# 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 start a derivation of $\alpha$. If $\alpha$ derives a string shorter than $k$, that string is included.

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

1. If $\alpha$ is a terminal $a$: Set $FIRST^k(\alpha) = \{a\}$.
2. If $\alpha$ is a non-terminal $A$:
   - Initialize $FIRST^k(A) = \emptyset$.
   - For each production $A \to \beta$, add $FIRST^k(\beta)$ to $FIRST^k(A)$.
   - Repeat until no new strings are added.
3. If $\alpha$ is a sequence $\alpha_1\alpha_2 \dots \alpha_n$:
   - Initialize $FIRST^k(\alpha) = \emptyset$.
   - For $i = 1$ to $n$:
     - If $\alpha_1, \dots, \alpha_{i-1}$ are all nullable, add all strings $s \in FIRST^k(\alpha_i)$ to $FIRST^k(\alpha)$.
     - For each $s \in FIRST^k(\alpha_i)$ with $|s| < k$, if $\alpha_{i+1}, \dots, \alpha_n$ are nullable or $i = n$, add $s$ to $FIRST^k(\alpha)$; otherwise, concatenate $s$ with $FIRST^{k-|s|}(\alpha_{i+1} \dots \alpha_n)$.
   - If all $\alpha_1, \dots, \alpha_n$ are nullable, add $\epsilon$ to $FIRST^k(\alpha)$.
4. Truncate all strings in $FIRST^k(\alpha)$ to their first $k$ terminals.

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 a derivation from the start symbol $S$. If $A$ can appear at the end of a derivation, include the end marker \$.

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

1. Initialize:
   - For the start symbol $S$, set $FOLLOW^k(S) = \{\$\}$.
   - For all other non-terminals $A$, set $FOLLOW^k(A) = \emptyset$.
2. For each production $B \to \alpha A \beta$:
   - Add all non-$\epsilon$ strings from $FIRST^k(\beta)$ to $FOLLOW^k(A)$.
   - If $\beta$ is nullable, add $FOLLOW^k(B)$ to $FOLLOW^k(A)$.
3. Repeat step 2 until no new strings are added to any $FOLLOW^k(A)$.
4. Truncate all strings in $FOLLOW^k(A)$ to their first $k$ terminals.

## LR Items

An **LR(k)** item is a production with a dot (•) 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 $CLOSURE(J)$.

## LR(k) Automaton Construction

1. **Augment the Grammar**: Add a new start symbol $S'$ and production $S' \to S\$$, where $S$ is the original start symbol.
2. **Initialize the First State**: Compute the initial state $I_0 = CLOSURE(\{[S' \to \bullet S\$]\})$, where the dot (•) indicates the current position in the production.
3. **Compute CLOSURE**: For a set of items $I$
4. **Compute GOTO**: For a state $I$ and symbol $X$ (terminal or nonterminal)
5. **Build the State Machine**:
   - Initialize the set of states $\mathcal{C} = \{I_0\}$ and worklist $W = \{I_0\}$.
   - While $W \neq \emptyset$:
     - Remove a state $I$ from $W$.
     - For each symbol $X$ where $GOTO(I, X) \neq \emptyset$:
       - 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.
6. **Construct the Parsing Table**:
   - For each state $I \in \mathcal{C}$:
     - If $[A \to \alpha \bullet a\beta, w] \in I$, $a$ is a terminal, and GOTO$(I, a) = J$, add *shift* to state $J$ for $a$ in the action table.
     - If $[A \to \alpha\bullet, w] \in I$ and $A \neq S'$, add *reduce* by $A \to \alpha$ for lookahead $w$ in the action table.
     - If $[S' \to S \bullet \$, w] \in I$, add *accept* for lookahead \$ in the action table.
     - For nonterminal $A$, if GOTO$(I, A) = J$, add *goto* state $J$ for $A$ in the goto table.
   - **Check for Conflicts**: For each state $I$ and lookahead $a$ in the action table:
     - If the cell for $(I, a)$ has both a *shift* and a *reduce* action, report a *shift-reduce* conflict.
     - If the cell for $(I, a)$ has multiple *reduce* actions, report a *reduce-reduce* conflict.
     - If any conflicts exist, the grammar is not LR(k).

## 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 $[state, w]$.

## Grammar Proofs

Note: if a language described by a grammar is asked, the grammar can be altered. The given grammar might be alterd to achieve a certain parser type, but the language remains the same.

**Preliminary Checks**

Before testing for specific parser types, perform these essential checks:

1. **Well-formed grammar verification**
   - Ensure the CFG has no useless symbols (nonterminals that cannot derive any terminal string or cannot be reached from the start symbol).
   - Remove useless symbols using standard algorithms (identify productive and reachable symbols).
2. **Ambiguity check**
   - An ambiguous grammar cannot be guaranteed to be LL(k), SLR(k), LALR(k), or LR(k) for any $k$, as ambiguity may lead to parsing conflicts.
   - Test by attempting to construct multiple leftmost or rightmost derivations for the same string.
   - If ambiguous, the grammar may require modification to be used with deterministic parsers.
3. **Left recursion elimination**
   - For LL(k): Grammar must be free of left recursion (direct or indirect).
   - Apply left recursion elimination techniques if needed.
   - For LR-based parsers (SLR, LALR, LR): Left recursion is generally acceptable and does not require elimination.

**LL(k) Parser Check**

**Definition:** An LL(k) grammar can be parsed top-down by a deterministic pushdown automaton using $k$ lookahead symbols, ensuring a unique production choice for each nonterminal and lookahead, without backtracking.

**Algorithm Steps**

1. **Compute FIRST$_k$ sets**
2. **Compute FOLLOW$_k$ sets**
3. **Construct LL(k) parsing table**
   - For each production $A \to \alpha$, compute the lookahead set:
     - If $\alpha$ derives a string $w$ of length $\geq k$, include FIRST$_k(w)$.
     - If $\alpha$ derives a string shorter than $k$ (including $\epsilon$), include prefixes of FIRST$_k(\alpha) \cdot$ FOLLOW$_k(A)$ of length $k$.
   - Place the production in table cell $[A, t]$ for each terminal string $t$ in the lookahead set.
4. **Check for conflicts**
   - The grammar is LL(k) if no table cell $[A, t]$ contains multiple productions, ensuring deterministic parsing.

**LR(k) Parser Check**
**Definition:** A grammar is LR(k) if a bottom-up parser can deterministically construct a rightmost derivation in reverse using a parsing table, $k$ lookahead symbols, and canonical LR(k) item sets.
**Algorithm Steps**

1. **Construct LR(k) item sets**
2. **Build parsing table**
3. **Check for conflicts**
   - Grammar is LR(k) if each table entry has at most one action.
   - Shift-reduce conflict: same entry has shift and reduce.
   - Reduce-reduce conflict: same entry has multiple reduces.
4. **Test incrementally**
   - Start with $k = 0$ (LR(0), reduce actions apply for all terminals in FOLLOW$(A)$).
   - If conflicts exist, increment $k$ (e.g., $k = 1$) and repeat.

**SLR(k) Parser Check**
**Definition:** SLR(k) (Simple LR(k)) is a bottom-up parsing technique that simplifies LR(k) by using FOLLOW$_k(A)$ sets to approximate k-token lookaheads for reduce actions. It is less powerful than LR(k) but easier to compute.
**Algorithm Steps**

1. **Build LR(0) DFA**
   - Construct the DFA where states are sets of LR(0) items (productions with a dot) and transitions represent shifts on terminals or nonterminals.
2. **Compute FOLLOW$_k$ sets**
3. **Construct SLR(k) parsing table**
   - Add shift and goto actions from the LR(0) DFA.
   - For each state $I$ with a completed item $[A \to \alpha\cdot]$, add a reduce action by $A \to \alpha$ for each $t \in$ FOLLOW$_k(A)$.
   - Check for conflicts (shift-reduce or reduce-reduce). If none exist, the grammar is SLR(k).
4. **Parse input**
   - Use the parsing table with a stack-based algorithm to process input strings, applying shift, reduce, or goto actions based on the current state and lookahead tokens.

**LALR(k) Parser Check**
**Definition:** LALR(k) (Look-Ahead LR) is a parsing technique that reduces the size of an LR(k) parser by merging states with the same core (items without lookaheads) while retaining k-symbol lookaheads. LALR(1) is most commonly used due to its efficiency.

**Algorithm Steps**

1. **Construct LR(0) DFA**
2. **Compute LALR(k) Lookaheads**
   - For each LR(0) item $[A \to \alpha \cdot \beta]$ in a state, compute the set of terminal strings of length up to $k$ (lookaheads) that can follow $\beta$.
   - Propagate lookaheads through the DFA by tracing paths from states where $\beta$ leads to a terminal or the endmarker.
   - Merge lookaheads for items in states with the same core (same LR(0) items) to form LALR(k) states.
3. **Construct LALR(k) Parsing Table**
   - Use the LR(0) DFA for shift and goto actions: for state $s$ and symbol $X$, transition to state $s'$ implies action shift (if $X$ is a terminal) or goto $s'$ (if $X$ is a nonterminal).
   - For each item $[A \to \alpha\cdot, w]$ in a state, where $w$ is a lookahead string of length up to $k$, add a reduce action by production $A \to \alpha$ on lookahead $w$.
4. **Check for Conflicts**
   - Verify the parsing table has no shift-reduce or reduce-reduce conflicts.
   - The grammar is LALR(k) if no conflicts exist. If conflicts occur, modify the grammar or use precedence rules to resolve them.

**Parser Type Relationships and Optimization**

- LR$(k) \supseteq$ LALR$(k) \supseteq$ SLR$(k)$
- LL$(k) \subsetneq$ LR$(k)$ for all $k \geq 0$
- LR(0) $\subsetneq$ SLR(1) $\subsetneq$ LALR(1) $\subsetneq$ LR(1) $\subsetneq$ LR(2) $\subsetneq \cdots \subsetneq$ LR$(k) \subsetneq$ LR$(k+1)$ $\subsetneq \cdots$

## Example Languages

**LR(0) language that is not LL(1)**
$L = \{a^n b^n | n \geq 0\}$
**LR(0) language that is not LL(k) for any k**
$L = \{a^n b^n c^m | n, m \geq 1\}$
**LL(2) language that is not LL(1)**
$L = \{a^n bc^n d \mid n \geq 1\} \cup \{a^n cb^n d \mid n \geq 1\}$
**CFL that is not LR(1)**
$L = \{ww^R \mid w \in \{a, b\}^*\}$ where $w^R$ is the reverse of $w$
**Example Grammars**
**LL(k) Grammar**
**LL(1) grammar that is not strongly LL(1)**
Theorem 5.4 "All LL(1) grammars are also strong LL(1), i.e. the classes of LL(1) and strong LL(1) grammars coincide." **LL(1) grammar that is not LALR(1)**
$S \to aX$
$S \to Eb$
$S \to Fc$
$X \to Ec$
$X \to Fb$
$E \to A$
$F \to A$
$A \to \epsilon$
**LL(2) grammar that is not strongly LL(2)**
$S \to aAa$
$S \to bABa$
$A \to b$
$A \to \epsilon$
$B \to b$
$B \to c$
**LL(2) and SLR(1) grammar but neither LL(1) nor LR(0)**
$S \to ab|ac|a$

**LALR(k) Grammar**
**LALR(1) grammar that is not SLR(1)**
$S \to Aa$
$S \to bAc$
$S \to dc$
$A \to d$

**LR(k) Grammar**
**LR(1) grammar that is not LALR(1) and not LL(1)**
$S \to aAd$
$S \to bBd$
$S \to aBe$
$S \to bAe$
$A \to c$
$B \to c$

**LR(k + 1) grammar that is not LR(k) and not LL(k + 1)**
$S \to Ab^k c$
$S \to Bb^k d$
$A \to a$
$B \to a$

**Context Free Grammar**
**CFG that is not LR(k)**
$S \to aAc$
$A \to bAb$
$A \to b$

**CFG that is not LR(0)**
$S \to a$
$S \to abS$

**SLR(k) Grammar**
**SLR(1) grammar that is not LR(0)**
$S \to Exp$
$Exp \to Exp + Prod$
$Exp \to Prod$
$Prod \to Prod * Atom$
$Prod \to Atom$
$Atom \to Id$
$Atom \to (Exp)$

**Special examples**
**LL(1) language with a non-LL(1) grammar**
$L = \{w \mid w$ is a string of balanced parentheses$\}$
Grammar:
$S \to SS$
$S \to (S)$
$S \to \epsilon$

## 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 Σ. 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 Σ. 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 Σ. 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. $((((((a + b) + c) + d) + e) + f) + g)$ **Q:** How do you enforce precedence in a grammar? Enforced by structuring the grammar rules to reflect the desired hierarchy of operations, typically using separate non-terminals for each precedence level. For example, in a simple arithmetic grammar, you define rules like Expr -> Expr + Term — Term and Term -> Term * Factor — Factor to ensure multiplication (*) is evaluated before addition (+).