INT201 Final

Lecture 1 Overview

Languages

- Several symbols and defination
  - Alphabet - a finite nonempty set Σ of symbols
  - Strings(word) - a finite sequence of symbols from the alphabet
  - Empty string(word) - the string with no symbols, donated as ϵ

$$|\epsilon| = 0$$

$$\epsilon w = w\epsilon = w$$

  - Length of a string - the number of symbols in the string, denoted as | ϵ |

$$|w_1 w_2| = |w_1| + |w_2|$$

  - Reverse of a string - obtained by writing the symbols in reverse order, denoted as w^R

$$w = a_1 a_2 \dots a_n$$

$$w^R = a_n \dots a_2 a_1$$

  - A palindrome is a string w satisfying $w = w^R$
  - The concatenation of 2 strings is obtained by appending the symbols to the right end

$$w = a_1 a_2 \dots a_n$$

$$v = b_1 b_2 \dots b_n$$

$$wv = a_1 a_2 \dots a_n b_1 b_2 \dots b_n$$

  - If $u, v, w$are strings and $w = uv$ then u is a prefix of w and v is a suffix of w. A proper prefix of w is a prefix that is not equal to ϵ of w
  - w^n denotes the concatenation of n copies of w
  - Σ^*(star closure of Σ) denotes the set of all strings(words) over Σ
  - Σ^+(positive closure of Σ) denotes the set of non-empty strings(words) over Σ
  - Both of Σ^* and Σ^+ are infinite set
- A **formal language** is a set of strings(sequence of symbols) constructed from a finite alphabet, according to specific set of precise, mathematical rules. The key idea is that it is defined by its form, no its meaning
- A formal language L over alphabet Σ is a subset of Σ^*

- We can express new languages in terms of other languages using concatenation and closure:

$$L_1L_2 = \{w_1w_2 : w_1 \in L_1 \; and \; w_2 \in L_2\}$$

$$L^* = \{w_1w_2 \ldots w_n : n \geq 0 \; and \; w_1, w_2, \ldots w_n \in L\}$$

Grammars

- Arbitrarily long sentences can be generated
- Semantics can be given with reference to grammar, e.g. logical conjunction of subsentences formed by word "and"
- parse Tree: identify relationships between tokens
    - parse tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar(CFG)

Mathematical preliminaries

Complexity theory

The main question asked in this area is "what makes some problems computationally hard and other problems easy"
如果一个问题是高效可解的，则称之为“容易”

Computation theory

Central question in Computability Theory: classify problems as being solvable or unsolvable

Automata theory

- Central question in Automata Theory: Do these models have the same power, or can one model solve more problems than others
    - FA(DFA / NFA) These are used in text processing, compilers, and hardware design
    - CFG These are used to define programming languages and in artificial Intelligence
    - Turing machines These form a simple abstract model of a "real" computer, such as your PC at home

Lecture 2 Deterministic Finite Automata

- DFA is a finite-state machine that accepts or rejects a given string of symbols, by running through a state sequence uniquely determined by the string
- DFA can be used to describe:
    - Any finite set of strings

    - Various infinite sets of strings
        - strings having exactly 2 occurrences of the letter a
        - strings having more than 6 letters
        - strings in which letter b never comes before letter a
- DFA can't be used in describe certain languages such as:
    - the set of strings containing more a's than b's
    - well-formed arithmetic expressions, if there is no limit on nesting of parentheses
    - all words that remain the same if you read them back to front

Definition

- A DFA is defined as a 5-tuple

$$M = (Q, \Sigma, \delta, q, F)$$

  Where

    1. Q is a finite set of states
    2. Σ is a finite set of symbols, called the alphabet of the automation
    3. δ: $Q \times \Sigma \to Q$ is a function, called the transition function
    4. $q \in Q$ is called the initial state
    5. $F \subseteq Q$ is a set of accepting/terminal states
- Transition function δ tells you how state should change when an additional letter is read by the DFA
    - If initially the state is i and if the input word is $w = a_1 a_2 \dots a_n$ then, as each letter is read, the state changes and we get $q_1, q_2 \dots q_n$ defined by:

$$q_1 = \delta(i, a_1)$$

$$q_2 = \delta(q_1, a_2)$$

$$q_1 = \delta(q_2, a_3)$$

$$\vdots$$

$$q_n = \delta(q_{n-1}, a_n)$$

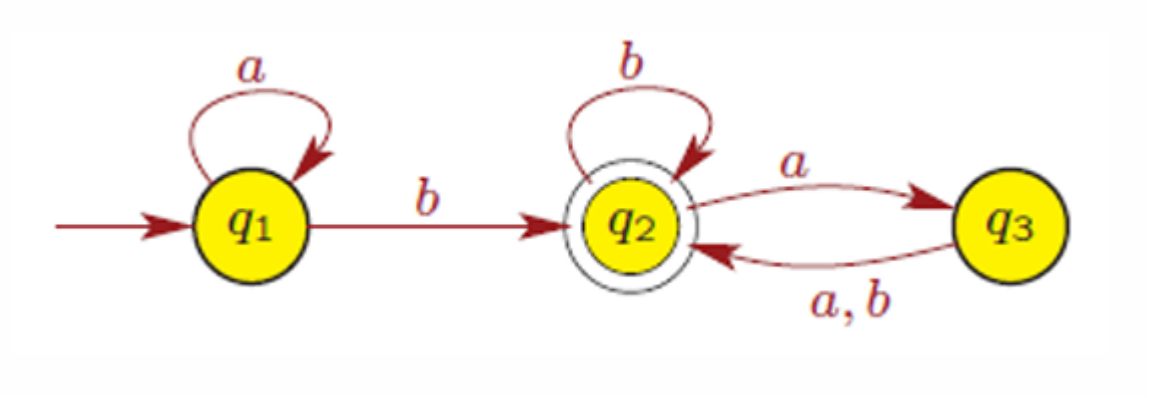
    - we can extend the definition of the transition function δ so that it tells us which state we reach after $\underline{a\ word}$ has been scanned
        - extend the map $\delta : Q \times \Sigma \to Q$ to $\delta^* : Q \times \Sigma^* \to Q$ by defining:

$$\delta^*(q, \epsilon) = q\ for\ all\ q \in Q$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a)\ for\ all\ q \in Q; w \in \Sigma^*; a \in \Sigma$$

- A DFA $M = (Q, \Sigma, \delta, q, F)$ is often depicted as a directed graph G_M(called transition graph) has exactly $|Q|$ vertices, each labeled with different $q_i \in Q$. For each transition

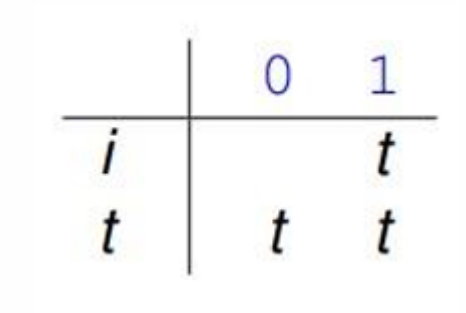
function $\delta(q_i, a) = q_j$, the graph has edges (q_i, q_j) labeled(a), (b), and (a, b). The vertex associated with q_1 is called the initial vertex, while those labeled with q_f are the final vertices.



- Let $M = (Q, \Sigma, \delta, q, F)$ be a finite automaton and let $w = w_0 w_1 \dots w_n$ be a string over. Define the sequence $q_0, q_1, \dots q_n$ of states, in the following way
  - $q_0 = q_1$
  - $q_{i+1} = \delta(q_i, W_{i+1})$, for i=0, 1, ... , n-1.
  1. If $q_n \in F$, then we say that M accepts w.
  2. If $q_n \notin F$, then we say that M rejects w.

Symbolic description of the example DFA

- If δ is a partial function(not defined for some state / letter pairs), then the DFA rejects an input if it ever encounters such a pair
- This convention often simplifies the definition of a DFA. We could use transition table



	0	1
i		*t*
t	*t*	*t*

Language defined by DFA

- Suppose we have a DFA M. A word $w \in \Segma^*$is said to be accepted or recognized by M if $\delta^*(q_0, w) \in F$, otherwise it is said to be rejected. The set of all words accepted by M is called the language accepted by M and will be denoted by $L(M)$. Thus

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$$

  Any finite language is accepted by some DFA
- A language A is called regular, if there exists a finite automaton M such that $A = L(M)$
- Regular operations on languages
  - let A and B be two languages over the same aphabet

- The union of A and B is defined as:

$$A \cup B = \{w : w \in A \text{ or } w \in B\}$$

- The concatenation of A and B is defined as

$$AB = \{ww' : w \in A \text{ and } w' \in B\}$$

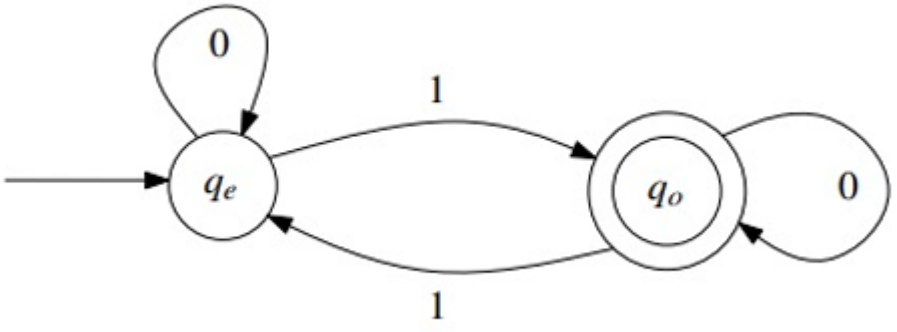
- The star of A is defined as:

$$A^* = \{u_1 u_2 \dots u_k : k \geq 0 \text{ and } u_i \in A \text{ for all } i = 1, 2 \dots k\}$$

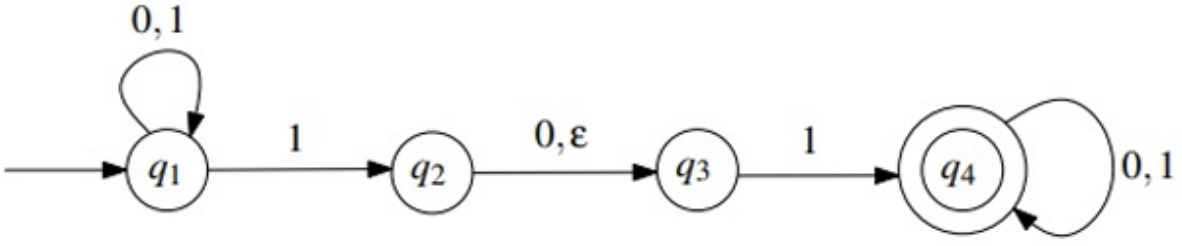
- Theorem
  The set of regular languages is closed under the union operation, i.e., if A and B are regular languages over the same alphabet Σ, then A $\cup$ B is also a regular language

Lecture 3 Nondeterministic Finite Automata

- A finite automata is deterministic, if the next state the machine goes on any given symbol is uniquely determined



  - DFA has exactly one transition leaving each state for each symbol
- A finite automata is nondeterministic, if the machine allows for seveal or no choices to exist for the next state on a given symbol



  - For a state q and symbol $s \in \Sigma$, NFA can have
    - Multiple edges leaving q labelled with the symbol s;
    - No edge leaving q labelled with symbol s;
    - Edge leaving q labelled with ϵ(without reading any symbol)

> - The NFA accepts the input string, if any copy ends in an accept state after reading entire string, otherwise, NFA rejects.

Formal Definition of NFA

For any alphabet Σ, we define Σ_ϵ to be the set

$$\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$$

Recall the notion of a power set: For any set Q, the power set of Q, denoted by $P(Q)$, is the set of all subsets of Q

$$P(Q) = \{R : R \subseteq Q\}$$

A NFA is a 5-tuple $M = (Q, \Sigma, \delta, q, F)$
where

1. Q is a finite set of states
2. Σ is a finite set of symbols, called the alphabet of the automation
3. $\delta : Q \times \Sigma_\epsilon \to P(Q)$ is a function, called the transition function
4. $q \in Q$ is called the initial state
5. $F \subseteq Q$ is a set of accepting / terminal states
   Let $M = (Q, \Sigma, \delta, q, F)$ be an NFA, and let $w \in \Sigma^*$. We say that M accepts w, if q can be written as $w = y_1 y_2 \dots y_m$ where $y_i \in \Sigma_\epsilon$ for all i with $1 \leq i \leq m$, and there exists a sequence of states $r_1, r_2, \dots r_m$ in Q, such that:
   > - $r_0 = q$
   > - $r_{i+1} \in \theta(r_i, y_{i+1})$ for $i = 0, 1, 2, \dots m-1$
   > - $r_m \in F$
   >
   > Otherwise, we say that M rejects the string w

Extend the map ϵ to a map $Q \times Sigma^* \to P(Q)$ by defining:

$$\delta(q, \epsilon) = \{q\}\ for\ all\ q \in Q$$

$$\delta(q, wa) = U_{p \in \delta(q,w)} \delta(p, a) for\ all\ q \in Q; w \in \Sigma^*; a \in \Sigma$$

Thus $\delta(q, w)$ is the set of all possible states that can arise when the input w is received in the state q. w is accepted provided that $\delta(q, w)$ contains an accepting state

Difference between DFA and NFA

- DFA has transition function $\delta : Q \times \Sigma \to Q$
- NFA has transition function $\delta : Q \times \Sigma_\epsilon \to P(Q)$
- Returns a set of states rather than a single state

- Allows for ϵ- transition because $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$
- Note that every DFA is also an NFA

Notation: accepting / rejecting paths

Suppose, in a DFA, we can get from state p to state q via transitions labelled by letters of a word w. Then we say that the states p and q are connected by a path with label w.
If $w = abc$ and the 2 intermediate states are r_1 and r_2 we could write this as:

$$p \xrightarrow{a} r_1 \xrightarrow{b} r_2 \xrightarrow{c} q$$

In NFA, if $\delta(p, a) = (p, r)$, we could write:

$$\{p\} \xrightarrow{a} \{q, r\}$$

and this would be an accepting path if any state on RHS is an accepting state, otherwise it would be rejecting path

Language accepted by NFA

Let $M = (Q, \Sigma, \delta, q, F)$ be an NFA. The language $L(M)$ accepted by M is defined as

$$L(M) = \{w \in \Sigma^* : M \; accepts \; w\}$$

Equivalence of DFAs and NFAs

Two machines(of any type) are equivalent if they recognize the same language
DFA is a restricted form (受限形式)of NFA

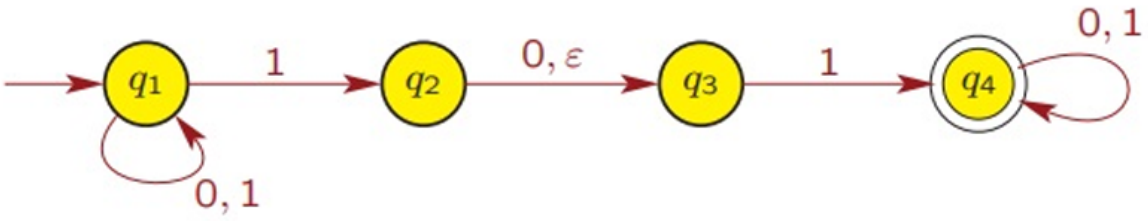
- Every NFA has an equivalent DFA
- We can convert an arbitrary NFA to an NFA that accepts the same. language
- DFA has the same power as NFA

DFA to NFA
The formal conversion of a DFA to NFA is done as follows: Let $M = (Q, \Sigma, \delta, q, F)$ be a DFA. Recall that δ is a function $\delta : Q \times \Sigma \to Q$. We define the function $\delta' : Q \times \Sigma^* \to P(Q)$ as follows. For any $r \in Q$ and for any $a \in \Sigma_\epsilon$

$$\delta'(r, a) = \begin{cases} \{\delta(r, a)\} & \text{if } a \neq \varepsilon \\ \varnothing & \text{if } a = \varepsilon \end{cases}$$

Then $N = (Q, \Sigma, \delta', q, F)$ is an NFA whose behavior the same as that of the DFA M; the easiest way to see this is by observing that the state diagrams of M and N are equal. Therefore, we have $L(M) = L(N)$

NFA to DFA



The original NFA:
$Q = q_1, q_2, q_3, q_4$
$\Sigma = \{0, 1\}$
δ:

- $\delta(q_1, 0) = q_1$, $\delta(q_1, 1) = \{q_1, q_2\}$
- $\delta(q_2, 0) = q_3$
- $\delta(q_3, 1) = q_4$
- $\delta(q_4, 0) = q_4$, $\delta(q_4, 1) = q_4$

$F = \{q_4\}$
Now define the state of DFA, the state of DFA is the subset of NFA
$q_0 = q_1$
compute the transition

- Start from $\{q_1\}$
  $\delta_{DFA}(\{q_1\}, 0) = \{q_1\}$,
  $\delta_{DFA}(\{q_1\}, 1) = \{q_1, q_2\}$
- Start from $\{q_1, q_2\}$
  $\delta_{DFA}(\{q_1, q_2\}, 0) = \{q_1, q_3\}$
  $\delta_{DFA}(\{q_1, q_2\}, 1) = \{q_1, q_2\}$
- Start from $\{q_1, q_3\}$
  $\delta_{DFA}(\{q_1, q_3\}, 0) = \{q_1\}$
  $\delta_{DFA}(\{q_1, q_3\}, 1) = \{q_1, q_2, q_4\}$
- Start from $\{q_1, q_2, q_4\}$
  $\delta_{DFA}(\{q_1, q_2, q_4\}, 0) = \{q_1, q_3, q_4\}$
  $\delta_{DFA}(\{q_1, q_2, q_4\}, 1) = \{q_1, q_2, q_4\}$
- Start from $\{q_1, q_3, q_4\}$
  $\delta_{DFA}(\{q_1, q_3, q_4\}, 0) = \{q_1, q_4\}$
  $\delta_{DFA}(\{q_1, q_3, q_4\}, 1) = \{q_1, q_2, q_4\}$
- Start from $\{q_1, q_4\}$
  $\delta_{DFA}(\{q_1, q_4\}, 0) = \{q_1, q_4\}$
  $\delta_{DFA}(\{q_1, q_4\}, 1) = \{q_1, q_2, q_4\}$

- Start from $\{q_4\}$
  $\delta_{DFA}(\{q_4\}, 0) = \{q_4\}$
  $\delta_{DFA}(\{q_4\}, 1) = \{q_4\}$

$F_{DFA} = \{\{q_4\}, \{q_1, q_4\}, \{q_1, q_2, q_4\}, \{q_1, q_3, q_4\}\}\}$

Lecture 4 Regular Language I

- Definition
  - Previous: A language is regular if it is recognized by some DFA
  - Now: A language is regular if and only if some NFA recognizes it
- Closed under operations
  - A collection S of objects is closed under operation f is applying f to member of S always returns an object still in S
  - Regular languages are indeed closed under the regular operations(union, concatenation, Star)

Regular Languages Closed Under Union

- Proof:

  $$L_1 \, and \, L_2 \, are \, both \, regular \, languages, \, then \, L_1 \cup L_2 \, is \, also \, regular \, language$$

  - Define $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, let $L(N_1) = L_1$; $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, let $L(N_2) = L_2$, and $Q_1 \cap Q_2 = \emptyset$
  - Construct a new machine N : $(Q, \Sigma, \delta, q_0, F)$
    - $Q = Q_1 \cup Q_2 \cup \{q_0\}$
    - $F = F_1 \cup F_2$
    - start state: q_0
    - δ:
      - new $\epsilon - transition$: $\delta(q_0, \epsilon) = \{q_1, q_2\}$
      - internal transition: keep the original transition in $N_1 \, and \, N_2$

      $$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \emptyset & q = q_0 \text{ and } a \neq \epsilon \end{cases}$$

  - Verify the correctness
    - If $w \in L_1$: the machine N can start from q_0, and $\epsilon - transition$ to q_1, and then following the path of N_1 to deal with w, finally stop at a state of F_1. Because of $F_1 \subseteq F$, N accepts w.
    - If $w \in L_2$: the machine N can start from q_0, and $\epsilon - transition$ to q_2, and then following the path of N_2 to deal with w, finally stop at a state of F_2. Because of $F_2 \subseteq F$, N accepts w.

    - So the language accept by N is all strings in L_1 and L_2; $L(N) = L_1 \cup L_2$
- We construct a NFA which recognizes $L_1 \cup L_2$, this language is also regular language

Regular Languages Closed Under Concatenation

- Proof:

$$L_1 \; and \; L_2 \; are \; both \; regular \; languages, \; then \; L_1L_2 \; is \; also \; regular \; language$$

    - Define $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, let $L(N_1) = L_1$; $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, let $L(N_2) = L_2$, and $Q_1 \cap Q_2 = \emptyset$
    - Construct a new machine N : $(Q, \Sigma, \delta, q_0, F)$
        - $Q = Q_1 \cup Q_2$
        - $F = F_2$
        - start state: $q_0 = q_1$
        - δ:
            - new $\epsilon - transition$: $\delta(q, \epsilon) = \{q_2\}, q \in F_1$
            - internal transition: keep the original transition in $N_1 \; and \; N_2$

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \epsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ and } a = \epsilon \\ \delta_2(q,a) & q \in Q_2 \end{cases}$$

    - Verify the correctness
        - If the input string w can be divided as $w = xy$, where $x \in L_1$ and $y \in L_2$
            - the machine start from q_1, when it reads x and it will stop in a state q in N_1, $q \in F_1$, by using $\epsilon - transition$, the machine jump to q_2, and q_2 reads y, finally it will stop in a state in N_2(which belongs to F_2), $F_2 \subseteq F$, so N accepts w
        - So the language accept by N is all strings in L_1L_2; $L(N) = L_1L_2$
- We construct a NFA which recognizes L_1L_2, this language is also regular language

Regular Languages Closed Under Kleene star

- The star of A is defined as:

$$A^* = \{u_1u_2 \dots u_k : k \geq 0 \; and \; u_i \in A \; for \; all \; i = 1, 2, \dots k\}$$

- Proof

$$L \; is \; regular \; languages, \; then \; L^* \; is \; also \; regular \; language$$

    - Define $N = (Q_1, \Sigma, \delta_1, q_1, F_1)$, let $L(N) = L$;

- Construct a new machine $N:(Q,\Sigma,\delta,q_0,F)$
  - $Q=Q_1\cup\{q_0\}$
  - $F=F_1\cup\{q_0\}$
  - start state: q_0
  - δ:
    - new $\epsilon-transition$: $\delta(q,\epsilon)=\{q_1\}$
    - internal transition: keep the original transition in $N_1\ and\ N_2$

$$\delta(q,a)=\begin{cases}\delta_1(q,a) & q\in Q_1 \text{ and } q\notin F_1\\ \delta_1(q,a) & q\in F_1 \text{ and } a\neq\epsilon\\ \delta_1(q,a)\cup\{q_1\} & q\in F_1 \text{ and } a=\epsilon\\ \{q_1\} & q=q_0 \text{ and } a=\epsilon\\ \emptyset & q=q_0 \text{ and } a\neq\epsilon\end{cases}$$

      - line 3 的解释：一旦你在N_1中读完了一段属于A的字符串，你可以选择停下来，也可以选择利用$\epsilon-transition$来回到起始位置
- The N can accept empty set, it can also accept the L^*, so $L(N)=L^*$, the language is closed under Kleene star

Regular Language Closed Under Complement

- 注：由于DFA是确定性而安全的（对于每一个输入字符都有且仅有一个确定的状态转移），我们只需要把接收状态和非接受状态互换即可，如果使用NFA证明，NFA对于拒绝的定义是不存在接受路径，简单翻转状态并不等同于逻辑取反
- Proof

$$If\ language\ L\ is\ regular\ language, the\ complement\ \bar{L}\ is\ also\ regular\ language$$

  - Suppose L is a regular language, and a DFA $M=(Q,\Sigma,\epsilon,\backslash q_0,F)$ recognizes it
  - Construct a new DFA $M'=(Q,\Sigma,\epsilon,\backslash q_0,F')$
    - $F'=Q-F$: it accepts all strings rejected before, and rejects all strings accepted before
  - Verify the correctness
    - When M reads w and stop at the state q.
    - If $w\in L$, $q\in F$, $q\notin F'$, so M' rejects w
    - If $w\notin L$, $q\notin F$, $q\in F'$, so M' accepts w
    - So M' recognizes $\bar{L}$
  - So Regular language closed under complement

Regular Language Closed Under Intersection

- Proof:

- By De Morgan's Law

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

- So regular language is also closed under intersection

Regular Expressions

- Regular expressions are means to describe certain languages
- Formal definition of regular expressions
  - Let Σ be a non-empty alphabet
    1. ϵ is a regular expression 接受空串
    2. $\emptyset$ is a regular expression 什么都不接受
    3. For each $a \in \Sigma$, a is a regular expression 接受单个字符的机器
    4. If R_1 and R_2 are regular expressions, then $R_1 \cup R_2$ is a regular expression
    5. If R_1 and R_2 are regular expressions, then $R_1 R_2$ is a regular expression
    6. If R is regular expressions, then R^* is a regular expression

Lecture 5 Regular Languages II

Kleene's Theorem

- Let L be a language. Then L is regular if and only if there exists a regular expression that describes L
  - If a language is describe by a regular expression, then it is regular
  - If a language is regular, then it has a regular expression

The language described by a regular expression is a regular language

- Proof: Convert a regular expression R into a NFA M
  - 1st case. If $R = \epsilon$, then $L(R) = \{\epsilon\}$. The NFA is $M = \{\{q\}, \Sigma, \delta, q, \{q\}\}$ where:(什么都接受 一开始就接受)

$$\delta(q, a) = \emptyset \ for\ all\ a \in \Sigma_\epsilon$$

  - 2nd case. If $R = \emptyset$, then $L(R) = \emptyset$. The NFA is $M = \{\{q\}, \Sigma, \delta, q, \emptyset\}$ where:(什么都不接受)

$$\delta(q, a) = \emptyset \ for\ all\ a \in \Sigma_\epsilon$$

  - 3rd case.If $R = a$ for $a \in \Sigma$, then $L(R) = \{a\}$. The NFA is $M = \{\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\}\}$ where:

$$\delta(q_1, a) = \{q_2\}$$

$$\delta(q_1, b) = \emptyset \ for\ all\ b \in \Sigma_\epsilon \backslash \{a\}$$

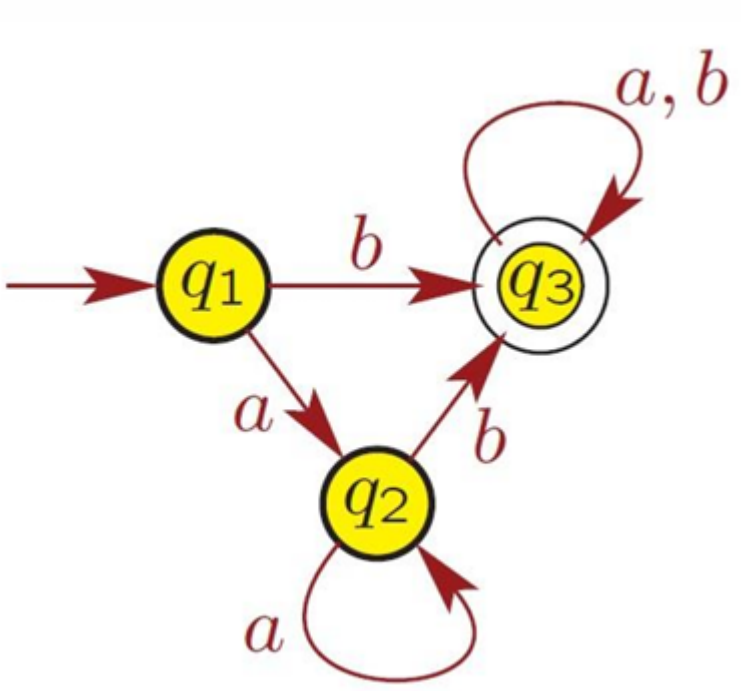
$$\delta(q_2, b) = \emptyset \ for\ all\ b \in \Sigma_\epsilon$$

        - 逻辑：如果你在起点读到了除了a以外的任何字符/或者epsilon转移 是不会接受的
- 4th case(union). If $R = (R_1 \cup R_2)$ and
    - $L(R_1)$ has NFA M_1
    - $L(R_2)$ has NFA M_2
    - Then $L(R) = L(R_1) \cup L(R_2)$ has NFA to describe according to Regular Languages Closed Under Union
- 5th case(concatenation). If $R = (R_1 R_2)$ and
    - $L(R_1)$ has NFA M_1
    - $L(R_2)$ has NFA M_2
    - Then $L(R) = L(R_1)L(R_2)$ has NFA to describe according to Regular Languages Closed Under Concatenation
- 6th case(Kleene star).
    - If $R = (R_1)^*$ and $L(R_1)$ has NFA N
    - Then $L(R) = (L(R_1))^*$ has NFA to describe according to Regular Languages Closed Under Kleene star

A regular language has a regular expression

- Convert DFA into regular expression
    - Every DFA can be converted to a regular expression that describes the language $L(M)$
    - Generalized NFA (GNFA)
        - A GNFA can be defined as a 5-tuple, $(Q, \Sigma, \delta, \{s\}, \{t\})$ consisting of
            - a finite set of states Q
            - a finite set called the alphabet Σ
            - a transition function $(\delta : (Q \backslash \{t\} \times (Q \backslash \{s\}) \to Q)$
            - a start state ($s \in Q$)
            - an accept state ($t \in Q$)
            - where R is the collection of all regular expressions over the alphabet Σ
        - 普通NFA/DFA，箭头标签上只能是一个字符，而GNFA标签可以是整个正则表达式；如果我想从状态A走到状态B，我必须读入一串符合该正则表达式的字符，发明它就是为了解决从复杂的DFA直接写出正则表达式的问题，通过一种“拆迁算法”，一步步拔掉中间状态，每拔掉一个状态，就把原来的路径信息压缩到剩下的边上，直到最后只剩两个状态，那条边上的正则表达式就是最终答案

- Iterative procedure for converting a DFA $M = (Q, \Sigma, \delta, q, F)$ into a regular expression
  1. Convert DFA $M = (Q, \Sigma, \delta, q, F)$ into equivalent GFNA G
     - introduce new start state s
     - introduce new start state t
     - Change edge labels into regular expressions
       e.g., "a, b" becomes $a \cup b$
  2. Iteratively eliminate a state from GNFA G until only 2 states remaining: start and accept
     - need to take into account all possible previous oaths
     - Never eliminate new start state s or new accept state t.
- example:



  - 1st step: DFA $\rightarrow$ GNFA
    - $s \xrightarrow{\epsilon} q_1$
    - $q_1 \xrightarrow{a} q_2$
    - $q_1 \xrightarrow{b} q_3$
    - $q_2 \xrightarrow{a} q_2$
    - $q_2 \xrightarrow{b} q_3$
    - $q_3 \xrightarrow{a \cup b} q_3$
    - $q_3 \xrightarrow{\epsilon} t$
  - 2nd step: Eliminate states 移除中间状态
    - eliminate q_2: we found that $q_1 \rightarrow q_2 \rightarrow q_3$
      - $q_1 \xrightarrow{aa^*b} q_3$

- update $q_1 \to q_3$
  - new path: $b \cup aa^*b$, b is the origin.
- now only the state s, q_1, q_3, t

- eliminate q_1: we found that $s \to q_1 \to q_3$
  - $s \xrightarrow{\epsilon} q_1$
  - $q_1 \xrightarrow{a^*b} q_3$
  - update $s \to q_3$
  - now only state s, q_3, t
- eliminate q_3: we found that $s \to q_3 \to t$
  - $s \xrightarrow{a^*b} q_3$
  - $q_3 \xrightarrow{a \cup b} q_3$
  - $q_3 \xrightarrow{\epsilon} t$

The final regular expression:

$$a^*b(a \cup b)^*$$

Pumping Lemma for Regular languages

A tool that can be used to prove that certain languages are not regular. This theorem states that all regular languages have a special languages have a special property.

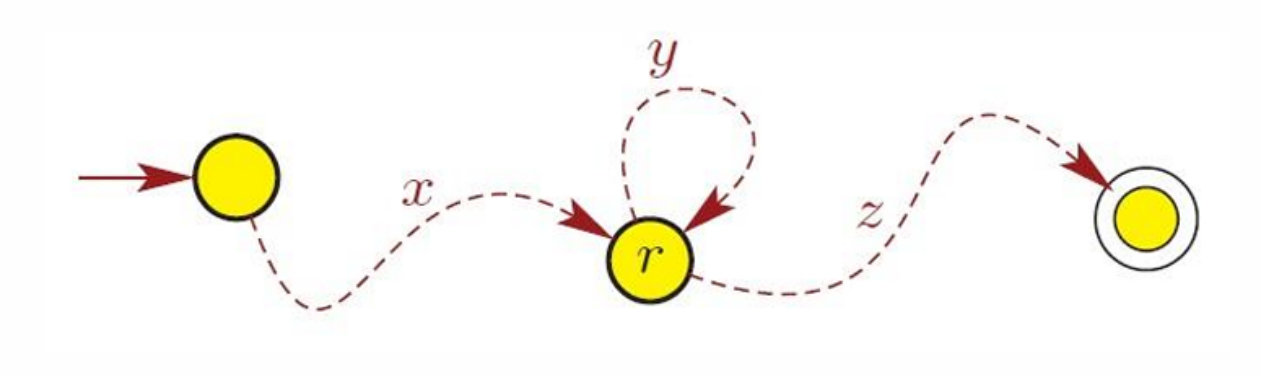
This property states that all strings in the language can be "pumped" if they are at least as long as a certain special value, called the **pumping length**

- If a language L is regular, it always satisfies pumping lemma, if there exists at least one string made from pumping which is not in L, then L is surely not regular
- The opposite may not be true. If pumping lemma holds, it doesn't mean that the language is regular

Consider

- language A with DFA M having p states(where p is number of states in DFA)
- strings $s \in A$ with $|s| \geq p$

- When processing s on M, guaranteed to visit some state twice



  - let r be first state visited twice
  - Using state r, can divide s as $s = xyz$
    - x are symbols read until first visit to r
    - y are symbols read from first to second visit to r
    - z are symbols read from second visit to r to end of s
- Because y corresponds to starting in r and returning to r

$$xy^iz \in A \; for \; each \; i \geq 1$$

- Also, note $xy^0z = xz \in A$, so

$$xy^iz \in A \; for \; each \; i \geq 0$$

- $|y| > 0$ because
  - y corresponds to starting in r and coming back
  - this consumes at least one symbol(because DFA), so y can't be empty
- $|xy| \leq p$, where p is number of states in DFA, because
  - xy are symbols read up to second visit to r
  - Because r is the first state visited twice, all states visits before second to r are unique
  - So just before visiting r for second time, DFA visited at most p states, which corresponds to reading at most $p - 1$ symbols
  - The second visit to r, which is after reading 1 more symbol, corresponds to reading at most p symbols

Definition

- Let A be a regular language. Then there exists an integer $p \geq 1$, called the pumping length, such that the following holds: Every string s in A, with $|s| \geq p$, can be written as $s = xyz$, such that
  - $y \neq \epsilon$
  - $|xy| \leq p$, and
  - for all $i \geq 0$, $xy^iz \in A$

- example: Language $L = \{o^n1^n | n \geq 0\}$is Non-regular
    - Proof
        - Assume $L = \{o^n1^n | n \geq 0\}$ is regular, it must satisfy the pumping lemma
        - choosing string s, let $s = 0^p1^p$, $|s| = 2p \geq p$ satisfying the request
        - Decomposition to $s = xyz$ which satisfying three conditions
            1. $y \neq \epsilon$
            2. $|xy| \leq p$, and
            3. for all $i \geq 0$, $xy^iz \in A$
        - Make contradiction
            - according to pumping lemma, the first xy must all be 0
            - let $i = 2$, the new string $xy^2z = xyyz$
            - the number of 0s becomes $p + k$, but the number of 1s still equal p
            - So it is contradict with the pumping lemma
        - Language $L = \{o^n1^n | n \geq 0\}$is Non-regular

Lecture 6 Context-Free Languages I

- Finite machine accept precisely the strings in the language
  perform a computation to determine whether a specific string is in the language
- Regular expressions describe precisely the strings in the language
  describe the general shape of all strings in the language
- Context-free grammar(CFG) is an entirely different formalism for defining a class of languages
  give a procedure for listing off all strings in the language

Definition

A context-free grammar is a 4-tuple $G = (V, \Sigma, R, S)$, where

1. V is a finite set, whose elements are called variables
2. Σ is a finite set, whose elements are called terminals
3. $V \cap \Sigma = \emptyset$
4. S is an element of V; it is called the start variable
5. R is a finite set, whose elements are called rules. Each rule has the form $A \to w$, where $A \in V and\, w \in (V \cup \Sigma)^*$

Deriving strings and languages using CFG

$\Rightarrow$: yield
Let $G = (V, \Sigma, R, S)$ be a context free grammar with

- $A \in V$
- $u, v, w \in (V \cup \Sigma)^*$
- $A \to w$ is a rule of the grammar

The string uwv can be derived in one step from the string uAv, written as

$$uAv \Rightarrow uwv$$

$\overset{*}{\Rightarrow}$: derive
Let $G = (V, \Sigma, R, S)$ be a context free grammar with

- $u, v \in (V \cup \Sigma)^*$

The string v can be derived from the string u, written as $u \overset{*}{\Rightarrow} v$, if one of the following conditions holds

1. $u = v$
2. there exists an integer $k \geq 2$ and a sequence $u_1, u_2, \dots u_k$ of strings in $(V \cup \Sigma)^*$, such that
   - $u = u_1$,
   - $v = u_k, and\, u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k$

Regular language are context-free

Theorem
Let Σ be an alphabet and let $L \subseteq \Sigma^*$ be a regular language. Then L is a context-free language.

Proof
Since L is a regular language, there exists a deterministic finite automaton $M = (Q, \Sigma, \delta, q, F)$ that accepts L. To prove that L is context-free, we have to define a context free grammar $G = (V, \Sigma, R, S)$, such that $L = L(M) = L(G)$. Thus, G must have the same following property:
For every string $w \in \Sigma^*$,

$$w \in L(M)\ if\ and\ only\ if\ w \in L(G)$$

which can be reformulated as

$$M\ accepts\ w\ if\ and\ only\ if\ S \overset{*}{\Rightarrow} w$$

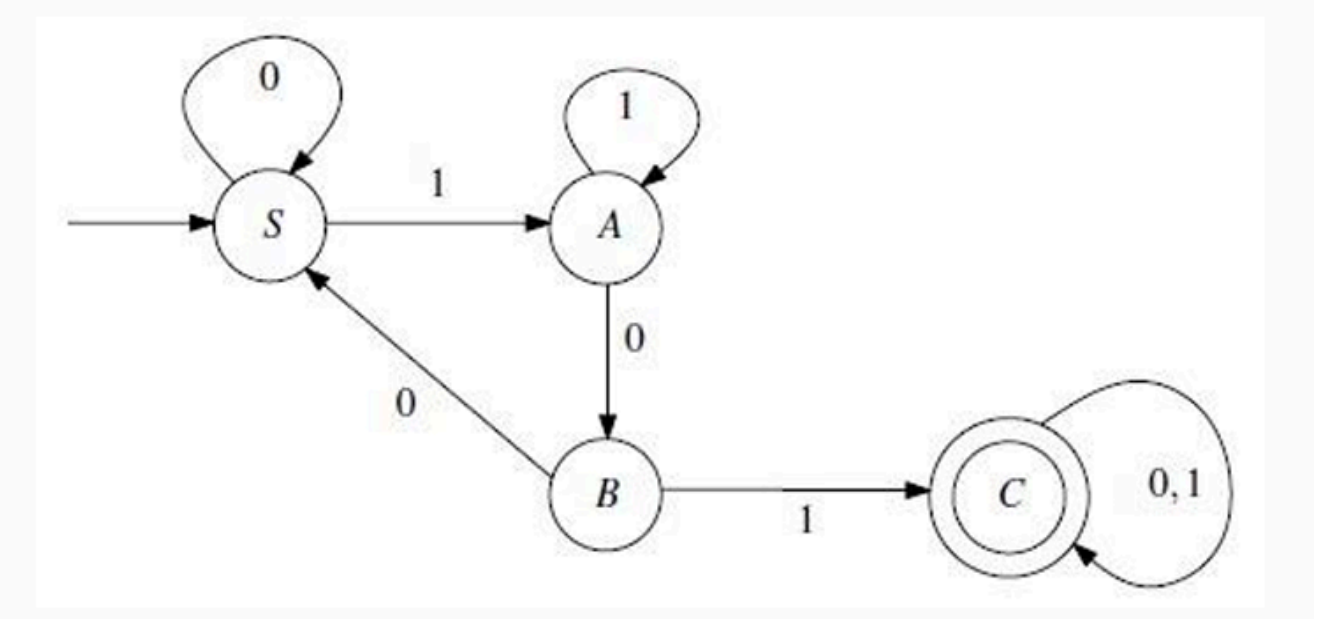
Set $V = \{R_i | q_i \in Q\}$(that is, G has a variable for every state of M). Now for every transition

$\delta(q_i, a) = q_j$ add a rule $R_i \to aR_j$. For every accepting state $q_i \in F$ add a rule $R_i \to \epsilon$. Finally, make the start variable $S = R_0$.

Example

Let L be the language defined as $L = \{w \in 0, 1^* : 101\ is\ a\ substring\ of\ w\}$

The DFA M accepts L



convert M to a context-free grammar G whose languages is L

Solution:

$V = \{S, A, B, C\}$

$\Sigma = \{0, 1\}$

The start variable: S

Rules R:

transit from S:

$$S \to 0S$$

$$S \to 1A$$

transit from A:

$$A \to 1A$$

$$A \to 0B$$

transit from B:

$$B \to 0S$$

$$B \to 1C$$

transit from C:

$$C \to 0C$$

$$C \to 1C$$

Closure properties of CFLs: CFLs are closed under operations like union and concatenation but not under intersection or complementation

Chomsky Normal Form(CNF)

Definition
A context-free grammar $G = (V, \Sigma, R, S)$ is said to be in Chomsky normal form, if every rule in R has one of the following three forms:

- $A \to BC$, where $A, B, andC$ are elements of $V, B \neq S, and\ C \neq S$ 非终结符生成两个非终结符
- $A \to a$, where A is an element of V and a is an element of Σ 非终结符生成一个终结符
- $S \to \epsilon$, where S is the start variable
  Grammars in Chomsky normal form are far easier to analyze

Theorem
Let Σ be an alphabet and let $L \subseteq \Sigma^*$ be a context-free language. There exists a context-free grammar in Chomsky normal form, whose language is L(Every CFL can be described by a CFG in CNF)

CFG$\to$ CNF
Given CFG $G = (V, \Sigma, R, S)$. Replace, one-by-one, every rule that is not "Chomsky"

- Start variable(not allowed on RHS of rules)
- $\epsilon - rule$($A \to \epsilon$ not allowed when A isn't start variable)
- all other violating rules($A \to B, A \to aBc, A \to BCDE$)

Transformation steps
Step 1. Eliminate the start variable from the RHS of the rules

- New start variable S_0
- New rule $S_0 \to S$

Step 2. Remove $\epsilon - rules$, where $A \in V - \{S\}$

- Before: $B \to xAy$ and $A \to \epsilon | \ldots$
- After: $B \to xAy | xy$ and $A \to \ldots$

when removing $A \to \epsilon$ rules, insert all new replacements:

- Before: $B \to AbA$ and $A \to \epsilon|\dots$
- After: $B \to AbA|bA|Ab|b$ and $A \to \dots$

Step 3. Remove unit rules $A \to B$, where $A \in V$

- Before: $A \to B$ and $B \to xCy$
- After: $A \to xCy$ and $B \to xCy$

Step 4. Eliminate all rules having more than 2 symbols on the RHS

- Before: $A \to B_1B_2B_3$
- After: $A \to B_1A_1, A_1 \to B_2B_3$

Step 5. Eliminate all rules of the form $A \to ab$, where a and b are not both variables

- Before: $A \to ab$
- After: $A \to B_1B_2, B_1 \to a, B_2 \to b$

Example:
Given a CFG $G = (V, \Sigma, R, S)$, where $V = \{A, B\}$,$\Sigma = \{0, 1\}$, A is the start variable, and R consists of the rules:

$$A \to BAB|B|\epsilon$$

$$B \to 00|\epsilon$$

convert this G to CNF

Step 1. Eliminate the start variable from the RHS of the rules

- New start variable: S_0
- New rule $S_0 \to S$

Step 2. Remove $\epsilon - rules$
New rule:
$S_0 \to A|\epsilon$
$A \to BAB|AB|BB|BA|A|B$
$B \to 00$

Step 3. Remove unit-rules
New rule:

$S_0 \to BAB|AB|BB|BA|00|\epsilon$
$A \to BAB|AB|BB|BA|00$
$B \to 00$

Step 4. Eliminate all rules having more than 2 symbols on the RHS
New rule:
$C_1 \to AB$
$S_0 \to BC_1|AB|BB|BA|00|\epsilon$
$A \to BC_1|AB|BB|BA|00$
$B \to 00$

Step 5. Eliminate all rules, whose RHS side contains exactly 2 symbols, which are not both variables
New rule:
$U \to 0$
$C_1 \to AB$
$S_0 \to BC_1|AB|BB|BA|UU|\epsilon$
$A \to BC_1|AB|BB|BA|UU$
$B \to UU$

先消除在右侧的起始变量（引入新的起始变量），然后消除ϵ规则，消除单一规则$X \to Y$，最后在转换成CNF，只能产生一个非终结符生成两个非终结符，或者一个非终结符到一个终结符的形式

Lecture 7 Context-Free Languages II

Closure Properties of Context Free Language

Theorem: If L_1 and L_2 are context-free languages, their union $L_1 \cup L_2$ is also context free.

Proof idea:
For L_1 and L_2, there exists corresponding CFG $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$. Let $G_3 = (S \cup V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1|S_2\}, S)$, clearly $L(G_3) = L_1 \cup L_2$

Theorem: If L_1 and L_2 are CFL, their concatenation $L_1 \circ L_2$ is also context free

Proof idea
For L_1 and L_2, there exists corresponding context free grammars $G_1 = (V_1, \Sigma_1, R_1, S_1)$ and $G_2 = (V_2, \Sigma_2, R_2, S_2)$. Let $G_3 = (S \cup V_1 \cup V_2, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S \to S_1S_2\}, S)$, clearly $L(G_3) = L_1 \circ L_2$

Theorem: If If L_1 is CFL, their Kleene closure L_1^* is also context free

Proof idea
For L_1, there exists corresponding context free grammars $G_1 = (V_1, \Sigma_1, R_1, S_1)$. Let $G_2 = (S \cup V_1\, \Sigma_1, R_1 \cup \{S \to S_1 S | \epsilon\}, S)$

CFLs are not closure under intersection & complement

Counterexample:
$L_1 = \{a^n b^n c^n | n, m \geq 0\}$
$L_2 = \{a^m b^n c^n | n, m \geq 0\}$
$L_1 \cup L_2 = \{a^n b^n c^n | n \geq 0\}$
By De Morgan's law
$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$
the complement is also not closure

Pushdown Automata(PDAs)

有限自动机（NFA）+ 一个栈（Stack）
The class of languages that can be accepted by pushdown automata is exactly the class of context-free languages(finite automata are for regular languages)
能被PDA接受的语言正好是CFLs

- The input for a pushdown automaton is a string w in Σ^*
- PDA accepts or doesn't accept w.
- Different from finite automata, PDAs have a stack
- Stack have 2 different operations:
    - push -adds item to top of stack
    - pop - removes item from top of stacks

- A PDA consists of a tape, a stack and a state control
    - Tape: divided into cells that store symbols belonging to $\Sigma_epsilon = \Sigma \cup \{\epsilon\}$
    - Tape head: move along the tape, one cell to the right per move
    - Stack: containing symbols from a finite set Γ, called the stack alphabet. This set contains a special symbol $(often mark bottom of stack) 标记栈底
    - Stack head: reads the top symbol of the stack. This head can also pop the top symbol, and it can push symbols of Γ onto the stack
    - State control: can be in ant one of a finite number of states. This set of states is denoted by Q. The set Q contains one special state q, called the start state

PDA definition

A pushdown automation is a 6-tuple $M = (Q, \Sigma, \Gamma, \delta, q, F)$:

- Q is finite set of states
- Σ is finite input tape alphabet
- Γ is finite stack alphabet
- δ is the transition function: $Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$
- q is start state, $q \in Q$
- F is set of accept states, $F \subseteq Q$, PDA accepts as long as it is in F regardless of the stack
  Let $r, r' \in Q, a \in \Sigma*$ and $b, c \in \Gamma^*$

  $$\delta(r, a, b) = \{(r', c')\}$$

  In state r, PDA reads a on the tape and pop b from the stack, move to state r' and push c to the stack. The tape head moves to the right

---

PDA is Nondeterministic, PDA transition function allows for nondeterminism

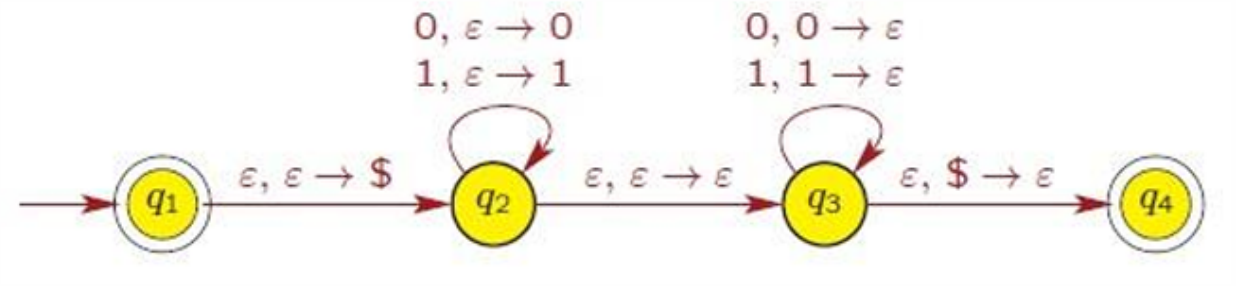
$$\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to P(Q \times \Gamma_\epsilon)$$

Language accepted by PDA

Definition
The set of all input strings that are accepted by PDA M is the language recognized by M and is denoted by $L(M)$

CFG $\Leftrightarrow$ PDA

Example:
PDA for language $\{ww^R | w \in \{0,1\}^*\}$



CFG: $G = \{\{S\}, \{0,1\}, R, S\}$
R:

$$S \to 0S0|1S1|\epsilon$$

Lecture 7 Context-Free languages III

Equivalence of PDA and CFGs

Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. Then A is a context-free if and only if there exists a nondeterministic pushdown automaton that recognizes A

- If $A = L(G)$ for some CFG G, then $A = L(M)$ for some PDA M
- If $A = L(M)$ for some PDA M, then $A = L(G)$ for some CFG G
  Basic idea: Given CFG G, convert it into PDA M with $L(M) = L(G)$ by building PDA that simulates a derivation, where stack symbols are CFG symbols and type reading happens when a stack head is terminal and matches with tape control 堆栈的符号是CFG符号 栈顶是终结符且与输入带符号匹配时读带，从而将其转换满足为$L(M) = L(G)$

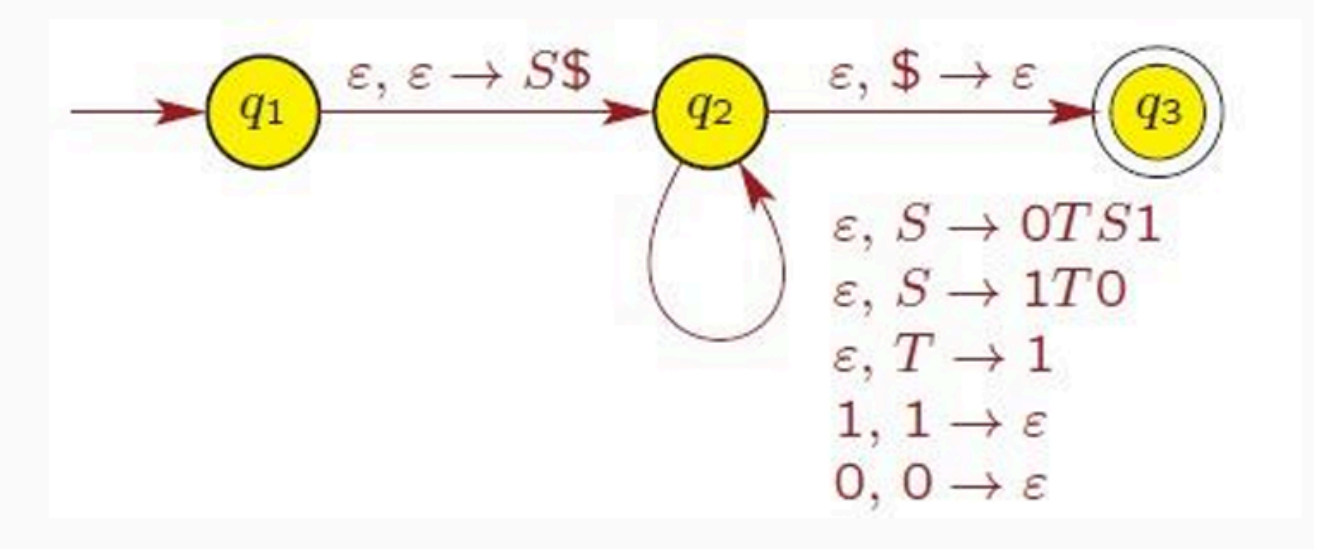
Convert CFG into PDA

CFG $G = (V, \Sigma, R, S)$
Start symbol S,
production rules $A \in V \rightarrow U \in (V \cup \Sigma)$,
Terminal Symbols $a \in \Sigma$,
End generation when no variables left
Problem: U could consist of multiple symbols, PDA only push one at a time.
Solution: enter intermediate states when a production rule is applied, and use epsilon transitions to add symbols one by one 在应用产生式规则时进入中间状态 并使用 $\epsilon - transition$逐个加入符号

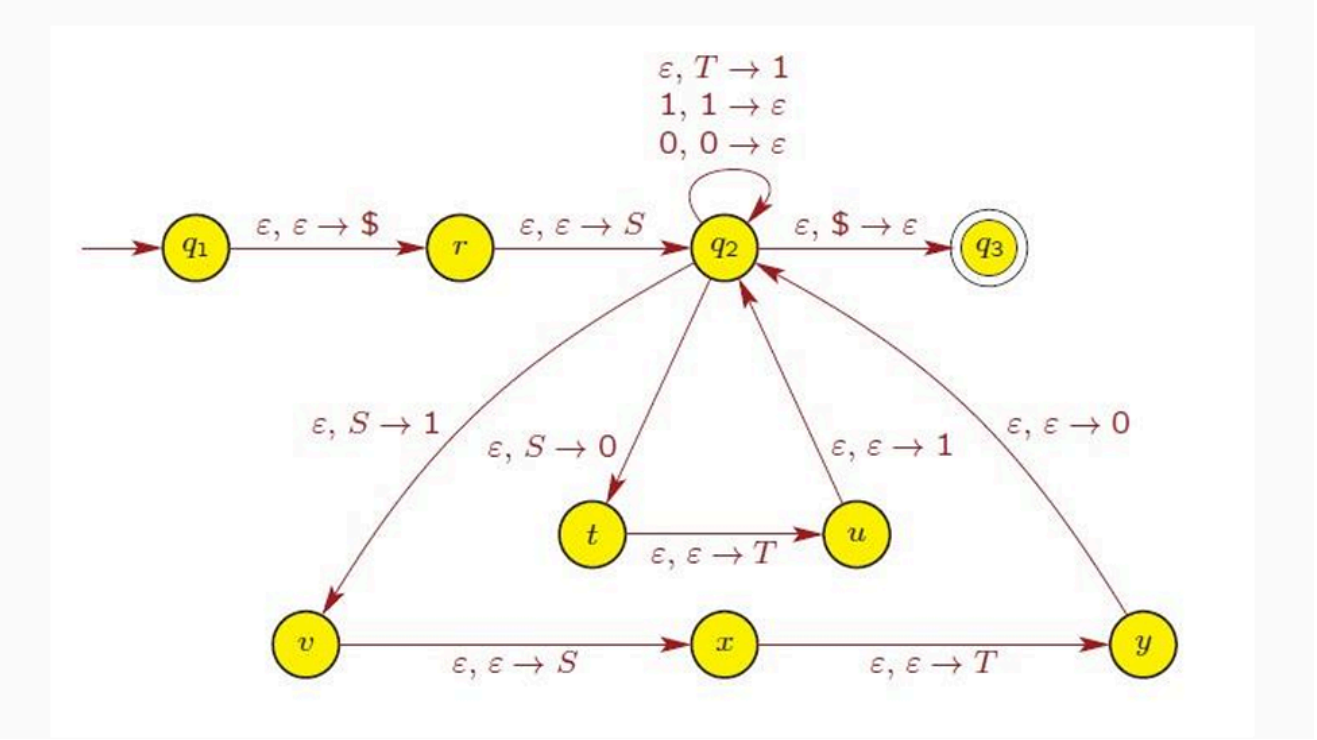
PDA works as follows

1. $\epsilon, \epsilon \rightarrow S\$$ 初始化栈 准备好模拟推导过程 S是CFG的起始变量
2. $\epsilon, A \rightarrow u$ 不读输入字符 且栈顶是变量A 将栈顶的变量A弹出 替换为产生式右侧的字符串（对应规则$A \rightarrow u$）
3. $a, a \rightarrow \epsilon$ 读取输入字符a，且栈顶也是终结符a匹配成功将栈顶a弹出，验证生成的字符是否与输入字符串一致
4. $\epsilon, \$ \rightarrow \epsilon$ 当栈中只剩下栈底符号时，进入接受状态

Example
Given CFG $G = (V, \Sigma, R, S)$ Variables $V = \{S, T\}$ Terminals $\Sigma = \{0, 1\}$ Rules $S \rightarrow 0TS1|1T0, T \rightarrow 1$ convert the above CFG into a PDA



but PDA cannot push strings(e.g. $S\$, 0TS1$) onto stack. We need to create the intermediate states



Convert PDA into CFG

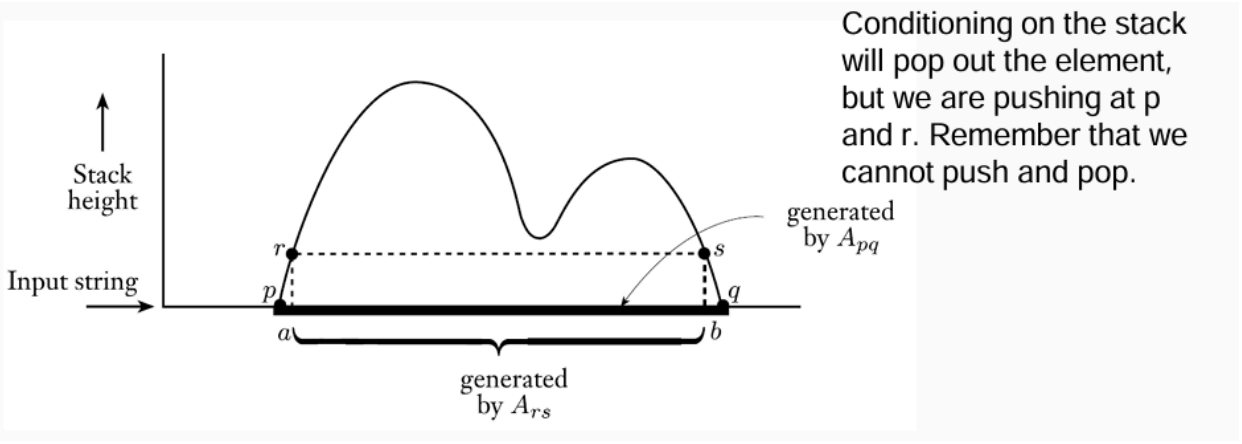
the PDA we built from CFG looks quite special. It has a single accepting state with empty stack. This suggests that we need to somehow convert all PDA to some simplified PDA 空栈作为单一接受状态

We propose the following simplification of PDA

1. PDA P has a single accepting state q_{accept}
2. PDA P accepts on empty stack
3. Each transition either pushes a symbol onto the stack(a push move) or pops one off the stack(a pop move), but it does not do both at the same time 每个转移要么将一个符号压到栈中，要么弹出，不会同时进行

Lemma:
PDA with the above simplifications are equivalent to standard PDA

Formally, for PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$
We construct $G = (V, \Sigma, R, A_{q_0 q_{accept}})$ with $V = \{A_p q | p, q \in Q\}$
and most importantly production rules in 3 parts 这3条规则涵盖了PDA从状态p走到状态q且清空栈的所有可能路径



1. For each $p \in Q$, put $A_{pp} \to \epsilon$ base case 起点就是重点 什么都不用做
2. For each $p, q, r \in Q$, put $A_{pq} \to A_{pr} A_{rq}$(路径拼接 我们可以通过一个中间状态来拆分旅程，如果想从p走到q且清空栈，我们可以先走到中间的某个状态（此时栈的状态必须回到空），然后再从中间状态到q)
3. For each $p, q, r, s \in Q, u \in \Gamma$(stack symbol), and $a, b \in \Sigma_\epsilon$, if $\delta(p, a, \epsilon)$ contains (r, u) and $\delta(s, b, u)$ contains (q, Σ), put $A_p q \to a A_{rs} b$
   整个过程被包在一个主要的压栈与出栈的操作中
   开始从状态p读取输入a，压入一个符号u，转移到状态r 此时栈变高了；
   中间$r \to s$ 不管发生什么复杂的计算栈的高度永远没有低过被压入的u，符号u一直在栈底；
   结束状态，在状态s读取输入b，看到栈顶是u将其弹出，转移到状态q，此时状态回到基准线

Example
$L = \{01\}$ now PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$
$Q = q_0, q_1, q_2$
$\delta(q_0, 0, \epsilon) \to \{(q_1, X)\}$
$\delta(q_1, 1, X) \to \{(q_2, \epsilon)\}$

Solution
generate the CFG rules
Step 1 Base case: For each $p \in Q$,$A_{pp} \to \epsilon$

- $A_{00} \to \epsilon$
- $A_{11} \to \epsilon$
- $A_{22} \to \epsilon$
  Step 2 (Rule 3)
- Push: ($p \to q$) read 0, and push X
- Pop:($p \to q$) read 1, and pop X
- So the rule $A_{02} \to 0A_{11}1$
  Step 3 (Rule 2)
- now we have$A_{00} \to \epsilon | A_{11} \to \epsilon | A_{22} \to \epsilon$ | $A_{02} \to 0A_{11}1$
- so we have $A_{02} \to 0\epsilon 1$

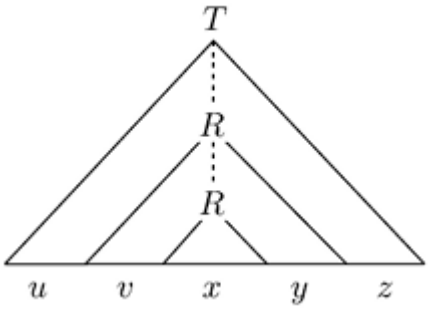
Pumping Lemma for CFLs

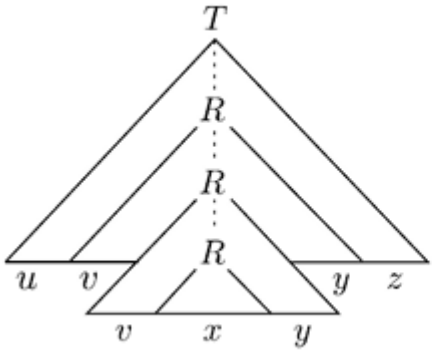
If we try to derive infinite long string without repeating variables, we run out of variables sooner or later

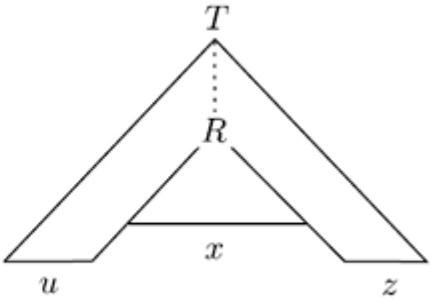
Let L be a context-free language. Then there exists an integer $p \geq 1$, called the pumping length, such that the following holds: Every string s in L, with $|s| \geq p$, can be written as $s = uvxyz$, such that

1. $|vy| \geq 1$(i.e., v and y are not both empty)
2. $|vxy| \leq p$ and
3. $uv^ixy^iz \in L$, for all $i \geq 0$

Proof idea:
Since the grammar is finite, if every variable only appear once in the parse tree, only strings pf finite length can be generated. For long enough strings, there must exists variables being used at least twice. Consequently, we can pump it.
u和z是两边的部分，v和y是上面的R生成但包裹在下面R之外的部分，x是由下面R生成的核心部分







Proof
Let b = length of the longest RHS of a rule = max branching of the parse tree
let h = height of the parse tree for (string) s
A tree of height h and max branching b has at most b^h leaves. So $|s| \leq b^h$
let $p = b^{|V|} + 1$ where $|V|$ = # variables in the grammar
So if $|s| \geq p > b^{|V|}$ then $|s| > b^{|V|}$ and $h > |V|$
As only variables can be nonterminal, at least $|V| + 1$ variables occur in the longest path. So some variable R must repeat on a path

---

Example
CFL: $S \to \epsilon|0S0|1S1$
Branching factor:$B = 3$
Variable numbers: $V = 1$
Pumping length $p = b^{|V|} + 1$
For strings of length, lets consider $s = 0110$
Pumping Lemma claims $s = uvxyz$, such that

1. $|vy| \geq 1$(i.e., v and y are not both empty)
2. $|vxy| \leq p$ and
3. $uv^ixy^iz \in L$, for all $i \geq 0$
   $u = 0, v = 1, x = \epsilon, y = 1, z = 0$

Example.2
$L = \{a^n b^n c^n | n\}$ is non-CFL
Proof by contradiction:
we assume that B is a CFL. Let p be the pumping length $s = a^p b^p c^p$. Clearly s is a member of B and of length at least p
we show that no matter how we divide s into $uvxyz$, one of the 3 conditions of the lemma is violated
Condition 2 require we find the vxy substring to be of length p at most
Case 1: If a^{p-k} where $k \geq 0$, and due to $|vy| \geq 1$, pumping will change the number a, and we get strings not in the language. Similarly, $b^{p-k} = vxy$ or $c^{p-k} = vxy$ won't work so vxy must be the combination of 2 alphabets
Case 2: $a^m b^{p-m-k} = vxy$ where $k \geq 0$. Still, after pumping either the number of a or b will be different from number c. Similarly, $b^m c^{p-m-k} = vxy$ won't work
In all, we get a contradiction by assuming L is CFL, so it must not be CFL.

Lecture 9 Turing machine

- Infinite long type, divided into cells. Each cell stores a symbol belonging to Γ (tape alphabet)
- Tape head(↓) can move both right and left, one cell per move. It read from or write to a tape 每移动一格，他可以读取或写入纸带
- State control can be in any one of a finite number of states Q. It is based on: state and symbol read from tape 状态控制器可以处于有限状态集中的任何一个状态，他基于状态和从纸带读取的符号
- Machine has one start state, and can accept or reject at any time
- Machine can run forever: infinite loop
- Properties of Turing machine
    - TM can both read from tape and write on it
    - Tape head can move both right and left
    - Tape is infinite and can be used for storage 纸带是无限的也可以用于出处
    - Accept and reject states take immediate effect
- Definition
  A Turing Machine (TM) is a 7-tuple $M = \{\Sigma, \Gamma, Q, \delta, q, q_{accept}, q_{reject}\}$,where
- Σ is a finite set, called the input alphabet, the blank symbol $\sqcup$ is not contained in Σ
- Γ is a finite set, called the tape alphabet, this alphabet contains the blank symbol, and $\sqcup$, and $\Sigma \subseteq \Gamma$
- Q is a finite set. whose elements are called states
- q is an element of Q, it is called the start state
- q_{accept} is an element of Q, it is called the accept state
- q_{reject} is an element of Q, it is called the reject state
- δ is called the transition function, which is a function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, L\}$

- L move to left, R move to right, N no move

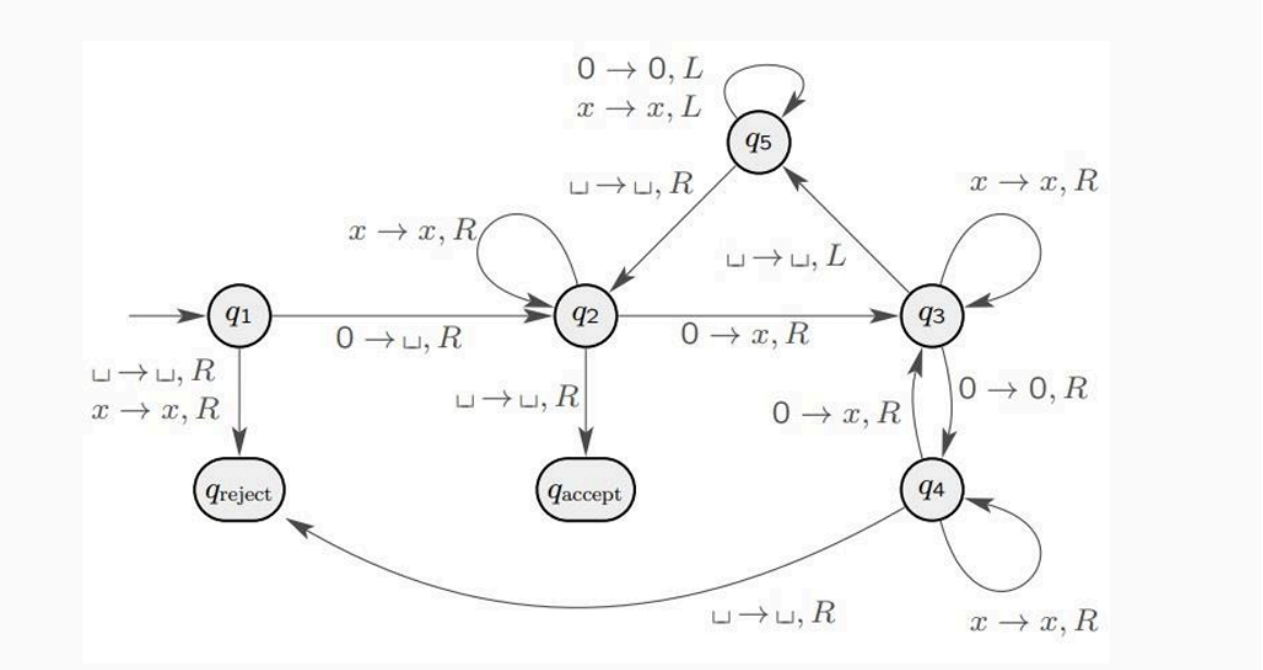
Example
TM M for language

$$A = \{0^{2^n} | n \geq 0\}$$

which consists of strings of 0s whose length is a power of 2

On input w:

- Sweep left to right across the tape, crossing off every other 0
- If in stage 1 the tape contains a single 0, accepts
- If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, reject
- Return the head to the left end of the tape



- Start configuration: The input is a string over the input alphabet Σ. Initially, this input string is stored on the first tape, and the head of this tape is on the leftmost symbol of the input string
- Computation and termination: Starting in the start configuration, the Turing machine performs a sequence of computation steps. The computation terminates at the moment when the Turing machine enters the accept state or reject state
- Acceptance: The TM M accepts the input string $w \in \Sigma^*$, if the computation on this input terminates in the state q_{accept}

TM configuration

Provides a "snapshot of TM" at any point during computation

- State
- tape contents
- head location
  Definition
  Configuration of a TM $M = (Q, \Sigma, \Gamma, \delta, q, q_{accept}, q_{reject})$is a string uqv with $u, v \in \Gamma^*$ and $q \in Q$, and specifies that currently
- M is in state q
- Tape contains uv
- tape head is pointing to the cell containing the first symbol in v

TM Transitions

- Definition
  Configuration $C1$ yields configuration $C2$ if the Turing machine can legally go from $C1$ to $C2$ in a single step. For TM $M = (Q, \Sigma, \Gamma, \delta, q, q_{accept}, q_{reject})$, suppose
  - $u, v \in \Gamma^*$
  - $a, b, c \in \Gamma$
  - $q_i, q_j \in Q$
  - transition function $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$
- Example
  Configuration uaq_ibv yields configuration uq_jacv

$$if\ \delta(q_i, b) = (q_j, c, L)$$

TM Computation

- Definition
  Given a TM $M = (Q, \Sigma, \Gamma, \delta, q, q_{accept}, q_{reject})$ and input string $w \in \Sigma^*$. M accepts input w if there is a finite sequence of configurations $C_1, C_2, \dots C_k$ for some $k \geq 1$ with
  C_1 is the starting configuration $q0w$
  C_i yields C_{i+1} for all $i = 1, \dots, k-1$(sequence of configurations obeys transition function δ)
  C_k is an accepting configuration $uq_{accept}v$ for some $u, v \in \Gamma^*$

Recognizable Languages

The language $L(M)$ accepted by the Turing Machine M is the set of all strings in Σ^* that accepted by M
Language A is Turing-recognizable if there is a TM M such that $A = L(M)$

- On an input $w \notin L(M)$, the machine M can either halt in a rejecting state, or it can loop indefinitely 要么机器M在拒绝状态停止 要么无限循环
- On an input $w \in L(M)$, the machine M will halt in accepting state
  Turing recognizable not practical because we never know if TM will halt

Multi-tape TM

Multi-tape TM has multiple tapes

- Each tape has its own head
- Transition determined by
  - state
  - the content read by all heads
- Reading and writing of each head are independent of others
  Definition
- A k-tape Turing Machine (TM) is a 7-tuple $M = \{\Sigma, \Gamma, Q, \delta, q, q_{accept}, q_{reject}\}$has k different tapes and k different read/write heads, where
- Σ is a finite set, called the input alphabet, the blank symbol $\sqcup$ is not contained in Σ
- Γ is a finite set, called the tape alphabet, this alphabet contains the blank symbol, and $\sqcup$, and $\Sigma \subseteq \Gamma$
- Q is a finite set. whose elements are called states
- q is an element of Q, it is called the start state
- q_{accept} is an element of Q, it is called the accept state
- q_{reject} is an element of Q, it is called the reject state
- δ is called the transition function, which is a function $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, N\}^k$
  - L move to left, R move to right, N no move
  - Given $\delta(q_i, a_1, a_2, \dots a_k) = (q_j, b_1, b_2, \dots b_k, L, R, \dots, L)$
    - TM is in state q_i
    - heads $1 - k$ read $a_1, a_2, \dots a_k$
  - Then
    - TM moves to q_j
    - heads $1 - k$ write $b_1, b_2, \dots, b_k$
    - Heads move left or right or don't move

Multi-tape TM equivalent to 1-tape TM

Let $k \geq 1$ be an integer. Any k-tape Turing machine can be converted to an equivalent 1-tape Turing machine. For every multi-tape TM M, there is a single-tape M' such that
$L(M) = L(M')$

Proof
Basic idea: simulate k-tape TM using 1-tape TM
Let TM $M = \{\Sigma, \Gamma, Q, \delta, q, q_{accept}, q_{reject}\}$ be a k-tape TM
M has:

- input $w = w_1, w_2, \dots w_k$ on tape 1
- other tapes contain only blanks $\sqcup$
- each heads points to first cell
  Construct 1-tape TM M' by extending tape alphabet

$$\Gamma' = \Gamma \cup \dot{\Gamma} \cup \{\#\}$$

  Note: head positions of different tapes are marked by dotted symbol

For each step of k-tape TM M, 1-tape M' operates its tape as:

- At the start of the simulation, the tape head of M' is on the leftmost #
- Scans the tape from first # to (k+1)st # to read symbols under heads 从第一个#扫描到第k+1个#，读取磁头下的符号
- rescans to write new symbols and move heads

Turing-recognizable $\leftrightarrow$ Multiple-tape Turing-recognizable
Language L is TM-recognizable if and only if some multi-tape TM recognizes L

Nondeterministic TM

A nondeterministic Turing machine(NTM) M can have several options at every step. It is defined by the 7-tuple $M = \{\Sigma, \Gamma, Q, \delta, q, q_{accept}, q_{reject}\}$, where

- Σ is a finite set, called the input alphabet, the blank symbol $\sqcup$ is not contained in Σ
- Γ is a finite set, called the tape alphabet, this alphabet contains the blank symbol, and $\sqcup$, and $\Sigma \subseteq \Gamma$
- Q is a finite set. whose elements are called states, transition function
  $\delta : Q \times \Gamma \to P(Q \times \Gamma \times \{L, R\})$
- q is an element of Q, it is called the start state
- q_{accept} is an element of Q, it is called the accept state
- q_{reject} is an element of Q, it is called the reject state

Computation

With any input w, computation of NTM is represented by a configuration tree
If $\exists$(at least) one accepting leaf, then NTM accepts

NTM equivalent to TM

Every nondeterministic TM has an equivalent deterministic TM
Proof

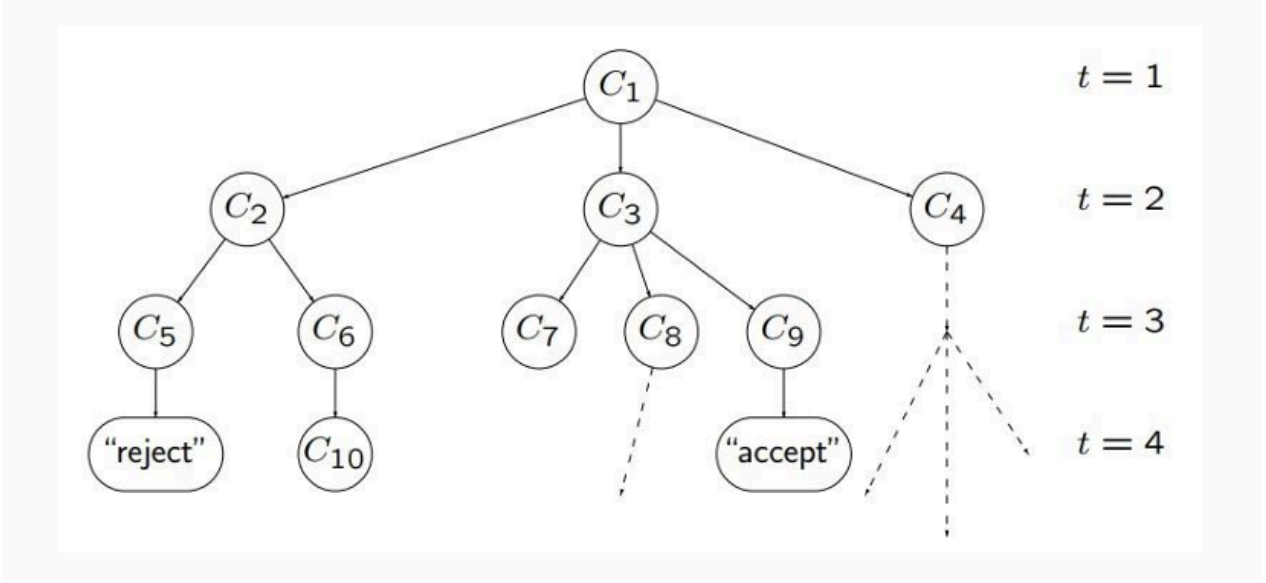
- Build TM D to simulate NTM N on each input w. D tries all possible branches of N's tree of configuration
- If D finds any accepting configuration, then it accepts input w
- If all branches reject, then D rejects input w
- If no branch accepts and at least one loops, then D loops on w

Address

- Every node in the tree has at most b children
- b is size of largest set of possible choices for N's transition function
- Every node in tree has an address that is a string over the alphabet $\Gamma_b = \{1, 2, \dots b\}$ To get to node with address 231:
  - start at root
  - take second branch
  - then take third branch
  - then take first branch
- Ignore meaningless address
- Visit nodes in breadth-first search order by listing addresses in Γ_b^* in string order:

$$\epsilon, 1, 2, \dots, b, 11, 12, \dots 1b, 21, 22$$

- example
  "accept" configuration has address 231
  Configuration C_6 has address 12
  Configuration C_1 has address ϵ
  Address 132 is meaningless



Simulating NTM by DTM

1. Initially, input tape contains input string w. Simulation and address tapes are initially empty
2. Copy input tape to simulation tape
3. Use simulation tape to simulation NTM N on input w on path in tree from root to the address on address tape
    - At each node, consult next symbol on address to determine which branch to take
    - Accept if accepting configuration reached
    - Skip to next step if
        - symbols on address tape exhausted 地址纸带上的符号耗尽
        - nondeterministic choice invalid
        - rejecting configuration reached
4. replace string on address tape with next string in Γ_b^* in string order, and go to Stage 2
   more explain:
   input tape存放原始输入w，作用是只读，用来当作存档，每次模拟一条新路径失败了，都要回来这里把原始数据重新拷贝一份，模拟带是NTM的沙盒或草稿纸，每一轮模拟都在这里进行，如果这条路走不通，就清空它准备下一轮，地址带相当于导航指令（广度搜索写入）

Turing machine Possible Outcomes
On input w a TM M may halt(enter q_{acc} or q_{rej}) or loop. So M has 3 possible outcomes for each input w

---

Decidable Languages

- Definition
  A ***decider*** is TM that halts on all inputs, i.e. never loops
  Language $A = L(M)$ is decided by TM M if on each possible input $w \in \Sigma^*$, the TM finishes in a halting configuration, i.e.
    - M ends in q_{accept} for each $w \in A$
    - M ends in q_{reject} for each $w \in A$
- A is Turing-decidable if $\exists$ TM M that decides A
- Differences to Turing-recognizable languages:
    - Turing-decidable language has TM that halts on every string $w \in \Sigma^*$ 图灵可判定要求对于每个输入都停机不会死循环

  - TM for Turing-recognizable language may loop on strings $w \notin$ this language 图灵可识别要可能会死循环 不一定会停机

- 所有有限语言都是可判定语言 这其中包括正则语言 上下文无关语言...
- 停机问题 确定图灵机是否接受任何字符串的问题 确定两个图灵机是否等价的问题 确定上下文无关法是否生成所有字符串的问题都属于不可判定问题

Lecture 10 Church-Turing Thesis and Limit of Computation

Diagonalization method

Questions:
If we use natural numbers to list natural numbers, couldn't we construct a new number by flipping the diagonal and get a new number that is not in the natural number set

Answers:
True but the number you get is of infinite length. All natural numbers are of finite length. Therefore, you actually constructs a real number. There is nothing wrong with a real not in the natural number set

The church-Turing Thesis

existence of non-Turing-recognizable languages
Proposition: Each Turing machine can be encoded by a distinct, finite string of 1's and 0's and some finite special symbols
Proof: encode TM as 7 tuple with special symbols, and encode alphabets in binary. Transitions can be encoded as sequence of 5 tuples(state, tape, new state, new tape, left or right)
Corollary: There are countable many Turing machines 图灵机是可数的 如果把它想成一段代码，虽然能有无限种输入 但是代码文件本身是有现成的

Proposition: There are uncountable many languages 有不可数多的语言
Proof: There is a bijection between languages and countable binary sequences that is uncountable by the diagonalization method 语言与可数二进制序列之间存在双射，根据对角线法，二进制序列是不可数的
Corollary: There exists non Turing-recognizable languages 图灵机有N个 但是语言有无数多个 所以鸽巢原理 一定会有一个语言是Non-Turing-recognizable language

---

Theorem **Universal Turing Machine** Exists 通用图灵机
There exists a TM U that takes a Turing machine description and input tape and simulated one step of that given Turing machine on input tape

- Input to TM U is in the format $< M, w >$ where M is the TM description, and w is the converted input

---

To prove the thesis, we need to show that the world is Turing computable
To disprove the thesis, we need to prove there is a non-Turing-recognizable/decidable languages that can be recognized or decided by a physical device

Decidability

Given a language L whose elements are pairs of the form (B, w), where

- B is some computation model(e.g. DFA, NFA...)
- w is a string over the alphabet Σ
  The pair $(B, w) \in L$ if and only if $w \in L(B)$
  Since the input to computation model B is a string over Σ, we must encode the pair (B, w) as a string

Acceptance problem for computational model

Decision problem: Does the given model accept/reject a given model w?
Instance $< B, w >$ is the encoding pair (B, w)
Universe Ω comprises every possible instance:

$$\Omega = \{< B, w > | B\ is\ a\ model\ and\ w\ is\ a\ string\}$$

Language comprises all "yes" instances:

$$L = \{< B, w > | B\ is\ a\ model\ that\ accept\ w\} \subseteq \Omega$$

- L_{DFA} is decidable

  $$L = \{< B, w > | B\ is\ a\ DFA\ that\ accept\ w\} \subseteq \Omega$$
  $$\Omega = \{< B, w > | B\ is\ a\ model\ and\ w\ is\ a\ string\}$$

  Basic idea:
  To prove L_{DFA} is decidable, we need to construct TM M that decides L_{DFA}
  For M that decides L_{PDA}:

  > - take $< B, w > \in \Omega$ as input
  > - halt and accept if $< B, w > \in L_{DFA}$
  > - halt and reject if $< B, w > \notin L_{DFA}$
  >   Proof
  >   On input $< B, w > \in \Omega$ where
  > - $B = (\Sigma, Q, \delta, q_0, F)$ is a DFA
  > - $w = w_1 w_2 \dots w_n \in \Sigma^*$ is input string to process on B

1. check if $< B, w >$ is "proper" encoding. If not, reject
2. Simulate B on w based on:
    - $q \in Q$, the current state of B
    - $i \in \{1, 2, \dots, |w|\}$, the pointer that illustrates the current position in w
    - q changes in accordance with w_i and the transition function $\delta(q, w_i)$
3. If B ends in $q \in F$, then M accepts; otherwise, reject.

- CFGs are decidable
  A context-free grammar $G = (V, \Sigma, R, S)$ is in Chomsky normal form if each rule is of the form

$$A \to BC \ or \ A \to x \ or \ S \to \epsilon$$

    - every CFG can be converted into Chomsky normal form, CFG G in CNF is easier to analyze
        - For any string $w \in L(G)$ with $w \neq \epsilon$ by derivation $S \overset{*}{\Rightarrow} w$ takes exactly $2|w| - 1$ steps. Base case where w is singular letter takes 1 step. Additional one step to increase number of variables and other to realize it
    - Proof
      S="On input $< G, w >$", where G is a CFG and w is a string:
        1. convert G into CNF
        2. List all derivations with $2n - 1$ steps, where n is the length of w; except if $n = 0$, then instead list all derivations with one step
        3. If any of these derivations generate w, accept; if not reject

Emptiness of CFLs are decidable

$E_{CFG} = \{< G > | G \ is \ a \ CFG \ and \ L(G) = \emptyset\}$
给定的CFG是否生成不了任何字符串
Proof idea:
Rule set is finite and try exhaustively build parse tree from all potential list of terminals, and check whether we can reach the start symbol 规则集是有限的 尝试从所有可能的终结符列表中穷举地分析语法分析树，并检查是否能达到起始符号

Proof
Construct a Turing Machine R
R="On input $< G >$, where G is a CFG"

1. Mark all terminal symbols in G
2. Repeat until no new variables get marked

3. Mark any variable A where G has a rule $A \to U_1U_2\dots U_k$ and each symbol $U_1U_2\dots U_k$ has already been marked
4. If the start variable is not marked, accept; otherwise, reject
   第二步中，检查文法中的每一个产生式，如果在箭头右边的所有符号都已经被标记过了，意味着它们能够生成终结符串，那么箭头左边的变量A
   也肯定能生成终结符串（反向标记）

The Language L_{TM} is Turing-recognizable

$$L_{TM} = \{< M, w >: M\ is\ a\ Turing\ machine\ that\ accepts\ the\ string\ w\}$$

If M accepts w, then $< M, w > \in L_{TM}$
If M doesn't accept w(reject or loop), then $< M, w > \notin L_{TM}$
The language L_{TM} is Turing-recognizable

Proof
A universal Turing machine U simulates M on w

- If M accepts w, simulation will halt and accept
- If M doesn't accept w(reject or loop),TM U either reject or loops

The Language L_{TM} is undecidable

$$L_{TM} = \{< M, w >: M\ is\ a\ Turing\ machine\ that\ accepts\ the\ string\ w\}$$

我们永远不知道通用图灵机是否会停机，内层的TM可能永远运行

Proof
We will prove by contradiction and use the diagonalization method. We assume such a decider H exists, then show that the set of Turing machine is uncountable
通过反证法并且使用对角线法来证明，我们假设存在这样的判定器H，然后根据图灵机的可数性推导出矛盾

$$H(\langle M, w\rangle) = \begin{cases} accept & \text{if } M \text{ accepts } w \\ reject & \text{if } M \text{ does not accept } w. \end{cases}$$

As the set of Turing machine is known to be countable, lets index them by M_i
Now, it takes sense to ask the answer $H < M_i, < M_j >>$ that is whether M_i accepts the description of M_j as input M_i是否接受M_j的描述作为输入
We may get a table which describes the answer of M_i是否接受M_j的描述作为输入

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	$accept$	$reject$	$accept$	$reject$	
M_2	$accept$	$accept$	$accept$	$accept$	$\cdots$
M_3	$reject$	$reject$	$reject$	$reject$	
M_4	$accept$	$accept$	$reject$	$reject$	
$\vdots$		$\vdots$			

By Diagonalization method:
we get a Turing Machine that is not in this table, so the set of Turing machines is not countable, and we have a contradiction, but how?

Implementation
we know this process will halt, as H is a decider
we construct another Turing Machine D that flips the diagonal(saying flipping is not enough, as the flipping needs to be done by a Turing Machine):
$D =$"On input $< M >$, where M is a TM"

1. run H on input $< M, < M >>$
2. Output the opposite of what H outputs, that is, if H accepts, reject, and if H rejects, accepts
   Clearly, D is a decider and hence a Turing machine, It should be in the list, but $\forall_i D(M_i) =!H(M_i, < M_i >=!M_i(< M_i >))$, Therefore, D is not in the list. This is contradiction to the fact TMs are countable
   OR
   An alternative proof:
   Should D accepts $< D >$?
   应该接受自己吗
   D本身是一个图灵机 如果D接受了$< D >$, 那么他在H那里就是接受的accept 但是他在对角线翻转中与自己矛盾了
   If D accepts $< D >$, In step 1 H accepts $< D, < D >>$, then in step 2 D rejects the $< D >$

Instance of non Turing-recognizable languages

Theorem: A language A is decidable if and only if it is Turing-recognizable and co-Turing-recognizable 语言A是可判定的当且仅当它是图灵可识别的且其补集也是图灵可识别的

Proof:
the only if part is simple, a decider always halts, and the decider accepts the language. For

the complement of the language, a Turing machine accepts when the decider rejects and vice versa
For the if part, if both A and $\bar{A}$ are Turing -recognizable, we let M_1 be the recognizer for A and M_2 be the recognizer for $\bar{A}$. The following Turing machine M is a decider for A
$M =$"On Input w:

1. Run both M_1 and M_2 on input w in parallel
2. If M_1 accepts, accepts; if M_2 accepts, rejects"
   Corollary $\overline{L_{TM}}$ is not Turing-recognizable

Lecture 11 Reducibility and Rice's Theorem

Undecidable Problems

- Definition:
  Undecidable problem The associates language of a problem cannot be recognized by a TM that halts for all inputs 一个问题的关联语言不能被一个对所有输入都停机的图灵机识别
  Unrecognizable problem The associated language of a problem cannot be recognized by a TM
  The language of TMs are undecidable but recognizable

$$L_{TM} = \{< M, w >: M\ is\ a\ Turing\ machine\ that\ accepts\ the\ string\ w\}$$

  We consider undecidable problems unsolvable(informal)
  对于可判定语言 等待时间有一个上界 一定会停机 对于不可判定语言 可能会死循环

Reducibility

- Definition:
  Reduction is a way of converting one problem to another problem, so that the solution to the second problem can be used to solve the first problem
  If A reduces to B, then any solution of B solves A (Reduction always involves two problems, A and B)
  > - If A is reducible to B, then A cannot be harder than B
  > - If A is reducible to B and B is decidable, then A is also decidable
  > - If A is reducible to B and A is undecidable, then B is also undecidable
- A common strategy for proving that a language L is undecidable is by reduction method, proceeding as follows:
  Typically approach to show L is undecidable via reduction from A to L:
  > - Find a problem A known to be undecidable (HALT problem)
  > - Suppose L is decidable
  > - Let R be a TM that decides L
  > - Using R as subroutine to construct another TM S that decides A

- But A is not decidable
- Conclusion: L is not decidable

Mapping reduction

- Definition:
  Suppose that A and B are two languages
  - A is defined over alphabet Σ_1^*, so $A \subseteq \Sigma_1^*$
  - B is defined over alphabet Σ_2^*, so $B \subseteq \Sigma_2^*$
    Then A is mapping reducible to B, written

    $$A \leq_m B$$

    if there is a computable(Turing Machine can simulate through the tape) function

    $$f : \Sigma_1^* \to \Sigma_2^*$$

    such that, for every $w \in \Sigma_1^*$

    $$w \in A \Leftrightarrow f(w) \in B$$

    The function f is called a reduction of A to B
- Theorem
  If $A \leq_m B$ and B is decidable, then A is decidable
  Proof
  - Let M_B be TM that decides B
  - Let f be reducing function from A to B
  - Consider the following TM:
    M_A = "On input w:
    1. Compute $f(w)$
    2. Run M_B on input $f(w)$ and give the same result."
  - Since f is a reducing function, $w \in A \Leftrightarrow f(w) \in B$
    - If $w \in A$, then $f(w) \in B$, so M_B and M_A accept
    - If $w \notin A$, then $f(w) \notin B$, so M_B and M_A reject
  - Thus, M_A decides A
- More explanation
  - 为了证明A是可判定的 需要造出一台能解决A问题的机器并且在所有输入结果下停机
- Corollary
  If $A \leq_m B$ and A is undecidable, then B is undecidable also.

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

---

- Theorem
  If $A \leq_m B$, then $\overline{A} \leq_m \overline{B}$
- Corollary
  L_{TM} is not mapping reducible to $E_{TN} = \{< M > | M\ is\ a\ TM\ and\ L(M) = \emptyset\}$

Mapping Reduction v.s. Turing(General) Reduction
Mapping Reducibility of A to B: translate A questions to B questions

$$w \in A \Longleftrightarrow f(w) \in B$$

Turing Reducibility of A to B: Use B solver to solve A
Clearly, we can use mapping reduction to construct general reduction

Emptiness of TMs is undecidable

$$E_{TM} = \{< M > | M\ is\ a\ TM\ and\ L(M) = \emptyset\}$$

Intuitively, we need to show the Turing machine won't enter the accept state for any strings. However, unlike the CFG/DFA cases, we find it hard to enumerate over all possible derivations
直观上 我们需要证明图灵机对于任何字符串都不会进入接受状态 然而 与CFG/DFA的情况不同 无法对所有可能的推导进行枚举

Proof by reduction from L_{TM}:
For a given TM M and input w, we construct a TM M_w s.t. M accepts w iff $L(M_w) \neq \emptyset$
We claim $M_w =$ "On input x:
1. If $x \notin w$, reject
2. If $x = w$, run M on input w and $accept$ if M does."
If M accepts w, in step 1 M_w rejects all strings other than w, and in step 2, M_w accepts it, and hence $L(M_w) \neq \emptyset$, then as it rejects all strings other than w, and accepts w only when M accepts w. Therefore M accepts w
L_{TM} is undecidable, so must E_{TM}.

核心思路是reduction 从L_{TM}归约到E_{TM} 因为L_{TM}是undecidable的 所以E_{TM} 也一定不可判定
通过先构造一个对外界任何输入的x的过滤器

1. 如果不等于w则直接拒绝

2. 如果等于 让内层机器M跑 如果它接受 那么 外层机器也接受 如果它不接受 这外层机器也不接受
   这种嵌套将M是否接受w的问题转换为了M_w是否是空集的问题
   如果原来的M接受w 那么我们制造出来的M_w 在接收到w之后 进入了第二步 运行内层 内层也接受 那么M_w 接受的语言$L(M_w)$里面至少有一个元素 是非空的
   如果原来的M不接受w 那么我们制造出的M_w 在接收到w之后 进入了第二步 运行内层 内层不接受 那么M_w 也拒绝了 那么M_w什么都不接受 所以$L(M_w)$是空集
   那我们制造出了一个机器能够判定机器是否为空 这就相当于我们也能检测L_{TM} 这是不可能成立的

Halting problem for TMs is undecidable

L_{TM} is undecidable, where

$$L_{TM} = \{< M, w > | M\ is\ a\ TM\ and\ M\ accepts\ string\ w\}$$

Define related problem

$$HALT_{TM} = \{< M, w > | M\ is\ a\ TM\ and\ M\ halts\ on\ string\ w\}$$

L_{TM} and $HALT_{TM}$ has same universe:

$$\Omega = \{< M, w > | M\ is\ TM,\ w\ is\ string\}$$

Given a $< M, w > \in Q$

- if M halts on input w, then $< M, w > \in HALT_{TM}$
- if M doesn't halt on input w, then $< M, w > \notin HALT_{TM}$

Proof by general reduction from L_{TM}
We reduce L_{TM} to $HALT_{TM}$. We claim the following machine S decides L_{TM}
$S =$"On input $< M, w >$, an encoding of a TM M and a string w
1. Run TM R on input $< M, w >$
2. If R rejects, $reject$
3. If R accepts, simulate M on w until it halts
4. If M has accepts, $accept$; if M has rejected, $reject$"
Clearly, S accepts $< M, w >$ iff M halts on w and M accept w, S rejects, iff M loops on w or M halts and rejects w
Now, as L_{TM} is undecidable, so must $HALT_{TM}$

Other Undecidable Problems

Define T to be the set pf all Turing machines descriptions, i.e.,

$$T = \{< M >: M\ is\ a\ Turing\ machine\}$$

- $INFINITE_{TM} = \{< M > | M\ is\ a\ TM\ and\ |L(M)| = \infty\}$
- $LT_{TM} = \{< M > | M\ is\ a\ TM\ and\ L(M) = L(T)\}$
- $FINITE_{TM} = \{< M > | M\ is\ a\ TM\ and\ \exists n \in N, |L(M)| = n\}$
- $ALL_{TM} = \{< M > | M\ is\ a\ TM\ and\ L(M) = \Sigma^*\}$
  Non-trivial properties in common:

1. none of them are empty set
2. none of them includes all Turing machines
3. $< M > \in P$ iff $L(M)$ satisfy some properties, i.e.,

$$P = \{< M > | M\ is\ a\ TM\ and\ p(L(M)) == 1\} where\ p : \{L(M) | M \in T\} \rightarrow \{1, 0\}$$

   Informal Rice's Theorem
   Any non-trivial property of Turing machine is undecidable

Rice's Theorem

Define T to be the set of all Turing machines descriptions, i.e.,

$$T = \{< M >: M\ is\ a\ Turing\ machine\}$$

Let P be a subset of T such that

1. $P \neq \emptyset$, i.e. there exists a Turing machine M such that $< M > \in P$,
2. P is a proper subset of T, i.e., there exists a Turing machine N such that $< N > \notin P$
3. for any two Turing machines M_1 and M_2 with $L(M_1) = L(M_2)$
   a) either both $< M_1 >$ and $< M_2 >$ are in P or
   b) none of $< M_1 >$ and $< M_2 >$ is in P
   Then the language P is undecidable
   非空真子集 并且对于相同语言的图灵机要么都在P中 要么都不在

$$E_{TM} = \{< M > | M\ is\ a\ TM\ and\ L(M) = \emptyset\}$$

1. A TM rejects all inputs will suffice is in the set
2. A TM accepts all input is not in the set
3. If $L(M_1) = L(M_2)$, they are both either $\emptyset$ or non-empty, either both in the set or not in the set respectively
   Therefore E_{TM} is undecidable by Rice's theorem

$$INFINITE_{TM} = \{< M > | M\ is\ a\ TM\ and\ |L(M)| = \infty\}$$

1. A TM rejects all inputs will suffice is in the set

2. A TM rejects all input is not in the set
3. If $L(M_1) = L(M_2), |L(M_1)| = |L(M_2)|$ so they are both either infinite or finite, either both in the set or bot in the set respectively
   Therefore $INFINITE_{TM}$ is undecidable by Rice's theorem

---

$$FIVE_{TM}\{< M > | M\ is\ a\ TM\ and\ M\ has\ 5\ states\}$$

We can't apply Rice's theorem here, as clearly we can have a UTM with 3 states recognize the same language as given 5 states TM by simulating that machine
In general, a property is about the language not about the TMs
Rice定理的核心是 你不存在一个适用于所有情况且能终止的机器 所有可证明的命题都是图灵可识别的 也就是说 如果你对可证明的命题进行穷尽搜索 总有一天能找到证明 然而 如果程序一直运行 你就不知道应该继续运行还是该命题是不可证明的

Rice's theorem and Proof

Let P be a proper non-empty subset of TM descriptions such that for $M1$ and $M2$ with $L(M1) = L(M2)$, $M1 \in P\ iff\ M2 \in P$. Then, P is undecidable

---

Proof attempt by reduction from L_{TM}
假设属性P 是可判定的 也就是说 存在一个图灵机判定器 R_p 他可以告诉你任何给定的图灵机M是否具有属性P 利用这个假设的R_p来构造一个能够解决$L(TM)$的机器 利用R_p来解决一个已知不可解的问题 推导出矛盾
归约：现在假设用判定器来解决L_{TM} 拿到了一对$< M, w >$ 根据以上的代码构造出M_w 的描述把M_w喂给R_p 如果R_p说是 即$M_w \in P$ 这意味着$L(M_w) = L(T)$ 倒退回去 意味着M接受了w 这相当于判定了 产生矛盾

---

we will reduce L_{TM} to all non-trivial property problems. Denote any given P, we have a decider R_p and a Turing machine T such that $< T > \in P$. For a given TM M and input w, ideally we construct a TM M_w s.t. M accepts w iff $< M_w > \in P$
$M_w^{(T)}$ = "On input w:
1. Simulate M on w. If it halts and rejects, reject. If it accepts, proceed to stage 2
2. Simulate T on x. If it accepts accept"
If M accepts w we have $L(M_w(T)) = L(T)$. As $< T > \in P$, we have $< M_w(T) \in P$
If M rejects or loops on w, we have $L(M_w(T)) = \emptyset$. Now $< M_w(T) > \notin P$ iff Turing machines with empty language is not in the property
obviously this is not true for all property