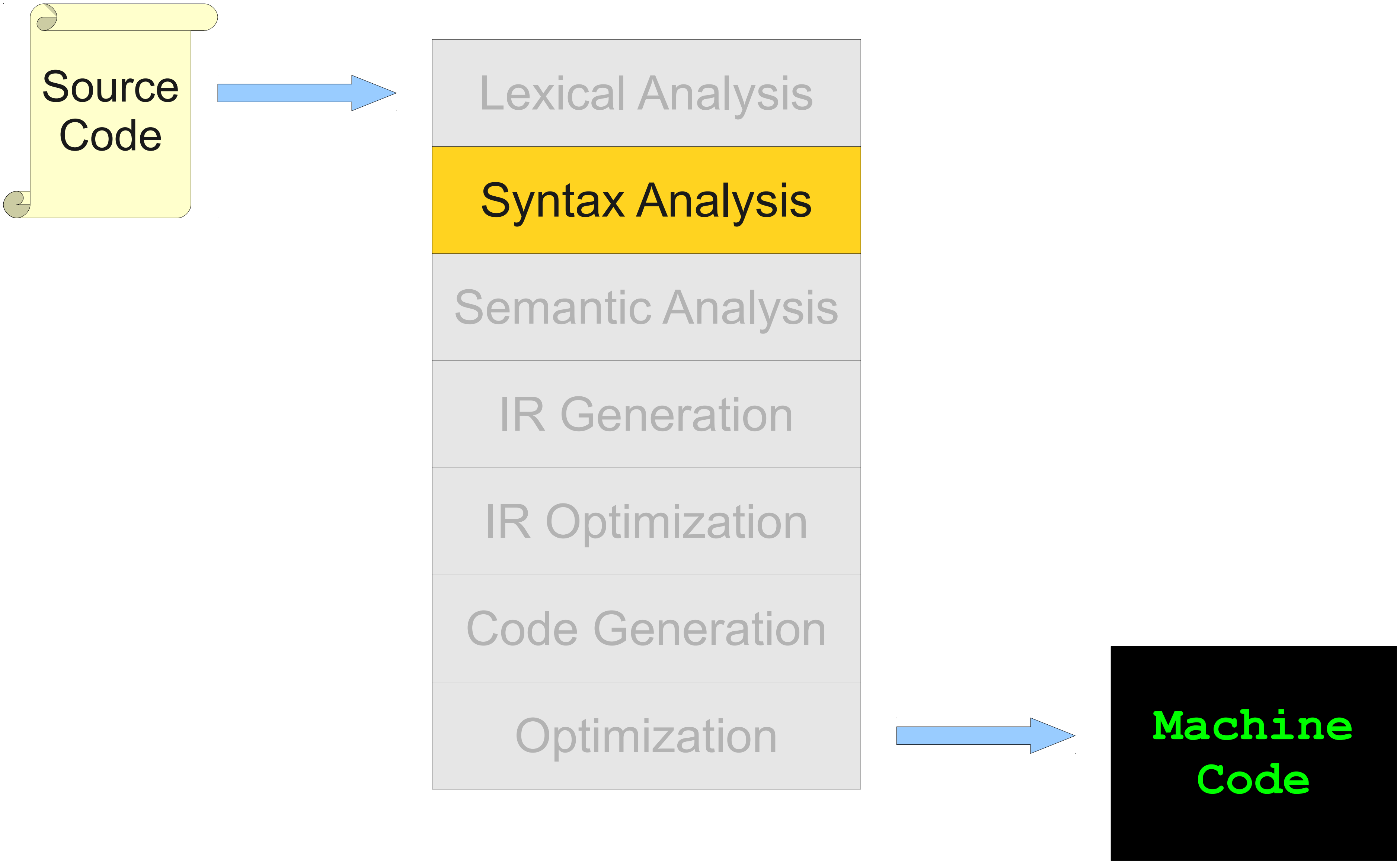
Top-Down Parsing

Announcements

- Office hours schedule posted on website.
  - Keith:
    - Mon 1:00PM – 3:00PM, Gates 160
    - Wed 1:00PM – 3:00PM, Gates 160
  - Hrysoula:
    - Thursday 7:00PM – 9:00PM, Gates 24A
    - Friday 1:00PM – 3:00PM, Gates 24A
  - Riddhi
    - Tuesday 7:00PM – 9:00PM, Gates 24A
    - Sunday 12:00PM – 2:00PM, Gates 24A
- Or ask questions on Piazzza: www.piazzza.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)
  - Beginning with the start symbol, try to guess the productions to apply to end up at the user's program.
- **Bottom-Up Parsing** (Wednesday / Friday)
  - Beginning with the user's program, try to apply productions in reverse to convert the program back into the start symbol.

Top-Down Parsing

$E \rightarrow T$

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

$T \rightarrow$ **int**

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

Top-Down Parsing

$E \to T$

$E \to T$ **+** E

$T \to$ **int**

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

int	**+**	**(**	**int**	**+**	**int**	**)**

Top-Down Parsing

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

int	**+**	**(**	**int**	**+**	**int**	**)**

Top-Down Parsing

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

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

Top-Down Parsing

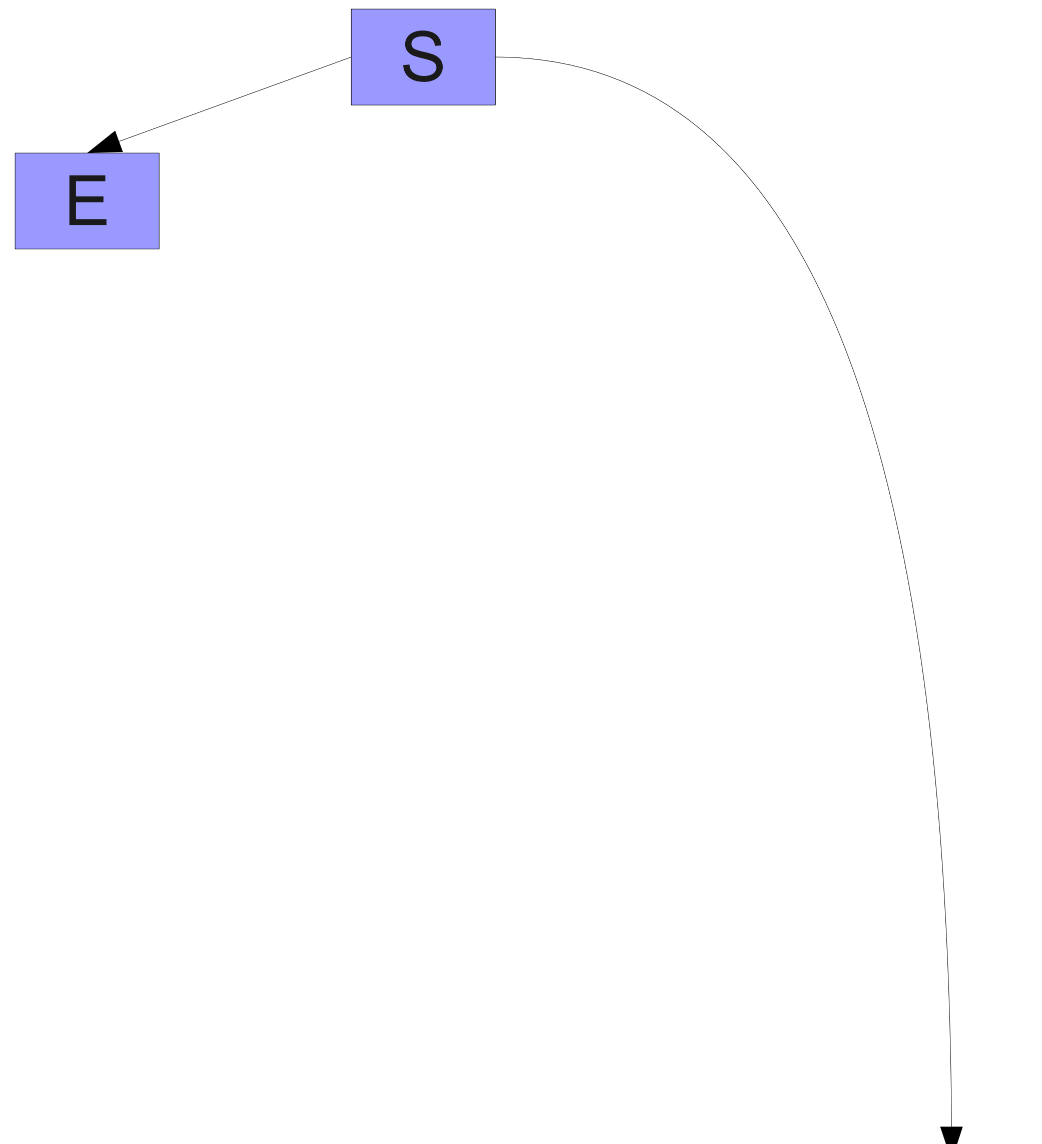
$S \rightarrow E\$$
$E \rightarrow T$
$E \rightarrow T$ **+** E
$T \rightarrow$ **int**
$T \rightarrow$ **(**E**)**

S

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

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

Top-Down Parsing



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

Top-Down Parsing

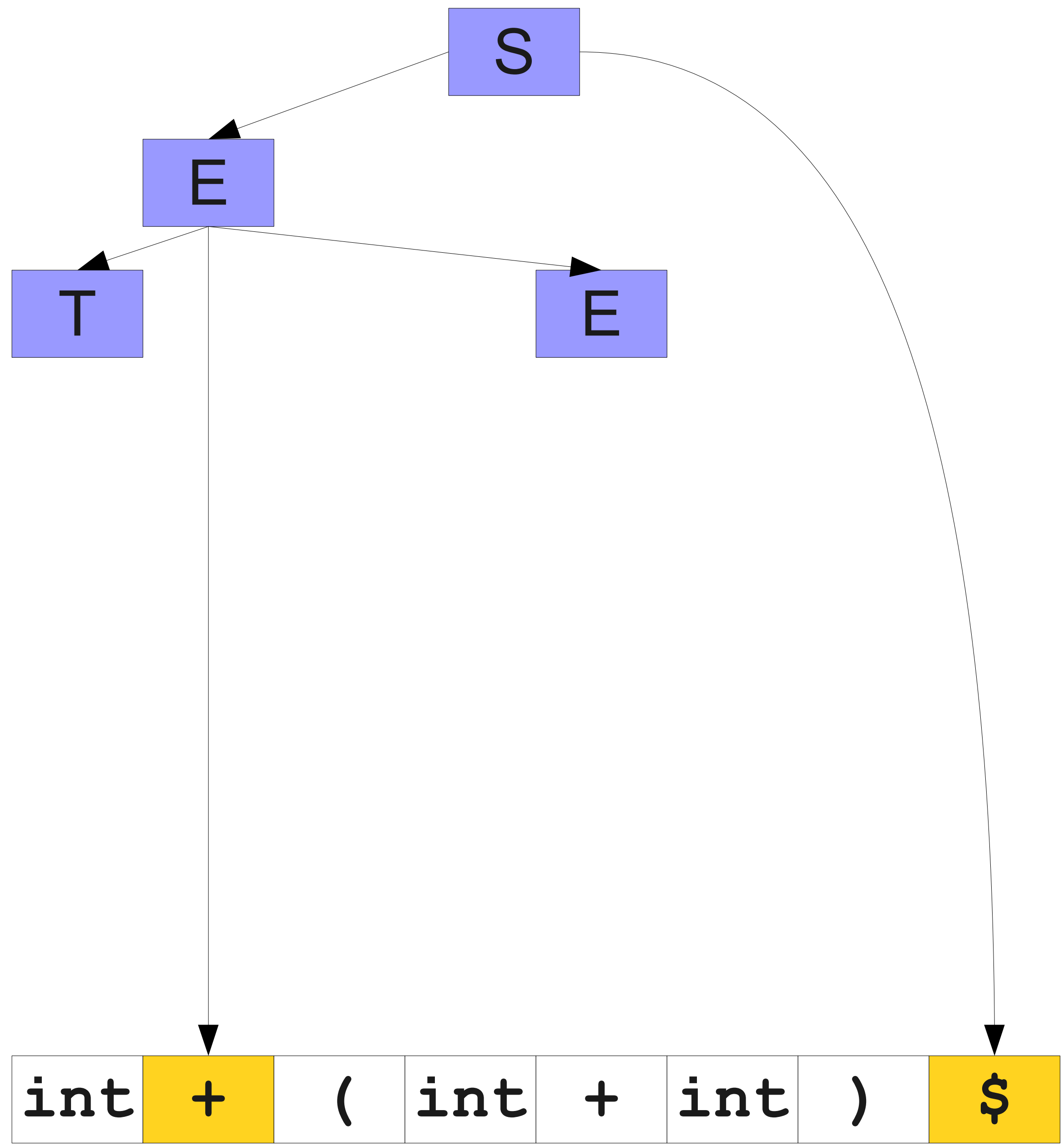
S → E$

E → T

E → T + E

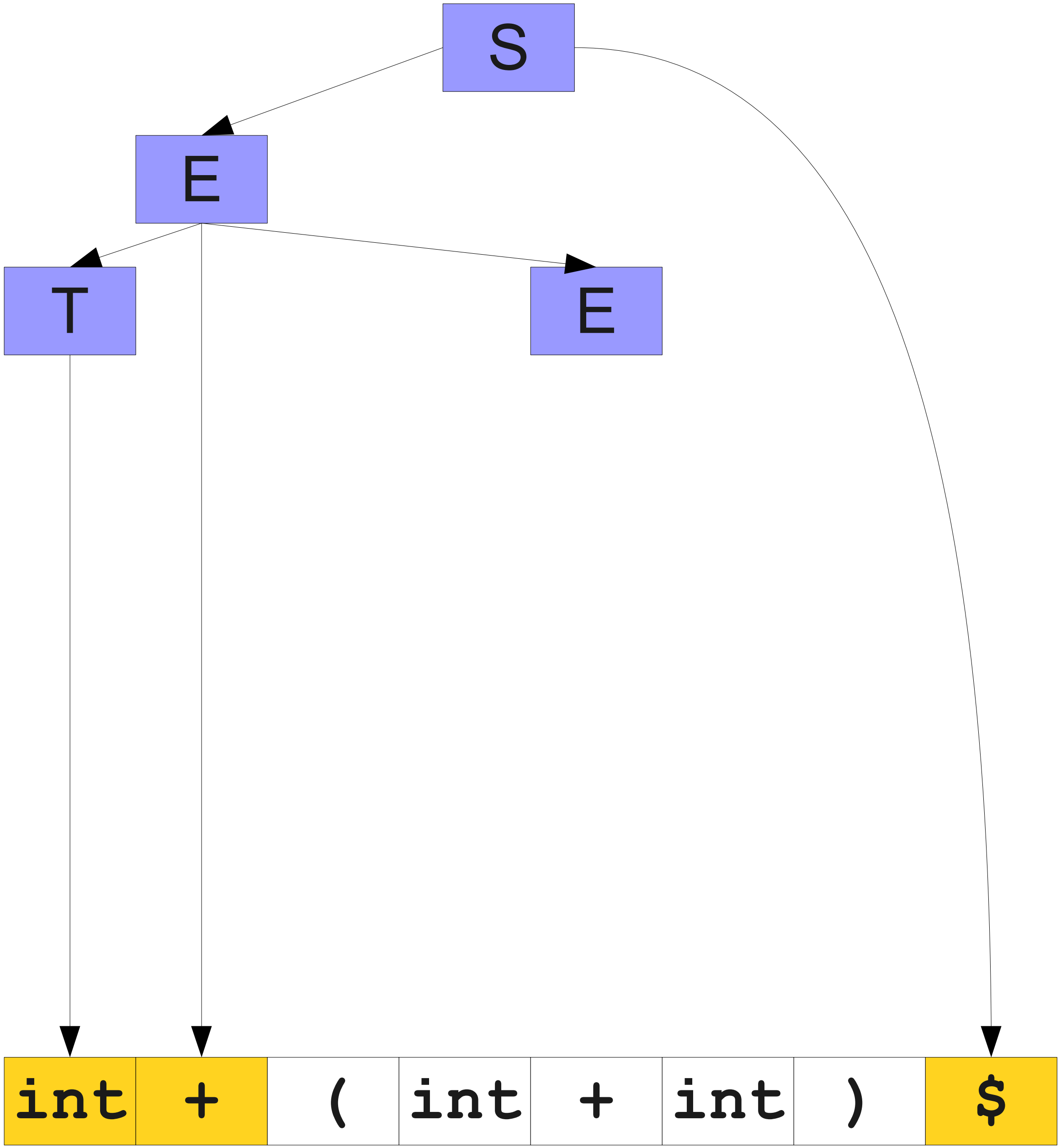
T → **int**

T → (E)



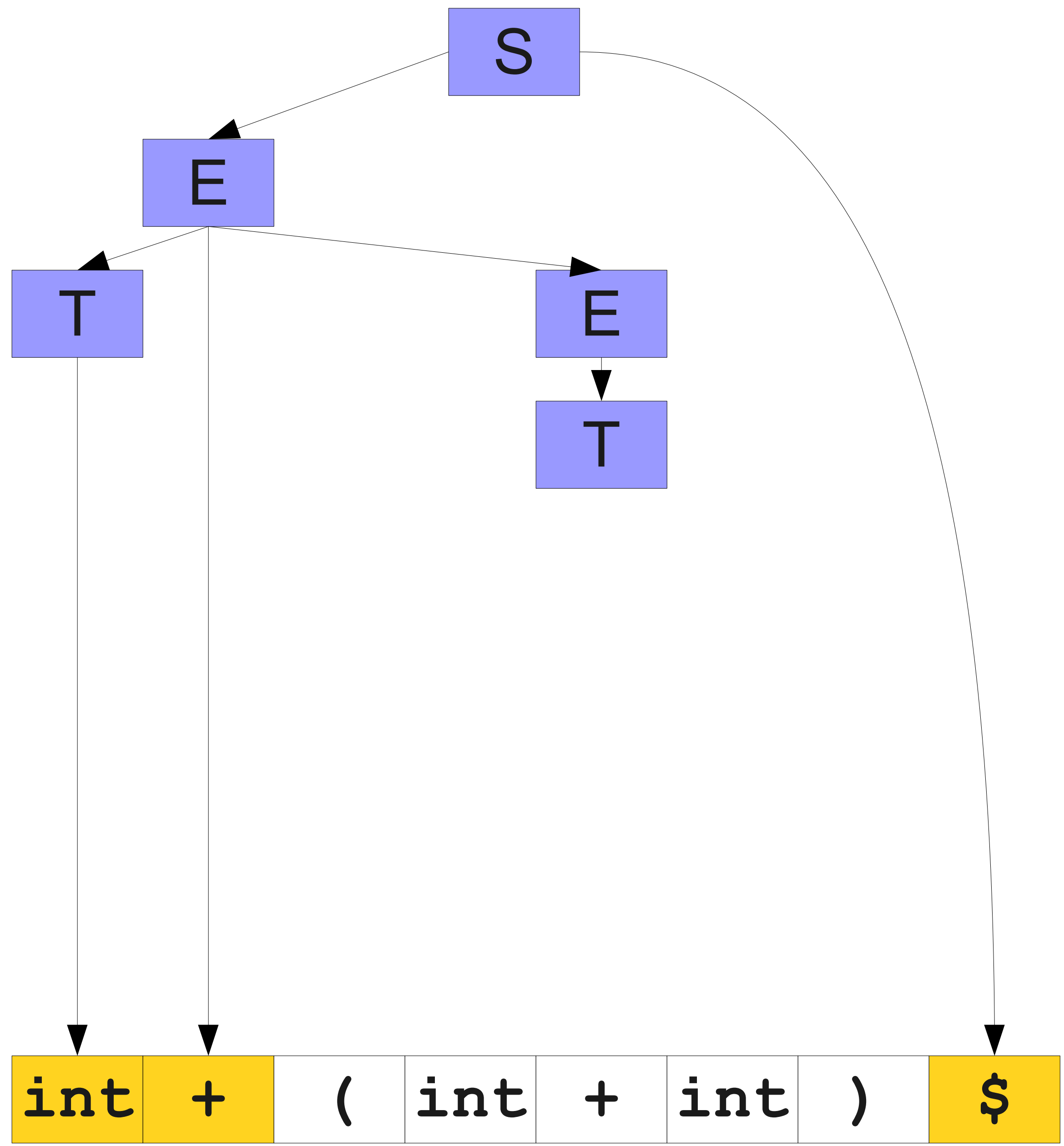
Top-Down Parsing

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



Top-Down Parsing

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



Top-Down Parsing

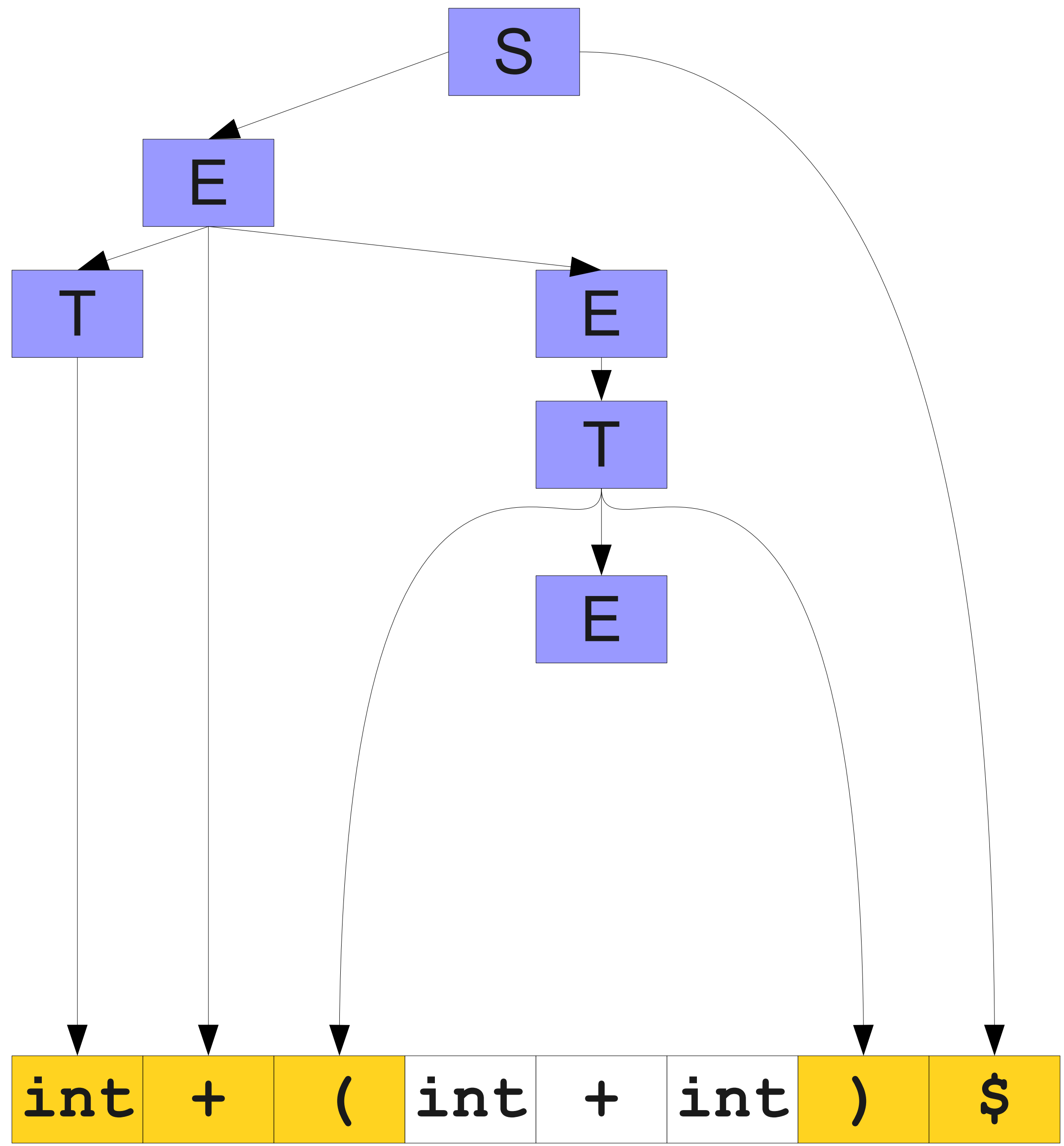
S → E$

E → T

E → T + E

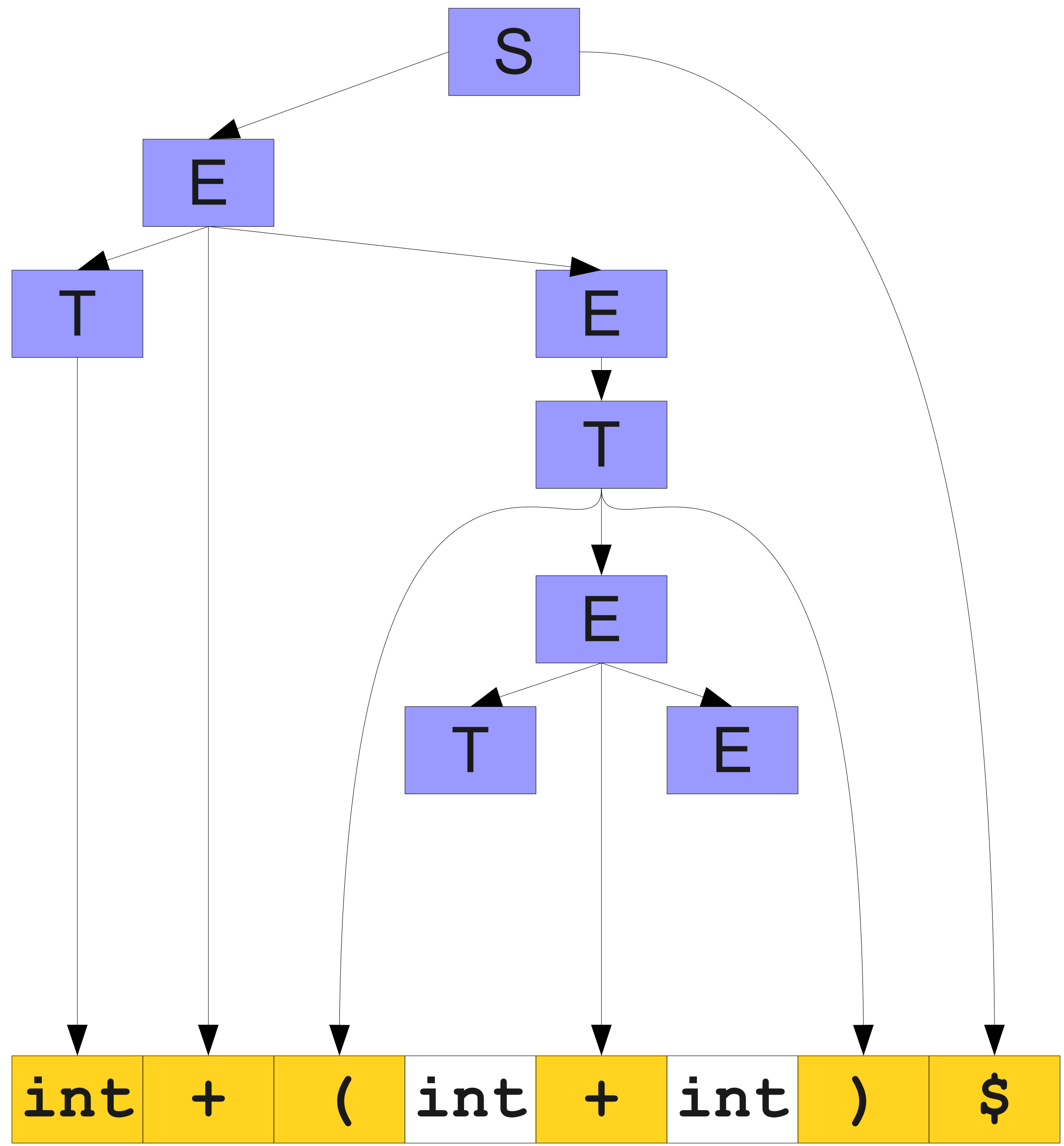
T → int

T → (E)



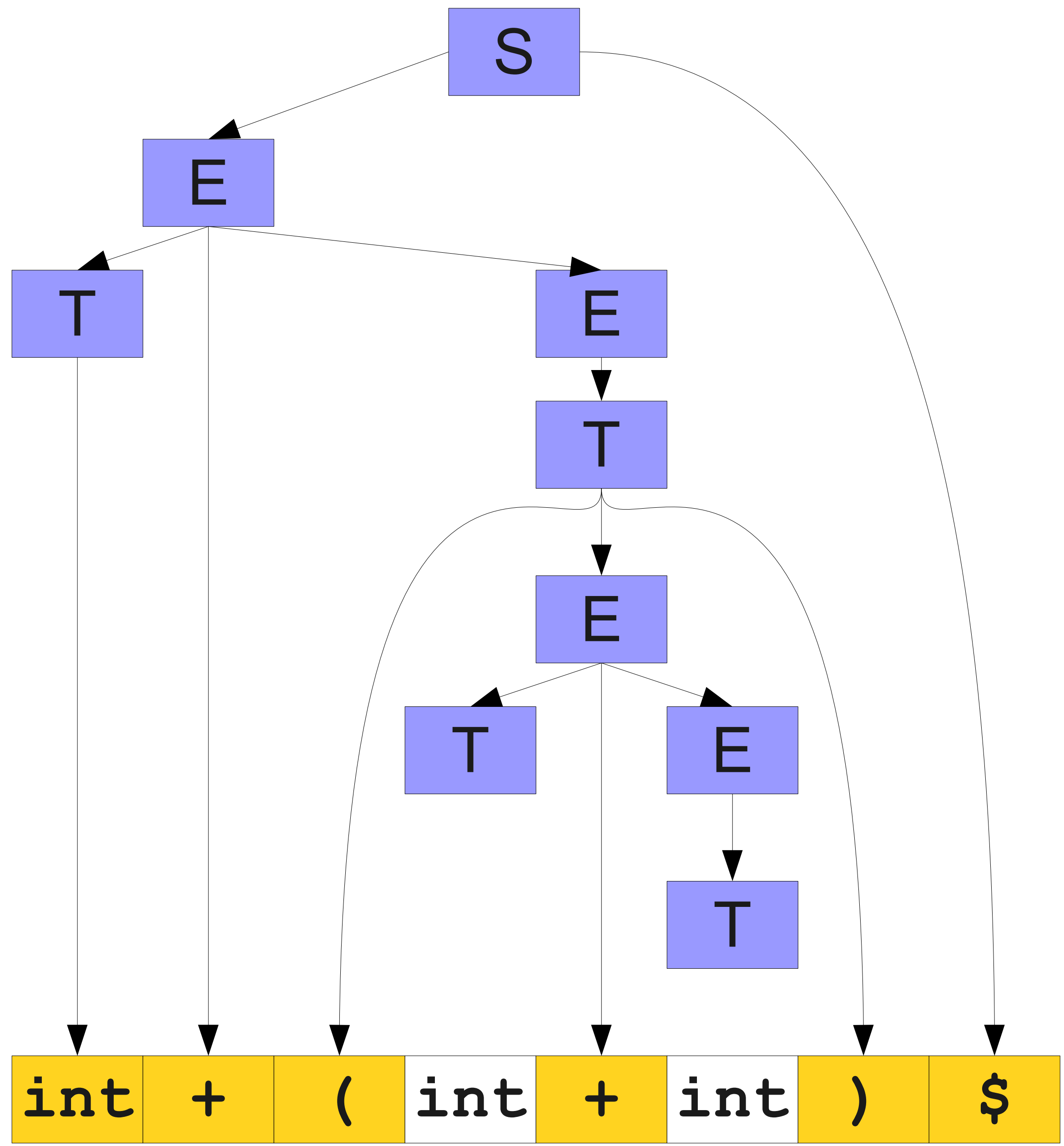
Top-Down Parsing

S → E$
E → T
E → T + E
T → int
T → (E)



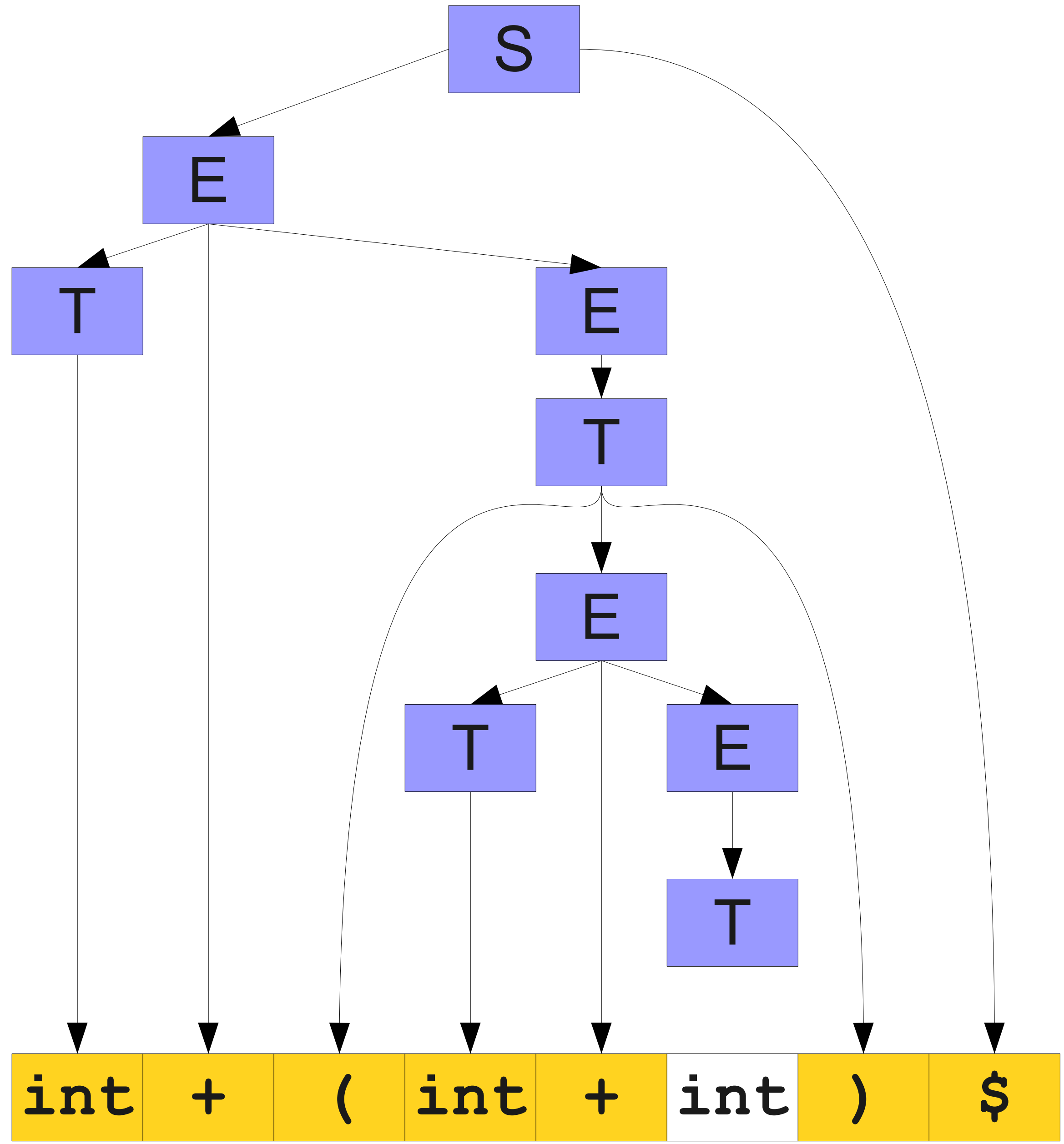
Top-Down Parsing

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



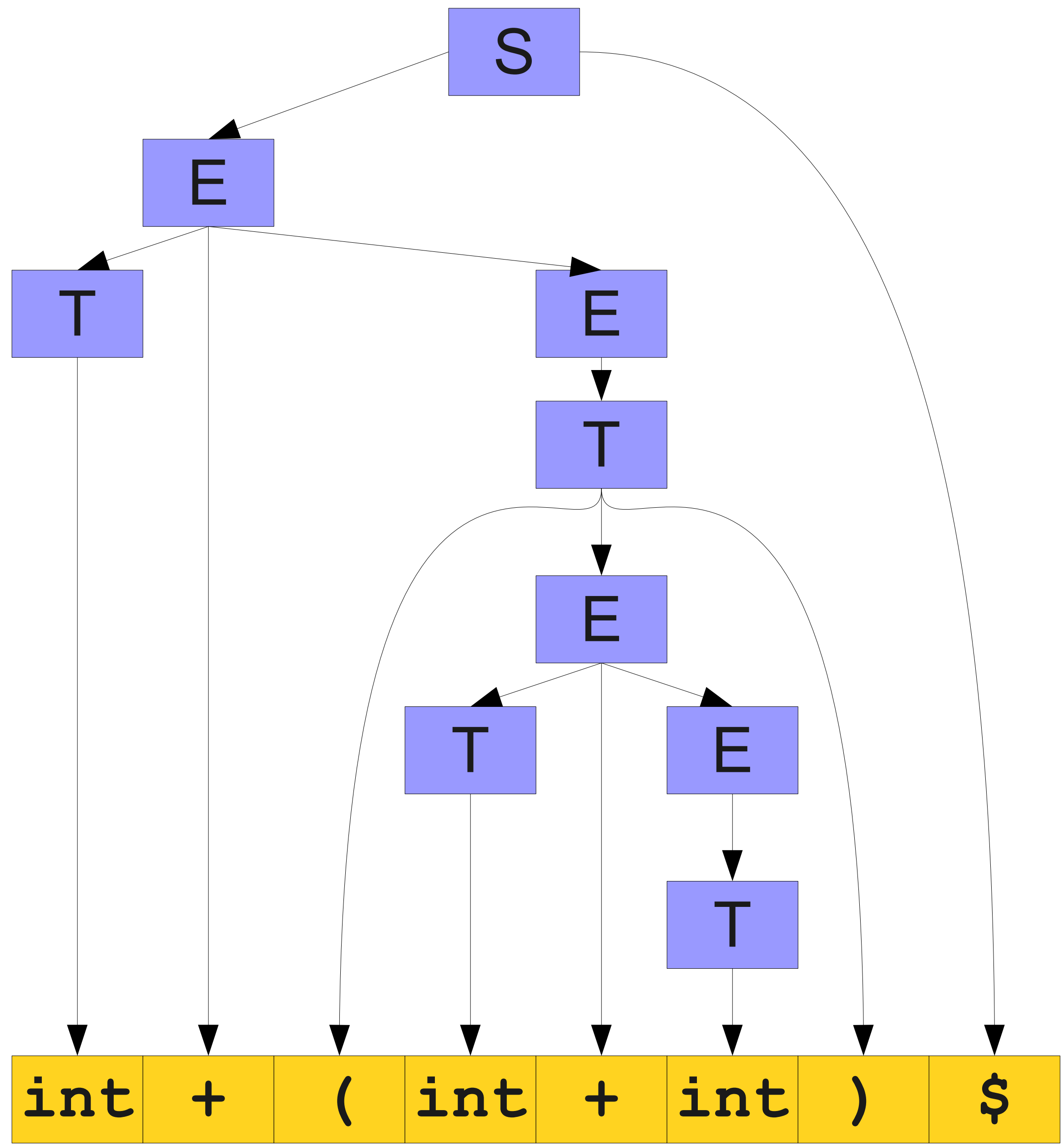
Top-Down Parsing

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



Top-Down Parsing

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



Challenges in Top-Down Parsing

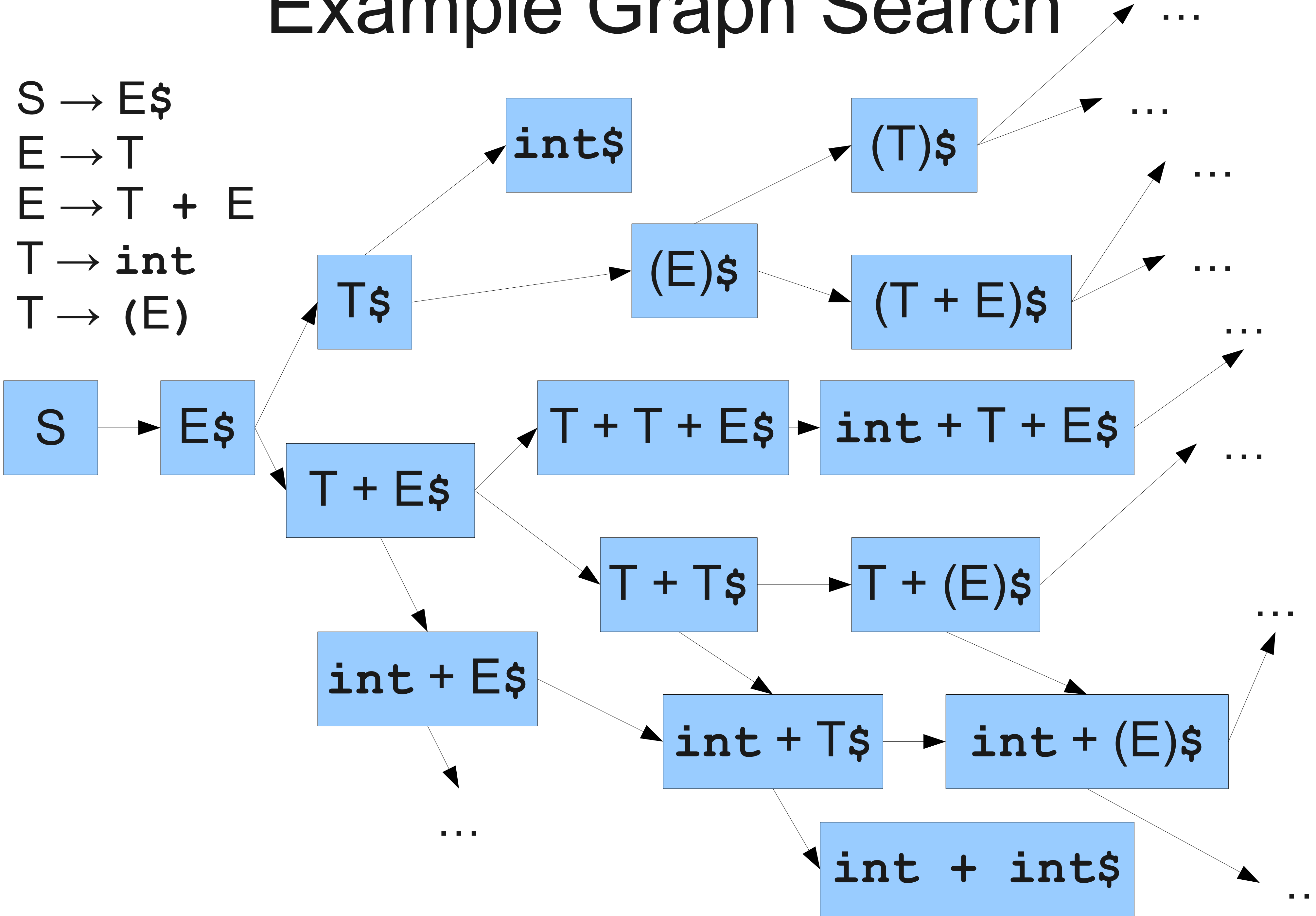
- Top-down parsing begins with virtually no information.
  - Begins with just the start symbol, which matches **every** program.
- How do we know which productions to apply?
- In general, **we can't**.
  - There are some grammars for which the best we can do is guess and backtrack if we're wrong.
  - 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).
- There is an edge from one node to another if there is a reduction from the first sentential form to the second.

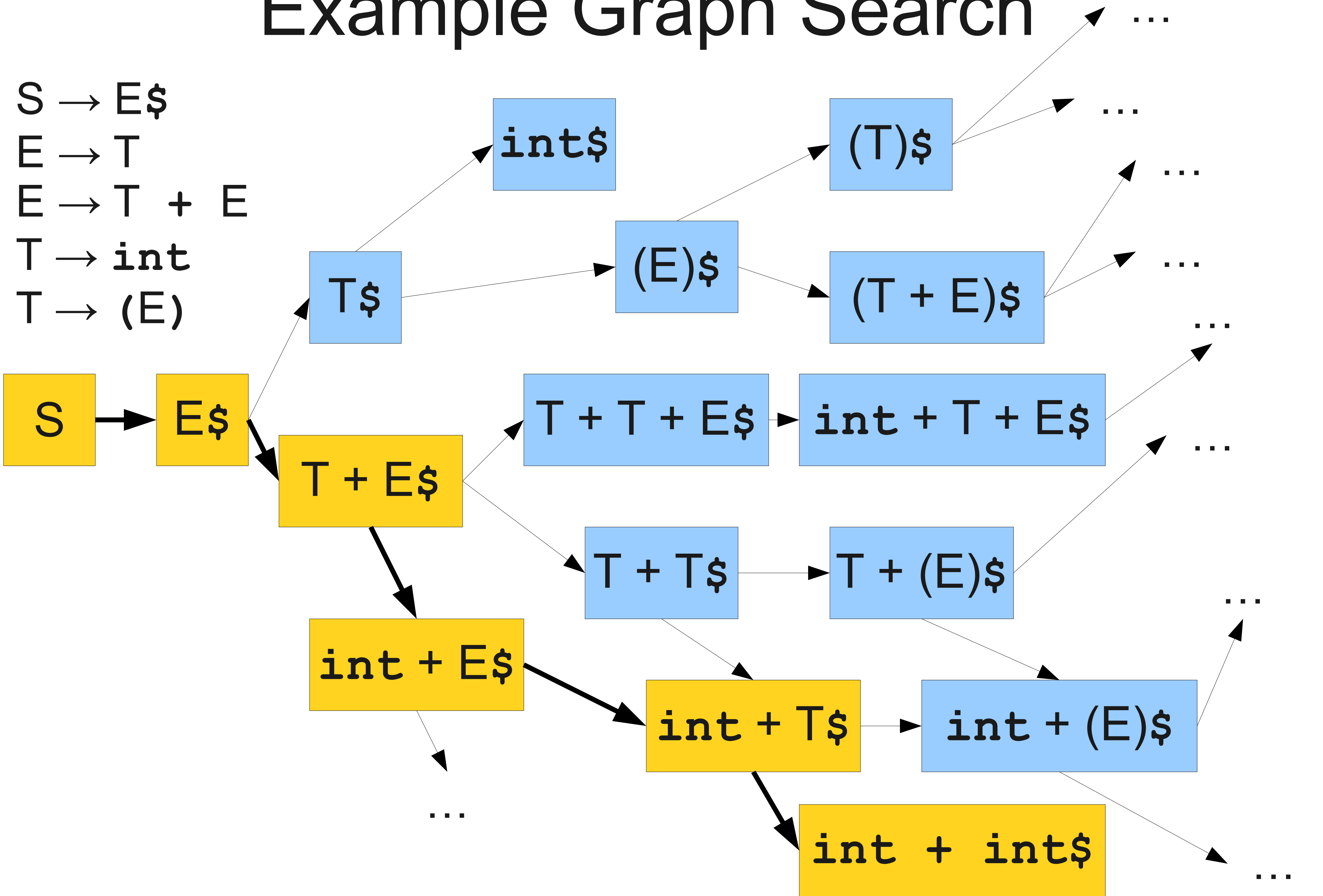
Example Graph Search

S → E$
E → T
E → T + E
T → int
T → (E)



Example Graph Search

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



Our First Top-Down Algorithm

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

Breadth-First Search Parsing

Worklist

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

`int + int$`

Breadth-First Search Parsing

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

`int + int$`

Breadth-First Search Parsing

Worklist

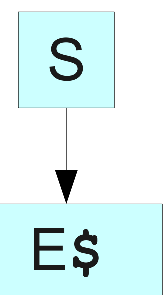
S

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

int + int$

Breadth-First Search Parsing

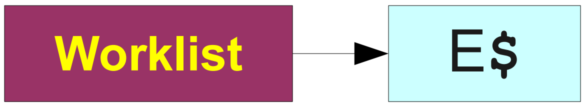
Worklist

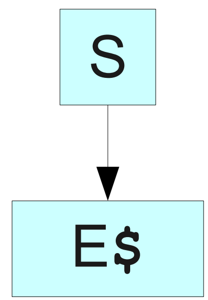


S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Breadth-First Search Parsing

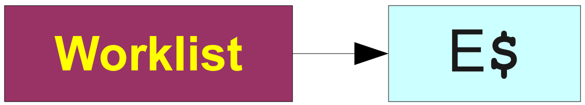




S → E$
E → T
E → T + E
T → `int`
T → (E)

`int + int$`

Breadth-First Search Parsing



$S \rightarrow$ E$

$E \rightarrow T$

$E \rightarrow T$ + E

$T \rightarrow$ **int**

$T \rightarrow$ (E)

int + int$

Breadth-First Search Parsing

Worklist

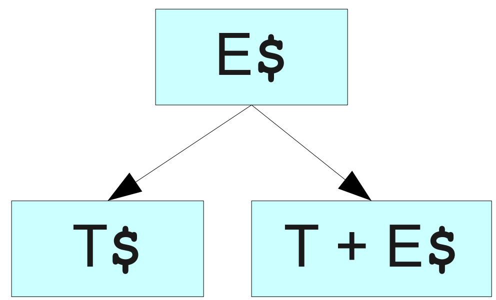
E$

S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Breadth-First Search Parsing

Worklist



$S \rightarrow$ E$

$E \rightarrow T$

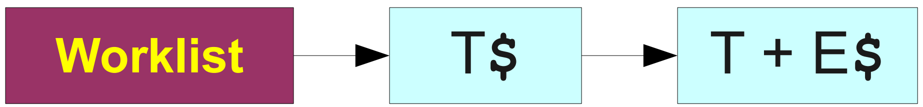
$E \rightarrow$ T + E

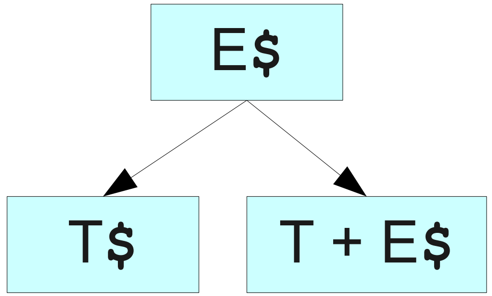
$T \rightarrow$ `int`

$T \rightarrow$ (E)

`int + int$`

Breadth-First Search Parsing

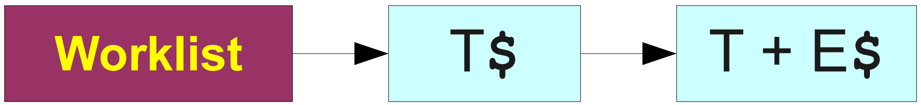




S → E$
E → T
E → T + E
T → **int**
T → (E)

int + int$

Breadth-First Search Parsing



$S \rightarrow E\$$

$E \rightarrow T$

$E \rightarrow T + E$

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

$T \rightarrow (E)$

`int + int$`

Breadth-First Search Parsing

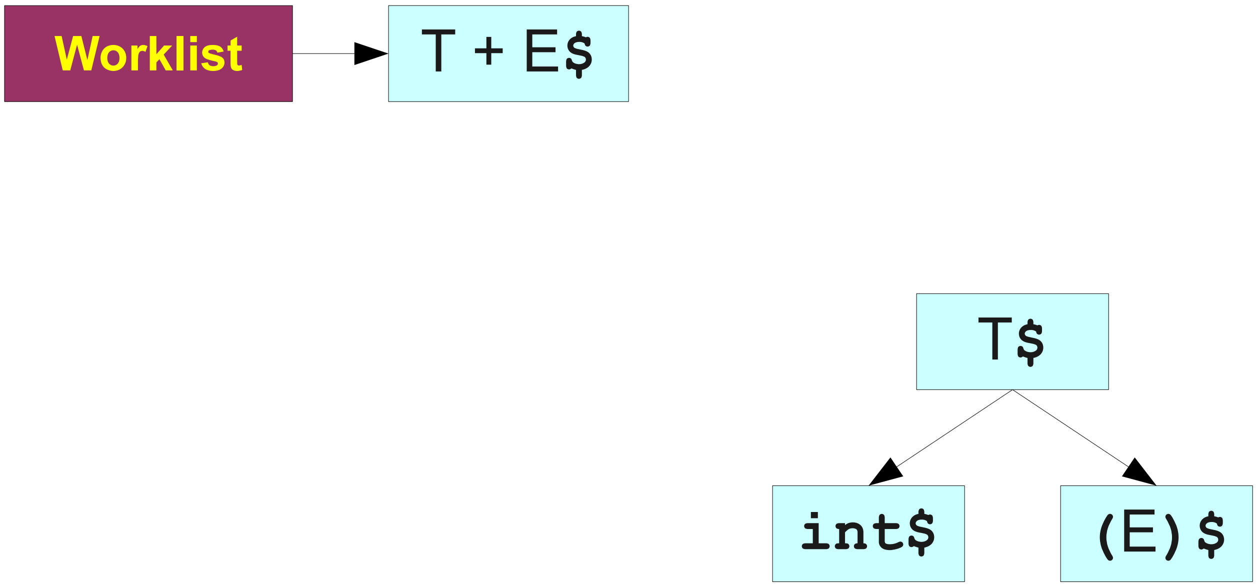
Worklist → T + E$

T$

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

int + int$

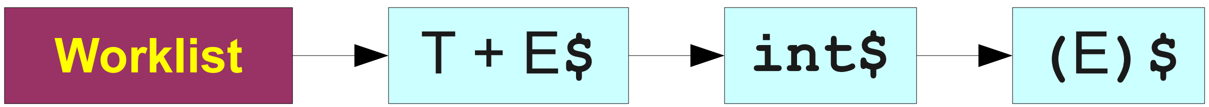
Breadth-First Search Parsing

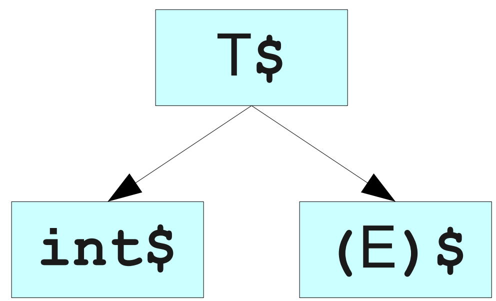


S → E$
E → T
E → T + E
T → int
T → (E)

`int + int$`

Breadth-First Search Parsing





S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Breadth-First Search Parsing

Worklist → T + E$ → int$ → (E)$

S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Breadth-First Search Parsing

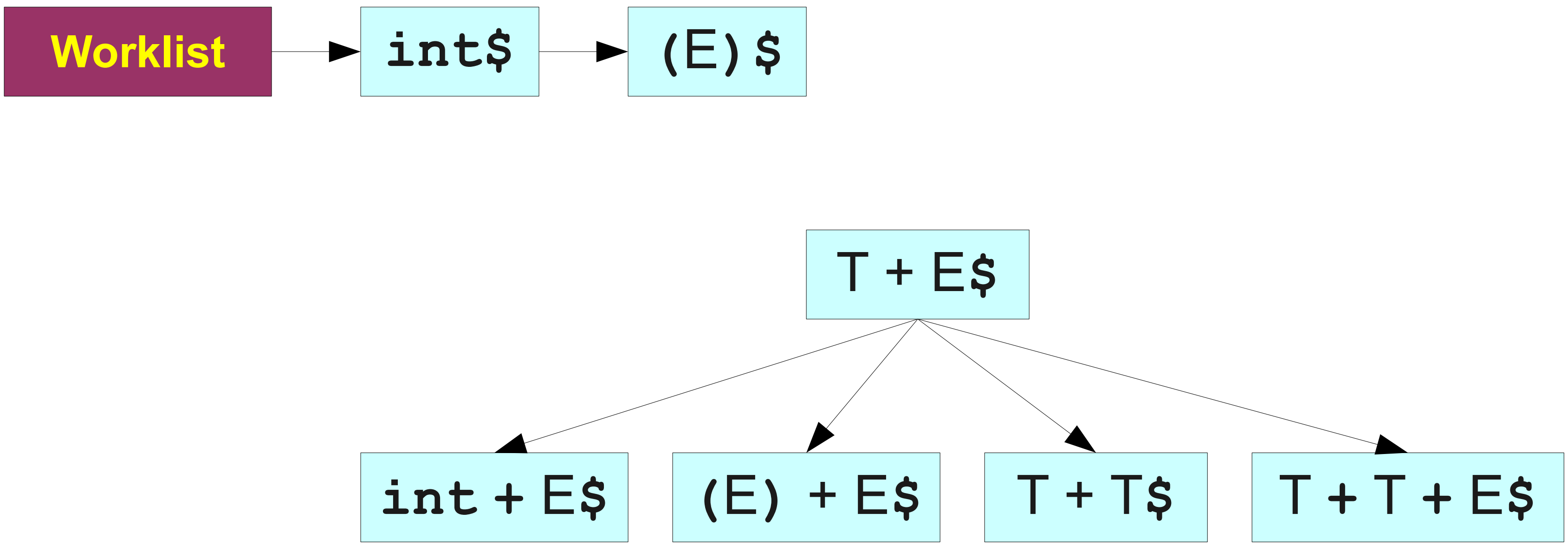
Worklist → `int$` → `(E)$`

T + E$

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

int + int$

Breadth-First Search Parsing



$S \to E\$$

$E \to T$

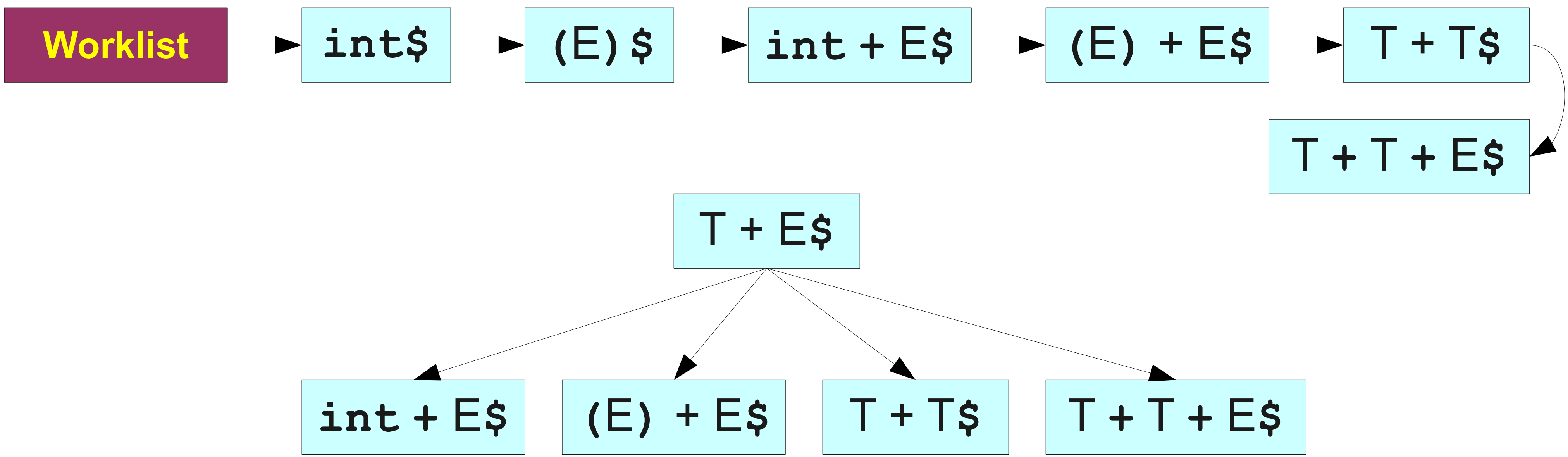
$E \to T + E$

$T \to$ **int**

$T \to (E)$

int + int$

Breadth-First Search Parsing



$S \to E\$$

$E \to T$

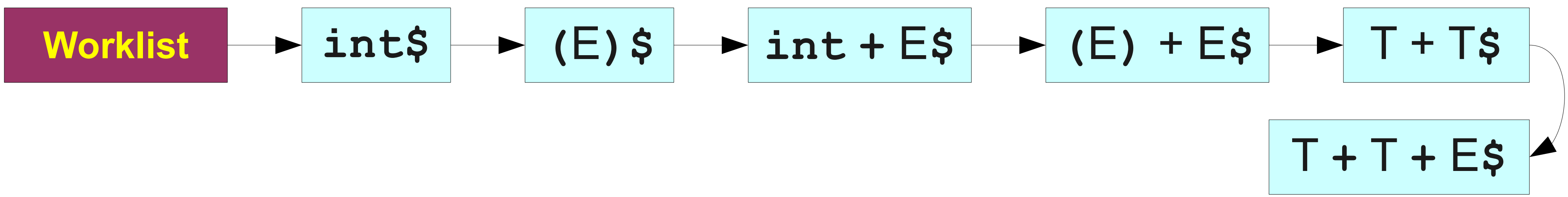
$E \to T + E$

$T \to \texttt{int}$

$T \to (E)$

`int + int$`

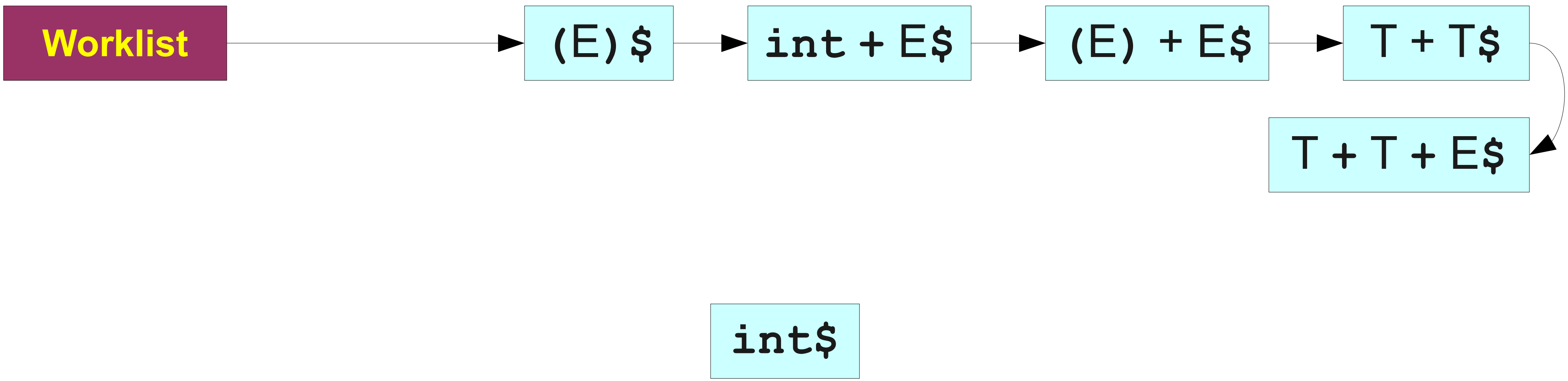
Breadth-First Search Parsing



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

int + int$

Breadth-First Search Parsing



$S \to E\$$

$E \to T$

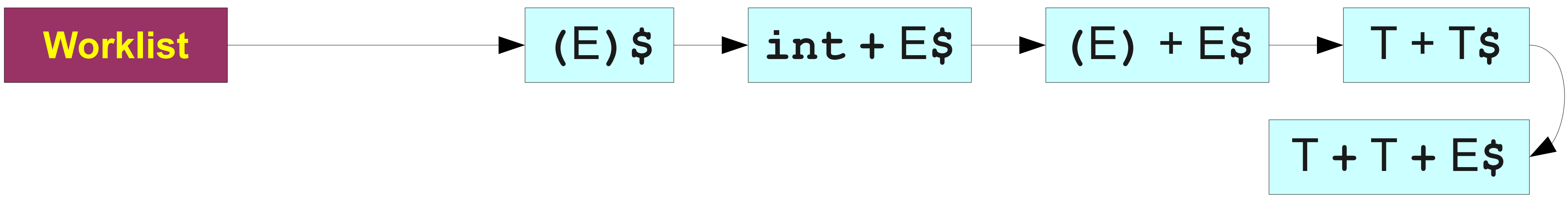
$E \to T + E$

$T \to \texttt{int}$

$T \to (E)$

`int + int$`

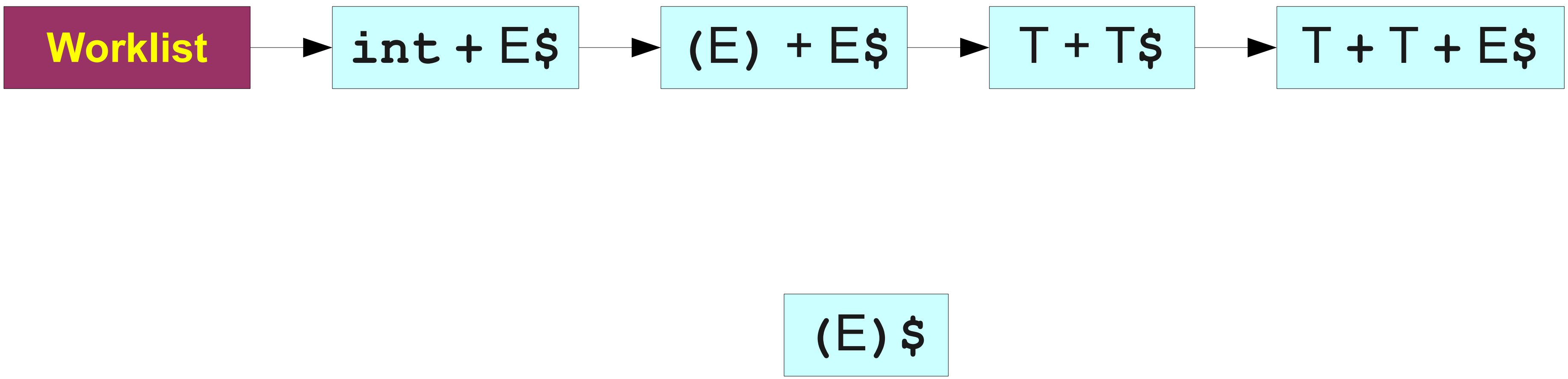
Breadth-First Search Parsing



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

int + int$

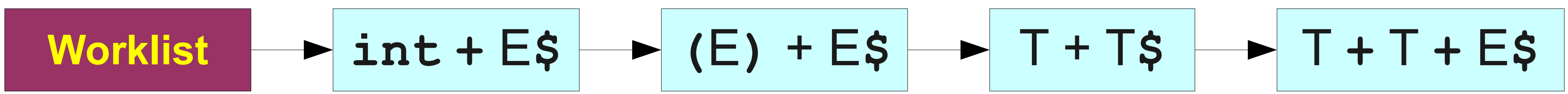
Breadth-First Search Parsing

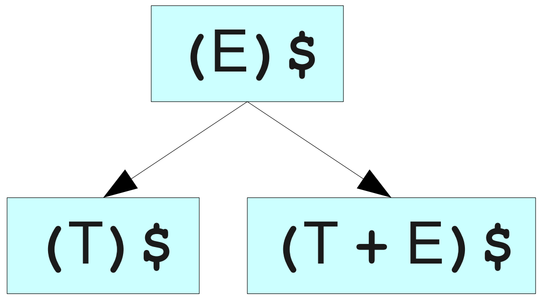


S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Breadth-First Search Parsing

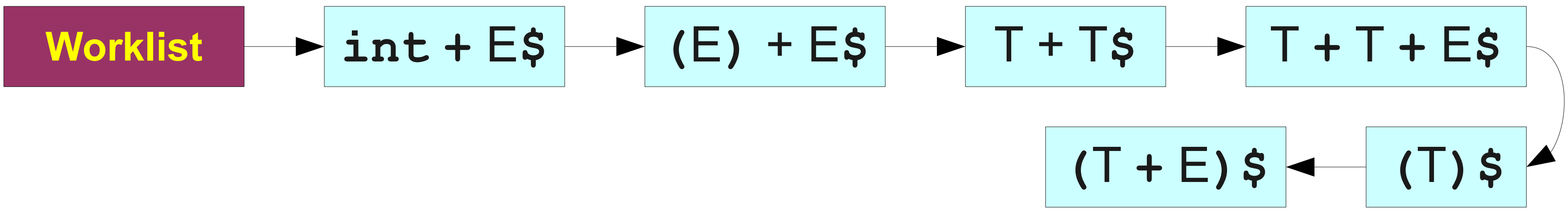


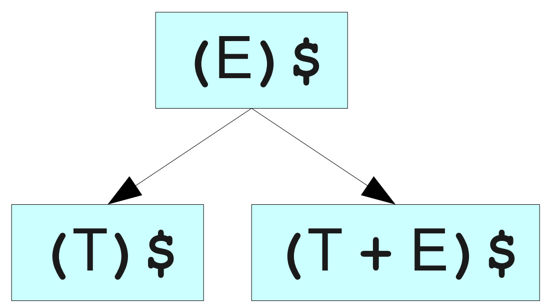


S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Breadth-First Search Parsing

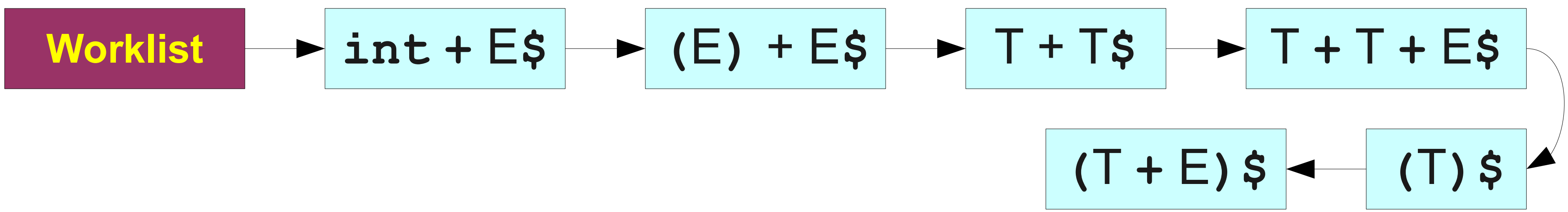




S → E$
E → T
E → T + E
T → **int**
T → (E)

int + int$

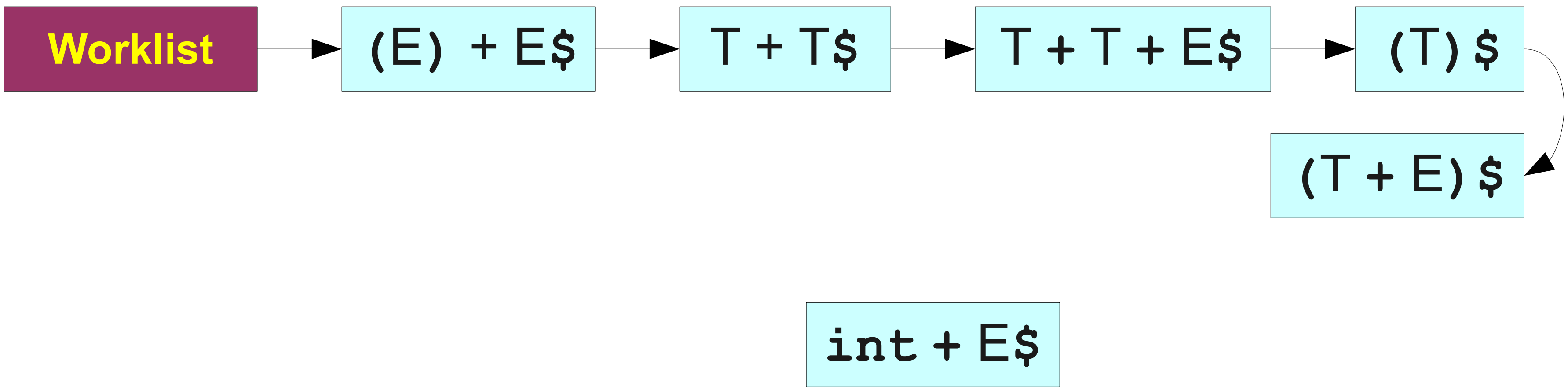
Breadth-First Search Parsing



S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

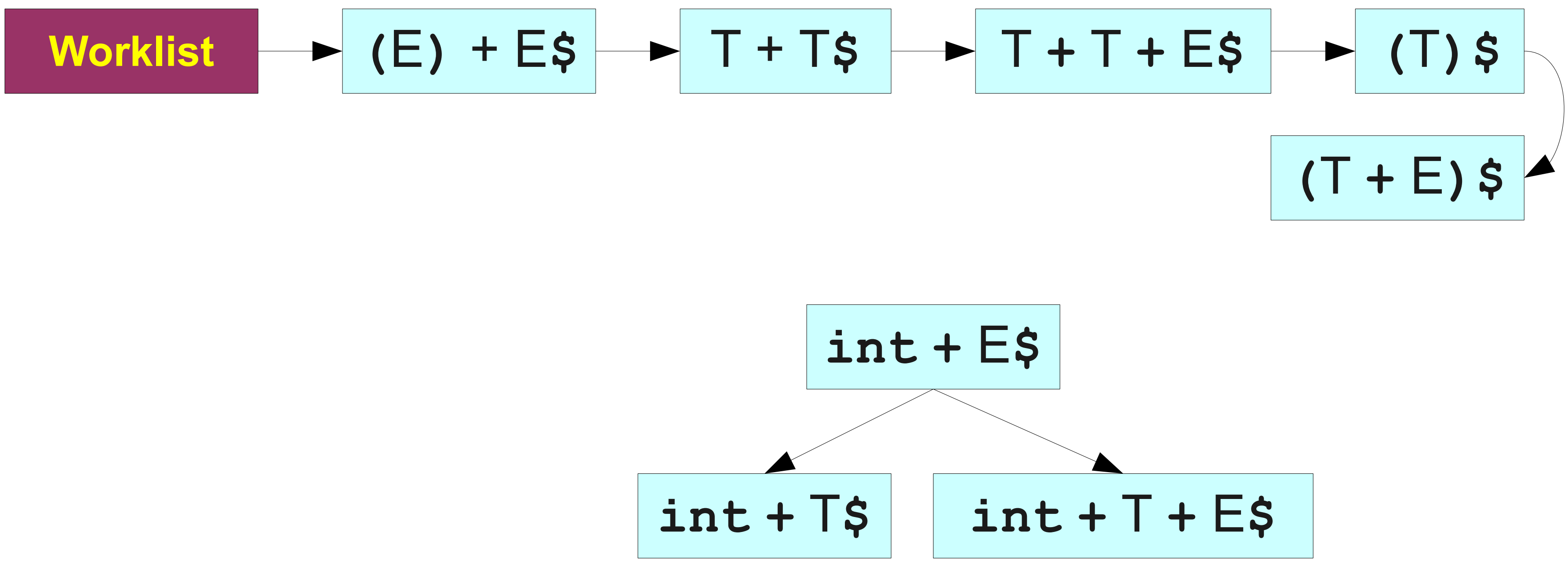
Breadth-First Search Parsing



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

int + int$

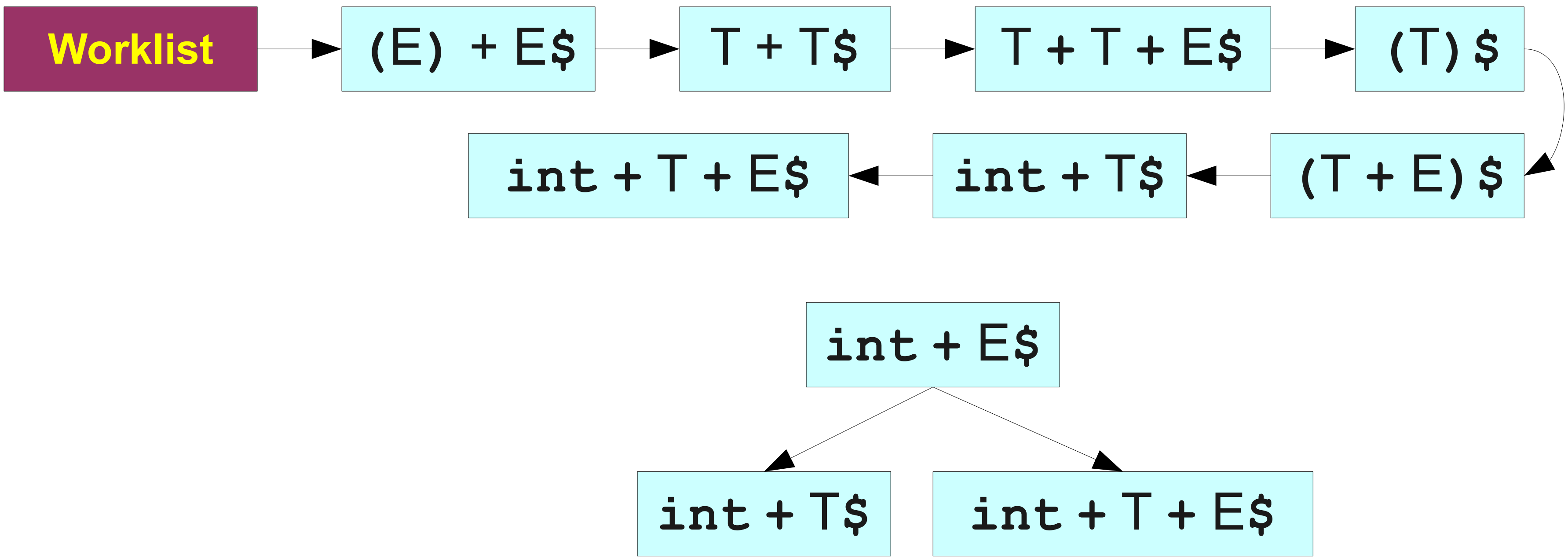
Breadth-First Search Parsing



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

int + int$

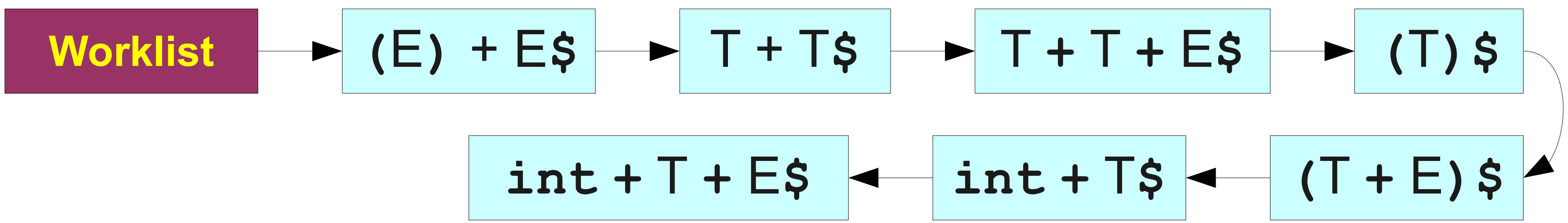
Breadth-First Search Parsing



S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Breadth-First Search Parsing



S → E$

E → T

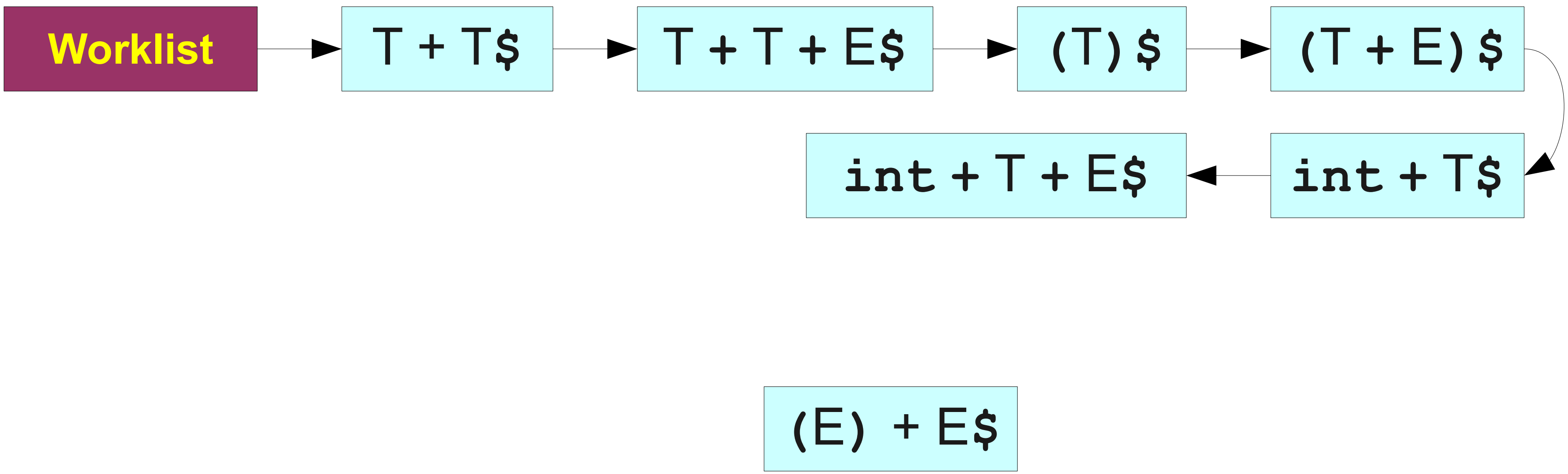
E → T + E

T → **int**

T → (E)

int + int$

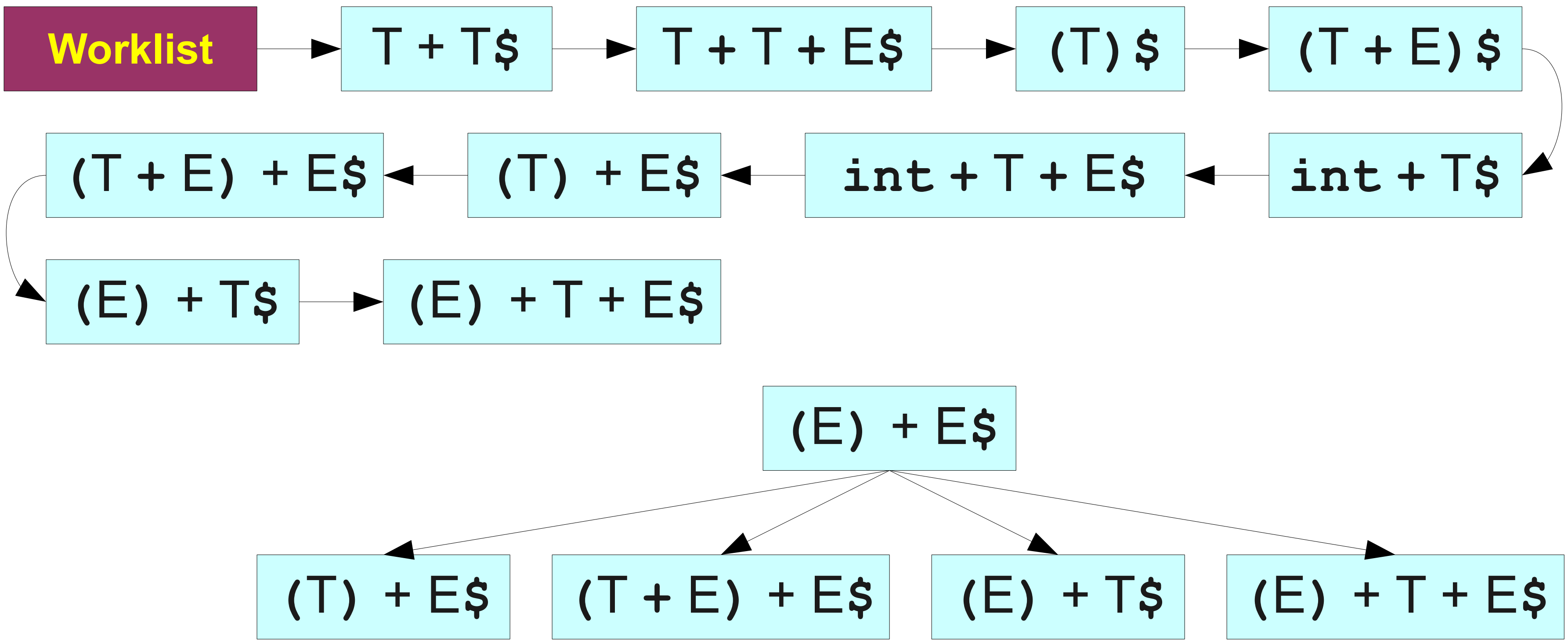
Breadth-First Search Parsing



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

int + int$

Breadth-First Search Parsing



$S \to E\$$

$E \to T$

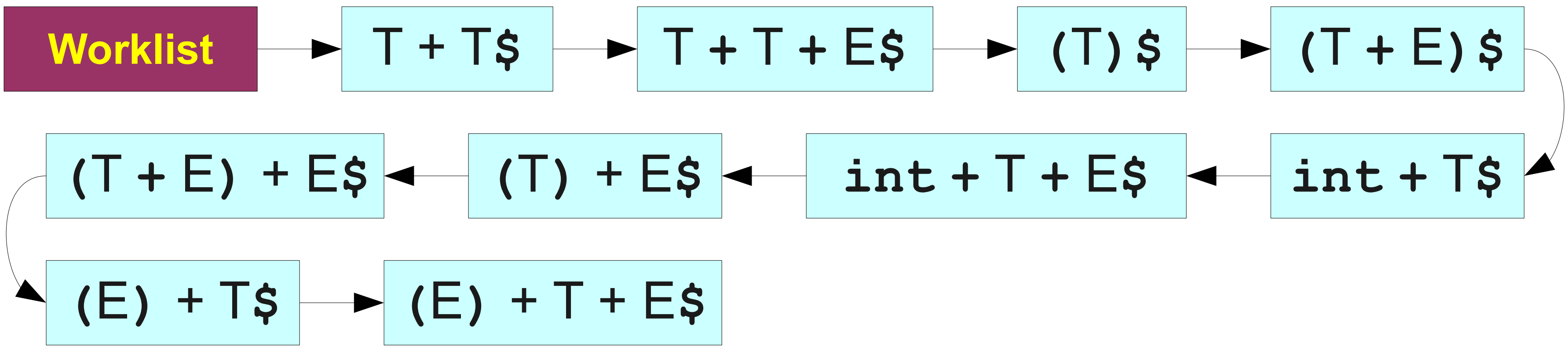
$E \to T + E$

$T \to \texttt{int}$

$T \to (E)$

int + int$

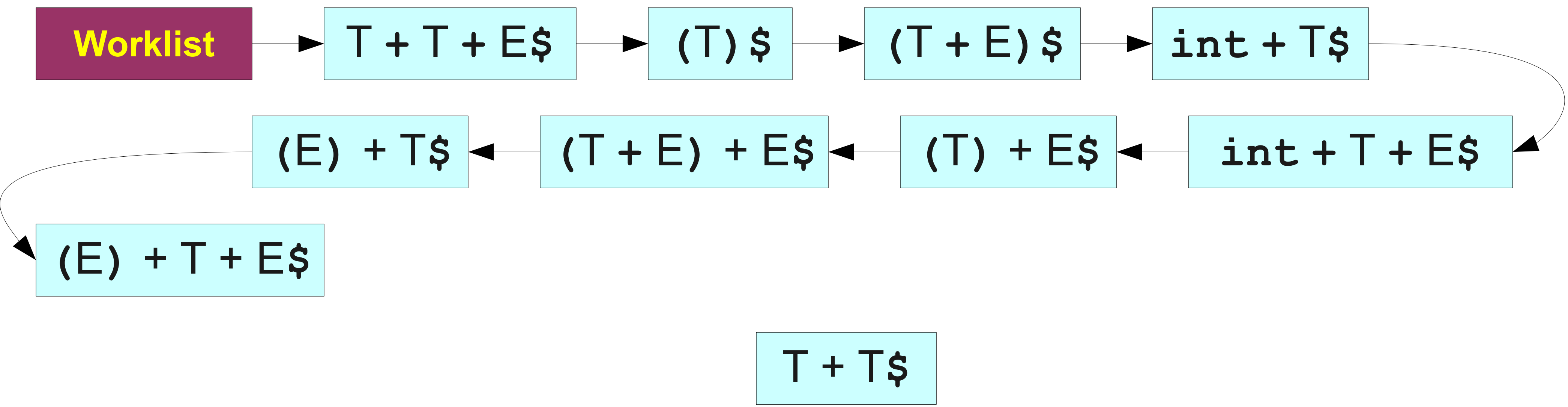
Breadth-First Search Parsing



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

int + int$

Breadth-First Search Parsing



$S \rightarrow E\$$

$E \rightarrow T$

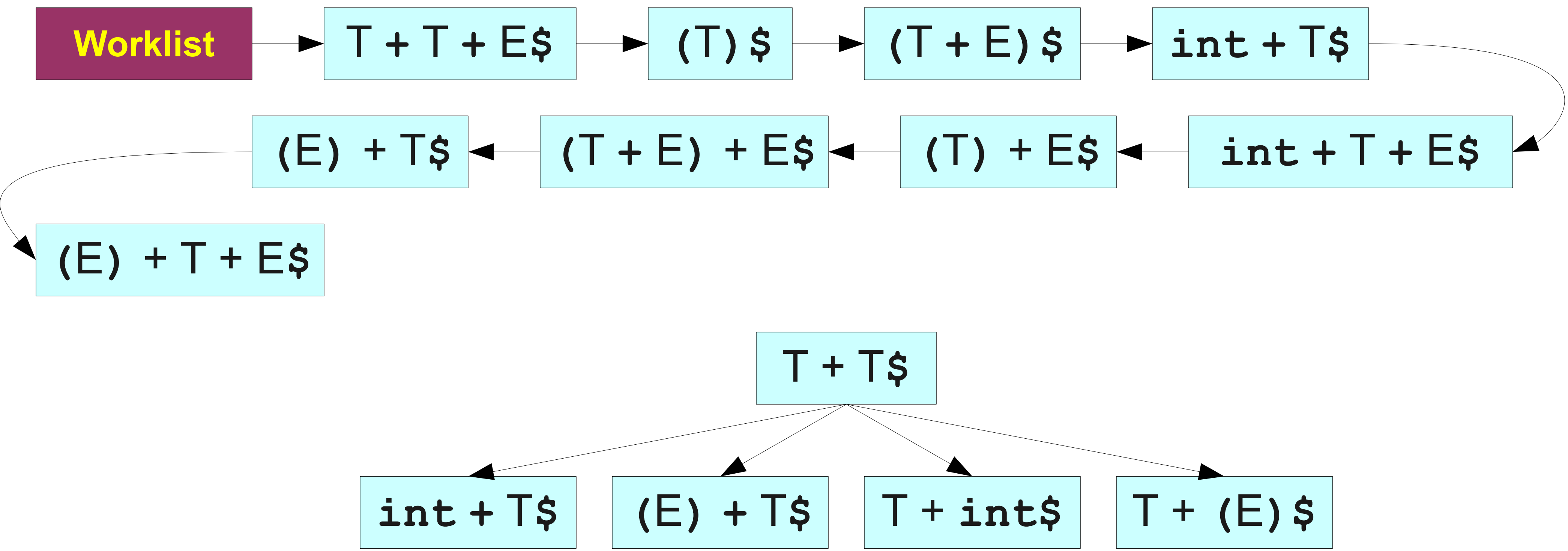
$E \rightarrow T + E$

$T \rightarrow$ **int**

$T \rightarrow (E)$

int + int$

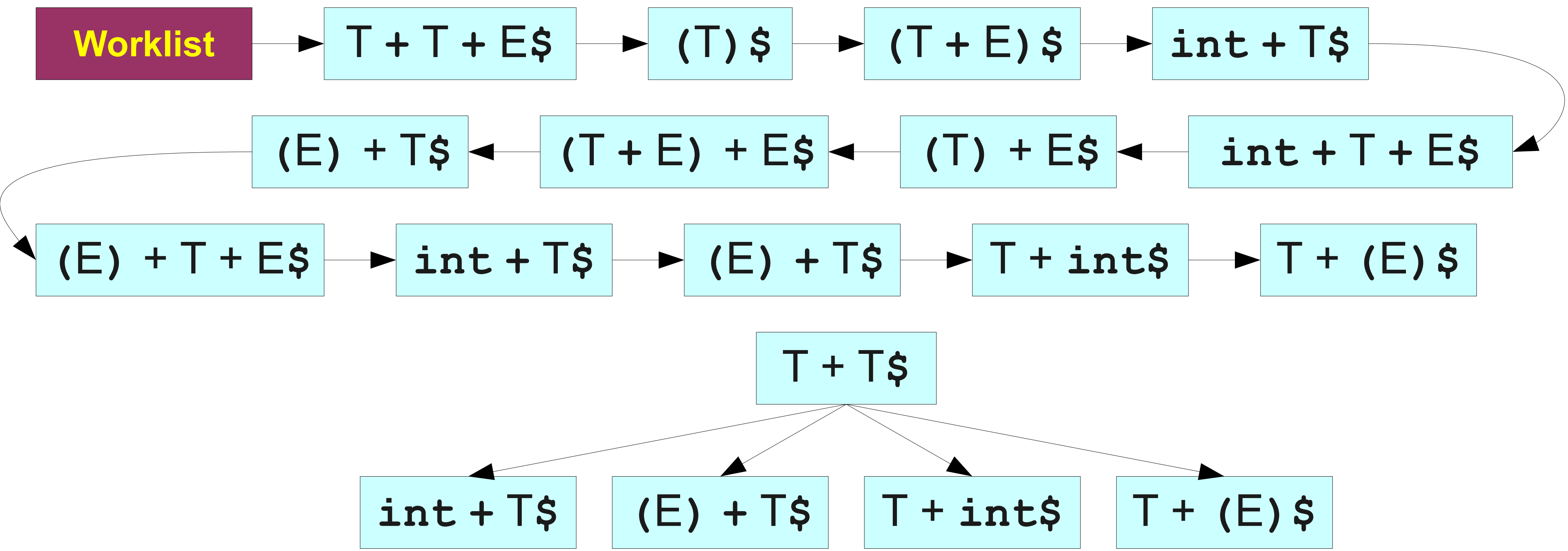
Breadth-First Search Parsing



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

int + int$

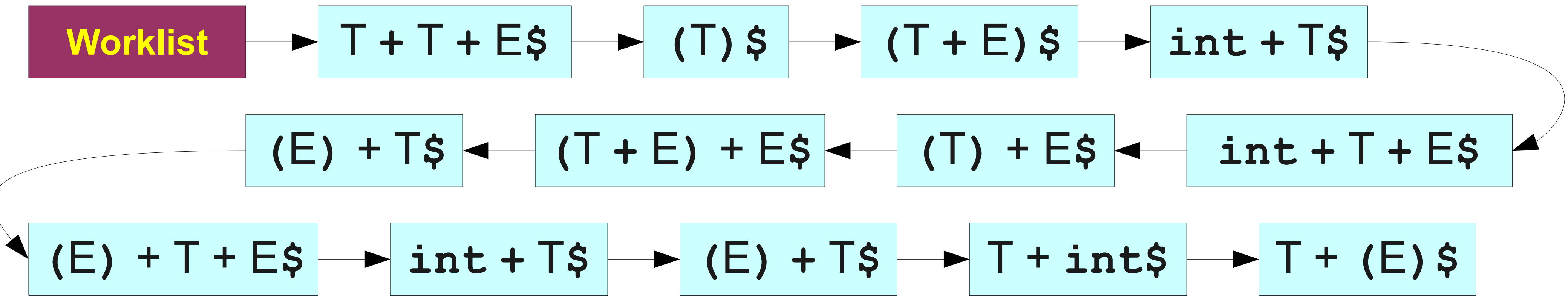
Breadth-First Search Parsing



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

int + int$

Breadth-First Search Parsing



S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Breadth-First Search Parsing

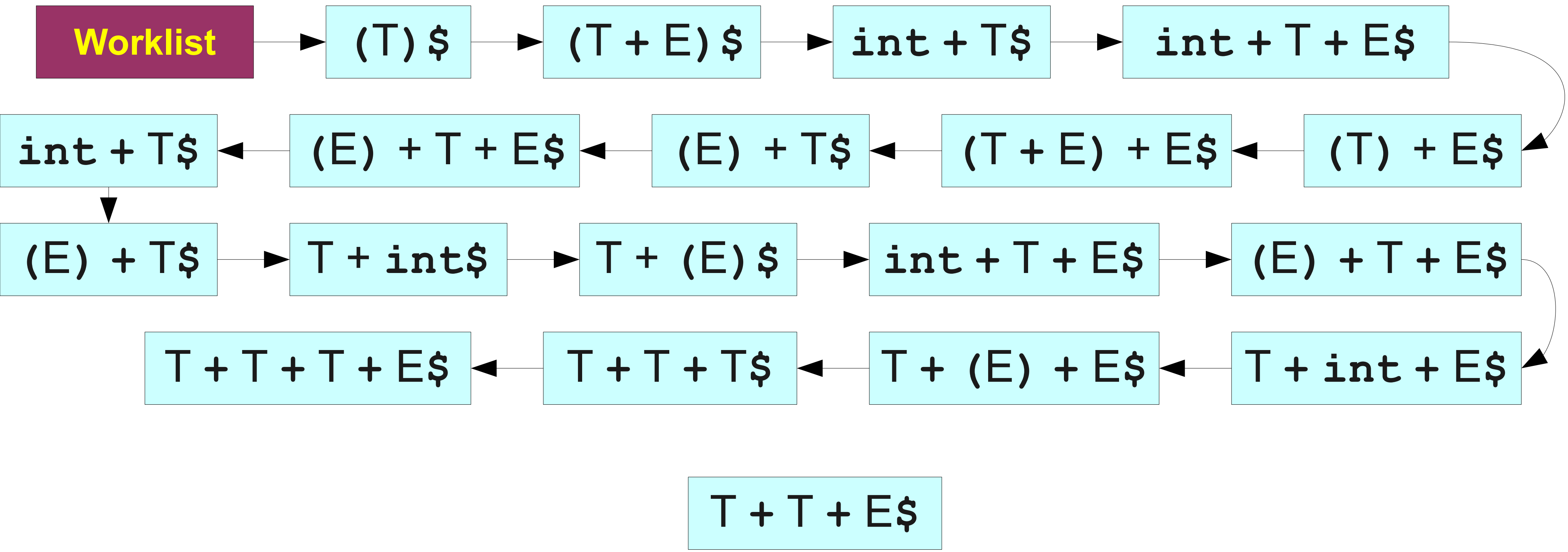
Worklist → (T)$ → (T + E)$ → int + T$ → int + T + E$ → (T) + E$ → (T + E) + E$ → (E) + T$ → (E) + T + E$ → int + T$ → (E) + T$ → T + int$ → T + (E)$

T + T + E$

S → E$
E → T
E → T + E
T → int
T → (E)

`int + int$`

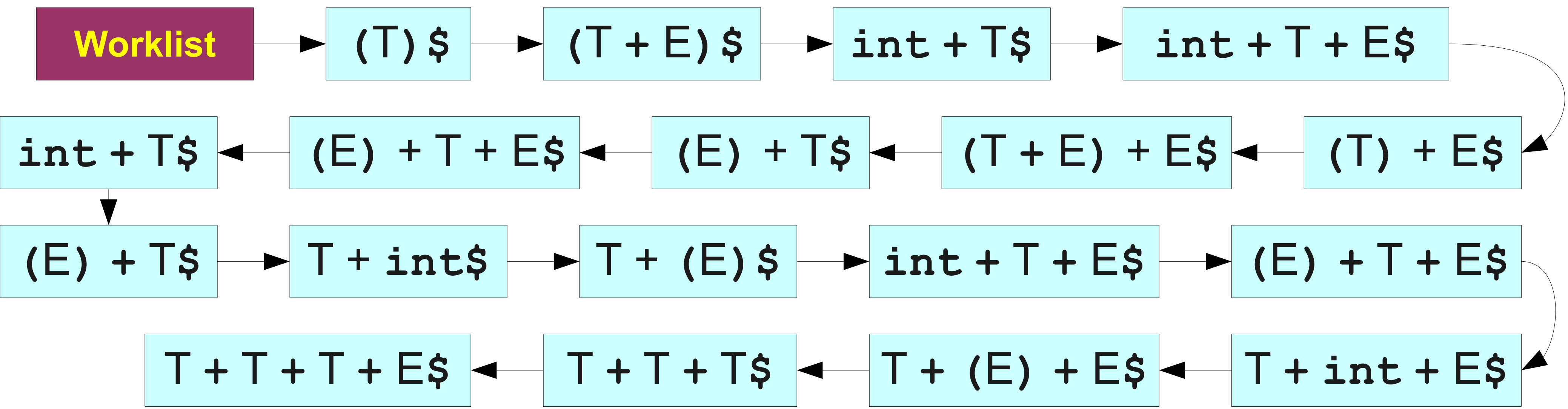
Breadth-First Search Parsing



S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

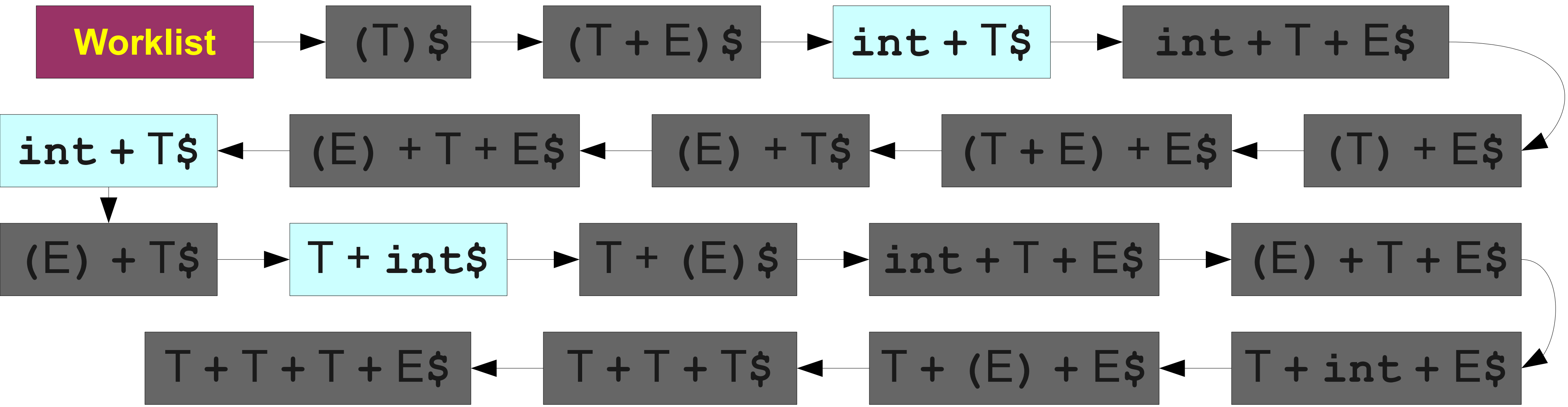
Breadth-First Search Parsing



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

int + int$

Breadth-First Search Parsing



S → 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 of terminals *w*.
- Suppose we have a sentential form *v* = *xy*, where *x* is a string of terminals and *y* is a string of terminals and nonterminals.
- If *x* isn't a prefix of *w*, then no string derived from *v* can **ever** match *w*.
- If we can find a way to try to get a prefix of terminals at the front of our sentential forms, then we can start pruning out impossible options.

Reducing the Branching Factor

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

Leftmost Derivations

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

Leftmost BFS

Worklist

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

int + int$

Leftmost BFS

S → E$

E → T

E → T + E

T → int

T → (E)

int + int$

Leftmost BFS

Worklist

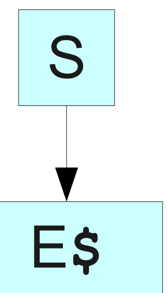
S

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

`int + int$`

Leftmost BFS

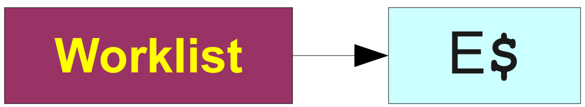
Worklist

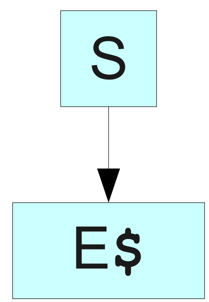


S → E$
E → T
E → T + E
T → `int`
T → (E)

`int + int$`

Leftmost BFS





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

int + int$

Leftmost BFS

Worklist → E$

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

`int + int$`

Leftmost BFS

Worklist

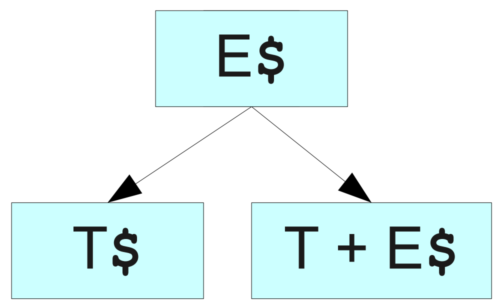
E$

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

`int + int$`

Leftmost BFS

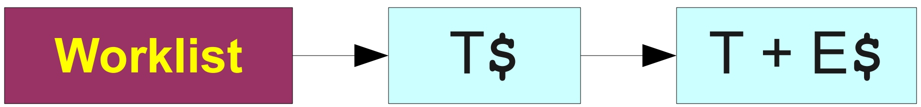
Worklist

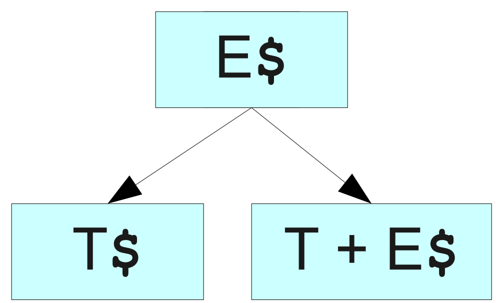


$S \to E\$$
$E \to T$
$E \to T + E$
$T \to \texttt{int}$
$T \to (E)$

`int + int$`

Leftmost BFS

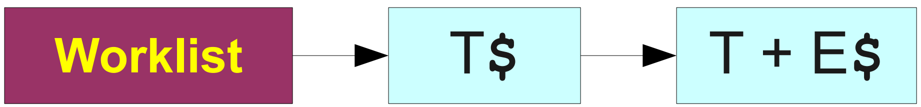




S → E$
E → T
E → T `+` E
T → `int`
T → `(`E`)`

`int + int$`

Leftmost BFS



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

int + int$

Leftmost BFS

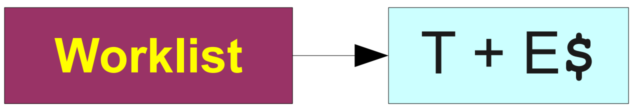
Worklist → T + E$

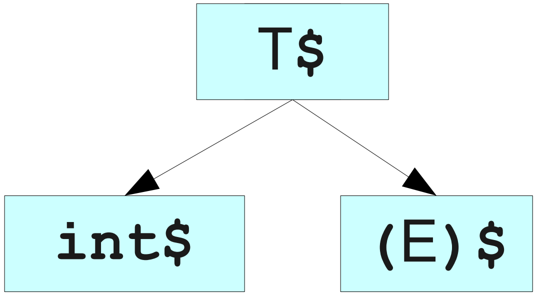
T$

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

`int + int$`

Leftmost BFS

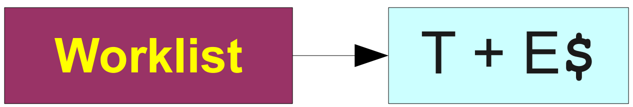


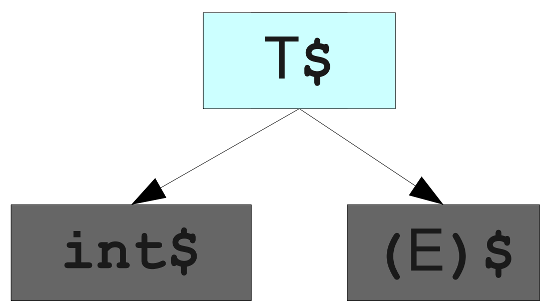


$S \rightarrow E\$$
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow$ **int**
$T \rightarrow (E)$

int + int$

Leftmost BFS





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

int + int$

Leftmost BFS

Worklist → T + E$

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

int + int$

Leftmost BFS

Worklist

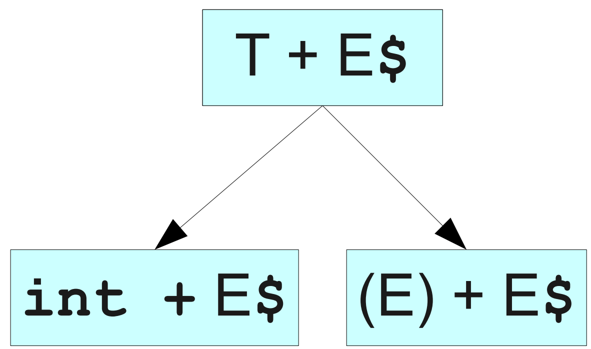
T + E$

S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Leftmost BFS

Worklist

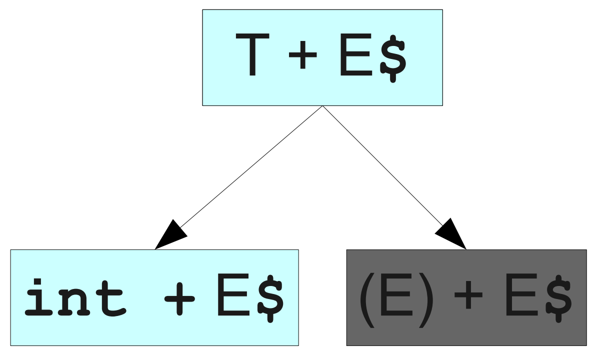


S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Leftmost BFS

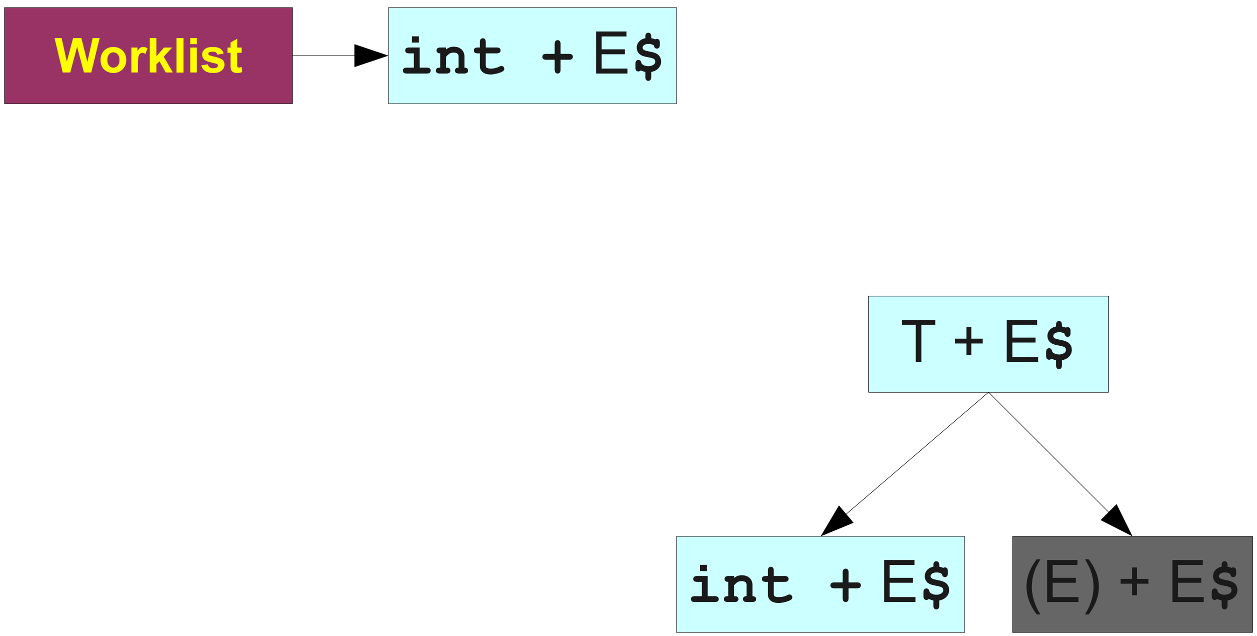
Worklist



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

int + int$

Leftmost BFS



S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Leftmost BFS

Worklist → `int` + E$

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

int + int$

Leftmost BFS

Worklist

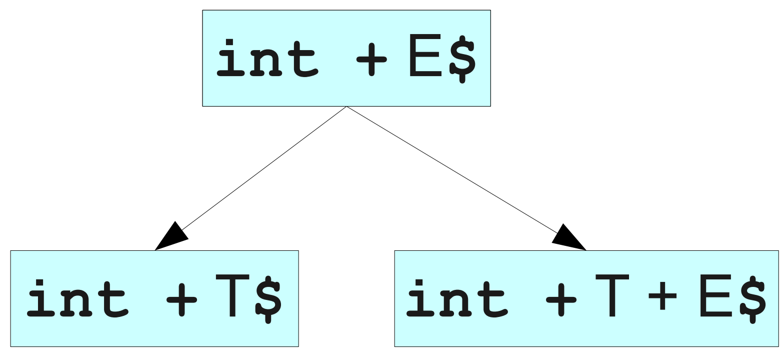
`int + E$`

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

`int + int$`

Leftmost BFS

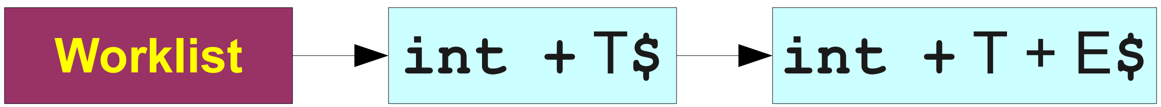
Worklist

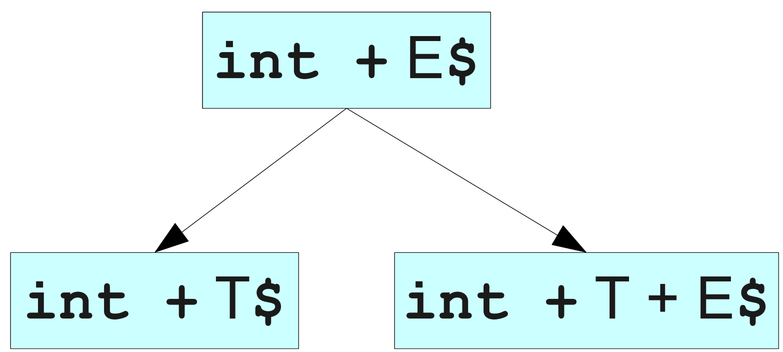


$S \rightarrow E\$$
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow$ int
$T \rightarrow$ (E)

int + int$

Leftmost BFS





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

int + int$

Leftmost BFS

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

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

int + int$

Leftmost BFS

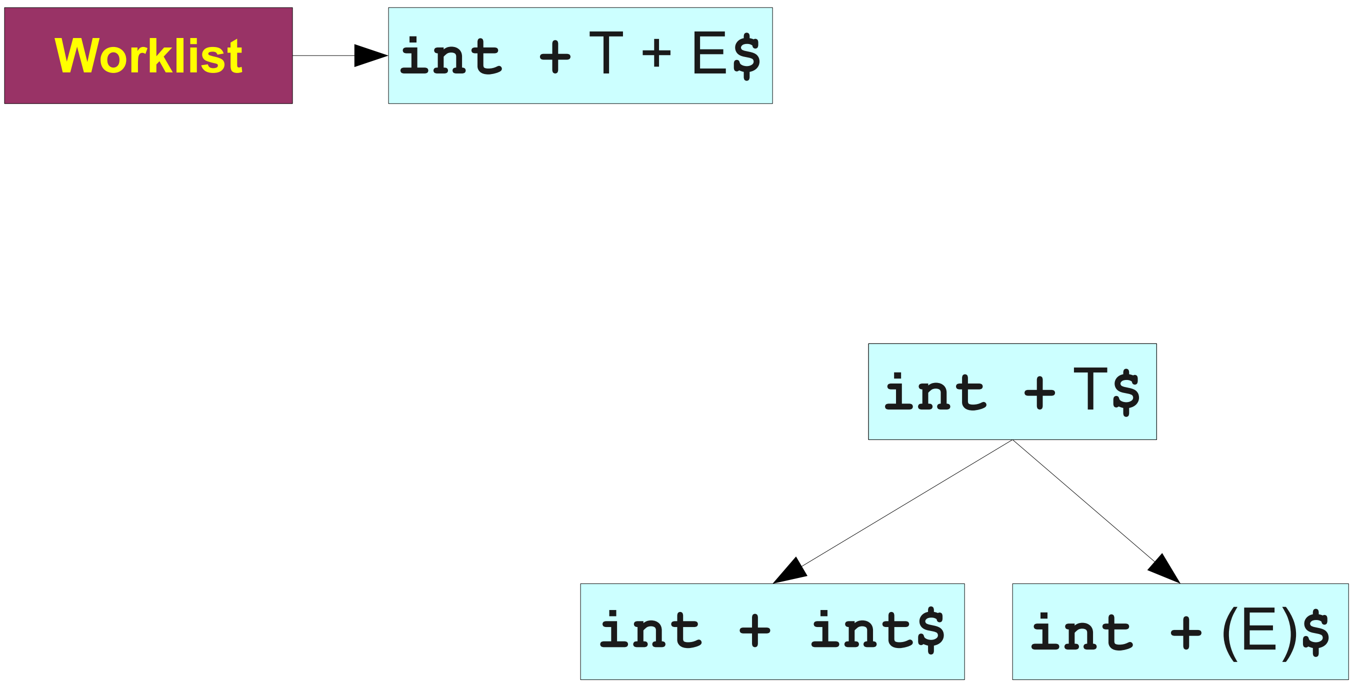
Worklist → `int` + T + E$

`int` + T$

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

`int + int$`

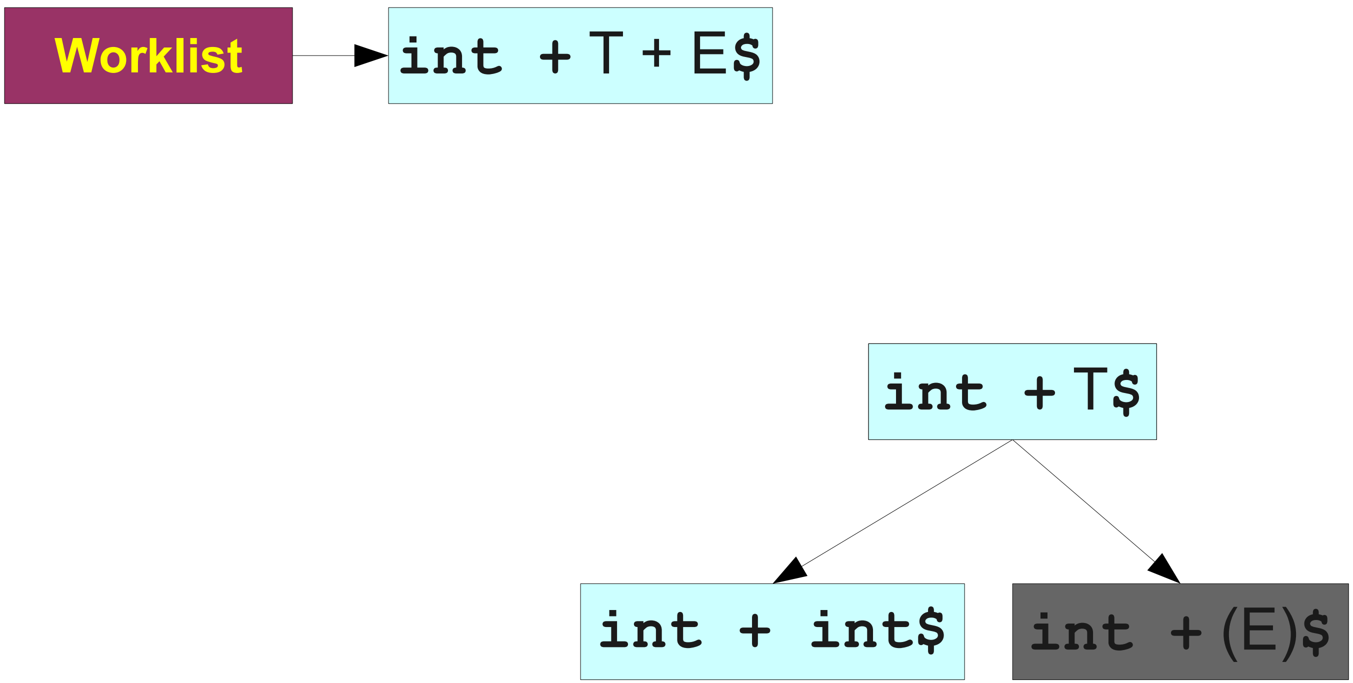
Leftmost BFS



S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

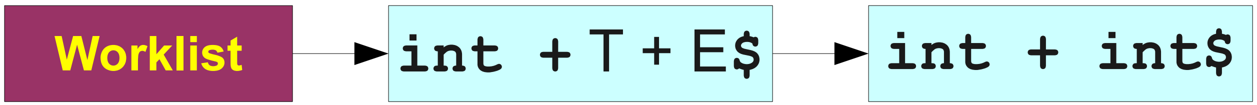
Leftmost BFS

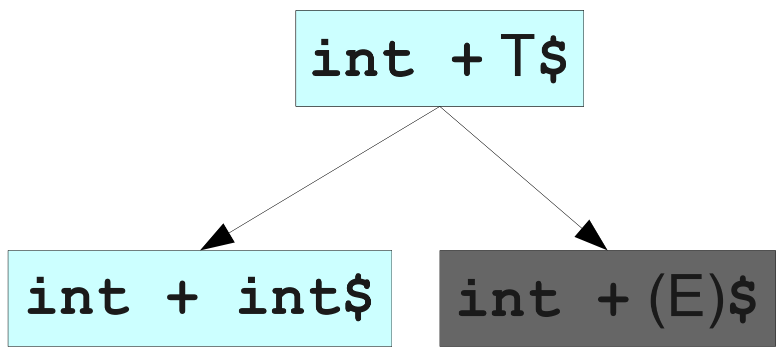


$S \rightarrow E\$$
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow$ **int**
$T \rightarrow (E)$

int + int$

Leftmost BFS





S → E$
E → T
E → T + E
T → int
T → (E)

int + int$

Leftmost BFS

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

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

`int + int$`

Leftmost BFS

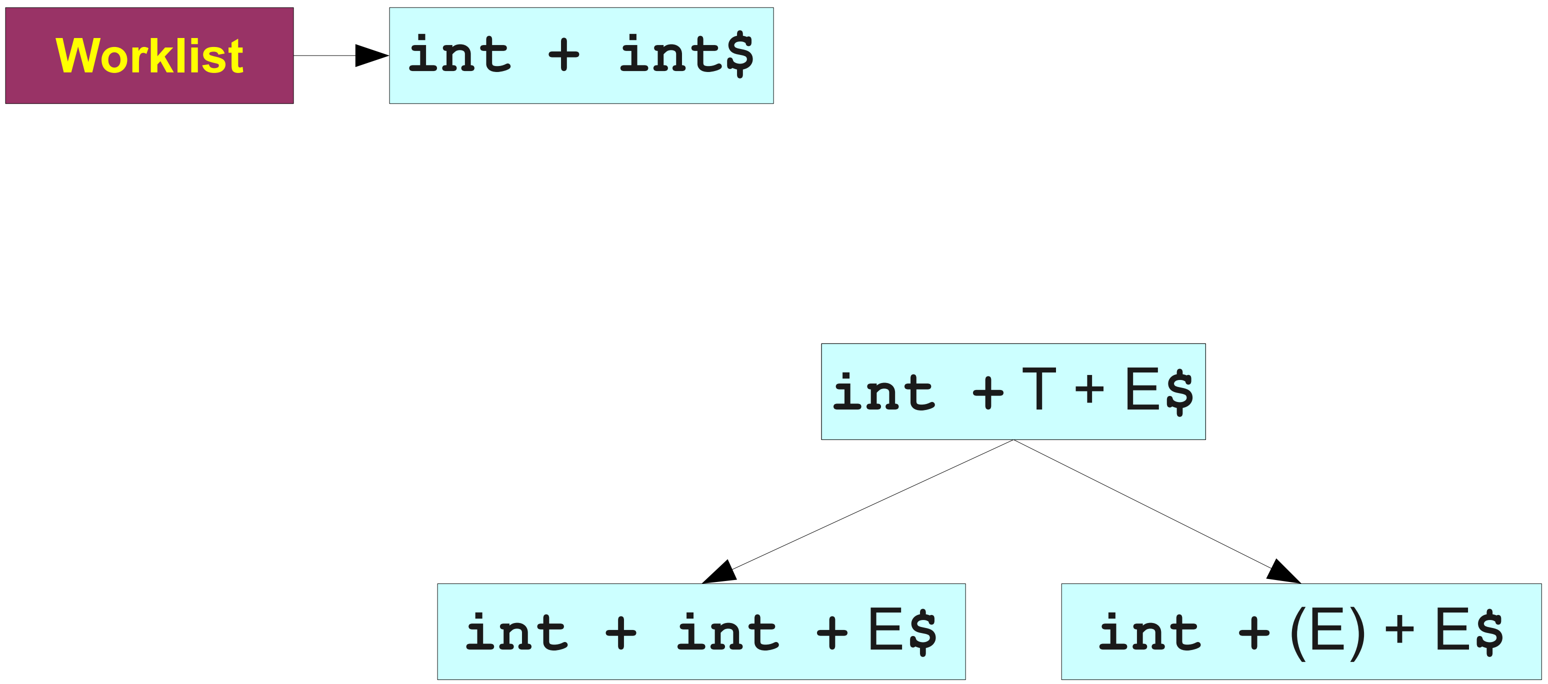
Worklist → `int + int$`

`int` + T + E`$`

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

`int + int$`

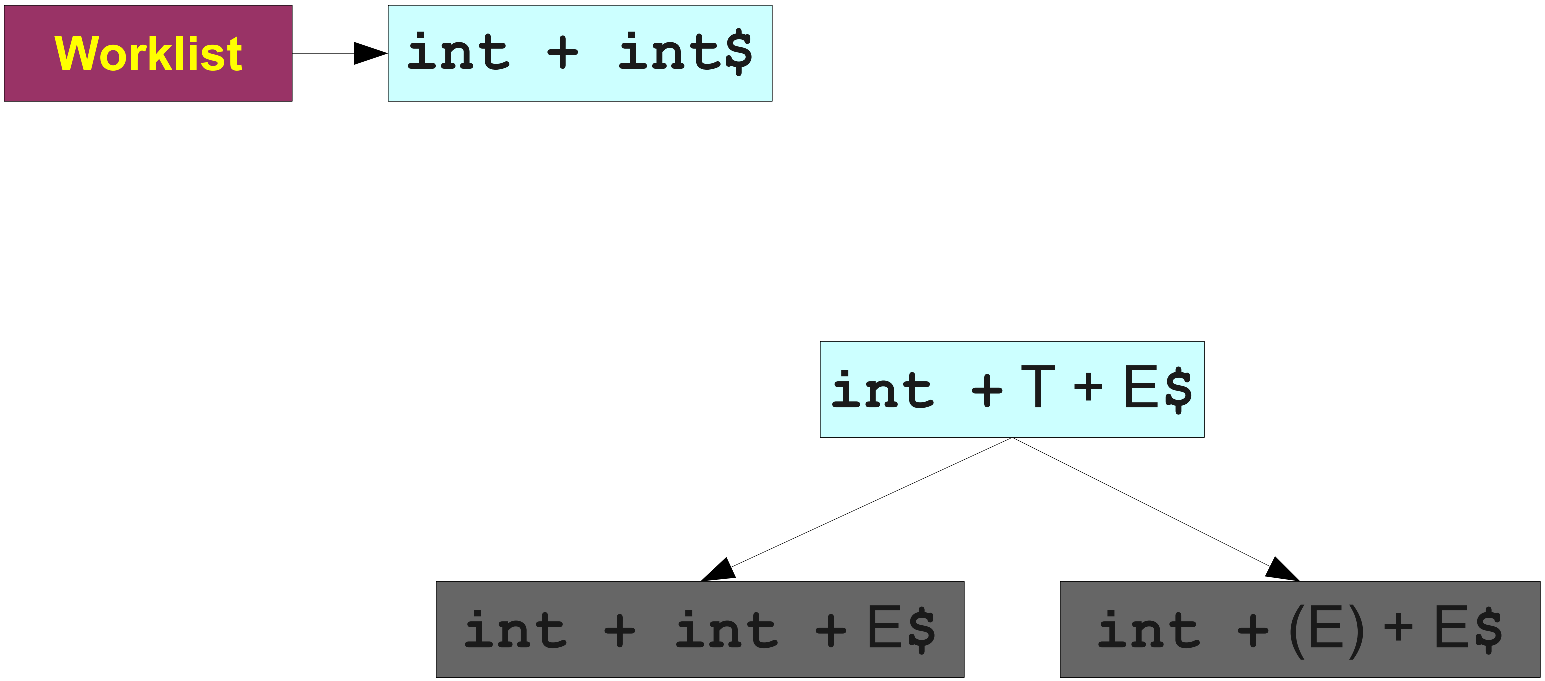
Leftmost BFS



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

int + int$

Leftmost BFS



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

int + int$

Leftmost BFS

Worklist	→	`int + int$`

$S \rightarrow E\mathtt{\$}$
$E \rightarrow T$
$E \rightarrow T + E$
$T \rightarrow \mathtt{int}$
$T \rightarrow (E)$

`int + int$`

Leftmost BFS

Worklist

`int + int$`

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

`int + int$`

Leftmost BFS

int + int$

Worklist

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

$S \rightarrow A\texttt{\$}$

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

Leftmost BFS Has Problems

Worklist

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

`caaaaaaaaaa$`

Leftmost BFS Has Problems

Worklist → S

$S \rightarrow A\$$

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

`caaaaaaaaaa$`

Leftmost BFS Has Problems

Worklist

S

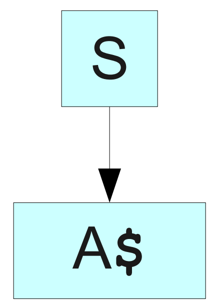
$S \rightarrow A\texttt{\$}$

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

`caaaaaaaaaa$`

Leftmost BFS Has Problems

Worklist

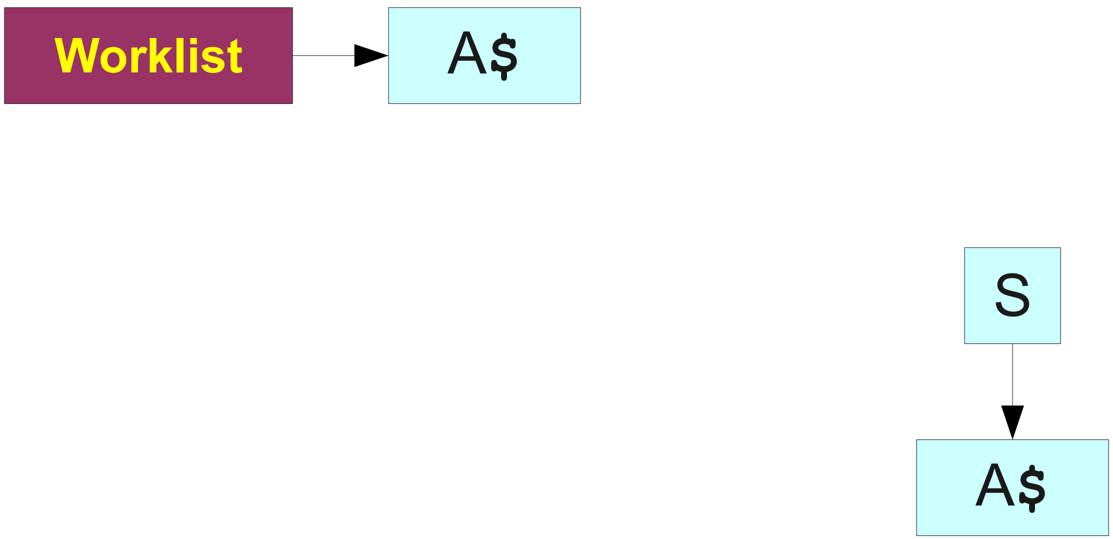


$S \rightarrow A\mathtt{\$}$

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

`caaaaaaaaa$`

Leftmost BFS Has Problems

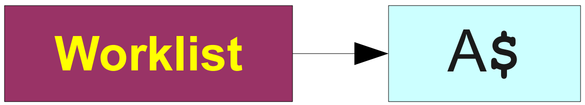


$S \rightarrow A\mathtt{\$}$

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

`caaaaaaaaaa$`

Leftmost BFS Has Problems



S → A$

A → Aa | Ab | c

caaaaaaaaa$

Leftmost BFS Has Problems

Worklist

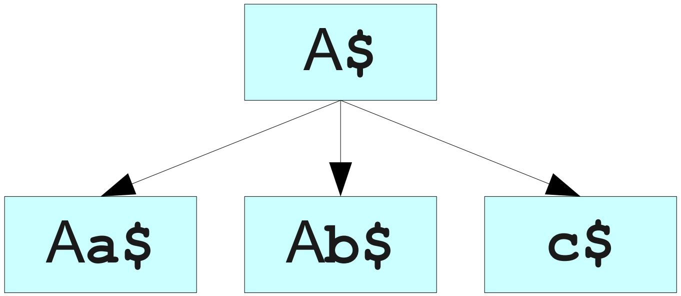
A$

S → A$
A → Aa | Ab | c

caaaaaaaaa$

Leftmost BFS Has Problems

Worklist



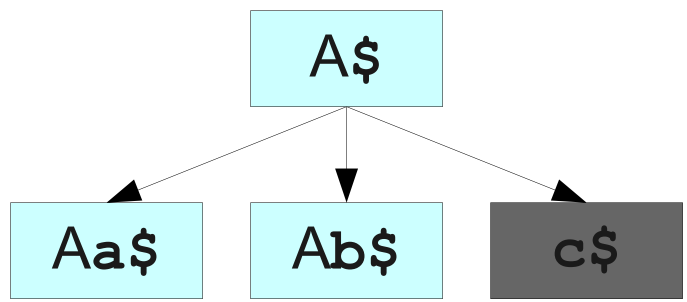
$S \rightarrow A\texttt{\$}$

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

caaaaaaaaa$

Leftmost BFS Has Problems

Worklist

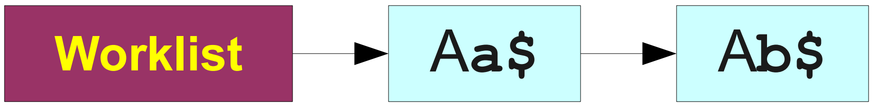


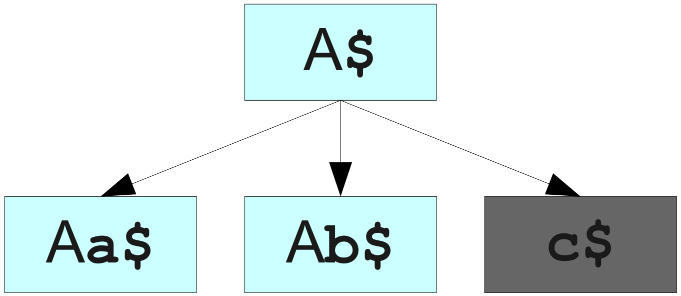
$S \rightarrow A\$$

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

caaaaaaaaa$

Leftmost BFS Has Problems





S → A$
A → Aa | Ab | c

caaaaaaaaa$

Leftmost BFS Has Problems

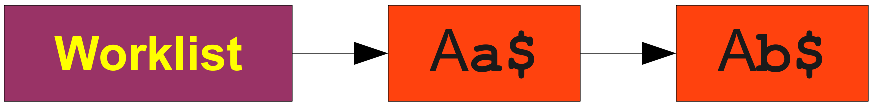
Worklist → Aa$ → Ab$

$S \rightarrow A\$$

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

caaaaaaaaa$

Leftmost BFS Has Problems

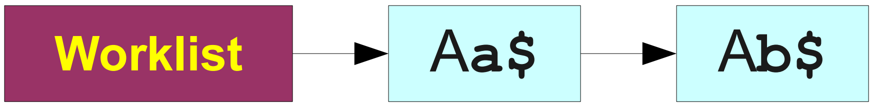


$S \rightarrow A\$$

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

caaaaaaaaa$

Leftmost BFS Has Problems



$S \rightarrow A\$$

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

caaaaaaaaaa$

Leftmost BFS Has Problems

Worklist → Ab$

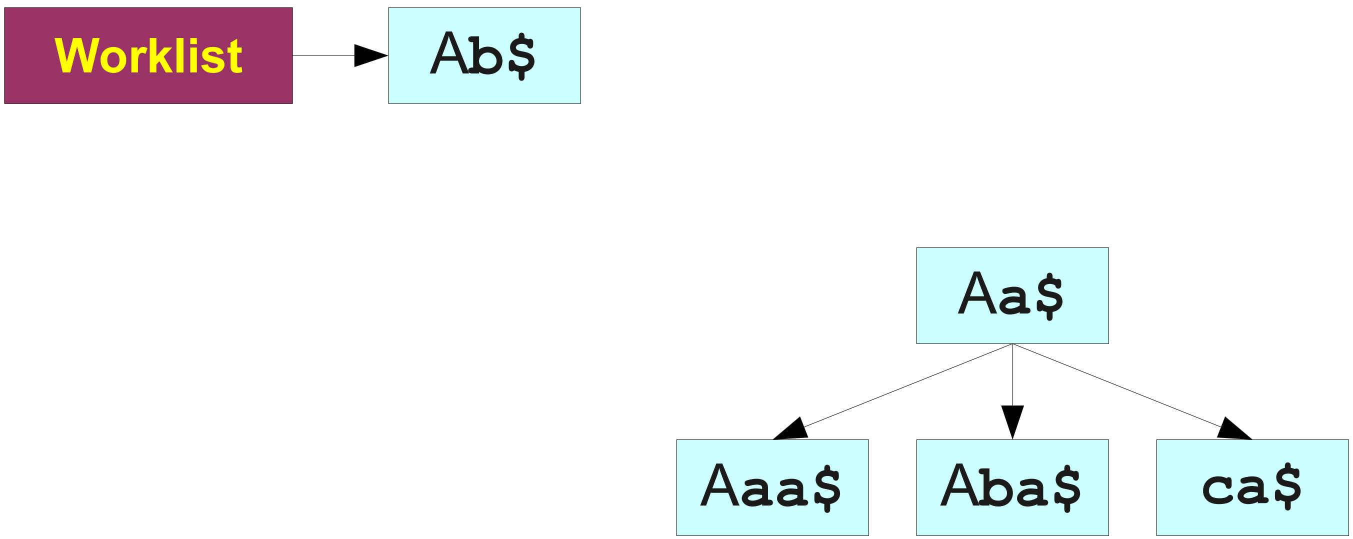
Aa$

$S \rightarrow$ A$

$A \rightarrow$ Aa | Ab | c

caaaaaaaaa$

Leftmost BFS Has Problems

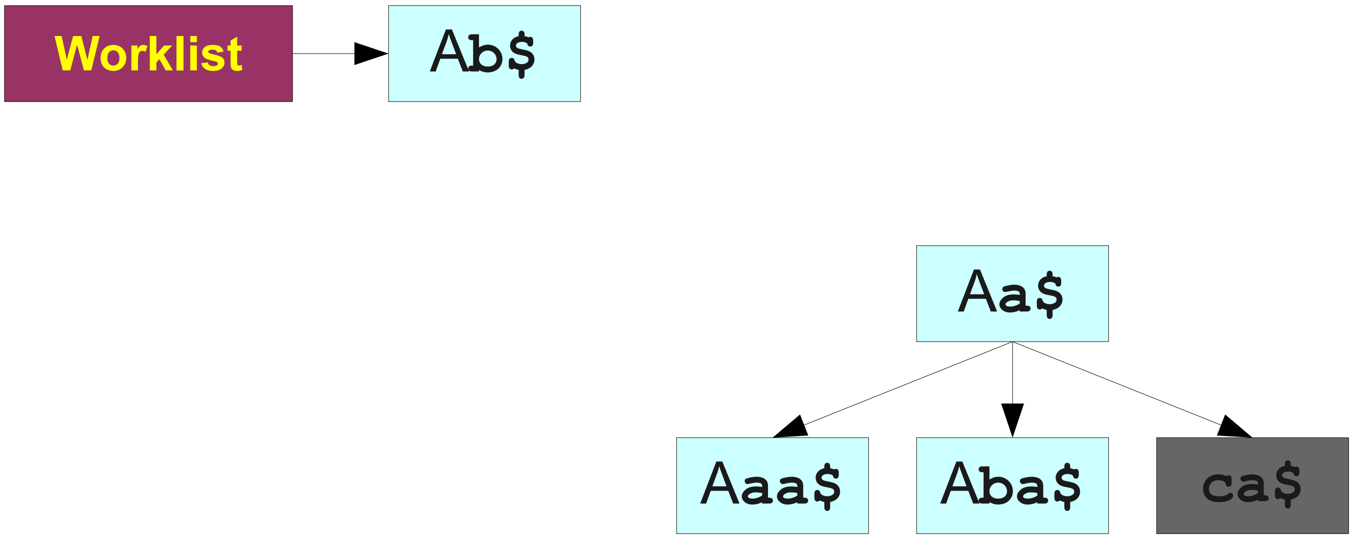


$S \rightarrow A\$$

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

caaaaaaaaa$

Leftmost BFS Has Problems

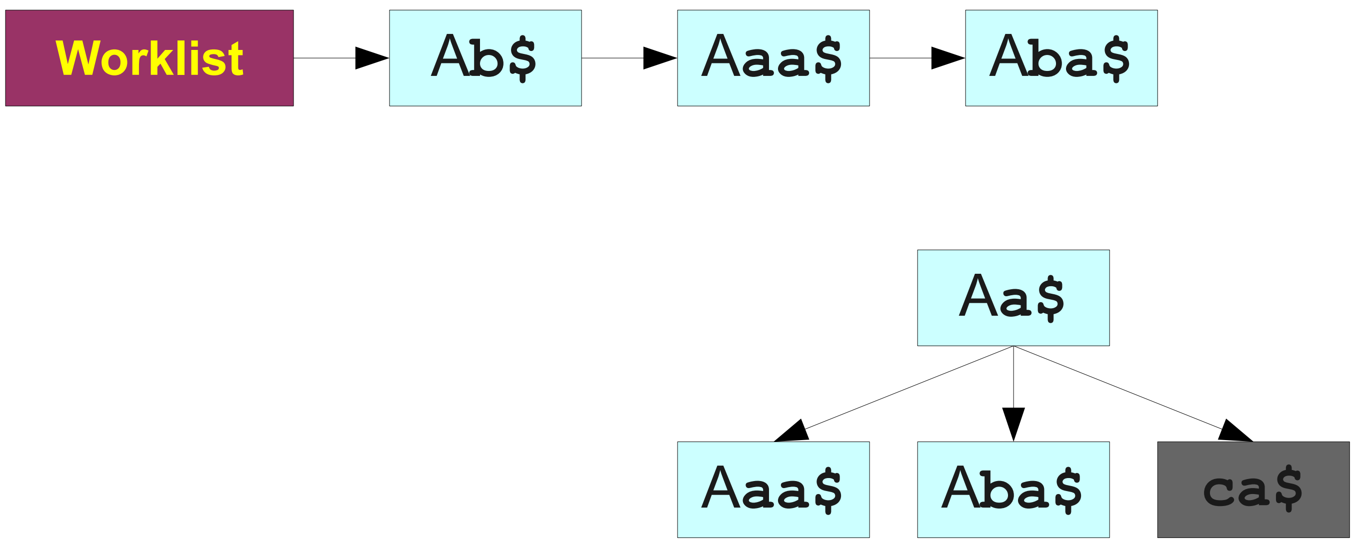


$S \rightarrow$ A$

$A \rightarrow$ Aa | Ab | c

caaaaaaaaa$

Leftmost BFS Has Problems



$S \rightarrow$ A$

$A \rightarrow$ Aa | Ab | c

caaaaaaaaa$

Leftmost BFS Has Problems

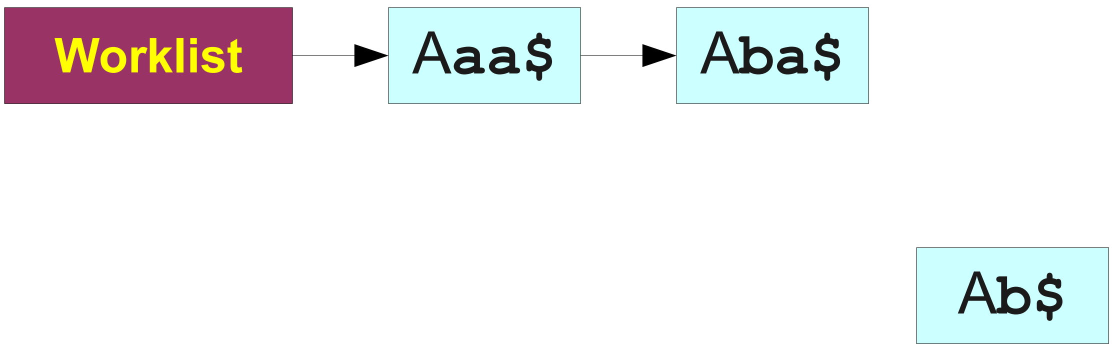
Worklist → `Ab$` → `Aaa$` → `Aba$`

$S \rightarrow A\texttt{\$}$

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

`caaaaaaaaaa$`

Leftmost BFS Has Problems

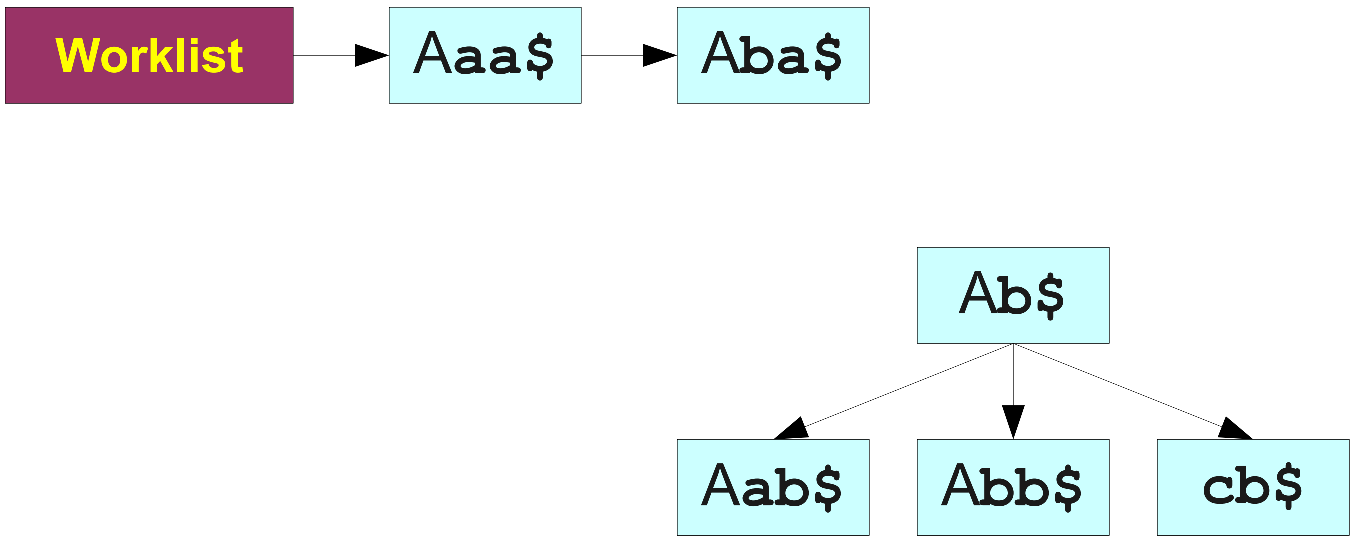


$S \to A\$$

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

caaaaaaaaaa$

Leftmost BFS Has Problems

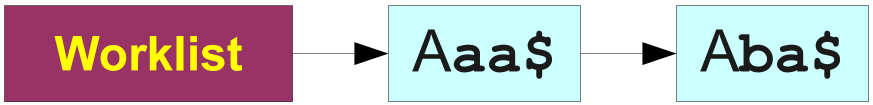


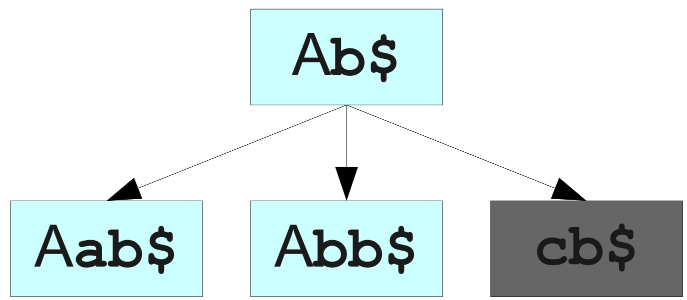
S → A$

A → Aa | Ab | c

caaaaaaaaa$

Leftmost BFS Has Problems



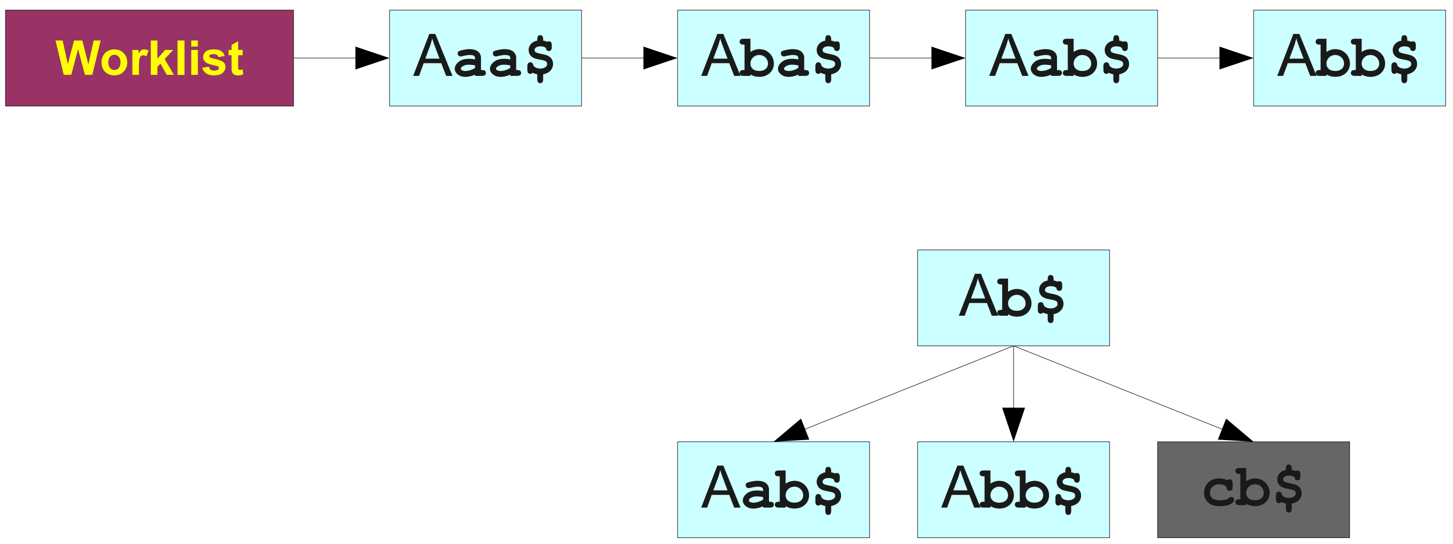


S → A$

A → Aa | Ab | c

caaaaaaaaa$

Leftmost BFS Has Problems

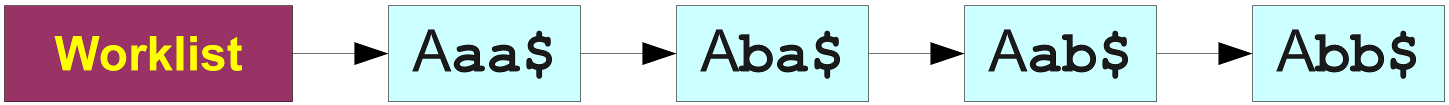


S → A$

A → Aa | Ab | c

caaaaaaaaa$

Leftmost BFS Has Problems

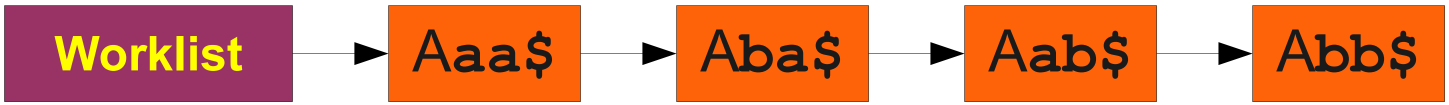


S → A$

A → Aa | Ab | c

caaaaaaaaa$

Leftmost BFS Has Problems



$S \rightarrow A\$$

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

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

$S \rightarrow E\mathtt{\$}$
$E \rightarrow T$
$E \rightarrow T\ \mathtt{+}\ E$
$T \rightarrow \mathtt{int}$
$T \rightarrow \mathtt{(}E\mathtt{)}$

Leftmost DFS

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

int + int$

Leftmost DFS

S

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

int + int$

Leftmost DFS

S
E**$**

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

int + int$

Leftmost DFS

S
E$
T$

$S \rightarrow E\$$

$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow$ **int**

$T \rightarrow$ (E)

int + int$

Leftmost DFS

S
E**$**
T**$**
int$

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

int + int$

Leftmost DFS

S
E`$`
T`$`
`int$`

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

`int + int$`

Leftmost DFS

S
E**$**
T**$**

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

`int + int$`

Leftmost DFS

S
E**$**
T**$**
(E**) $**

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

int + int$

Leftmost DFS

S
E**$**
T**$**
(E**) $**

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

int + int$

Leftmost DFS

S
E**$**
T**$**

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

int + int$

Leftmost DFS

S
E$
T$

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

int + int$

Leftmost DFS

S
E$

$S \rightarrow E\mathtt{\$}$
$E \rightarrow T$
$E \rightarrow T\ \mathtt{+}\ E$
$T \rightarrow \mathtt{int}$
$T \rightarrow \mathtt{(}E\mathtt{)}$

`int + int$`

Leftmost DFS

S
E**$**
T **+** E**$**

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

int + int$

Leftmost DFS

S
E**$**
T **+** E**$**
int **+** E**$**

$S \rightarrow E\$$

$E \rightarrow T$

$E \rightarrow T + E$

$T \rightarrow$ **int**

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

int + int$

Leftmost DFS

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

S
E$
T + E$
int + E$
int + T$

int + int$

Leftmost DFS

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

S
E$
T + E$
int + E$
int + T$
int + int$

int + int$

Leftmost DFS

S
E$
T + E$
int + E$
int + T$
int + int$

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

int + int$

Problems with Leftmost DFS

Problems with Leftmost DFS

$S \rightarrow A\mathtt{\$}$
$A \rightarrow A\mathtt{a} \mid \mathtt{c}$

Problems with Leftmost DFS

$S \rightarrow A\mathtt{\$}$
$A \rightarrow A\mathtt{a} \mid \mathtt{c}$

c$

Problems with Leftmost DFS

S

$S \rightarrow A\texttt{\$}$

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

c$

Problems with Leftmost DFS

S
A**$**

S → A**$**

A → A**a** | **c**

c$

Problems with Leftmost DFS

S
A$
Aa$

$S \rightarrow A\mathtt{\$}$

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

c$

Problems with Leftmost DFS

S
A$
Aa$
Aaa$

S → A$

A → Aa | c

c$

Problems with Leftmost DFS

$S \rightarrow A\$$
$A \rightarrow A\mathtt{a} \mid \mathtt{c}$

S
A$
Aa$
Aaa$
Aaaa$

c$

Problems with Leftmost DFS

$S \rightarrow A\mathtt{\$}$
$A \rightarrow A\mathtt{a} \mid \mathtt{c}$

S
A**$**
A**a$**
A**aa$**
A**aaa$**
A**aaaa$**

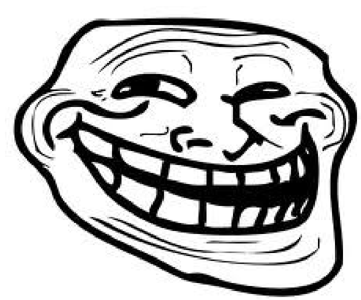
c$

Problems with Leftmost DFS

$S \rightarrow A\$$
$A \rightarrow A\mathbf{a} \mid \mathbf{c}$

S
A$
Aa$
Aaa$
Aaaa$
Aaaaa$

Problem?



c$

Left Recursion

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

  $$A \rightarrow^* Aw$$

  For some string of terminals and nonterminals w.
- Leftmost DFS may fail on left-recursive grammars.
- Fortunately, in many cases it is possible to eliminate left recursion (more on this later).

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: **Lookahead tokens**.
- Use knowledge of what terminals need to be matched to pick which production to use.

Our First 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

$S \to E\texttt{\$}$

$E \to \texttt{int}$

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

$Op \to \texttt{+}$

$Op \to \texttt{*}$

LL(1) Parse Tables

$S \to E\$$

$E \to$ **int**

$E \to$ (E Op E)

$Op \to +$

$Op \to *$

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

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

S	**(int + (int * int))$**

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	**(**	**)**	**+**	*****	**$**
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

S	`(int + (int * int))$`

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	`int`	`(`	`)`	`+`	`*`	`$`
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

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

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → -

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

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

Predictive Top-Down Parsing

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

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

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

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

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

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

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

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

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

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

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	**(**	**)**	**+**	*****	**$**
S	1	1				
E	2	3				
Op				4	5	

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

Predictive Top-Down Parsing

1. S → E$
2. E → **int**
3. E → (E Op E)
4. Op → **+**
5. Op → **-**

S	(int + (int * int))$
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	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
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))$

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	**(**	**)**	**+**	*****	**$**
S	1	1				
E	2	3				
Op				4	5	

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

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

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

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

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

Predictive Top-Down Parsing

1. S → E$
2. E → **int**
3. E → (E Op E)
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
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))$

Predictive Top-Down Parsing

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

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
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))$

Predictive Top-Down Parsing

1. S → E`$`
2. E → **`int`**
3. E → `(`E Op E`)`
4. Op → `+`
5. Op → `-`

	`int`	`(`	`)`	`+`	`*`	`$`
S	1	1				
E	2	3				
Op				4	5	

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

Predictive Top-Down Parsing

1. S → E$
2. E → **int**
3. E → (E Op E)
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
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))$

Predictive Top-Down Parsing

1. S → E$
2. E → **int**
3. E → (E Op E)
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
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))$

Predictive Top-Down Parsing

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

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
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))$

Predictive Top-Down Parsing

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
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))$

Predictive Top-Down Parsing

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → -

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
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))$

Predictive Top-Down Parsing

1. S → E`$`
2. E → `int`
3. E → `(`E Op E`)`
4. Op → `+`
5. Op → `-`

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
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))$
))$	))$

Predictive Top-Down Parsing

1. S → E`$`
2. E → **`int`**
3. E → `(`E Op E`)`
4. Op → `+`
5. Op → `-`

	`int`	`(`	`)`	`+`	`*`	`$`
S	1	1				
E	2	3				
Op				4	5	

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

Predictive Top-Down Parsing

1. S → E$
2. E → **int**
3. E → (E Op E)
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int + (int * int))$
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))$
))$	))$
)$	)$
$	$

Predictive Top-Down Parsing

1. S → E**$**
2. E → **`int`**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op →

	int	(	)
S	1	1	
E	2	3	
Op			

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

Error Detection

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection

S	int + int$

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection

S	`int + int$`

1. S → E`$`
2. E → `int`
3. E → `(`E Op E`)`
4. Op → `+`
5. Op → `-`

	`int`	`(`	`)`	`+`	`*`	`$`
S	1	1				
E	2	3				
Op				4	5	

Error Detection

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → -

S		int + int$
E$		int + int$

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection

1. S → E`$`
2. E → `int`
3. E → `(`E Op E`)`
4. Op → `+`
5. Op → `-`

	`int`	`(`	`)`	`+`	`*`	`$`
S	1	1				
E	2	3				
Op				4	5	

S	`int + int$`
E`$`	`int + int$`

Error Detection

S	int + int$
E$	int + int$
int $	int + int$

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

S	int + int$
E$	int + int$
int $	int + int$
$	+ int$

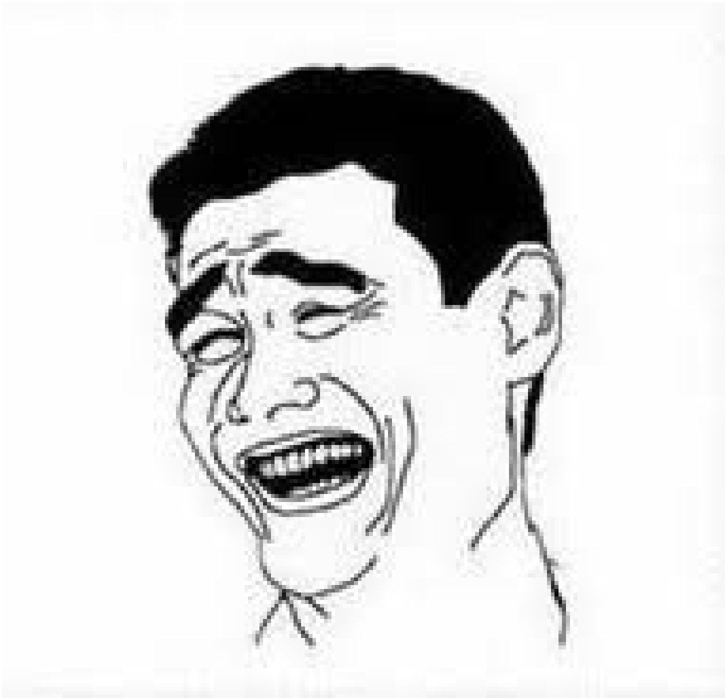
	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection

1. S → E$
2. E → **int**
3. E → (E Op E)
4. Op → +
5. Op → -

S	int + int$
E$	int + int$
int $	int + int$
$	+ int$

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	



Error Detection II

1. S → E`$`
2. E → `int`
3. E → `(`E Op E`)`
4. Op → `+`
5. Op → `-`

	`int`	`(`	`)`	`+`	`*`	`$`
S	1	1				
E	2	3				
Op				4	5	

Error Detection II

S	(int (int))$

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection II

S	(int (int))$

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → -

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection II

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

S	(int (int))$
E$	(int (int))$

Error Detection II

S	`(int (int))$`
E`$`	`(int (int))$`

1. S → E`$`
2. E → **`int`**
3. E → `(`E Op E`)`
4. Op → **`+`**
5. Op → **`-`**

	`int`	`(`	`)`	`+`	`*`	`$`
S	1	1				
E	2	3				
Op				4	5	

Error Detection II

1. S → E$
2. E → int
3. E → (E Op E)
4. Op → +
5. Op → -

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

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

Error Detection II

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

	int	**(**	**)**	**+**	*****	**$**
S	1	1				
E	2	3				
Op				4	5	

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

Error Detection II

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

S	**(int (int))$**
E**$**	**(int (int))$**
(E Op E**)$**	**(int (int))$**
E Op E**)$**	**int (int))$**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection II

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

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

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection II

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

S	**(int (int))$**
E**$**	**(int (int))$**
(E Op E**)$**	**(int (int))$**
E Op E**)$**	**int (int))$**
int Op E**)$**	**int (int))$**
Op E**)$**	**(int))$**

	int	**(**	**)**	**+**	*****	**$**
S	1	1				
E	2	3				
Op				4	5	

Error Detection II

1. S → E**$**
2. E → **int**
3. E → **(**E Op E**)**
4. Op → **+**
5. Op → **-**

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

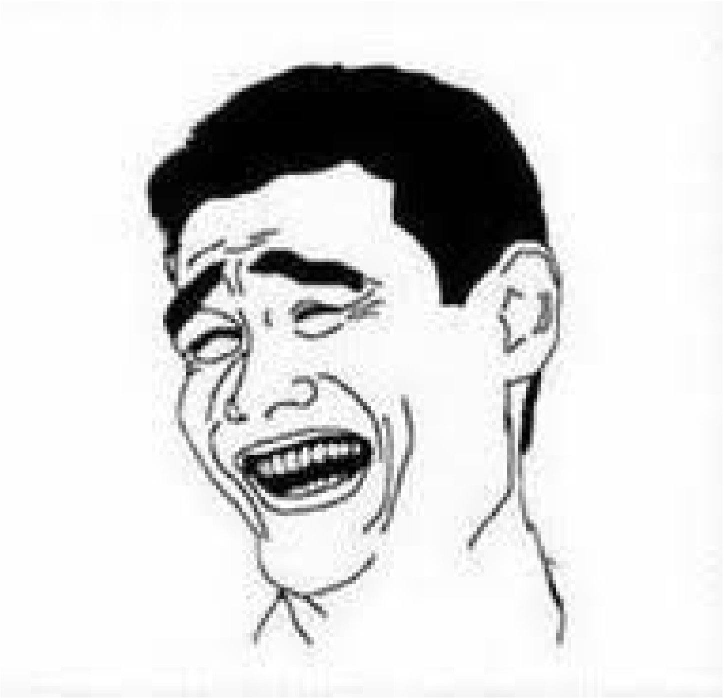
	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

Error Detection II

1. S → E$
2. E → **int**
3. E → (E Op E)
4. Op → **+**
5. Op → **-**

	int	(	)	+	*	$
S	1	1				
E	2	3				
Op				4	5	

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



The LL(1) Algorithm

- Given an LL(1) parsing table T and input w:
- Initialize a stack containing S.
- Repeat until the stack is just **$**:
  - Let the next character of *w* be *c*.
  - If the top of the stack is a terminal *t*:
    - If *c* and *t* don't match, report an error.
    - Otherwise consume the character *c* and pop *t* from the stack.
  - Otherwise, the top of the stack is a nonterminal *A*:
    - If T[A, *c*] is undefined, report an error.
    - Replace the top of the stack with T[A, c].

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)

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												
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												
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	

Filling in Table Entries

- Intuition: The next character should uniquely identify a production, so we should pick a production that ultimately starts with that character.
- T[A, c] should be a production

  $$A \to A_1\ A_2\ \ldots\ A_n$$

  if A_1 ultimately derives something starting with c.
- More rigorously:

  $$T[A, c] = A_1\ A_2\ \ldots\ A_n \text{ if } A \to A_1\ A_2\ \ldots\ A_n \text{ and } A_1 \to^* c.$$

FIRST Sets

- In what follows, **assume there are no ε-productions** in the grammar (we'll relax that condition later.)
- We can determine what tokens a nonterminal A can derive by using FIRST sets.
- Definition: FIRST(A) = $\{ t \mid A \rightarrow^* t v \}$
  - The set of tokens that appear first in some production of A.
- Set T[A, c] = $A_1\ A_2\ \dots\ A_n$ if c in FIRST(A_1)

Computing FIRST Sets

- Initially, for all nonterminals A, set

  FIRST(A) = { t | A → t*w* }

- Then, for each A → B*w,* iteratively compute

  FIRST(A) = FIRST(A) ∪ FIRST(B)

- When no changes occur, the resulting sets are the FIRST sets.
- 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`
	`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`
	`++`	
	`--`	

Iterative FIRST Computations

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

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

TERM → `id`
| `constant`

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

Iterative FIRST Computations

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

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

TERM → **id**
| **constant**

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

Iterative FIRST Computations

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

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

TERM → **id**
  | **constant**

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

Iterative FIRST Computations

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

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

TERM → `id`
  | `constant`

STMT	EXPR	TERM
`if`	`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`	**`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`	`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** **++** **--** **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)

STMT	EXPR	TERM

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

From FIRST Sets to LL(1) Tables

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

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

	if	**then**	**while**	**do**	**zero?**	**not**	**++**	**--**	**→**	**id**	**const**	**;**
STMT												
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		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					5	6	7	8				
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												

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	

Expanding our 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;
```

Expanding our Grammar

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

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

TERM → **id**
  | **constant**

BLOCK → STMT
  | { STMTS }

STMTS → STMT STMTS
  | ε

```
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**
  | `while` EXPR `do` **BLOCK**
  | EXPR `;`

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

TERM → `id`
  | `constant`

BLOCK → STMT
  | { STMTS }

STMTS → STMT STMTS
  | ε

```
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
 | **`while`** EXPR **`do`** BLOCK
 | EXPR **`;`**

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

TERM → **`id`**
 | **`constant`**

BLOCK → STMT
 | { STMTS }

STMTS → STMT STMTS
 | ε

```
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
  | **while** EXPR **do** BLOCK
  | EXPR **;**

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

TERM → **id**
  | **constant**

BLOCK → STMT
  | { STMTS }

STMTS → STMT STMTS
  | ε

```
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

Number → Sign Digits
Digits → Digit | Digit Digits
Digit → **0** | **1** | **2** | … | **9**
Sign → **+** | **-** | ε

FIRST Sets with ε

Number → Sign Digits
Digits → Digit | Digit Digits
Digit → **0** | **1** | **2** | … | **9**
Sign → **+** | **-** | ε

Number	Digits	Digit	Sign

FIRST Sets with ε

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

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

FIRST Sets with ε

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

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

FIRST Sets with ε

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

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

FIRST Sets with ε

Number → Sign Digits
Digits → Digit | Digit Digits
Digit → **0** | **1** | **2** | … | **9**
Sign → **+** | **-** | ε

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

FIRST Sets with ε

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

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

FIRST Sets with ε

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

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

FIRST Sets with ε

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

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

FIRST Sets with ε

Number → Sign **Digits**
Digits → Digit | Digit Digits
Digit → **0** | **1** | **2** | … | **9**
Sign → **+** | **-** | ε

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

FIRST Sets with ε

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

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

Interestingly, this grammar isn't LL(1).

Updated FIRST Set Computation

- Idea: Want FIRST(A) to contain all possible first terminals derivable from A.
- If $A \rightarrow A_1 A_2 \ldots A_n$ and A_1 cannot produce ε, FIRST(A) contains FIRST(A_1)
- If $A \rightarrow A_1 A_2 \ldots A_n$ and A_1 **can** produce ε, FIRST(A) contains FIRST(A_1) and FIRST(A_2)
- If $A \rightarrow A_1 A_2 \ldots A_n$ and *all* A_i can produce ε, FIRST(A) contains all FIRST(A_i) and ε.

LL(1) Tables with ε-Productions

LL(1) Tables with ε-Productions

$S \rightarrow A\$$ (1)
$A \rightarrow \varepsilon$ (2)

LL(1) Tables with ε-Productions

$S \to A\$$ (1)
$A \to \varepsilon$ (2)

S	A
$	ε

LL(1) Tables with ε-Productions

$S \to A\$$ (1)
$A \to \varepsilon$ (2)

S	A
$	ε

	$
S	
A	

LL(1) Tables with ε-Productions

$S \rightarrow \textcolor{red}{\mathbf{A}}\$$ (1)
$A \rightarrow \varepsilon$ (2)

S	A
$	ε

	$
S	
A	

LL(1) Tables with ε-Productions

$S \rightarrow A\$$ (1)

$A \rightarrow \varepsilon$ (2)

S	A
$	ε

	$
S	
A	

LL(1) Tables with ε-Productions

$S \rightarrow A\$$ (1)
$A \rightarrow \varepsilon$ (2)

S	A
\$	ε

	\$
S	1
A	

LL(1) Tables with ε-Productions

$S \rightarrow A\$$ (1)
$A \rightarrow \varepsilon$ (2)

S	A
$	ε

	$
S	1
A	

LL(1) Tables with ε-Productions

$S \rightarrow A\$$ (1)
$A \rightarrow \varepsilon$ (2)

S	A
$	ε

	$
S	1
A	

LL(1) Tables with ε-Productions

$S \to A\$$ (1)
$A \to \varepsilon$ (2)

S	A
$	ε

	$
S	1
A	2

LL(1) Tables with ε-Productions

$S \rightarrow A\$$ (1)
$A \rightarrow \varepsilon$ (2)

S	A
$	ε

	$
S	1
A	2

LL(1) Tables with ε-Productions

$S \rightarrow A\$$ (1)
$A \rightarrow \varepsilon$ (2)

S	A
$	ε

	$
S	1
A	2

S	$

LL(1) Tables with ε-Productions

$S \rightarrow A\$$ (1)
$A \rightarrow \varepsilon$ (2)

S	A
$	ε

	$
S	1
A	2

S	$

LL(1) Tables with ε-Productions

$S \rightarrow A\$$ (1)
$A \rightarrow \varepsilon$ (2)

S	A
$	ε

	$
S	1
A	2

S	$
A$	$

LL(1) Tables with ε-Productions

$S \to A\$$ (1)
$A \to \varepsilon$ (2)

S	A
$	ε

	$
S	1
A	2

S	$
A$	$

LL(1) Tables with ε-Productions

$S \rightarrow A\$$ (1)
$A \rightarrow \varepsilon$ (2)

S	A
$	ε

	$
S	1
A	2

S	$
A$	$
$	$

LL(1) Tables with ε-Productions

$S \rightarrow A\$$ (1)
$A \rightarrow \varepsilon$ (2)

S	A
$	ε

	$
S	1
A	2

S	$
A$	$
$	$

FOLLOW Sets

- Intuition: Keep track of what terminals can eventually follow a nonterminal.
- Formally:

  $$\text{FOLLOW}(A) = \{ t \mid B \to^* wAtv \}$$

- Used in construction of LL(1) parse tables to determine when to use ε productions.

Computation of FOLLOW Sets

- Another fixed-point iteration.
- Initialization: For all productions $B \rightarrow wAv$, set FOLLOW(A) = FIRST(v) – {ε}
  - Note that v is a **string**, not a terminal or nonterminal; you must take ε into account.
- Iteration: For all productions $B \rightarrow wAv$ where v can derive the empty string (i.e. ε in FIRST(v)):
  - Set FOLLOW(A) = FOLLOW(A) ∪ FOLLOW(B)
- Stop when no more changes occur.

The Final LL(1) Table Algorithm

- Compute FIRST(A) and FOLLOW(A) for all nonterminals A.
- For each rule A → *w*, for each **terminal** t in FIRST(*w*), set T[A, t] = *w*.
  - Note that ε is **not** a terminal!
- For each rule A → *w* with ε in FIRST(*w*), for each t in FOLLOW(A), set T[A, t] = *w*.

Next Time

- A More Elaborate Example
- The Limits of LL(1)
- Bottom-Up Parsing
  - Shift/reduce parsing
  - LR(0)
  - LR(1)