Theory of Computation

ECE 1762 Algorithms and Data Structures
Fall Semester, U of Toronto

Computations are designed for processing information. They can be as simple as an estimation for driving time between cities, and as complex as weather prediction. An *algorithm* may be described as finite sequence of instructions that – when confronted with a question of some kind – will invariably compute the correct answer.

The study of computation aims at providing an insight into the characteristics of computations. The theory of computation reveals that *there are problems that cannot be solved*. And of the problems that can be solved, there are some that require infeasible amounts of resources (e.g., millions of years of computation time). These revelations might seem discouraging, but they have the benefit of warning against trying to solve such problems. On the other hand, the study of computation provides tools for identifying problems that can feasibly be solved, as well as tools for designing such solutions.

1 Preliminaries

Any formal discussion about computation and programs requires a clear understanding of these notions. In this section, we define some basic concepts and notation.

A finite, nonempty ordered set will be called an *alphabet* if its elements are *symbols* , or *characters*. A finite sequence of symbols from a given alphabet will be called a *string* over the alphabet. A string that consists of a sequence $a_1, a_2, \ldots, a_n$ of symbols will be denoted by the juxtaposition $a_1 a_2 \ldots a_n$. Strings that have zero symbols, called empty strings, will be denoted by ϵ.

Example 1 $\Sigma_1 = \{a, ..., z\}$ *and* $\Sigma_2 = \{0, ..., 9\}$ *are alphabets. abb is a string over* Σ_1*, and* 123 *is a string over* Σ_2*. ba*12 *is not a string over* Σ_1*, because it contains symbols that are not in* Σ_1*. Similarly,* 314... *is not a string over* Σ_2*, because it is not a finite sequence. On the other hand,* ϵ *is a string over any alphabet. The empty set* $\emptyset$ *is not an alphabet because it contains no elements. The set of natural numbers is not an alphabet, because it is not finite. The union* $\Sigma_1 \cup \Sigma_2$ *is an alphabet only if an ordering is placed on its symbols.*

An alphabet of cardinality 2 is called a *binary alphabet*, and strings over a binary alphabet are called *binary strings*. Similarly, an alphabet of cardinality 1 is called a *unary alphabet*, and strings over a unary alphabet are called *unary strings*. The *length* of a string α is denoted $|\alpha|$ and assumed to equal the number of symbols in the string.

Example 2 $\{0, 1\}$ *is a binary alphabet, and* $\{1\}$ *is a unary alphabet.* 11 *is a binary string over the alphabet* $\{0, 1\}$*, and a unary string over the alphabet* $\{1\}$*.* $|11| = 2, |\epsilon| = 0$*, and* $|01| + |1| = 3$*.*

The string consisting of a sequence α followed by a sequence β is denoted $\alpha\beta$. The string $\alpha\beta$ is called the *concatenation* of α and β. The notation α^i is used for the string obtained by concatenating i copies of the string α.

Example 3 *The concatenation of the string* 01 *with the string* 100 *gives the string* 01100*. The concatenation* $\epsilon\alpha$ *of* ϵ *with any string* α*, and the concatenation* $\alpha\epsilon$ *of any string* α *with* ϵ *give the string* α*. In particular,* $\epsilon\epsilon = \epsilon$*. If* $\alpha = 01$*, then* $\alpha^0 = \epsilon$*,* $\alpha^1 = 01$ *,* $\alpha^2 = 0101$*.*

If $\alpha = a_1...a_n$ for some symbols $a_1, ..., a_n$ then $a_n...a_1$ is called the reverse of α, denoted α^R. For instance, if $\alpha = 001$, then $\alpha^R = 100$.

The set of all the strings over an alphabet Σ is denoted by Σ^*. Σ^+ denotes the set $\Sigma^* - \{\epsilon\}$. The lexicographical ordering of strings in $\Sigma^* = \{0,1\}^*$ follows the following order $\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, ...$

1.1 Languages

In general, if Σ is an alphabet and L is a (possibly infinite) subset of Σ^*, then L is said to be a *language* over Σ, or simply a language if Σ is implied. Each element of L is said to be a *string* of the language.

Example 4 *$\{0, 11, 001\}$, $\{\epsilon, 10\}$, and $\{0,1\}^*$ are subsets of $\{0,1\}^*$, and so they are languages over the alphabet $\{0,1\}$.*

The empty set $\emptyset$ and the set $\{\epsilon\}$ are languages over every alphabet. $\emptyset$ is a language that contains no string. $\{\epsilon\}$ is a language that contains just the empty string.

The *union* of two languages L_1 and L_2, denoted $L_1 \cup L_2$, refers to the language that consists of all the strings that are either in L_1 or in L_2, that is, to $\{x | x \in L_1 \text{ or } x \in L_2\}$. The intersection of L_1 and L_2, denoted $L_1 \cap L_2$, refers to the language that consists of all the strings that are both in L_1 and L_2, that is, to $\{x | x \in L_1 \text{ and } x \in L_2\}$. The complementation of a language L over Σ, denoted $\bar{L}$, refers to the language that consists of all the strings over Σ that are not in L, that is, to $\{x | x \in \Sigma^* \text{ but not in } L\}$.

Example 5 *Consider the languages $L_1 = \{\epsilon, 0, 1\}$ and $L_2 = \{\epsilon, 01, 11\}$. The union of these languages is $L_1 \cup L_2 = \{\epsilon, 0, 1, 01, 11\}$, their intersection is $L_1 \cap L_2 = \{\epsilon\}$, and the complementation of L_1 is $\bar{L_1} = \{00, 01, 10, 11, 000, 001, ...\}$.*

$\emptyset \cup L = L$ for each language L. Similarly, $\emptyset \cap L = \emptyset$ for each language L. On the other hand, $\bar{\emptyset} = \Sigma^*$ and $\bar{\Sigma^*} = \emptyset$ for each alphabet Σ.

The *difference* of L_1 and L_2, denoted $L_1 - L_2$, refers to the language that consists of all the strings that are in L_1 but not in L_2, that is, to $\{x | x \in L_1 \text{ but not in } L_2\}$. The concatenation of L_1 with L_2, denoted $L_1 L_2$, refers to the language $\{xy | x \in L_1 \text{ and } y \in L_2\}$.

L^i will be used to denote the concatenating of i copies of a language L, where L^0 is defined as $\{\epsilon\}$. The set $L^0 \cup L^1 \cup L^2 \cup ...$, called the *Kleene closure* or just the *closure* of L, is denoted by L^*. The set $L^1 \cup L^2 \cup ...$, called the positive closure of L, will be denoted by L^+. L^i consists of those strings that can be obtained by concatenating i strings from L. L^* consists of those strings that can be obtained by concatenating an arbitrary number of strings from L.

Example 6 *Consider the pair of languages $L_1 = \{\epsilon, 0, 1\}$ and $L_2 = \{01, 11\}$. For these languages $L_1^2 = \{\epsilon, 0, 1, 00, 01, 10, 11\}$, and $L_2^3 = \{010101, 010111, 011101, 011111, 110101, 110111, 111101, 111111\}$. In addition, ϵ is in L_1^*, in L_1^+, and in L_2^* but not in L_2^+.*

A language can also be described using the properties of its member strings. For example, the language of all strings with exactly one 0 over $\{0,1\}$ is $L = \{0, 01, 10, 011, 101, ...\}$. The language $L = \{0^p | p \text{ is a prime number}\}$ is the infinite set $\{00, 000, 00000, ...\}$.

1.2 Regular Languages

A *regular expression* over a finiate alphabet Σ is defined as follows:

- ϵ is a regular expression.
- $\forall a \in \Sigma$, a is a regular expression.
- If R, S are regular expressions, then $R + S$ is a regular expression ("R or S").

- If R, S are regular expressions, then RS is a regular expression (concatenation).
- If R is a regular expression, then R^* is a regular expression (Kleene star).
- If R is a regular expression, then (R) is a regular expression (parenthesization).

A *regular language* corresponding to a regular expression R, denoted $\mathcal{L}(R)$ is the set of all words that the regular expression R denotes. We will see that not all languages can be expressed using regular expressions.

Example 7 *Consider the following regular expressions.*

- $\mathcal{L}(0) = \{0\}$.
- $\mathcal{L}((0+1)(0+1)) = \{00, 01, 10, 11\}$.
- $\mathcal{L}(0^*) = \{\epsilon, 0, 00, 000, ...\}$.
- *The regular expression* $(0+1)^*1$ *denotes all strings of* 0*s and* 1*s that end with a* 1.
- *The regular expression* 1^*01^* *denotes the set of strings with exactly one* 0 *over* $\{0,1\}$, *i.e.* $\mathcal{L}(1^*01^*) = \{0, 01, 10, 011, 101, ...\}$.
- $\mathcal{L}((1^*01^*01^*)^*)$ *denotes the set of strings with an even number of* 0*'s.*
- $\mathcal{L}((0+1)^*001)$ *denotes the set of strings ending with* 001.
- $\mathcal{L}(c^*(a+(bc^*))^*)$ *denotes the set of all strings over* $\{a, b, c\}$ *that do not have the substring* ac.

2 Finite Automata

Modern computers are often viewed as having three main components: the central processing unit, auxiliary memory, and input-output devices. In this section, we take up a severely restricted model of an actual computer called a *finite automaton*. It shares with a real computer the fact that its central processor is of fixed finite capacity. However, a finite automaton has no auxiliary memory at all. It receives its input as a string; the input string is delivered to it on an input tape. Its output is a yes/no (or equivalently 0/1) answer stating whether the input is considered acceptable.

A finite automata is a *language recognition* device. We say that a finite automaton *accepts language* L, if it *accepts* (i.e. answers yes for) all strings in L and *rejects* (i.e. answers no for) all strings outside L. We can think of a finite automata as a specifically designed algorithm for some language L, to answer the questions of the form "Is the input string w a member of L?".

We will be avoiding formal definitions throughout these notes and will instead rely on figures and examples to illustrate ideas. In the context of a compiler, a finite automaton corresponds to a lexical analyzer, which recognizes and groups the tokens of a language.

2.1 Deterministic Finite Automata

The simplest way to describe a Deterministic Finite Automaton (DFA) is using an example:

Example 8 *The following is the transition diagram of a DFA* M*:*

In the DFA M *shown in the following figure,* q_0 *is the* initial state *(has a floating arrow pointing to it),* q_1 *is an* accepting final state *(has two circles), and the arrows denote state transitions according to the input*

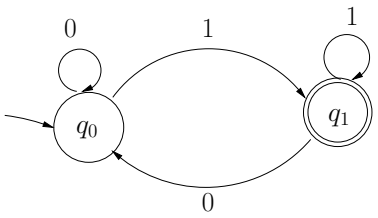

Figure 1: A Deterministic Finite Automata

Table 1: State transitions of M

current state	*input symbol*	*next state*
q_0	0	q_0
q_0	1	q_1
q_1	0	q_0
q_1	1	q_1

symbols. The language recognized *or* accepted *by a DFA is the set of strings such that there exists a path from the initial state to some accepting final state following the edges in the transition diagram of the DFA.*

Given the string $w = 1101$*, let us go through the computation of this DFA. The input string* w *is read from left to right. Starting from* q_0*, the leftmost symbol* 1 *is first read, so the DFA goes to state* q_1*. Next,* 1 *is read again, and the DFA stays in state* q_1*. Then* 0 *is read, and the DFA goes back to* q_0*. Finally,* 1 *is read, and* M *returns to* q_1*. Since* q_1 *is an accepting state,* M accepts *the string* w*. In other terms,* $w = 1101$ *belongs to the language of* M*, denoted as* $w \in \mathcal{L}(\mathcal{M})$*.*

This finite automaton is deterministic because there is one possible state transition for each state and input symbol.

Think about M *and try to see that* $\mathcal{L}(\mathcal{M}) = \{w \in \{0,1\}^* | w$ *ends with* $1\}$*. This language can be represented by the regular expression* $(0+1)^*1$*.*

Theorem 1 *A language is regular if and only if it is accepted by some DFA.*

Theorem 1 implies that any language which can be expressed using a regular expression can be accepted by some DFA. And conversely, if a language cannot be expressed using a regular expression, there is no DFA accepting it.

2.2 Nondeterministic Finite Automata

The concept of nondeterminism alludes to one or more choice points where multiple different computations are possible, without any specification of which one will be taken.

A Nondeterministic Finite Automaton (NFA) is a finite automaton that can have many possible state transitions from a given state and for a given input symbol. An NFA can also have transitions with the ϵ symbol (i.e. without reading any input symbol). For each of these possibilities, the NFA "splits" into another copy of itself, where each copy follows one state transition possibility. By the end of the input string, if any of the copies reach an accepting final state, the NFA accepts the input string. Let us illustrate this using an example.

Example 9 *Consider the following NFA* N*:*

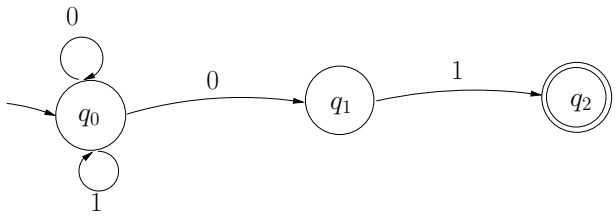

Figure 2: A Nondeterministic Finite Automata

Notice that when the NFA N is at state q_0 and the input symbol 0 is read, there are two possible state transitions: Either N stays in q_0 or it goes to q_1.

Let us follow the execution of N over the input string $w = 1001$. After reading 1, N stays in q_0. Then 0 is read, and N splits into two: One version stays in q_0 and the other goes to q_1. After reading the next 0, the version that was in q_1 "dies" because it has nowhere to go, whereas the version that was in q_0 again splits into two just like before, one going to q_1 and the other staying in q_0. Finally once the last symbol in w, 1, is read, the version of N which was at q_1 can go to q_2, which is an accepting final state. Therefore, N accepts $w = 1001$.

Think about N and try to see that $\mathcal{L}(\mathcal{N}) = \{w \in \{0,1\}^* | w$ ends with $01\}$. This language can be represented by the regular expression $(0+1)^*01$.

Theorem 2 *For each NFA, there is an equivalent DFA accepting the same language.*

Theorem 2 implies that NFAs are equivalent to DFAs in their expressive power. In other terms, any language accepted by an NFA can be accepted by some DFA. In fact, there are procedures to build an equivalent DFA for each NFA.

We end this section by giving a few examples of languages which are not accepted by NFAs or DFA (i.e. are non-regular languages).

Example 10 *Consider the following languages.*

- $\{0^p | p$ *is prime*$\}$ *is not a regular language. Intuitively, the reason is that a finite automaton, having a finite memory represented by its states, cannot "remember" infinitely many prime numbers.*
- *The language* $\{0^n 1^n | \forall$ *integers* $n \geq 0\} = \{\epsilon, 01, 0011, 000111, ...\}$ *is not regular. Here, the finite automaton cannot remember the number of* 0*'s in order to start counting the* 1*'s because it has a finite memory, and the number of* 0*'s can be increased indefinitely.*
- *The language* $L = \{ww^R | w \in \{0,1\}^*\}$ *is not regular for similar reasons.*

In the following section, we will show that the last two languages can be accepted by an automaton which uses an infinite stack as an auxiliary memory.

3 Context-Free Languages

Let us consider the language $L = \{0^n 1^n | \forall$ integers $n \geq 0\} = \{\epsilon, 01, 0011, 000111, ...\}$. In the previous section, we noted that this language is not regular. Now, we will try to *generate* this language using certain *rules*.

Let S be a new symbol interpreted as "a string in the language". Now observe that if S is a string in the described language L, then $0S1$ is also a string in L. Furthermore, as a base case, $S = \epsilon \in L$. Therefore, we can generate L by repeated applications of the followsing two *rules*:

$$S \to \epsilon$$
$$S \to 0S1$$

These rules can be abbreviated as $S \leftarrow 0S1|\epsilon$. For example, using these rules, we can have a *derivation* for $0^3 1^3$ as follows:

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111$$

These rules are called Context-Free Grammars (CFGs) and languages derived by such rules are called Context-Free Languages (CFLs). Intuitively, a CFG strict language specs by which all words of a language can be generated by well defined rules. Therefore, $L = \{0^n 1^n | \forall \text{ integers } n \geq 0\}$ is a CFL.

Example 11 *Consider the non-regular language $L = \{ww^R | w \in \{0,1\}^*\}$. The strings in this language L are called palindromes. Let us try to derive a CFG for deriving palindromes. Notice that if S is a palindrome, then $0S0$ and $1S1$ are also palindromes. Furthermore, as a base case, $S = \epsilon$ is a palindrome. Therefore, a CFG deriving all palindromes is: $S \to 0S0|1S1|\epsilon$.*

3.1 Nondeterministic Pushdown Automata

Just as NFAs and DFAs recognize regular languages, Nondeterministic Pushdown Automata (NPDAs) are the machines that recognize CFLs. An NPDA is basically an NFA which has also access to an infinite stack, as shown in the following figure. A stack is a memory storage device which can only be accessed from the top by pushing or popping elements. In the context of compilers, an NPDA corresponds to the syntax analyzer (or parser), which checks whther the stream of tokens obeys language specifications.

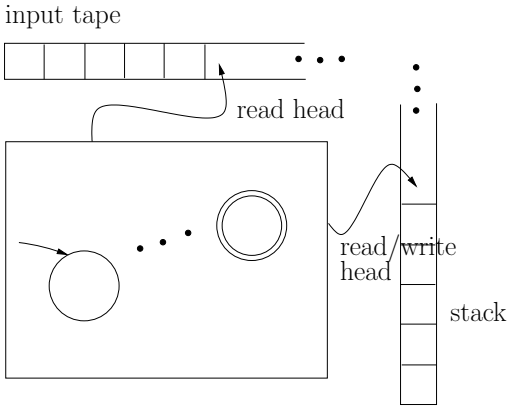

Figure 3: A Nondeterministic Pushdown Automaton

In Figure 3, the NPDA has a read head, which reads from the input tape, along with read/write head, which can write into or read from the stack. The NPDA can recognize when the stack is empty, and hence uses the stack to "count" symbol occurrences in the input string. We illustrate this using the following examples.

Example 12 *Let us consider the CFL $L = \{0^n 1^n | \forall$ integers $n \geq 0\}$. An NPDA accepting L operates as follows. Initially, the stack is empty. All the encountered 0's in the input string are pushed into the stack. As*

soon as the first 1 *is encountered in the input, the NPDA starts popping elements from the stack. Basically, a* 0 *is popped for each* 1 *read. If the input terminates exactly when the stack becomes empty, then the NPDA accepts. Otherwise (i.e. if a* 0 *is encountered after a* 1*, or if the input terminates and the stack is not empty, or if the stack is empty and the input is not fully read yet), the NPDA rejects. Clearly, such an NPDA accepts* L*.*

Example 13 *Next, consider the CFL* $L = \{ww^R | w \in \{0,1\}^*\}$*. An NPDA accepting all palindromes will need to use non-determinism. The NPDA will push every symbol of the input into the stack and split into two: The first version will always assume that* w *is over and will try to start reading* w^R *by popping elements of the stack if they are in the correct order, while the second version will assume* w *is not over yet, and will keep pushing the symbols of* w*. If any of the copies of the NPDA succeeds in popping the whole stack correctly exactly when the input terminates, the NPDA accepts. Otherwise, the NPDA rejects the input string.*

Theorem 3 *The class of languages accepted by NPDAs is exactly the class of CFLs.*

Finally, note that there are languages that are not CFLs (i.e. that are not accepted by any NPDA). For example, the languages $\{www | \forall w \in \{0,1\}^*\}$, $\{0^n 1^n 2^n | \forall n \geq 0\}$ are not CFLs. Intuitively, the stack cannot be used to detect the "middle" of the string as we did for the two CFLs we analyzed. Furthermore, $\{0^p | p \text{ is prime}\}$ is also not a CFL.

4 Turing Machines

We have seen in the last two sections that neither finite automata nor pushdown automata can be regarded as truly general models for computers, since they are not capable of recognizing even such simple languages as $\{0^n 1^n 2^n | \forall n \geq 0\}$. In this section, we take up the study of devices, called *Turing Machines* (TMs) that can recognize this and many more complicated languages.

The TM is named after its inventor Alan Turing (1912-1954), who was an English mathematician, logician, cryptanalyst, and computer scientist. During the Second World War, Turing worked for the Government Code and Cypher School at Bletchley Park, Britain's codebreaking center. For a time he was head of Hut 8, the section responsible for German naval cryptanalysis. He devised a number of techniques for breaking German ciphers, including the method of *the bombe*, an electromechanical machine that could find settings for the Enigma machine.

Turing's homosexuality, which was illegal and considered to be a mental illness during his lifetime, resulted in a criminal prosecution in 1952. He accepted treatment with female hormones as an alternative to going to prison. He died in 1954, several weeks before his 42nd birthday, from an apparently self-administered cyanide poisoning.

A TM is a finite state machine connected to an infinite memory tape which initially holds the input string and which can be written to and read from using a read/write head. Communication between the finite state machine and the memory tape is provided by this read/write head which can move in both directions.

The figure on the next page shows a typical TM.

The TM operates in discrete steps; at each step it performs two functions in a way dependent on the current state of the finite state machine and the tape symbol currently scanned by the read/write head.

1. Put the finite state machine in a new state.
2. Either
   (a) Write a symbol in the tape square currently scanned, replacing the one already there; or
   (b) Move the read/write head one tape square to the left or right.

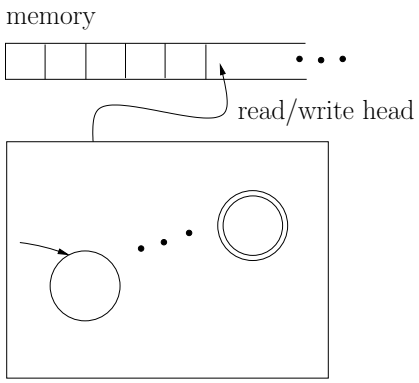

Figure 4: A Turing Machine

The tape has a left end, but it extends indefinitely to the right.

A TM is supplied with input by inscribing that input string on tape squares at the left end of the tape. The rest of the tape initially contains blank symbols. The machine is free to alter its input in any way it sees fit, as well as write on the unlimited blank portion of the tape to the right.

Because a TM can write on its tape, it can leave an answer on the tape at the end of a computation. Therefore, we do not need to provide special final states, as before. There is a special *halt* state that is used to signal the end of computation and which takes effect immediately. Unlike DFAs or NPDAs, it is possible for a TM to never halt on a certain input, and instead *loop* over non-halting states indefinitely.

We say that a TM T *accepts* a language L, if T accepts (i.e. outputs 'yes' for) all strings in L, whereas it can reject or loop on strings not in L. We say that a TM T *decides* a language L, if T accepts all strings in L and rejects all strings not in L.

Here, we should note the correspondance between languages and *decision problems*, which are problems whose answer are 'yes' or 'no' for each input instance. A language corresponding to a decision problem contain all the input strings with answer 'yes'. A TM which decides such a language equivalently solves the corresponding decision problem. Also note that any decision problem can be encoded using a binary alphabet $\{0,1\}$ without asymptotically increasing the problem size. We will denote the encoding of an object O in binary as $\langle O \rangle$.

Example 14 *Consider the language* $\{\langle p \rangle \in \{0,1\}^* | p \text{ is prime}\} = \{10, 11, 101, 111, ...\}$ *where the corresponding binary number encoding is used for each integer. This language corresponds to the decision problem which asks whether* $\langle p \rangle$ *is prime.*

The *Church-Turing thesis* states that if an algorithm exists then there is an equivalent Turing machine for that algorithm. Today the thesis has near-universal acceptance.

A *Universal TM* U works as follows: On input $\langle M, w \rangle$ it simulates the TM M on w. It accepts if M accepts and rejects if M rejects. This is the basis of today's computer, which "simulates" different TMs (programs).

Is every problem acceptable? Each TM accepts a certain language. However, it can be proved that the set of all TMs is countably infinite, whereas the set of all languages is uncountably infinite. Therefore, there are some languages which do not have any TMs which accept them. I.e. not all problems are acceptable.

Here, we will show an example of a language which is not decidable. This is called the Halting problem, and the proof of undecidability is done using a technique called *diagonalization*. The halting problem is famous because it was one of the first problems proven to be algorithmically undecidable. This means there is no algorithm which can be applied to any arbitrary program and input to decide whether the program stops

when run with that input.

Let $L = \{\langle M, w\rangle | M \text{ is a } TM \text{ and } M \text{ halts (i.e. does not loop) on } w\}$. We will show that L is not decidable by contradiction.

Assume towards a contradiction that there exists a TM H deciding L. This means that H takes $\langle M, w\rangle$ and returns true iff M halts on w. It must return false otherwise, and not go into an infinite loop.

Now consider another TM $D(\langle M\rangle)$ which calls H on $\langle M, \langle M\rangle\rangle$ and outputs the opposite of H's output. Since H always terminates, D must also always terminate.

Finally, let us give D its own description, i.e. run $D(\langle D\rangle)$. This will call $H(\langle D, \langle D\rangle\rangle)$. Now H accepts if D halts, in which case D, which must output the opposite of H, rejects. And H rejects if D loops, in which case D must accept. Note the contradiction in the second case where D must both loop and accept. Therefore, L is not decidable.

References

[1] Harry R. Lewis and Christos H. Papadimitriou, *Elements of the Theory of Computation*. Prentice-Hall, 1981.

[2] Hopcroft, John E. and Ullman, Jeffrey D., *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc., 1990.