6 Bottom-up parsers

6.1 Principles of Bottom-Up Parsing

Core Parsing Strategy

Bottom-up parsers **construct parse trees from leaves to root** by reversing derivation rules. Unlike top-down approaches starting with start symbols, they use **shift-reduce actions** on terminal symbols from input strings. Key components: stack for partial reductions and input buffer.

Key Operations

- **Shift**: Move terminal from input to stack
- **Reduce**: Replace handle (RHS of rule) on stack with LHS non-terminal
- Accept when stack contains only start symbol and input is empty.

Comparison with Top-Down Parsing

- **Power**: Bottom-up methods handle broader grammar classes (e.g., left-recursive rules)
- **Efficiency**: Decisions based on actual input rather than predictions
- **Derivation Type**: Builds **rightmost derivation in reverse** (Example 6.1: `Id + Id * Id` reduction sequence).

6.2 Shift-Reduce Parsers and Handle Identification

Handle Detection Mechanics

A **handle** is the RHS α of a production rule $A \to \alpha$ appearing on stack top. Parsers must:

1. Identify valid handles using grammar rules
2. Choose between shifting or reducing (conflict resolution critical).

Conflict Types

- **Shift-Reduce**: Handle detected but next input could extend potential handle
- **Reduce-Reduce**: Multiple valid handles for stack top (Example 6.10: Arithmetic expression ambiguity).

Viable Prefixes

Definition: Prefixes of right-sentential forms that can appear on parser stack during valid reductions. The CFSM (Section 6.2) recognizes these prefixes to guide parsing decisions.

6.3 Canonical Finite State Machine (CFSM)

Construction Algorithm

1. **Closure Operation**: Expand items with all possible productions for non-terminals following dot

   ```
   def closure(I):
       repeat:
           for A → alpha·Bbeta in I:
               add B \rightarrow ·gamma for all B \rightarrow gamma
   ```

2. **Goto Function**: Compute state transitions for symbols (Algorithm 9).

Example Grammar:

$$
\begin{aligned}
S &\to A\$ \\
A &\to aCD \\
&\to ab \\
C &\to c \\
D &\to d
\end{aligned}
$$

CFSM:

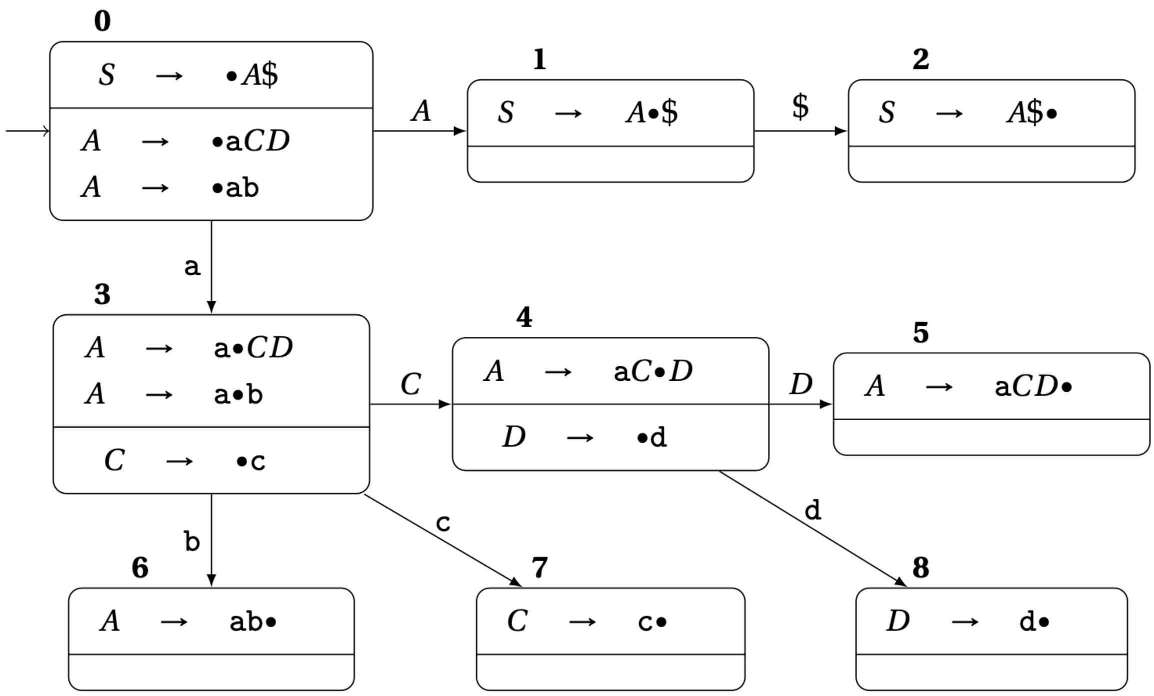

Figure 1: Alt text

Item Types

- **Kernel Items**: Initial items defining a state's core
- **Non-Kernel Items**: Added via closure (e.g., state 0 in Figure 6.3 includes closure items for $A \to aCD$ and $A \to ab$).

6.4 LR(0) Parsers

Deterministic Parsing Without Lookahead

- **Action Table**: Maps CFSM states to shift/reduce/accept actions
- **State Stack**: Tracks CFSM states instead of symbols for efficiency (Figure 6.4).

Limitations

- Fails on grammars needing lookahead (Example 6.10: Shift/Reduce conflict in state 1).

6.5 SLR(1) Parsers

Simple Lookahead Resolution

Uses **Follow sets** to resolve conflicts:

- Reduce only if lookahead $\in$ Follow(A) for rule $A \to \alpha$

- Shift if lookahead $\in$ First(β) for incomplete item $A \rightarrow \alpha \cdot a\beta$.

Table Construction

- Augment LR(0) table with Follow-set checks (Algorithm 12).

6.6 LR(k) Parsers

Precise Lookahead Handling

- **Items**: Augmented with lookahead strings (e.g., $A \rightarrow \alpha \cdot \beta, u$)
- **CFSM States**: Split based on specific lookahead contexts (Example 6.15: Distinct states for $ vs =).

LR(k) Grammars

Formal Definition: No ambiguous reductions when k-symbol lookahead distinguishes between valid handles (Definition 6.19). Non-LR(k) example: Knuth's `S \rightarrow aAbc | aBbd` (Figure 6.20).

6.7 LALR(k) Parsers

State Merging for Efficiency

- Merge LR(k) states with **identical hearts** (LR(0) items) but different lookaheads
- Preserves deterministic behavior while reducing table size (Proposition 6.4: Same state count as LR(0)).

Tradeoffs

- May reintroduce conflicts absent in LR(k) (Example 6.26: Merged states 9 & 19 in Table 6.4).

6.8 Bottom-Up Hierarchy

Grammar Classes

- **Inclusions**: $LR(0) \subset SLR(1) \subset LALR(1) \subset LR(1)$ (Theorem 6.6)
- **Undecidable**: Whether a grammar is LR(k) for some k (Knuth 1965).

Language Classes

- **DCFL = LR(1)**: All deterministic CFLs have LR(1) grammars (Theorem 6.8)
- **Endmarker Trick**: $ suffix enables LR(0) parsing for DCFLs (Theorem 6.9).

6.9 Top-Down vs Bottom-Up

Syntactic Power

- **LL(k) $\subset$ LR(k)**: All LL(k) grammars are LR(k), but not vice versa (Theorem 6.10)
- **Hierarchy Collapse**: LR(k) lang = DCFL vs infinite LL(k) hierarchy.

Key Points to Remember

- **Shift-Reduce Conflict → Use Lookahead**: SLR(1) uses Follow sets; LR(k) uses precise contexts

- **Viable Prefixes → CFSM States**: Critical for tracking valid reductions
- **LALR Efficiency**: Merges LR(k) states but risks conflicts
- **Grammar Power**: Bottom-up > Top-down (LR handles left recursion/ambiguity)
- **DCFL = LR(1)**: All deterministic CFLs have LR(1) grammars
- **Endmarker $**: Enables LR(0) parsing for DCFL+ languages
- **Undecidability**: No algorithm detects if grammar is LR(k) for any k