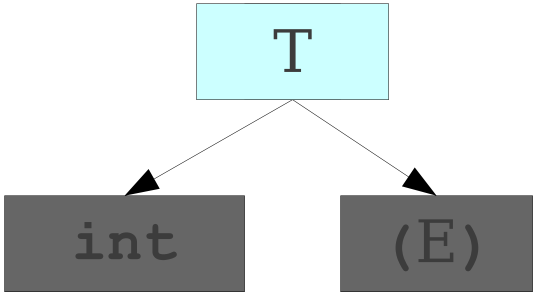


E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

`int + int`

Leftmost BFS





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

`int + int`

Leftmost BFS

Worklist → T + E

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

`int + int`

Leftmost BFS

Worklist

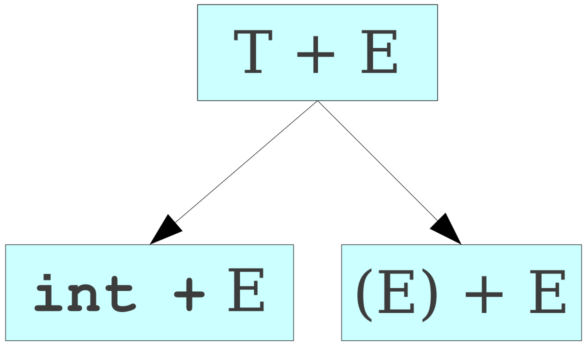
T + E

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

`int + int`

Leftmost BFS

Worklist

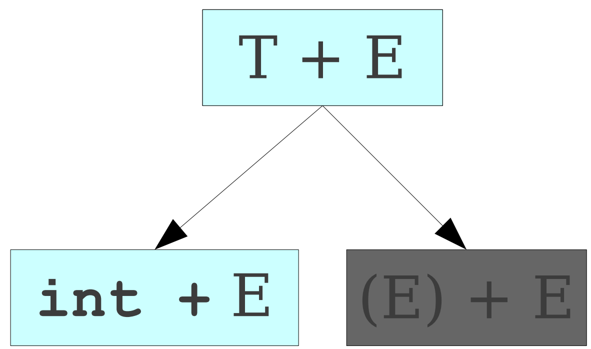


$E \rightarrow T$
$E \rightarrow T$ **+** E
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**

int + int

Leftmost BFS

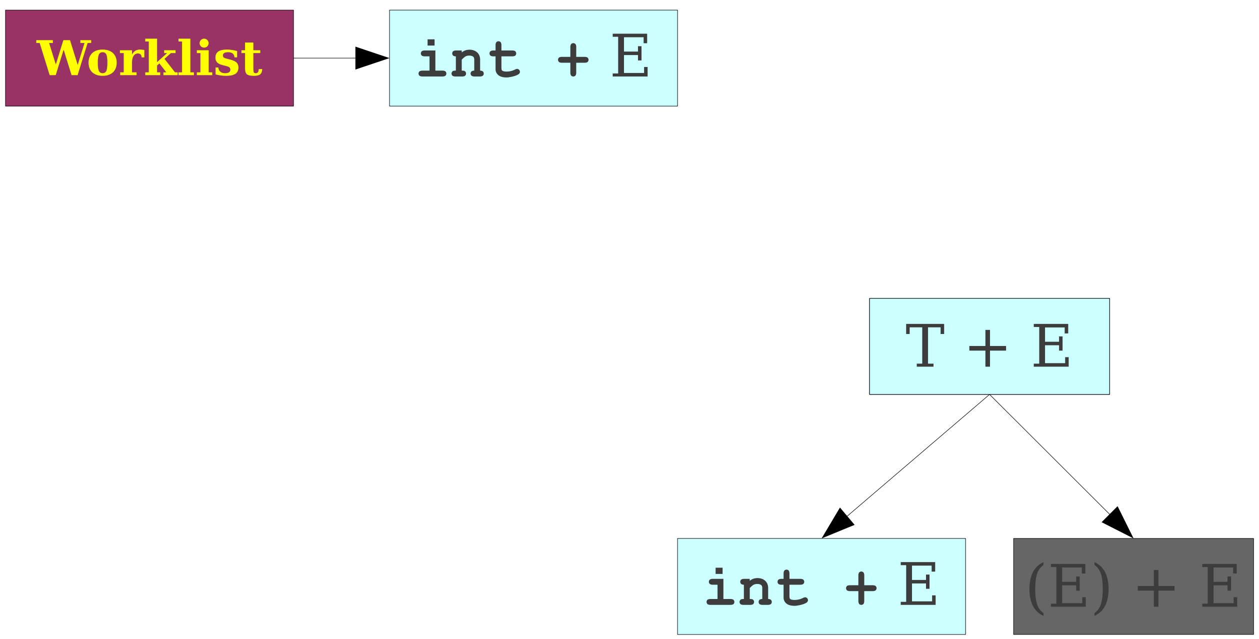
Worklist



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

Worklist → `int + E`

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

`int + int`

Leftmost BFS

Worklist

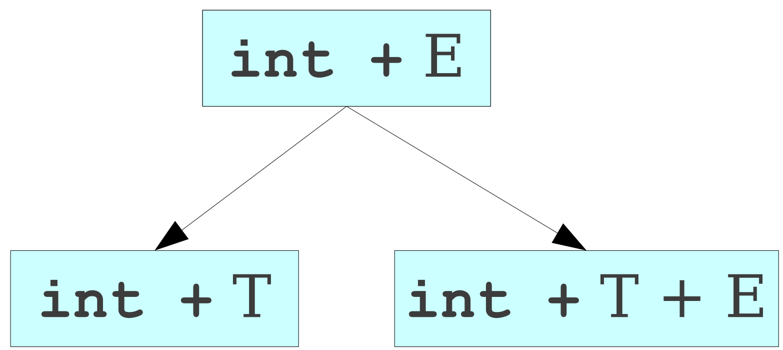
`int +` E

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

`int + int`

Leftmost BFS

Worklist



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

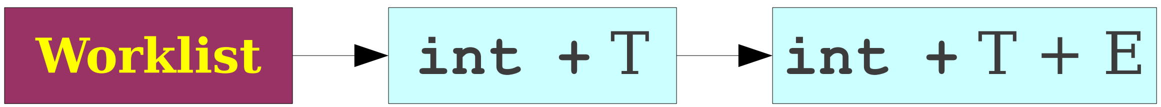
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

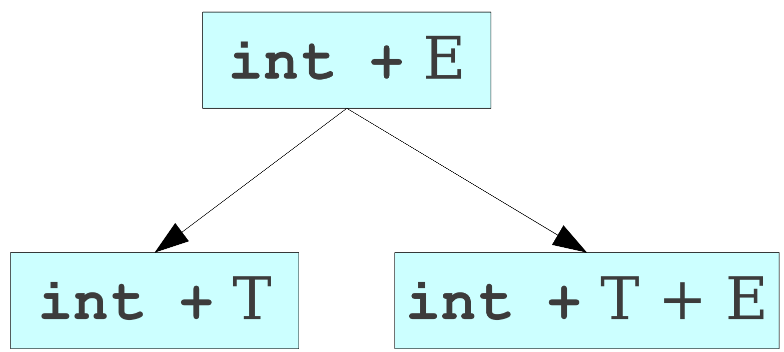
$\mathbf{T} \rightarrow$ `int`

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

`int + int`

Leftmost BFS





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

`int + int`

Leftmost BFS

Worklist → `int` + T → `int` + T + E

$E \rightarrow T$

$E \rightarrow T$ `+` E

$T \rightarrow$ `int`

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

`int + int`

Leftmost BFS

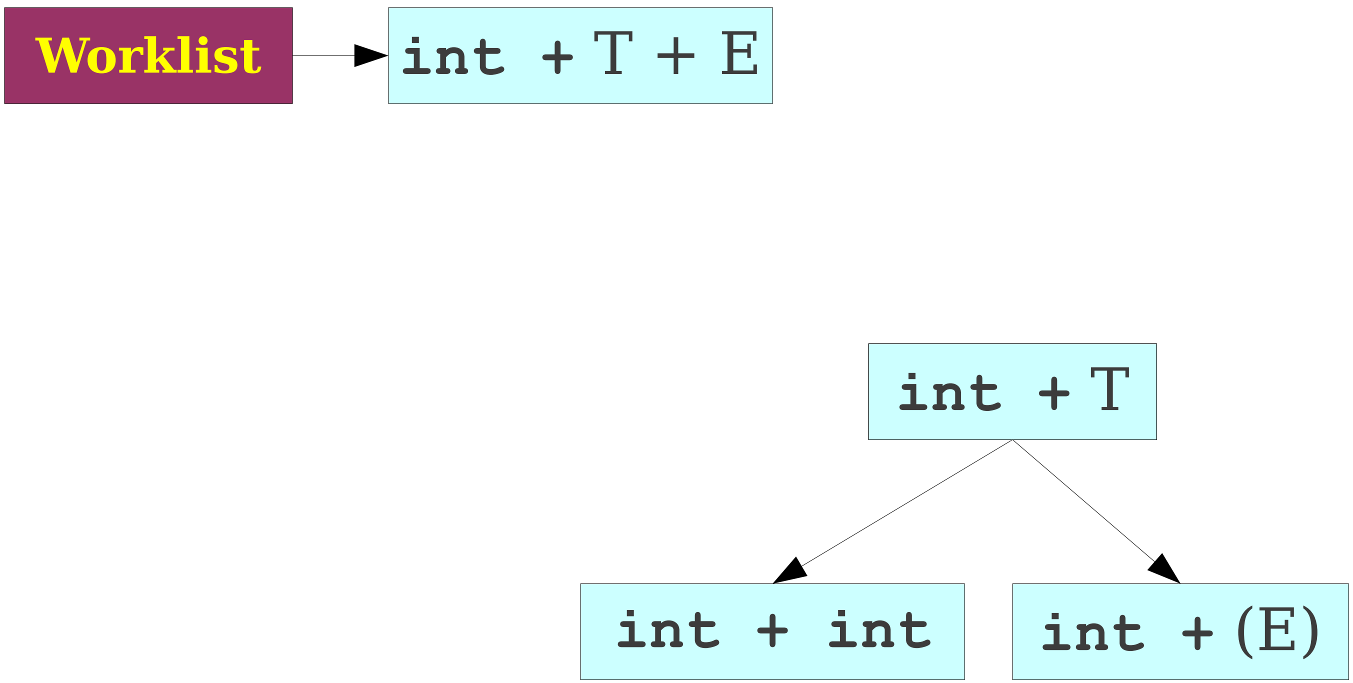
Worklist → `int` + T + E

`int` + T

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

`int + int`

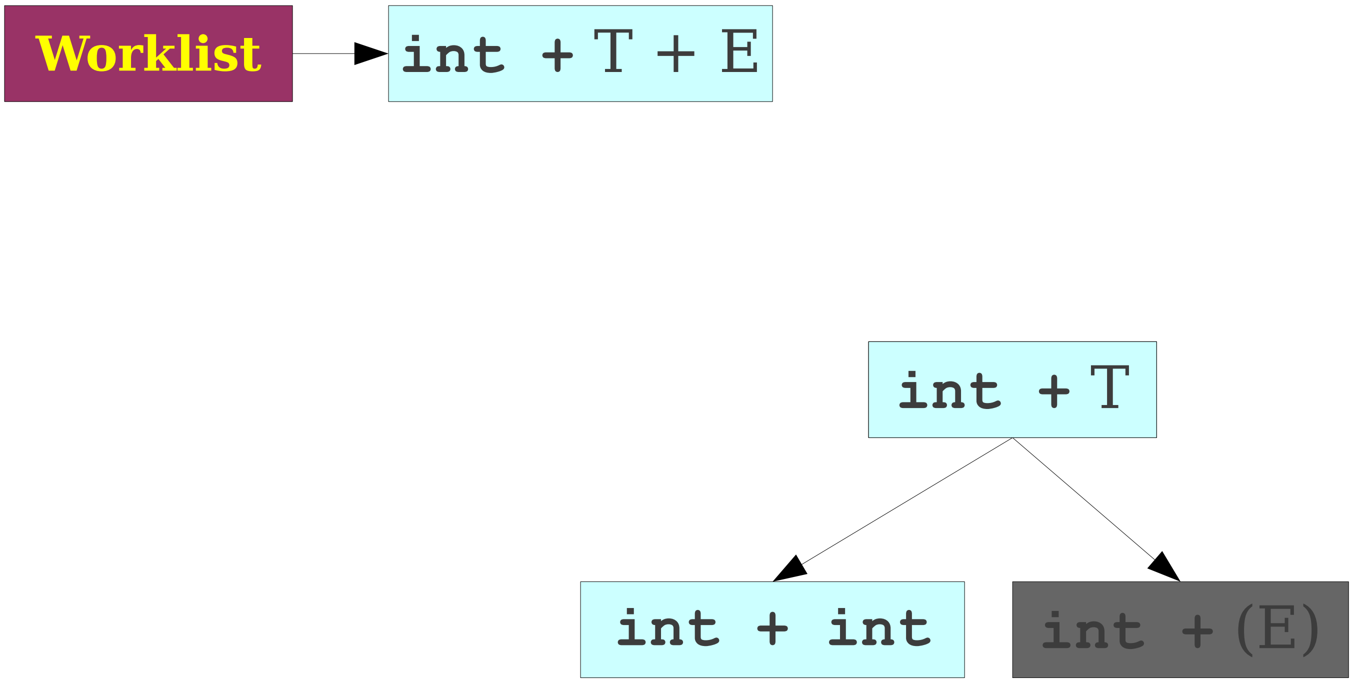
Leftmost BFS



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

int + int

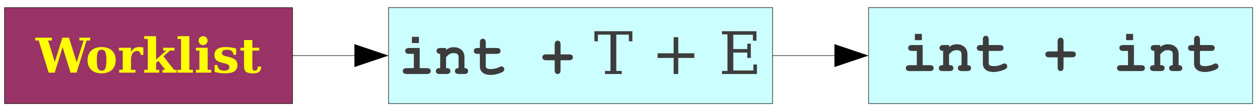
Leftmost BFS

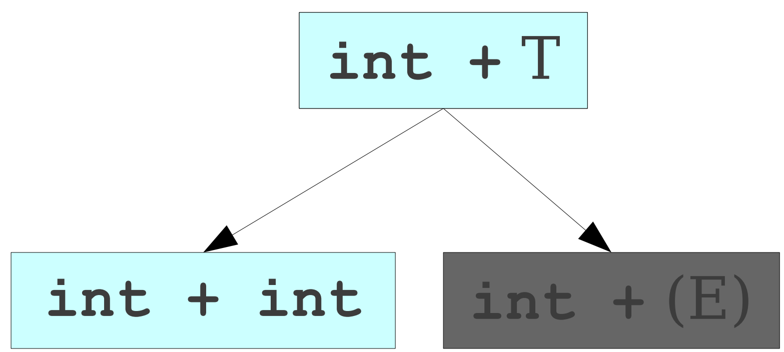


$\mathbf{E} \rightarrow \mathbf{T}$
$\mathbf{E} \rightarrow \mathbf{T}$ **+** $\mathbf{E}$
$\mathbf{T} \rightarrow$ **int**
$\mathbf{T} \rightarrow$ **(E)**

int + int

Leftmost BFS





E → **T**

E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

Leftmost BFS

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

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

`int + int`

Leftmost BFS

Worklist → `int + int`

`int +` T + E

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

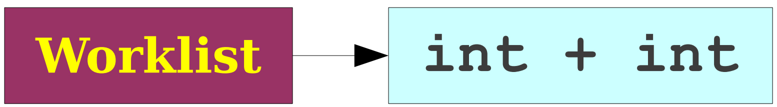
$\mathbf{E} \rightarrow \mathbf{T}$ `+` $\mathbf{E}$

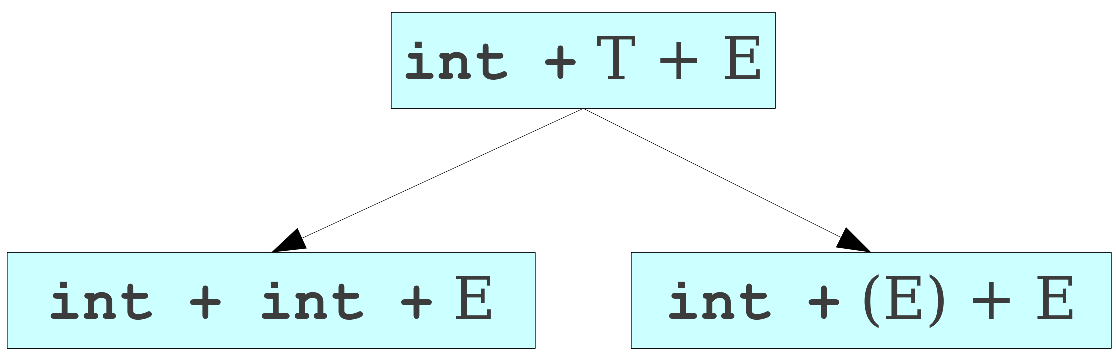
$\mathbf{T} \rightarrow$ `int`

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

`int + int`

Leftmost BFS





E → **T**

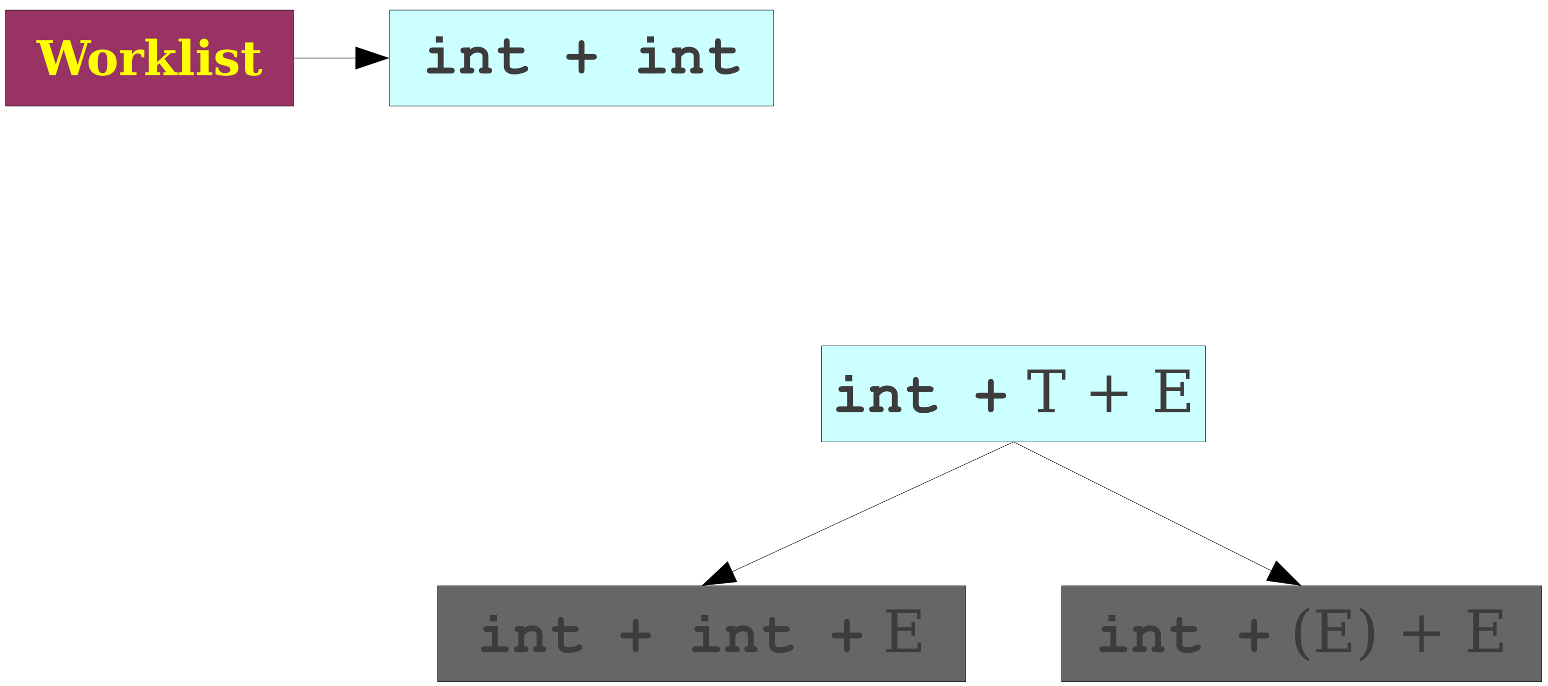
E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

Leftmost BFS



$E \rightarrow T$

$E \rightarrow T$ `+` E

$T \rightarrow$ `int`

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

`int + int`

Leftmost BFS

Worklist → `int + int`

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

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

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

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

`int + int`

Leftmost BFS

Worklist

`int + int`

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

`int + int`

Leftmost BFS

Worklist

`int + int`

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

`int + int`

Leftmost BFS

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

Leftmost BFS Has Problems

Worklist

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

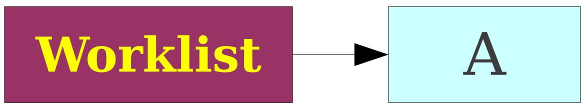
Leftmost BFS Has Problems

Worklist

A → **A**`a` | **A**`b` | `c`

caaaaaaaaaa

Leftmost BFS Has Problems



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

`caaaaaaaaaa`

Leftmost BFS Has Problems

Worklist

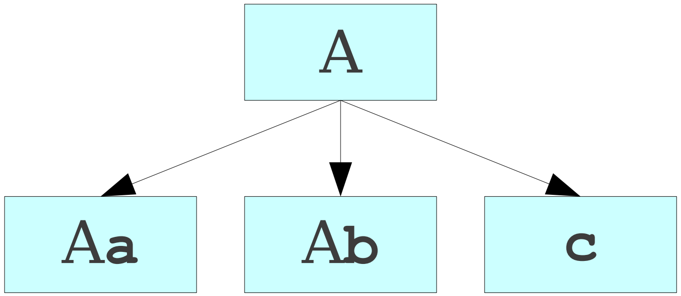
A

A → **A**`a` | **A**`b` | `c`

`caaaaaaaaa`

Leftmost BFS Has Problems

Worklist

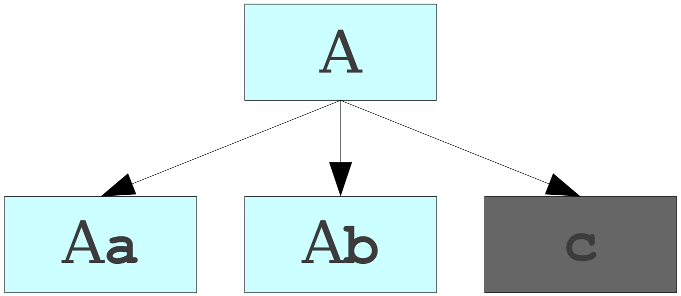


$\mathbf{A} \rightarrow \mathbf{A}\mathtt{a} \mid \mathbf{A}\mathtt{b} \mid \mathtt{c}$

`caaaaaaaaaa`

Leftmost BFS Has Problems

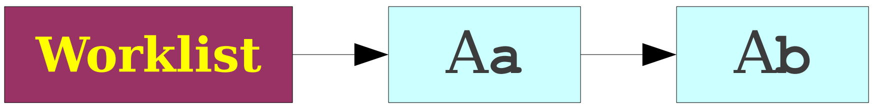
Worklist

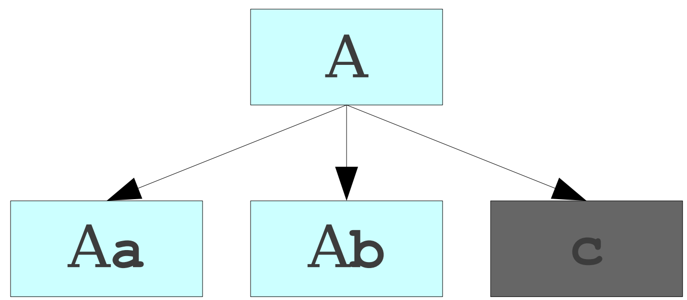


$\mathbf{A} \rightarrow \mathbf{A}\mathtt{a} \mid \mathbf{A}\mathtt{b} \mid \mathtt{c}$

`caaaaaaaaaa`

Leftmost BFS Has Problems

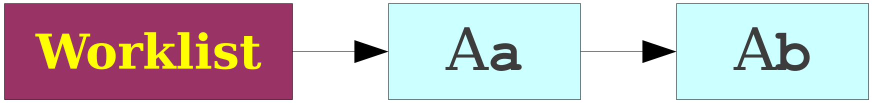




$A \rightarrow Aa \mid Ab \mid c$

caaaaaaaaaa

Leftmost BFS Has Problems



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

caaaaaaaaaa

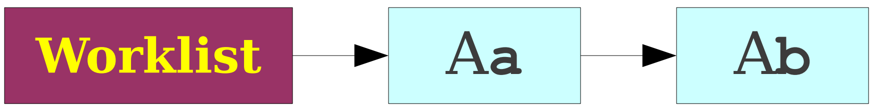
Leftmost BFS Has Problems

Worklist → A`a` → A`b`

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

`caaaaaaaaaa`

Leftmost BFS Has Problems



A → **Aa** | **Ab** | **c**

caaaaaaaaaa

Leftmost BFS Has Problems

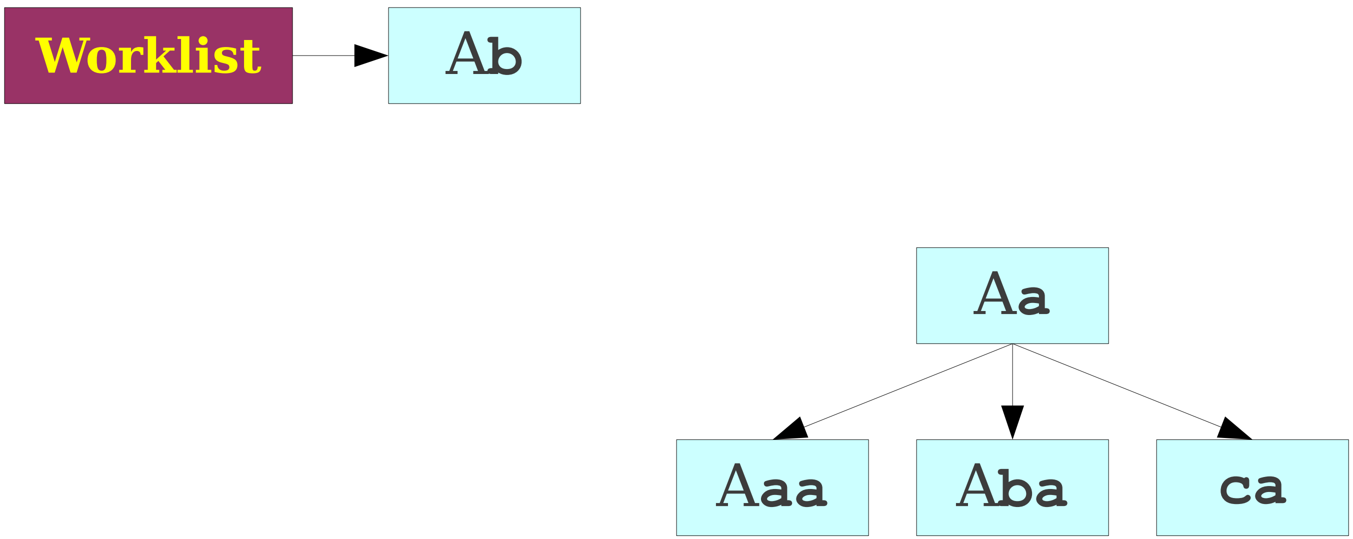
Worklist → Ab

Aa

A → **A**a | **A**b | c

caaaaaaaaa

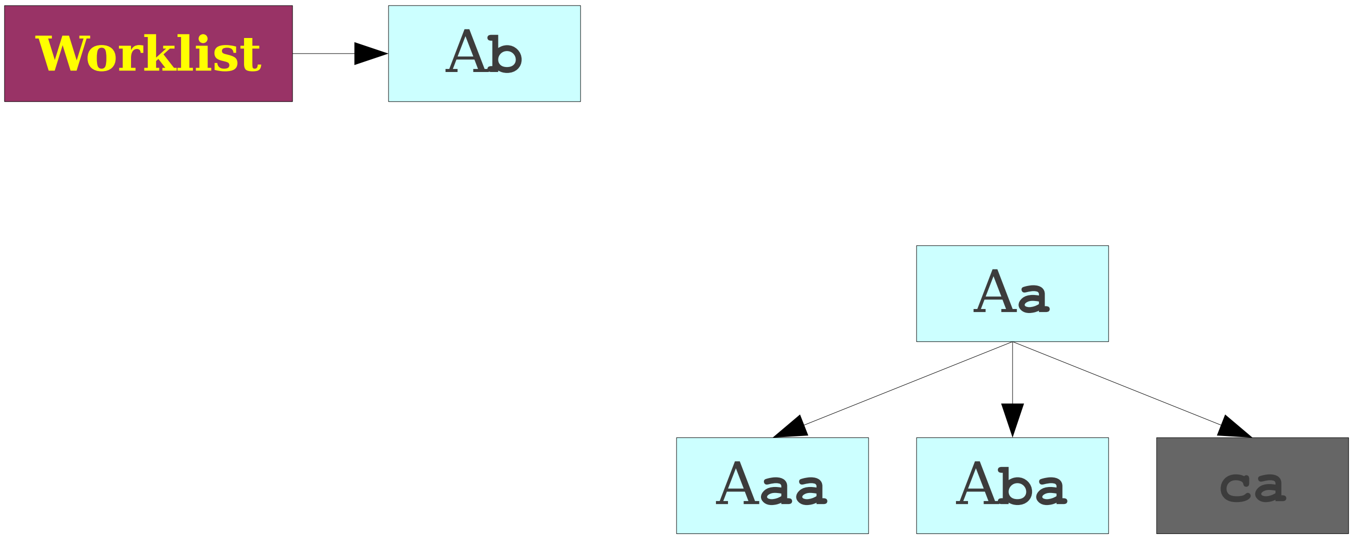
Leftmost BFS Has Problems



A → **Aa** | **Ab** | **c**

caaaaaaaaaa

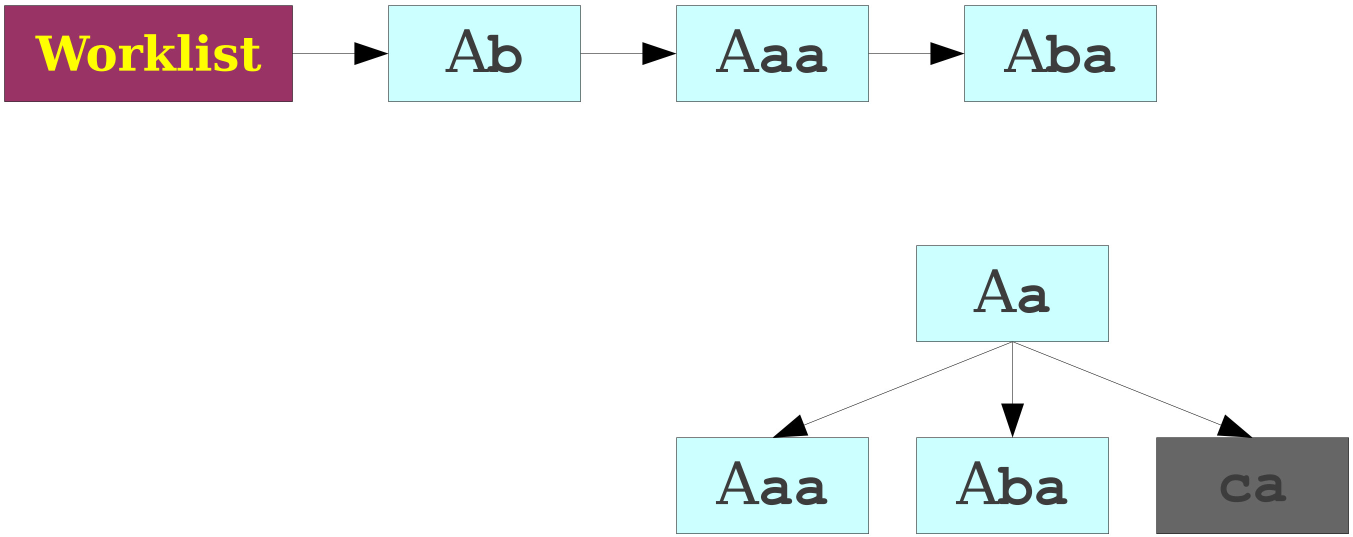
Leftmost BFS Has Problems



A → **A**`a` | **A**`b` | `c`

`caaaaaaaaaa`

Leftmost BFS Has Problems



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

caaaaaaaaa

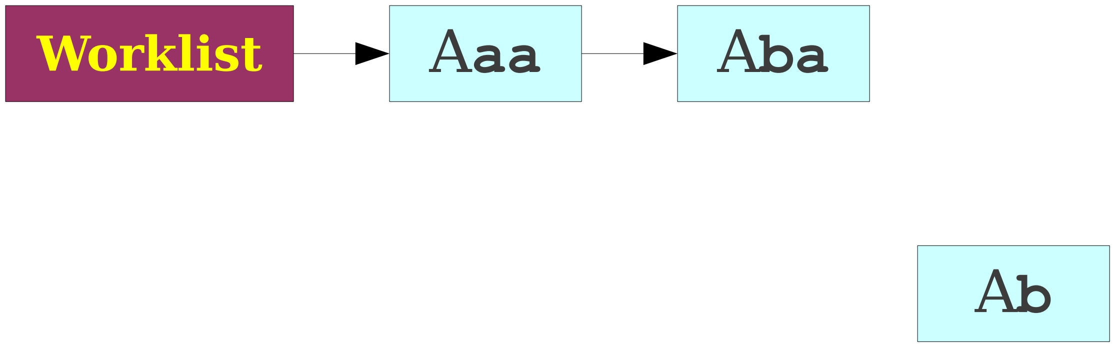
Leftmost BFS Has Problems

Worklist → Ab → Aaa → Aba

A → **A**`a` | **A**`b` | `c`

`caaaaaaaaaa`

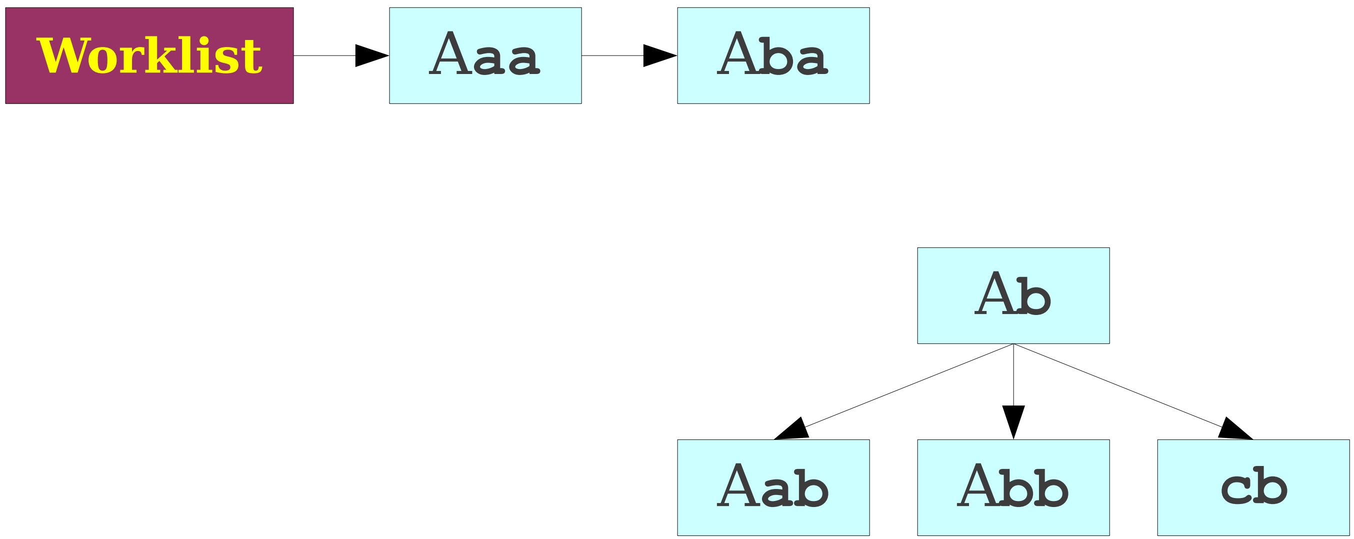
Leftmost BFS Has Problems



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

caaaaaaaaa

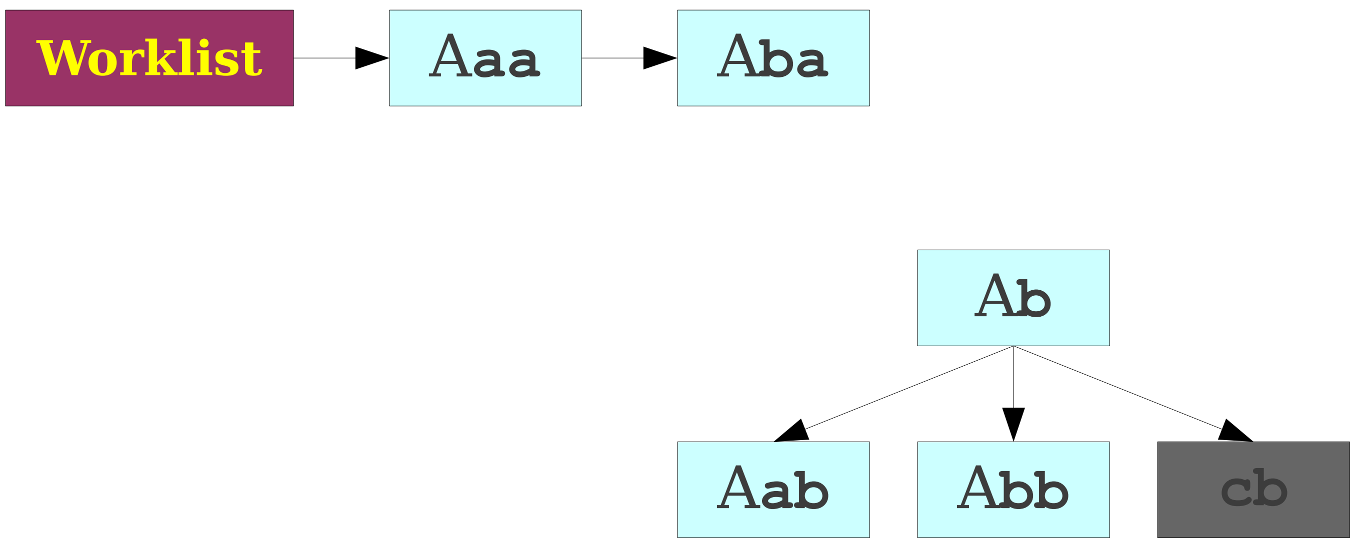
Leftmost BFS Has Problems



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

caaaaaaaaa

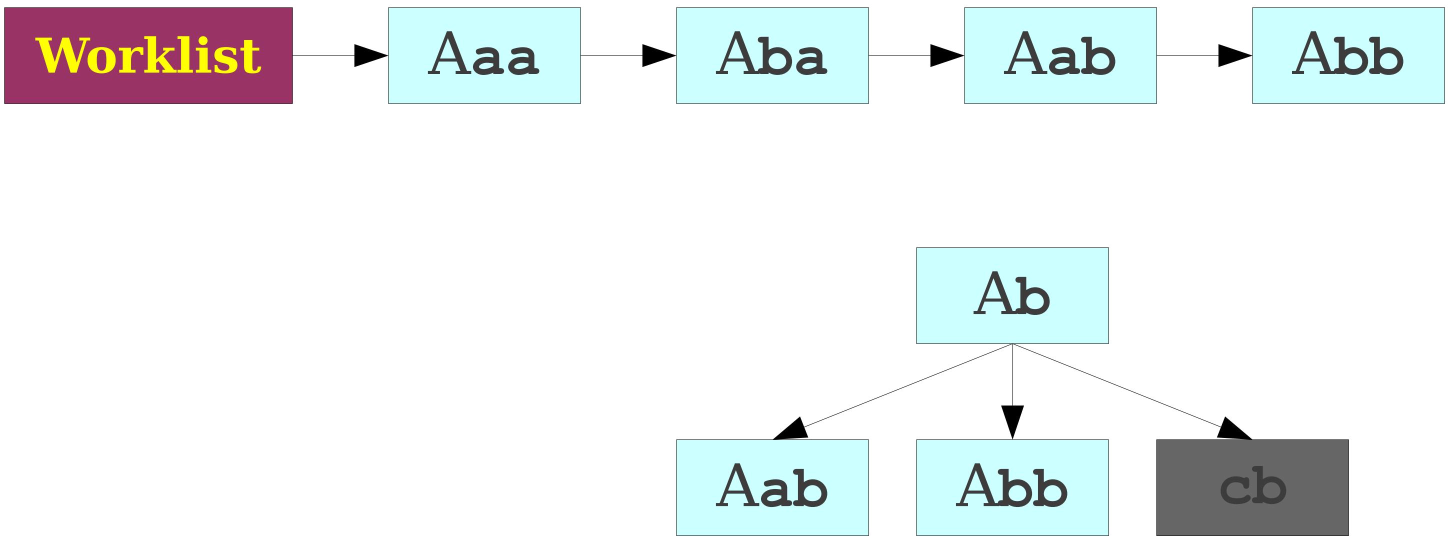
Leftmost BFS Has Problems



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

caaaaaaaaa

Leftmost BFS Has Problems



A → **A**a | **A**b | c

caaaaaaaaa

Leftmost BFS Has Problems

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

A → **A**`a` | **A**`b` | `c`

`caaaaaaaaa`

Leftmost BFS Has Problems

Worklist → Aaa → Aba → Aab → Abb

A → **A**a | **A**b | c

caaaaaaaaaa

Problems with Leftmost BFS

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

Leftmost DFS

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

Leftmost DFS

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

Leftmost DFS

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

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

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

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

`int + int`

Leftmost DFS

E

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

int + int

Leftmost DFS

E
T

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

int + int

Leftmost DFS

E
T
`int`

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

`int + int`

Leftmost DFS

E
T
`int`

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

`int + int`

Leftmost DFS

E
T

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

`int + int`

Leftmost DFS

E
T
(E)

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

`int + int`

Leftmost DFS

E
T
(E)

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

`int + int`

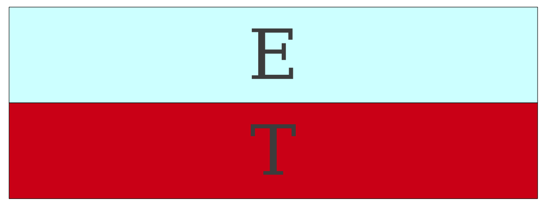
Leftmost DFS

E
T

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

`int + int`

Leftmost DFS



$E \rightarrow T$

$E \rightarrow T$ `+` E

$T \rightarrow$ `int`

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

`int + int`

Leftmost DFS

E

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

`int + int`

Leftmost DFS

E
T **+** E

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

int + int

Leftmost DFS

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

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

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

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

`int + int`

Leftmost DFS

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

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

int + int

Problems with Leftmost DFS

$A \rightarrow Aa \mid c$

A
Aa
Aaa
Aaaa
Aaaaa

c

Left Recursion

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

$$\mathbf{A} \Rightarrow^* \mathbf{A}\boldsymbol{\omega}$$

  for some string $\boldsymbol{\omega}$.
- Leftmost DFS may fail on left-recursive grammars.
- Fortunately, in many cases it is possible to eliminate left recursion (see Handout 08 for details).

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

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

int	+	(	int	+	int	)

Predictive Parsing

E

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

int	+	(	int	+	int	)

Predictive Parsing

E

$E \to T$

$E \to T$ + E

$T \to$ int

$T \to$ (E)

int	+	(	int	+	int	)

Predictive Parsing

E
T + E

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

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

Predictive Parsing

E
T + E

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

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

Predictive Parsing

E
T + E

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

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

Predictive Parsing

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

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

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

Predictive Parsing

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

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

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

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

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

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

Predictive Parsing

E
T + E
int + E

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing

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

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

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

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

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

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

Predictive Parsing

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

$E \rightarrow T$

$E \rightarrow T$ `+` E

$T \rightarrow$ `int`

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

int	+	(	int	+	int	)

Predictive Parsing

E
T + E
int + E
int + T

$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow$ **int**

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

int	+	(	int	+	int	)

Predictive Parsing

E
T + E
int + E
int + T

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

int	+	(	int	+	int	)

Predictive Parsing

E
T + E
int + E
int + T
int + (E)

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing

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

E → **T**

E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

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

Predictive Parsing

E
T + E
int + E
int + T
int + (E)

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing

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

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

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

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

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

int	+	(	int	+	int	)

Predictive Parsing

E
T + E
int + E
int + T
int + (E)
int + (T + E)

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing

E
T + E
int + E
int + T
int + (E)
int + (T + E)

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

int	+	(	int	+	int	)

Predictive Parsing

E
T + E
int + E
int + T
int + (E)
int + (T + E)

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing

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

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)

int	+	(	int	+	int	)

Predictive Parsing

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

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)

int	+	(	int	+	int	)

Predictive Parsing

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)

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

int	+	(	int	+	int	)

Predictive Parsing

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

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)

int	+	(	int	+	int	)

Predictive Parsing

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

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)
int + (int + T)

int	+	(	int	+	int	)

Predictive Parsing

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

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)
int + (int + T)

int	+	(	int	+	int	)

Predictive Parsing

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

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)
int + (int + T)

int	+	(	int	+	int	)

Predictive Parsing

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

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)
int + (int + T)
int + (int + int)

int	+	(	int	+	int	)

Predictive Parsing

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

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)
int + (int + T)
int + (int + int)

int	+	(	int	+	int	)

Predictive Parsing

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

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)
int + (int + T)
int + (int + int)

int	+	(	int	+	int	)

Predictive Parsing

E → T
E → T + E
T → int
T → (E)

E
T + E
int + E
int + T
int + (E)
int + (T + E)
int + (int + E)
int + (int + T)
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.**

LL(1) Parse Tables

LL(1) Parse Tables

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

LL(1) Parse Tables

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

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

LL(1) Parsing

E	(int + (int * int))

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

This is called a **predict** step.

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

This is called a match step.

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

(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

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))$

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

	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 → `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))$

	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 → *

	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) $E \to$ `int`
(2) $E \to$ `(`E Op E`)`
(3) $Op \to$ `+`
(4) $Op \to$ `*`

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

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

E$	int + int$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

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

E$	int + int$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

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

E$	int + int$
int $	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$

	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$

	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$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

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

`(int (int))$`

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

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

E$		(int (int))$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

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

E$	`(int (int))$`

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

LL(1) Error Detection, Part II

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

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

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

LL(1) Error Detection, Part II

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

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

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 → `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** → **`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

The LL(1) Algorithm

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

A Simple LL(1) Grammar

$$
\begin{array}{lcl}
\textbf{STMT} & \to & \texttt{if}\ \textbf{EXPR}\ \texttt{then}\ \textbf{STMT} \\
 & | & \texttt{while}\ \textbf{EXPR}\ \texttt{do}\ \textbf{STMT} \\
 & | & \textbf{EXPR}\ \texttt{;} \\
\textbf{EXPR} & \to & \textbf{TERM}\ \texttt{->}\ \texttt{id} \\
 & | & \texttt{zero?}\ \textbf{TERM} \\
 & | & \texttt{not}\ \textbf{EXPR} \\
 & | & \texttt{++}\ \texttt{id} \\
 & | & \texttt{--}\ \texttt{id} \\
\textbf{TERM} & \to & \texttt{id} \\
 & | & \texttt{constant}
\end{array}
$$

A Simple LL(1) Grammar

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

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

TERM → `id`
  | `constant`

```
id -> id;
```

```
while not zero? id
    do --id;
```

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

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

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

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** → ***ω*** iff ***ω*** derives something starting with `t`.
- More rigorously:

$$T[\mathbf{A}, \mathtt{t}] = \mathbf{B}\boldsymbol{\omega} \text{ iff } \mathbf{A} \to \boldsymbol{\omega} \text{ and } \boldsymbol{\omega} \Rightarrow^* \mathtt{t}\boldsymbol{\omega}'$$

In what follows, assume that our grammar does not contain any ε-productions.

(We'll relax this restiction later.)

FIRST Sets

- We want to tell if a particular nonterminal **A** derives a string starting with a particular nonterminal **`t`**.
- We can formalize this with **FIRST sets**.

$$\text{FIRST}(\mathbf{A}) = \{ \; \mathtt{t} \mid \mathbf{A} \Rightarrow^* \mathtt{t}\boldsymbol{\omega} \text{ for some } \boldsymbol{\omega} \; \}$$

- Intuitively, FIRST(**A**) is the set of terminals that can be at the start of a string produced by **A**.
- If we can compute FIRST sets for all nonterminals in a grammar, we can efficiently construct the LL(1) parsing table. Details soon.

Computing FIRST Sets

- Initially, for all nonterminals **A**, set

  FIRST(**A**) = { `t` | **A** → `t`$\boldsymbol{\omega}$ for some $\boldsymbol{\omega}$ }

- Then, repeat the following until no changes occur: For each nonterminal **A**, for each production **A** → **B**$\boldsymbol{\omega}$, set

  FIRST(**A**) = FIRST(**A**) ∪ FIRST(**B**)

- This is known a **fixed-point iteration** or a **transitive closure algorithm**.

Iterative FIRST Computations

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

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

TERM → id
     | constant
```

Iterative FIRST Computations

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

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

TERM → `id`
| `constant`

STMT	EXPR	TERM

Iterative FIRST Computations

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

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

TERM → id
     | constant
```

STMT	EXPR	TERM
if while		

Iterative FIRST Computations

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

STMT	EXPR	TERM
if while	zero? not ++ --	

Iterative FIRST Computations

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

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

TERM → id
     |  constant
```

STMT	EXPR	TERM
if while	zero? not ++ --	id constant

Iterative FIRST Computations

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

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

TERM → `id`
| `constant`

STMT	EXPR	TERM

`if`
`while`
	`zero?`
	`not`
	`++`
	`--`
		`id`
		`constant`

Iterative FIRST Computations

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

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

TERM → `id`
| `constant`

STMT	EXPR	TERM
`if`	`zero?`	`id`
`while`	`not`	`constant`
`zero?`	`++`	
`not`	`--`	
`++`		
`--`		

Iterative FIRST Computations

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

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

TERM → `id`
  | `constant`

STMT	EXPR	TERM
if while zero? not ++ --	zero? not ++ --	id constant

Iterative FIRST Computations

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

STMT	EXPR	TERM
if	zero?	id
while	not	constant
zero?	++	
not	--	
++		
--		

Iterative FIRST Computations

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

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

TERM → id
| constant

STMT	EXPR	TERM
if while zero? not ++ --	zero? not ++ -- id constant	id constant

Iterative FIRST Computations

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

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

TERM → id
     |  constant
```

STMT	EXPR	TERM
if	zero?	id
while	not	constant
zero?	++	
not	--	
++	id	
--	constant	

Iterative FIRST Computations

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

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

TERM → id
  | constant

STMT	EXPR	TERM
if while zero? not ++ --	zero? not ++ -- id constant	id constant

Iterative FIRST Computations

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

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

TERM → id
| constant

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

Iterative FIRST Computations

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

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

TERM → `id`
| `constant`

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

From FIRST Sets to LL(1) Tables

From FIRST Sets to LL(1) 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)**

From FIRST Sets to LL(1) Tables

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

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

From FIRST Sets to LL(1) 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)

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

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

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

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

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM

`if`
`while`
`zero?`
`not`
`++`
`--`
`id`
`constant`
	`zero?`
	`not`
	`++`
	`--`
	`id`
	`constant`
		`id`
		`constant`

	`if`	`then`	`while`	`do`	`zero?`	`not`	`++`	`--`	→	`id`	`const`	`;`
STMT	1											
EXPR												
TERM												

From FIRST Sets to LL(1) 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)

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1											
EXPR												
TERM												

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2									
EXPR												
TERM												

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2									
EXPR												
TERM												

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2									
EXPR												
TERM												

From FIRST Sets to LL(1) 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)

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2		3	3	3	3		3	3	
EXPR												
TERM												

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if	zero?	id
while	not	constant
zero?	++	
not	--	
++	id	
--	constant	
id		
constant		

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2		3	3	3	3		3	3	
EXPR												
TERM												

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2		3	3	3	3		3	3	
EXPR												
TERM												

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2		3	3	3	3		3	3	
EXPR										4	4	
TERM												

From FIRST Sets to LL(1) 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)
```

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

	if	then	while	do	zero?	not	++	--	→	id	const	;
STMT	1		2		3	3	3	3		3	3	
EXPR										4	4	
TERM												

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

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

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

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

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

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

From FIRST Sets to LL(1) 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)

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

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

From FIRST Sets to LL(1) 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)

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

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

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

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

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

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

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM

`if`
`while`
`zero?`
`not`
`++`
`--`
`id`
`constant`
	`zero?`
	`not`
	`++`
	`--`
	`id`
	`constant`
		`id`
		`constant`

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

From FIRST Sets to LL(1) 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)

STMT	EXPR	TERM
if	zero?	id
while	not	constant
zero?	++	
not	--	
++	id	
--	constant	
id		
constant		

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

From FIRST Sets to LL(1) 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)

STMT	EXPR	TERM

`if`
`while`
`zero?`
`not`
`++`
`--`
`id`
`constant`
	`zero?`
	`not`
	`++`
	`--`
	`id`
	`constant`
		`id`
		`constant`

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

From FIRST Sets to LL(1) 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)**

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

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

From FIRST Sets to LL(1) 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)
```

STMT	EXPR	TERM
if while zero? not ++ -- id constant	zero? not ++ -- id constant	id constant

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

From FIRST Sets to LL(1) Tables

STMT	→	if EXPR then STMT	(1)
	|	while EXPR do STMT	(2)
	|	EXPR ;	(3)
EXPR	→	TERM -> id	
	|	zero? TERM	
	|	not EXPR	
	|	++ id	
	|	-- id	
TERM	→	id	
	|	constant	

STMT	EXPR	TERM
if	zero? not ++ -- id onstant	id constant

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

ε-Free LL(1) Parse Tables

- The following algorithm constructs an LL(1) parse table for a grammar with no ε-productions.
- Compute the FIRST sets for all nonterminals in the grammar.
- For each production $\mathbf{A} \to \mathtt{t}\boldsymbol{\omega}$, set $T[\mathbf{A}, \mathtt{t}] = \mathtt{t}\boldsymbol{\omega}$.
- For each production $\mathbf{A} \to \mathbf{B}\boldsymbol{\omega}$, set $T[\mathbf{A}, \mathtt{t}] = \mathbf{B}\boldsymbol{\omega}$ for each $\mathtt{t} \in \mathrm{FIRST}(\mathbf{B})$.

Expanding our Grammar

```
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)
```

```
id → id;

while not zero? id do --id;

if not zero? id then
    if not zero? id then
        constant → id;
```

Expanding our Grammar

```
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)

BLOCK→   STMT                     (11)
      |  { STMTS }                (12)

STMTS→   STMT STMTS               (13)
      |  ε                        (14)
```

```
id → id;

while not zero? id do --id;

if not zero? id then
   if not zero? id then
       constant → id;
```

Expanding our Grammar

STMT	→	`if` **EXPR** `then` **BLOCK**	**(1)**
	|	`while` **EXPR** `do` **BLOCK**	**(2)**
	|	**EXPR** `;`	**(3)**
EXPR	→	**TERM** `-> id`	**(4)**
	|	`zero?` **TERM**	**(5)**
	|	`not` **EXPR**	**(6)**
	|	`++ id`	**(7)**
	|	`-- id`	**(8)**
TERM	→	`id`	**(9)**
	|	`constant`	**(10)**
BLOCK	→	**STMT**	**(11)**
	|	`{` **STMTS** `}`	**(12)**
STMTS	→	**STMT STMTS**	**(13)**
	|	ε	**(14)**

```
id → id;

while not zero? id do --id;

if not zero? id then
   if not zero? id then
      constant → id;
```

Expanding our Grammar

STMT	→	`if` **EXPR** `then` **BLOCK**	**(1)**
	|	`while` **EXPR** `do` **BLOCK**	**(2)**
	|	**EXPR** `;`	**(3)**
EXPR	→	**TERM** `-> id`	**(4)**
	|	`zero?` **TERM**	**(5)**
	|	`not` **EXPR**	**(6)**
	|	`++ id`	**(7)**
	|	`-- id`	**(8)**
TERM	→	`id`	**(9)**
	|	`constant`	**(10)**
BLOCK	→	**STMT**	**(11)**
	|	`{` **STMTS** `}`	**(12)**
STMTS	→	**STMT STMTS**	**(13)**
	|	ε	**(14)**

```
id → id;
```

```
while not zero? id do --id;
```

```
if not zero? id then
   if not zero? id then
       constant → id;
```

```
if zero? id then
    while zero? id do {
        constant → id;
        constant → id;
    }
```

LL(1) with ε-Productions

- Computation of FIRST is different.
  - What if the first nonterminal in a production can produce ε?
- Building the table is different.
  - What action do you take if the correct production produces the empty string?

FIRST Sets with ε

FIRST Sets with ε

Num	→ **Sign Digits**
Sign	→ `+` | `-` | **ε**
Digits	→ **Digit More**
More	→ **Digits** | **ε**
Digit	→ `0` | `1` | `2` | ... | `9`

FIRST Sets with ε

Num → **Sign Digits**
Sign → **+** | **-** | **ε**
Digits → **Digit More**
More → **Digits** | **ε**
Digit → **0** | **1** | **2** | ... | **9**

Num	Sign	Digit	Digits	More

FIRST Sets with ε

Num	→	**Sign Digits**
Sign	→	`+` | `-` | ε
Digits	→	**Digit More**
More	→	**Digits** | ε
Digit	→	`0` | `1` | `2` | ... | `9`

Num	Sign	Digit	Digits	More
	+ -	0 5 1 6 2 7 3 8 4 9		

FIRST Sets with ε

Num → Sign Digits
Sign → + | - | ε
Digits → Digit More
More → Digits | ε
Digit → 0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More
	+ -	0 5 1 6 2 7 3 8 4 9		

FIRST Sets with ε

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More
+ -	+ -	0 5 1 6 2 7 3 8 4 9		

FIRST Sets with ε

Num	→	**Sign Digits**
Sign	→	+ | - | ε
Digits	→	**Digit More**
More	→	**Digits** | ε
Digit	→	0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More
+ -	+ -	0 5 1 6 2 7 3 8 4 9		

FIRST Sets with ε

Num	→	**Sign Digits**
Sign	→	**+ | – | ε**
Digits	→	**Digit More**
More	→	**Digits | ε**
Digit	→	**0 | 1 | 2 | ... | 9**

Num	Sign	Digit	Digits	More
+ –	+ –	0 5 1 6 2 7 3 8 4 9		

FIRST Sets with ε

Num	→	**Sign Digits**
Sign	→	+ | - | ε
Digits	→	**Digit More**
More	→	**Digits** | ε
Digit	→	0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More
+ -	+ -	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	

FIRST Sets with ε

Num → **Sign Digits**
Sign → **+** | **–** | **ε**
Digits → **Digit More**
More → **Digits** | **ε**
Digit → **0** | **1** | **2** | … | **9**

Num	Sign	Digit	Digits	More
+ –	+ –	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	

FIRST Sets with ε

Num	→	**Sign Digits**
Sign	→	+ | - | ε
Digits	→	**Digit More**
More	→	**Digits** | ε
Digit	→	0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More
+ -	+ -	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	

FIRST Sets with ε

Num	→	**Sign Digits**
Sign	→	+ | – | ε
Digits	→	**Digit More**
More	→	**Digits** | ε
Digit	→	0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More
+ –	+ –	0 5	0 5	0 5
		1 6	1 6	1 6
		2 7	2 7	2 7
		3 8	3 8	3 8
		4 9	4 9	4 9

FIRST Sets with ε

Num → **Sign Digits**
Sign → **+** | **-** | **ε**
Digits → **Digit More**
More → **Digits** | **ε**
Digit → **0** | **1** | **2** | ... | **9**

Num	Sign	Digit	Digits	More
+ -	+ -	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9

FIRST Sets with ε

Num → **Sign Digits**
Sign → **+** | **-** | **ε**
Digits → **Digit More**
More → **Digits** | **ε**
Digit → **0** | **1** | **2** | ... | **9**

Num	Sign	Digit	Digits	More
+ -	+ - ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

FIRST Sets with ε

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More
+ -	+ - ε	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9	0 1 2 3 4 5 6 7 8 9 ε

FIRST Sets with ε

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | 2 | ... | 9

Num	Sign	Digit	Digits	More
+ - 0 5 1 6 2 7 3 8 4 9	+ - ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

FIRST Sets with ε

Num → **Sign Digits**
Sign → `+` | `-` | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → `0` | `1` | `2` | … | `9`

Num	Sign	Digit	Digits	More
+ − 0 5 1 6 2 7 3 8 4 9	+ − ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

FIRST and ε

- When computing FIRST sets in a grammar with ε-productions, we often have to "look through" nonterminals.
- Rationale: Might have a derivation like this:

$$\mathbf{A} \Rightarrow \mathbf{B}\texttt{t} \Rightarrow \texttt{t}$$

- So $\texttt{t} \in$ FIRST($\mathbf{A}$).

FIRST Computation with ε

- Initially, for all nonterminals **A**, set

$$\text{FIRST}(\mathbf{A}) = \{ \mathtt{t} \mid \mathbf{A} \rightarrow \mathtt{t}\boldsymbol{\omega} \text{ for some } \boldsymbol{\omega} \}$$

- For all nonterminals **A** where **A** → **ε** is a production, add **ε** to FIRST(**A**).
- Repeat the following until no changes occur:
  - For each production **A** → ***α***, where ***α*** is a string of nonterminals whose FIRST sets contain **ε**, set FIRST(**A**) = FIRST(**A**) ∪ { **ε** }.
  - For each production **A** → ***α*****t*****ω***, where ***α*** is a string of nonterminals whose FIRST sets contain **ε**, set FIRST(**A**) = FIRST(**A**) ∪ { **t** }
  - For each production **A** → ***α*****B*****ω***, where ***α*** is string of nonterminals whose FIRST sets contain **ε**, set FIRST(**A**) = FIRST(**A**) ∪ (FIRST(**B**) - { **ε** }).

A Notational Diversion

- Once we have computed the correct FIRST sets for each nonterminal, we can generalize our definition of FIRST sets to strings.
- Define FIRST*($\boldsymbol{\omega}$) as follows:
  - FIRST*($\boldsymbol{\varepsilon}$) = { $\boldsymbol{\varepsilon}$ }
  - FIRST*($\mathbf{t}\boldsymbol{\omega}$) = { $\mathbf{t}$ }
  - If $\boldsymbol{\varepsilon} \notin$ FIRST($\mathbf{A}$):
    - FIRST*($\mathbf{A}\boldsymbol{\omega}$) = FIRST($\mathbf{A}$)
  - If $\boldsymbol{\varepsilon} \in$ FIRST($\mathbf{A}$):
    - FIRST*($\mathbf{A}\boldsymbol{\omega}$) = (FIRST($\mathbf{A}$) - { $\boldsymbol{\varepsilon}$ }) $\cup$ FIRST*($\boldsymbol{\omega}$)

FIRST Computation with ε

- Initially, for all nonterminals **A**, set

  FIRST(**A**) = { **`t`** | **A** → **`t`*ω*** for some ***ω*** }
- For all nonterminals **A** where **A** → **ε** is a production, add **ε** to FIRST(**A**).
- Repeat the following until no changes occur:
  - For each production **A** → ***α***, set FIRST(**A**) = FIRST(**A**) ∪ FIRST*(***α***)

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

	`hello`	`heya`	`yo`	`world!`
Msg				
Hi				
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → **`hello`** | **`heya`** | **`yo`**
End → **`world!`** | **ε**

Msg	Hi	End

	hello	heya	yo	world!
Msg				
Hi				
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

Msg	Hi	End

	`hello`
	`heya`
	`yo`	

	`hello`	`heya`	`yo`	`world!`
Msg				
Hi				
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

Msg	Hi	End

	`hello`
	`heya`
	`yo`
		`world`
		ε

	`hello`	`heya`	`yo`	`world!`
Msg				
Hi				
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

Msg	Hi	End
	hello **heya** **yo**	**world** **ε**

	hello	**heya**	**yo**	**world!**
Msg				
Hi				
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

Msg	Hi	End
hello **heya** **yo**	**hello** **heya** **yo**	**world** **ε**

	`hello`	`heya`	`yo`	`world!`
Msg				
Hi				
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

Msg	Hi	End
hello heya yo	hello heya yo	world ε

	hello	heya	yo	world!
Msg				
Hi				
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

Msg	Hi	End
hello **heya** **yo**	**hello** **heya** **yo**	**world** ε

	hello	heya	yo	world!
Msg				
Hi				
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

Msg	Hi	End

`hello`
`heya`
`yo`
	`hello`
	`heya`
	`yo`
		`world`
		ε

	`hello`	`heya`	`yo`	`world!`
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi				
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

Msg	Hi	End

`hello`
`heya`
`yo`
	`hello`
	`heya`
	`yo`
		`world`
		ε

	`hello`	`heya`	`yo`	`world!`
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi				
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

Msg	Hi	End

`hello`
`heya`
`yo`
	`hello`
	`heya`
	`yo`
		`world`
		ε

	`hello`	`heya`	`yo`	`world!`
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi	`hello`	`heya`	`yo`	
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → **hello** | **heya** | **yo**
End → **world!** | **ε**

Msg	Hi	End
hello heya yo	hello heya yo	world ε

	hello	heya	yo	world!
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi	hello	heya	yo	
End				

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

Msg	Hi	End

`hello`
`heya`
`yo`
	`hello`
	`heya`
	`yo`
		`world`
		ε

	`hello`	`heya`	`yo`	`world!`
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi	`hello`	`heya`	`yo`	
End				`world!`

LL(1) Tables with ε

Msg → **Hi End**
Hi → `hello` | `heya` | `yo`
End → `world!` | ε

Msg	Hi	End

`hello`
`heya`
`yo`
	`hello`
	`heya`
	`yo`
		`world`
		ε

	`hello`	`heya`	`yo`	`world!`
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi	`hello`	`heya`	`yo`	
End				`world!`

LL(1) Tables with ε

	hello	heya	yo	world!
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi	**hello**	**heya**	**yo**	
End				**world!**

LL(1) Tables with ε

Msg **$**	**hello $**

	hello	**heya**	**yo**	**world!**
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi	**hello**	**heya**	**yo**	
End				**world!**

LL(1) Tables with ε

Msg **$**	**hello $**

	hello	**heya**	**yo**	**world!**
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi	**hello**	**heya**	**yo**	
End				**world!**

LL(1) Tables with ε

Msg **$**	**hello $**
Hi End **$**	**hello $**

	hello	**heya**	**yo**	**world!**
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi	**hello**	**heya**	**yo**	
End				**world!**

LL(1) Tables with ε

Msg **$**	**hello $**
Hi End **$**	**hello $**

	hello	**heya**	**yo**	**world!**
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi	**hello**	**heya**	**yo**	
End				**world!**

LL(1) Tables with ε

Msg **$**	**hello $**
Hi End **$**	**hello $**
hello End **$**	**hello $**

	hello	**heya**	**yo**	**world!**
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi	**hello**	**heya**	**yo**	
End				**world!**

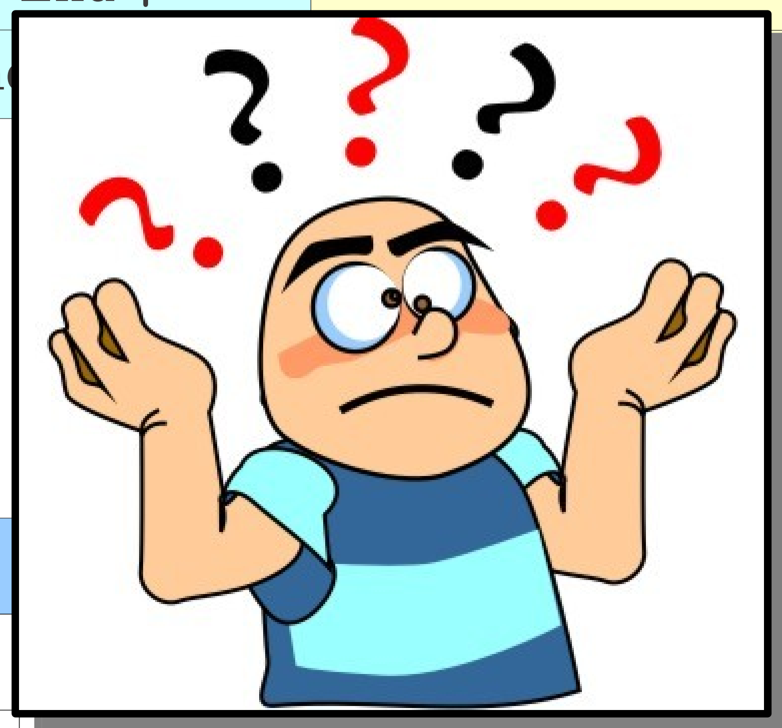
LL(1) Tables with ε

Msg **$**	**hello $**
Hi End **$**	**hello $**
hello End **$**	**hello $**
End **$**	**$**

	hello	**heya**	**yo**	**world!**
Msg	**Hi End**	**Hi End**	**Hi End**	
Hi	**hello**	**heya**	**yo**	
End				**world!**

LL(1) Tables with ε

Msg $	hello $
Hi End $	hello $
hello End $	hello $
En	$



	hello			orld!
Msg	Hi End			
Hi	hello	heya	yo	
End				world!

ε is Complicated

- When constructing LL(1) tables with ε-productions, we need to have an extra column for **$**.

Msg	→	**Hi End**
Hi	→	**hello** | **heya** | **yo**
End	→	**world!** | **ε**

ε is Complicated

- When constructing LL(1) tables with ε-productions, we need to have an extra column for **$**.

Msg → **Hi End**
Hi → **hello** | **heya** | **yo**
End → **world!** | **ε**

	hello	heya	yo	world!	$
Msg	Hi End	Hi End	Hi End		
Hi	hello	heya	yo		
End				world!	

ε is Complicated

- When constructing LL(1) tables with ε-productions, we need to have an extra column for **$**.

Msg → **Hi End**
Hi → **hello** | **heya** | **yo**
End → **world!** | **ε**

	hello	heya	yo	world!	$
Msg	Hi End	Hi End	Hi End		
Hi	hello	heya	yo		
End				world!	ε

LL(1) Tables with ε

LL(1) Tables with ε

Msg **`$`**	**`hello $`**
Hi End **`$`**	**`hello $`**
`hello` End **`$`**	**`hello $`**
End **`$`**	**`$`**

LL(1) Tables with ε

Msg **$**	**hello $**
Hi End **$**	**hello $**
hello End **$**	**hello $**
End **$**	**$**

	hello	**heya**	**yo**	**world!**	**$**
Msg	**Hi End**	**Hi End**	**Hi End**		
Hi	**hello**	**heya**	**yo**		
End				**world!**	**ε**

LL(1) Tables with ε

Msg **$**	**hello $**
Hi End **$**	**hello $**
hello End **$**	**hello $**
End **$**	**$**

	hello	**heya**	**yo**	**world!**	**$**
Msg	**Hi End**	**Hi End**	**Hi End**		
Hi	**hello**	**heya**	**yo**		
End				**world!**	**ε**

LL(1) Tables with ε

Msg **$**	**hello $**
Hi End **$**	**hello $**
hello End **$**	**hello $**
End **$**	**$**
$	**$**

	hello	**heya**	**yo**	**world!**	**$**
Msg	**Hi End**	**Hi End**	**Hi End**		
Hi	**hello**	**heya**	**yo**		
End				**world!**	**ε**

It Gets Trickier

Num	→ **Sign Digits**
Sign	→ `+` | `-` | ε
Digits	→ **Digit More**
More	→ **Digits** | ε
Digit	→ `0` | `1` | ... | `9`

It Gets Trickier

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

	+	-	#	$
Num				
Sign				
Digits				
More				
Digit				

It Gets Trickier

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ - 0 5 1 6 2 7 3 8 4 9	+ - ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num				
Sign				
Digits				
More				
Digit				

It Gets Trickier

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ - 0 5 1 6 2 7 3 8 4 9	+ - ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num				
Sign				
Digits				
More				
Digit				

It Gets Trickier

Num → **Sign Digits**
Sign → `+` | `-` | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → `0` | `1` | … | `9`

Num	Sign	Digit	Digits	More
+ – 0 5 1 6 2 7 3 8 4 9	+ – ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign				
Digits				
More				
Digit				

It Gets Trickier

Num → **Sign Digits**
Sign → **+** | **-** | **ε**
Digits → **Digit More**
More → **Digits** | **ε**
Digit → **0** | **1** | … | **9**

Num	Sign	Digit	Digits	More
+ – 0 5 1 6 2 7 3 8 4 9	+ – ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign				
Digits				
More				
Digit				

It Gets Trickier

Num → **Sign Digits**
Sign → **+** | **-** | **ε**
Digits → **Digit More**
More → **Digits** | **ε**
Digit → **0** | **1** | … | **9**

Num	Sign	Digit	Digits	More
+ - 0 5 1 6 2 7 3 8 4 9	+ - ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign	+	-		
Digits				
More				
Digit				

It Gets Trickier

Num → **Sign Digits**
Sign → **+** | **-** | **ε**
Digits → **Digit More**
More → **Digits** | **ε**
Digit → **0** | **1** | … | **9**

Num	Sign	Digit	Digits	More
+ – 0 5 1 6 2 7 3 8 4 9	+ – ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	–	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign	+	-		
Digits				
More				
Digit				

It Gets Trickier

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ − 0 5 1 6 2 7 3 8 4 9	+ − ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign	+	-		
Digits			**Digits More**	
More				
Digit				

It Gets Trickier

Num → Sign Digits
Sign → + | - | ε
Digits → Digit More
More → Digits | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ − 0 5 1 6 2 7 3 8 4 9	+ − ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	Sign Digits	Sign Digits		
Sign	+	-		
Digits			Digits More	
More				
Digit				

It Gets Trickier

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ – 0 5 1 6 2 7 3 8 4 9	+ – ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign	+	-		
Digits			**Digits More**	
More			**Digits**	
Digit				

It Gets Trickier

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ - 0 5 1 6 2 7 3 8 4 9	+ - ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign	+	-		
Digits			**Digits More**	
More			**Digits**	
Digit				

It Gets Trickier

Num → Sign Digits
Sign → + | - | ε
Digits → Digit More
More → Digits | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ - 0 5 1 6 2 7 3 8 4 9	+ - ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	Sign Digits	Sign Digits		
Sign	+	-		
Digits			Digits More	
More			Digits	
Digit			#	

It Gets Trickier

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ – 0 5 1 6 2 7 3 8 4 9	+ – ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign	+	-		
Digits			**Digits More**	
More			**Digits**	
Digit			#	

It Gets Trickier

Num	→	**Sign Digits**
Sign	→	**+** | **-** | **ε**
Digits	→	**Digit More**
More	→	**Digits** | **ε**
Digit	→	**0** | **1** | ... | **9**

Num	Sign	Digit	Digits	More
+ - 0 5 1 6 2 7 3 8 4 9	+ - ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**		
Sign	+	-		
Digits			**Digits More**	
More			**Digits**	
Digit			**#**	

It Gets Trickier

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ − 0 5 1 6 2 7 3 8 4 9	+ − ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	Sign Digits	Sign Digits	Sign Digits	
Sign	+	-		
Digits			Digits More	
More			Digits	
Digit			#	

It Gets Trickier

Num → **Sign Digits**
Sign → **+** | **-** | **ε**
Digits → **Digit More**
More → **Digits** | **ε**
Digit → **0** | **1** | ... | **9**

Num	Sign	Digit	Digits	More
+ − 0 5 1 6 2 7 3 8 4 9	+ − ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**	**Sign Digits**	
Sign	+	-		
Digits			**Digits More**	
More			**Digits**	
Digit			**#**	

It Gets Trickier

Num → **Sign Digits**
Sign → + | – | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ – 0 5 1 6 2 7 3 8 4 9	+ – ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	–	#	$
Num	**Sign Digits**	**Sign Digits**	**Sign Digits**	
Sign	+	-	ε	
Digits			**Digits More**	
More			**Digits**	
Digit			#	

It Gets Trickier

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ − 0 5 1 6 2 7 3 8 4 9	+ − ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	Sign Digits	Sign Digits	Sign Digits	
Sign	+	-	ε	
Digits			Digits More	
More			Digits	
Digit			#	

It Gets Trickier

Num → **Sign Digits**
Sign → **+** | **-** | **ε**
Digits → **Digit More**
More → **Digits** | **ε**
Digit → **0** | **1** | ... | **9**

Num	Sign	Digit	Digits	More
+ – 0 5 1 6 2 7 3 8 4 9	+ – ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**	**Sign Digits**	
Sign	+	-	ε	
Digits			**Digits More**	
More			**Digits**	
Digit			#	

It Gets Trickier

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ – 0 5 1 6 2 7 3 8 4 9	+ – ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	–	#	$
Num	**Sign Digits**	**Sign Digits**	**Sign Digits**	
Sign	+	-	ε	
Digits			**Digits More**	
More			**Digits**	ε
Digit			#	

It Gets Trickier

Num → **Sign Digits**
Sign → **+** | **–** | **ε**
Digits → **Digit More**
More → **Digits** | **ε**
Digit → **0** | **1** | … | **9**

Num	Sign	Digit	Digits	More
+ – 0 5 1 6 2 7 3 8 4 9	+ – ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	–	#	$
Num	**Sign Digits**	**Sign Digits**	**Sign Digits**	
Sign	+	-	ε	
Digits			**Digits More**	
More			**Digits**	ε
Digit			#	

It Gets Trickier

Num → **Sign Digits**
Sign → + | - | ε
Digits → **Digit More**
More → **Digits** | ε
Digit → 0 | 1 | … | 9

Num	Sign	Digit	Digits	More
+ − 0 5 1 6 2 7 3 8 4 9	+ − ε	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9	0 5 1 6 2 7 3 8 4 9 ε

	+	-	#	$
Num	**Sign Digits**	**Sign Digits**	**Sign Digits**	
Sign	+	-	ε	
Digits			**Digits More**	
More			**Digits**	ε
Digit			#	

FOLLOW Sets

- With ε-productions in the grammar, we may have to "look past" the current nonterminal to what can come after it.
- The **FOLLOW set** represents the set of terminals that might come after a given nonterminal.
- Formally:

$$\text{FOLLOW}(\mathbf{A}) = \{ \mathtt{t} \mid \mathbf{S} \Rightarrow^* \boldsymbol{\alpha}\mathbf{A}\mathtt{t}\boldsymbol{\omega} \text{ for some } \boldsymbol{\alpha}, \boldsymbol{\omega} \}$$

  where **S** is the start symbol of the grammar.
- Informally, every nonterminal that can ever come after **A** in a derivation.

Computation of FOLLOW Sets

- Initially, for each nonterminal **A**, set

  FOLLOW(**A**) = { **t** | **B** → ***α*A**t***ω*** is a production }

- Add **$** to FOLLOW(**S**), where **S** is the start symbol.
- Repeat the following until no changes occur:
  - If **B** → ***α*A*ω*** is a production, set FOLLOW(**A**) = FOLLOW(**A**) ∪ FIRST*(***ω***) - { **ε** }.
  - If **B** → ***α*A*ω*** is a production and **ε** ∈ FIRST*(***ω***), set FOLLOW(**A**) = FOLLOW(**A**) ∪ FOLLOW(**B**).

The Final LL(1) Table Algorithm

- Compute FIRST(**A**) and FOLLOW(**A**) for all nonterminals **A**.
- For each rule **A** → ***ω***, for each terminal **`t`** ∈ FIRST*(***ω***), set T[**A**, **`t`**] = ***ω***.
  - Note that **ε** is not a terminal.
- For each rule **A** → ***ω***, if **ε** ∈ FIRST*(***ω***), set T[**A**, **`t`**] = ***ω*** for each **`t`** ∈ FOLLOW(**A**).

An Egregious Abuse of Notation

- Compute FIRST(**A**) and FOLLOW(**A**) for all nonterminals **A**.
- For each rule **A** → $\boldsymbol{\omega}$, for each terminal `t` ∈ FIRST*($\boldsymbol{\omega}$FOLLOW(**A**)), set T[**A**, `t`] = $\boldsymbol{\omega}$.

Example LL(1) Construction

The Limits of LL(1)

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → **A**`b` | `c`

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

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  $A \to Ab \mid c$

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

	b	c
A		$A \to Ab$ $A \to c$

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → **A**`b` | `c`

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

	b	c
A		**A** → **A**`b` **A** → `c`

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

Eliminating Left Recursion

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

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

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

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

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

Another Non-LL(1) Grammar

- Consider the following grammar:

  E → **T**

  E → **T** `+` **E**

  T → `int`

  T → `(`**E**`)`

- FIRST(**E**) = { `int`, `(` }
- FIRST(**T**) = { `int`, `(` }
- Why is this grammar not LL(1)?

Another Non-LL(1) Grammar

- Consider the following grammar:

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

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

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

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

How do you predict which of these to use?

- FIRST(**E**) = { `int`, `(` }
- FIRST(**T**) = { `int`, `(` }
- Why is this grammar not LL(1)?

Left-Factoring

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

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

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

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

Left-Factoring

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

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

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

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

Left-Factoring

$\mathbf{E} \to \mathbf{TY}$

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

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

Left-Factoring

$E \rightarrow TY$

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

Left-Factoring

E → **TY**		**1**
T → `int`		**2**
T → `(`**E**`)`		**3**
Y → `+` **E**		**4**
Y → `ε`		**5**

Left-Factoring

$E \to TY$		1
$T \to$ `int`		2
$T \to$ `(`E`)`		3
$Y \to$ `+` E		4
$Y \to \varepsilon$		5

Left-Factoring

E → TY		1
T → `int`		2
T → `(`E`)`		3
Y → `+` E		4
Y → ε		5

FIRST		
E	T	Y

FOLLOW		
E	T	Y

Left-Factoring

E → **TY**		**1**
T → `int`		**2**
T → **(E)**		**3**
Y → **+ E**		**4**
Y → **ε**		**5**

FIRST		
E	T	Y
	`int` `(`	

FOLLOW		
E	T	Y

Left-Factoring

E → **TY**		**1**
T → `int`		**2**
T → **(E)**		**3**
Y → **+ E**		**4**
Y → **ε**		**5**

FIRST		
E	T	Y
	`int` `(`	`+` `ε`

FOLLOW		
E	T	Y

Left-Factoring

E → TY		**1**
T → `int`		**2**
T → `(`E`)`		**3**
Y → `+` E		**4**
Y → ε		**5**

FIRST		
E	T	Y
`int` `(`	`int` `(`	`+` ε
FOLLOW		
E	T	Y

Left-Factoring

E → TY		1
T → int		2
T → (E)		3
Y → + E		4
Y → ε		5

FIRST		
E	T	Y
int (	int (	+ ε

FOLLOW		
E	T	Y
$		

Left-Factoring

E	→ TY	1
T	→ int	2
T	→ (E)	3
Y	→ + E	4
Y	→ ε	5

FIRST		
E	T	Y
int (	int (	+ ε

FOLLOW		
E	T	Y
\$)		

Left-Factoring

E → TY		1
T → `int`		2
T → `(`E`)`		3
Y → `+` E		4
Y → ε		5

FIRST		
E	T	Y
`int` `(`	`int` `(`	`+` ε

FOLLOW		
E	T	Y
`$` `)`	`+`	

Left-Factoring

E → TY		1
T → `int`		2
T → `(`E`)`		3
Y → `+` E		4
Y → ε		5

FIRST		
E	T	Y
`int` `(`	`int` `(`	`+` ε

FOLLOW		
E	T	Y
`$` `)`	`+`	`$` `)`

Left-Factoring

E → TY		1
T → `int`		2
T → `(E)`		3
Y → `+` E		4
Y → ε		5

FIRST		
E	T	Y
`int` `(`	`int` `(`	`+` ε

FOLLOW		
E	T	Y
`$` `)`	`+` `$` `)`	`$` `)`

Left-Factoring

E → TY		1
T → `int`		2
T → (E)		3
Y → + E		4
Y → ε		5

FIRST		
E	T	Y
`int` `(`	`int` `(`	`+` `ε`

FOLLOW		
E	T	Y

`$`
`)`
	`+`
	`$`
	`)`
		`$`
		`)`

	`int`	`(`	`)`	`+`	`$`
E					
T					
Y					

Left-Factoring

E → TY		**1**
T → `int`		**2**
T → (E)		**3**
Y → + E		**4**
Y → ε		**5**

FIRST		
E	T	Y
`int` `(`	`int` `(`	`+` `ε`

FOLLOW		
E	T	Y

`$`
`)`
	`+`
	`$`
	`)`
		`$`
		`)`

	`int`	`(`	`)`	`+`	`$`
E	**1**	**1**			
T					
Y					

Left-Factoring

$E \rightarrow TY$		1
$T \rightarrow$ `int`		2
$T \rightarrow$ `(`E`)`		3
$Y \rightarrow$ `+` E		4
$Y \rightarrow \varepsilon$		5

FIRST		
E	T	Y
int (	int (	+ ε

FOLLOW		
E	T	Y
$)	+ $)	$)

	int	(	)	+	$
E	1	1			
T	2	3			
Y					

Left-Factoring

E → TY		1
T → `int`		2
T → `(`E`)`		3
Y → `+` E		4
Y → ε		5

FIRST		
E	T	Y
`int` `(`	`int` `(`	`+` ε

FOLLOW		
E	T	Y
`$` `)`	`+` `$` `)`	`$` `)`

	`int`	`(`	`)`	`+`	`$`
E	1	1			
T	2	3			
Y				4	

Left-Factoring

E → TY		1
T → int		2
T → (E)		3
Y → + E		4
Y → ε		5

FIRST		
E	T	Y
int (	int (	+ ε

FOLLOW		
E	T	Y
\$)	+ \$)	\$)

	int	(	)	+	\$
E	1	1			
T	2	3			
Y			5	4	5

A Formal Characterization of LL(1)

- A grammar G is LL(1) iff for any productions $\mathbf{A} \to \boldsymbol{\omega}_1$ and $\mathbf{A} \to \boldsymbol{\omega}_2$, the sets

  $$\text{FIRST}(\boldsymbol{\omega}_1\ \text{FOLLOW}(\mathbf{A}))$$

  and

  $$\text{FIRST}(\boldsymbol{\omega}_2\ \text{FOLLOW}(\mathbf{A}))$$

  are disjoint.
- This condition is equivalent to saying that there are no conflicts in the table.

The Strengths of LL(1)

LL(1) is Straightforward

- Can be implemented quickly with a table-driven design.
- Can be implemented by **recursive descent**:
  - Define a function for each nonterminal.
  - Have these functions call each other based on the lookahead token.
- See Handout #09 for more details.

LL(1) is Fast

- Both table-driven LL(1) and recursive-descent-powered LL(1) are fast.
- Can parse in O(n $|G|$) time, where n is the length of the string and $|G|$ is the size of the grammar.

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.
- **FIRST sets** contain terminals that may be the first symbol of a production.
- **FOLLOW sets** contain terminals that may follow a nonterminal in a production.
- **Left recursion** and **left factorability** cause LL(1) to fail and can be mechanically eliminated in some cases.