Compilers

Key Definitions and Concepts

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. **Regular Language** A language recognized by a DFA or NFA, or described by a regular expression. Regular languages are less expressive than context-free languages. **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** In lexical analysis, the scanner selects the longest prefix of the input that matches a token pattern (Maximal munch). **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.

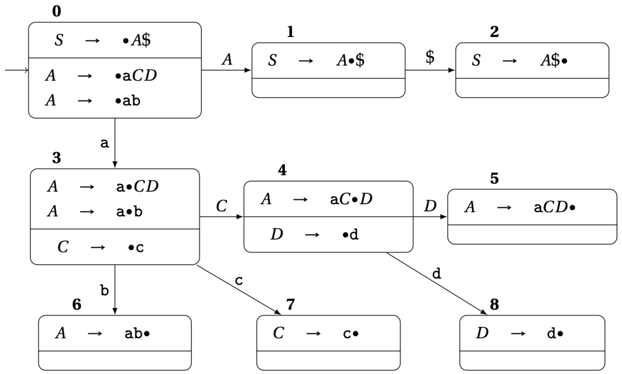
The $\mathbf{FOLLOW^k(A)}$ set for a non-terminal A contains all sequences of up to k terminals that can appear immediately after A in some derivation from the start symbol. If A appears at the end of a derivation, include the end-of-input marker (e.g., \$).

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.

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).

Grammars

Grammar Fundamentals

Context-Free Grammar (CFG): $G = (V, T, P, S)$ where: - V: finite set of non-terminals - T: finite set of terminals - P: finite set of productions $A \to \alpha$ where $A \in V$, $\alpha \in (V \cup T)^*$ - S: start symbol $\in V$

Derivation: $\alpha \Rightarrow \beta$ if $\alpha = \gamma A \delta$, $\beta = \gamma\omega\delta$, and $A \to \omega \in P$

Language: $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$

Ambiguous Grammar: A grammar where some string has multiple parse trees.

Grammar Transformations

Left Recursion: $A \to A\alpha \mid \beta$ - *Elimination*: Replace with $A \to \beta A'$, $A' \to \alpha A' \mid \epsilon$

Left Factoring: $A \to \alpha\beta_1 \mid \alpha\beta_2$ - *Factor out*: $A \to \alpha A'$, $A' \to \beta_1 \mid \beta_2$

Grammar Hierarchy: LL(k) $\subset$ LR(k), SLR(k) $\subset$ LALR(k) $\subset$ LR(k)

LL(k) Grammar

A CFG is LL(k) if: The grammar can be parsed by a top-down parser with k lookahead tokens. The parsing table has no ambiguity and is deterministic.

SLR(k) Grammar

A CFG is SLR(k) if: It is LR(k) (i.e., it can be parsed by an LR(k) parser). The parsing table has no shift/reduce or reduce/reduce conflicts when using the FOLLOW sets to resolve conflicts.

LALR(k) Grammar

A CFG is LALR(k) if: It is LR(k). The parsing table can be constructed using LALR(k) lookahead by merging states with the same core (LALR(k) states).
LALR(k) is a refinement of SLR(k) that reduces the number of states while maintaining LR(k) power.

LR(k) Grammar

A CFG is LR(k) if: It has no shift/reduce or reduce/reduce conflicts for a given k. The grammar can be parsed by an LR(k) parser using the complete parsing table.

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)
English language

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)
$S \rightarrow a$
$S \rightarrow abS$

SLR(k) Grammar

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

Special examples

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

Parsing Grammar Proofs

LL(k) Proof

A grammar is **LL(k)** if:

1. No left recursion exists.
2. For every non-terminal A with productions $A \rightarrow \alpha \mid \beta$:

$$\text{FIRST}_k(\alpha \cdot \text{FOLLOW}_k(A)) \cap \text{FIRST}_k(\beta \cdot \text{FOLLOW}_k(A)) = \emptyset$$

   (If α/β can derive ϵ, include $\text{FOLLOW}_k(A)$.)

SLR(k) Proof

A grammar is **SLR(k)** if:

1. Construct SLR(1) states (LR(0) items + FOLLOW for reduce actions).
2. No *shift-reduce* or *reduce-reduce* conflicts in the parsing table.
3. Conflicts tested using FOLLOW sets (not lookaheads).

LR(k) Proof

A grammar is **LR(k)** if:

1. Construct canonical LR(k) states (with full lookaheads).
2. No conflicts in the parsing table (even with precise lookaheads).

LALR(k) Proof

A grammar is **LALR(k)** if:

1. Merge LR(1) states with identical *cores* (same productions, different lookaheads).
2. Ensure no conflicts arise after merging (common in practice but weaker than LR).

Hierarchy

$$\text{SLR} \subset \text{LALR} \subset \text{LR}, \quad \text{LL} \subset \text{CFG (non-overlapping)}$$

Key Notes: - For **LL(k)**: Compute FIRST_k and FOLLOW_k rigorously. - For **SLR**: Conflicts = overlap between shift terminals and FOLLOW of reduced non-terminal. - **LALR** merges LR(1) states; conflicts here imply not LALR but might still be LR. - **Left recursion** invalidates LL(k); **ambiguity** invalidates all.

Constructing an LR Parsing Table

To determine whether a grammar is LR, SLR, or LALR, an LR parsing table must be constructed. The steps are as follows:

1. **Augment the Grammar**: Add a new start symbol S' and production $S' \rightarrow S$, where S is the original start symbol. This ensures the parser recognizes when it has fully parsed the input.
2. **Compute LR(0) Items**: Create the set of LR(0) items for the grammar. Each item represents a position within a production (e.g., $A \rightarrow \alpha \cdot \beta$).
3. **Closure Operation**: For each item $A \rightarrow \alpha \cdot B\beta$, add all productions of B with a dot at the beginning ($B \rightarrow \cdot\gamma$) to the closure. Repeat until no new items can be added.
4. **Goto Function**: For each item set I and each grammar symbol X (terminal or non-terminal), compute the *goto*(I, X) by advancing the dot over X in all items of I and taking the closure of the result.
5. **Build the Canonical Collection**: Start with the initial state $I_0 = \text{closure}(\{S' \rightarrow \cdot S\})$. Use the *goto* function to generate all reachable states. This forms the canonical collection of LR(0) item sets.
6. **Action and Goto Tables**:
   - For each state I_i and terminal a, if $goto(I_i, a)$ leads to state I_j, add a **shift** action to the parsing table: $\text{ACTION}[i, a] = \text{shift } j$.
   - For each state I_i containing an item $A \rightarrow \alpha\cdot$ (complete production), add a **reduce** action: $\text{ACTION}[i, a] = \text{reduce } A \rightarrow \alpha$ for all $a \in \text{FOLLOW}(A)$ (for SLR). For LR(1) and LALR, use the lookahead symbols associated with the item.
   - For the state containing $S' \rightarrow S\cdot$, add an **accept** action: $\text{ACTION}[i, \$] = \text{accept}$.
7. **Check for Conflicts**:
   - **Shift-Reduce Conflict**: Occurs if a state has both a shift and reduce action for the same terminal.
   - **Reduce-Reduce Conflict**: Occurs if a state has multiple reduce actions for the same terminal.
   - If no conflicts exist, the grammar is:
     - **SLR**: If conflicts are resolved using FOLLOW sets.
     - **LR(1)**: If conflicts are resolved using precise lookahead symbols.
     - **LALR**: If the grammar remains conflict-free after merging LR(1) states with identical cores.
8. **Example Construction**: Consider the grammar: Augment the grammar, compute the LR(0) items, build the canonical collection, and populate the parsing table. Check for conflicts to classify the grammar.

Notes on Classification: - If the grammar has no conflicts in the LR(0) table, it is **LR(0)**. - If conflicts are resolved using FOLLOW sets, it is **SLR**. - If conflicts are resolved using precise lookaheads, it is **LR(1)**. - If merging LR(1) states resolves conflicts, it is **LALR**.