Compilers v2

By Cedric De Vijt

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 α 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 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 $\mathbf{FOLLOW^k}(\mathbf{A})$ 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 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 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

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 string or cannot be reached from the start symbol)
   - Remove useless symbols using standard algorithms (find productive and reachable symbols)
2. **Ambiguity check**
   - An ambiguous grammar cannot be LL(k), SLR(k), LALR(k), or LR(k) for any k
   - Test by attempting to construct multiple leftmost or rightmost derivations for the same string
   - If ambiguous, the grammar fails for all parser types
3. **Left recursion elimination**
   - For LL(k): 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 acceptable

LL(k) Parser Check

Definition: A grammar is LL(k) if it can be parsed top-down with k lookahead symbols without backtracking, using a deterministic parsing table.

Algorithm Steps

1. **Compute FIRST$_k$ sets**
   - For each nonterminal A, compute all possible strings of length $\leq k$ that can be derived from A
   - For terminals: FIRST$_k(a) = \{a\}$ (or prefixes of length k)
2. **Compute FOLLOW$_k$ sets**
   - For each nonterminal A, compute strings of length $\leq k$ that can follow A in a derivation
3. **Construct LL(k) parsing table**
   - For each production $A \to \alpha$, compute lookahead sets as FIRST$_k(\alpha \cdot$ FOLLOW$_k(A))$
   - Place production in table at $[A, t]$ for each terminal string t in the lookahead set
4. **Check for conflicts**
   - Grammar is LL(k) if parsing table has no conflicts
   - No cell $[A, t]$ should contain multiple productions

LR(k) Parser Check

Definition: A grammar is LR(k) if it can be parsed bottom-up with k lookahead symbols using a deterministic LR parsing table based on canonical LR(k) items.

Algorithm Steps

1. **Construct LR(k) item sets**
   - LR(k) item format: $[A \to \alpha \cdot \beta, w]$
   - α: part already parsed, β: remaining part, w: lookahead string of length k
   - Build DFA of LR(k) items using closure and goto operations
2. **Build parsing table**
   - **Shift:** If $[A \to \alpha \cdot a\beta, w]$ and goto$(I, a) = J$, add shift to state J
   - **Reduce:** If $[A \to \alpha \cdot, w]$, add reduce by $A \to \alpha$ for lookahead w
   - **Accept:** If $[S' \to S \cdot, \epsilon]$ (augmented start), accept on end-of-input
3. **Check for conflicts**
   - Grammar is LR(k) if no shift-reduce or reduce-reduce conflicts exist
4. **Test incrementally**
   - Start with $k = 0$ (LR(0) uses empty lookaheads $w = \epsilon$)

SLR(k) Parser Check

Definition: SLR(k) is a simplified LR(k) that uses FOLLOW sets for lookaheads instead of precise LR(k) lookaheads. Less powerful but easier to compute.

Algorithm Steps

1. **Build LR(0) DFA**
   - Construct DFA of LR(0) items (same as LR(k) but with $k = 0$)
2. **Compute FOLLOW$_k$ sets**
   - For each nonterminal A, compute terminal strings of length $\leq k$ that can follow A
3. **Construct SLR(k) parsing table**
   - Shift and goto actions same as LR(0)
   - For item $[A \to \alpha \cdot]$ in state I, add reduce by $A \to \alpha$ for each $t \in$ FOLLOW$_k(A)$
4. **Check for conflicts and test incrementally**
   - Grammar is SLR(k) if table has no conflicts
   - Try $k = 0, 1, 2, 3$ (SLR(0) often has conflicts due to empty lookaheads)

LALR(k) Parser Check

Definition: LALR(k) is a compromise between SLR(k) and LR(k), using LR(0) DFA but with lookaheads computed by merging LR(k) states.

Algorithm Steps

1. **Build LR(0) DFA**
   - Same as for SLR(k)
2. **Compute LALR(k) lookaheads**
   - Propagate lookaheads through LR(0) DFA
   - Compute lookahead strings of length $\leq k$ for each item $[A \to \alpha \cdot]$
   - Use spontaneous generation and propagation analysis (simplified LR(k) lookaheads)
3. **Construct LALR(k) parsing table**
   - Shift and goto actions identical to LR(0)
   - For item $[A \to \alpha \cdot, w]$, add reduce by $A \to \alpha$ for lookahead w
4. **Check for conflicts and test incrementally**
   - Grammar is LALR(k) if table has no conflicts

Parser Type Relationships and Optimization

Hierarchy	Optimization Strategy
LR(k) $\supseteq$ LALR(k) $\supseteq$ SLR(k)	If not SLR(k), cannot be LALR(k) or LR(k)
LL(k) $\not\subseteq$ LR(k)	LL(k) and LR(k) require separate checks
If LR(k), then LALR(k) and SLR(k)	Use hierarchy to optimize testing order

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 $L = \{w \mid w$ 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)