Turing Machine

Dr. Mousumi Dutt

Unrestricted Grammar

Type-0 grammars generate recursively enumerable languages. The productions have no restrictions. They are any phase structure grammar including all formal grammars.

They generate the languages that are recognized by a Turing machine.

The productions can be in the form of $\alpha \rightarrow \beta$ where α is a string of terminals and nonterminals with at least one non-terminal and α cannot be null. β is a string of terminals and non-terminals.

Example

$S \rightarrow ACaB$

$Bc \rightarrow acB$

$CB \rightarrow DB$

$aD \rightarrow Db$

Unrestricted Grammar

The productions of a grammar have the form $(V \cup T)+ \rightarrow (V \cup T)*$

The other grammar types we have considered (left linear, right linear, linear, context free) restrict the form of productions in one way or another. An *unrestricted grammar* does not.

In what follows, we will attempt to show that unrestricted grammars are equivalent to Turing machines. Bear in mind that

- A language is *recursively enumerable* if there exists a Turing machine that accepts every string of the language, and does not accept strings that are not in the language.
- "Does not accept" is *not* the same as "reject" -- the Turing machine could go into an infinite loop instead, and never get around to either accepting *or* rejecting the string.

Our plan of attack is to show that the languages generated by unrestricted grammars are precisely the recursively enumerable languages.

Turing Machine: Introduction

In automaton, **Unrestricted Grammar** or **Phrase Structure Grammar** is most general in the **Chomsky Hierarchy of classification**. This is **type0** grammar, generally used to generate **Recursively Enumerable languages**. It is called unrestricted because no other restriction in made on this except each of their left hand sides being non empty. The left hand sides of the rules can contain terminal and non terminal, but the condition is at least of them must be non terminal.

A **Turning Machine** can simulate an **Unrestricted Grammar** and an **Unrestricted Grammar** can simulate **Turning Machine** configurations. It can always be found for the language recognized or generated by any **Turning Machine**.

Turing Machine: Introduction

Up to now we have seen solution for searching text, parsing, etc.

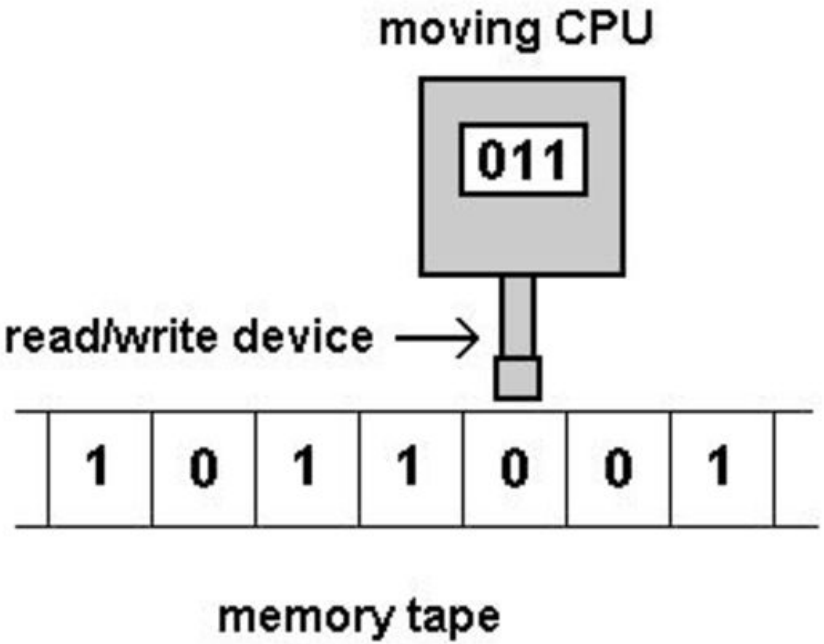
Undecidable Problems: cannot solve by computers. A vulnerable formalism for computers called Turing Machine is introduced which does not like PC

Intractable Problems: There are some problems - decidable but requires large amount of time to solve

(greater difficulty for programmer -> heuristic approach)

The theory of undecidability based not on programs in C or another language but based on a very simple model of a computer called the Turing Machine. Its a finite automaton that has a single tape of infinite length on which it may read and write data

Turing Machine



A turing machine consists of a tape of infinite length on which read and writes operation can be performed. The tape consists of infinite cells on which each cell either contains input symbol or a special symbol called blank. It also consists of a head pointer which points to cell currently being read and it can move in both directions.

Turing Machine

A TM is expressed as a 7-tuple (Q, T, B, ∑, δ, q0, F) where:

- **Q** is a finite set of states
- **T** is the tape alphabet (symbols which can be written on Tape)
- **B** is blank symbol (every cell is filled with B except input alphabet initially)
- ∑ is the input alphabet (symbols which are part of input alphabet)
- **δ** is a transition function which maps $Q \times T \rightarrow Q \times T \times \{L,R\}$. Depending on its present state and present tape alphabet (pointed by head pointer), it will move to new state, change the tape symbol (may or may not) and move head pointer to either left or right.
- **q0** is the initial state
- **F** is the set of final states. If any state of F is reached, input string is accepted.

Turing Machine: Example

Let us construct a turing machine for L={0^n1^n|n>=1}

- Q = {q0,q1,q2,q3} where q0 is initial state.
- T = {0,1,X,Y,B} where B represents blank.
- ∑ = {0,1}
- F = {q3}

	0	1	X	Y	B
q0	(q1,X,R)			(q3,Y,R)	
q1	(q1,0,R)	(q2,Y,L)		(q1,Y,R)	
q2	(q2,0,L)		(q0,X,R)	(q2,Y,L)	
q3				(q3,Y,R)	**Halt**

Turing Machine: Example

Let us see how this turing machine works for 0011. Initially head points to 0 which is underlined and state is q0 as:

The move will be δ(q0, 0) = (q1, X, R). It means, it will go to state q1, replace 0 by X and head will move to right as:

The move will be δ(q1, 0) = (q1, 0, R) which means it will remain in same state and without changing any symbol, it will move to right as:

Turing Machine: Example

The move will be δ(q1, 1) = (q2, Y, L) which means it will move to q2 state and changing 1 to Y, it will move to left as:

Working on it in the same way, the machine will reach state q3 and head will point to B as shown:

Using move δ(q3, B) = halt, it will stop and accepted.

Turing Machine: Example

Question: A single tape Turing Machine M has two states q0 and q1, of which q0 is the starting state. The tape alphabet of M is {0, 1, B} and its input alphabet is {0, 1}. The symbol B is the blank symbol used to indicate end of an input string. The transition function of M is described in the following table.

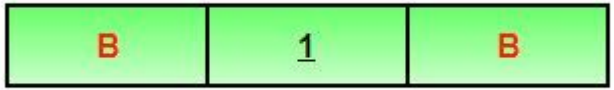
	0	1	B
q0	q1,1,R	q1,1,R	Halt
q1	q1,1,R	q0,L,R	q0,B,L

The entry (q1, 1, R) in row q0 and column 1 signifies that if M is in state q0 and reads 1 on the current tape square, then it writes 1 on the same tape square, moves its tape head one position to the right and transitions to state q1. Which of the following statements is true about M?

1. M does not halt on any string in (0 + 1)+
2. M does not halt on any string in (00 + 1)*
3. M halts on all string ending in a 0
4. M halts on all string ending in a 1

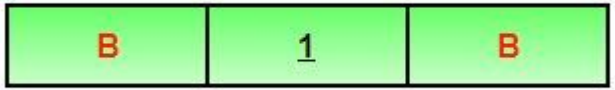
Turing Machine: Example

Solution: Let us see whether machine halts on string ‘1’. Initially state will be q0, head will point to 1 as:



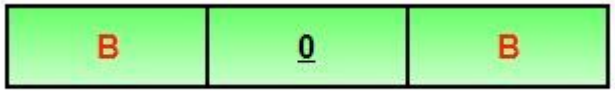
Using δ(q0, 1) = (q1, 1, R), it will move to state q1 and head will move to right as:

Using δ(q1, B) = (q0, B, L), it will move to state q0 and head will move to left as:



It will run in the same way again and again and not halt.

Option D says M halts on all string ending with 1, but it is not halting for 1. So, option D is incorrect.

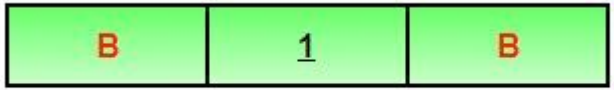


Let us see whether machine halts on string ‘0’. Initially state will be q0, head will point to 1 as:

Turing Machine: Example

Using δ(q0, 0) = (q1, 1, R), it will move to state q1 and head will move to right as:

Using δ(q1,B)=(q0,B,L), it will move to state q0 and head will move to left as:

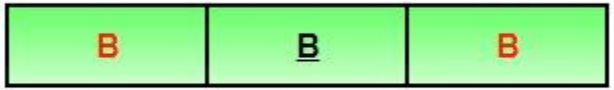


It will run in the same way again and again and not halt.

Option C says M halts on all string ending with 0, but it is not halting for 0. So, option C is incorrect.

Option B says that TM does not halt for any string (00 + 1)*. But NULL string is a part of (00 + 1)* and TM will halt for NULL string. For NULL string, tape will be,

Turing Machine: Example



Using δ(q0, B) = halt, TM will halt. As TM is halting for NULL, this option is also incorrect.
So, option (A) is correct.

Turing Machine: Instantaneous Description

Let us consider the string $X_1\ X_2\ \ldots\ X_{i-1}\ q\ X_i\ X_{i+1}\ \ldots\ X_n$

1. q is the state of the turing machine
2. The tape head is running the ith symbol from the left
3. Between two blanks the portion of the tape is $X_1\ X_2\ \ldots\ X_n$

Turing Machine: Instantaneous Description

Let the Turing Machine be $M=(Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Suppose $\delta(q, X_i) = (p, Y, L)$ -> next move is leftward
Then
$X_1 X_2 \dots X_{i-1} q X_i X_{+1} \dots X_n \vdash_M X_1 X_2 \dots X_{i-2} p X_{i-1} Y X_{i+1} \dots X_n$

If i=1, then M moves to the blank to the left of X1->
$X_1 X_2 \dots X_n \vdash_M p B Y X_2 \dots X_n$

If i=n and Y=B,
$X_1 X_2 \dots X_{n-1} q X_n \vdash_M X_1 X_2 \dots X_{n-2} p X_{n-1}$

Turing Machine: Instantaneous Description

Let the Turing Machine be $M=(Q, \Sigma, \Gamma, \delta, q_0, B, F)$

Suppose $\delta(q, X_i) = (p, Y, R)$ -> next move is rightward
Then
$X_1 X_2 \dots X_{i-1} q X_i X_{+1} \dots X_n \vdash_M X_1 X_2 \dots X_{i-1} Y p X_{i+1} \dots X_n$

Head is moved toward i+1

If i=1, then M moves to the blank to the left of X1->
$X_1 X_2 \dots X_{n-1} q X_n \vdash_M X_1 X_2 \dots X_{n-1} Y p B$

If i=1 and Y=B,
$q X_1 X_2 \dots X_n \vdash_M p X_2 \dots X_n$

Turing Machine: Language

Let the Turing Machine be M=(Q, Σ, Γ, δ, q_0, B, F)
L(M) is the set of strings w in Σ* such that q0 w |-* α p β for some state p in F and any tape string α and β

The set of languages we can accept using a Turing Machine is often called the recursively enumerable languages or RE languages

Assume
A turing machine always halts when it is an accepting state

Turing Machine: Variants

Multiple Tracks
Each track can hold one symbol and the tape alphabet of the TM consists of tuples, with one component for each track

Subroutines
A TM subroutines is a set of states that perform some useful process
It includes a start state and another state that temporarily has no moves and that serves as the return state to pass control to whatever other set of states called to subroutines

Turing Machine: Variants

1. Multiple track Turing Machine:

- A k-tack Turing machine(for some k>0) has k-tracks and one R/W head that reads and writes all of them one by one.
- A k-track Turing Machine can be simulated by a single track Turing machine

Turing Machine: Variants

2. Two-way infinite Tape Turing Machine:

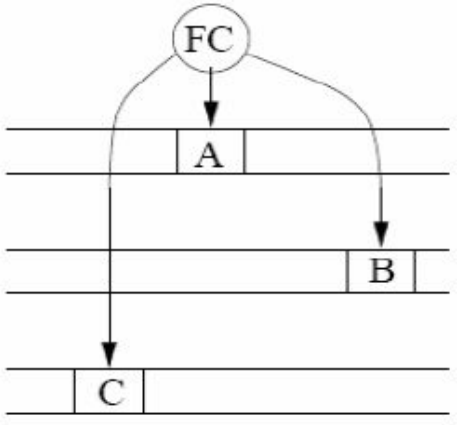
- Infinite tape of two-way infinite tape Turing machine is unbounded in both directions left and right.
- Two-way infinite tape Turing machine can be simulated by one-way infinite Turing machine(standard Turing machine).

Turing Machine: Variants

3. Multi-tape Turing Machine:

- It has multiple tapes and controlled by a single head.
- The Multi-tape Turing machine is different from k-track Turing machine but expressive power is same.
- Multi-tape Turing machine can be simulated by single-tape Turing machine.

Suppose we have a 3-tape TM.



Turing Machine: Variants

4. Multi-tape Multi-head Turing Machine:

- The multi-tape Turing machine has multiple tapes and multiple heads
- Each tape controlled by separate head
- Multi-Tape Multi-head Turing machine can be simulated by standard Turing machine.

Turing Machine: Variants

5. Multi-dimensional Tape Turing Machine:

- It has multi-dimensional tape where head can move any direction that is left, right, up or down.
- Multi dimensional tape Turing machine can be simulated by one-dimensional Turing machine

Turing Machine: Variants

6. Multi-head Turing Machine:

- A multi-head Turing machine contain two or more heads to read the symbols on the same tape.
- In one step all the heads sense the scanned symbols and move or write independently.
- Multi-head Turing machine can be simulated by single head Turing machine.

Turing Machine: Variants

7. Non-deterministic Turing Machine:

- A non-deterministic Turing machine has a single, one way infinite tape.
- For a given state and input symbol has atleast one choice to move (finite number of choices for the next move), each choice several choices of path that it might follow for a given input string.
- A non-deterministic Turing machine is equivalent to deterministic Turing machine.

Turing Machine: Variants

Turing machine with semi-infinite tape
Multistack Machines
Counter Machines
Universal Turing Machine
Turing Machine as Language Enumerator

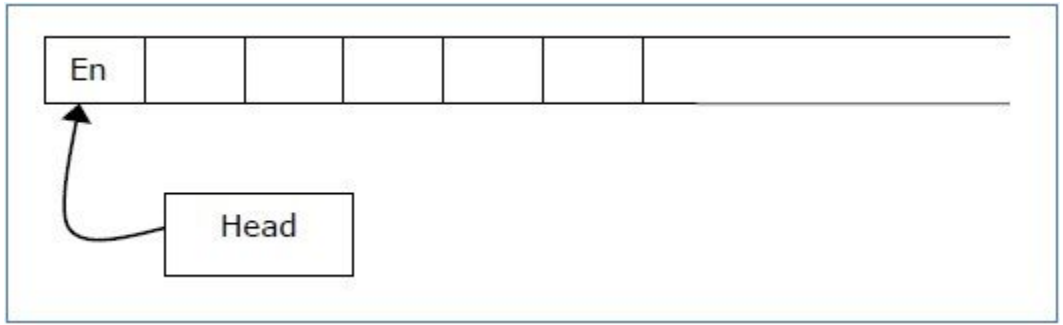
Turing machine with semi-infinite tape

A Turing Machine with a semi-infinite tape has a left end but no right end. The left end is limited with an end marker.

It is a two-track tape −

- Upper track − It represents the cells to the right of the initial head position.
- Lower track − It represents the cells to the left of the initial head position in reverse order.

The infinite length input string is initially written on the tape in contiguous tape cells.



Turing machine with semi-infinite tape

The machine starts from the initial state q_0 and the head scans from the left end marker ‘End’. In each step, it reads the symbol on the tape under its head. It writes a new symbol on that tape cell and then it moves the head either into left or right one tape cell. A transition function determines the actions to be taken.

It has two special states called accept state and reject state. If at any point of time it enters into the accepted state, the input is accepted and if it enters into the reject state, the input is rejected by the TM. In some cases, it continues to run infinitely without being accepted or rejected for some certain input symbols.

Note − Turing machines with semi-infinite tape are equivalent to standard Turing machines

Multistack Machine

A k-stack machine is a deterministic PDA with k stacks

It has a finite control, which is in one of a finite set of states

The input symbol read, which is chosen from the finite input alphabet

The multistack machine can make a move using ϵ input, but to make the machine deterministic, there cannot be a choice of an ϵ-move or a non-ϵ-move in any situation

Counter Machine

1. Same as multistack machine where in place of stack there is a counter

   In one move, the counter machine can change state and add or subtract 1 from any of its counters, independently

2. Restricted multistack machine
   a. Only 2 stack symbols

Counter Machine

Every language accepted by a counter machine is recursively enumerable

Counter machine is a special case of stack machine

A stack machine is a special case of a multitape turing machine which accepts only recursively enumerable languages

Every language accepted by a one-counter machine is a CFL

Every recursively enumerable language is accepted by a two counter machine

Universal Turing Machine

A Turing Machine is the mathematical tool equivalent to a digital computer. It was suggested by the mathematician Turing in the 30s, and has been since then the most widely used model of computation in computability and complexity theory.

The model consists of an input output relation that the machine computes. The input is given in binary form on the machine's tape, and the output consists of the contents of the tape when the machine halts.

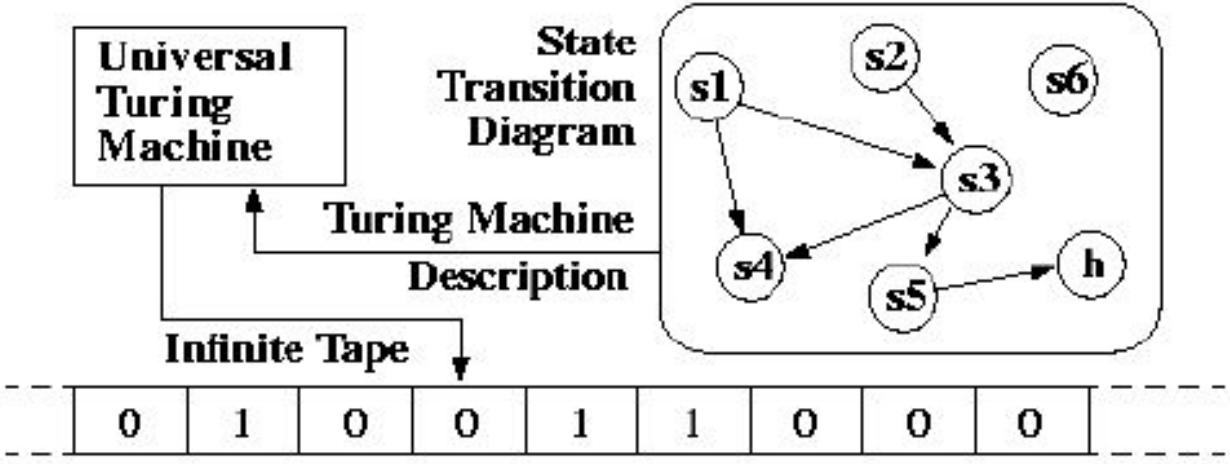
What determines how the contents of the tape change is a finite state machine (or FSM, also called a finite automaton) inside the Turing Machine. The FSM is determined by the number of states it has, and the transitions between them.

At every step, the current state and the character read on the tape determine the next state the FSM will be in, the character that the machine will output on the tape (possibly the one read, leaving the contents unchanged), and which direction the head moves in, left or right.

The problem with Turing Machines is that a different one must be constructed for every new computation to be performed, for every input output relation.

Universal Turing Machine

This is why we introduce the notion of a universal turing machine (UTM), which along with the input on the tape, takes in the description of a machine M. The UTM can go on then to simulate M on the rest of the contents of the input tape. A universal turing machine can thus simulate any other machine.



Turing Machine as Language Enumerator

Produce exhaustive list of strings of the language, progressively longer and longer

Enumeration has no input, and its computation continues indefinitely if the language is infinite as well as when it is finite

For enumeration, a TM of tape k>=2 is used, tape 1 is output tape, which would help all the generated strings, separated #, and often tapes are working tapes

Turing Machine: To Note

- In non-deterministic turing machine, there can be more than one possible move for a given state and tape symbol, but non-deterministic TM does not add any power.
- Every non-deterministic TM can be converted into deterministic TM.
- In multi-tape turing machine, there can be more than one tape and corresponding head pointers, but it does not add any power to turing machine.
- Every multi-tape TM can be converted into single tape TM.

Turing Machine: To Note

If M_N is a nondeterministic Turing Machine, then there is a deterministic Turing Machine MD that $L(M_N)=L(M_D)$

The set of all Turing Machines, although infinite is countable

Recursive and Recursively Enumerable Languages

A language L is said to be recursively enumerable if there exists a Turing Machine that accepts it

A language L on Σ is said to be recursive if there exist a TM M that accepts L and that halts on every w in Σ^+. A language is recursive iff there exists a membership algorithm for it

Recursive and Recursively Enumerable Languages

Let S be an infinite countable set. Then its power set 2^S is not countable.

For nonempty Σ, there exist languages that are not recursively enumerable

Not Recursively Enumerable Languages

There exists a recursively enumerable language whose complement is not recursively enumerable

Recursively Enumerable Languages, Not Recursive

If a language L and its complement L’ are both recursively enumerable, then both languages are recursive, If L is recursive, then L’ is also recursive and consequently both are recursively enumerable

Recursively Enumerable Languages, Not Recursive

If a language L and its complement L’ are both recursively enumerable, then both languages are recursive, If L is recursive, then L’ is also recursive and consequently both are recursively enumerable

There exists a recursively enumerable language that not recursive, that is the family of recursive languages is a proper subset of the family of recursively enumerable languages

Unrestricted Grammar

Any language generated by an unrestricted grammar is recursively enumerable

For every recursively enumerable language L, there exists an unrestricted grammar G, such that L = L(G)

Closure Properties of Decidable and Turing Recognizable Languages

Both decidable and turing recognizable languages are closed under union

Both decidable and turing recognizable languages are closed under concatenation

Both decidable and turing recognizable languages are closed under star closure

Both decidable and turing recognizable languages are closed under intersection

Decidable languages are closed under complementation

Closure Properties of Decidable and Turing Recognizable Languages

Turing recognizable languages are not complementation

A language is decidable if and only of both L and L’ are Turing recognizable

Languages not Recursively Enumerable

A language L is recursively enumerable (RE), if L=L(M) for some TM M

Our long-range goal is to prove undecidable the language consisting of pairs (M, w) such that:

1. M is a Turing machine (suitably coded in binary) with input alphabet {0,1}
2. w is a string of 0's and 1's
3. M accepts input w

The Diagonalization Language

1. Reducing One Problem to Another

- Suppose we have an algorithm A to transform instances of one problem P_1 to instances of another problem P_2 in such a way that a string w is in P_1 if and only if the transformed string $A(w)$ is in P_2. What this implies is that we can solve instances of problem P_1 by converting them to instances of problem P_2 and then using the solution to problem P_2 to answer the original question. If we can do this, then we will say that A is a *reduction* of P_1 to P_2.
- Note that a reduction from P_1 to P_2 must turn every instance of P_1 with a yes answer to an instance of P_2 with a yes answer, and every instance of P_1 with a no answer to an instance of P_2 with a no answer.
- We will frequently use this technique to show that problem P_2 is as hard as problem P_1.
- The direction of the reduction is important.
    - For example, if there is a reduction from P_1 to P_2 and if P_1 is not recursive, then P_2 cannot be recursive. (Why?)
    - Similarly, if there is a reduction from P_1 to P_2 and if P_1 is not recursively enumerable, then P_2 cannot be recursively enumerable. (Why?)

The Diagonalization Language

2. Enumerating the Binary Strings

- In many proofs involving Turing machines, we need to enumerate the binary strings and encode Turing machines so that we can refer to the *i*th binary string as w_i and the *i*th Turing machine as M_i.
- The binary strings are easy to enumerate. If *w* is a binary string, we shall treat 1*w* as the binary integer *i* so we can call *w* the *i*th string.
- Using this encoding, the empty string is the first string, 0 the second, 1 the third, 00 the fourth, 01 the fifth, and so on. Henceforth we will refer to the *i*th string as w_i.

The Diagonalization Language

3. Codes for Turing machines

- We will now define a binary code for all Turing machines with the input alphabet {0, 1} so that each Turing machine can be represented by a binary string. This will allow us to talk about the *i*th Turing machine as M_i. We will adopt the following conventions:
  - We assume the states are q_1, q_2, . . . , q_r for some r. We assume q_1 will always be the start state. We assume q_2 will be the only accepting state. We need only one accepting state if we assume the Turing machine halts whenever it enters an accepting state.
  - We assume the tape symbols are X_1, X_2, . . . , X_s for some s. We assume X_1 is 0, X_2 is 1, and X_3 is the blank.
  - We assign integers D_1 and D_2 to the tape head directions left and right.
  - Now we can encode a transition rule $\delta(q_i, X_j) = (q_k, X_l, D_m)$ as a binary string C of the form
  - $0^i10^j10^k\ 10^l10^m$Suppose there are n transition rules. The binary code for the entire Turing machine will be the concatenation of the codes for all of the transitions (in some order) separated by pairs of 1's:
  - $C_1$11$C_2$11 … C_{n-1}11C_nNote that there can be many encodings for the same Turing machine.
- We can encode a pair (M_i, w) consisting of a Turing machine and a string by appending 111 to the encoding of the Turing machine and then appending the string w.

The Diagonalization Language

4. The Diagonalization Language L_d is not Recursively Enumerable

- We define L_d, the diagonalization language, as follows:
  - Let w_1, w_2, w_3, . . . be an enumeration of all binary strings.
  - Let M_1, M_2, M_3, . . . be an enumeration of all Turing machines.
  - Let $L_d = \{ w_i \mid w_i \text{ is not in } L(M_i) \}$.
- Theorem: L_d is not a recursively enumerable language.
  Proof:
  - Suppose $L_d = L(M_i)$ for some TM M_i.
  - This gives rise to a contradiction. Consider what M_i will do on an input string w_i.
    - If M_i accepts w_i, then by definition w_i cannot be in L_d.
    - If M_i does not accepts w_i, then by definition w_i is in L_d.
  - Since w_i can neither be in L_d nor not be in L_d, we must conclude there is no Turing machine that can define L_d.

The Diagonalization Language

5. Complements of Recursive and Recursively Enumerable Languages

- A recursive language is one that is accepted by a TM that halts on all inputs.
- The complement of a recursive language is recursive.
- If a language *L* and its complement are RE, then *L* is recursive.
- A language can be RE but its complement need not be RE.

The diagonalization language is not a recursively enumerable language. That is there is no turing machine accepts this