CS251: Introduction to Language Processing

Top-Down Parsing

Vishwesh Jatala

Department of CSE

Indian Institute of Technology Bhilai

vishwesh@iitbhilai.ac.in

2023-24-M

Acknowledgement

- Today's slides are modified from that of
  - *Stanford University:*
    - *https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/*

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.

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.

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.**

Predictive Parsing

E → int
E → (E Op E)
Op → +
Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

E → int
E → (E Op E)
Op → +
Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

E
(E Op E)

E → int

E → (E Op E)

Op → +

Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

E
(E Op E)
(int Op E)

E → int

E → (E Op E)

Op → +

Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

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

E → int

E → (E Op E)

Op → +

Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

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

E → int

E → (E Op E)

Op → +

Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

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

E → int
E → (E Op E)
Op → +
Op → *

(	int	+	(	int	*	int	)	)

Predictive Parsing

E → int
E → (E Op E)
Op → +
Op → *

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

(	int	+	(	int	*	int	)	)

Predictive Parsing

E → int

E → (E Op E)

Op → +

Op → *

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

(	int	+	(	int	*	int	)	)

Predictive Parsing

E → int
E → (E Op E)
Op → +
Op → *

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

(	int	+	(	int	*	int	)	)

LL(1) Parse Tables

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

LL(1) Parse Tables

E → int
E → (E Op E)
Op → +
Op → *

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

LL(1) Parsing

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

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

LL(1) Parsing

E	(int + (int * int))

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

LL(1) Parsing

E	(int + (int * int))

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Parsing

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

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

This is called a match step.

LL(1) Parsing

(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))$

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))$
	int))$

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Parsing

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Error Detection

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

```
int + int$
```

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

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

E$	int + int$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

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

E$	int + int$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Error Detection

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Error Detection

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

LL(1) Error Detection, Part II

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

```
(int (int))$
```

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

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

E$	(int (int))$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

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

E$	(int (int))$

	int	(	)	+	*
E	1	2			
Op				3	4

LL(1) Error Detection, Part II

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

	int	(	)	+	*
E	1	2			
Op				3	4

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

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))$

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))$

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))$

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))$

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))$

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**].

LL(1) Parse Tables

E → int
E → (E Op E)
Op → +
Op → *

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

LL(1) Parse Tables

$$E \rightarrow AS$$
$$A \rightarrow a \mid Tb$$
$$T \rightarrow t$$
$$S \rightarrow s$$

FIRST(E) = FIRST(A)
= {a} U FIRST(T)
= {a, t}

LL(1) Parse Tables

$$E \rightarrow AS$$
$$A \rightarrow \varepsilon$$
$$S \rightarrow s$$

FIRST(E) = FIRST(AS)
= {s}

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 $\Rightarrow^*$ tω for some ω }

- For all nonterminals **A** where **A** $\rightarrow$ **ε** is a production, add **ε** to FIRST(**A**).

- Repeat the following until no changes occur:

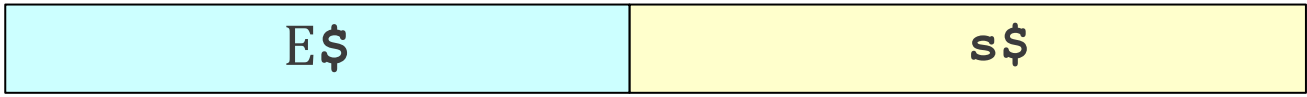
  - For each production **A** $\rightarrow$ ***a***, set FIRST(**A**) = FIRST(**A**) $\cup$ FIRST*(***a***)

LL(1) Parsing

(1) E → AS

(2) A → ε

(3) S → s



LL(1) Parsing

(1) **E** → **AS**
(2) **A** → **ε**
(3) **S** → **s**

E$	**s$**
AS$	**s$**

Apply **A** → **ε**, **if s follows A**
FOLLOW(A) = FIRST(S)
= {s}

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:

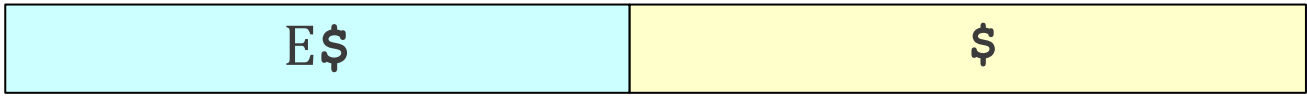
  FOLLOW(**A**) = { **t** | **S** $\Rightarrow^*$ ***a*****A****t*****ω*** for some ***a***, ***ω*** } where **S** is the start symbol of the grammar.

- Informally, every nonterminal that can ever come after **A** in a derivation.

LL(1) Parse Tables

$E \rightarrow A$

$A \rightarrow \varepsilon$



FOLLOW(E) = {$}

FOLLOW(A) = FOLLOW(E)

= {$}

LL(1) Parse Tables

$E \rightarrow A$

$A \rightarrow \varepsilon$

E$	$
A$	$

FOLLOW(E) = {$}

FOLLOW(A) = FOLLOW(E)

= {$}

Computation of FOLLOW Sets

- Initially, for each nonterminal **A**, set
  FOLLOW(**A**) = { **t** | **B** → ***a*****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** → ***a*****A*****ω*** is a production, set
    FOLLOW(**A**) = FOLLOW(**A**) ∪ FIRST*(***ω***) - { **ε** }.
  - If **B** → ***a*****A*****ω*** is a production and **ε** ∈ FIRST*(***ω***), set
    FOLLOW(**A**) = FOLLOW(**A**) ∪ FOLLOW(**B**).