FTP_TheoComp Summary

Valentin Huber
valentinhuber.me

June 24, 2024

iif

if and only if

1 Elementary Definitions

Alphabet

A finite set of symbols such as letters, digits, etc.

Examples:

- $\Sigma_1 = \{a, b, c\}$
- $\Sigma_2 = \{a, b, c, \ldots, z\}$

Word

A word over an alphabet Σ is a string of finite length, in other words a finite sequence of symbols of Σ

Language

A (formal) language over an alphabet Σ is defined as a set of words (defined over Σ).

The letters w, u, and v are usually used for words, s, t, and r for single characters. Languages are usually denoted by capital letters. Examples (over $\Sigma = \{a, b, c\}$):

- $L_1 = \{a,\ b,\ aab,\ abbcc\}$, a finite language
- $L_2 = \{acbb,\ accbb,\ acccbb,\ accccbb,\ \ldots\}$, words starting with a, then $n > 0$ cs and finally bb

We only consider finite alphabets, otherwise important and useful properties are no longer valid. A language can be infinite, but each element (i.e. word) is finite.

1.1 Operations on Words

Word Concatenation

Appending the second word to the end of the first one, denoted by either $w_1 \cdot w_2$, or simply $w_1 w_2$

Empty String

ϵ, where $|\epsilon| = 0$ and $w \cdot \epsilon = \epsilon \cdot w = w$

Operations On Words

- Word length: Number of symbols in a word, denoted as $|w|$
- Mirror (reverse): $w = s_1 s_2 \ldots s_n \Rightarrow w^R = s_n \ldots s_2 s_1$
- Palindrome: if $w = w^R$, e.g. *sugus*
- Concatenation with self: $w^n, n \geq 0$, concat of w with itself n times

Set Of All Words

Σ^+ is the set of all words w with $|w| > 0$ over Σ, $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$

Σ^+ and Σ^* are infinite iif $\Sigma \neq \varnothing$

1.2 Operations on Languages

Language-Operations

Union: $L_1 \cup L_2$
Intersection: $L_1 \cap L_2$
Complement: $\overline{L} = \Sigma^* \backslash L$
Concatenation: $L_1 \cdot L_2 = \{w_1 \cdot w_2 | w_1 \in L_1, w_2 \in L_2\}$
Neutral Element: $L_\epsilon = \{\epsilon\} (L \cdot L_\epsilon = L_\epsilon \cdot L = L)$
Absorbant Element: $\varnothing (L \cdot \varnothing = \varnothing \cdot L = \varnothing \neq L_\epsilon)$
Self-Concatenation: $L^n, n \geq 0$, $(L^0 = L_\epsilon = \{\epsilon\})$

Iterative Or Kleene's Closure

L^*, is the set of words resulting of the concatenation of a finite number of words of L, where

$$L^+ = L^1 \cup L^2 \cup L^3 \ldots = \bigcup_{i \geq 1} L^i = L \cdot L^*$$

All words in a Kleene's closure have finite length. ϵ belongs to L^*, but not always to L^+ ($\varnothing^* = L^0 = \{\epsilon\}$).

2 Finite State Automata and Regular Languages

2.1 Finite State Automata (FSA)

A FSA is equivalent to an oriented graph and composed of

- finite set of states (drawn as circles),
- transition function describing the actions which allow to move from one state to another (these are the arrows), and
- the initial state in which the system is at the beginning (this state is indicated by an arrow pointing at it from nowhere).

2.2 Deterministic FSA (DFA)

Deterministic FSA (DFA)

A DFA is a 4-tuple $\langle Q, \Sigma, \delta, q_0 \rangle$ where

- Q is a finite set of states (therefore **F**SA),
- Σ is a finite set of symbols (alphabet),
- $\delta : Q \times \Sigma \to Q$ is a transition function,
- $q_0 \in Q$ is the initial state.

The extension of δ is the function $\delta^* : Q \times \Sigma^* \to Q$ which provides the next state for a sequence of symbols.

Extension Of Transition Function

For a state $q \in Q$, a word $w \in \Sigma^*$, and a symbol $s \in \Sigma$, the function $\delta^* : Q \times \Sigma^* \to Q$ is recursively defined as follows:

1. $\delta^*(q, \epsilon) = q$,
2. $\delta^*(q, ws) = \delta(\delta^*(q, w), s)$.

The sequence of symbols is actually a word.

DFA Language Recognizer

A DFA together with a set of final states $F \subseteq Q$, drawn as double circles.
A word $w \in \Sigma^*$ is recognized by a DFA of $\langle Q, \Sigma, \delta, q_0, F \rangle$ iif $\delta^*(q_0, w) \in F$
$L(M)$ is the language as recognized by such a language recognizer.

Regular Language

Every language recognized by a finite state recognizer automaton.

2.3 Non-deterministic Finite Automata (NFA)

Non-Deterministic Finite State Automaton (NFA)

A NFA is a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ similar to a DFA, except the transition function δ is defined as $Q \times \Sigma \to \mathcal{P}(Q)$ where $\mathcal{P}(Q)$ is the power function of Q.

Extension Of Transition Function For NFA

For a state $q \in Q$, a word $w \in \Sigma^*$, and a symbol $s \in \Sigma$, the function $\delta^* : Q \times \Sigma^* \to \mathcal{P}(Q)$ is defined as follows:

1. $\delta^*(q, \epsilon) = \{q\}$,
2. $\delta^*(q, ws) = \bigcup_{q' \in \delta^*(q,w)} \delta(q', s)$.

This means that for a word ws, the set of possible states is obtained by applying the transition function δ to all states reached by $\delta^*(q, w)$ with symbol s and taking the union of all resulting states.

Extension Of Transition Function To $\mathcal{P}(Q)$

$$\delta : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$$
$$(P, s) \mapsto \bigcup_{p \in P} \delta(p, s)$$
$$\delta^* : \mathcal{P}(Q) \times \Sigma^* \to \mathcal{P}(Q)$$
$$(P, w) \mapsto \bigcup_{p \in P} \delta^*(p, w)$$

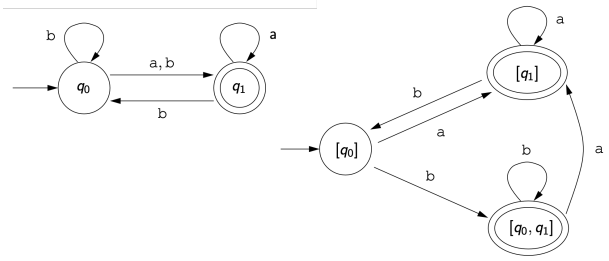
2.3.1 Equivalence of DFA and NFA

Equivalence Of Automata

Two finite state automata M_1 and M_2 are equivalent if and only if they recognize the same language, i.e., $L(M_1) = L(M_2)$.

Construction of DFA from NFA Let $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ be a NFA. We construct a DFA $M' = \langle Q', \Sigma, \delta', q_0', F' \rangle$ as follows:

- $Q' = \mathcal{P}(Q)$, the power set of Q, where an element of Q' will be denoted $[q_1, q_2, \ldots, q_i]$,
- $q_0' = \{q_0\}$,
- $F' = \{S \subseteq Q \mid S \cap F \neq \varnothing\}$, the set of members of Q' composed of at least one final state M,
- $\delta' : Q' \times \Sigma \to Q'$ with $\delta'(S, a) = \bigcup_{q \in S} \delta(q, a)$, so $\delta'([q_1, \ldots, q_i], s) = [p_1, \ldots, p_j] \iff \delta(\{q_1, \ldots, q_i\}, s) = \{p_1 \ldots, p_j\}$.



Proof We prove by induction on the length of w that for any $w \in \Sigma^*$:

$$\delta'^*(q_0', w) = \delta^*(q_0, w)$$

Base Case: For $w = \epsilon$, $\delta'^*(q_0', \epsilon) = q_0' = \{q_0\}$ and $\delta^*(q_0, \epsilon) = \{q_0\}$.

Induction Step: Let us assume that the above expression is true for some words of length $\leq m$. Let $s \in \Sigma$ and ws be a string of length $m + 1$. By definition of δ'^*

$$\delta'^*(q_0', ws) = \delta'(\delta'^*(q_0', w), s)$$

By induction hypothesis,

$$\delta'^*(q_0', w) = [p_1, \ldots, p_j] \iff \delta^*(q_0, w) = \{p_1, \ldots, p_j\}.$$

But, by definition of δ',

$$\delta'([p_1, \ldots, p_j], s) = [r_1, \ldots, r_k]$$
$$\iff \delta(\{p_1, \ldots, p_j\}, s) = \{r_1, \ldots, r_k\}$$

Consequently,

$$\delta'^*(q_0', ws) = [r_1, \ldots, r_k] \iff \delta^*(q_0, ws) = \{r_1, \ldots, r_k\}.$$

Finally, we have to add the following $\delta'^*(q_0', w) \in F'$ only when $\delta^*(q_0, w)$ contains a state of Q that belongs to F. Thus, $L(M) = L(M')$

Construction of DFA from NFA To convert a NFA $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ to a DFA $M' = \langle Q', \Sigma, \delta', q_0', F'\rangle$:

- Q': Create a new set of states, which is the power set of Q (all possible subsets of Q). Each state in the DFA represents a set of states in the NFA.
- q_0': The initial state in the DFA is the set containing the initial state of the NFA, q_0.
- F': The final states in the DFA are all sets of states in Q' that include at least one final state from F.
- δ': Define the transition function for the DFA. For each set of states S in Q' and each input symbol $a \in \Sigma$ (start with q_0):
  1. Compute $\delta(S, a)$ by finding the set of all possible states the NFA can transition to from any state in S on input a.
  2. Formally, $\delta'(S, a) = \bigcup_{q \in S} \delta(q, a)$.
  3. Example: If $S = q_1, q_2$ and $\delta(q_1, a) = q_3$ and $\delta(q_2, a) = q_4, q_5$, then $\delta'(q_1, q_2, a) = q_3, q_4, q_5$.

2.4 NFAs with ϵ-Transitions

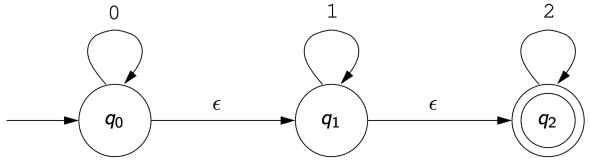
Automaton With ϵ-Transitions

A finite state automaton with ϵ-transition is defined in the same manner as a classic nondeterministic automata except that the symbol ϵ is in the alphabet

ϵ-Closure

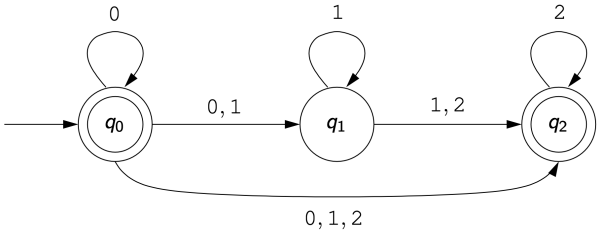
The set of all states of an automaton M reachable from a state q by a sequence of empty transitions

Removing ϵ-transitions Calculate δ^* as a table of rows correlating with each state and columns of each input and fields as sets of states. Maybe add q_0 if the empty string is recognized by the automaton.



The computation of the ϵ-closure provides the new transition function δ^*:

δ^*	0	1	2
q_0	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
q_1	$\varnothing$	$\{q_1, q_2\}$	$\{q_2\}$
q_2	$\varnothing$	$\varnothing$	$\{q_2\}$



2.5 Minimal DFA

Minimal DFA

A DFA M' is minimal iif for any other DFA M that recognizes the same language $|Q'| \leq |Q|$

Algorithm to minimize a DFA Idea: determine all the classes of states that are equivalent, and use these equivalence classes as the states of the minimal automaton. Two states are equivalent if and only if from both states any word leads or does not lead to a final state.

1. Remove all stats of M that are useless. A state q is useless iif there is no word w such that $\delta^*(q_0, w) = q$ (unreachable state) or if there is no word w such that $\delta^*(q, w) \in F$ (dead-end)
2. Find all the equivalence classes of states according to the following relation R: the relation $R(q, q')$ is true iif for any word $w \in \Sigma'^*$
   - either $\delta^*(q, w) \in F$ and $\delta^*(q', w) \in F$
   - or $\delta^*(q, w) \notin F$ and $\delta^*(q', w) \notin F$

   It is sufficient to consider the words w with $|w| \leq |Q|$
3. The minimal automaton is $M' = \langle Q', \Sigma, \delta', q_0', F'\rangle$ where
   - $Q' = \{[q]\}$
   - $q_0' = [q_0]$
   - $\delta'([q], a) = [\delta(q, a)]$
   - $F' = \{[q] \mid q \in F\}$

2.6 Regular Languages

Regular Language

A language recognized by a FSA recognizer.

- The **union of two regular languages** L_1 and L_2 $L_1 \cup L_2$ is a regular language. Construct a ϵ-NFA from FSA_{L_1} and FSA_{L_1} by introducing a new q_0 with ϵ-transitions to q_0 of the FSAs and ϵ-transitions from all accepting states from the FSAs to a new accepting global state.
- The **concatenation of two regular languages** L_1 and L_2 is still regular. ϵ-transitions from a new q_0 to the initial state of L_1, then from all accepting of L_1 to the initial of L_2, then from all accepting of L_2 to a new global accepting.
- The **complement of a regular language** $L(\overline{L} = \Sigma^* \backslash L)$ is regular. Make every accepting state non-accepting and every non-accepting state accepting.
- The **iterative closure** L^* is regular.

2.7 Grammar

Automatons recognize a language by consuming words, grammars generate words. Grammars are also called rewriting systems and rules rewriting rules or productions.

- A **rule** is composed of a left-hand side α and a right-hand side β surrounding an arrow: $\alpha \to \beta$

- Symbols are either **terminals** (symbols that compose the words generated by the grammar, denoted lowercase) or **nonterminals** (symbols mainly used in the rewriting process, denoted uppercase)
- A **term** is a sequence of term. and nonterm. symbols
- **rewriting** a term consists of applying a rule, thus replacing the l.h.s of the term with the r.h.s of the rule.
- Starting the rewriting process from a special rule called **axiom** and applying repetitively the rules, we obtain (not always) a term composed of terminal only. This is a word **generated** by the grammar.
- The set of words generated by the rewriting process is called the language **generated** by the grammar.
- The rewriting process is potentially nondeterministic, because any rules can be selected.

Grammar

A grammar is a 4-tuple $G = \langle V_T, V_N, P, S \rangle$ where

- V_T is a set of terminal symbols,
- V_N is a set of nonterminal symbols such that $V_T \cap V_N = \varnothing$,
- P is a set of rules $\alpha \to \beta$ where α and β are sequences of terminals or nonterminals, but α must contain at least one nonterminal,
- $S \in V_N$ is the axiom or the initial symbol.

Applying a single rule is called a **one-step derivation**, denoted by $\alpha \Rightarrow_G \beta$. A **derivation** is a sequence of one-step derivations, denoted by $\alpha \Rightarrow_G^* \beta$.

The language $L(G)$ **generated** by a grammar is the set of all the words which can be obtained or derived by rewriting from the axiom and do not contain any nonterminals.

2.7.1 Regular Grammar

Regular Grammars

A grammar is right/left regular if for all rewriting rules $\alpha \to \beta$:

1. $|\alpha| = 1$
2. $\beta = a$, $\beta = \epsilon$, or
   - $\beta = aB$ for right regular languages
   - $\beta = Ba$ for left regular languages

A language L is recognized by a FSA iif L is generated by a regular grammar.

2.8 Regular Expressions

Regular Expression (RE)

Over an alphabet Σ:

1. $\varnothing$ is a RE that denotes the empty set,
2. ϵ is a RE that denotes the set $\{\epsilon\}$
3. for any $a \in \Sigma, a$ is a RE that denotes the set $\{a\}$
4. if r and s are two REs which denotes the sets R and S, respectively, then $(r + s), (rs)$ and (r^*) are REs that denote the sets $R \cup S, R \cdot S$, and R^*, respectively.

Pumping Lemma

The pumping lemma, states that if a language is regular, then it is possible to identify, for every word w of the language, at least one way to split w as $w = xyz$. If the word y is pumped (w^i for any $i \geq 0$), then xy^iz belongs to the language as well.

Let L be an infinite regular language. Then, there exists a constant n such that, for any word w of L, where $|w| \geq n$, we can write $w = xyz$ with $|xy| \leq n$, $|y| \geq 1$ and for all $i \geq 0$, xy^iz belongs to L. $n \leq |Q|$ where $|Q|$ is the number of states of the minimal automaton that accepts L.

tl;dr: w is a word accepted by a minimal FSA with n states, but $|w| > n$. Thus, a certain path through the FSA is repeated. Any word where this path is repeated an arbitrary amount of times is also accepted by the FSA and thus belongs to the language.

Used to proof a language is not regular. To do this, split a word at each possible place, show how a changed i is not in the language anymore.

3 Pushdown Automata and Context Free Languages

3.1 Pushdown Automaton (PDA)

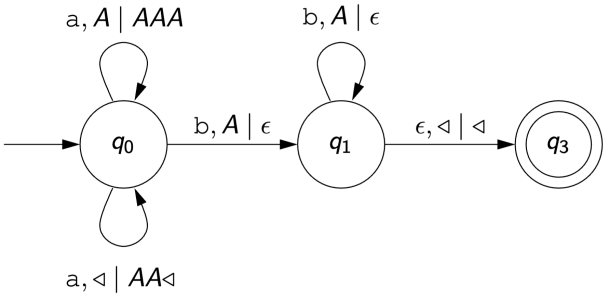
Pushdown Automaton

A 6-tuple $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \triangleleft \rangle$ where

- Q is a finite set of states
- Σ is an input alphabet
- Γ is an auxiliary alphabet (symbols of the stack) such that $\Gamma \cap \Sigma = \varnothing$
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to Q \times \Gamma^*$ is a transition function
- $q_0 \in Q$ is the initial state
- $\triangleleft \in \Gamma$ is a special symbol which indicates that the stack is empty

PDA recognizers further add a set of final states F.

Convention (in graphs): $(a, Z|\alpha)$ is a transition where a is the input symbol, Z is the symbol at the top of the stack before applying the transition function, and α is the content of the stack after.



Nondeterministic PDAs

A PDA with $\delta : Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$.

The classes of languages recognized by DPDAs and NPDAs are *not the same*. There exists a context-free language which is not recognized by any DPDA.

Example: $L = \{a^n b^n | n \geq 1\} \cup \{a^n b^2 n | n \geq 1\}$, where there are two transitions to the second part of the automaton, one for the right, one for the left version.

3.2 Context-Free Grammars

Context-Free Grammar

A context-free grammar $G = \langle V_T, V_N, P, S \rangle$ is a grammar whose productions $\alpha \to \beta$:

1. $|\alpha| = 1$ and $\alpha \in V_N$
2. β is a sequence of terminals and/or nonterminals

The language generated by a context-free grammar G is called a context-free language, we denote this language $L(G)$.

Backus-Naur Form (BNF)

A notation of context-free grammars:

- Nonterminals are strings
- Terminals are (single or double) quoted strings or symbols
- = corresponds to the arrow of a production
- | corresponds to alternative
- , concatenates elements
- ; terminates a production

Extended BNF (EBNF)

Superset of BNF:

- Parentheses can be use to group elements
- $[\gamma]$ means γ is optional (appear 0 or 1 times)
- $\{\gamma\}$ means γ can be repeated 0 or more times

Ambiguous Grammar And Language

A grammar G is ambiguous iif there exists a word $w \in L(G)$ such that w has more than one parse tree ($\stackrel{?}{=}$ derivation).
A language L is ambiguous iif all grammars that generate L are ambiguous.

A language L is recognized by a nondeterministic PDA iif L is a context-free language.

Pumping Lemma for Context-Free Languages If L is a context-free language, then there exists a constant n such that for any word w of L, with $|w| \leq n$, we can split the word $w = uvxyz$ with

1. $|vy| \geq 1$
2. $|vxy| \leq n$
3. $\forall i \geq 0, uv^i xy^i z \in L$

This is used to proof that a language is not context free.

4 Lexical Analysis and Parsing

Parsing

Processing an input text consists generally in, verifying that the structure of the text is correct, and in performing some operations using the text under analysis.
In general, a grammar describes the syntactical structure of the input text. Thus, verifying that the structure of the text is correct consists in making sure that the text is in conformity with the grammar. This is parsing.

4.1 Lexical Analysis

Lexical Analysis

The lexical analysis activity consists in reading a text character by character and in grouping the characters that form terminals of the grammar. Such terminals are usually called **tokens**.

A **lexer** or **scanner** is a program that performs the lexical analysis of an input text.

Terminals can generally be described by regular expressions, therefore lexers are usually implemented based on RE and/or FSAs and then transformed to minimized DFAs.

4.2 Syntax Analysis

Syntax Analysis

Syntax analysis is veryifing that the input text is well structured, i.e. the input is in cinformity with a given grammar.

For the verification of the conformity with respect of a given grammar, we build the parse tree of the input text. An input text is syntactically correct iif the parse tree can be built.

A program that performs this activity is called a parser. The main task of a parser is to create the parse tree of its

input using a grammar that describes the structure of its input. Note that a parser uses a lexer for the lexical analysis.

For context-free languages, this can be done easily, and since programming languages generally are context-free and unambiguous, this can be done, but certain aspects of programming languages cannot be expressed by context-free grammars, such as declare-before-use rules or type checking.

Parse Tree

A parse tree illustrates graphically how a given word is generated by a grammar, i.e. which productions are used to generate a given word.

Bottom-up analysis: "Use grammar backwards", start at the bottom of the parse tree and thus with terminals, combine them into nonterminals until you find the initial symbol.

Top-down analysis: Apply derivations successively from the first rewriting rule. This is easier but has some limitations.

4.2.1 Recursive Descent

Recursive Descent

Recursive Descent is a top-down approach in which we execute a program that applies a set of recursive functions (or methods) to process the input.

LL(1) Grammars

The class of LL(1) grammars is a sub-set of context-free grammars. One way of parsing such grammars is to use recursive descent parsers in which no backtracking is required. **L**eft to right **L**eft-most derivation with **1** token lookahead.

Left-most derivation: left-most rule is expanded first. (=Depth-First Search)

Left Recursion

A rule of a context-free grammar has **immediate left recursion** if it has the form $X \to X\alpha|\beta$ (or multiple $X\alpha_i$s or β_is). **General left recursion** can occur through several rules, e.g. (this is both)

$$S \to Aa|b$$
$$A \to Ac|Sd|\epsilon$$

Left Recursion Removal for Immediate LR

$$X \to X\alpha_1|X\alpha_2|\dots|X\alpha_n|\beta_1|\beta_2|\dots|\beta_m$$

becomes

$$X \to \beta_1 X'|\beta_2 X'|\dots|\beta_m X'$$
$$X' \to \alpha_1 X'|\alpha_2 X'|\dots|\alpha_n X'|\epsilon$$

Left-Factoring

$$X \to \alpha\beta|\alpha\gamma$$

becomes

$$X \to \alpha X'$$
$$X' \to \beta|\gamma$$

LL(K) Grammars

A grammar is LL(k) iif there exists a recursive descent parser that uses k tokens lookahead.

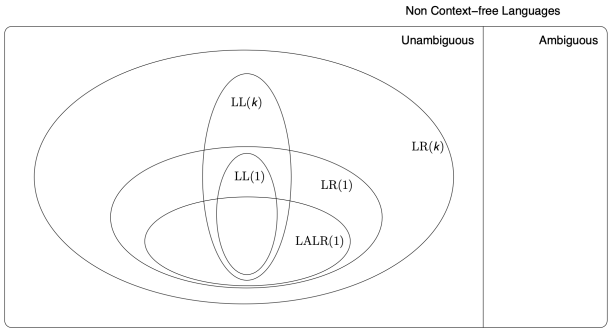
The class of LL(k) grammars strictly includes the class of LL(k-1) grammars. Most LL(k) grammars can be translated to LL(1), but not all.

LR(K) Grammars

LR(k) grammars can be analyzed with parsers based on the bottom-up technique with k token lookahead.

Left to right **R**ight-most derivation with k tokens lookahead. Any LR(k) grammar can be translated into a LR(1) grammar. If actions can be arbitrarily inserted into production rules, then LR(k) class strictly includes LR(k-1).

Comparing LL and LR parsers LR parsers are much more complex, more difficult to use, less efficient, require more memory, only used through parser generators. Due to the size of tables of LR parsers, parser generators work only for a sub-class of LR grammars called LALR(1) (Lookahead LR). For LR parsers left-recursion is allowed, more complex context-free languages can be analyzed, errors are detected as soon as they are encountered, almost all programming languages can be parsed by bottom-up parsers.



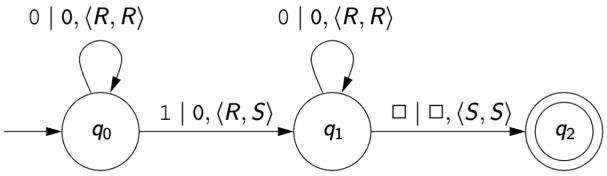
5 Turing Machines (TM)

Deterministic k-Tapes Turing Machine

A deterministic k-tapes TM with $k \geq 0$ (auxiliary tapes) is a 8-tuple $M = \langle Q, \Sigma, \Gamma, O, \delta, q_0, \square, F\rangle$ where

- Q is a finite set of states,
- Σ is an input alphabet,
- Γ is an auxiliary alphabet (for auxiliary tapes),
- O is an output alphabet,
- $F \subseteq Q$ is a set of final states,
- q_0 an initial state,
- $\square$ is the blank symbol,
- $\delta : Q \times \Sigma \times \Gamma^k \to Q \times \Gamma^k \times O \times \{R, S\} \times \{R, L, S\}^k \times \{R, S\}$ is a transition function with **R**ight, **L**eft, or **S**tationary

A language recognized by a Turing Machine is called a **recursively enumerable language**.



Transitions are marked as follows:

$$i, \quad r_1, \ldots, r_k \quad | \quad r'_1, \ldots, r'_k, \quad o, \quad \langle M_0, \ldots, M_{k+1} \rangle$$

with the input tape symbol i, aux tape symbols $r_1, \ldots, r_k$, new aux tape symbols $r'_1, \ldots, r'_k$, output tape symbol o and movements M_0 (input tape), $M_1, \ldots M_k$ (aux tapes) and M_{k+1} (output tape). Input and output tapes cannot move to the left.

Single Tape TM

A single tape TM M is a 7-tuple $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \square, F \rangle$ where

- Q is a finite set of states,
- Γ is the finite set of symbols of the tape,
- Σ is the input alphabet with $\Sigma = \Gamma \backslash \square$,
- $\delta : Q \times \Sigma \to Q \times \Gamma \times \{L, R\}$ is a transition function with **R**ight or **L**eft
- q_0 an initial state,
- $\square$ is the initial blank symbol,
- $F \subseteq Q$ is a set of final states,

Nondeterministic k-Tapes TM

A nondeterministic k-tapes Turing machine is a 8-tuple $M = \langle Q, \Sigma, \Gamma, O, \delta, q_0, \square, F \rangle$ where only the transition function

$$\delta : Q \times \Sigma \times \Gamma^k \to \mathcal{P}(Q) \times \Gamma^k \times \{R, L, S\}^{k+1}$$

is different from the deterministic model. The power set is used to model nondeterminism, as for nondeterministic FSA.

Single tape nondeterministic TMs recognize a language iif a deterministic TM recognizes it.

Grammar Without Restrictions

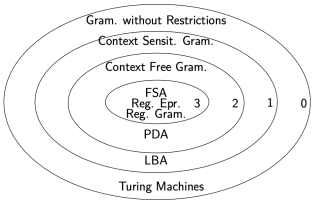
A grammar without restrictions is a grammar $G = \langle V_T, V_N, P, S \rangle$ where the rules looke like $\alpha \to \beta$ with $\alpha \neq \epsilon$ and α, β are sequences of terminals and nonterminals.

A language L is generated by a grammar without restrictions iff a TM recognizes L.

Context Sensitive Languages And Grammars

A context sensitive language is generated by a context sensitive grammar, i.e. a grammar that has rewriting rules $\alpha \to \beta$ with $|\alpha| \leq |\beta|$.

Context-sensitive grammars correspond to linear bounded automata (LBA).



3- Regular Languages,
2- Context-Free Languages,
1- Context-Sensitive Languages,
0- Recursively Enumerable Languages.

6 Decidability and Reducibility

Algorithm

A Finite collection of basic instructions or operations that perform some tasks. An algorithm always stops. Church-Turing thesis: TM algorithms equal intuitive notion of algorithms.

A language is called **Turing-recognizable**, if a TM accepts all words that belong to it, as opposed to rejecting it or looping infinitely. It is called **Turing-decidable**, if the machine stops for all inputs.

6.1 Decidability

6.1.1 Decidable problems

- Acceptance problem for DFAs A_{DFA} is decidable (Whether a DFA B accepts a given string w, or testing whether $\langle B, w \rangle$ is a member of A_{DFA}), by building a TM that simulates the DFA.
- Acceptance problem for NFAs A_{NFA} is decidable by building a TM that turns the NFA into a DFA and simulates it.
- Testing if a DFA accepts only the empty language (E_{DFA}) is decidable given an algorithm that tests if there is a path between any two nodes of a graph.
- EQ_{DFA} is decidable by designing a DFA such that $L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$ with $L(C) = \varnothing$ iif $L(A) = L(B)$.
- A_{CFG} is decidable by designing a TM that converts the context-free grammar into Chomsky normal form, listing all derivations with $2n - 1$ steps where $n = |w|$ (except $n = 0$, then list all derivations with 1 step), and accepting if any of these derivations generate w.
- Every context-free language is decidable. Turning a PDA/CFG into a TM does not generally work because of possible infinite loops (non-deterministic PDAs $\not\equiv$ deterministic PDAs). Just use the above though.

Universal Turing Machine

A TM that can simulate any other TM from a description.

6.1.2 Undecidable problems

- $A_{\text{TM}} = \{\langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w\}$ is undecidable. It is Turing-recognizable by constructing a universal TM that simulates M, but will loop if M loops. Halting problem: suppose that $H(\langle M, w \rangle)$ is a decider for A_{TM}, now build $D(\langle M \rangle) = !H(\langle M, \langle M \rangle \rangle)$. Now, run $D(\langle D \rangle)$. It accepts if D does not accept $\langle D \rangle$ and rejects if D accepts $\langle D \rangle$. Neither D nor H can exist.

- $HALT_{\text{TM}}$ is undecidable, because if there was such a TM, it could be used to calculate A_{TM}, by checking if it halts, returning false otherwise and if true, simulating the TM.

- E_{TM} is undecidable, because one can construct a wrapper for the TM under test M, that rejects any input x if it isn't equal to w and otherwise runs M. Assuming there is a TM that solves E_{TM}, run it on the wrapper and invert the result. This is equal to A_{TM}.

- EQ_{TM} is undecidable because if one of the TMs passed is a TM that only accepts the empty language, this would solve E_{TM}.

Diagonalization Imagine a table with rows of all possible TMs, columns of all possible inputs, and field with the evaluation of the TM on the input. D is a valid TM and a valid input and thus a row and a column. The row with D is the opposite as the values on the diagonal. There's an obvious contradiction when D comes to $\langle D \rangle$, because it needs to be the opposite of itself.

6.1.3 Problems that are not Turing-recognizable

Basics:

- If a language A is decidable by a decider M, its complement is also decidable (invert the answer of M).

- A language is decidable iif it is Turing-recognizable and its complement is also Turing-recognizable.

  - The complement of a decidable language is decidable and every decidable language is Turing-recognizable.
  - Let M_1 and M_2 be two recognizers for languages A and $\overline{A}$. Run both on an input, at least one needs to halt (since recognizers halt on inputs that belong to the language), use that information to decide.

Examples:

- $\overline{A_{\text{TM}}}$ is not Turing-recognizable, since A_{TM} is Turing-recognizable, but not decidable. So if $\overline{A_{\text{TM}}}$ were Turing-recognizable, it would also need to be decidable (see Basics).

- $\overline{EQ_{\text{TM}}}$ is not Turing-recognizable, because there is a reduction from A_{TM} to EQ_{TM}.

6.2 Reducibility

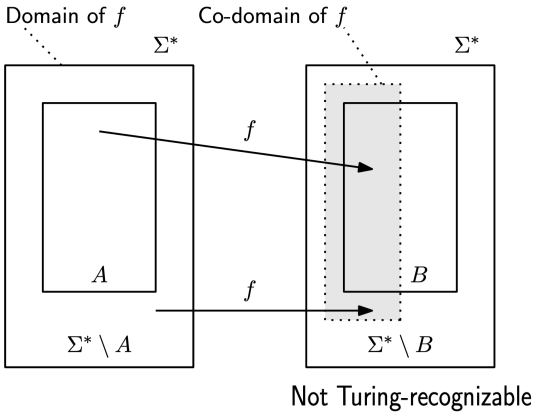
Computable Function

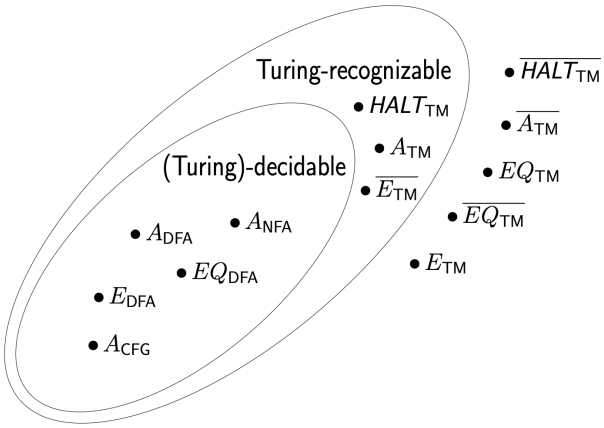
A function $f : \Sigma^* \to \Sigma^*$, if some TM M halts on every input w and writes $f(w)$ on its tape.

Mapping Reducible

A language A is mapping reducible to a language B, written $A \leq_m B$, if there is a computable function f such that for every w, $w \in A \iff f(w) \in B$.

f is called the **reduction** of A to B.





6.2.1 Theorems

Meta:

- If $A \leq_m B$ and B is solvable, then A is solvable.

- If $A \leq_m B$ and A is unsolvable, then B is unsolvable.

- $A \leq_m B$ says that a computer can transform any instance of A into an instance of B such that the answer to B is the answer to A.

Actual:

- If $A \leq_m B$ and B is decidable, then A is decidable, because we can apply f to w on a TM and use that result to decide w on A.

- If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable, same argument.

- If $A \leq_m B$ and A is undecidable, then B is undecidable.

  - Assume B is decidable, construct a TM that uses B to decide A, therefore contradiction.
  - Problem A reduces to problem B iff a solver for B can be used to solve problem A.
  - tl;dr: Construct a machine using a solver to B that solves the halting problem, $\text{HALT}_{\text{TM}} \leq_m B$ notated, where B is your problem. Alternatively, reduce to $\overline{B}$, since if $\overline{B}$ is decidable, B is also decidable.

- If $A \leq_m B$ and A is not Turing-recognizable, then B is not Turing-recognizable.

- $A \leq_m B \iff \overline{A} \leq_m \overline{B}$

- To prove that B is not Turing-recognizable, we may show that $A_{\text{TM}} \leq_m \overline{B}$. If this holds, and B were also Turing-recognizable, then A would be Turing-recognizable (rule 2 here), thus A and $\overline{A}$ would be Turing-recognizable, thus A would be decidable, which it obviously isn't.

7 Complexity, NP, NP-Completeness

Running Time Or Time Complexity

The running time or time complexity of a deterministic TM M that always halts is the function $F : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the number of steps used by M on any input of length n.

If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time TM.

Big-O Notation

Let f and g be the functions $f, g : \mathbb{N} - > \mathbb{R}^+$. We say that $f(n) = O(g(n))$ if $c, n_0 \in \mathbb{N}$ exist such that for every $n \geq n_0$ $f(n) \leq cg(n)$.

$f(n) = O(g(n))$ means that $g(n)$ is an upper bound for $f(n)$, or more precisely, $g(n)$ is an asymptotic upper bound for $f(n)$.

Because changing the base of a log only adds a factor, we can ignore it.

Small-O Notation

Let f and g be the functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We say that $f(n) = o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

In other words, $f(n) = o(g(n))$ means that for any real number $c > 0$, there exists n_0 such that $f(n) < cg(n)$ for all $n \geq n_0$. Informally, $g(n)$ grows much faster than $f(n)$.

Examples:

- $\sqrt{n} = o(n)$
- $n = o(n \log \log n)$
- $n \log \log n = o(n \log)$
- $n \log n = o(n^2)$
- $n^2 = o(n^3)$
- $n^2 \neq o(3n^2)$

Time Complexity Class

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the time complexity class, TIME($t(n)$) to be the collection of all languages that are decidable by a deterministic Turing machine in time $O(t(n))$.

7.1 Time Complexities of TMs

- Let $t(n)$ be a function, where $t(n) \geq n$. Then every $t(n)$ time multi tape TM has an equivalent $O(t^2(n))$ time single tape TM.
- Let N be a nondeterministic TM decider. The running time of N is the function $f : \mathbb{N} \to \mathbb{N}$, where $f(n)$ is the maximum number of steps that N uses on any branch of its computation any input of length n.
- Let $t(n)$ be a function where $t(n) \geq n$. Then every $t(n)$ time nondeterministic single tape TM has an equivalent $2^{O(t(n))}$ time deterministic single tape TM.

7.2 The Classes P and NP

There is at most a quadratic or **polynomial** difference between deterministic single and multi tape TMs, but an **exponential** differences between deterministic and nondeterministic TMs.

The Class P

P is the class of languages that are decidable in polynomial time with a deterministic single tape TM, or $\text{P} = \bigcup_k \text{TIME}(n^k)$.

Examples of languages in P:

- PATH: Does a directed graph have a directed path between two given nodes? (Proof: Starting with the start node, iteratively taint new reachable nodes until there are none (at most m steps). Accept if the target is tainted.)
- RELPRIME: Are two natural numbers relatively prime?
- Every context-free language decision. (Thus parsing of programming languages!)
- COMPOSITE: $\{x | x = pq,$ for integers $p, q > 1\}$
- 2SAT! (2SAT $\leq_P$ 3SAT, but 3SAT $\not\leq_P$ 2SAT)

Verifier

A verifier for a language A is an algorithm V, where $A = \{w | V$ accepts $\langle w, c\rangle$ for some string $c\}$.

The time of a verifier is only measured in terms of the length of w. A language is **polynomially verifiable** if it has a polynomial time verifier. c is called the certificate of membership in A, e.g. the certificate for a string $\langle G, s, t\rangle \in$ HAMPATH is the path from s to t, for COMPOSITE, it is one of the divisors of x.

The Class NP

NP is the class of languages that have polynomial time verifiers. A language is in NP iif it is decided by a nondeterministic polynomial time TM.

A language may belong to NP only (HAMPATH), or to both NP and P (COMPOSITE). P is a subset of NP.

NTIME

NTIME($t(n)$) is the nondeterministic time complexity class that contains all languages decided by a $O(t(n))$ time nondeterministic TM.

$\text{NP} = \bigcup_k \text{NTIME}(n^k)$

Examples of languages in NP:

- HAMPATH: What is the directed path through a directed graph that visits each node exactly once?

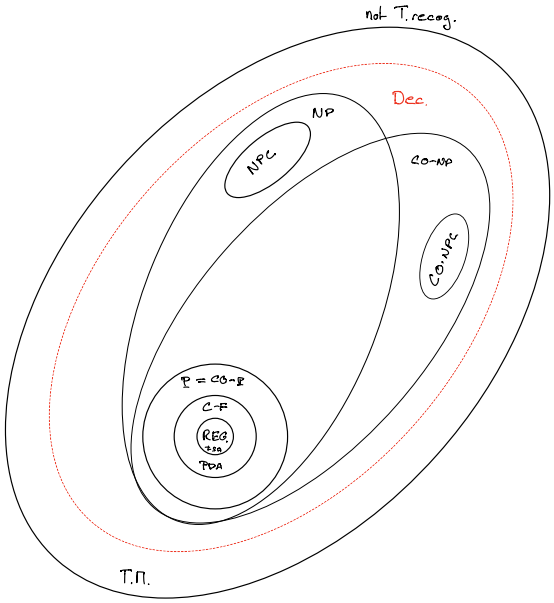
- CLIQUE: A $k-$clique is a subgraph of a undirected graph wherein every two nodes are connected by an edge with k nodes. Does a certain undirected graph contain a clique of size k? (Proof: Either build a verifier that tests whether enough nodes are in the subgraph and if all edges are there, or build a nondeterministic TM that (nondeterministically) selects a subset of nodes and checks if all edges are present).

- SUBSET-SUM: Does a subcollection (repetition allowed!) of a collection of integers $x_1, \ldots, x_k$ sum up to a target number t? (Proof: Verifier that tests whether all given elements belong to the collection, and if the sum matches)

The Class co-NP

Languages for which the complement is in NP.

$\overline{\text{HAMPATH}}$, $\overline{\text{CLIQUE}}$, and $\overline{\text{SUBSET-SUM}}$ are not obviously in NP. Probably NP≠co-NP, but certainly P=co-P.



7.3 NP-Completeness

- SAT: Can a boolean formula (combination of boolean variables and logic variables $(\wedge, \vee, \neg)$) be satisfied?

Polynomial Computable Function

A function $f : \Sigma^* \to \Sigma^*$, if some polynomial time TM exists that halts with just $f(w)$ on its tape when input w.

Polynomial Time Mapping Reducible

Language A is polynomical time mapping reducible, or simply polynomial time reducible, to language B, written $A \leq_P B$, if a polynomical time computable function $f : \Sigma^* \to \Sigma^*$ exists, where for every w,

$$w \in A \iff f(w) \in B.$$

f is the polynomial time reduction of A to B.

- If $A \leq_P B$ and $B \in$ P, then $A \in$ P

3SAT to CLIQUE A literal is a boolean variable x or its negation $\overline{x}$. A clause is several literals connected with $\vee$s (e.g. $x_1 \vee \overline{x_2} \vee \overline{x_3}$). A conjunctive normal boolean formula (**cnf-formula**) is composed of several clauses connected with $\wedge$s. A **3ncf-formula** is a cnf-formula that has always combines three literals $((x_1 \vee \overline{x_2} \vee x_3) \wedge (x_4 \vee x_5 \vee \overline{x_6}))$

- 3SAT $= \{\langle\phi\rangle | \phi$ is a satisfiable 3cnf-formula.$\}$

Reduce 3SAT to CLIQUE: Create a graph with nodes for each literal, no edge between literals of the same clause and between x_2 and $\overline{x_2}$, all others are connected. Turns out there is a k-clique where k corresponds to the number of clauses in ϕ. This is because there is at least one true literal in each clause of a satisfied cnf-formula. If the formula is not satisfiable, then there is a contradiction, but no contradictory literals are connected.

SAT to SAT-CNF See references.

SAT-CNF to 3SAT

1. Clause with 2 literals: $x \vee y = (x \vee y \vee u) \wedge (x \vee y \vee \overline{u})$
2. 1: $(x) = (x \vee u \vee v) \wedge (x \vee u \vee \overline{v}) \wedge (x \vee \overline{u} \vee v) \wedge (x \vee \overline{u} \vee \overline{v})$
3. k: $(x_1 \vee \ldots \vee x_k) = (x_1 \vee x_2 \vee u_1) \wedge (\overline{u_1} \vee x_3 \vee u_2) \wedge \ldots \wedge (\overline{u_{k-4}} \vee x_{k-2} \vee u_{k-3}) \wedge (\overline{u_{k-3}} \vee x_{k-1} \vee x_k)$ (slides: $\overline{x_k}$)

NP-Completeness

A language B is NP-complete if

1. B is in NP
2. Every $A \in$ NP is pol. time reducible to B

- If B is NP-complete and $B \in$ P, then P=NP.
- If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.