Compilers

Key Definitions and Concepts

Regular language: languages that can be recognized by finite automata or described by regular expressions
CFG: formal grammar where every production rule has a single non-terminal symbol on the left-hand side and a string of terminals andor non-terminals on the right-hand side.
CFL: if it can be generated by a context-free grammar (CFG)
LL(k) language: if there is an LL(k) grammar G_L that accepts it, i.e.: $L(G_L) = L$.
Maximal munch: the longest possible string that can be matched by a token is matched.
Strongly LL(k) Grammar: A $CFGG = \langle V, T, P, S\rangle$ is strong LL(k) iff, for all pairs of rules $A \to \alpha_1$ and $A \to \alpha_2$ in P. (with $\alpha_1 \neq \alpha_2$):
$First^k(\alpha_1 \cdot Follow^k(A)) \cap First^k(\alpha_2 \cdot Follow^k(A)) = \emptyset$

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).

First and Follow

Calculating FIRST

The $FIRST^k(\alpha)$ set for a non-terminal S is defined as: $FIRST^k(\alpha)$ includes terminals that appear as the first k symbols in any derivation of S.
To compute $FIRST^k(\alpha)$:

1. For each production $S \to \alpha$, add the first terminal of α to $FIRST^k(\alpha)$.
2. If α starts with a non-terminal A, include $FIRST^k(A)$.
3. Consider k-length lookahead, ensuring that $FIRST^k(\alpha)$ captures all terminals up to k symbols.

Calculating FOLLOW

The $FOLLOW^k(\alpha)$ set for a non-terminal S is defined as: $FOLLOW^k(\alpha)$ includes terminals that can appear immediately after S in some string derivable from the start symbol, considering up to k symbols.
To compute $FOLLOW^k(\alpha)$:

1. For each production $A \to \alpha S\beta$, add $FIRST^k(\beta)$ to $FOLLOW^k(\alpha)$.
2. If β can derive the empty string ϵ, add $FOLLOW^k(A)$ to $FOLLOW^k(\alpha)$.
3. Ensure that $FOLLOW^k(\alpha)$ includes all possible terminals appearing after S within k symbols.

Grammars

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.

Time Complexities

DFA: $\mathcal{O}(|w|)$
CFG: polynomial in $|w|$
Grammar (not necessarily context free): undecidable problem

Trivia

Q:Give an advantage of hand-built scanners over automata-based scanners? A:They can go beyond regular languages
Q: How can we make sure the sequence 'true' is always deamed a contant and not a string? A: Constants like 'true' should be recognized as a single token
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: Argue that the exact of both type checking and reachability analysis are in fact undecidable. A: The exact versions of both type checking and reachability analysis are undecidable because they would require solving the halting problem, which is proven to be undecidable. For type checking, determining the exact type of an expression in all possible cases would involve analyzing all potential program executions, which is equivalent to solving whether a program halts with certain inputs. Similarly, exact reachability analysis involves determining whether a specific program state can be reached, which also boils down to predicting the behavior of all possible executions, again leading to the halting problem. Since the halting problem is undecidable, both exact type checking and exact reachability analysis are also undecidable.

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)
$S \to a$
$S \to abS$

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

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