LL(1) Parser

March 10, 2025

Introduction to LL(1) Parsing

LL(1) Parser: Also known as non-recursive descent parsers or table-driven parsers or predictive parsers

- A top-down parser that uses a single token lookahead.
- "L" = Left-to-right parsing.
- "L" = Leftmost derivation.
- "1" = One token lookahead.

Conditions for LL(1) Grammar

To construct a working LL(1) parsing table, a grammar must satisfy these conditions:

- **No Left Recursion:** Avoid recursive definitions like $A \to A + b$, as LL(1) parsers cannot handle infinite recursion.
- **Unambiguous Grammar:** Ensure each string can be derived in only one way, preventing multiple parse trees for the same input.
- **Left Factoring (Determinism):** If a non-terminal has multiple productions starting with the same prefix, it must be rewritten to make parsing decisions based on a single lookahead token.

Steps to Construct LL(1) Parsing Table

1. Compute FIRST and FOLLOW sets.
2. Construct parsing table:
   - For each production $A \to \alpha$, add it to table entry $M[A, a]$ for each $a \in$ FIRST(α).
   - If $\varepsilon \in$ FIRST(α), add the rule to $M[A, b]$ for each $b \in$ FOLLOW(A).
3. If multiple entries exist for a cell, the grammar is not LL(1).

Example Grammar

Example Grammar:

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \varepsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \varepsilon \\
F &\rightarrow id \mid (E)
\end{aligned}
$$

Stepwise Explanation:

1. Compute FIRST and FOLLOW sets.
2. Identify table entries for each production.
3. Check for conflicts in table.

FIRST and FOLLOW Sets

FIRST and FOLLOW Sets:

Non-Terminal	FIRST Set	FOLLOW Set
E	{ id, (}	{ $,) }
E'	{ +, ε}	{ $,) }
T	{ id, (}	{ +, $,) }
T'	{ *, ε}	{ +, $,) }
F	{ id, (}	{ *, +, $,) }

LL(1) Parsing Table

Parsing Table:

	id	+	*	(	$
E	E → TE’			E → TE’	
E’		E’ → +TE’	E’ → ε		E’ → ε
T	T → FT’			T → FT’	
T’		T’ → ε	T’ → *FT’		T’ → ε
F	F → id			F → (E)	

Steps to Parse an Expression Using the LL(1) Table

Algorithm:

1. Initialize the stack with \$ and the start symbol E.
2. Repeat until stack is empty:
   - Let **X** be the top of the stack.
   - If **X** is a terminal and matches input, pop **X** and advance input.
   - If **X** is a non-terminal, consult the parsing table $M[X, \text{current\_input}]$ and replace **X** with the corresponding production.
   - If **X** is \$ and input is exhausted, accept.
   - If no rule exists in the table, report an error.

Example: Parsing the Expression "id + id * id"

Parsing Steps:

1. Initialize stack: $ E

Example: Parsing the Expression "id + id * id"

Parsing Steps:

1. Initialize stack: $ E
2. Read input: `id + id * id $`

Example: Parsing the Expression "id + id * id"

Parsing Steps:

1. Initialize stack: $ E
2. Read input: `id + id * id $`
3. Replace E with TE'

Example: Parsing the Expression ”id + id * id”

Parsing Steps:

1. Initialize stack: $ E
2. Read input: `id + id * id $`
3. Replace E with TE’
4. Replace T with FT’

Example: Parsing the Expression "id + id * id"

Parsing Steps:

1. Initialize stack: $ E
2. Read input: `id + id * id $`
3. Replace E with TE'
4. Replace T with FT'
5. Match `id`, pop from stack and advance input.

Example: Parsing the Expression "id + id * id"

Parsing Steps:

1. Initialize stack: $ E
2. Read input: `id + id * id $`
3. Replace E with TE'
4. Replace T with FT'
5. Match `id`, pop from stack and advance input.
6. Replace E' with +TE'

Example: Parsing the Expression "id + id * id"

Parsing Steps:

1. Initialize stack: $ E
2. Read input: `id + id * id $`
3. Replace E with TE'
4. Replace T with FT'
5. Match `id`, pop from stack and advance input.
6. Replace E' with +TE'
7. Match `+`, pop and advance input.

Example: Parsing the Expression ”id + id * id”

Parsing Steps:

1. Initialize stack: $ E
2. Read input: `id + id * id $`
3. Replace E with TE’
4. Replace T with FT’
5. Match `id`, pop from stack and advance input.
6. Replace E’ with +TE’
7. Match +, pop and advance input.
8. Replace T with FT’

Example: Parsing the Expression "id + id * id"

Parsing Steps:

1. Initialize stack: $ E
2. Read input: `id + id * id $`
3. Replace E with TE'
4. Replace T with FT'
5. Match `id`, pop from stack and advance input.
6. Replace E' with +TE'
7. Match `+`, pop and advance input.
8. Replace T with FT'
9. Match `id`, pop and advance input.

Example: Parsing the Expression ”id + id * id”

Parsing Steps:

1. Initialize stack: $ E
2. Read input: id + id * id $
3. Replace E with TE’
4. Replace T with FT’
5. Match id, pop from stack and advance input.
6. Replace E’ with +TE’
7. Match +, pop and advance input.
8. Replace T with FT’
9. Match id, pop and advance input.
10. Replace T’ with *FT’

Example: Parsing the Expression ”id + id * id”

Parsing Steps:

1. Initialize stack: $ E
2. Read input: id + id * id $
3. Replace E with TE’
4. Replace T with FT’
5. Match id, pop from stack and advance input.
6. Replace E’ with +TE’
7. Match +, pop and advance input.
8. Replace T with FT’
9. Match id, pop and advance input.
10. Replace T’ with *FT’
11. Match *, pop and advance input.

Example: Parsing the Expression ”id + id * id”

Parsing Steps:

1. Initialize stack: $ E
2. Read input: `id + id * id $`
3. Replace E with TE’
4. Replace T with FT’
5. Match `id`, pop from stack and advance input.
6. Replace E’ with +TE’
7. Match `+`, pop and advance input.
8. Replace T with FT’
9. Match `id`, pop and advance input.
10. Replace T’ with *FT’
11. Match `*`, pop and advance input.
12. Replace F with `id`, match and advance input.

Example: Parsing the Expression ”id + id * id”

Parsing Steps:

1. Initialize stack: $ E
2. Read input: `id + id * id $`
3. Replace E with TE’
4. Replace T with FT’
5. Match `id`, pop from stack and advance input.
6. Replace E’ with +TE’
7. Match `+`, pop and advance input.
8. Replace T with FT’
9. Match `id`, pop and advance input.
10. Replace T’ with *FT’
11. Match `*`, pop and advance input.
12. Replace F with `id`, match and advance input.
13. All input is consumed, accept the string.