COMP 330 - Theory of Computation

Émile Bessette (260930793)

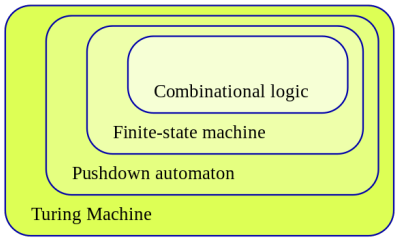
1 Regular Languages

A language is called a regular if some finite automaton recognizes it. That is, if you can construct some DFA/NFA recognizing the language then that language is regular.

1.1 Regular Operations

Let A and B be languages. We define the regular operations union, concatenation, and star as follows:

- Union: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$.
- Concatenation: $A \circ B = \{xy \mid x \in A \text{ and } y \in B\}$.
- Star: $A^* = \{x_1 x_2 \dots x_k \mid k \geq 0 \text{ and each } x_i \in A\}$.



We have the following properties about the above operations:

- If A_1 and A_2 regular languages, so is $A_1 \cup A_2$. Closure under union.
- If A_1 and A_2 regular languages, so is $A_1 \circ A_2$. Closure under concatenation.
- The class of regular languages is also closed under complementation and intersection.

1.2 Finite Automatons

1.2.1 Deterministic Finite Automaton (DFA)

At every state, a DFA has an outgoing arrow for every symbol of the alphabet. Thus, we can determine what some next state may be. Formally: A finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$.

1.2.2 Nondeterministic Finite Automaton (NFA)

A NFA offers more flexibility since at each state we can have zero or multiple outgoing arrows for each letter of the alphabet. Moreover, NFA allows empty transitions.

1.2.3 Relation

Deterministic and nondeterministic finite automaton recognize the same class of languages. That is, every NFA has a corresponding DFA. Also, both a DFA and NFA can be formally defined as:

- Q is a finite set called the states,
- Σ is a finite set called the alphabet,
- $\delta : Q \times \Sigma \to Q$ is the transition function,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ is the set of accept states.

1.3 Regular Expressions

A DFA/NFA and its corresponding regular language are equivalent: they represent the same regular language. One can go from DFA/NFA to regular language and vice versa.

2 Non-Regular Languages

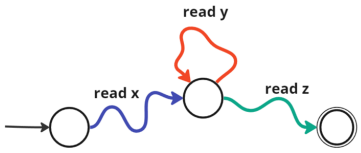
A non-regular language is one that cannot be recognized by a finite automaton (such as a deterministic finite automaton or DFA). Such languages typically exhibit characteristics that necessitate an infinite number of states in a DFA to properly recognize them.

Equivalently, a language is non-regular if it possesses infinitely many equivalence classes. In the context of DFA states, each equivalence class corresponds to a state. If two strings lead to the same state in some DFA, they are considered part of the same equivalence class. Thus, when a language requires infinitely many states for recognition, it implies the existence of infinitely many equivalence classes, indicating non-regularity.

2.1 Proving Non regularity

2.1.1 Pumping Lemma

If A is a regular language, then there exists a number p (the pumping length) such that if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:



1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

2.1.2 Myhill-Nerode Theorem

Two strings x and y share an equivalence relation $x \equiv_L y$ if both xz and yz are either in or out of the language for some arbitrary z. If there are finitely many equivalence classes, then the language is regular; however, if there are infinitely many equivalence classes, then the language is non-regular.

3 Context-Free Languages

3.1 Context-Free Grammars

A context-free grammar is a 4-tuple (V, Σ, R, S), where

- V is a finite set called the variables,
- Σ is a finite set, disjoint from V, called the terminals,
- R is a finite set of rules, with each rule being a variable or a terminal followed by a string of variables and/or terminals,
- $S \in V$ is the start variable.

The grammar is said to be ambiguous if at least two distinct derivations yield the same string. A grammar in Chomsky normal form ($A \to BC$, etc) generates a string x in exactly $2|\text{x}| - 1$ steps and any CFL can be generated by a CFG in CNF. **Note: The class of context-free languages is not closed under complementation or intersection.**

3.2 Pushdown Automata

Another way to prove a language is context-free is by constructing a pushdown automata (PDA) recognizing the language. It is similar to a finite automaton, but we can use a stack by pushing/popping items as follows: We write $a, b \to c$ to signify that when the machine is reading an a from the input, it may replace the symbol b on the top of the stack with a c. Any of a, b, and c may be ε. If a is ε, the machine may make this transition without reading any symbol from the input. If b is ε, the machine may make this transition without reading and popping any symbol from the stack. If c is ε, the machine does not write any symbol on the stack when going along this transition.
A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

- Q is the set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
- $q_0 \in Q$ is the start state, and
- $F \subseteq Q$ is the set of accept states.

3.3 Pumping Lemma for Context-Free Languages

If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions:

- For each $i \geq 0$, $uv^ixy^iz \in A$,
- $|vy| > 0$, and
- $|vxy| \leq p$.

4 Turing Machines

Initially the tape contains only the input string and is blank everywhere else. If the machine needs to store information, it may write this information on the tape. To read the information that it has written, the machine can move its head. The machine continues computing until it decides to produce an output. The outputs accept and reject are obtained by entering designated accepting and rejecting states. If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting.

Call a language **Turing-recognizable** if some Turing machine recognizes it. That is, it reaches an accept state for all strings of the language. If a Turing machine recognizes the complement of some language, then the language is **co-Turing-recognizable.**

Note: on input of a string that is not in the language, rejection is not guaranteed and the machine may loop.

Call a language **Turing-decidable** or simply decidable if some Turing machine decides it. That is, it halts (reject/accept) on all input.

4.1 Formal Definition

A Turing machine is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where:

- Q, Σ, Γ are all finite sets and
- Q is the set of states,
- Σ is the input alphabet not containing the blank symbol b,
- Γ is the tape alphabet, where $b \in \Gamma$ and $\Sigma \subseteq \Gamma$,
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is the transition function,
- $q_0 \in Q$ is the start state,
- $q_{\text{accept}} \in Q$ is the accept state, and
- $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$

4.2 State Diagram

On input we start with the string written on the tape with the head placed on the first bit of the string. We transition states from what is read from the tape. We define transitions $a \to b, R/L$ to signify that upon reading a on the tape, write b on the tape and move the head right(R) or left(L). We can also translate without writing on the tape.

4.3 Definition of Algorithm

We use Turing machines to implement high-level descriptions on how it proceeds to accomplish some task instead of describing low-level implementation. That is, we mix low-level and high-level description to achieve intermediate description of how a Turing machine works. That is, we describe what we do on input, how the head moves, how the tape is altered and how we reach accept/reject states.

5 Decidability

5.1 Decidable Problems

$A_{DFA} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}$

$A_{NFA} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}$

$A_{REX} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$

$E_{DFA} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$

$EQ_{DFA} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$

$A_{CFG} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$

$E_{CFG} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$

Note: Every context-free language is decidable.

Note: A language is decidable iff it is Turing-recognalizable and co-Turing-recognizable.

5.2 Undecidable Problems

$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$

$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

$E_{TM} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$

$REGULAR_{TM} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$

$EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$

6 Reducibility

If some problem A reduces to B, then we can use a solution to B to solve A. One could also say there exist a reduction from A to B. A generic proof would follow the following format:

- Show how problem A reduces to problem B
- Assume problem A is decidable and construct some Turing Machine deciding it, call it R.
- Construct some Turing machine deciding problem B, call it S.
- Use R in S to solve B.
- If R decides A then S decides B but since B is undecidable then so is A.

6.1 Reductions via Computation Histories

Let M be a Turing machine and w an input string. An accepting computation history for M on w is a sequence of configurations, $C_1, C_2, ..., C_l$, where C_1 is the start configuration of M on w, C_l is an accepting configuration of M, and each C_i legally follows from C_{i-1} according to the rules of M. A rejecting computation history for M on w is defined similarly, except that C_l is a rejecting configuration.

6.2 Post Correspondence Problem (PCP)

$PCP = \{\langle P\rangle \mid P \text{ is an instance of the Post Correspondence Problem with a match}\}$

6.3 Mapping Reducibility

A function $f : \Sigma^* \to \Sigma^*$ is a computable function if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Language A is mapping reducible to language B, written $A \leq_m B$, if there is a computable function $f : \Sigma^* \to \Sigma^*$, where for every w, $w \in A \Leftrightarrow f(w) \in B$. The function f is called the reduction from A to B.

6.4 Rice's theorem

We call a language $P \subseteq \{\langle M \rangle : M \text{ is a TM}\}$ a Turing machine property. Moreover, we say that P is

- non-trivial if there exist TMs T_0, T_1 such that $\langle T_0 \rangle \notin P$ and $\langle T_1 \rangle \in P$,
- semantic if for any TMs M_1 and M_2 such that $L(M_1) = L(M_2)$, we have $\langle M_1 \rangle \in P$ if and only if $\langle M_2 \rangle \in P$.

Rice's Theorem: If P is a nontrivial, semantic Turing machine property, then P is undecidable.

Prove some arbitrary machine is in the language and some other isn't. Then show that if two machines recognize the same language, then they are either both in the language or both outside.

6.5 Turing Reducibility

Language A is Turing reducible to language B, written $A \leq_T B$, if A is decidable relative to B. A B-oracle Turing machine is an algorithm which uses an oracle for problem B to decide A.

7 Appendix

7.1 Conversion of NFA to DFA

Any NFA has a corresponding DFA. We proceed as follows:

- The new states are any combination (empty, single, pairs, etc) of the initial states.
- The new accept states are any new states containing the old accept state.
- The new start state is the old one but if ϵ transition(s) exit that old start state then the new start state is the one containing the set of reachable states using ϵ transition(s) from the old start state and including the old start state.
- For transitions, we will iterate our new states and check transitions on alphabet input in NFA. If no transition is defined on some input, we go to empty state. Otherwise, we take each new state and determine which states we can reach (ϵ transitions included) on some input of the alphabet and set the transition to that new state. **Note: For transitions, we must exit a state on some input before following ϵ transitions**
- We remove states that are not reachable from another state.

7.2 NFA to Regular Expression

- Convert NFA to GNFA by creating a new start state with ϵ transition to old start state. We also have a new single accept state which has ϵ transitions from old accept states to new one.
- When you remove a state, model possible transitions to the remove state and back to current states as a transition to itself. When you have a choice of path, use the $\cup$.
- Remove states one by one until we only have the new start and accept states in the diagram. The result is the regular expression of the NFA.