# Compilers

## Foundations of Formal Languages

**Regular Language**: A language that can be expressed using regular expressions or recognized by finite automata (DFA or NFA). Regular languages are closed under union, concatenation, and Kleene star operations. Regular languages are less expressive than context-free languages. **Regular Expression**: A pattern describing a regular language, using symbols like '*' (zero or more), '+' (one or more), '|' (union), and '()' (grouping). **Context-Free Language (CFL)**: Languages defined by context-free grammars (CFGs) and recognized by pushdown automata (PDAs). **Context-Free Grammar (CFG)**: A grammar consisting of rules of the form $A \to \alpha$, where $A$ is a nonterminal, and $\alpha$ is a string of terminals and/or nonterminals. CFGs generate context-free languages, which can be recognized by pushdown automata. **Strongly LL(k)**: A grammar is strongly LL(k) if, for every nonterminal $A$ and every lookahead string of k tokens $w$, there is at most one production rule $A \to \alpha$ that can be applied, determined solely by the first k tokens of the input (the lookahead). This eliminates any need for backtracking or additional context beyond the k-token lookahead.

## First and Follow Sets

The $\mathbf{FIRST^k}(\alpha)$ set for a string $\alpha$ (terminal, non-terminal, or sequence) contains all sequences of up to $k$ terminals that can appear as the first $k$ symbols in any derivation of $\alpha$. If $\alpha$ derives a string shorter than $k$, the entire derived string is included.

**Steps to Compute** $FIRST^k(\alpha)$:

1. If $\alpha$ is a terminal $a$: Add the string $a$ (of length 1) to $FIRST^k(\alpha)$.
2. If $\alpha$ is a sequence $\alpha_1\alpha_2\ldots\alpha_n$:
   - Initialize $FIRST^k(\alpha) = \emptyset$.
   - For $i = 1$ to $n$:
     - Add to $FIRST^k(\alpha)$ all strings of length up to $k$ from $FIRST^k(\alpha_i)$ if $\alpha_1, \ldots, \alpha_{i-1}$ can all derive the empty string $\epsilon$.
     - For strings shorter than $k$, concatenate with $FIRST^{k-|s|}(\alpha_{i+1}\ldots\alpha_n)$ (where $|s|$ is the length of the string).
   - If $\alpha$ can derive $\epsilon$, include $\epsilon$ in $FIRST^k(\alpha)$.
3. If $\alpha$ is a non-terminal $A$:
   - For each production $A \to \beta$, compute $FIRST^k(\beta)$ and add its strings to $FIRST^k(A)$.
   - Repeat until $FIRST^k(A)$ stabilizes (no new strings are added).
4. Handle $k$-length lookahead: Truncate strings longer than $k$ to their first $k$ symbols.

The $\mathbf{FOLLOW^k(A)}$ set for a non-terminal $A$ contains all sequences of up to $k$ terminals that can appear immediately after $A$ in some derivation from the start symbol. If $A$ appears at the end of a derivation, include the end-of-input marker (e.g., \$).

**Steps to Compute** $FOLLOW^k(A)$:

1. Initialize:
   - For the start symbol $S$, add \$ (or a $k$-length end marker) to $FOLLOW^k(S)$.
   - Set $FOLLOW^k(A) = \emptyset$ for all other non-terminals $A$.
2. For each production $B \to \alpha A\beta$ (where $A$ is a non-terminal):
   - Compute $FIRST^k(\beta)$ and add its strings to $FOLLOW^k(A)$.
   - If $\beta$ can derive $\epsilon$, add $FOLLOW^k(B)$ to $FOLLOW^k(A)$.
3. Iterate:
   - Repeat step 2 across all productions until no new strings are added to any $FOLLOW^k(A)$.
4. Handle $k$-length lookahead: Ensure all strings in $FOLLOW^k(A)$ are of length up to $k$, truncating longer strings to their first $k$ symbols.

## LR Items

An $\mathbf{LR(k)}$ item is a production with a dot ($\bullet$) marking a position in the right-hand side, along with a $k$-symbol lookahead string. The dot indicates how much of the production has been recognized so far. An item $[A \to \alpha \bullet \beta, w]$ means we have seen $\alpha$ and expect to see $\beta$, with lookahead $w$.

**Types of LR Items:**

- **Shift item**: $[A \to \alpha \bullet a\beta, w]$ where $a$ is a terminal
- **Reduce item**: $[A \to \alpha\bullet, w]$ where the dot is at the end
- **Goto item**: $[A \to \alpha \bullet B\beta, w]$ where $B$ is a non-terminal

**Steps to Compute CLOSURE(I)** for a set of items $I$:

1. Initialize $CLOSURE(I) = I$.
2. For each item $[A \to \alpha \bullet B\beta, w]$ in $CLOSURE(I)$ where $B$ is a non-terminal:
   - For each production $B \to \gamma$:
     - Compute $FIRST^k(\beta w)$ (concatenate $\beta$ and lookahead $w$, then take first $k$ symbols).
     - For each string $u \in FIRST^k(\beta w)$:
       - Add item $[B \to \bullet\gamma, u]$ to $CLOSURE(I)$ if not already present.
3. Repeat step 2 until no new items are added to $CLOSURE(I)$.

**Steps to Compute GOTO(I, X)** for item set $I$ and symbol $X$:

1. Initialize $J = \emptyset$.
2. For each item $[A \to \alpha \bullet X\beta, w]$ in $I$:
   - Add item $[A \to \alpha X \bullet \beta, w]$ to $J$.
3. Return $CLOSURE(J)$.

**Steps to Construct the LR(k)** Automaton:

1. Create the initial state $I_0 = CLOSURE(\{[S' \to \bullet S, \$^k]\})$ where $S'$ is the augmented start symbol.
2. Initialize the set of states $\mathcal{C} = \{I_0\}$ and a worklist $W = \{I_0\}$.
3. While $W \neq \emptyset$:
   - Remove a state $I$ from $W$.
   - For each symbol $X$ (terminal or non-terminal) such that $GOTO(I, X) \neq \emptyset$:
     - Let $J = GOTO(I, X)$.
     - If $J \notin \mathcal{C}$:
       - Add $J$ to $\mathcal{C}$ and $W$.
     - Add transition $I \xrightarrow{X} J$ to the automaton.
4. The resulting automaton defines the LR parsing table.

## Parsing Conflicts

**Shift-Reduce Conflict**

A shift-reduce conflict occurs when the parser cannot decide whether to:

1. **Shift**: Move the next input symbol onto the stack
2. **Reduce**: Apply a production rule to reduce symbols on the stack

In terms of LR items, this happens when a state contains both:

1. A shift item: $[A \to \alpha \bullet a\beta, w]$ (expecting terminal $a$)
2. A reduce item: $[B \to \gamma\bullet, a]$ (ready to reduce with lookahead $a$)

The conflict manifests in the parsing table as both a shift action and a reduce action for entry $[state, a]$. **Reduce-Reduce Conflict**

A reduce-reduce conflict occurs when the parser cannot decide which production to use for reduction. This happens when a state contains multiple reduce items with overlapping lookaheads:

1. $[A \to \alpha\bullet, w]$
2. $[B \to \beta\bullet, w]$

Both items indicate that reduction is possible with the same lookahead $w$, but the parser cannot determine which production rule to apply.

The conflict manifests in the parsing table as multiple reduce actions for the same entry $[state, w]$.

**Resolution**

These conflicts indicate that the grammar is not in the respective LR class (LR(k), LALR(k), or SLR(k)). Resolution strategies include:

1. Increasing the lookahead length $k$
2. Grammar transformation (left-factoring, eliminating ambiguity)
3. Using precedence and associativity rules

## Top-Down Parsing (LL)

**Remove Unnecessary Productions** 1) **Unproductive Symbols**: Variables that cannot derive terminal strings. 2) **Unreachable Symbols**: Symbols not reachable from the start variable. **Procedure**: Remove unproductive symbols first, then unreachable ones.

**Left Recursion Elimination** 1) **Immediate Left Recursion**: For productions of the form $A \to A\alpha|\beta$, replace with:

$$A \to \beta A'$$

$$A' \to \alpha A'|\epsilon$$

2) **Indirect Left Recursion**: For productions $A \to B\alpha|\beta$ and $B \to A\gamma|\delta$, eliminate by substituting $B$ in $A$'s productions. **Left Factoring** 1) **Common Prefixes**: For productions of the form $A \to \alpha\beta_1|\alpha\beta_2$, replace with:

$$A \to \alpha A'$$

$$A' \to \beta_1|\beta_2$$

2) **Multiple Common Prefixes**: For productions $A \to \alpha\beta_1|\alpha\beta_2|\alpha\beta_3$, factor out the common prefix:

$$A \to \alpha A'$$

$$A' \to \beta_1|\beta_2|\beta_3$$

**LL(k) Grammar Requirements** 1) **No Left Recursion**: The grammar must not have left recursion. 2) **Disjoint First/Follow Sets**: For any variable $A$, the sets $FIRST(A)$ and $FOLLOW(A)$ must be disjoint. 3) **Predictive Parsing Table**: The parsing table must be constructed such that for each variable $A$ and terminal $a$, there is at most one production $A \to \alpha$ where $a \in FIRST(\alpha)$ or $a \in FOLLOW(A)$. **LL Parsing Table Construction** 1) **Compute FIRST Sets**: For each variable, compute the $FIRST$ set, which contains the terminals that can appear at the beginning of any string derived from that variable. 2) **Compute FOLLOW Sets**: For each variable, compute the $FOLLOW$ set, which contains the terminals that can appear immediately after that variable in any derivation. 3) **Construct Parsing Table**: For each variable $A$ and terminal $a$: - If $a \in FIRST(A)$, add the production $A \to \alpha$ to the table. - If $\epsilon \in FIRST(A)$ and $a \in FOLLOW(A)$, add the production $A \to \epsilon$. - If $a \notin FIRST(A)$ and $a \notin FOLLOW(A)$, the grammar is not LL(k). **LL Parsing Algorithm** 1) **Initialization**: Start with the stack containing the start variable and the input string. 2) **Top of Stack**: If the top of the stack is a terminal, match it with the input symbol. 3) **Variable Handling**: If the top of the stack is a variable $A$: - Look up the parsing table for $A$ and the current input symbol. - If a production $A \to \alpha$ is found, replace $A$ on the stack with $\alpha$. - If no production is found, report an error. 4) **Continue Until Done**: Repeat until the stack is empty or an error occurs.

# Bottom-Up Parsing (LR)

Bottom-up parsing, also known as shift-reduce parsing, builds parse trees from the leaves up to the root. The LR family of parsers represents the most powerful class of deterministic bottom-up parsers.

# LR Parsing Fundamentals

LR parsers read input from **L**eft to right and produce a **R**ightmost derivation in reverse. They use a stack to maintain parsing state and a parsing table to determine actions.

# LR Parser Actions

- **Shift**: Move input symbol onto stack
- **Reduce**: Replace handle on stack with LHS of production
- **Accept**: Parsing completed successfully
- **Error**: Invalid input detected

# LR(k) Canonical Parser

The canonical LR(k) parser uses complete item sets with k-symbol lookahead.

# LR Items

An LR(k) item is a production with a dot indicating parsing progress and a lookahead string:

$$[A \to \alpha \cdot \beta, u]$$

where $u$ is a k-symbol lookahead string.

# Closure and Goto Operations

**Closure**($I$):

```
J ← I
repeat
    for each item [A → α · Bβ, u] in J do
        for each production B → γ do
            for each v ∈ FIRST_k(βu) do
                Add [B → ·γ, v] to J
            end for
        end for
    end for
until no new items added
return J =0
```

**Goto**($I$, $X$):

$$\text{Goto}(I, X) = \text{Closure}(\{[A \to \alpha X \cdot \beta, u] \mid [A \to \alpha \cdot X\beta, u] \in I\})$$

# SLR(k) Parser

Simple LR parsers use FOLLOW sets instead of computing exact lookaheads, reducing table size but accepting fewer grammars.

# SLR Table Construction

1. Construct canonical collection of LR(0) items
2. For each state $I_i$:
   - If $[A \to \alpha \cdot a\beta] \in I_i$ and $\text{Goto}(I_i, a) = I_j$, set ACTION$[i, a]$ = shift $j$
   - If $[A \to \alpha \cdot] \in I_i$ and $A \neq S'$, set ACTION$[i, a]$ = reduce $A \to \alpha$ for all $a \in \text{FOLLOW}(A)$
   - If $[S' \to S \cdot] \in I_i$, set ACTION$[i, \$]$ = accept
3. Set GOTO$[i, A] = j$ if $\text{Goto}(I_i, A) = I_j$

# LALR(k) Parser

LALR parsers merge LR states with identical cores, providing a compromise between SLR and canonical LR.

# Core and Lookahead Merging

Two states with identical cores (same items ignoring lookaheads) are merged:

$$I_i = \{[A \to \alpha \cdot \beta, u_1], [B \to \gamma \cdot \delta, v_1], \ldots\} \quad (1)$$

$$I_j = \{[A \to \alpha \cdot \beta, u_2], [B \to \gamma \cdot \delta, v_2], \ldots\} \quad (2)$$

Merged state: $\{[A \to \alpha \cdot \beta, u_1 \cup u_2], [B \to \gamma \cdot \delta, v_1 \cup v_2], \ldots\}$

# Parser Hierarchy

The parsing methods form a hierarchy based on the classes of grammars they can handle:

$$\text{LL}(k) \subset \text{SLR}(k) \subset \text{LALR}(k) \subset \text{LR}(k) \subset \text{CFG}$$

# Comparison Properties

| Parser | Table Size | Grammar Class | Conflicts | Practical Use |
|---|---|---|---|---|
| LL(1) | Small | Restricted | Predict | Yes |
| SLR(1) | Medium | Moderate | Shift/Reduce | Limited |
| LALR(1) | Medium | Large | Reduce/Reduce | Yes |
| LR(1) | Large | Largest | Rare | Theoretical |

# Canonical FSM Construction

The Canonical Finite State Machine (CFSM) construction algorithm:

# Algorithm Steps

1. **Augment Grammar**: Add $S' \to S$ where $S'$ is new start symbol
2. **Initial State**: $I_0 = \text{Closure}(\{[S' \to \cdot S, \$]\})$
3. **State Generation**:

```
C ← {I_0}
repeat
    for each state I in C do
        for each symbol X following a dot in I do
            J ← Goto(I, X)
            if J ∉ C and J ≠ ∅ then
                Add J to C
            end if
        end for
    end for
until no new states added =0
```

4. **Transition Construction**: Add edge $I \xrightarrow{X} J$ if $J = \text{Goto}(I, X)$

# Example: CFSM for Simple Expression Grammar

Consider grammar:

$$E \to E + T \mid T \quad (3)$$

$$T \to T * F \mid F \quad (4)$$

$$F \to (E) \mid \text{id} \quad (5)$$

Augmented: $E' \to E$
State $I_0$: Closure($\{[E' \to \cdot E, \$]\}$)

$I_0 = \{[E' \to \cdot E, \$], [E \to \cdot E{+}T, \$], [E \to \cdot T, \$], [T \to \cdot T{*}F, \$], [T \to \cdot F, \$], [F \to \cdot(E), \$], [F \to \cdot \text{id}, \$]\}$

The complete CFSM contains states representing all possible parsing configurations.

# Conflict Resolution

# Shift/Reduce Conflicts

Occur when parser cannot decide between shifting and reducing. Resolved by:

- Operator precedence
- Associativity rules
- Grammar restructuring

# Reduce/Reduce Conflicts

Multiple reductions possible in same state. Usually indicates:

- Ambiguous grammar
- Need for stronger parser (SLR → LALR → LR)
- Grammar redesign required

# Practical Considerations

# Error Recovery

LR parsers detect errors early but recovery is complex:

- **Panic Mode**: Skip input until synchronizing token
- **Phrase Level**: Local corrections to input
- **Error Productions**: Add error handling to grammar

# Parser Generators

Tools like Yacc/Bison generate LALR parsers from grammar specifications, handling:

- Automatic table construction
- Conflict detection and resolution
- Code generation for target language

# Trivia

**Q:** Advantage of automata-based scanners over hand-built scanners: They require less coding effort. **Q:** Advantage of hand-built scanners over automata-based scanners: They can go beyond regular languages. **Q:** If you are implementing an LL(k) parser based on a given grammar, which version of recursion should you favor when manipulating or rewriting the grammar? Right recursion, LL parsers work top-down, predicting which production to use by looking ahead at the next k tokens. Left recursion causes problems for LL parsers because it leads to infinite recursion during parsing. **Q:** LLVM intermediate representation language: True: The number of registers is unlimited, The code has to be in SSA: single static assignment, Jumps can only target the start of a basic block. False: Basic blocks can end without a terminator **Q:** Argue that the exact versions of type checking and reachability analysis are in fact undecidable. Exact type checking and reachability analysis are undecidable because they reduce to the halting problem. For type checking, determining if a program's type is exactly correct requires predicting all possible execution paths, which may not terminate, akin to deciding if a Turing machine halts. Similarly, exact reachability analysis involves determining whether a program state is reachable, which also requires solving the halting problem, as it involves predicting if a computation leads to a specific state. Since the halting problem is undecidable, both tasks are undecidable in their exact forms. **Q:** Context free grammar for a language L on $\Sigma$. For all words $w \in \sigma^*$, we can use the grammar to determine whether $w \in L$ in time: polynomial in $|w|$: CYK (Cocke–Younger–Kasami) algorithm: Runs in $O(n^3)$ time where $n = |w|$, assuming the grammar is in Chomsky Normal Form. **Q:** Suppose we are given a grammar (not necessarily context free!) for a language L on $\Sigma$. For all words $w \in \sigma^*$, we can use the grammar to determine whether $w \in L$ in time: nope, that is an undecidable problem: For a general grammar (not necessarily context-free), the problem of determining whether a word $w \in L$ is undecidable. This is because general grammars correspond to Turing machines, and the membership problem for languages generated by unrestricted grammars is equivalent to the halting problem, which is undecidable. There is no algorithm that can decide, for all words $w \in \Sigma^*$, whether $w \in L$ in a finite amount of time. **Q:** Suppose we are given a deterministic finite automaton with state set Q for a language L on $\Sigma$. For all words $w \in \sigma^*$, we can use the automaton to determine whether $w \in L$ in time: $O(|w|)$: A deterministic finite automaton (DFA) processes an input word $w$ by transitioning between states for each symbol in $w$. Since the state set $Q$ is fixed, each symbol is processed in constant time, $O(1)$, by looking up the transition in the DFA's transition table. For a word of length $|w|$, the DFA makes $|w|$ transitions, resulting in a total time complexity of $O(|w|)$. **Q:** How can we make sure the sequence 'true' is always deamed a contant and not a string? Define "true" as a reserved keyword or boolean literal in the lexer rules, distinct from string literals (e.g., quoted strings like "true"). **Q:** What is the longest match principle? A: The principle states that when there is a choice between several possible matches during tokenization, the lexer should choose the longest possible match that forms a valid token. **Q:** Give an arithmetic-expression tree such that the minimal number of registers required to evaluate it is exactly 7: We need to ensure that the computation's register usage peaks at 7, meaning at least one point in the evaluation requires 7 registers to hold intermediate results, and no evaluation requires more. $((((((a + b) + c) + d) + e) + f) + g)$