

CS 273: Intro to Theory of Computation, Fall 2007

Review of topics covered

Madhusudan Parthasarathy

1 Introduction

The theory of computation is perhaps *the* fundamental theory of computer science. It sets out to define, mathematically, what exactly computation is, what is feasible to solve using a computer, and also what is *not* possible to solve using a computer.

The main objective is to define a computer mathematically, without the reliance on real-world computers, hardware or software, or the plethora of programming languages we have in use today. The notion of a Turing machine serves this purpose and defines what we believe is the crux of all computable functions.

The course is also about weaker forms of computation, concentrating on two classes, regular languages and context-free languages. These two models help understand what we can do with restricted means of computation, and offer a rich theory using which you can hone your mathematical skills in reasoning with simple machines and the languages they define. However, they are not simply there as a weak form of computation—the most attractive aspect of them is that problems formulated on them are *tractable*, i.e. we can build efficient algorithms to reason with objects such as finite automata, context-free grammars and pushdown automata. For example, we can model a piece of hardware (a circuit) as a finite-state system and solve whether the circuit satisfies a property (like whether it performs addition of 16-bit registers correctly). We can model the syntax of a programming language using a grammar, and build algorithms that check if a string parses according to this grammar.

On the other hand, most problems that ask properties about Turing machines are *undecidable*. Undecidability is an important topic in this course. You have seen and proved yourself that several tasks involving Turing machines are unsolvable— i.e. no computer, no software, can solve it. For example, you know now that there is *no software* that can check whether a C-program will halt on a particular input. This is quite amazing, if you think about it. To prove something is possible is, of course, challenging, and you will learn in other courses several ways of showing how something is possible. But to show something is *impossible* is rare in computer science, and you will probably see no other instance of it in any other undergraduate course. To show something is impossible requires an argument quite unlike any other, and you have seen the method of *diagonalization* to prove impossibilities and *reduction* that help you prove infer one impossibility from another. Impossibility results for regular languages and context-free languages are shown using the pumping lemma.

In conclusion, you have formally learnt how to define a computer, and analyze the properties of computable functions, which surely is the theoretical foundation of computer science.

2 The players

An alphabet (usually denoted by Σ) for us is always some finite set; words are sequences (strings) of letters in the alphabet. And a language is a set of words over the alphabet.

The main players in our drama have been the four classes of languages: regular language (REG), context-free languages (CFL), Turing-decidable languages (TM-DEC) and Turing-recognizable languages (TM-REC).

Regular languages are the languages accepted by deterministic finite automata (DFAs) and context-free languages are those languages generated by context-free grammars (CFGs). Turing-decidable languages are those languages L for which there are Turing machines that always halt on every input, and decide whether a word is in L or not.

Turing-recognizable languages are more subtle. A language L is Turing-recognizable if there is a TM M which (a) when run on a word in L , halts eventually and accepts, and (b) when run on a word not in L , M either halts and rejects, or does not halt. In other words, a TM recognizing L has to halt and accept all words in L , and for words not in L , can reject or go off into a loop.

The main things to remember are:

- $\text{REG} \subset \text{CFL} \subset \text{TM-DEC} \subset \text{TM-REC}$.
- Each of the above inclusions is *strict*: i.e. there is a language that is context-free but not regular, there is a language that is TM-DEC but not context-free, etc.
- There are languages that are not even TM-REC.

Regular languages are trivially contained within context-free languages (as DFAs can be easily converted to PDAs). However, it is not easy to see that a CFG/PDA for L can be converted to a TM deciding L . However, this is possible (see Theorem 4.9). TM-DEC languages are clearly TM-REC as well, by definition.

For example, if $\Sigma = \{a, b\}$, then

- $\{a^i b^j \mid i, j \in \mathbb{N}\}$ is regular (and hence also a CFL, and TM-DEC and TM-REC),
- $\{a^n b^n \mid n \in \mathbb{N}\}$ is a CFL but not regular (but is TM-DEC and TM-REC),
- $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is TM-decidable (and TM-REC) but not context-free (nor regular).
- A language that is Turing-recognizable but not Turing-decidable is $A_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM that accepts } w\}$.
- A language that is not even Turing-recognizable is $DOESNOT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM that does not accept } w\}$.

The notion of what we call an “algorithm” in computer science accords with Turing-decidability. In other words, when we build an algorithm for a decision problem in computer science, we want it to always halt and say ‘yes’ or ‘no’. Hence the notion of a computable function is that it be TM-decidable.

3 Regular languages

Let us review what we learnt about regular languages (Chapter 1 of Sipser).

Fix an alphabet Σ . A regular language $L \subseteq \Sigma^*$ is any language accepted by a deterministic finite automaton (DFA).

A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. The (deterministic) transition function is the function $\delta : Q \times \Sigma \rightarrow Q$.

A *non-deterministic finite automaton* (NFA) is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where Q , q_0 and F are as in a DFA, and the nondeterministic transition function is $\delta : Q \times \Sigma_\epsilon \rightarrow \mathcal{P}(Q)$.

A regular expression is formed using the syntax:

$$\epsilon \mid a \mid \emptyset \mid R_1 \cup R_2 \mid R_1 \cdot R_2 \mid R_1^*$$

Here are some properties of regular languages:

- Non-deterministic finite automata can be converted to equivalent DFAs. This construction is the “subset construction” and is important. See Theorem 1.39 in Sipser. Intuitively, for any NFA, we can build a DFA that tracks the *set of all states* the NFA can be in. Handling ϵ -transitions is a bit complex, and you should know this construction. Hence NFAs are equivalent to DFAs.
- Regular languages are closed under union, intersection, complement, concatenation and Kleene-* (Theorems 1.45, 1.47 and 1.49 in Sipser).
- Regular expressions define exactly the class of regular languages (Theorem 1.54). In other words, any language generated by a regular expression is accepted by some DFA (Lemma 1.55) and any language accepted by a DFA/NFA can be generated by a regular expression (Lemma 1.60).
- So the trinity: $\text{DFA} \equiv \text{NFA} \equiv \text{RegExp}$ holds.
- The pumping lemma says that, if L is regular, then there is a $p \in \mathbb{N}$ such that for every $s \in L$ with $|s| > p$, there are words x, y, z with $s = xyz$ such that (a) $|y| > 0$ (b) $|xy| \leq p$ and (c) for every i , $xy^iz \in L$. In mathematical language,

L is regular \Rightarrow

$$[\exists p \in \mathbb{N}. \forall s \in L [(s \in L \wedge |s| > p) \Rightarrow (\exists x, y, z \in \Sigma^* : s = xyz \wedge |y| > 0 \wedge |xy| \leq p \wedge (\forall i. xy^iz \in L))]]]$$

- The contrapositive to the pumping lemma says that, if for every $p \in \mathbb{N}$, there is an $s \in L$ with $|s| > p$, such that for every x, y, z with $s = xyz$, $|y| > 0$, and $|xy| \leq p$, there is some i such that $xy^iz \notin L$, then L is not regular.

In mathematical language,

$$[\forall p \in \mathbb{N}. \exists s \in L [s \in L \wedge |s| > p \wedge (\forall x, y, z \in \Sigma^* : (s = xyz \wedge |y| > 0 \wedge |xy| \leq p) \rightarrow (\exists i. xy^iz \notin L))]]]$$

$\Rightarrow L$ is not regular

- The contrapositive to the pumping lemma gives us a way to prove a language is not regular. We take an arbitrary p , and construct a particular word w_p , which depends on p , such that $w_p \in L$ and $|w_p| > p$. Then we show that *no matter* which x, y, z is chosen such that $s = xyz$, $|xy| \leq p$ and $|y| > 1$, there is an i such that $xy^iz \in L$.

Knowing how to prove a language non-regular using the pumping lemma is important.

- We can using the above technique, show several languages to be non-regular, for example (see Eg.1.73, 1.74, 1.75, 1.76, 1.77):
 - $\{0^n 1^n | n \geq 0\}$ is not regular.
 - $\{w | w \text{ has an equal number of 0s and 1s}\}$ is not regular.
 - $\{ww | w \in \Sigma^*\}$ is not regular.
 - $\{1^{n^2} | n \geq 0\}$ is not regular.

Choosing $w_p \in L$ should be done carefully and cleverly. However, the choice of i being 0 or 2 usually works for most example.

Note that you are allowed to pick w_p (but not p), and allowed to pick i (not x, y or z).

- Deterministic finite automata can be *uniquely minimized*. In other words, for any regular language, there is a unique minimal automaton accepting it (here, by minimal, we mean an automaton with the least number of states). Moreover, given a DFA A , we can build an efficient algorithm to build the minimal DFA for the language $L(A)$. This is not covered in Sipser; see the handout on suffix languages and minimization: (<http://www.cs.uiuc.edu/class/fa07/cs273/Handouts/minimization/suffix.pdf> and the minimization algorithm: <http://www.cs.uiuc.edu/class/fa07/cs273/Handouts/minimization/minimization.pdf>). For the final exam, you are not required to know this algorithm, but just know that regular languages have a unique minimal DFA.

Turning to algorithms for manipulating automata, here are some things worth knowing (read Sipser Section 4.1):

- We can build an algorithm that checks, given a DFA/NFA A , whether $L(A) \neq \emptyset$. In other words, the problem of checking emptiness of an automaton is decidable. (see Sipser Theorems 4.1 and 4.2). In fact, this algorithm runs in linear (i.e. $O(n)$) time.
- Automata are closed under operations union, intersection, complement, concatenation, Kleene*, etc. Moreover, we can build algorithms to do all these closures. That is, we can build algorithms that will take two automata and compute an automaton accepting the union of the languages accepted by the two automata, etc.

All constructions we did on automata are actually computable algorithmically. For example, we can build algorithms to convert regular expressions to automata, automata to regular expressions, etc.

Several other questions regarding automata are also decidable: For example:

- Given two automata A and B , we can decide whether $L(A) \subseteq L(B)$.
Note that $L(A) \subseteq L(B)$ iff $L(A) \cap \overline{L(B)} = \emptyset$. So we can build the complement C of B , intersect C with A to get D , and check D for emptiness.
- Given two automata A and B , we can decide if $L(A) = L(B)$, by checking if $L(A) \subseteq L(B)$ and if $L(B) \subseteq L(A)$, which we have show above to be decidable. (See Sipser Theorem 4.5.)

In general, most reasonable questions about automata are decidable.

4 Context-free Languages

A context-free grammar is a 4-tuple (V, Σ, R, S) , where V is a finite set of variable, Σ is a finite set of terminals, $S \in V$ is the start variable, and R is a set of rules of the form $A \rightarrow w$ where $A \in V$ and $w \in (V \cup \Sigma)^*$.

A context-free grammar generates a set of words over its terminal alphabet Σ , and is called the language generated by the grammar. A word w is generated by a CFG if we can derive, starting with the start symbol S , and using the rules a finite number of times, the word w . A derivation of a word can also be seen as a *parse tree*, where the root is labeled with S , and the leaves, read left to write, give w , and each node with its children encode some derivation rule in R .

A CFG is in Chomsky normal form if every rule is of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B, C are non-terminals, and a is a terminal. (There's a slightly more complex definition to allow generating ϵ .)

A string is derived ambiguously in a CFG G if it has two or more different left-most derivations (or two or more parse tree derivations) in G . A grammar G is ambiguous if it can derive some word w ambiguously.

A language over Σ is a *context-free language* if it is generated by some context-free grammar with terminal alphabet Σ .

A *pushdown automaton* (PDA) is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q , q_0 and F are as in a DFA, Γ is a finite stack alphabet, and the nondeterministic transition function is $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow \mathcal{P}(Q \times \Gamma_\epsilon)$.

A pushdown automaton is, by convention, nondeterministic. It uses a FIFO stack to store data, and accepts when it has some run on a word on which it reaches a final state.

Here are some important facts about context-free languages:

- Context-free grammars can be converted to Chomsky normal form (Theorem 2.9).
- Pushdown automata define exactly the class of context-free languages. I.e. $\text{PDA} \equiv \text{CFG}$. (Sipser Theorems 2.20).
- Context-free languages are closed under union, concatenation and Kleene-*, but not under intersection or complement. (See the last section in this article for more details.)

- Deterministic pushdown automata are strictly weaker than pushdown automata (since they can be complemented by toggling the final states).
- The membership problem for CFGs and PDAs is decidable. In particular, the CYK algorithm uses dynamic programming to solve the membership problem for CFGs, and in fact produces a parse tree as well, in $O(n^3)$ time.
- The problem of checking if a context-free language generates all words (i.e. if $L(G) = \Sigma^*$) is undecidable. This is proved in Theorem 5.13, using context-free grammars that check computation histories of Turing machines.
- The problem of checking if a context-free language is ambiguous is undecidable (Exercise 5.21 in Sipser), and is proved by a reduction from the Post's correspondence problem. You need to know this fact, not the proof.
- The language $\{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not a context-free language. There is a pumping lemma for context-free languages, and we can use it to show that this language is not context-free. You are not required to know this proof.

5 Turing machines and computability

Turing machines

A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ where Q is a finite set of states, Σ is a finite alphabet, Γ is the tape-alphabet that includes Σ and a particular symbol $\#$ which is the blank symbol, $q_0 \in Q$ is the initial state, $q_{accept} \in Q$ is the accept state and q_{reject} is the reject state ($q_{accept} \neq q_{reject}$). The transition function is deterministic: $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$.

A Turing machine has access to a one-way infinite tape, and hence has unbounded memory. It can go back and forth on the tape rewriting symbol to do its computation.

A Turing machine halts if it reaches an accepting or rejecting configuration (i.e. reaches a configuration with state q_{accept} or q_{reject}). A Turing machine stops when it reaches such a configuration.

There are two main classes of languages defined using Turing machines:

- A language L is Turing-decidable if there is a Turing machine M such that M halts on all inputs, and accepts all words in L and rejects all words not in L .
- A language L is Turing-recognizable if there is a Turing machine M such that (a) on any word in L , M halts and accepts, and (b) on any word not in L , either M does not halt, or halts and rejects.

The notion of Turing-recognizability and Turing-decidability are robust concepts. Changing the definition of Turing machines in any reasonable way does not change this notion. For example, a multi-tape Turing machine is not more powerful, and can be converted to a single-tape Turing machine. Also, giving two-way-infinite tapes do not make Turing machines more powerful.

A nondeterministic Turing machine (NTM) is a Turing machine which has a non-deterministic transition function. The languages decided and recognized by non-deterministic Turing machines are precisely those decided and recognized by deterministic Turing machines. This proof is done using *dovetailing*: a deterministic Turing machine simulates an NTM by systematically exploring all runs of the NTM for time i steps, for increasing values of i .

By definition, a Turing-decidable language is also Turing-recognizable.

An important language is:

- $A_{TM} = \{\langle M, w \rangle \mid M \text{ is a Turing machine that halts and accepts } w\}$.

Here are some of the important results:

- A_{TM} is *not* Turing-decidable (Theorem 4.11). This is the fundamental undecidable problem and is shown undecidable using a *diagonalization* method, which takes the code for a purported TM deciding A_{TM} and pits it against itself to lead to a contradiction. Diagonalization is an important technique to prove impossibility results (almost the only technique we know!).
- A_{TM} is Turing-recognizable. This is easy to show: we can build a Turing machine that on input $\langle M, w \rangle$, simulates M on w and accepts if M accepts w . Hence the class TM-DEC is a strict subclass of TM-REC.
- A language L is Turing-decidable iff L and \bar{L} are Turing-recognizable (Theorem 4.22). If L is decidable, then \bar{L} is decidable as well, and so L and \bar{L} are Turing-recognizable. If L and \bar{L} are both Turing-recognizable, we can build a decider for L by simulating the machines for L and \bar{L} in “parallel”, and accept or reject depending on which of them accepts.
- A corollary to the above theorem is that if L is Turing-recognizable and not Turing-decidable, then \bar{L} is not Turing-recognizable. Hence, $\overline{A_{TM}}$ is not even Turing-recognizable (Corollary 4.23).

Reductions

Reductions are a technique to deduce undecidability of problems using another problem that is known to be undecidable.

A language S reduces to a language T if, given a TM deciding T , we can build a TM that decides S .

In other words, if T is decidable, then S is decidable. Which, paraphrased, says that if S is undecidable then T is undecidable.

Hence, to show T is undecidable, we choose a language S that we know is undecidable, and reduce S to T .

Many reductions are from A_{TM} ; to show L is undecidable, we try to reduce A_{TM} to L , i.e. assuming we have a decider for L , we show that we can build a decider for A_{TM} . Since A_{TM} has no decider, it follows that L has no decider.

Reduction proofs are important to understand and learn. Reductions from A_{TM} to languages that accept Turing machine descriptions often go roughly like this:

- Assume L has a decider R ; we build a decider D for A_{TM} .
- D takes as input $\langle M, w \rangle$.
 D then *modifies* M to construct a TM $N_{M,w}$.
 D then feeds this machine $N_{M,w}$ to R .
Depending on whether R accepts or rejects, D accepts or rejects (sometimes switching the answer).

Using reductions we can prove several languages undecidable. For example, (see Theorems 5.1, 5.2, 5.3, 5.4)

- $HALT_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$ is undecidable.
It is Turing-recognizable, though.
- $E_{TM} = \{\langle M, w \rangle \mid M \text{ is a TM and } L(M) = \emptyset\}$ is undecidable.
It is not even Turing-recognizable since its complement is Turing-recognizable.
- $REGULAR_{TM} = \{\langle M \rangle \mid L(M) \text{ is regular}\}$ is undecidable.
- $EQ_{TM} = \{\langle M_1, M_2 \rangle \mid L(M_1) = L(M_2)\}$ is undecidable.
It is not even Turing-recognizable since its complement is Turing-recognizable.

Rice's theorem generalizes many undecidability results. Consider a class P of Turing machine descriptions. Assume that P is a property of Turing machines that depends only on the language of the Turing machines (i.e. if M and N are Turing machines accepting the *same* language, then either both are in P or both are not in P). Also assume that P is not the empty set nor the set of all Turing machines. Then P is *undecidable*.

Note that if P was the empty set or the set of all Turing machine descriptions, then clearly it is decidable.

Note: There will be a question on reductions in the exam. The reduction will be one which is a direct corollary of Rice's theorem, but you will be asked to give a proof without using Rice's theorem.

Other undecidability problems

There were several other problems that were shown to be undecidable. Knowing these are undecidable is important; you will not be asked for proofs of any of these, however:

- A linear bounded automaton (LBA) is a Turing machine that uses only the space occupied by the input, and does not use any extra cells. The emptiness problem for LBAs is undecidable (Theorem 5.10): i.e. $E_{LBA} = \{\langle M \rangle \mid M \text{ is an LBA and } L(M) \neq \emptyset\}$ is undecidable.
However, the membership problem for LBAs is decidable (Theorem 5.9):
i.e. $A_{LBA} = \{\langle M, w \rangle \mid M \text{ is an LBA accepting } w\}$ is decidable.

- Checking if a context-free grammar accepts all words is undecidable (Theorem 5.13).
- The Post Correspondence Problem is undecidable (Theorem 5.15).

6 Summary of closure properties and decision problems

Closure Properties

Here’s a summary of closure properties for the various classes of languages:

	Union	Intersection	Complement	Kleene-*	Homomorphisms
REG	Yes	Yes	Yes	Yes	Yes
CFL	Yes	No	No	Yes	Yes
TM-DEC	Yes	Yes	Yes	Yes	Yes
TM-REC	Yes	Yes	No	Yes	Yes

The results for regular languages are in Sipser and class notes.

See also <http://www.cs.uiuc.edu/class/fa07/cs273/Handouts/closure/regular-closure.html>.

Sipser doesn’t cover closure properties of context-free languages very clearly. However, note that closure under union is easy as it is simple to combine two grammars to realize their union. Non-closure under intersection follows from the fact that $L_1 = \{a^i b^j c^k \mid i = j\}$ and $L_2 = \{a^i b^j c^k \mid j = k\}$ are both context-free, but their intersection $L_1 \cap L_2 = \{a^i b^j c^k \mid i = j = k\}$ is not. Non-closure under complement is easy to see as $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not context-free but its complement is context-free. Closure under Kleene-* and homomorphisms are easy as one can easily transform a grammar to do these operations.

See <http://www.cs.uiuc.edu/class/fa07/cs273/Handouts/closure/cfl-closure.html> for more detailed proofs.

Turning to TM-DEC, they are closed under union as you can run TM M_1 followed by M_2 , and accept if one of them accept. For intersection, you can run them one after the other, and accept if both accept. Closure under complement is easy as we can swap the accept and reject states of a TM. Kleene-* and homomorphisms were not covered, but it is easy to see that TM-DEC languages are closed under these operations (try them as an exercise!).

Finally, TM-REC is closed under union as you can run two Turing machines in “parallel”, and accept if one of them accepts. The class is closed under intersection, as we can run them one after another, and accept if both accept. (Note the subtleties of the construction here; simulating a TM that recognizes a language has to be done carefully as it may not halt). The class of TM-REC languages is not closed under complement (for example, A_{TM} is TM-REC but its complement is not). In fact, if L is TM-REC and its complement is also TM-REC, then L is TM-DEC. Since we know A_{TM} is not TM-DEC, it follows that its complement is not TM-REC. TM-REC languages are closed under Kleene-* and homomorphisms— we leave these as exercises.

Decision problems

For each class of languages, let's consider four problems—

Membership: Given a language L , and a word w , check if $w \in L$.

Emptiness: Given a language L , check if $L = \emptyset$.

Inclusion: Given two languages L_1 and L_2 , check if $L_1 \subseteq L_2$.

Equivalence: Given two languages L_1 and L_2 , check if $L_1 = L_2$.

For each of the problems above, we will consider the cases when the language(s) are given as finite automata, PDAs or TM. Note that the problem does not change much if we give it using a grammar instead of a PDA, or a regular expression instead of an automaton, because we can always convert them to PDAs or automata.

	Membership	Emptiness	Inclusion	Equivalence
REG	Yes	Yes	Yes	Yes
CFL	Yes	Yes	No	No
TM-DEC	No	No	No	No
TM-REC	No	No	No	No

Notice that regular languages are the most tractable class, and context-free languages have the important property that membership (Theorem 4.7) and emptiness (Theorem 4.8) are decidable. In particular, membership of context-free languages is close to the problem of parsing, and hence is an important algorithmic problem. Context-free languages do not admit a decidable inclusion or equivalence problem (Theorem 5.13 shows that checking if a CFG generates all words is undecidable; we can reduce this to both the problem of inclusion— $L(A) \subseteq L(B)$ —and equivalence— $L(A) = L(B)$ —by setting A to be a CFG generating all words).

For Turing machines, almost nothing interesting is decidable. However, note that the membership problem for Turing machines (A_{TM}) and the emptiness problem for Turing machines (E_{TM}) are both Turing-recognizable.