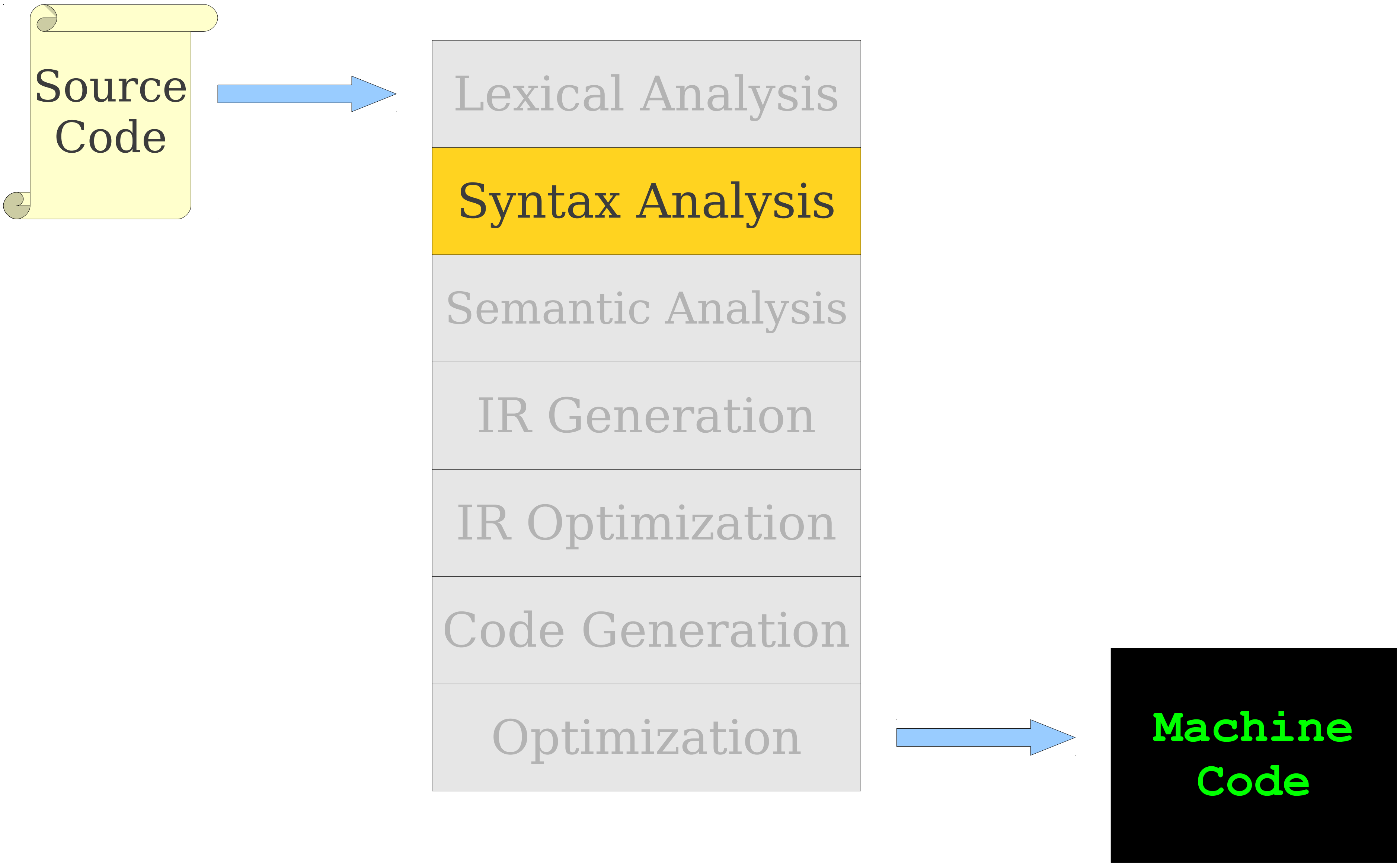
Top-Down Parsing

Announcements

- Office hours schedule posted on Piazza.
  - Keith:
    - Monday/Tuesday, 2PM – 4PM in Gates 178.
  - Jinchao:
    - Wednesday/Thursday, 6PM – 8PM in Gates B26.
- Feel free to email us with questions!
- Sign up for Piazza (www.piazza.com).

Where We Are



Source Code

Lexical Analysis

Syntax Analysis

Semantic Analysis

IR Generation

IR Optimization

Code Generation

Optimization

Machine Code

Review from Last Time

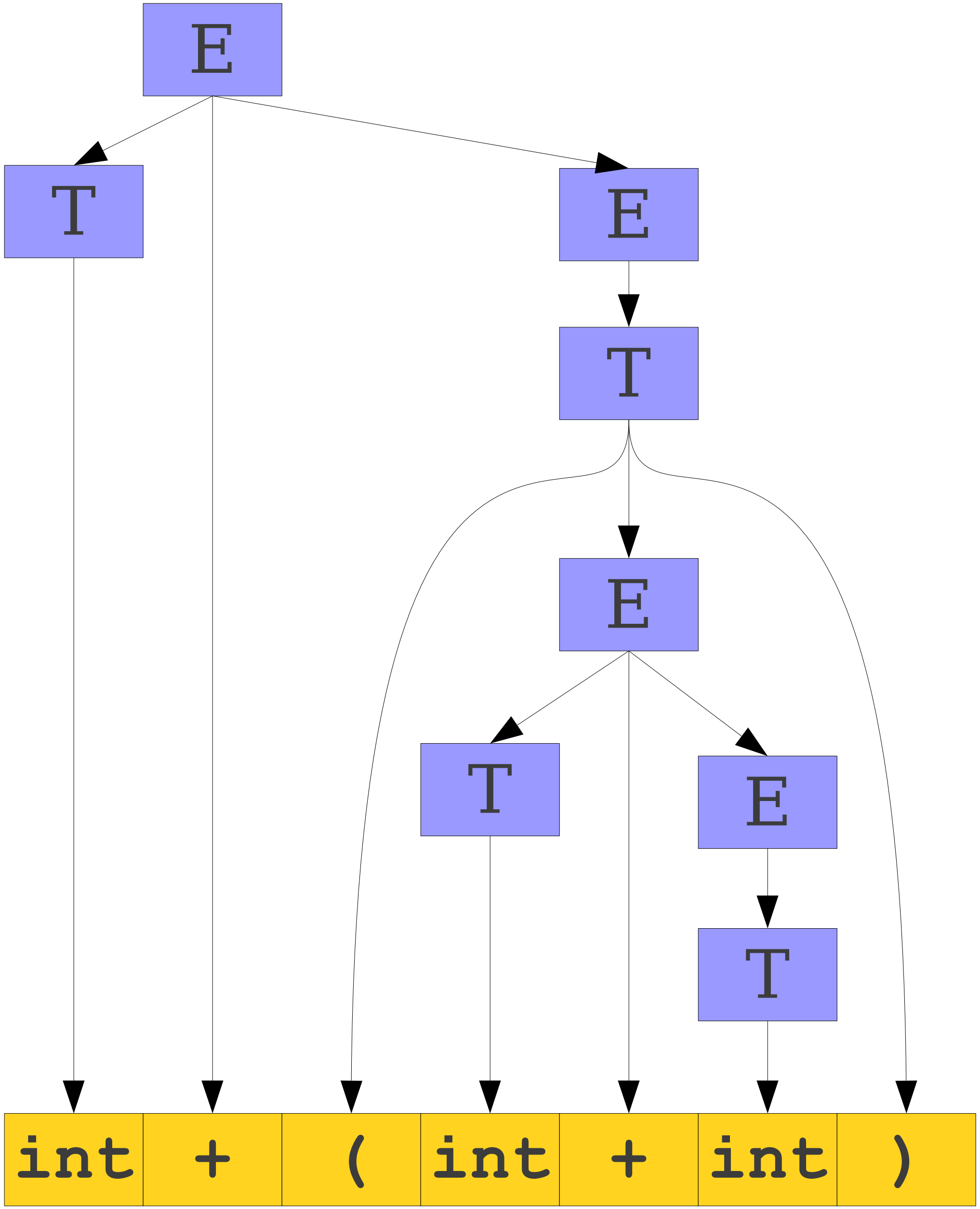
- Goal of syntax analysis: recover the intended structure of the program.
- Idea: Use a **context-free grammar** to describe the programming language.
- Given a sequence of tokens, look for a **parse tree** that generates those tokens.
- Recovering this syntax tree is called **parsing** and is the topic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

Different Types of Parsing

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

Top-Down Parsing

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



Challenges in Top-Down Parsing

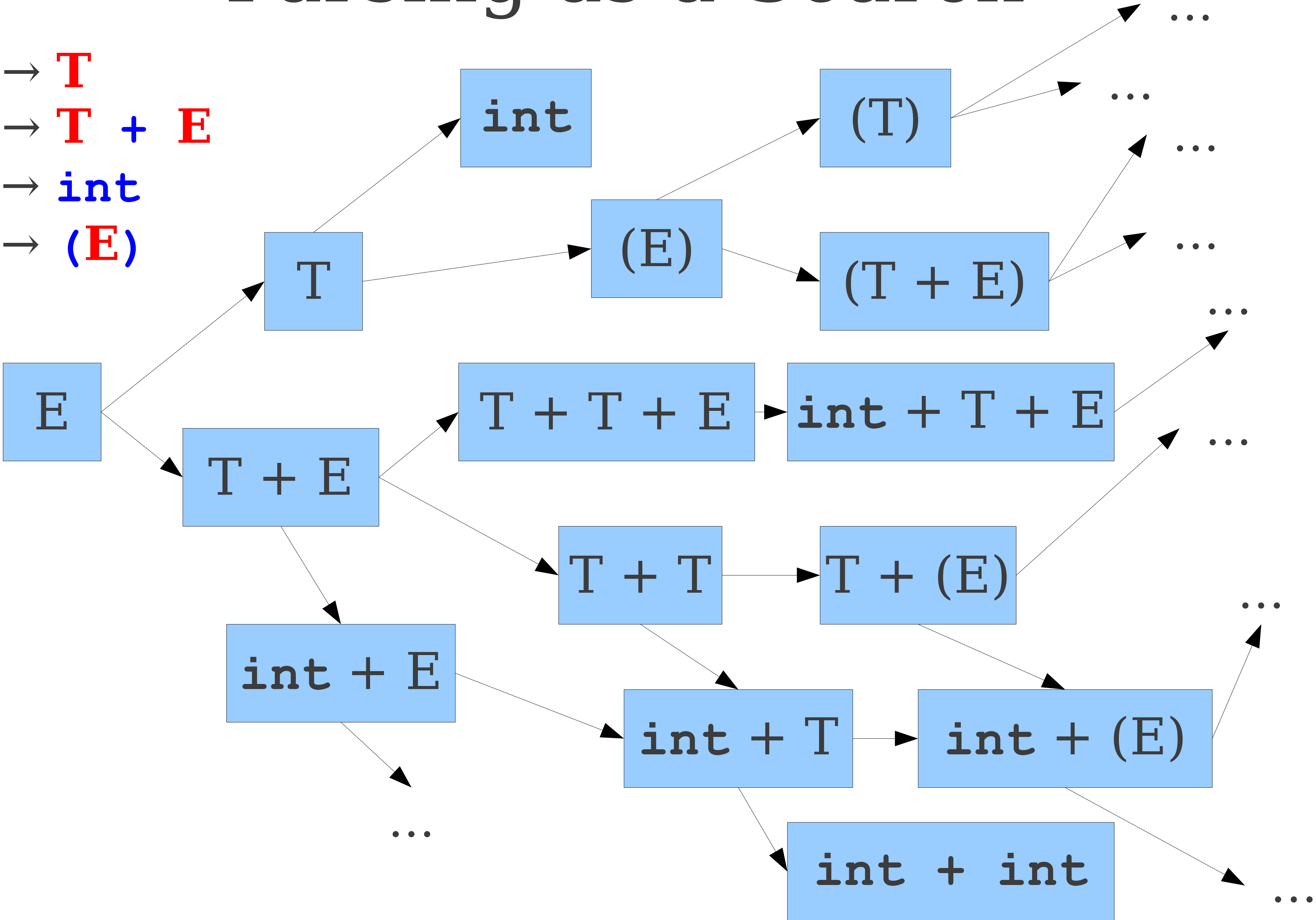
- Top-down parsing begins with virtually no information.
    - Begins with just the start symbol, which matches *every* program.
- How can we know which productions to apply?
- In general, we can't.
    - There are some grammars for which the best we can do is guess and backtrack if we're wrong.
   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



E

T

T + E

int

(E)

T + T + E

T + T

int + E

(T)

(T + E)

int + T + E

T + (E)

int + T

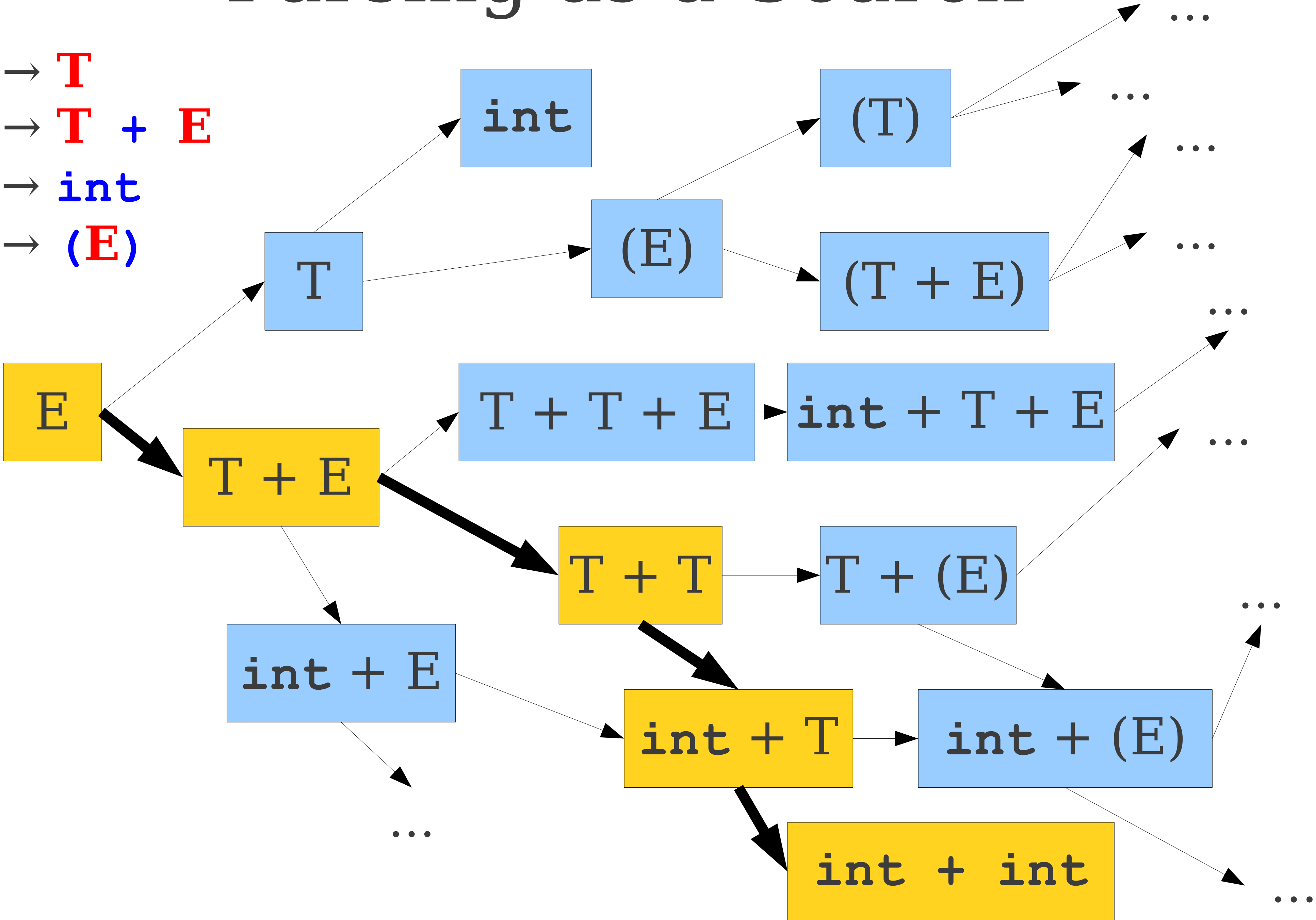
int + (E)

int + int

…

Parsing as a Search

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



Our First Top-Down Algorithm

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

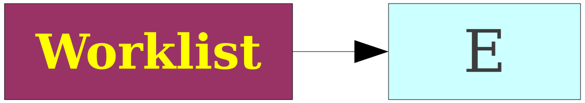
Breadth-First Search Parsing

Worklist

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

`int + int`

Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing

Worklist

E

E → **T**

E → **T** `+` **E**

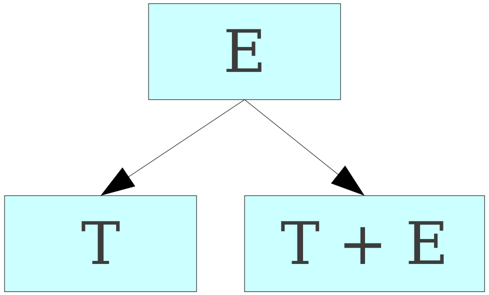
T → `int`

T → `(`**E**`)`

`int + int`

Breadth-First Search Parsing

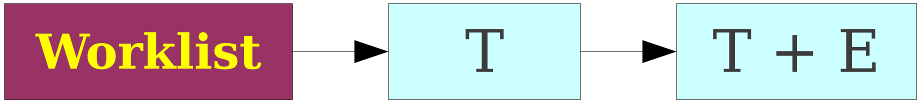
Worklist



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

`int + int`

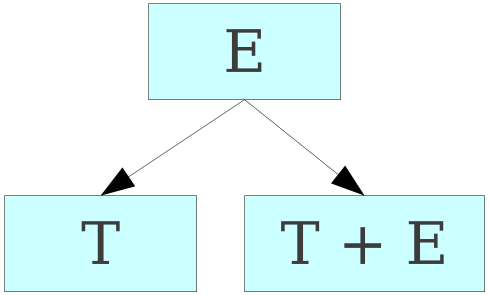
Breadth-First Search Parsing



Worklist

T

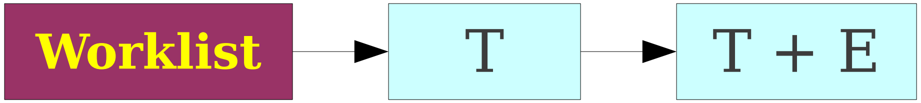
T + E



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

`int + int`

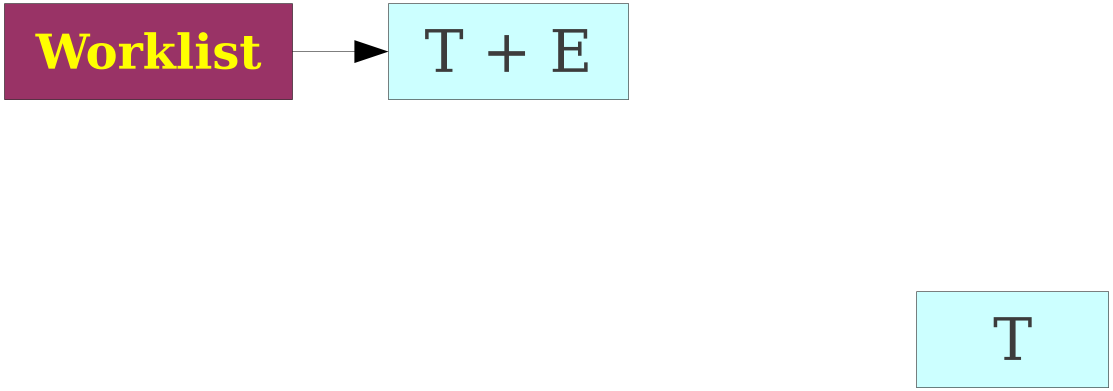
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing



E → **T**

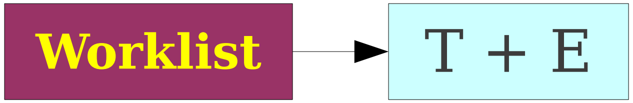
E → **T** `+` **E**

T → `int`

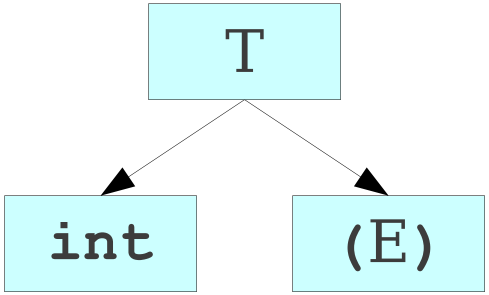
T → `(`**E**`)`

`int + int`

Breadth-First Search Parsing


Worklist

T + E


T

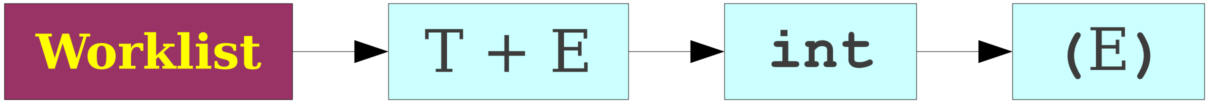
`int`

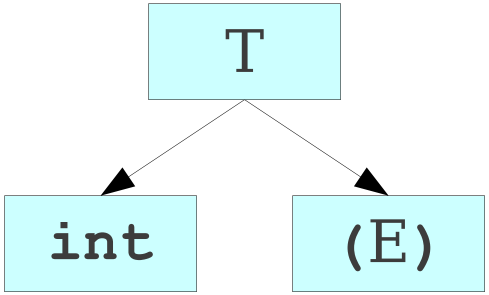
(E)

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

`int + int`

Breadth-First Search Parsing

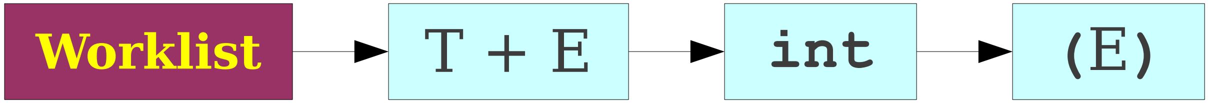




E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

`int + int`

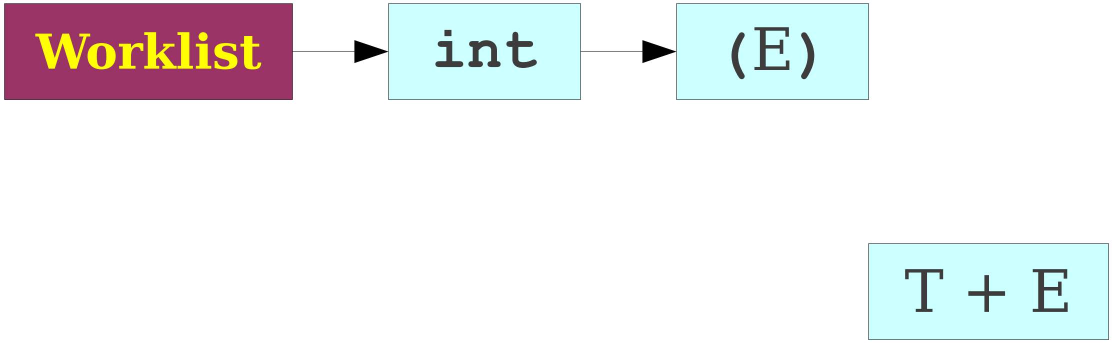
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing


Worklist

`int`

(E)

T + E

E → **T**

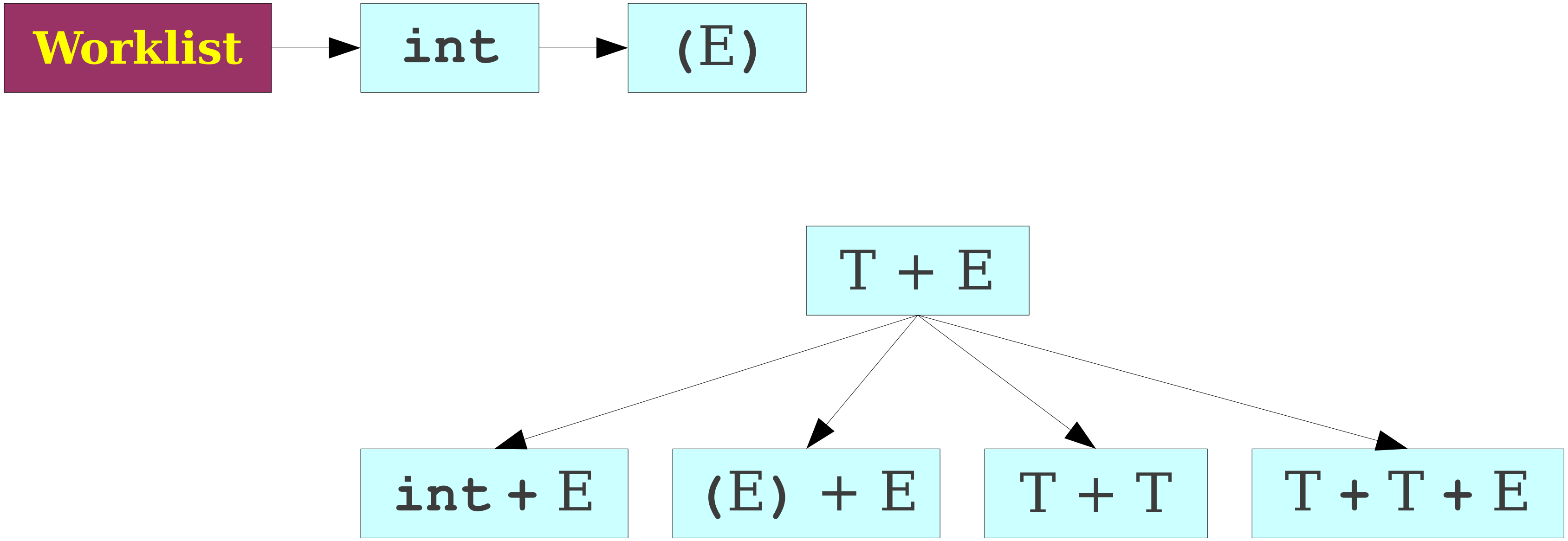
E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

Breadth-First Search Parsing



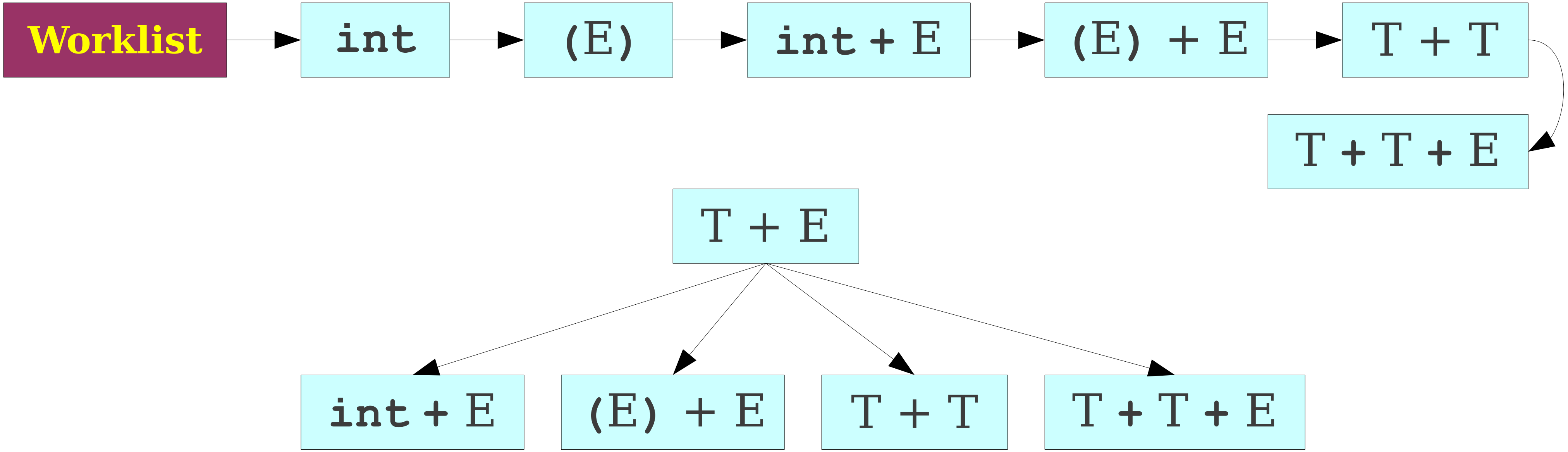
Worklist → `int` → (E)

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

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

`int + int`

Breadth-First Search Parsing



Worklist

`int`

(E)

`int` + E

(E) + E

T + T

T + T + E

T + E

`int` + E

(E) + E

T + T

T + T + E

E → T

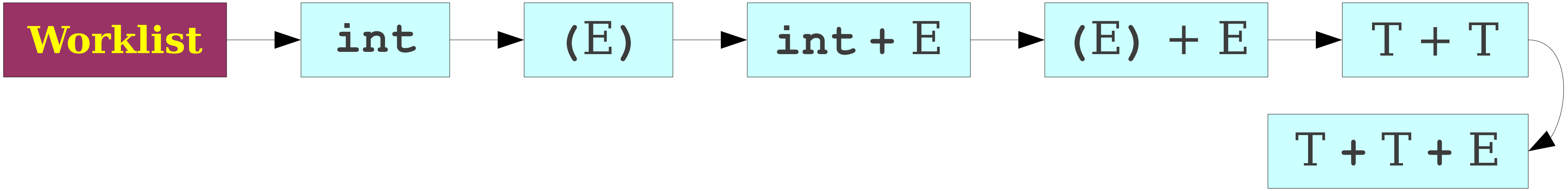
E → T `+` E

T → `int`

T → `(`E`)`

`int + int`

Breadth-First Search Parsing


Worklist

`int`

(E)

`int` + E

(E) + E

T + T

T + T + E

E → T

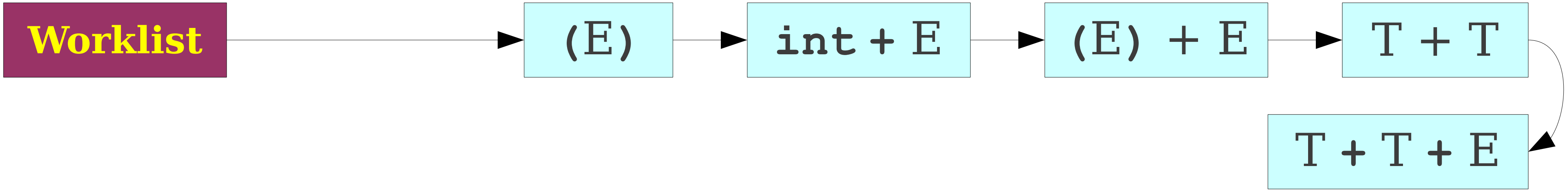
E → T `+` E

T → `int`

T → `(`E`)`

`int + int`

Breadth-First Search Parsing



Worklist

(E)

int + E

(E) + E

T + T

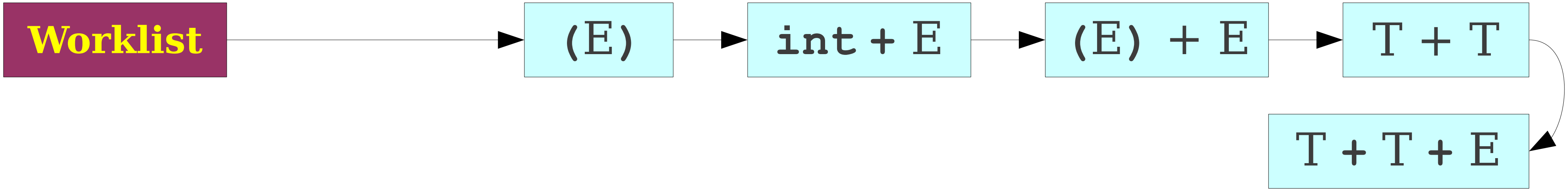
T + T + E

int

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

int + int

Breadth-First Search Parsing



Worklist

(E)

`int` + E

(E) + E

T + T

T + T + E

E → T

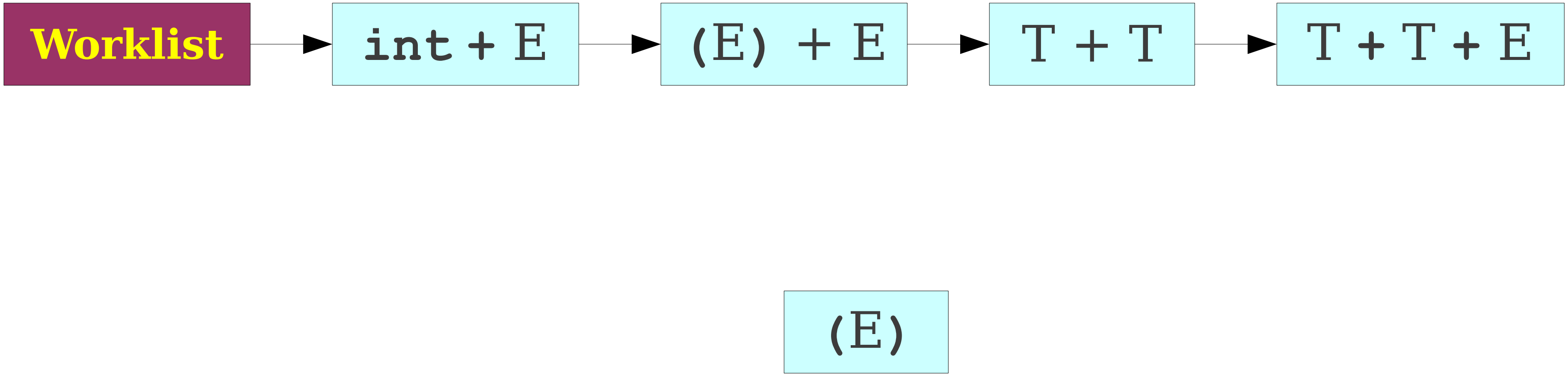
E → T `+` E

T → `int`

T → `(`E`)`

`int + int`

Breadth-First Search Parsing



Worklist

`int` + E

`(`E`)` + E

T + T

T `+` T `+` E

`(`E`)`

E → **T**

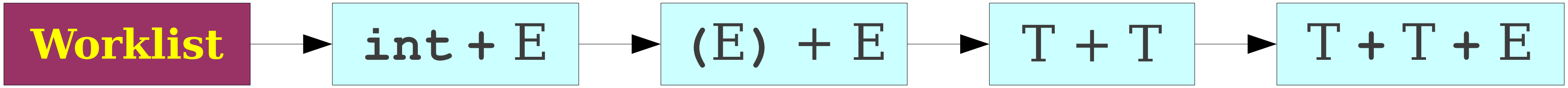
E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

Breadth-First Search Parsing



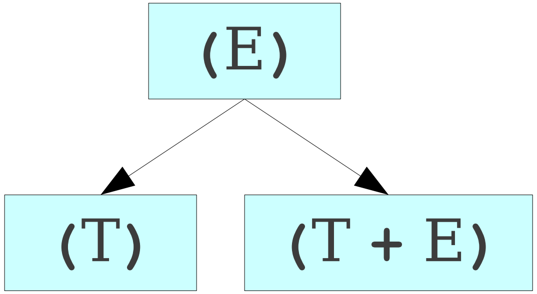
Worklist

`int` + E

`(`E`)` + E

T + T

T `+` T `+` E



`(`E`)`

`(`T`)`

`(`T `+` E`)`

E → **T**

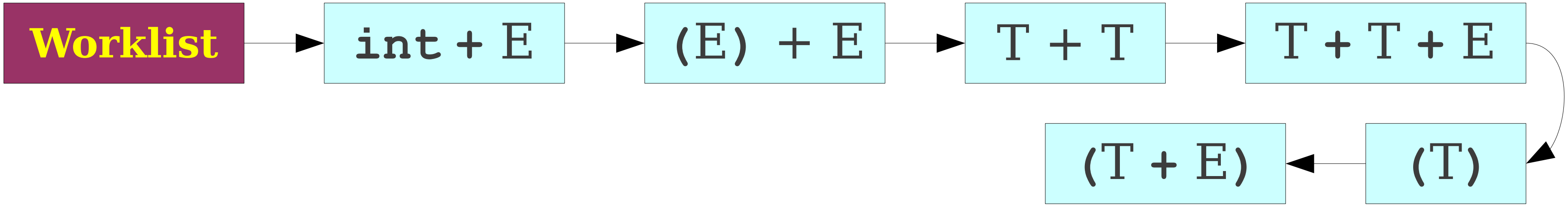
E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

Breadth-First Search Parsing



Worklist

`int` + E

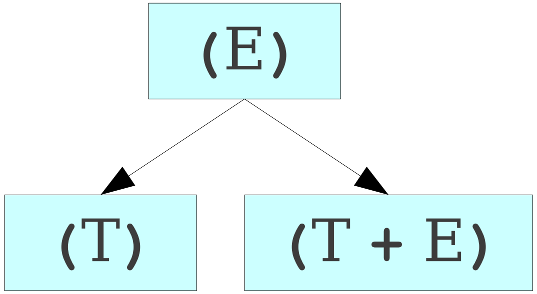
(E) + E

T + T

T + T + E

(T)

(T + E)



(E)

(T)

(T + E)

E → **T**

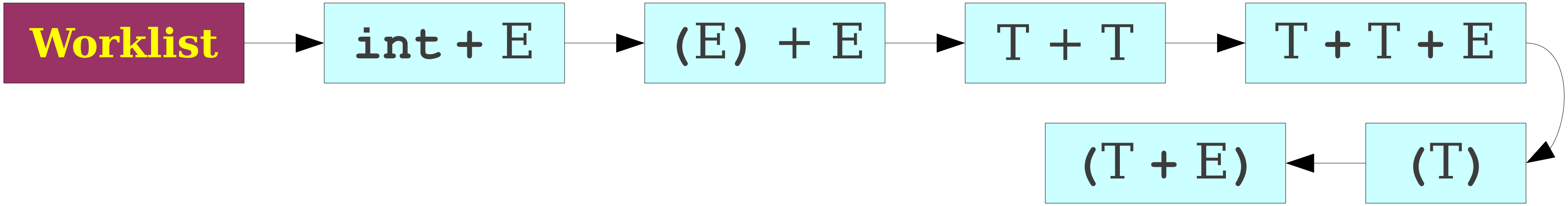
E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

Breadth-First Search Parsing



Worklist

int + E

(E) + E

T + T

T + T + E

(T)

(T + E)

E → **T**

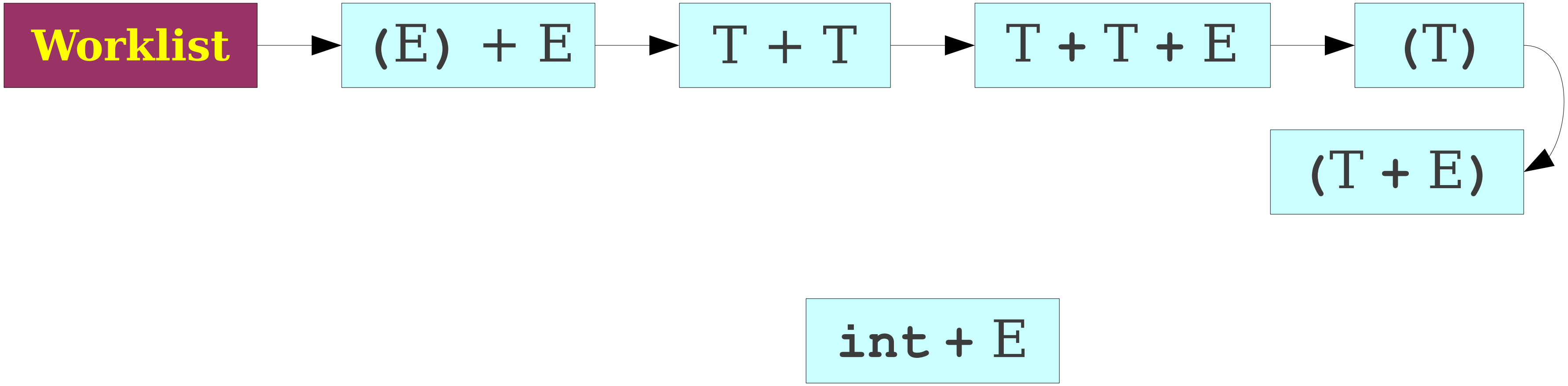
E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

Breadth-First Search Parsing



Worklist

(E) + E

T + T

T + T + E

(T)

(T + E)

int + E

E → T

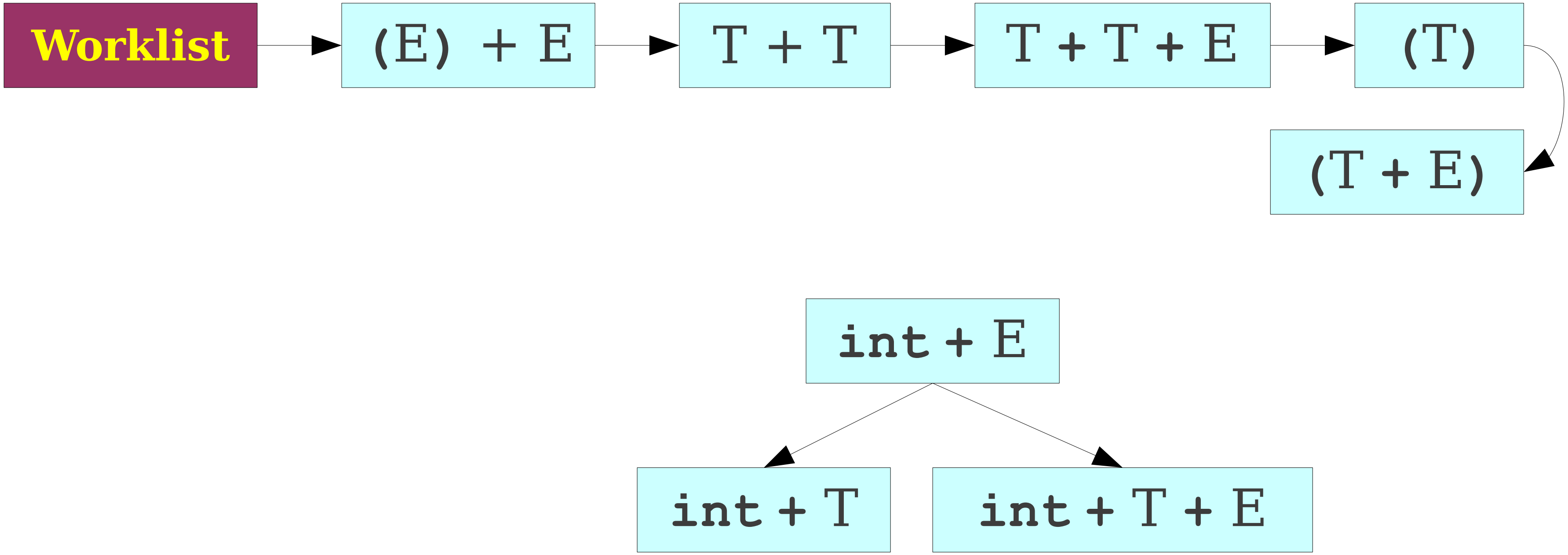
E → T + E

T → int

T → (E)

int + int

Breadth-First Search Parsing



Worklist

(E) + E

T + T

T + T + E

(T)

(T + E)

int + E

int + T

int + T + E

E → **T**

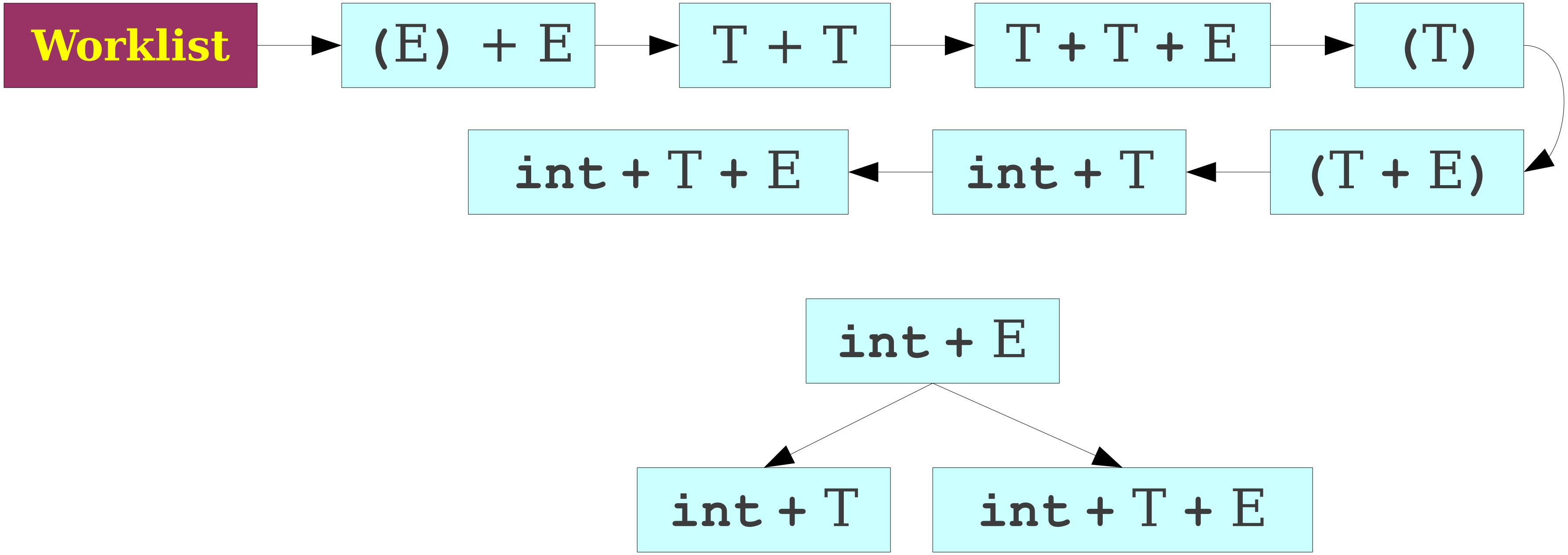
E → **T** **+** **E**

T → **int**

T → **(E)**

int + int

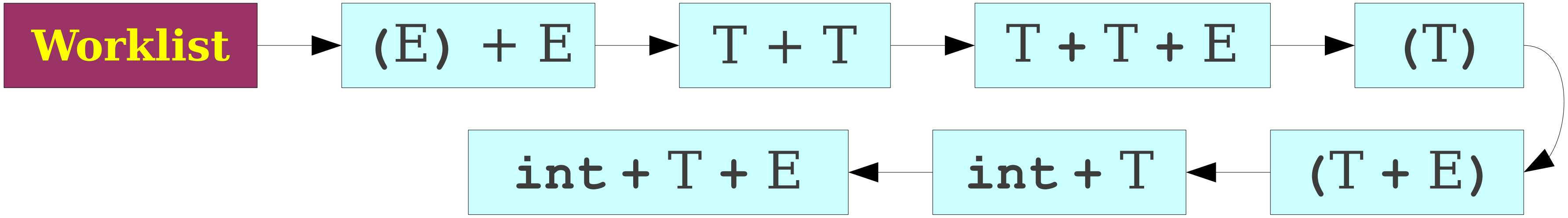
Breadth-First Search Parsing



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

int + int

Breadth-First Search Parsing



Worklist

(E) + E

T + T

T + T + E

(T)

(T + E)

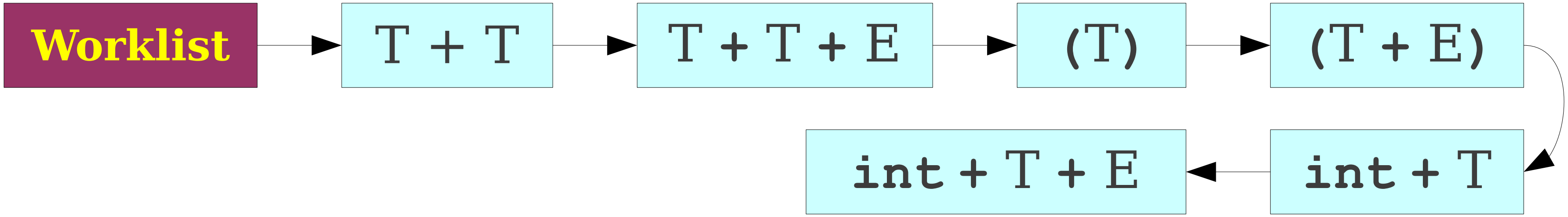
int + T

int + T + E

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

`int + int`

Breadth-First Search Parsing



Worklist

T + T

T + T + E

(T)

(T + E)

int + T

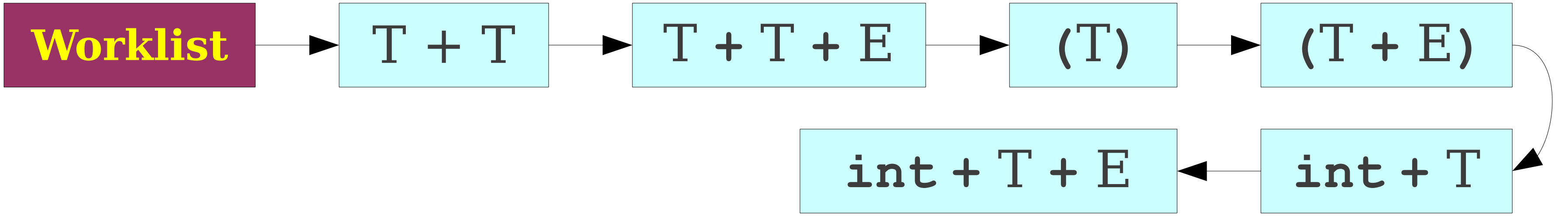
int + T + E

(E) + E

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

`int + int`

Breadth-First Search Parsing



Worklist

T + T

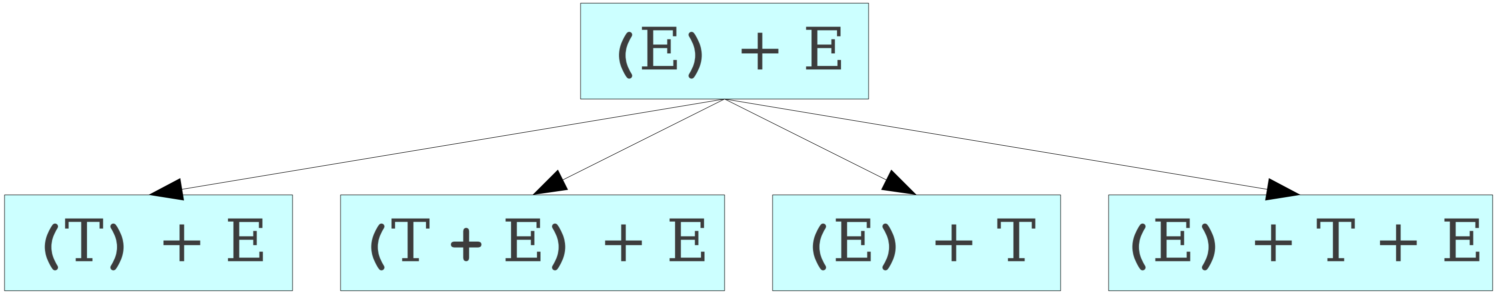
T + T + E

(T)

(T + E)

`int` + T

`int` + T + E



(E) + E

(T) + E

(T + E) + E

(E) + T

(E) + T + E

E → **T**

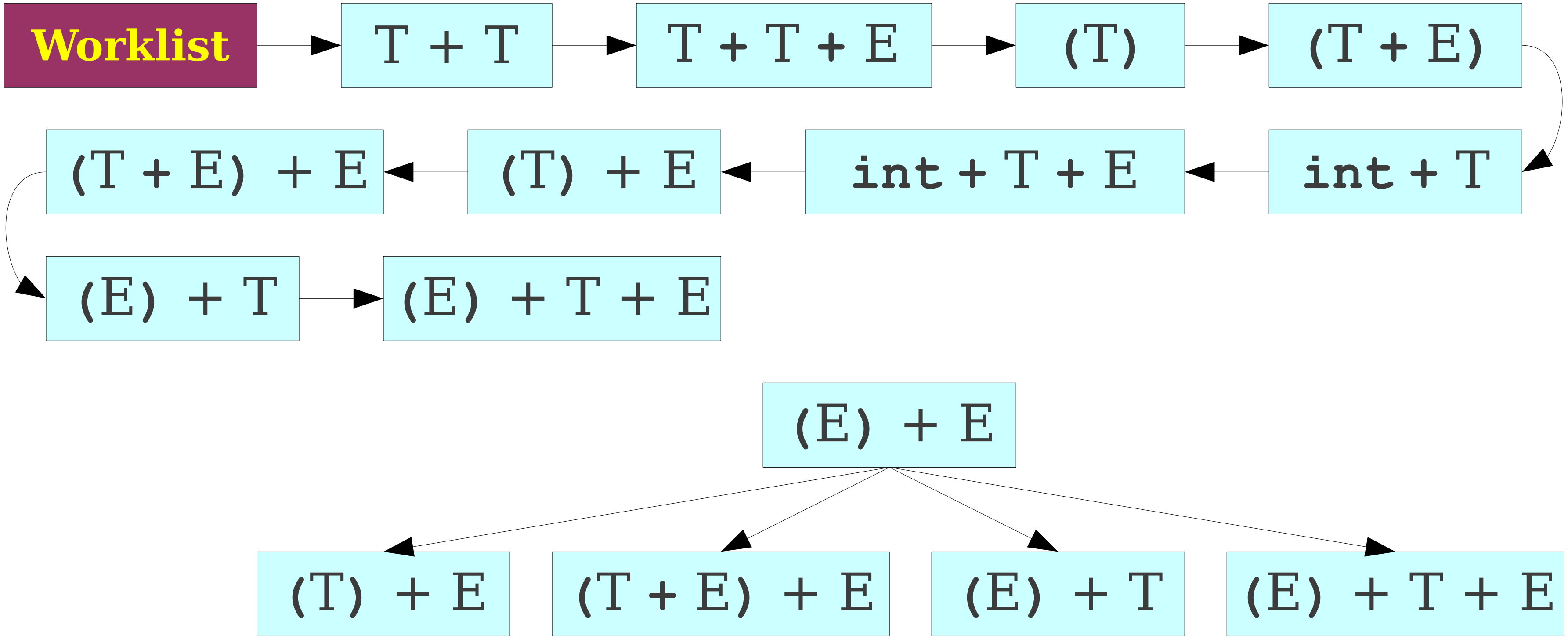
E → **T** **+** **E**

T → **`int`**

T → **`(`E`)`**

`int + int`

Breadth-First Search Parsing



Worklist

T + T

T + T + E

(T)

(T + E)

int + T

int + T + E

(T) + E

(T + E) + E

(E) + T

(E) + T + E

(E) + E

(T) + E

(T + E) + E

(E) + T

(E) + T + E

E → **T**

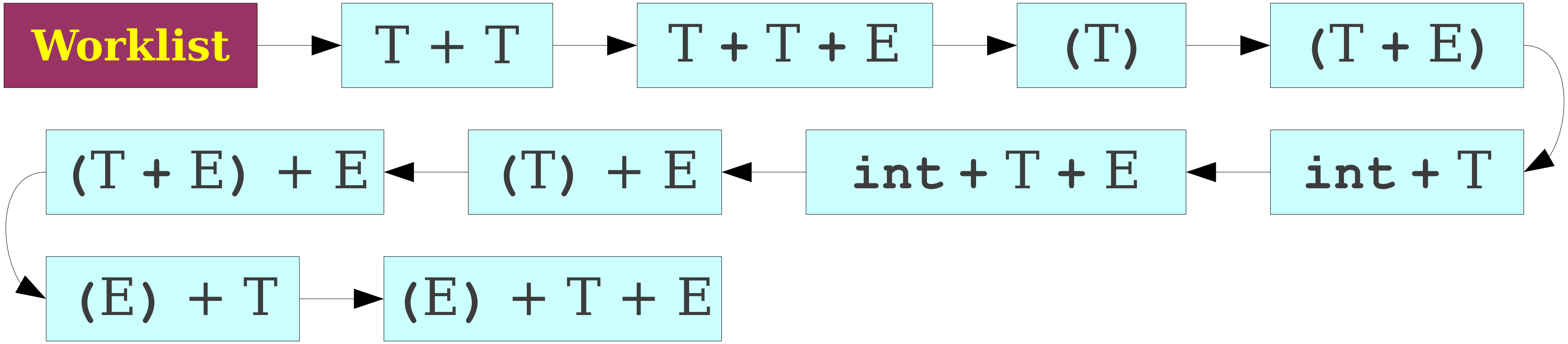
E → **T** **+** **E**

T → **int**

T → **(E)**

`int + int`

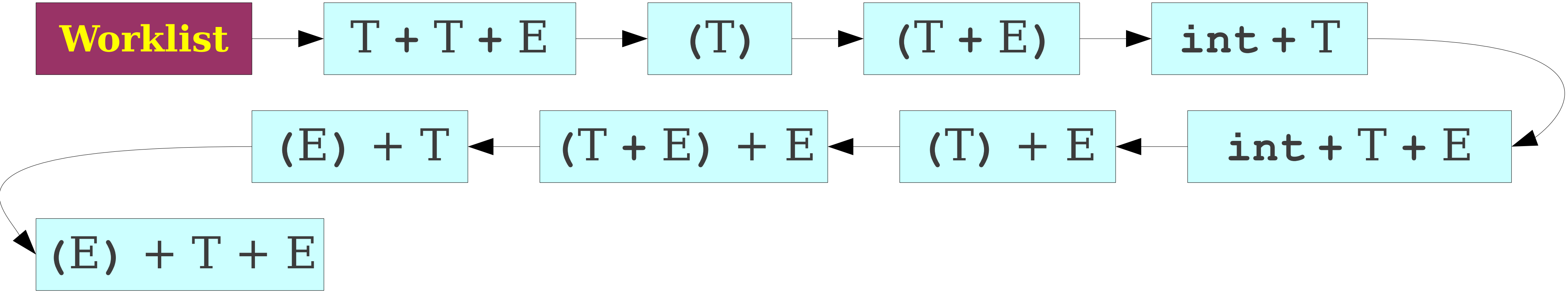
Breadth-First Search Parsing



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

`int + int`

Breadth-First Search Parsing

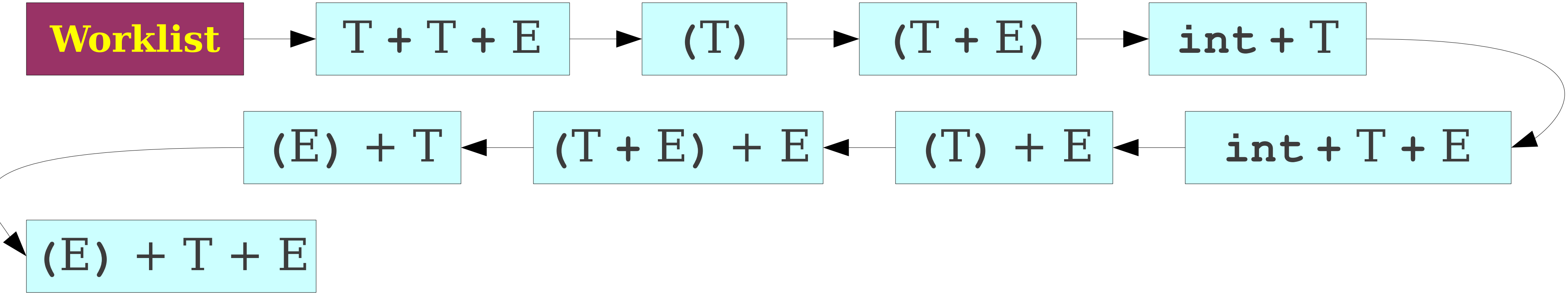


T + T

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

`int + int`

Breadth-First Search Parsing



Worklist

T + T + E

(T)

(T + E)

int + T

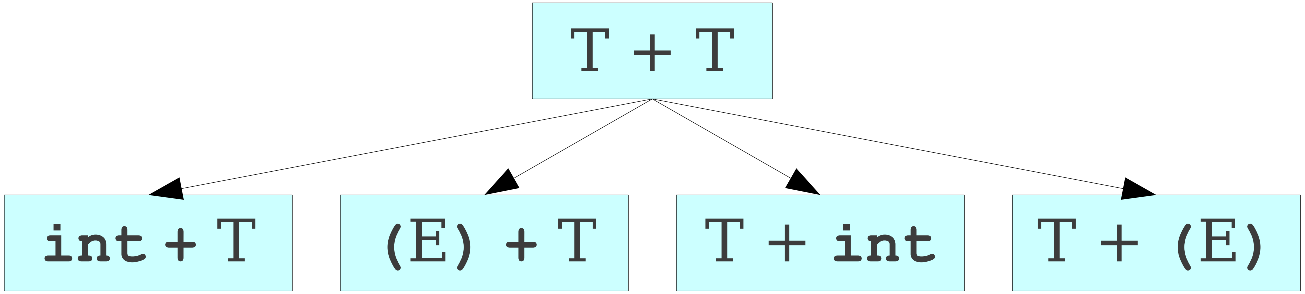
int + T + E

(T) + E

(T + E) + E

(E) + T

(E) + T + E



T + T

int + T

(E) + T

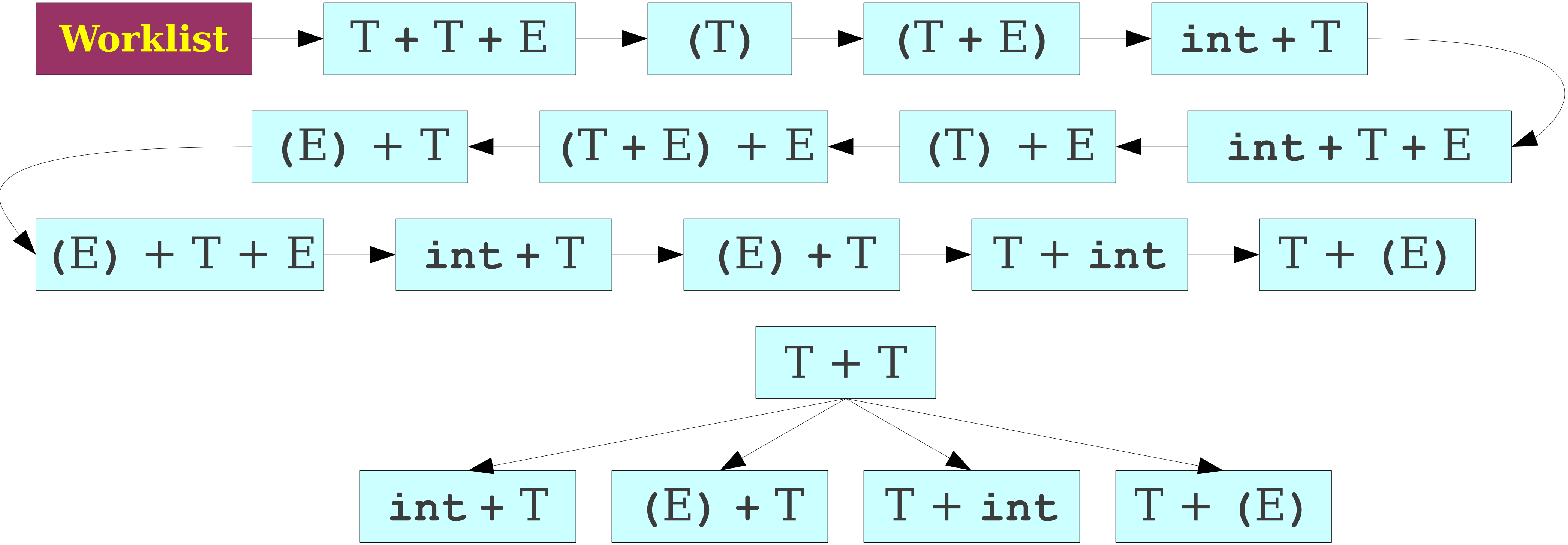
T + int

T + (E)

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

int + int

Breadth-First Search Parsing



Worklist

T + T + E

(T)

(T + E)

int + T

int + T + E

(T) + E

(T + E) + E

(E) + T

(E) + T + E

int + T

(E) + T

T + int

T + (E)

T + T

int + T

(E) + T

T + int

T + (E)

E → T

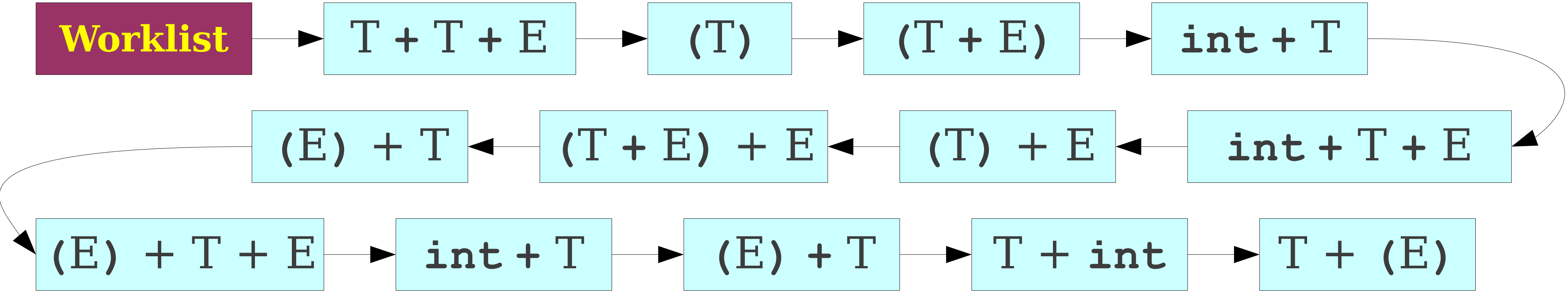
E → T + E

T → int

T → (E)

int + int

Breadth-First Search Parsing



Worklist

T + T + E

(T)

(T + E)

int + T

int + T + E

(T) + E

(T + E) + E

(E) + T

(E) + T + E

int + T

(E) + T

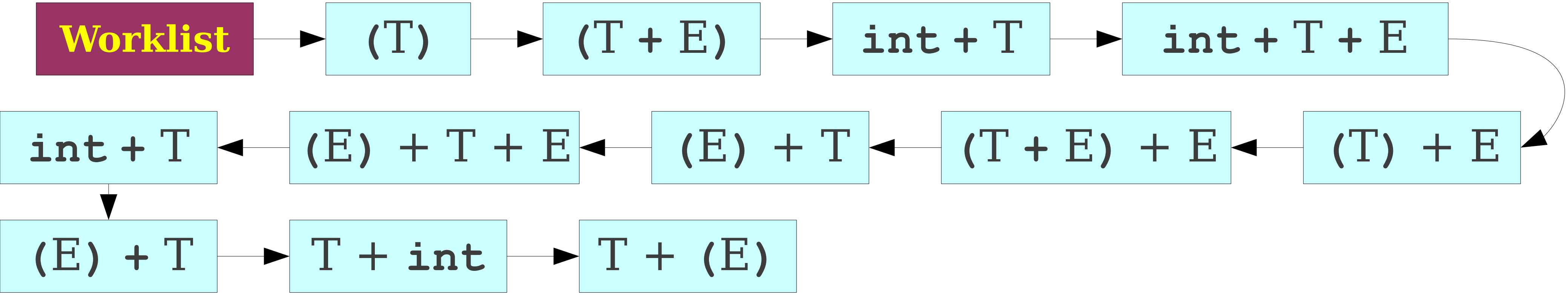
T + int

T + (E)

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

int + int

Breadth-First Search Parsing



Worklist

(T)

(T + E)

int + T

int + T + E

(T) + E

(T + E) + E

(E) + T

(E) + T + E

int + T

(E) + T

T + int

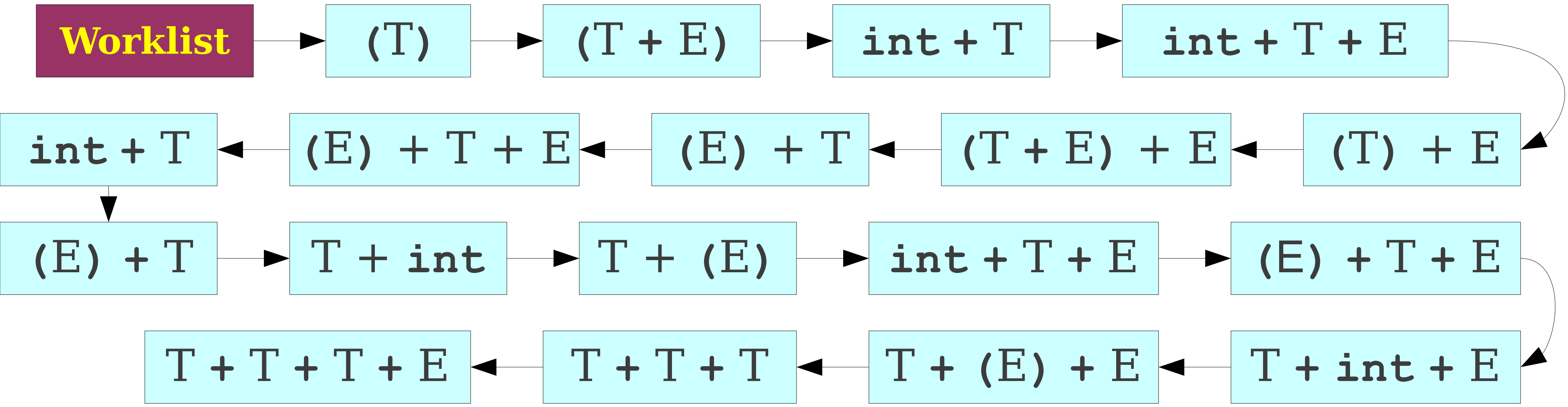
T + (E)

T + T + E

E → T
E → T + E
T → int
T → (E)

int + int

Breadth-First Search Parsing



Worklist

(T)

(T + E)

`int` + T

`int` + T + E

(T) + E

(T + E) + E

(E) + T

(E) + T + E

`int` + T

(E) + T

T + `int`

T + (E)

`int` + T + E

(E) + T + E

T + `int` + E

T + (E) + E

T + T + T

T + T + T + E

T + T + E

E → **T**

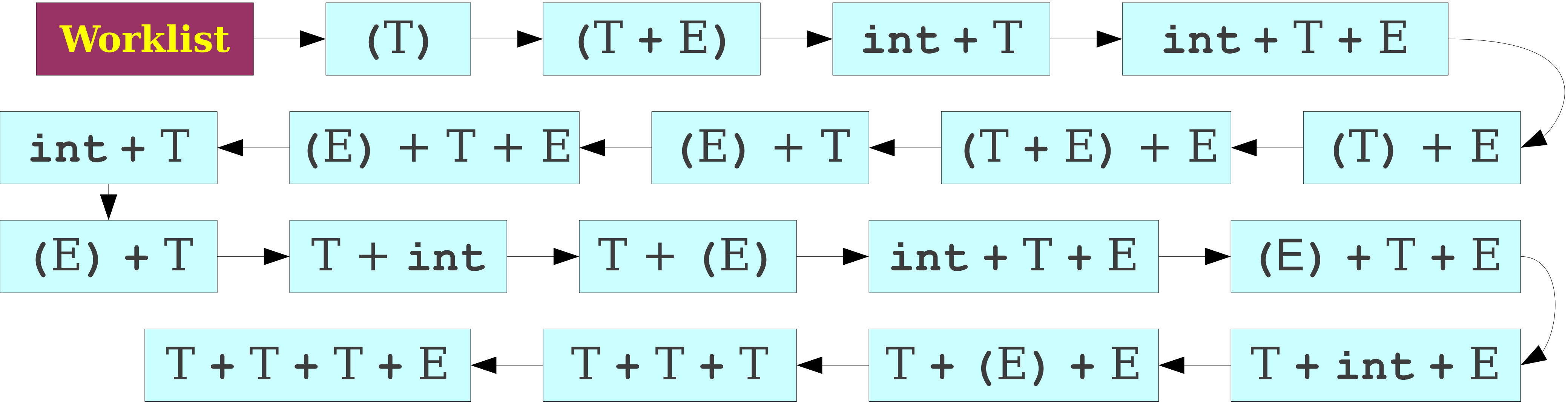
E → **T** **+** **E**

T → **`int`**

T → **(E)**

`int + int`

Breadth-First Search Parsing



Worklist

(T)

(T + E)

`int` + T

`int` + T + E

(T) + E

(T + E) + E

(E) + T

(E) + T + E

`int` + T

(E) + T

T + `int`

T + (E)

`int` + T + E

(E) + T + E

T + `int` + E

T + (E) + E

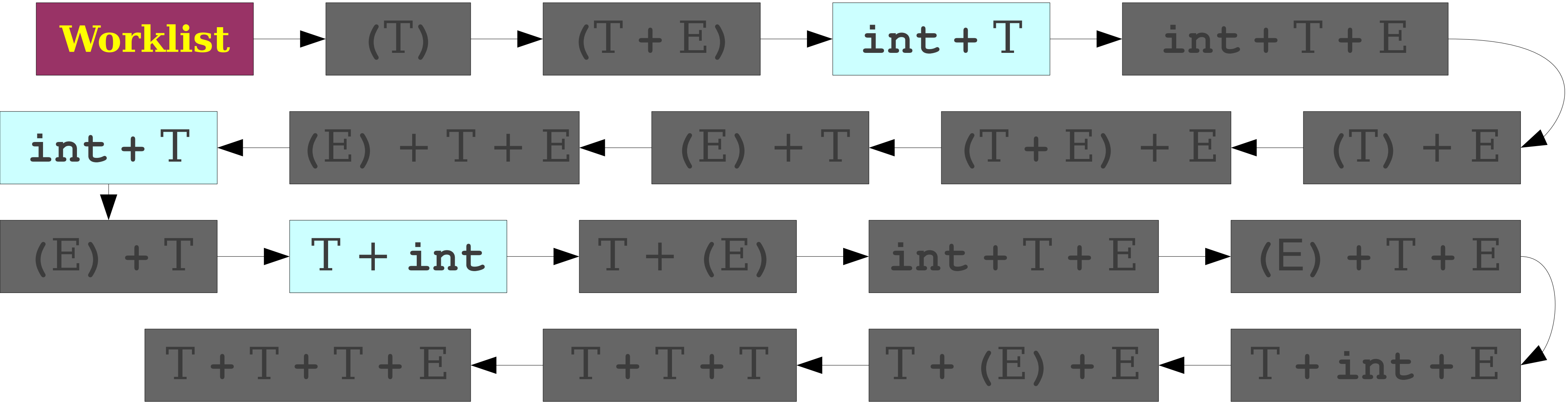
T + T + T

T + T + T + E

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

`int + int`

Breadth-First Search Parsing



Worklist

(T)

(T + E)

`int` + T

`int` + T + E

(T) + E

(T + E) + E

(E) + T

(E) + T + E

`int` + T

(E) + T

T + `int`

T + (E)

`int` + T + E

(E) + T + E

T + `int` + E

T + (E) + E

T + T + T

T + T + T + E

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.

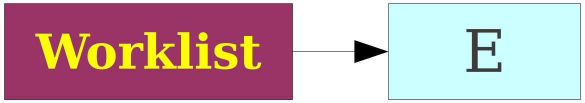
Reducing the Branching Factor

- 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



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

`int + int`

Leftmost BFS

Worklist

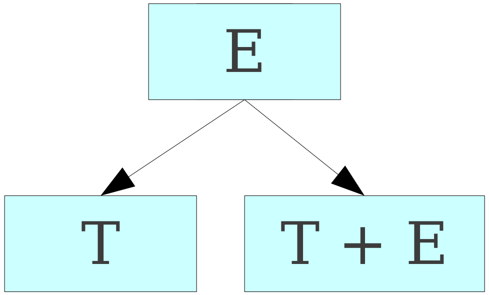
E

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

`int + int`

Leftmost BFS

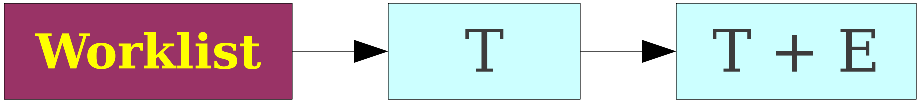
Worklist

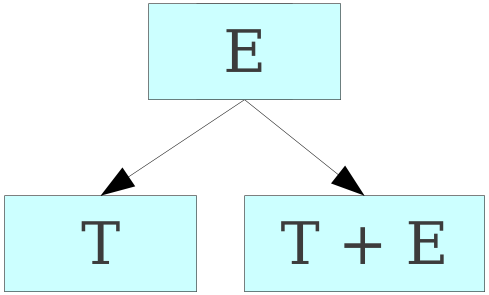


E → **T**
E → **T** **+** **E**
T → **`int`**
T → **(E)**

`int + int`

Leftmost BFS

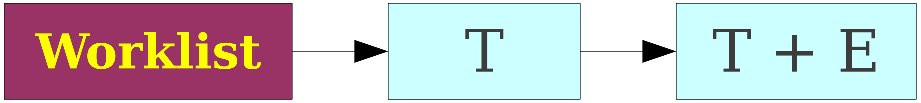




E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

`int + int`

Leftmost BFS



Worklist

T

T + E

E → T

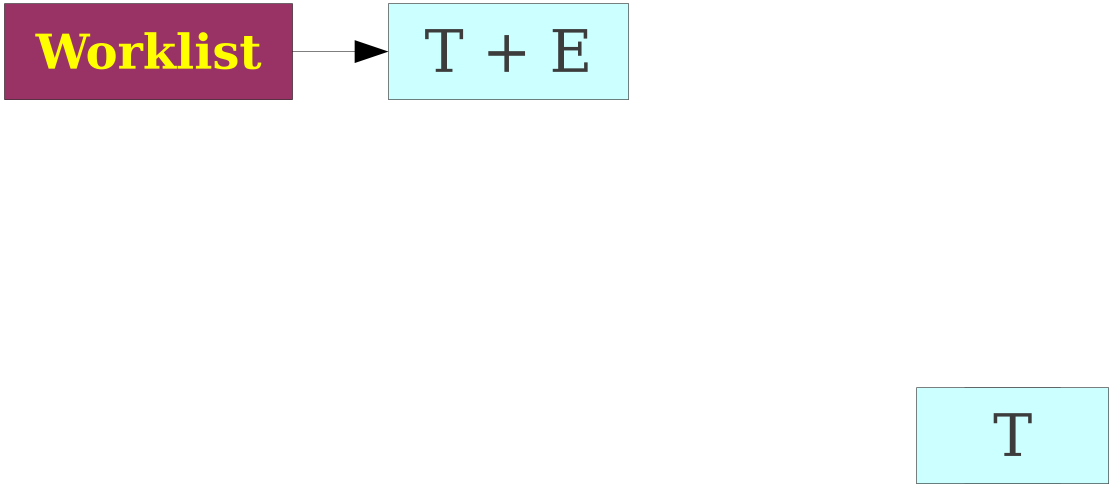
E → T `+` E

T → `int`

T → `(`E`)`

`int + int`

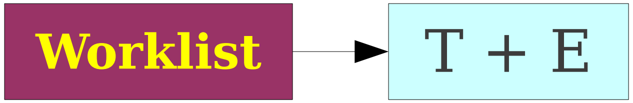
Leftmost BFS



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

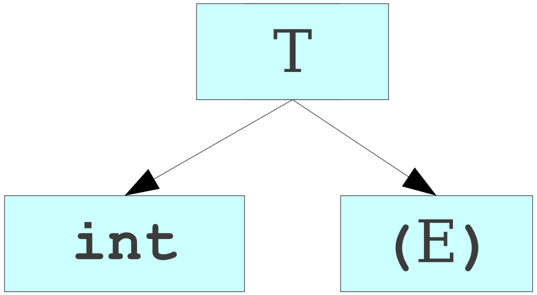
`int + int`

Leftmost BFS



Worklist

T + E



T

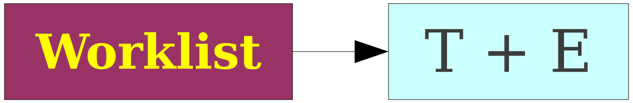
`int`

(E)

E → T
E → T `+` E
T → `int`
T → `(E)`

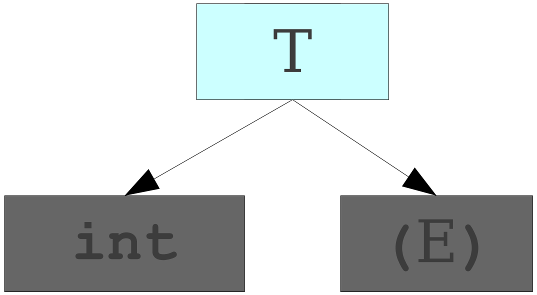
`int + int`

Leftmost BFS



Worklist

T + E



T

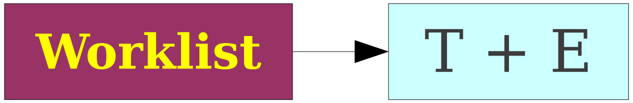
int

(E)

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

`int + int`

Leftmost BFS



Worklist → T + E

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

`int + int`

Leftmost BFS

Worklist

T + E

E → **T**

E → **T** **+** **E**

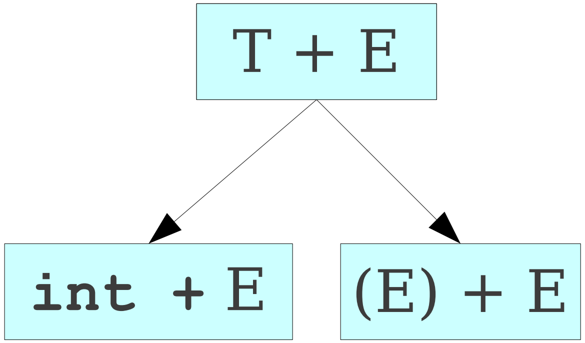
T → **int**

T → **(E)**

`int + int`

Leftmost BFS

Worklist

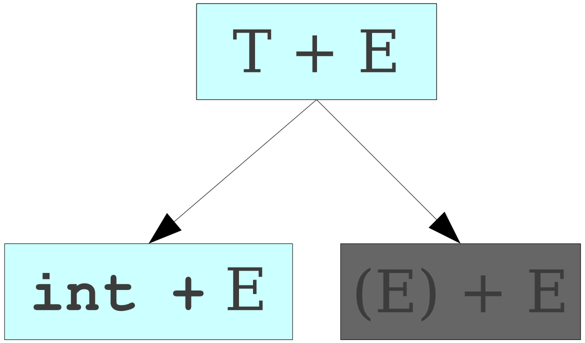


E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

`int + int`

Leftmost BFS

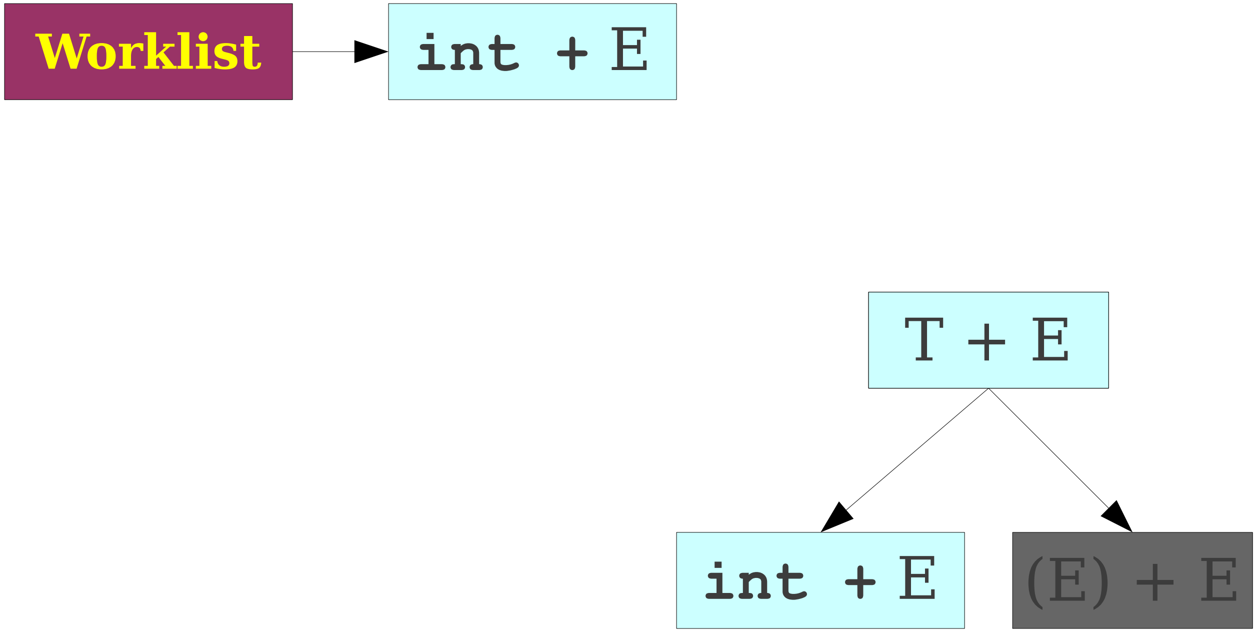
Worklist



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

`int + int`

Leftmost BFS



Worklist

`int` + E

T + E

`int` + E

(E) + E

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

`int + int`

Leftmost BFS

Worklist → `int` + E

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

`int + int`

Leftmost BFS

Worklist

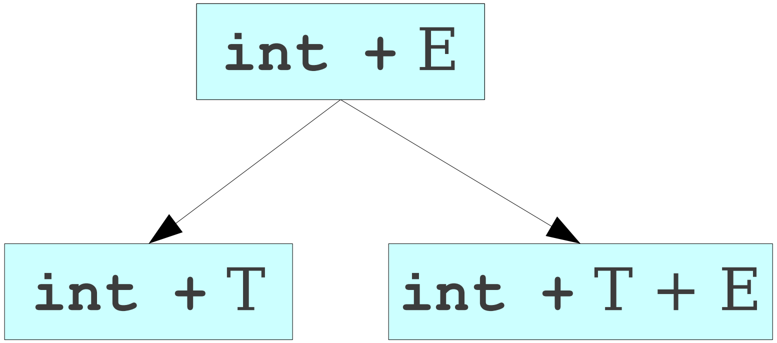
`int +` E

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

`int + int`

Leftmost BFS

Worklist



int + E

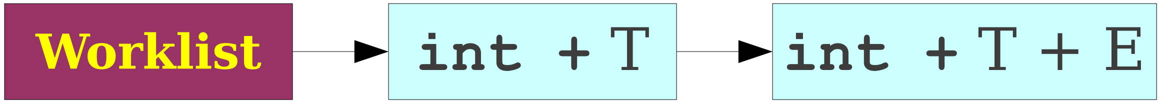
int + T

int + T + E

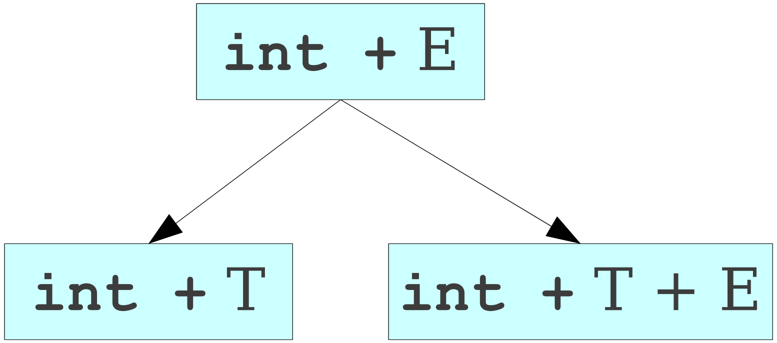
E → T
E → T + E
T → int
T → (E)

int + int

Leftmost BFS



Worklist → int + T → int + T + E



int + E

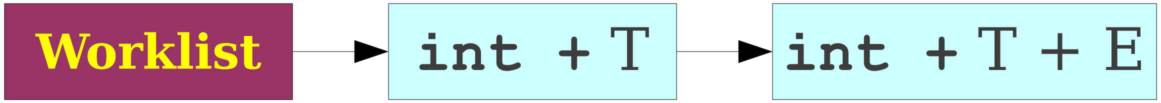
int + T

int + T + E

E → T
E → T + E
T → int
T → (E)

int + int

Leftmost BFS

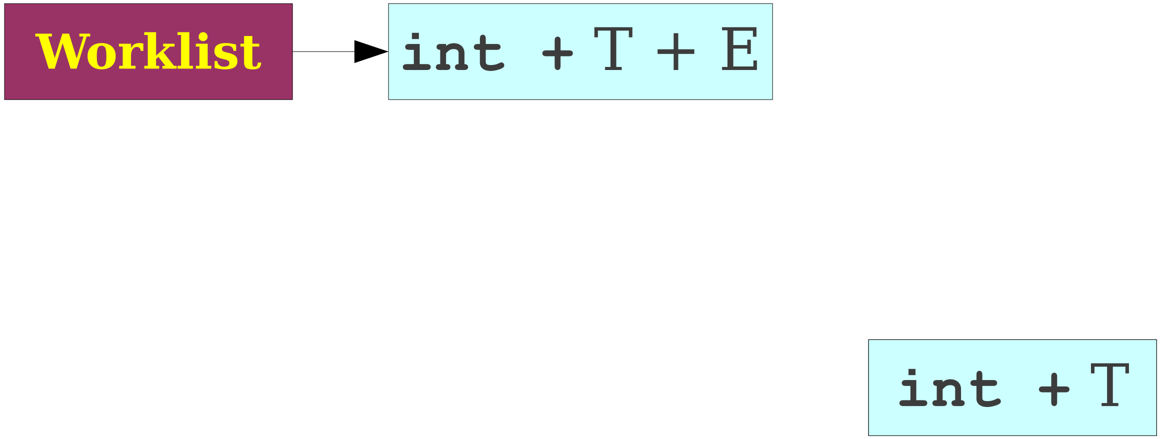


Worklist → `int` + T → `int` + T + E

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

`int + int`

Leftmost BFS



Worklist

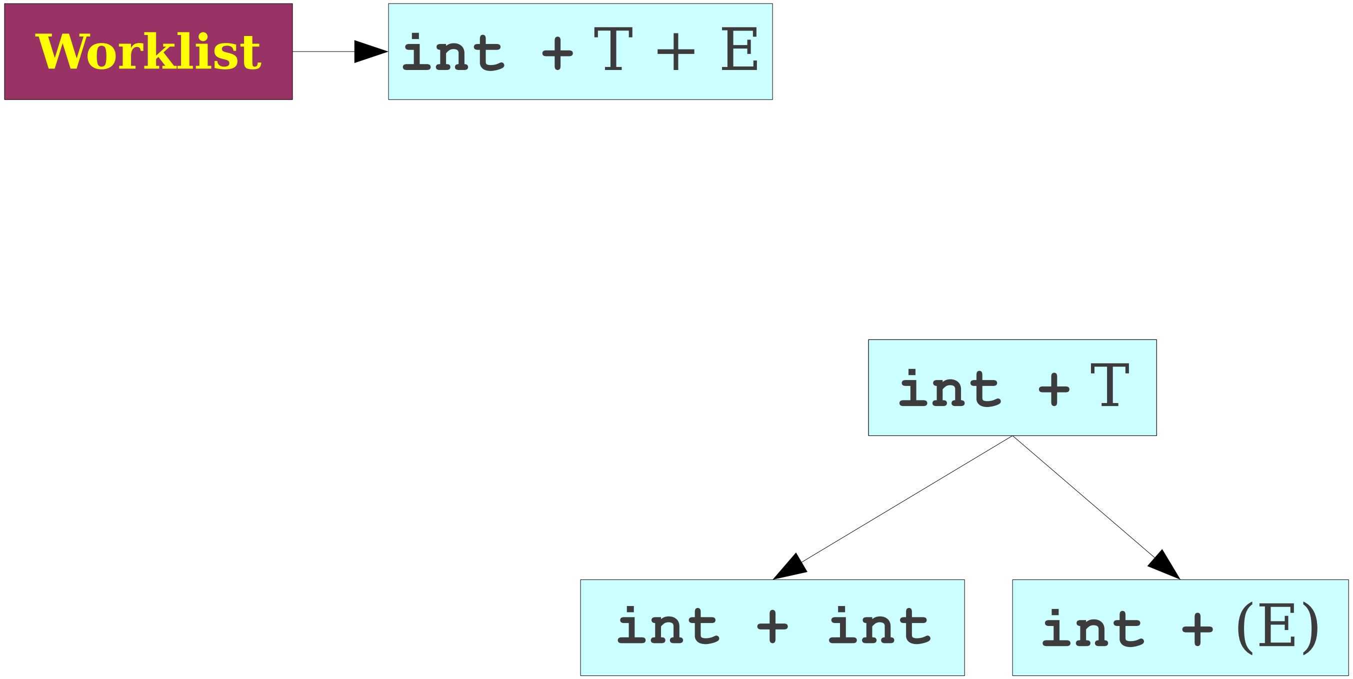
`int` + T + E

`int` + T

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

`int + int`

Leftmost BFS



Worklist

`int` + T + E

`int` + T

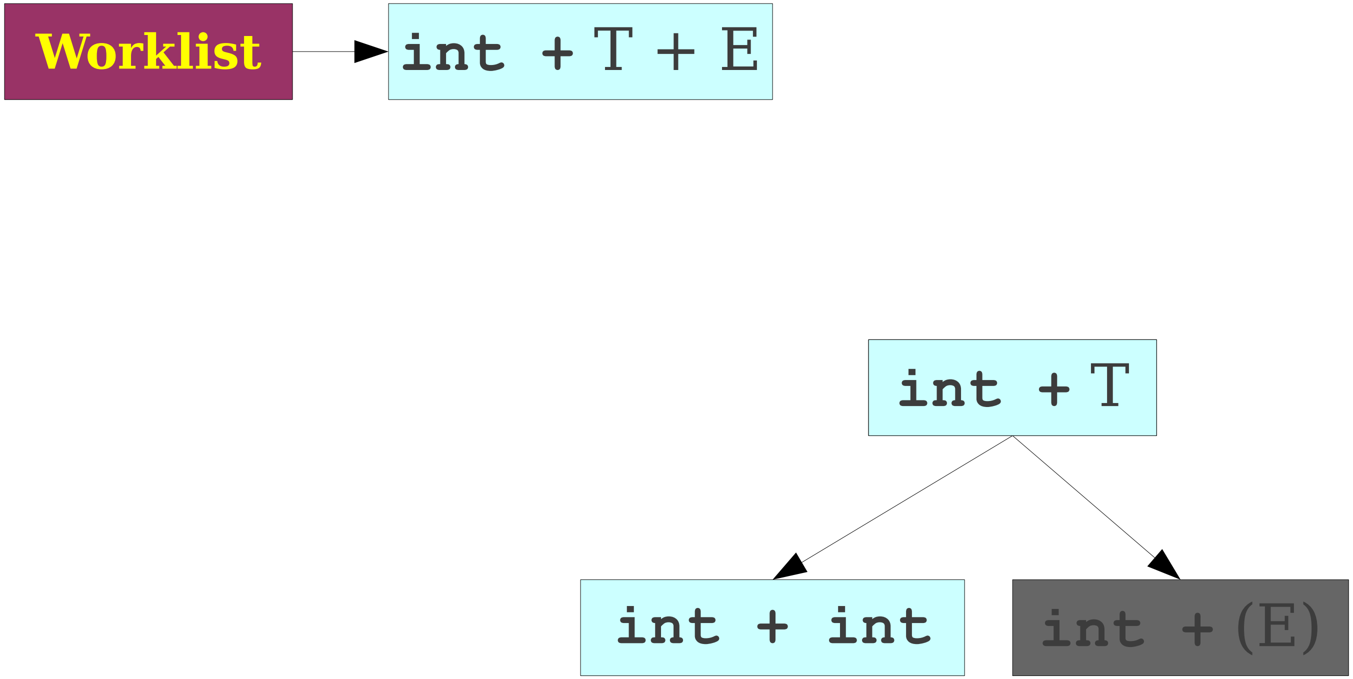
`int + int`

`int` + (E)

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

`int + int`

Leftmost BFS



Worklist

`int` + T + E

`int` + T

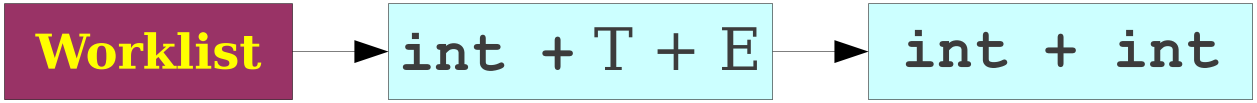
`int + int`

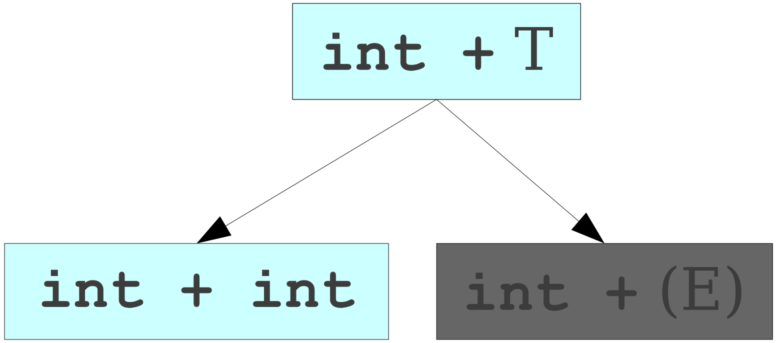
`int` + (E)

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

`int + int`

Leftmost BFS





E → T
E → T + E
T → int
T → (E)

int + int

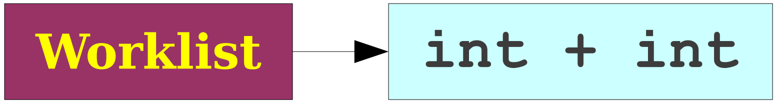
Leftmost BFS

Worklist → int + T + E → int + int

E → T
E → T + E
T → int
T → (E)

int + int

Leftmost BFS



Worklist

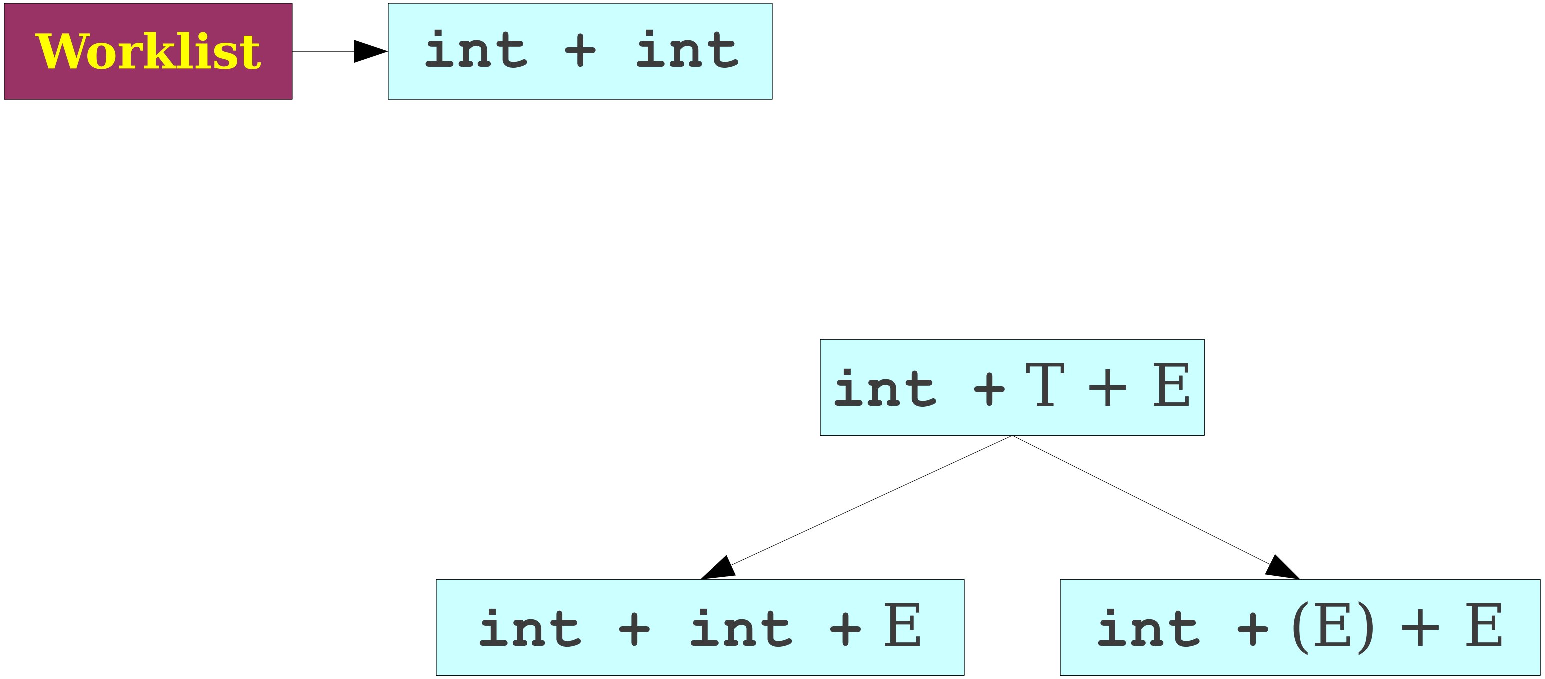
`int + int`

`int +` T + E

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

`int + int`

Leftmost BFS



E → T

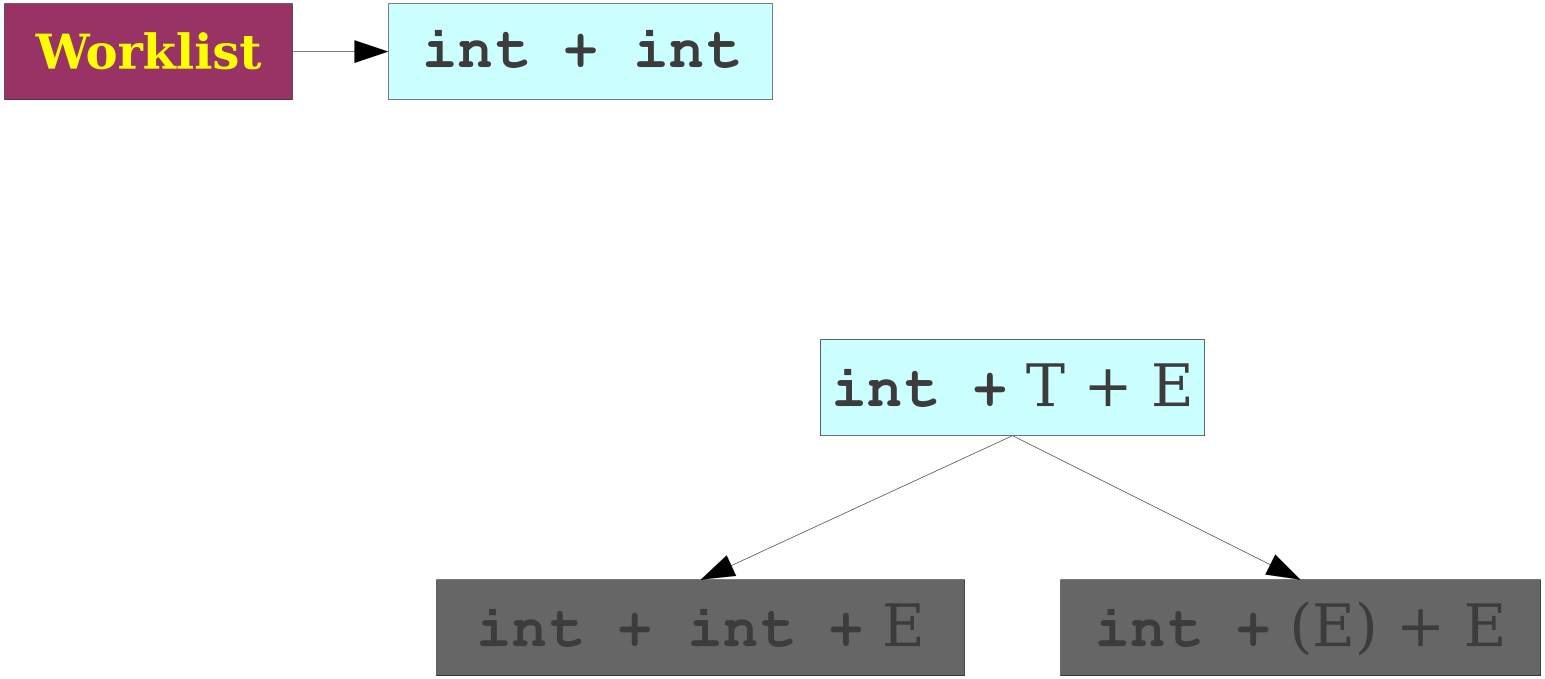
E → T + E

T → int

T → (E)

int + int

Leftmost BFS



Worklist

`int + int`

`int +` T + E

`int + int +` E

`int +` (E) + E

E → **T**

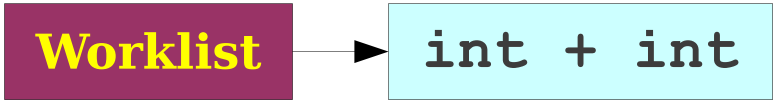
E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

`int + int`

Leftmost BFS



Worklist → `int + int`

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

`int + int`

Leftmost BFS

Worklist

`int + int`

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

`int + int`

Leftmost BFS

Worklist

`int + int`

E → T
E → T + E
T → int
T → (E)

`int + int`

Leftmost BFS

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

Leftmost BFS Has Problems

Worklist

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

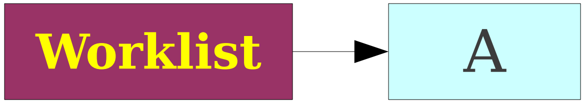
Leftmost BFS Has Problems

Worklist

A → **A**`a` | **A**`b` | `c`

`caaaaaaaaaa`

Leftmost BFS Has Problems



Worklist → A

A → Aa | Ab | c

`caaaaaaaaaa`

Leftmost BFS Has Problems

Worklist

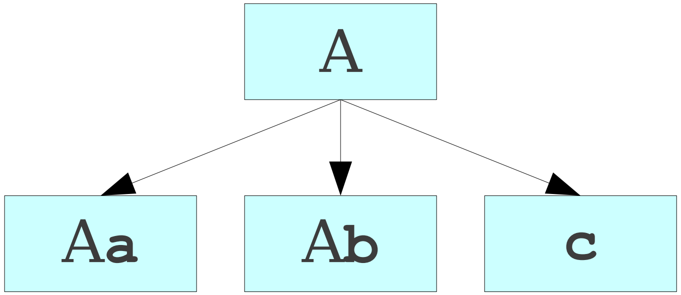
A

A → Aa | Ab | c

caaaaaaaaaa

Leftmost BFS Has Problems

Worklist

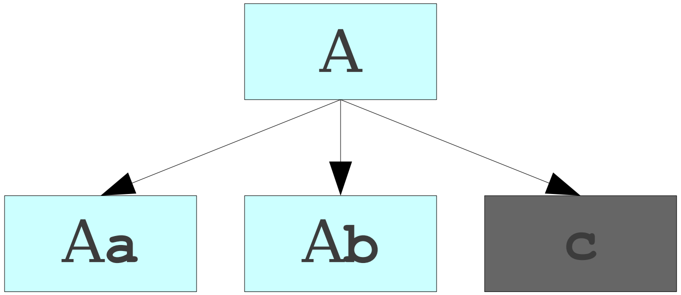


A → **A**`a` | **A**`b` | `c`

`caaaaaaaaaa`

Leftmost BFS Has Problems

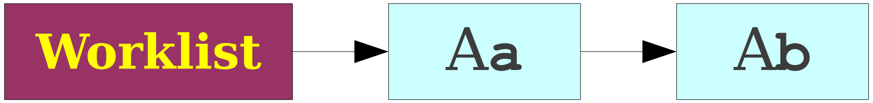
Worklist



A → **A**a | **A**b | c

caaaaaaaaaa

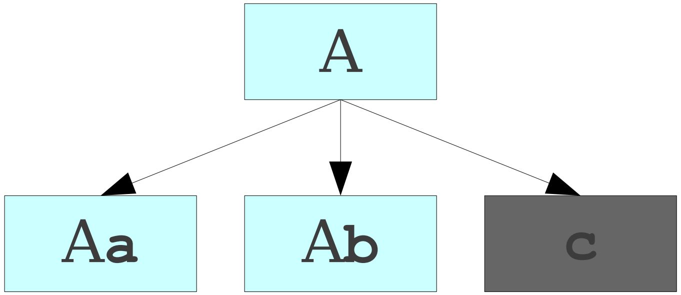
Leftmost BFS Has Problems



Worklist

Aa

Ab



A

Aa

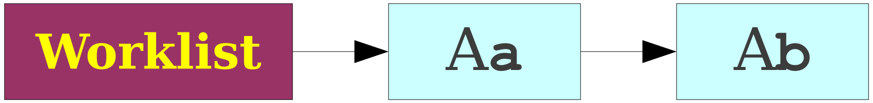
Ab

c

A → Aa | Ab | c

caaaaaaaaaa

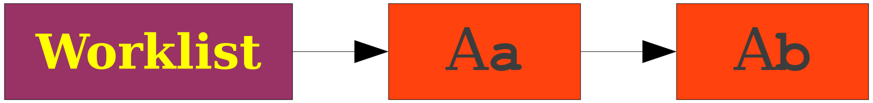
Leftmost BFS Has Problems



A → **A**a | **A**b | c

caaaaaaaaaa

Leftmost BFS Has Problems


Worklist

Aa

Ab

A → **A**a | **A**b | c

`caaaaaaaaaa`

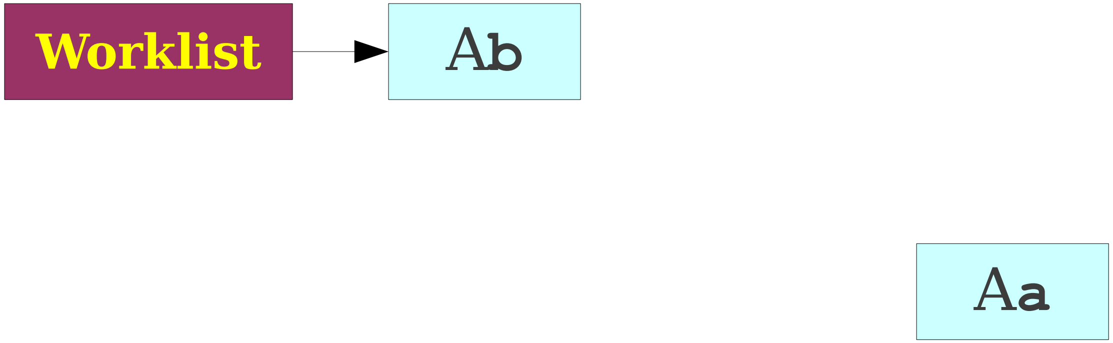
Leftmost BFS Has Problems

Worklist → Aa → Ab

A → **A**`a` | **A**`b` | `c`

`caaaaaaaaaa`

Leftmost BFS Has Problems



Worklist

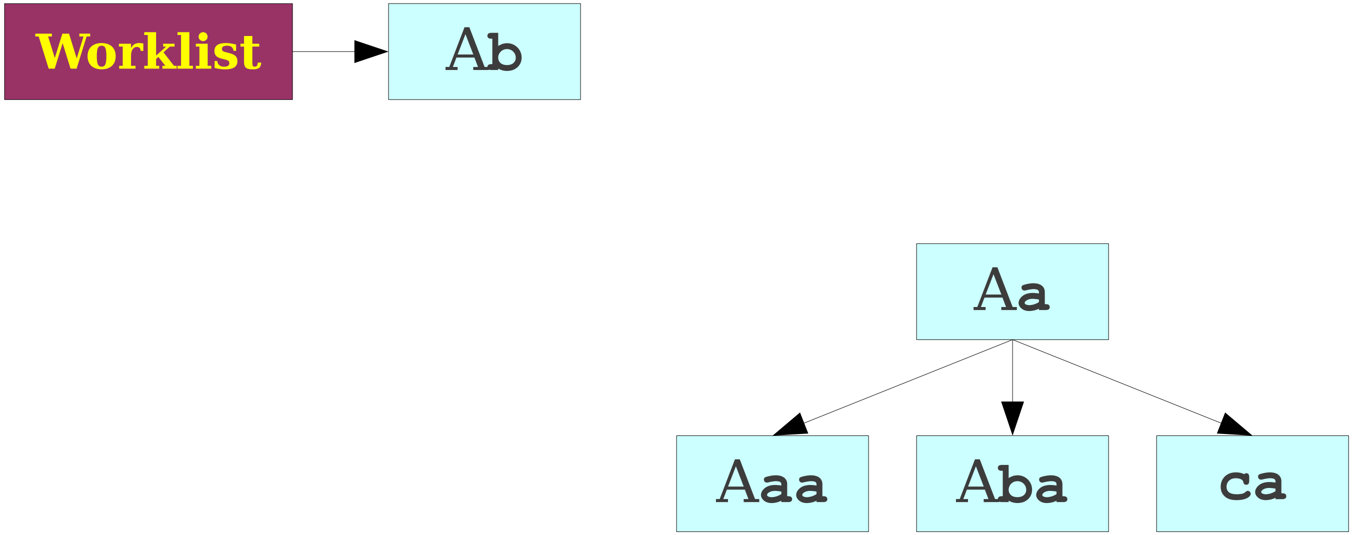
Ab

Aa

A → **Aa** | **Ab** | **c**

caaaaaaaaa

Leftmost BFS Has Problems



Worklist

Ab

Aa

Aaa

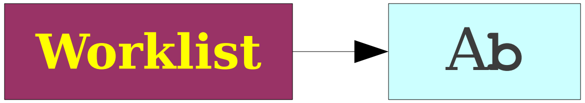
Aba

ca

A → **A**a | **A**b | c

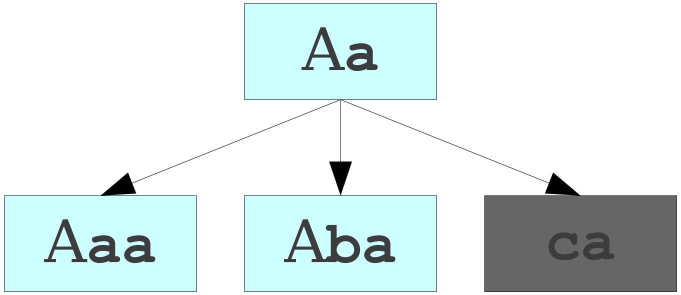
caaaaaaaaa

Leftmost BFS Has Problems



Worklist

Ab



Aa

Aaa

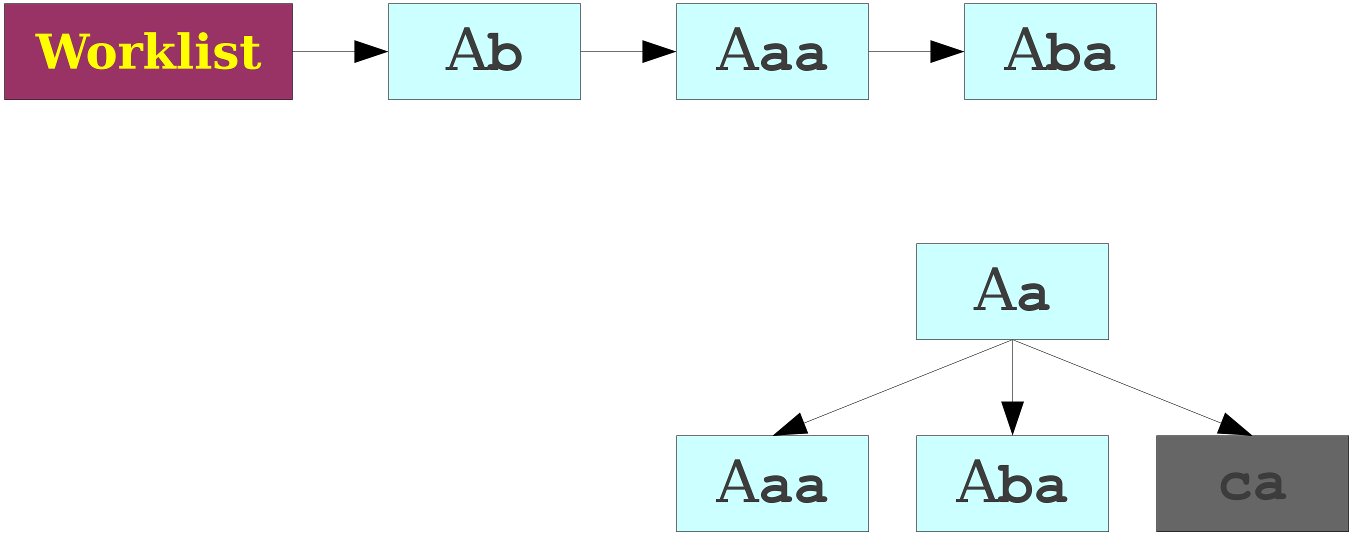
Aba

ca

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

`caaaaaaaaaa`

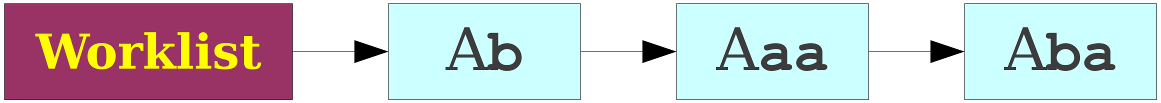
Leftmost BFS Has Problems



A → Aa | Ab | c

caaaaaaaaaa

Leftmost BFS Has Problems

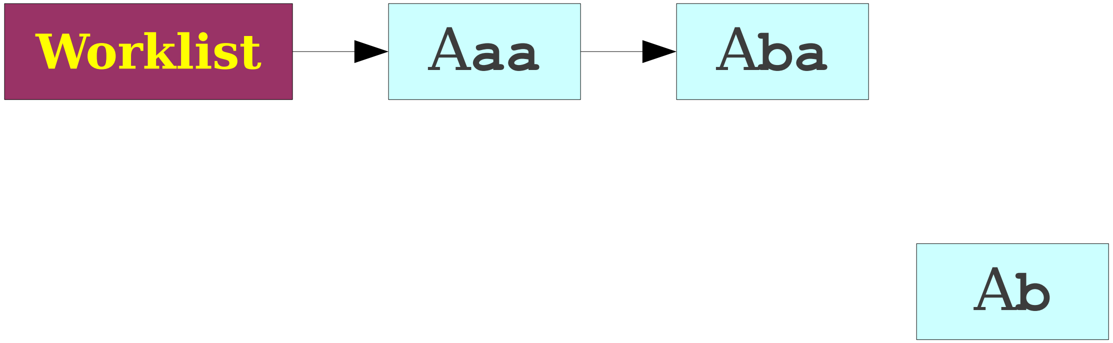


Worklist → Ab → Aaa → Aba

A → **A**a | **A**b | c

caaaaaaaaaa

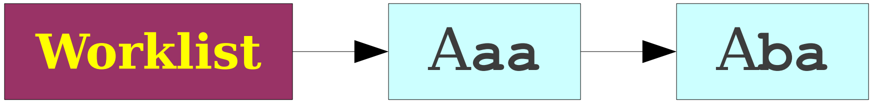
Leftmost BFS Has Problems



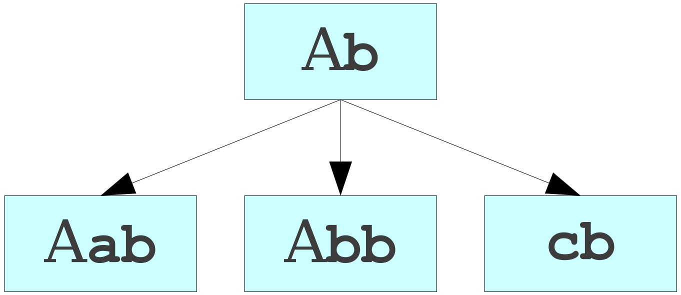
A → **A**a | **A**b | c

caaaaaaaaaa

Leftmost BFS Has Problems



Worklist → Aaa → Aba

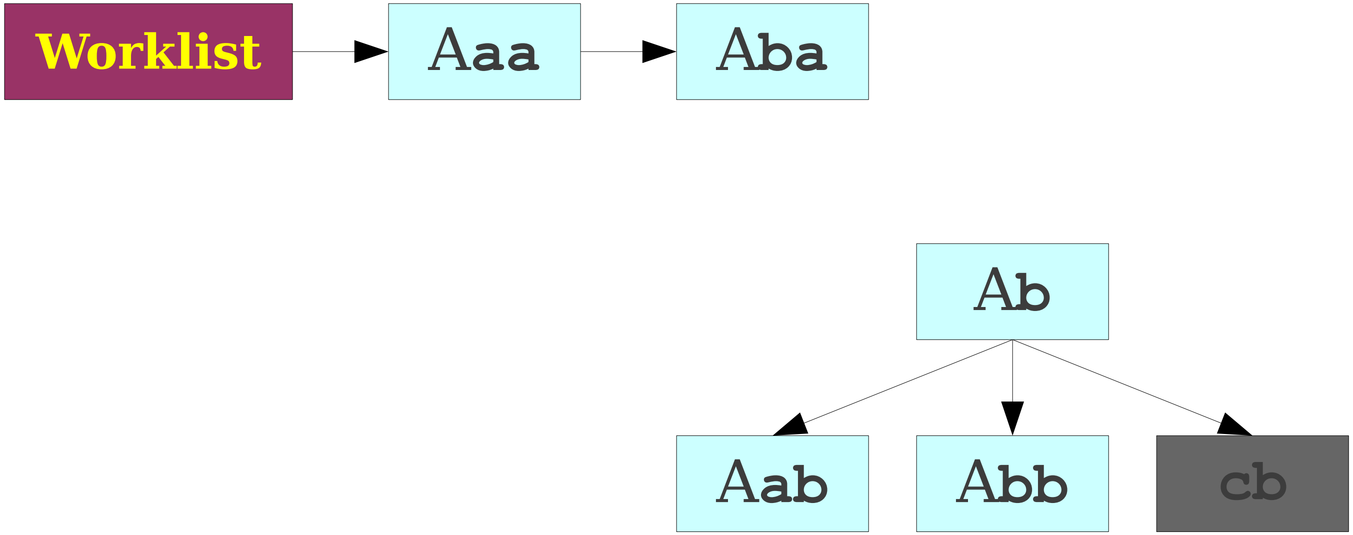


Ab → Aab, Abb, cb

A → **A**`a` | **A**`b` | `c`

`caaaaaaaaaa`

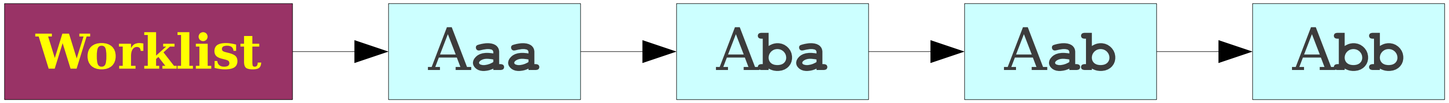
Leftmost BFS Has Problems

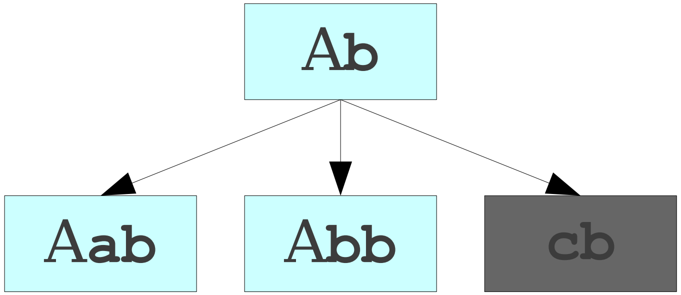


A → **A**a | **A**b | c

`caaaaaaaaaa`

Leftmost BFS Has Problems

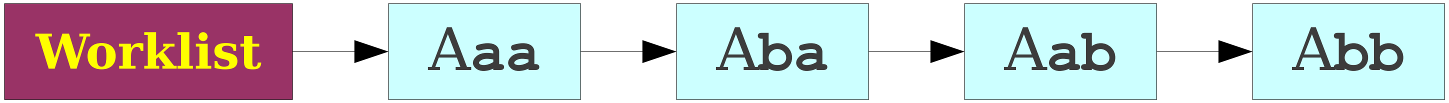




A → Aa | Ab | c

caaaaaaaaaa

Leftmost BFS Has Problems



Worklist

Aaa

Aba

Aab

Abb

A → **A**a | **A**b | c

caaaaaaaaaa

Leftmost BFS Has Problems

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

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

`caaaaaaaaaa`

Problems with Leftmost BFS

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

Leftmost DFS

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

Leftmost DFS

E → **T**

E → **T** `+` **E**

T → `int`

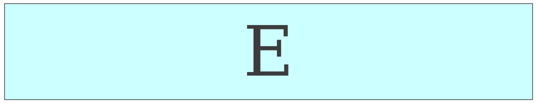
T → `(`**E**`)`

Leftmost DFS

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

`int + int`

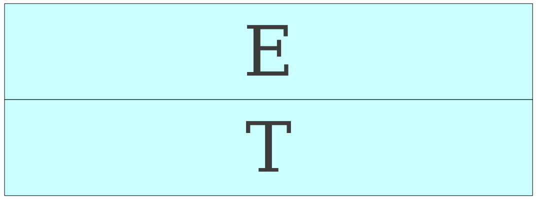
Leftmost DFS



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

`int + int`

Leftmost DFS



E

T

E → **T**

E → **T** `+` **E**

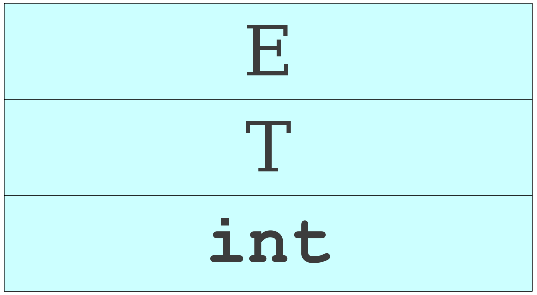
T → `int`

T → `(`**E**`)`

`int + int`

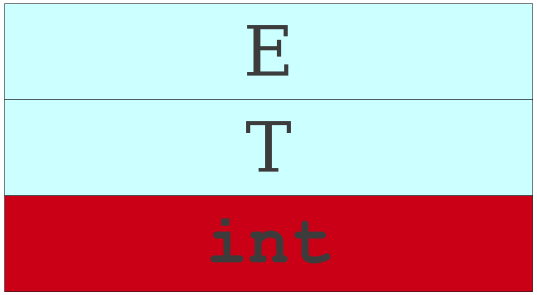
Leftmost DFS

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


E
T
`int`

`int + int`

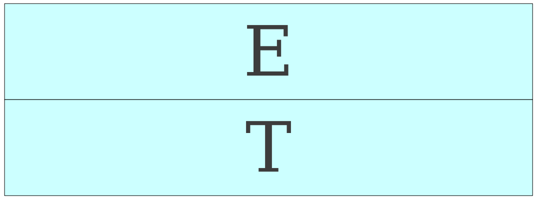
Leftmost DFS



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

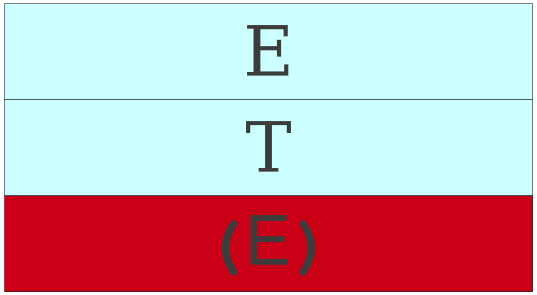
T

(E)

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

`int + int`

Leftmost DFS


E

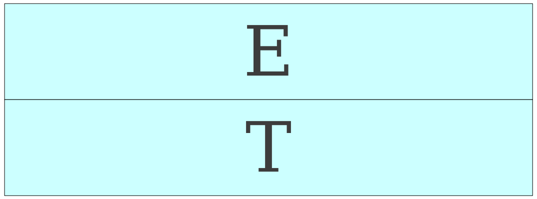
T

(E)

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

int + int

Leftmost DFS



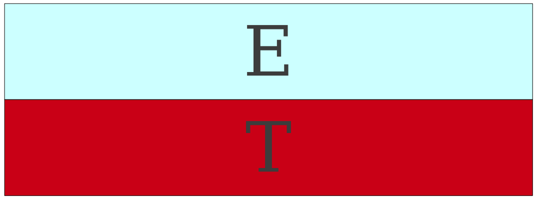
E

T

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

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

E
T + E
`int` + E

`int + int`

Leftmost DFS

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

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

`int + int`

Problems with Leftmost DFS

A → **A`a`** | `c`

A
Aa
Aaa
Aaaa
Aaaaa

`c`

Left Recursion

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

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

  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

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

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

Predictive Parsing

E

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

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

Predictive Parsing

E

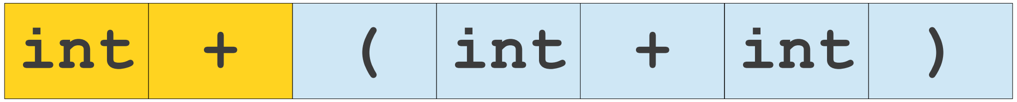
E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

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

Predictive Parsing

E
T + E

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



int	+	(	int	+	int	)

Predictive Parsing

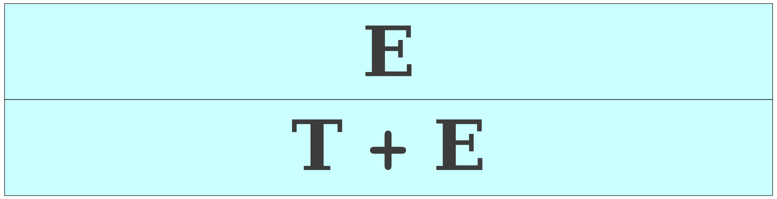
E

T + E

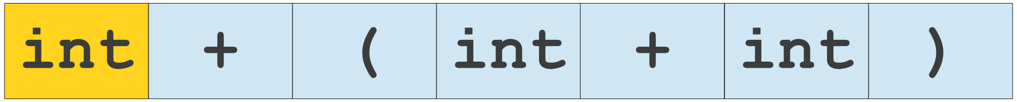
E → **T**
E → **T** `+` **E**
T → `int`
T → `(`**E**`)`

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

Predictive Parsing



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



Predictive Parsing

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

E → **T**

E → **T** `+` **E**

T → `int`

T → `(`**E**`)`

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

Predictive Parsing

E
T + E
int + E

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing

E
T + E
`int` + E

E → T
E → T + E
T → `int`
T → (E)

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

Predictive Parsing

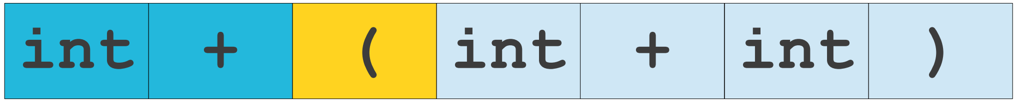
E
T + E
`int` + E

E → T

E → T + E

T → `int`

T → (E)

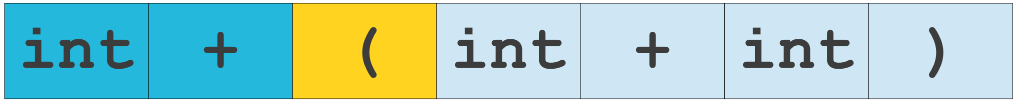


`int`	+	(	`int`	+	`int`	)

Predictive Parsing

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

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



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

Predictive Parsing

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

E → T
E → T + E
T → `int`
T → (E)

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

Predictive Parsing

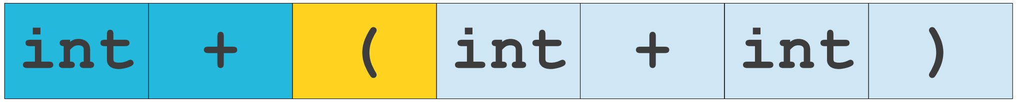
E

T + E

`int` + **E**

`int` + **T**

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

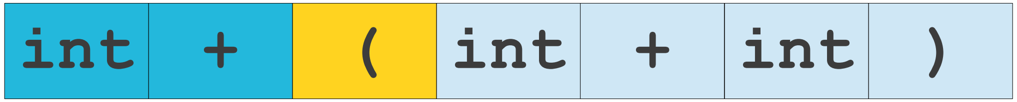


`int`	+	(	`int`	+	`int`	)

Predictive Parsing

E → T
E → T + E
T → int
T → (E)

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

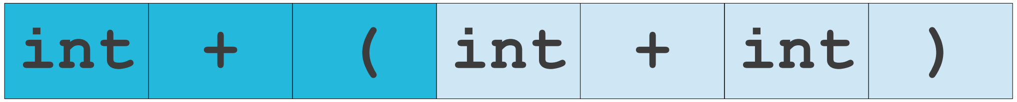
`int` + (E)

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)**

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)

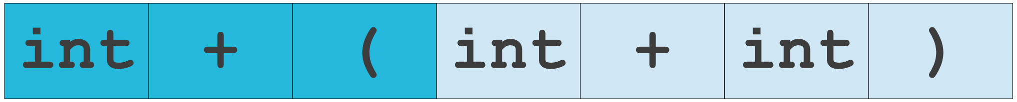
`int` + (T + E)

E → T

E → T + E

T → `int`

T → (E)



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 → T + E
T → int
T → (E)

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

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	+	int	)

Predictive Parsing

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

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

`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`	`)`

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`	`)`

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)

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 → 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	)

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	)

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** → **`int`**

(2) **E** → **`(`E Op E`)`**

(3) **Op** → **`+`**

(4) **Op** → **`*`**

LL(1) Parsing

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

(1) **E** → `int`

(2) **E** → `(`**E Op E**`)`

(3) **Op** → `+`

(4) **Op** → `*`

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

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

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

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

LL(1) Parsing

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

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

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

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

This is called a **predict** step.

LL(1) Parsing

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

(1) **E** → `int`

(2) **E** → `(`**E Op E**`)`

(3) **Op** → `+`

(4) **Op** → `*`

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

LL(1) Parsing

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

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

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

LL(1) Parsing

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

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

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

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

This is called a **match** step.

LL(1) Parsing

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

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

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

LL(1) Parsing

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

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

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

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

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

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

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

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) 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 → `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	(	)	+	*
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

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

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

TERM   →  id
       |  constant
```

A Simple LL(1) Grammar

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

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

TERM  →  id
      |  constant
```

```
id -> id;
```

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

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** (1)
| `while` **EXPR** `do` **STMT** (2)
| **EXPR** `;` (3)

EXPR → **TERM** `->` `id` (4)
| `zero?` **TERM** (5)
| `not` **EXPR** (6)
| `++` `id` (7)
| `--` `id` (8)

TERM → `id` (9)
| `constant` (10)

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** **(1)**
| `while` **EXPR** `do` **STMT** **(2)**
| **EXPR** `;` **(3)**

EXPR → **TERM** `->` `id` **(4)**
| `zero?` **TERM** **(5)**
| `not` **EXPR** **(6)**
| `++` `id` **(7)**
| `--` `id` **(8)**

TERM → `id` **(9)**
| `constant` **(10)**

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

Constructing LL(1) Parse Tables

STMT → if EXPR then STMT (1)
| while EXPR do STMT (2)
| EXPR ; (3)

EXPR → TERM -> id (4)
| zero? TERM (5)
| not EXPR (6)
| ++ id (7)
| -- id (8)

TERM → id (9)
| constant (10)

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** **(1)**
| `while` **EXPR** `do` **STMT** **(2)**
| **EXPR** `;` **(3)**

EXPR → **TERM** `->` `id` **(4)**
| `zero?` **TERM** **(5)**
| `not` **EXPR** **(6)**
| `++` `id` **(7)**
| `--` `id` **(8)**

TERM → `id` **(9)**
| `constant` **(10)**

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** **(1)**
| `while` **EXPR** `do` **STMT** **(2)**
| **EXPR** `;` **(3)**

EXPR → **TERM** `->` `id` **(4)**
| `zero?` **TERM** **(5)**
| `not` **EXPR** **(6)**
| `++` `id` **(7)**
| `--` `id` **(8)**

TERM → `id` **(9)**
| `constant` **(10)**

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

Constructing LL(1) Parse Tables

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

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** **(1)**
| `while` **EXPR** `do` **STMT** **(2)**
| **EXPR** `;` **(3)**

EXPR → **TERM** `->` `id` **(4)**
| `zero?` **TERM** **(5)**
| `not` **EXPR** **(6)**
| `++` `id` **(7)**
| `--` `id` **(8)**

TERM → `id` **(9)**
| `constant` **(10)**

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** **(1)**
| `while` **EXPR** `do` **STMT** **(2)**
| **EXPR** `;` **(3)**

EXPR → **TERM** `->` `id` **(4)**
| `zero?` **TERM** **(5)**
| `not` **EXPR** **(6)**
| `++` `id` **(7)**
| `--` `id` **(8)**

TERM → `id` **(9)**
| `constant` **(10)**

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** **(1)**
| `while` **EXPR** `do` **STMT** **(2)**
| **EXPR** `;` **(3)**

EXPR → **TERM** `->` `id` **(4)**
| `zero?` **TERM** **(5)**
| `not` **EXPR** **(6)**
| `++` `id` **(7)**
| `--` `id` **(8)**

TERM → `id` **(9)**
| `constant` **(10)**

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** **(1)**
| `while` **EXPR** `do` **STMT** **(2)**
| **EXPR** `;` **(3)**

EXPR → **TERM** `->` `id` **(4)**
| `zero?` **TERM** **(5)**
| `not` **EXPR** **(6)**
| `++` `id` **(7)**
| `--` `id` **(8)**

TERM → `id` **(9)**
| `constant` **(10)**

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** (1)
| `while` **EXPR** `do` **STMT** (2)
| **EXPR** `;` (3)

EXPR → **TERM** `->` `id` (4)
| `zero?` **TERM** (5)
| `not` **EXPR** (6)
| `++` `id` (7)
| `--` `id` (8)

TERM → `id` (9)
| `constant` (10)

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** **(1)**
| `while` **EXPR** `do` **STMT** **(2)**
| **EXPR** `;` **(3)**

EXPR → **TERM** `->` `id` **(4)**
| `zero?` **TERM** **(5)**
| `not` **EXPR** **(6)**
| `++` `id` **(7)**
| `--` `id` **(8)**

TERM → `id` **(9)**
| `constant` **(10)**

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** **(1)**
| `while` **EXPR** `do` **STMT** **(2)**
| **EXPR** `;` **(3)**

EXPR → **TERM** `->` `id` **(4)**
| `zero?` **TERM** **(5)**
| `not` **EXPR** **(6)**
| `++` `id` **(7)**
| `--` `id` **(8)**

TERM → `id` **(9)**
| `constant` **(10)**

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

Constructing LL(1) Parse Tables

STMT → `if` **EXPR** `then` **STMT** **(1)**
| `while` **EXPR** `do` **STMT** **(2)**
| **EXPR** `;` **(3)**

EXPR → **TERM** `->` `id` **(4)**
| `zero?` **TERM** **(5)**
| `not` **EXPR** **(6)**
| `++` `id` **(7)**
| `--` `id` **(8)**

TERM → `id` **(9)**
| `constant` **(10)**

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

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.
- $T[\mathbf{A}, \mathtt{t}]$ should be a production $\mathbf{A} \to \boldsymbol{\omega}$ iff $\boldsymbol{\omega}$ derives something starting with $\mathtt{t}$.
- 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

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

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

$$\text{FIRST}(\mathbf{A}) = \text{FIRST}(\mathbf{A}) \cup \text{FIRST}(\mathbf{B})$$

- 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`	`zero?`	
`while`	`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`	`zero?`	`id`
`while`	`not`	`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 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	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 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												
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 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		

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 nstant	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 **A** → `t`***ω***, set T[**A**, `t`] = `t`***ω***.
- For each production **A** → **B*ω***, set T[**A**, `t`] = **B*ω*** for each `t` ∈ FIRST(**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	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
+ -	+ - 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 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 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
0 5	ε	1 6	1 6	1 6
1 6		2 7	2 7	2 7
2 7		3 8	3 8	3 8
3 8		4 9	4 9	4 9
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 $\texttt{t} \in \text{FIRST}(\mathbf{A})$.

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***, 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*(ω) as follows:
  - FIRST*(ε) = { ε }
  - FIRST*($\mathtt{t}\omega$) = { $\mathtt{t}$ }
  - If $\varepsilon \notin$ FIRST($\mathbf{A}$):
    - FIRST*($\mathbf{A}\omega$) = FIRST($\mathbf{A}$)
  - If $\varepsilon \in$ FIRST($\mathbf{A}$):
    - FIRST*($\mathbf{A}\omega$) = (FIRST($\mathbf{A}$) - { ε }) $\cup$ FIRST*(ω)

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!`

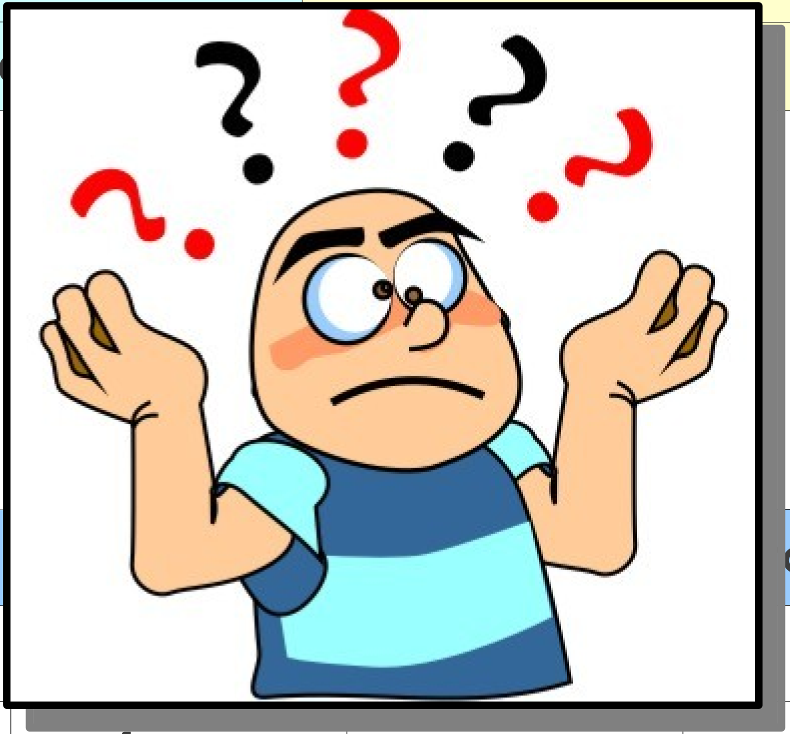
LL(1) Tables with ε

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.