

Exercise walk-through

11.5

Parsing

University Computer Science

Parsing: table plus stack

- 1 **Trace** $E \rightarrow T E_p$, $E_p \rightarrow + T E_p \mid \text{empty}$, $T \rightarrow \text{id}$: $\text{id} + \text{id}$ parses in eight stack snapshots, suite-asserted.
- 2 **Conflicts** A cell with two productions means one lookahead cannot choose - the grammar is not LL(1).
- 3 **Tail** $E \rightarrow E+T$ loops the stack; $E \rightarrow T E_p$ makes the loop a tail - associativity moves to the AST build.

Parse table and stack trace

the LL(1) table

nonterm	lookahead	production
E	id	T Ep
Ep	+	+ T Ep
Ep	\$	empty

FIRST/FOLLOW fill the rows;

conflicts reject the grammar

left recursion $E \rightarrow E+T$ loops the

stack; the tail trick fixes it:

$E \rightarrow T Ep$, $Ep \rightarrow + T Ep$ | empty

the stack trace of id + id

1. E
2. T Ep
3. id Ep
4. Ep
5. + T Ep
6. T Ep
7. id Ep
8. Ep
9. (empty)

eight moves, one tree, one symbol of lookahead

The LL(1) table and the eight-snapshot stack trace of id + id.

Exercises 2.1-2.3

2.1 What fills an LL(1) table cell?

[3]

2.2 What does a conflict in a cell mean?

[3]

2.3 Why does left recursion break LL(1) parsing?

[3]

Exercises 2.1-2.3

2.1 What fills an LL(1) table cell? [3]

2.1

- The production whose FIRST contains the lookahead (FOLLOW, for an ε production) ✓.

2.2 What does a conflict in a cell mean? [3]

2.2

- Two productions land in one cell: one lookahead cannot choose – the grammar is not LL(1) ✓.

2.3 Why does left recursion break LL(1) parsing? [3]

2.3

- The leftmost nonterminal expands to itself: the stack never shrinks and no input is consumed – an infinite loop ✓.

Exercise 3.1

3.1 Grammar: $S \rightarrow \mathbf{id} = E$; $E \rightarrow TE'$; $E' \rightarrow +TE' \mid \varepsilon$; $T \rightarrow \mathbf{id}$.

Build the LL(1) table (all nonempty cells with justification). Parse $\mathbf{id} = \mathbf{id} + \mathbf{id}$: give each stack snapshot with the lookahead, naming the table cell or the match at each move. State the count of moves and the final verdict. [8]

Exercise 3.1

3.1 Grammar: $S \rightarrow \mathbf{id} = E$; $E \rightarrow TE'$; $E' \rightarrow +TE' \mid \varepsilon$; $T \rightarrow \mathbf{id}$.

Build the LL(1) table (all nonempty cells with justification). Parse $\mathbf{id} = \mathbf{id} + \mathbf{id}$: give each stack snapshot with the lookahead, naming the table cell or the match at each move. State the count of moves and the final verdict. [8]

3.1

- Table: on lookahead $\mathbf{id}$, S expands to $\mathbf{id} = E$ and E expands to $T E_p$ and T expands to $\mathbf{id}$; on $+$, E_p expands to $+ T E_p$; at end of input, E_p expands to empty ✓.
- Snapshots, written stack | lookahead: $S \mid \mathbf{id}$; then $\mathbf{id} = E \mid \mathbf{id}$ (apply S); $= E \mid =$ (match); $E \mid \mathbf{id}$ (match); $T E_p \mid \mathbf{id}$ (apply E); $\mathbf{id} E_p \mid \mathbf{id}$ (apply T); $E_p \mid +$ (match); $+ T E_p \mid +$ (apply E_p); $T E_p \mid \mathbf{id}$ (match); $\mathbf{id} E_p \mid \mathbf{id}$ (apply T); $E_p \mid \text{dollar}$ (match; apply E_p to empty) ✓.
- Eleven moves; accept (stack empty at end of input) ✓.

Exercise 3.2

3.2 Remove left recursion from $A \rightarrow Aa \mid Ab \mid c$.

Give the transformed grammar. Derive $c a b a$ under it (leftmost). State what the original grammar generated, in set notation, to show the transformation preserved the language. State the associativity cost and where it is paid. [7]

Exercise 3.2

3.2 Remove left recursion from $A \rightarrow A\mathbf{a} \mid A\mathbf{b} \mid \mathbf{c}$.

Give the transformed grammar. Derive $\mathbf{c} \mathbf{a} \mathbf{b} \mathbf{a}$ under it (leftmost). State what the original grammar generated, in set notation, to show the transformation preserved the language. State the associativity cost and where it is paid. [7]

3.2

- Transformed: $A \rightarrow \mathbf{c}A'; A' \rightarrow \mathbf{a}A' \mid \mathbf{b}A' \mid \varepsilon$ ✓.
- Derivation: A , then $\mathbf{c} A$ -prime, then $\mathbf{c} \mathbf{a} A$ -prime, then $\mathbf{c} \mathbf{a} \mathbf{b} A$ -prime, then $\mathbf{c} \mathbf{a} \mathbf{b} \mathbf{a} A$ -prime, then $\mathbf{c} \mathbf{a} \mathbf{b} \mathbf{a}$ ✓.
- Original language: $\mathbf{c}$ followed by any string of $\mathbf{a}$'s and $\mathbf{b}$'s – $\mathbf{c}(\mathbf{a} \mid \mathbf{b})^*$; the transformation generates exactly that ✓.
- Cost: right-recursive trees lean right; left associativity must be restored when building the AST ✓.