COSC252: Programming Languages Lecture: Parsing

Jeremy Bolton, PhD

Asst Professor

GEORGETOWN UNIVERSITY

Outline

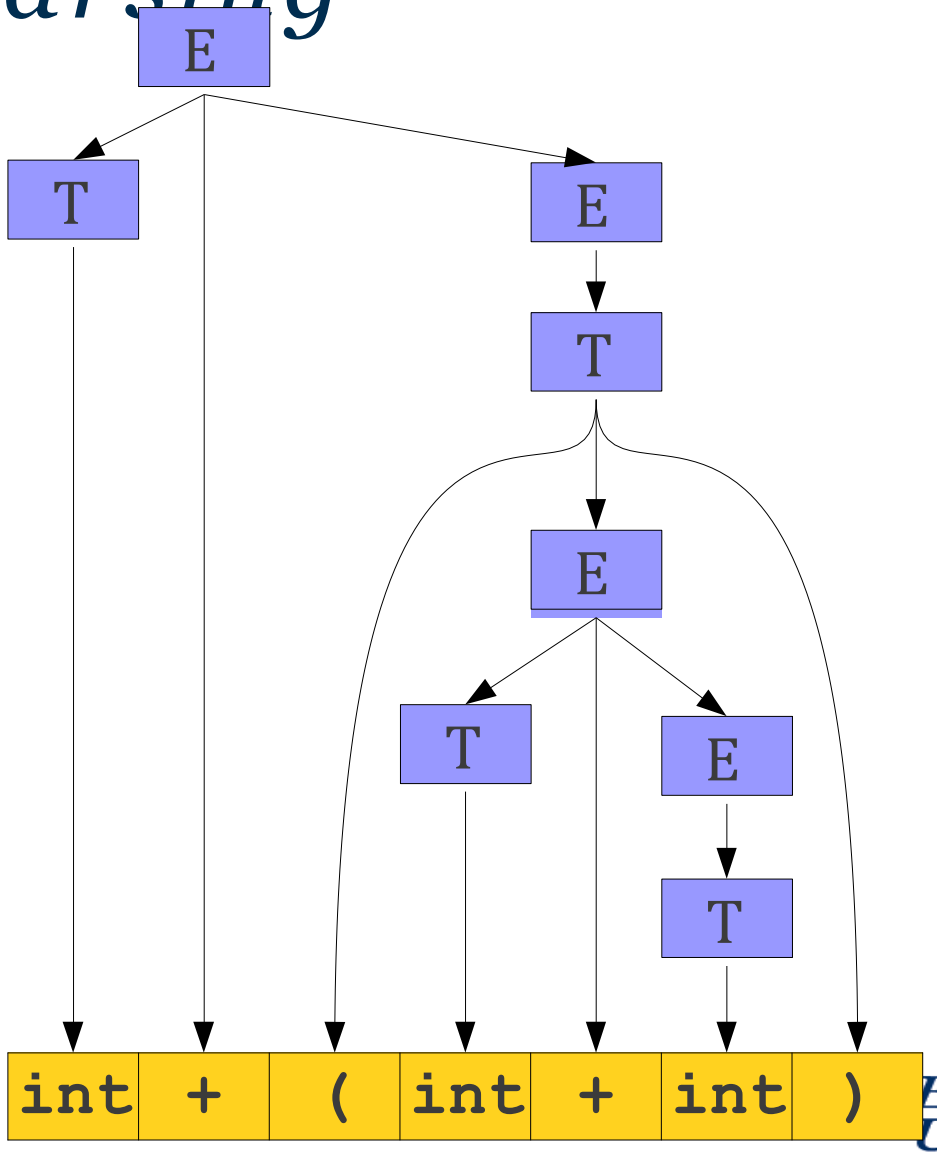
I. Parsing Intro
II. Top-Down
    I. Method
        I. Issues
        II. Fixes
    II. Recursive decent
    III. Table Implementation

Different Types of Parsing

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

Top-Down Parsing

E → T
E → T + E
T → int
T → (E)



Challenges in Top-Down Parsing

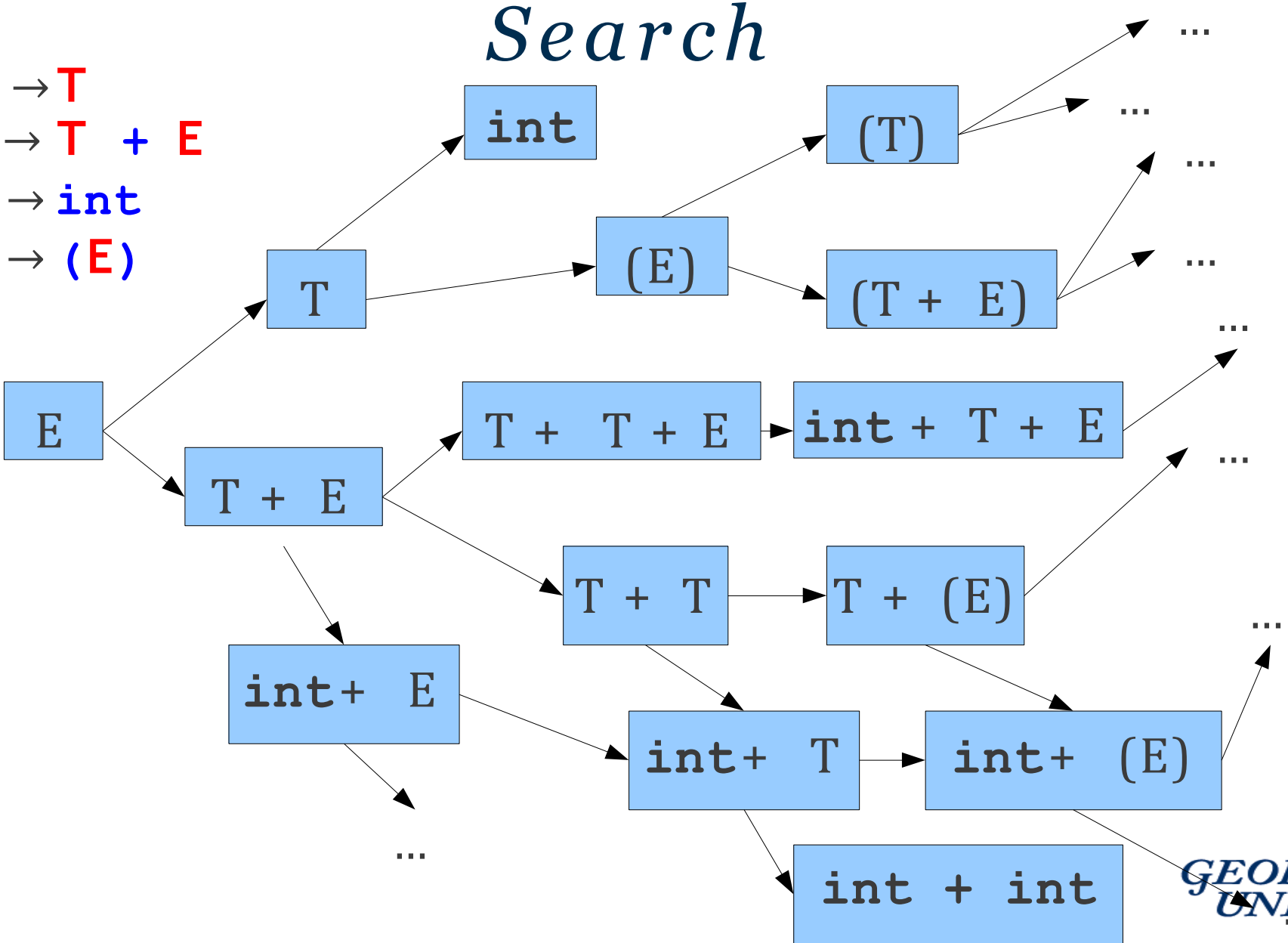
- Top-down parsing begins with virtually no information.
  - Begins with just the start symbol, which matches *every* program.
- How can we know which productions to apply?
- In general, we can't.
  - There are some grammars for which the best we can do is guess and backtrack if we're wrong.
  - If we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

Parsing as a Search

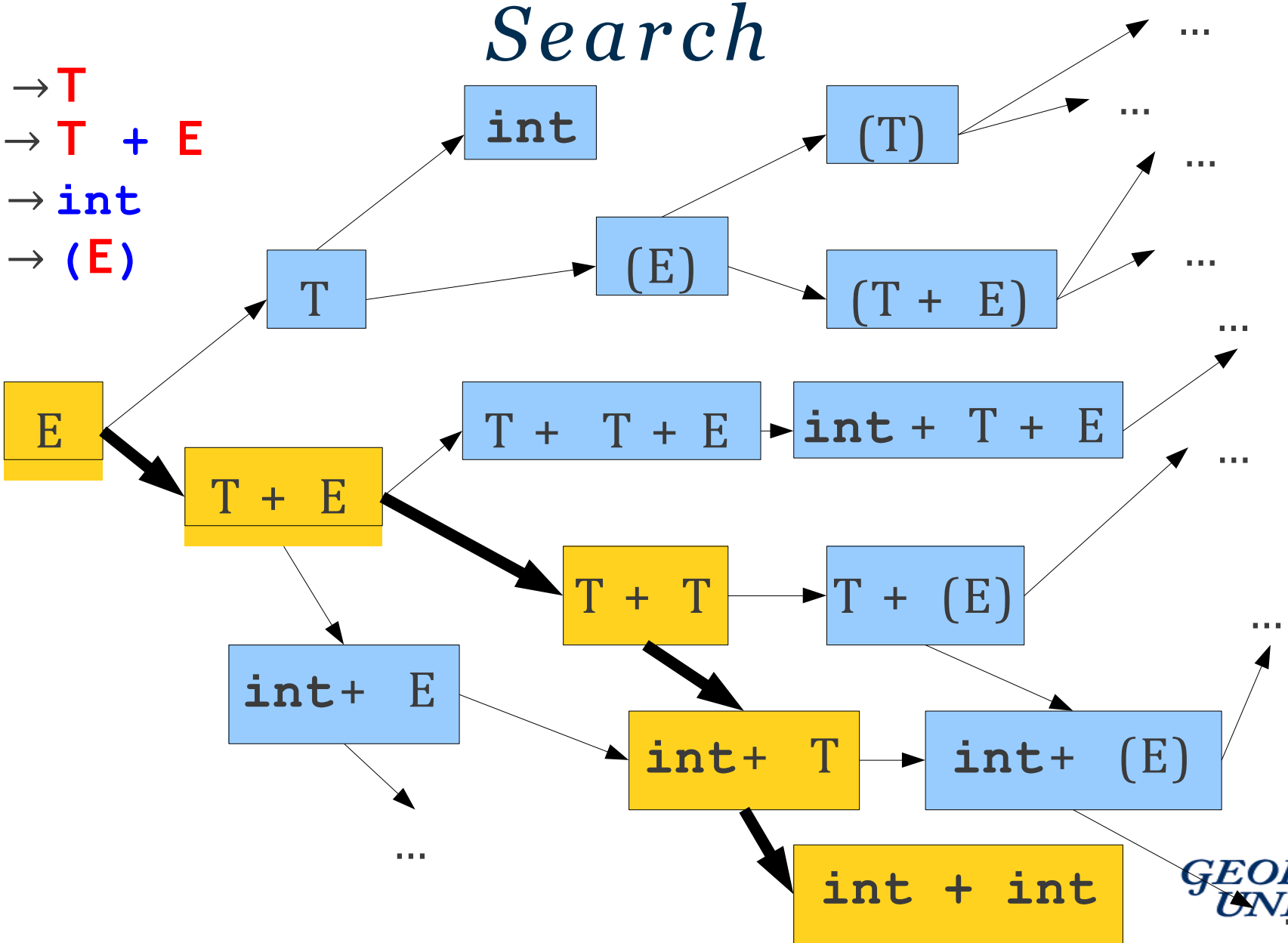
E → T
E → T + E
T → int
T → (E)



GEORGETOWN UNIVERSITY

Parsing as a Search

E → T
E → T + E
T → int
T → (E)



GEORGETOWN UNIVERSITY

Our First Top-Down Algorithm

Breadth-First Search

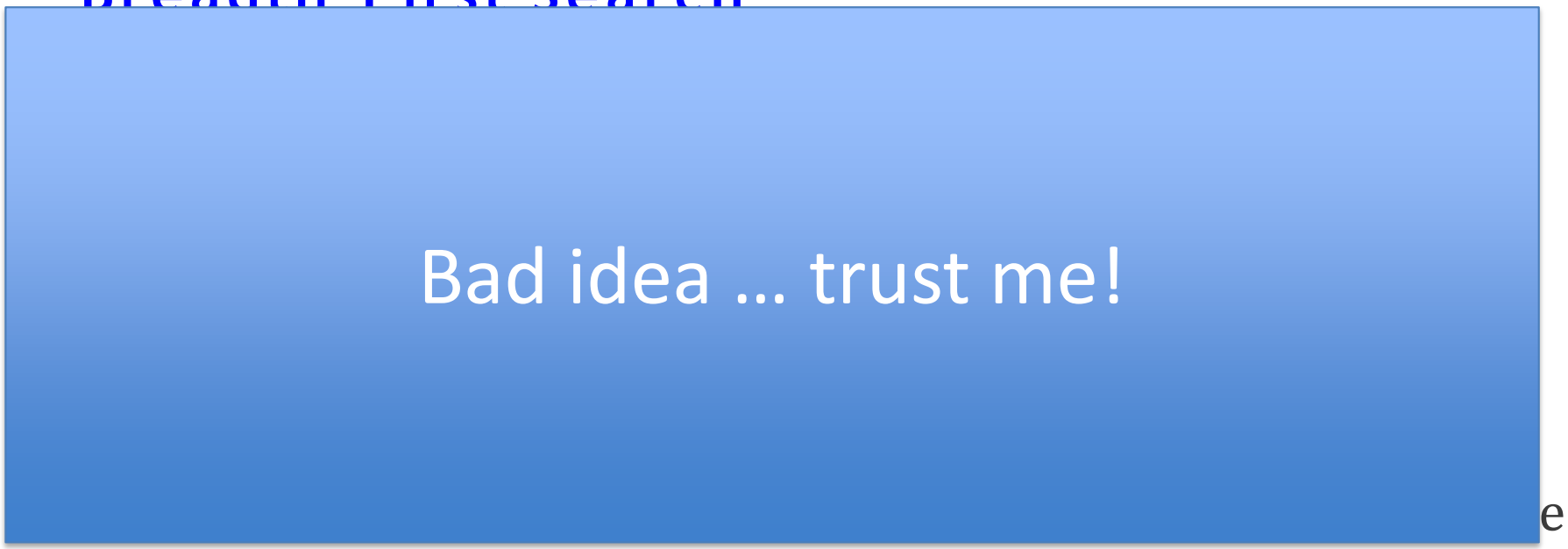
Maintain a worklist of sentential forms, initially just the start symbol **S**.

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.

Our First Top-Down Algorithm

Breadth-First Search



e
derived in one step, add that string to the worklist.

GEORGETOWN UNIVERSITY

Leftmost DFS

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

Leftmost DFS

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

Leftmost DFS

E → T
E → T + E
T → int
T → (E)

int + int

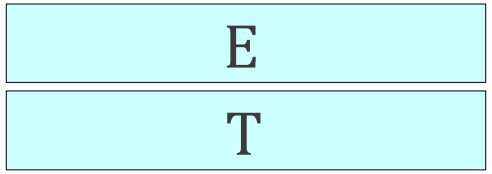
Leftmost DFS

E

E → T
E → T + E
T → int
T → (E)

int + int

Leftmost DFS

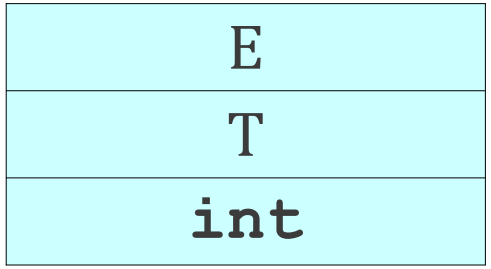


E → T
E → T + E
T → int
T → (E)

int + int

Leftmost DFS

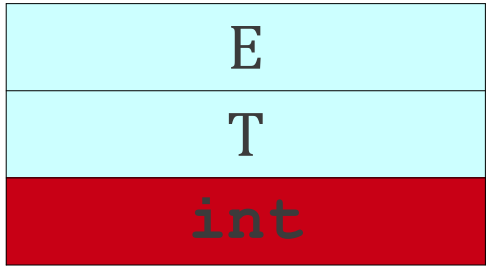
E → T
E → T + E
T → int
T → (E)



int + int

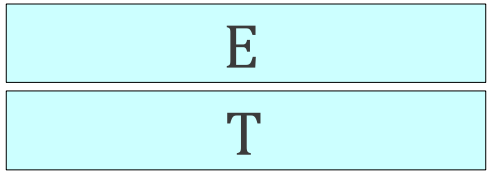
Leftmost DFS

E → T
E → T + E
T → int
T → (E)



int + int

Leftmost DFS

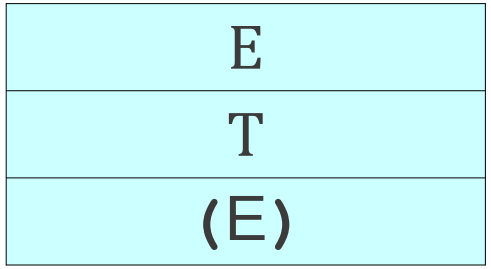


E → T
E → T + E
T → int
T → (E)

int + int

Leftmost DFS

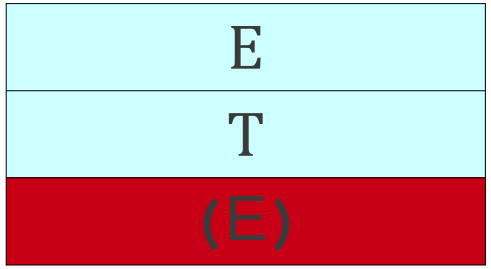
E → T
E → T + E
T → int
T → (E)



int + int

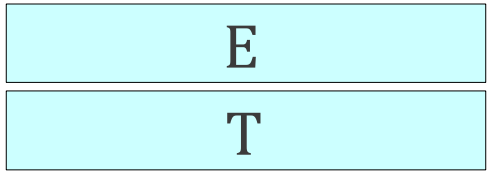
Leftmost DFS

E → T
E → T + E
T → int
T → (E)



int + int

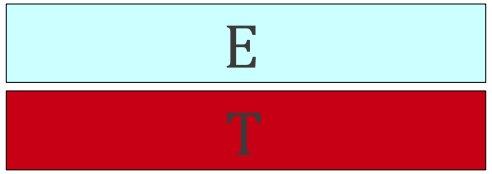
Leftmost DFS



E → T
E → T + E
T → int
T → (E)

int + int

Leftmost DFS



E → T
E → T + E
T → int
T → (E)

int + int

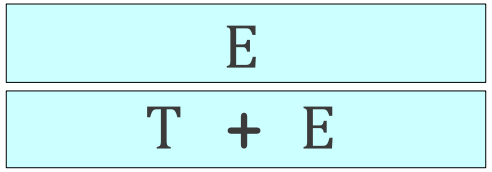
Leftmost DFS

E

E → T
E → T + E
T → int
T → (E)

int + int

Leftmost DFS



E → 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

$A \rightarrow Aa \mid c$

A
Aa
Aaa
Aaaa
Aaaaa

c

Left Recursion

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

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

  for some string ω.

- Leftmost DFS may fail on left-recursive grammars.
- Fortunately, in many cases it is possible to eliminate left recursion (see Handout 08 for details).

Notes concerning 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**.

GEORGETOWN UNIVERSITY

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	)

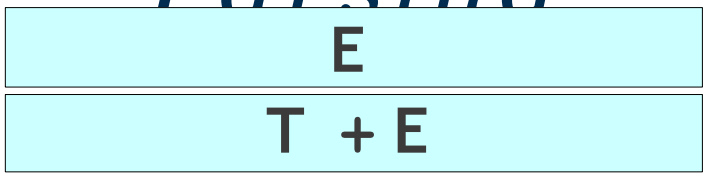
Predictive Parsing

E

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

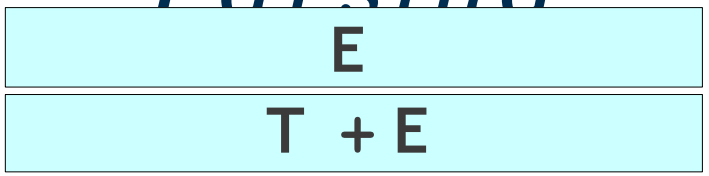
Predictive Parsing



E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

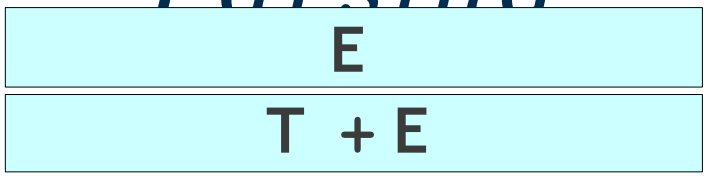
Predictive Parsing



E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing



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	)

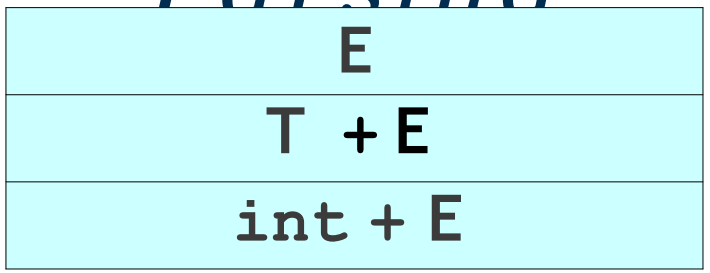
Predictive Parsing

E
T + E
int + E

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing



E → T
E → 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
int + E
int + T
int + (E)

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing

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

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing

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

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing

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

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	+	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	+	int	)

Predictive Parsing

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

E → T
E → T + E
T → int
T → (E)

int	+	(	int	+	int	)

Predictive Parsing

E → T
E → 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 → 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 → 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)

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	)

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.**

Common LL Parser Implementations

- Use stack to maintain non-terminals encountered which still need to be resolved.
  - Table Driven Parser
  - Recursive descent parser

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.

GEORGETOWN UNIVERSITY

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

(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))$

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 firstsymbol of the string to parse.

This is called a **match** step.

GEORGETOWN UNIVERSITY

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))$

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))$

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))$

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))$

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))$

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))$

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))$

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))$

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))$

GEORGETOWN UNIVERSITY

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))$

GEORGETOWN UNIVERSITY

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))$

GEORGETOWN UNIVERSITY

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))$

GEORGETOWN UNIVERSITY

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))$

GEORGETOWN UNIVERSITY

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))$

GEORGETOWN UNIVERSITY

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))$

GEORGETOWN UNIVERSITY

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))$

GEORGETOWN UNIVERSITY

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))$
))$	))$

GEORGETOWN UNIVERSITY

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))$
))$	))$
)$	)$

GEORGETOWN UNIVERSITY

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))$
))$	))$
)$	)$
$	$

Parse Errors

- Arriving at an empty location in the parse table indicates a parse error.

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	(	)	+	*
E	1	2			
Op				3	4

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

Parsing Algorithms

- Algorithms: LL(k)

- There are different LL algorithms for different k
  - Table construction
  - Table usage

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**].

But how do we build the table??

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

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**).

Filling in Table Entries

- Intuition: The next character should uniquely identify a production, so we should pick a production that ultimately starts with that character.
- T[**A**, **t**] should be a production **A** → ω iff ω derives something starting with **t**.
- More rigorously:

  T[**A**, **t**] = **B** ω iff **A** → ω and $\omega \Rightarrow^*$ **t** ω'

LL(1) Parse Tables

E → int
E → (E Op E)
Op → +
Op → *

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

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**.

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

- We also include **ε** in FIRST(**A**) if A can produce the empty string.
- Intuitively, FIRST(**A**) is the set of terminals that can be at the start of a string produced by **A**.
- We can generalize FIRST to strings with FIRST*(*ω*) being the set of all terminals (or **ε**) that can appear at the start of a string derived from *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

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

LL(1) Tables with ε

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

	+	-	#	$
Num				
Sign				
Digits				
More				
Digit				

LL(1) Tables with ε

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

	+	-	#	$
Num				
Sign				
Digits				
More				
Digit				

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

	+	-	#	$
Num				
Sign				
Digits				
More				
Digit				

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

LL(1) Tables with ε

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

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

GEORGETOWN UNIVERSITY

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:

$$\mathtt{A} \Rightarrow \mathtt{B}\,\mathtt{t} \Rightarrow \mathtt{t}$$

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

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** }).

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*)

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:

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

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

Computation of FOLLOW Sets

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

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

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

The Final LL(1) Table Algorithm

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

An Egregious Abuse of Notation

- Compute FIRST(**A**) and FOLLOW(**A**) for all nonterminals **A**.
- For each rule **A** → ω, for each terminal **t** ∈ FIRST*(ωFOLLOW(**A**)), set T[**A**, **t**] = ω.

Example LL(1) Construction

The Limits of LL(1)

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  $A \rightarrow Ab \mid c$

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

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → **A** **b** | **c**

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

	b	c
A		**A** → **A** **b** **A** → **c**

A Grammar that is Not LL(1)

- Consider the following (left-recursive) grammar:

  A → **A** **b** | **c**

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

	b	c
A		**A** → **A** **b** **A** → **c**

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

GEORGETOWN UNIVERSITY

Eliminating Left Recursion

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

$$A \rightarrow A\omega \mid a$$

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

$$A \rightarrow aB$$

$$B \rightarrow \varepsilon \mid \omega B$$

Another Non-LL(1) Grammar

- Consider the following grammar:

  E → **T**

  E → **T** **+** **E**

  T → **int**

  T → **(** **E** **)**

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

Another Non-LL(1) Grammar

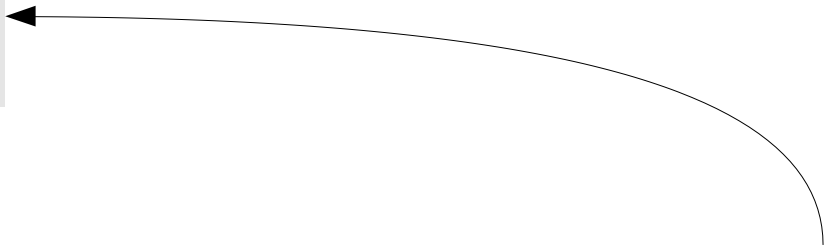
- Consider the following grammar:

  E → **T**

  E → **T** + **E**

  T → **int**

  T → (**E**)



How do you predict which of these to use?

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

Left-Factoring

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

Left-Factoring

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

Left-Factoring

```
E → TY

T → int
T → (E)
```

Left-Factoring

```
E → TY

T → int
T → (E)
Y → +E
Y → ε
```

Left-Factoring

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

Left-Factoring

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

Left-Factoring

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

FIRST

	T	Y
E		

FOLLOW

	T	Y
E		

GEORGETOWN UNIVERSITY

Left-Factoring

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

FIRST

E	T	Y
	int (	

FOLLOW

E	T	Y

GEORGETOWN UNIVERSITY

Left-Factoring

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

FIRST

	T	Y
E	int (	+ ε

FOLLOW

	T	Y
E		

GEORGETOWN UNIVERSITY

Left-Factoring

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

FIRST

	T	Y
int	int	+
(	(	ε

FOLLOW

	T	Y
E		

Left-Factoring

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

FIRST

	T	Y
int	int	+
(	(	ε

FOLLOW

	T	Y
$		

GEORGETOWN UNIVERSITY

Left-Factoring

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

FIRST

	T	Y
E		
int	int	+
(	(	ε

FOLLOW

	T	Y
E		
$		
)		

Left-Factoring

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

FIRST

	T	Y
int	int	+
(	(	ε

FOLLOW

	T	Y
$	+	
)		

Left-Factoring

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

FIRST

	T	Y
int	int	+
(	(	ε

FOLLOW

	T	Y
$	+	$
)		)

GEORGETOWN UNIVERSITY

Left-Factoring

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

FIRST

	T	Y
E	int (	+ ε
int		
(		

FOLLOW

	T	Y
E	+ $)	$)
$		
)		

GEORGETOWN UNIVERSITY

Left-Factoring

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

FIRST

E	T	Y
int	int	+
(	(	ε

FOLLOW

E	T	Y
$	+	$
)	$	)
	)	

	int	(	)	+	$
E					
T					
Y					

GEORGETOWN UNIVERSITY

Left-Factoring

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

FIRST

	T	Y
E	int (	+ ε
int		
(		

FOLLOW

	T	Y
E	+ $)	$)
$		
)		

	int	(	)	+	$
E	1	1			
T					
Y					

GEORGETOWN UNIVERSITY

Left-Factoring

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

FIRST

	T	Y
E	int (	+ ε
int		
(		

FOLLOW

	T	Y
E	+ $)	$)
$		
)		

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

GEORGETOWN UNIVERSITY

Left-Factoring

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

FIRST

	T	Y
E	int (	+ ε
int		
(		

FOLLOW

	T	Y
E	+ $)	$)
$		
)		

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

GEORGETOWN UNIVERSITY

Left-Factoring

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

FIRST

	T	Y
E	int (	+ ε
int		
(		

FOLLOW

	T	Y
E	+ $)	$)
$		
)		

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

GEORGETOWN UNIVERSITY

A Formal Characterization of LL(1)

- A grammar G is LL(1) iff for any productions $\mathbf{A} \rightarrow \omega_1$ and $\mathbf{A} \rightarrow \omega_2$, the sets

$$\text{FIRST}(\omega_1\ \text{FOLLOW}(\mathbf{A}))$$

and

$$\text{FIRST}(\omega_2\ \text{FOLLOW}(\mathbf{A}))$$

are disjoint.

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

The Strengths of LL(1)

LL(1) is Straightforward

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

LL(1) is Fast

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

Recursive Descent Parsing

GEORGETOWN UNIVERSITY

Parsing

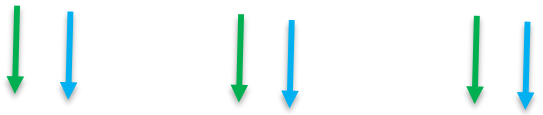
- Goals
  - Find syntax errors
  - Produce or trace parse tree

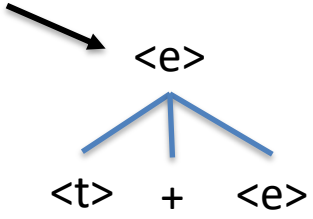
- Types of parsers
  - Top-down: build tree from top down
  - Bottom-up: build tree from bottom up

Top-Down Parsers

- Produces a left-most derivation
- Traces or builds tree in pre-order
- Crux: Must choose / ***predict*** correct RHS when expanding a node in the parse tree.
  - Eg: 3+4

  - For efficient design best to use next token or lookahead only one token





Top Down Parsing

- Formally, a top down parser must do the following:
  - Given a sentence in the form xAα , the parser must choose the correct A=> rule .
    - Notation
      - Lowercase: terminal sequence
      - Uppercase: non-terminal
      - Greek: sequence of terminals and non-terminals
  - Its best if this decision can be made based on the first tokens of A's RHSs.
    - This is true if all first terminals of A productions are different.
- Implementation of Top Down Parser
  - Recursive decent
  - Table driven

Building a Recursive Descent Parser: Must remove left recursion. Two options shown here.

```
/* term
Parses strings in the language generated by the rule:
<term> -> <term> * <factor> | …
*/
void term() {

/* Parse the first factor */
 term();

/* As long as the next token is * or /,
   next token and parse the next factor */
  if(nextToken == MULT_OP || nextToken == DIV_OP) {
    MultDiv();
    factor();
  }
} /* End of function term */
```

Remove Left Recursion: Recursive Descent Parser Option 1

```
/* term
Parses strings in the language generated by the rule:
<term> -> <factor>  <term_prime>
<term_prime> -> * <factor> <term_prime> | epsilon

*/
void term() {

/* Parse the first factor */
 factor();

/* Lookahead:  As long as the next token is * or /,
   next token and parse the next factor */
  if(nextToken == MULT_OP || nextToken == DIV_OP) {
     MultDiv();    // match star
    factor();
    // NOTE: CAN EVAL BEFORE RECURSIVE CALL TO ENFORCE LEFT ASSOCIATIVE OPERATORS
     term_prime(); // recursion
  }
} /* End of function term */
```

GEORGETOWN UNIVERSITY

Remove Left Recursion: Using EBNF as template Option 2.

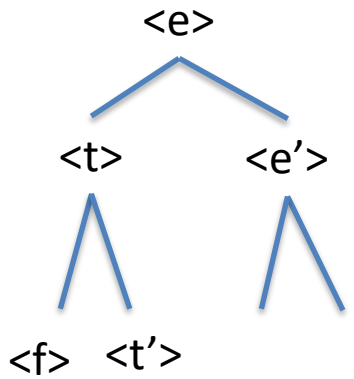
```
/* term
Parses strings in the language generated by the EBNF rule:
<term> -> <factor> {(* | /) <factor>)
*/
void term() {

/* Parse the first factor */
  factor();

/* Lookahead:  As long as the next token is * or /,
   next token and parse the next factor */
// Uses while loop to allow for arbitrary repetition. Not as elegant, but practical.
  while (nextToken == MULT_OP || nextToken == DIV_OP) {
    MultDiv();
    factor();
  }
} /* End of function term */
```

Recursive Decent Parser a "limited" implementation of a PDA

- A Recursive Decent Parser performs a Predictive DFS of the parse tree
  - DFS inherently uses a stack to traverse tree and "store" unresolved symbols to be traversed later on



Recursive Decent Parser and the parse tree

- Recursive Decent Parser Traces out Parse Tree with recursive call chain

Example Grammar

$$< \boldsymbol{expr} > \rightarrow < \boldsymbol{term} > \{(+|-) < \boldsymbol{term} >\}$$
$$< \boldsymbol{term} > \rightarrow < \boldsymbol{factor} > \{(* | /) < \boldsymbol{factor} >\}$$
$$< \boldsymbol{factor} > \rightarrow \mathbf{0} \mid \mathbf{1} \mid \ldots \mid \mathbf{9}$$

```
void expr() {

term(); // nextToken is Lookahead
while (nextToken == PLUS_OP || nextToken == SUB_OP) {
    PlusSub();
    term();
  }
```

```
void term() {

factor(); // nextToken is Lookahead
while (nextToken == MULT_OP || nextToken == DIV_OP) {
    MultDiv();
    factor();
  }
```

```
void PlusSub() { // expects to match '+' or '-'

if(thisToken == PLUS_OP|| thisToken == SUB_OP) {
 incrementTokenCounter() // Terminal Matched!
    //Proceed to next token
}
else
    throw parseError("Parse Error: + or - expected,
    but instead found ", nextToken )
```

```
void factor() { // expects a numeric terminal

if(thisToken == NUM) {
 incrementTokenCounter() // Terminal Matched!
    //Proceed to next token
}
else
    throw parseError("Parse Error: NUM expected, but
    instead found ", nextToken )
```

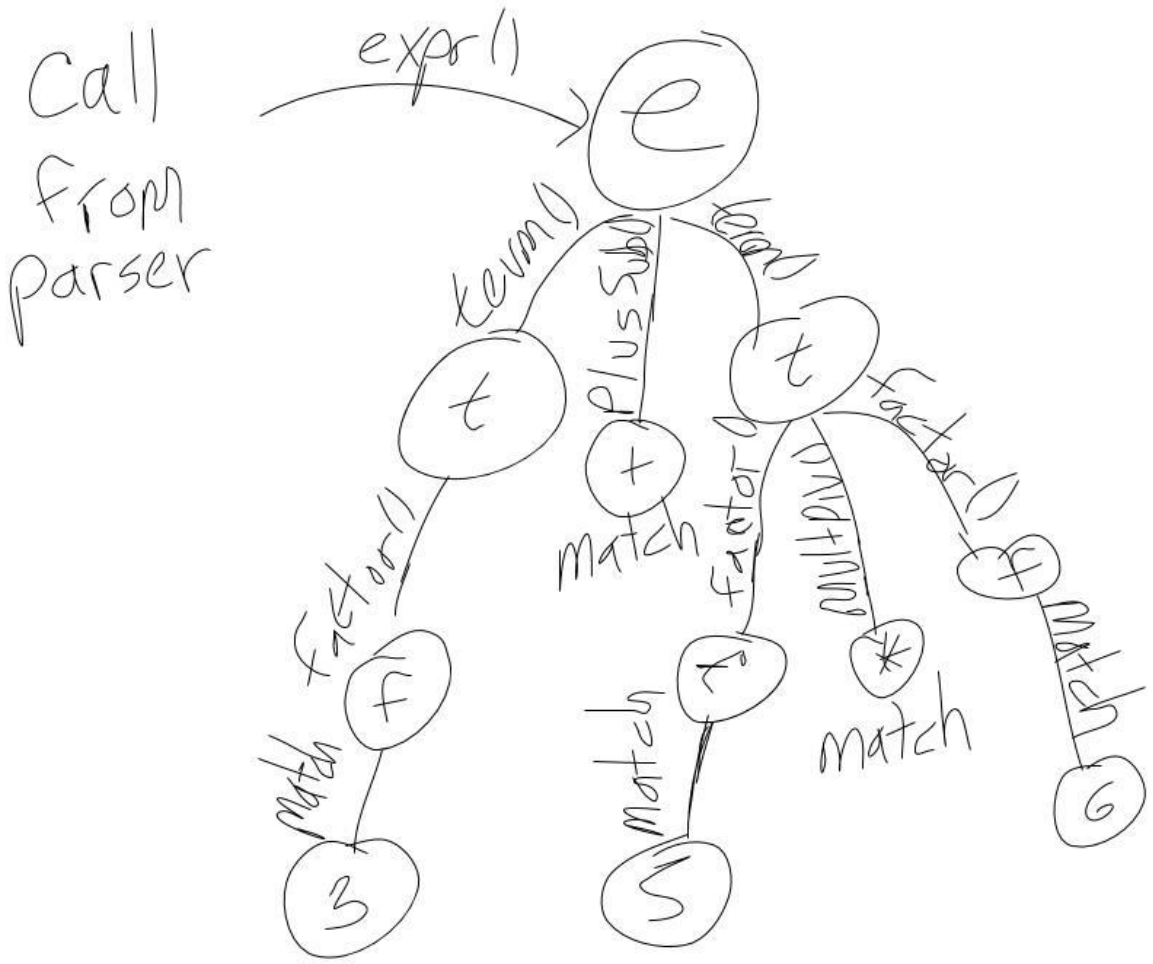
GEORGETOWN UNIVERSITY

Recursive Decent Parser Example

Assume input 3 + 5 * 6

Example Grammar

$< expr > \rightarrow < term > \{(+|-) < term >\}$
$< term > \rightarrow < factor > \{(*|/) < factor >\}$
$< factor > \rightarrow 0 \mid 1 \mid \ldots . \mid 9$




```
void expr() {

term();
while (nextToken == PLUS_OP || nextToken == SUB_OP) {
    PlusSub();
    term();
  }
```

```
void term() {

factor();
while (nextToken == MULT_OP || nextToken == DIV_OP) {
    MultDiv();
    factor();
  }
```

```
void factor() {

if(nextToken == NUM) {
 incrementTokenCounter() // Terminal Matched! //Proceed to next token
}
else
    throw parseError("Parse Error: NUM expected, but instead found ", nextToken )
```

GEORGETOWN UNIVERSITY

Recursive Decent Parser Example (In Class)

- Try

  3 + 5 + -

  3 * 5

  3 -- 3

Example Grammar

$< \boldsymbol{expr} > \rightarrow < \boldsymbol{term} > \{(+|-) < \boldsymbol{term} >\}$

$< \boldsymbol{term} > \rightarrow < \boldsymbol{factor} > \{(*|/) < \boldsymbol{factor} >\}$

$< \boldsymbol{factor} > \rightarrow \mathbf{0} \mid \mathbf{1} \mid \ldots . \mid \mathbf{9}$

GEORGETOWN UNIVERSITY

Top-Down Parsing Table Implementation

- Non – recursive implementation of Top-Down Parser
  - More “explicitly” builds a limited PDA using a Table and Stack

- Table Driven Implementation
  1. Eliminate Left Recursion ** (discussed)
  2. Build Table
  3. Perform parse algorithm using table and stack

Build Top-Down Parse Table M

Table M: numAbstractions x numCharacters. Each entry is a production

$: end of input symbol

1. For each production A=> α do the following
2. For each terminal a in FIRST(α), add A=> α to M[A,a]
3. If ε is in FIRST(α), add A=> α to M[A,b] for each b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A), add A=> α to M[A,$]

3?: If α can be the empty string, then the next input char may be from FOLLOW(A)

FOLLOW(A): All terminals that can directly follow A (not necessarily finish A) in a derivation. Be aware of ε productions.

Creating A Top-Down Parse Table. EG

- Start with Grammar:

  <e> => <e> + <t>

  <t> => x | y

- Step 1: Remove Left Recursion + Left Factor

<e> => <t><e’>

<e’>=> + <t><e’> | ε

<t> => x | y

Creating A Top-Down Parse Table. EG

FIRST(<t><e'>) = {x,y}

<e> => <t><e'>

<e'>=> + <t><e'> | ε

<t> => x | y

M	x	y	+	$
<e>	<e> => <t><e'>	<e> => <t><e'>		
<e'>			<e'>=> + <t><e'>	<e'>=> ε
<t>	<t> => x	<t> => y		

Table M: numAbstractions x numCharacters. Each entry is a production

For each production A=> α do the following

For each terminal a in FIRST(α), add A=> α to M[A,a]

If ε is in FIRST(α), add A=> α to M[A,b] for each b in FOLLOW(A). If ε is in FIRST(α) and $ is in FOLLOW(A), add A=> α to M[A,$]

Example: Perform a Top-Down Parse

- Step 3: Using a Top-Down Parse Table

```
    set pos to first symbol
    set stack to S$ // S = startSymbol
Let:  X = peek(); a = input [ pos ] ;
    do {
        if X is terminal or $ {
            if X == a
                {pop X; pos++; }
            else
                syntax error
        }
        else if M[ X, a] ==  X=>α1 , … αk {
            pop X;
            push αk , … α1 ;}
        else error
        } while(  X != $ )

    // accept
```

M	x	y	+	$
<e>	<e> => <t><e’>	<e> => <t><e’>		
<e’>			<e’>=> + <t><e’>	<e’>=> ε
<t>	<t> => x	<t> => y		

Example:
x+y

Stack	Input
$<e>	x+y$
$<e’><t>	x+y$
$<e’>x	x+y$
$<e’>	+y$
$<e’> <t>+	+y$
$<e’> <t>	y$
$<e’> y	y$
$<e’>	$
$	$

Try Example (20)

A => Ab | AD | a | b

D => c | d

Step 1: Remove Left Recursion + Left Factor

LL Grammars

- All Grammars that can be parsed using a top-down parser with k lookahead tokens are called LL(k) grammars.
  - Our top-down, table parsing algorithm can parse any LL(1) grammar. 1 lookahead token
    - Note: multiple entries in the table implies an ambiguous grammar (not LL(1))
  - LL: Left to right scan, leftmost derivation
  - LL(1) have the following characteristics,
    - Grammars cannot be left recursive
    - Whenever A=> α | β
      - » FIRST(α) ∩ FIRST(β) = ϕ
      - » At most only one of α and β can produce ε
      - » If β =>* ε , then α does not produce FIRST(α) ∩ FOLLOW(A) = ϕ
  - LL(k) is a strict subset of CFG