Compilers

Key Definitions and Concepts

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. **Context-Free Grammar (CFG)** A grammar consisting of rules of the form $A \to \alpha$, where A is a nonterminal, and α is a string of terminals andor nonterminals. CFGs generate context-free languages, which can be recognized by pushdown automata. **Context-Free Language (CFL)**: if it can be generated by a context-free grammar (CFG) **Finite-State Machine (FSM)** A model of computation with states, transitions, and an alphabet. A deterministic finite automaton (DFA) has exactly one transition per input symbol per state, while a nondeterministic finite automaton (NFA) may have multiple or ϵ-transitions. **Pushdown Automaton (PDA)** An automaton with a stack, capable of recognizing context-free languages. A deterministic PDA (DPDA) has at most one possible move per configuration, while a nondeterministic PDA (NPDA) may have multiple. **LL(k) Grammar** A grammar that can be parsed by a top-down parser with k-token lookahead, where the parser can deterministically choose the next production based on the first k tokens. **LR(k) Grammar** A grammar that can be parsed by a bottom-up parser with k-token lookahead, using a shift-reduce strategy. Variants include SLR(1) (simple LR), LALR(1) (lookahead LR), and LR(1) (canonical LR). **First(k) Set** For a nonterminal A, the set of all possible k-token prefixes that can begin a string derived from A. **Follow(k) Set** For a nonterminal A, the set of all possible k-token prefixes that can appear immediately after A in a derivation. **Closure in Parsing** In LR parsing, the closure of a set of items (e.g., $A \to \alpha \cdot \beta$) includes all items that can be reached by following nonterminal transitions (adding rules like $B \to \cdot\gamma$ if β starts with B). **Regular Expression** A pattern describing a regular language, using symbols like '*' (zero or more), '+' (one or more), '|' (union), and '()' (grouping). **Longest Match Principle (Maximal munch)** In lexical analysis, the scanner selects the longest prefix of the input that matches a token pattern. **Strong Language** A context-free language is strong if it can be parsed deterministically by an LR(1) parser. Strong languages are a proper subset of context-free languages and include all LR(k) languages. They are characterized by having unambiguous grammars that can be parsed bottom-up with bounded lookahead. **Strong Grammar** A context-free grammar is strong if it generates a strong language and can be parsed deterministically by an LR(1) parser without conflicts. Strong grammars are unambiguous and have the property that every viable prefix can be extended to a complete derivation in at most one way with bounded lookahead.

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. $((((((a+b)+c)+d)+e)+f)+g)$

First and Follow Sets

The $\mathbf{FIRST^k}(\alpha)$ set for a string α (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 α. If α derives a string shorter than k, the entire derived string is included.

Steps to Compute $FIRST^k(\alpha)$: 1. If α is a terminal a: Add the string a (of length 1) to $FIRST^k(\alpha)$. 2. If α 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 ϵ. - 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 α can derive ϵ, include ϵ in $FIRST^k(\alpha)$. 3. If α 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 β can derive ϵ, 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 α and expect to see β, 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 $\mathbf{CLOSURE(I)}$ 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 β 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 $\mathbf{GOTO(I, X)}$ 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 $\mathbf{LR(k)}$ 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.

Grammar Proofs

Preliminary Checks 1) Ensure the grammar is well-formed: The CFG should have no useless symbols (nonterminals that cannot derive any string or cannot be reached from the start symbol). Remove them using standard algorithms (e.g., find productive and reachable symbols). 2) Check for ambiguity: An ambiguous grammar (one that allows multiple parse trees for the same string) cannot be LL(k), SLR(k), LALR(k), or LR(k) for any k. Test for ambiguity by attempting to construct multiple leftmost or rightmost derivations for a string. If ambiguous, the grammar fails for all these parser types. 3) Eliminate left recursion: For LL(k), the grammar must be free of left recursion (direct or indirect). Use left recursion elimination techniques if needed. For LR-based parsers (SLR, LALR, LR), left recursion is not an issue.

Check for LL(k) A grammar is LL(k) if it can be parsed top-down with k lookahead symbols without backtracking, using a deterministic parsing table. Steps: 1. Compute FIRST_k sets: For each nonterminal A, compute the set of all possible strings of length $\leq k$ that can be derived from A. For terminals, $\text{FIRST}_k(a) = \{a\}$ (or prefixes of length k). 2. Compute FOLLOW_k sets: For each nonterminal A, compute the set of strings of length $\leq k$ that can follow A in a derivation. 3. Construct the LL(k) parsing table: For each production $A \to \alpha$, compute the lookahead sets as $\text{FIRST}_k(\alpha \cdot \text{FOLLOW}_k(A))$. Place the production in the table at $[A, t]$ for each terminal string t in the lookahead set. 4. Check for conflicts: The grammar is LL(k) if the parsing table has no conflicts (i.e., no cell $[A, t]$ contains multiple productions). 5. Test for each k: If the table has conflicts for $k = 1$, try $k = 2$ or $k = 3$. If conflicts persist, the grammar is not LL(k) for the tested k. - For k = 0: LL(0) grammars are rare and require each nonterminal to have a single production with distinct terminals in its derivation. Check if the grammar is essentially deterministic without lookahead. Proof: If the table is conflict-free for some k, the grammar is LL(k). Provide the parsing table as evidence. If conflicts exist for all $k \leq 3$, state the conflicting entries to prove it's not LL(k).

Check for LR(k) A grammar is LR(k) if it can be parsed bottom-up with k lookahead symbols using a deterministic LR parsing table based on the canonical collection of LR(k) items. Steps: 1. Construct the LR(k) item sets: An LR(k) item is of the form $[A \to \alpha \cdot \beta, w]$, where α is the part already parsed, β is the remaining part, and w is a lookahead string of length k. Build the DFA of LR(k) items (states) using closure and goto operations. 2. Build the parsing table: For each state, define actions (shift, reduce, accept) based on the items: - Shift: If $[A \to \alpha \cdot a\beta, w]$ and $\text{goto}(I, a) = J$, add a shift to state J. - Reduce: If $[A \to \alpha\cdot, w]$, add a reduce by $A \to \alpha$ for lookahead w. - Accept: If $[S' \to S\cdot, \epsilon]$ (augmented start symbol), accept on end-of-input. 3. Check for conflicts: The grammar is LR(k) if there are no shift-reduce or reduce-reduce conflicts in the table. 4. Test for each k: Start with $k = 0$. If conflicts arise, try $k = 1, 2, 3$. Larger k resolves conflicts by distinguishing lookaheads. - For k = 0: Use empty lookaheads ($w = \epsilon$). Conflicts are common, as LR(0) is restrictive. - Proof: If the table is conflict-free for some k, the grammar is LR(k). Show the DFA and table. If conflicts exist, list them to prove it's not LR(k).

Check for SLR(k) SLR(k) is a simplified version of LR(k) that uses FOLLOW sets for lookaheads instead of precise LR(k) lookaheads, making it less powerful but easier to compute. - Steps: 1. Build the LR(0) DFA: Construct the DFA of LR(0) items (same as LR(k) but with no lookahead, i.e., $k = 0$). 2. Compute FOLLOW_k sets: For each nonterminal A, compute the set of terminal strings of length $\leq k$ that can follow A. 3. Construct the SLR(k) parsing table: - Shift and goto actions are the same as in LR(0). - For an item $[A \to \alpha\cdot]$ in state I, add a reduce by $A \to \alpha$ for each terminal string $t \in \text{FOLLOW}_k(A)$. 4. Check for conflicts: The grammar is SLR(k) if the table has no shift-reduce or reduce-reduce conflicts. 5. Test for each k: Try $k = 0, 1, 2, 3$. SLR(0) uses empty lookaheads, often leading to conflicts. - Proof: If conflict-free, the grammar is SLR(k); provide the table. If conflicts exist, show conflicting actions to prove it's not SLR(k).

Check for LALR(k) - Definition: LALR(k) is a compromise between SLR(k) and LR(k), using the LR(0) DFA but with lookaheads computed by merging LR(k) states. - Steps: 1. Build the LR(0) DFA: Same as for SLR(k). 2. Compute LALR(k) lookaheads: Propagate lookaheads through the LR(0) DFA to compute the set of lookahead strings of length $\leq k$ for each item $[A \to \alpha\cdot]$. This is done by analyzing spontaneous generation and propagation of lookaheads (a simplified version of LR(k) lookaheads). 3. Construct the LALR(k) parsing table: - Shift and goto actions are identical to LR(0). - For an item $[A \to \alpha\cdot, w]$, add a reduce by $A \to \alpha$ for lookahead w. 4. Check for conflicts: The grammar is LALR(k) if the table has no conflicts. 5. Test for each k: Try $k = 0, 1, 2, 3$. LALR(0) is equivalent to SLR(0) in practice, as lookaheads are minimal. - Proof: If conflict-free, the grammar is LALR(k); show the table and lookaheads. If conflicts exist, list them to prove it's not LALR(k).

Relationships and Optimization - Hierarchy: If a grammar is LR(k), it is LALR(k). If it is LALR(k), it is SLR(k). If it is LL(k), it is not necessarily LR(k), but LL(k) grammars are often simpler. Use this to optimize checks: - If the grammar is not SLR(k), it cannot be LALR(k) or LR(k). - If it is LR(k), it is also LALR(k) and SLR(k). - If it is LL(k), it may still need separate checks for LR-based parsers. - Start with k = 1: Most practical grammars are tested for $k = 1$. $k = 0$ often fails due to insufficient lookahead, and $k > 1$ is rarely needed for well-designed grammars.

Document the Proof - For each type (LL, SLR, LALR, LR) and each k, provide: - The parsing table or DFA (as applicable). - Any conflicts found (e.g., specific table entries with multiple actions). - A conclusion: "The grammar is [type](k) because the table is conflict-free" or "The grammar is not [type](k) due to conflicts at [state/entry]."

Notes For k = 0: LL(0) and SLR(0)/LALR(0) are restrictive and rarely practical. Focus on $k = 1$ unless specified. Ambiguity: If the grammar is ambiguous, it fails all tests. Check this first to avoid unnecessary work.

Other

Rewriting Grammars for Precedence Identify Operators: E.g., + (addition), * (multiplication). Assign Precedence: Higher precedence (e.g., *) gets lower-level nonterminals. Lower precedence (e.g., +) gets higher-level nonterminals. Rewrite CFG: For multiplication before addition: Exp → Exp + Term — Term, Term → Term * Factor — Factor. Add parentheses: Factor → (Exp) — cst. Verify: Ensure parse trees respect operator precedence.

LLVM IR LLVM IR: A low-level, platform-independent representation used in compilers. Unlimited Registers: Uses an infinite set of virtual registers. Static Single Assignment (SSA): Each variable is assigned exactly once, simplifying optimization. Basic Blocks: Sequences of instructions with a single entry and exit point. Must end with a terminator (e.g., branch, return). Jumps: Target the start of basic blocks, ensuring structured control flow.

Obtaining an LL(1) grammar Three main obstacles to LL(1) and their solutions: 1. Ambiguity: Grammar must be unambiguous first. Use operator precedence and associativity rules. 2. Left Recursion: Eliminate by converting to right recursion. - $A \to A\alpha|\beta$ becomes $A \to \beta A'$, $A' \to \alpha A'|\epsilon$ 3. Common Prefixes: Use left factoring to eliminate shared prefixes. - $A \to \alpha\beta|\alpha\gamma$ becomes $A \to \alpha A'$, $A' \to \beta|\gamma$ Key Properties: - LL(k) grammars are necessarily unambiguous - Parser decides rules based on lookahead tokens - First and Follow sets must be disjoint for LL(1)

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) $S \to ab$
$S \to ac$
$S \to a$

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

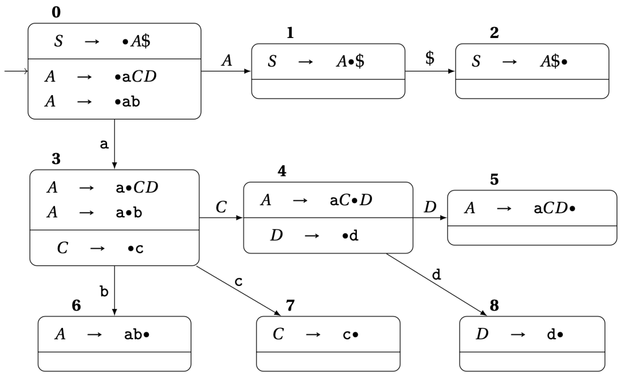
LL(1) language with a non-LL(1) grammar $L = \{w \mid w \text{ is a string of balanced parentheses}\}$

Grammar:
$S \to SS$
$S \to (S)$
$S \to \epsilon$

Canonical Finite State Machine Example

Grammar:
$S \to A\$$
$A \to aCD$
$A \to ab$
$C \to c$
$D \to d$

CFSM:



The language of this grammar is LR(1).