RESUME Theoretical Computer Science

STRINGS AND LANGUAGES

- The string of the length zero is called the empty string and is written as ε.
- A language is a set of strings.

Chap 1 : Regular languages

INTRODUCTION

An idealized computer is called a "computational model" which allows us to set up a manageable mathematical theory of it directly. As with any model in science, a computational model may be accurate in some ways but perhaps not in others. The simplest model is called "finite state machine" or "finite automaton".

FINITE AUTOMATA

- Finite Automata are good models for computers with an extremely limited amount of memory, like for example an automatic door, elevator or digital watches.
- Finite automata and their probabilistic counterpart "Markov chains" are useful tools when we are attempting to recognize patterns in data. These devices are used in speech processing and in optical character recognition. Markov chains have even been used to model and predict price changes in financial markets.
- The output of an finite automaton is "accepted" if the automaton is now in an accept state (double circle) and reject if it is not.
- A finite automaton is a list of five objects:
  - Set of states
  - Start state
  - Rules for moving.
  - Input alphabet
  - Accepts states
- A finite automaton has a transition for every possible letter from the alphabet going out every possible state and no ε transition.
- δ (x,1) = y , means that a transition from x to y exists when the machine reads a 1.
- Definition: A finite automaton is a 5 – tuple $(Q, \Sigma, \delta, q_0, F)$ where
  1. Q is a finite set called the states.
  2. Σ is a finite set called the alphabet (Epsilon).
  3. $\delta : Q \times \Sigma \rightarrow Q$ is the transition function (Sigma).
  4. $q_0 \in Q$ is the start state
  5. $F \subseteq Q$ is the set of accept states.
- If A is the set of all strings that machine M accepts, we say that A is the language of machine M and write L(M) = A. We say M recognizes A.
- A language is called a "regular language" if some finite automaton recognizes it.
- A finite automaton has only a finite number of states, which means a finite memory.
- Fortunately, for many languages (although they are infinite) you don't need to remember the entire input (which is not possible for a finite automaton). You only need to remember certain crucial info.

THE REGULAR OPERATIONS

We define 3 operations on languages, called the regular operations, and use them to study properties of the regular languages. Definition: Let A and B be languages. We define the regular operations union, concatenation and star as follows:

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

The class of regular languages is closed under the following operations : union, concatenation, intersection, star and complement.

Example: Let the alphabet Σ be the standard 26 letters {a, b, …, z}. If language A = {good, bad} and language B = {boy, girl}, then:

- $A \cup B$ = {good, bad, boy, girl}
- $A \circ B$ = {goodboy, goodgirl, badboy, badgirl}
- A^* = {ε, good, bad, goodgood, goodbad, badgood, badbad, goodgoodgood, goodgoodbas, …}

NONDETERMINISM

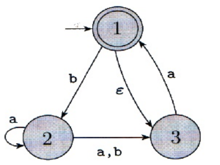
- Nondeterminism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton.
- In a DFA (deterministic finite automaton), every state always has exactly one exiting transition arrow for each symbol in the alphabet. In an NFA (nondeterministic finite automaton) a state may have zero, one or many exiting arrows for each alphabet symbol.
- Nondeterministic finite automata are useful in several respects. As we will show, every NFA can be converted into an equivalent DFA, and constructing NFAs is sometimes easier than directly constructing DFAs. An NFA may be much smaller than its deterministic counterpart, or its functioning may be easier to understand.

How does an NFA compute? Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. Fro example, say that we are in state q1 in NFA N1 and that the next input symbol is a 1. After reading that symbol, the machine splits into multiple copies of itself and follows all the possibilities in parallel. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it. Finally, if any one of these copies of the machine is in an accepts state ate the end of the input, the NFA accepts the input string. If a state with an ε symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple copies, one following each of the exiting ε - labelled arrows and one staying at the current state. Then the machine proceeds nondeterministically as before.

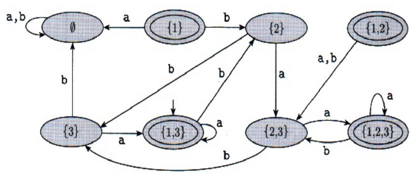
NFA – NNONDETERMINISTIC FINITE AUTOMATA

- Definition: A nondeterministic finite automaton is a 5 – tuple $(Q, \Sigma, \delta, q_0, F)$, where
  1. Q is a finite set of states.
  2. Σ is a finite alphabet.
  3. $\delta : Q \times \Sigma_\varepsilon \rightarrow P(Q)$ is the transition function, $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$
  4. $q \in Q_0$ is the start state.
  5. $F \subseteq Q$ is the set of accept states.
- In a DFA the transition function takes a state and an input symbol and produces the next state. In a NFA the transition function takes a state and an input symbol or the empty string and produces the set of possible next states.
- For any set Q we write P(Q) to be the collection of all subsets of Q (Power ser of Q).
- Deterministic and nondeterministic finite automaton recognize the same class of languages.
- Two machines are equivalent if they recognize the same language.
- Every NFA has an equivalent DFA.
- If k is the number of states of the NFA, it has 2k subsets of states. Each subset corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2k states.
- Transforming NFA to DFA : The DFA M accepts (means it is in an accept state) if one of the possible states that the NFA N could be in at this point, is an accept state.
- A language is regular if and only if some NFA recognizes it.

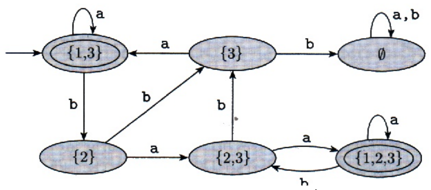
Example for transforming an NFA into equivalent DFA :



1. Determine D states : $2^3 = 8$ states : {∅}, {1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}}
2. D start state is the set of states that are reachable from 1 by traveling along ε arrows plus the state 1 itself = {1,3}



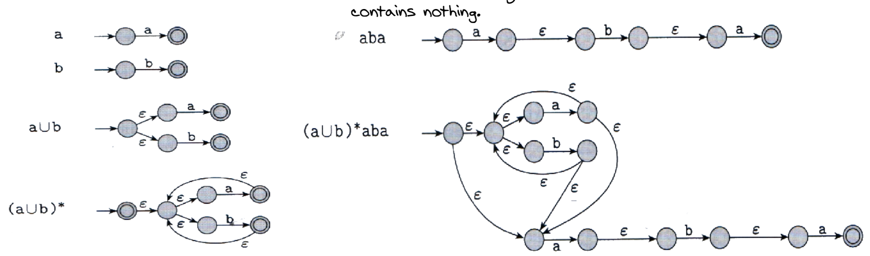
3. D start state is the set of states that are reachable from 1 by traveling along ε arrows plus the state 1 itself = {1,3}
4. D accept states are the states containing state 1, hence {1},{1,2},{1,3},1,2,3}
5. Then for each of the new state, arrows are added for each symbol of the alphabet depending on what was possible on the NFA.



6. Finally unnecessary states are removed

REGULAR EXPRESSIONS

- Definition: Say that R is a regular expression if R is:
  1. a for some a in the alphabet Σ.
  2. ε.
  3. $\emptyset$, $1^*\emptyset = \emptyset$, $\emptyset^* = \{\varepsilon\}$, $R \circ \emptyset = \emptyset$
  4. $(R1 \cup R2)$, where R1 and R2 are regular expressions.
  5. $(R1 \circ R2)$, where R1 and R2 are regular expressions.
  6. $(R1^*)$, where R1 is a regular expression.
- The value of a regular expression is a language.
- Regular expressions have an important role in computer science applications. In applications involving text, user may want to search for strings that satisfy certain patterns . Regular expressions provide a powerful method for describing such patterns.
- A language is regular if and only if some regular expression describes it. (Equivalence with finite automaton)
- Example for converting a regular expression to an finite Automaton (NFA) :
- This example consists of converting the RE $(a \cup b)^*aba$
- See how the various elements are put in place step by step
- See how the proofs for the closing of Regular languages under concatenation, union and star are used.

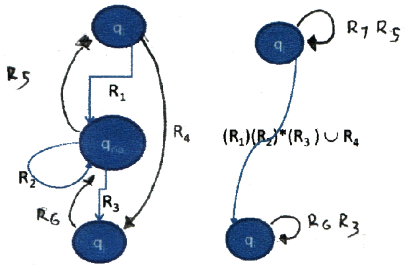


- We can write Σ as shorthand for the regular expression $(0 \cup 1)$. More generally, if Σ is any alphabet, the regular expression Σ describes the language consisting of all strings of length 1 over that alphabet, and Σ^* describes the language consisting of all strings over that alphabet. Similarly Σ^*1 is the language that contains all strings that end in a 1. The language $(0\Sigma^*) \cup (\Sigma^*1)$ consists of all strings that either start with a 0 or end with a 1.
- Precedence in regular expressions: $* > \circ > \cup$
- When we want to make clear a distinction between a regular expression R and the language that it describes, we write L(R) to be the language of R.
- Don't confuse between the ε regular expression representing the language containing the empty string an d the ∅ representing the language that contains nothing.

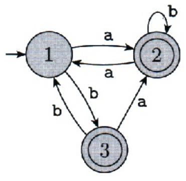
GENERALIZED NONDETERMINISTIC FINITE AUTOMATON

- The Generalized Nondeterministic Finite Automaton is an important concept as it allows to transfrom a Finite Automaton to a Regular Expression. The Tranformation path is DFA → GNFA → RE.
- Definition: A generalized nondeterministic finite automaton, $(Q, \Sigma, \delta, q_{start}, q_{accept})$ is a 5 – tuple where
  1. Q is the finite set of states.
  2. Σ is the input alphabet.
  3. $\delta : (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \to R$ is the transition function.
  4. q_{start} is the start state.
  5. q_{accept} is the accept state.
- The GNFA reads blocks of symbols form the input, not necessarily just one symbol at a time as in an ordinary NFA.
- For convenience we require that GNFAs always have a special form that meets the following conditions:
  - The start state has transition arrows going to every other state but no arrows coming in from any other state.
  - There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
  - Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.
- We can easily convert a DFA into a GNFA in the special form.
  - We simply add a new start state with an ε arrow to the old start state and a new accept state with ε arrows form the old accept states.
  - If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels.
  - Finally, we add arrows labeled ∅ between states that had no arrows. This last step won't change the language recognized because a transition labeled with ∅ can never be used.
- We let M be the DFA for language A. The we convert M to a GNFA G by adding a new start state and a new accept state and additional transition arrows as necessary. We use the procedure CONVERT(G), which takes a GNFA and returns an equivalent regular expression.

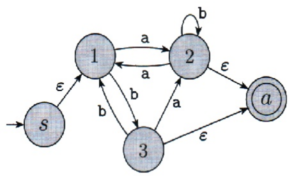
CONVERT(G): Generates a regular expression R out of a GNFA G:
- Let k be the number of states of G (the GNFA)
- if k = 2, then G must consist of a start state, an accept state and a single arrow connecting them and labeled with a regular expression R. return R
- if k > 2, we select any state $q_{rip} \in Q$ and different from the start and accept states and eliminate it from G. We obtain G' where labels between each q_i and q_j other than the start and accept states are as follows (see here ->)



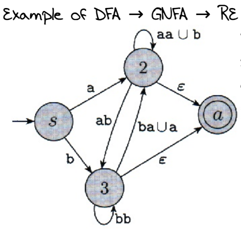
Example of DFA → GNFA → RE conversion, seeing every step :



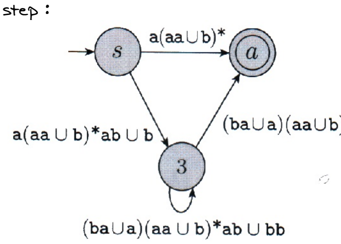
Step 0 : The initial DFA



Step 1 : From DFA to GNFA : new start state and accept state with ε arrows.

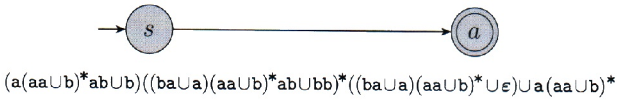


Step 2 : Getting rid of state 1 and replacing arrows as appropriate



Step 3 : Getting rid of the state 2 and replacing arrows as appropriate.

Step 4 : last step : getting rid of state 3 and replacing the last arrows. The resulting expression is the RE to be returned.



NON-REGULAR LANGUAGES

- To understand the power of finite automata you must also understand their limitations. In this section we show how to prove that certain languages cannot be recognized by any finite automaton.
- Let's take the language $B = \{0^n1^n \mid n \geq 0\}$. If we attempt to find a DFA that recognizes B, we discover that the machine seems to need to remember how many 0s have been seen so far as it reads the input.
- Because the number of 0s isn't limited, the machine will have to keep track of an unlimited number of possibilities. But it cannot do so with any finite number of states.
- Our technique for proving nonregularity stems from a theorem about regular languages, traditionally called the pumping lemma. This theorem states that all regular languages have a special property. If we can show that a language does not have this property, we are guaranteed that it is not regular. The 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. That means each such string contains a section that can be repeated any number of times with the resulting string remaining in the language.
- Pumping Lemma: If A is a regular language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:
  1. for each $i \geq 0$, $xy^iz \in A$
  2. $|y| > 0$
  3. $|xy| \leq p$

  We note that $y \neq \varepsilon$ while x or z can be ε
- To use the pumping lemma to prove that a language B is not regular, first assume that B is regular in order to obtain a contradiction. The use the pumping lemma to guarantee the existence of a pumping length p such that all strings of length p or greater in B can be pumped. Next, find a string s in B that has length p or greater but that cannot be pumped. Finally, demonstrate that s cannot be pumped by considering all ways of dividing s into x, y and z (taking condition 3 of the pumping lemma into account if convenient) and, for each such division, finding a value i where $xy^iz \notin B$. This final step often involves grouping the various ways of dividing s into several cases and analysing them individually. The existence of s contradicts the pumping lemma if B were regular. Hence B cannot be regular.
- Finding s sometimes takes a bit of creative thinking. You may need to hunt through several candidates for s before you discover one that works. Try members of B that seem to exhibit the "essence" of B's nonregularity.

Example 1
- Let $B=\{0^n1^n \mid n \geq 0\}$
- We suppose that B is regular
- let p be the pumping length
- We choose $s=0^p1^p$
- $s=xyz$ also $xy^iz \in B$. This is impossible because:
  - If y consists only of 0s, then $xyyz \notin B$ (more 0s than 1s)
  - If y consists only of 1s, then $xyyz \notin B$
  - If y consists of both 0s and 1s, then $xyyz \notin B$ (disorder of 0s and 1s in s)
- This B is not regular (contradiction)

Example 2
- $C=\{w \mid w$ has an equal number of 0s and 1s$\}$
- Suppose C is regular and p the pumping length
- let $s=0^p1^p$ then $s=xyz$ and also $xy^iz \in C$
- If we take $y=0^p1^p$ and x and z be empty then $xy^iz \in C$!
- BUT the condition 3 ($|xy| \leq p$) is not respected
- Thus y must consist only of 0s (or only of 1s).
- BUT $xyyz \notin C$ in this case
- Thus C is not regular

Example 3
- $D = \{a^{2^n} \mid n \geq 0\}$
- let's assume that D is regular
- Let s be string a^{2p} then s=xyz
- Third condition tell that $|xy| \leq p$. Furthermore $p < 2^p$ and so $|y| < 2^p$
- Therefore $|xyyz| = |xyz| + |y| < 2^p + 2^p = 2^{p+1}$
- The second condition requires $|y| > 1$ so $2^p < |xyyz| < 2^{p+1}$
- The length of xyyz cannot be a power of 2. Hence xyyz is not a member of D.
- Thus D is not regular

Example 4
- ε is described by the RE 0^*1^*
- The minimum pumping length is 1
- The pumping length cannot be 0 because ε is in the language
- Every non-empty string in the language can be divided into xyz wjere x = ε and y is the first character and z is the remainder.
- Hence is regular

Chap 2 : Context-Free languages

INTRODUCTION

In this chapter we introduce context – free grammars, a more powerful method, than finite automata and regular expressions, of describing languages. Such grammars can describe certain features that have a recursive structure which makes them useful in a variety of applications (study of human languages, compilation of prog. Languages).

CONTEXT-FREE GRAMMARS

- A grammar consists of a collection of substitution rules, also called productions. Each rule appears as a line in the grammar and comprises a symbol and a string, separated by an arrow. The symbol is called a variable. The string consists of variables and other symbols called terminals.
- You use a grammar to describe a language by generating each string of that language in the following manner.
  1. Write down the start variable. It is the variable on the left – hand side of the top rule, unless specified otherwise.
  2. Find a variable that is written down and a rule that starts with that variable. Replace the written down variable with the right – hand side of that rule.
  3. Repeat step 2 until no variables remain.
- All strings generated in this way constitute the language of the grammar. We write L(G) for the language of grammar G.
- Definition: A context – free grammar is a 4 – tuple (V,S, R, S), where
  1. V is a finite set called the variables.
  2. Σ is a finite set, disjoint from V, called terminals.
  3. R is a finite set of rules, with each rule being a variable and a string of variables and terminals.
  4. $S \in V$ is the start variable.
- We write $u \Rightarrow v$ if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v$
- Any language that can be generated by some context – free grammar is called a context – free language (CFL).
- The class of context-free languages is closed under union, concatenation and star.
- The class of context-free languages is NOT closes under intersection and concatenation

Example
- G=(V,E,R,S)
- Σ={that,this,a,the..man,book,flight,meal,include, read,does}
- V={S,NP,NOM,VP,Det,Noun,Verb,Aux}
- S=S

R={S	→ NP VP | Aux NP VP | VP	S → NP VP
NP	→ Det NOM	→ Det NOM VP
NOM	→ Noun | Noun NOM	→ The NOM VP
VP	→ Verb | Verb NP	→ The man VP
Det	→ that | this | a | the	→ The man Verb VP
Noun	→ book | flight | meal | man	→ The man read VP
Verb	→ book | include | read	→ The man read Det NOM
Aux	→ does	→ The man read this NOM
		→ The man read this Noun
		→ The man read this book

The language of the grammar is $\{w \in \Sigma^* \mid S \Rightarrow w\}$ }

DESIGNING CONTEXT-FREE GRAMMARS

You can convert any DFA into an equivalent CFG as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \rightarrow aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \rightarrow \varepsilon$ if q_i is an accept state of the DFA. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine. Verify on your own that the resulting CFG generates the same language that the DFA recognizes.

- If a grammar generates the same string in several different ways, we say that the string is derived ambiguously in that grammar. If a grammar generates some string ambiguously we say that the grammar is ambiguous.
- A derivation of a string w in a grammar G is leftmost derivation of at every step the leftmost remaining variable is the one replaced.
- A string w is derived ambiguously in context – free grammar G if it has two or more different leftmost derivations. Grammar G is ambiguous if it generates some string ambiguously.

CHOMSKY NORMAL FORM

- A context – free grammar is in Chomsky normal form, if every rule is of the form
  - $A \rightarrow BC$
  - $A \rightarrow a$
- where a is any terminal and A, B and C are any variables – except that B and C may not be the start variable. In addition we permit the rule $S \rightarrow \varepsilon$, where S is start variable.
- Any context – free language is generated by a context – free grammar in Chomsky normal form.

1. We add a new start variable S_0 and the rule $S_0 \rightarrow S$ (guarantees that the start variable occurs only on the left-hand side)
2. We remove all ε rules (in the form $A \rightarrow \varepsilon$ where $A \neq S$). If we find a rule in the form of $R \rightarrow uAv$ then for each occurrence of A we add a new rule $R \rightarrow uv$ (note that u and v are strings of variable terminals)
   - If we have $R \rightarrow uAvAw$ then we add $R \rightarrow uvAw$, $R \rightarrow uAvw$ and $R \rightarrow uvw$
   - If we have the rule $R \rightarrow A$, we add $R \rightarrow \varepsilon$ unless it has already be removed
3. We remove all unit rules in the form $A \rightarrow B$ and for each rule $B \rightarrow u$ we add the rule $A \rightarrow u$ unless this is a unit rule previously removed
4. We convert remaining rules $A \rightarrow u_1u_2...u_k$ where $k >= 3$ and u_i is a variable or terminal symbol to the rules $A \rightarrow u_1A_1$, $A_1 \rightarrow u_2A_2$, $A_2 \rightarrow u_3A_3$... and $A_{k-2} \rightarrow u_{k-1}u_k$. If $k = 2$ we replace in these rules and terminal u_i with a new variable U_i and the new rule $U_i \rightarrow u_i$.

Example :

	Add new start S_0	Remove ε rules (1)	Remove ε rules (2)	Remove unit rules(1)	Remove unit rules (2)	Remove unit rules(3)	Remove unit rules (4)	Proper form
$S \rightarrow ASA \mid aB$ $A \rightarrow B \mid S$ $B \rightarrow b \mid \varepsilon$	$S_0 \rightarrow S$ $S \rightarrow ASA \mid aB$ $A \rightarrow B \mid S$ $B \rightarrow b \mid \varepsilon$	$S_0 \rightarrow S$ $S \rightarrow ASA \mid aB \mid \mathbf{a}$ $A \rightarrow B \mid S \mid \boldsymbol{\varepsilon}$ $B \rightarrow b \cancel{\mid \boldsymbol{\varepsilon}}$	$S_0 \rightarrow S$ $S \rightarrow ASA \mid aB \mid a \mid \mathbf{SA} \mid \mathbf{AS} \mid \mathbf{S}$ $A \rightarrow B \mid S \cancel{\mid \boldsymbol{\varepsilon}}$ $B \rightarrow b$	$S_0 \rightarrow S$ $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS \cancel{\mid \mathbf{S}}$ $A \rightarrow B \mid S$ $B \rightarrow b$	$S_0 \rightarrow \cancel{\mathbf{S} \mid} \mathbf{ASA} \mid \mathbf{aB} \mid \mathbf{a} \mid \mathbf{SA} \mid \mathbf{AS}$ $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$ $A \rightarrow B \mid S$ $B \rightarrow b$	$S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS$ $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$ $A \rightarrow \cancel{\mathbf{B} \mid} S \mid \mathbf{b}$ $B \rightarrow b$	$S_0 \rightarrow ASA \mid aB \mid a \mid SA \mid AS$ $S \rightarrow ASA \mid aB \mid a \mid SA \mid AS$ $A \rightarrow \cancel{\mathbf{S} \mid} b \mid \mathbf{ASA} \mid \mathbf{aB} \mid \mathbf{a} \mid \mathbf{SA} \mid \mathbf{AS}$ $B \rightarrow b$	$S_0 \rightarrow AA_1 \mid UB \mid a \mid SA \mid AS$ $S \rightarrow AA_1 \mid UB \mid a \mid SA \mid AS$ $A \rightarrow S \mid b \mid AA_1 \mid UB \mid a \mid SA \mid AS$ $A_1 \rightarrow SA$ $U \rightarrow a$ $B \rightarrow b$

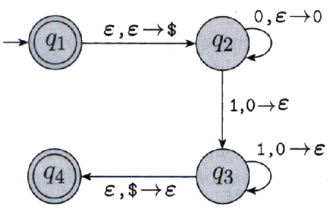
PUSDOWN AUTOMATA (PDA)

- These automata are like NFA but have an extra component called a stack. The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some Nonregular languages.
- Pushdown automata are equivalent in power to context – free grammars.
- A stack is valuable because it can hold an unlimited amount of information.
- The current state, the next input symbol read and the top symbol of the stack determine the next move of a pushdown automaton.
- Definition: A pushdown automaton is a 6 – tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ and F are all finite sets, and:

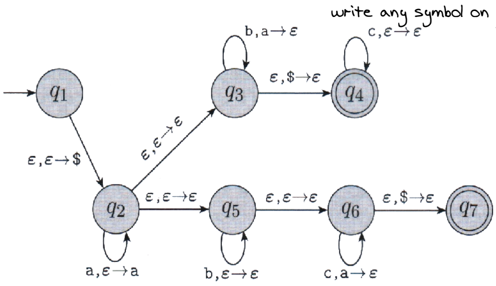
1. Q is the set of states.
2. Σ is the input alphabet.
3. Γ is the stack alphabet.
4. $\delta : Q_\varepsilon \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma_\varepsilon)$ is the transition function.
5. $q_0 \in Q$ is the start state.
6. $F \subseteq Q$ is the set of accept states.

We write $a, b \rightarrow c$ to signify that when the machine is reading an a from the input it may replace the symbol b on the top of the stack with a c. Any of a, b and c may be ε. If a is ε, the machine may make this transition without reading any symbol form the input. If b is ε, the machine may make this transition without reading and popping any symbol from the stack. If c is ε, the machine does not write any symbol on the stack when going along this transition.

Examples :

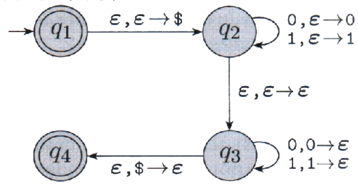


PDA M_1 that recognizes $\{0^n1^n \mid n \geq 0\}$



PDA M_2 that recognizes $\{a^ib^jc^k \mid i,j,k \geq 0$ and $i=j$ or $i=k\}$

See the nondeterminism here



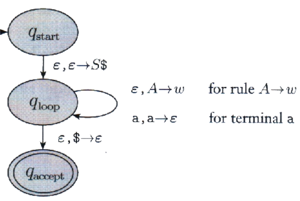
PDA M_3 that recognizes $\{ww^R \mid w \in \{0,1\}^*\}$ and w^R means w written backwards

EQUIVALENCE WITH CONTEXT-FREE GRAMMARS

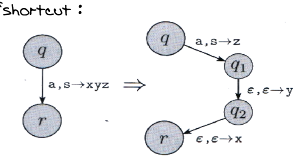
- A language is context free if and only if some pushdown automaton recognizes it.
- Every regular language is context – free.

Converting a CFG to a PDA, method :

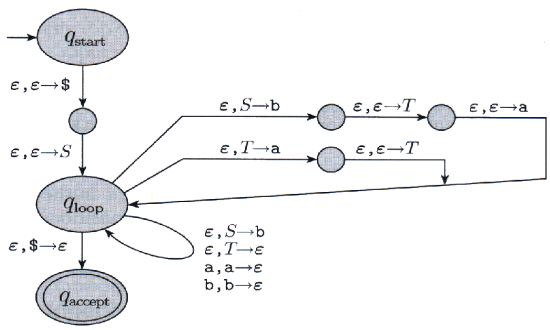
1. Place the marker symbol $ and the start variable on the stack
2. Repeat the following steps for ever :
   1. If the top of the stack is a variable symbol **A**, non-deterministically select one of the rules for **A** and substitute **A** by the string on the right-hand side.
   2. If the top of the stack is a terminal symbol **a**, read the next symbol from the input and compare it to **a**. If they match, repeat (consume it from the input and the stack). If they do not match, reject this branch of the non-determinism.
   3. If the top of the stack is the symbol **$**, enter the accept state. Doing so accepts the input IFF it has all been read



The method above assumes the following shortcut :



Example for the CFG : $S \rightarrow aTb \mid b$
$T \rightarrow Ta \mid \varepsilon$



Converting a PDA to a CFG, method :

- First we simplify our task by modifying P slightly in order to give the following three features
  1. It has a single accept state, q_{accept}
  2. It empties its stack before accepting
  3. Each transition either pushes a symbol onto the stack or pops one off the stack, but does not both at the same time
- To give it feature 3, we replace each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state, and replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol
- Say that P= $(Q, \Sigma, \Gamma, \delta, q_0, \{q_{accept}\})$ and construct G. The variables of G are $\{A_{pq} \mid p,q \in Q\}$. The start variable is $A_{q0,qaccept}$. Now we describe G's rules
- For each $p, q, r, s \in Q$, $t \in \Gamma$, and $a, b \in \Sigma_\varepsilon$, if $\delta(p, a, \varepsilon)$ contains (r, t) and $\delta(s, b, t)$ contains (q, ε), put the rule $A_{pq} \rightarrow aA_{rs}b$ in G
- For each $p, q, r \in Q$, put the rule $A_{pq} \rightarrow A_{pr}A_{rq}$ in G
- Finaly for each $p \in Q$, put the rule $A_{pp} \rightarrow \varepsilon$ in G

PUMPING LEMMA FOR CONTEXT-FREE LANGUAGES

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

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

We note that either $v \neq \varepsilon$ or $y \neq \varepsilon$

Let $B = \{a^nb^nc^n \mid n \geq 0\}$

- We suppose that B is a CFL let's try to find a contradiction
- B is a CFL then there is a p (the pumping length) where the selected string $s = a^pb^pc^p$ is in B. The pumping lemma stated that s can be pumped, but we show that it cannot.
- B= uvxyz. Condition 2 states that either v or y are non empty. Then we consider one of the two cases:
  - When both v or y contain only one type of alphabet symbol; v (the same for y) does not contain both a's and b's or both b's and c's thus uv^2xy^2z cannot contain equal number of a's, b's and c's
  - When either v or y contain more than one type of symbol, uv2xy2z may contain equal numbers of symbols

Let $C = \{ww \mid w \in \{0,1\}^*\}$

- We suppose that C is a CFL. Let p be the pumping length given by the pumping lemma. We show that the string $x = 0^p1^p0^p1^p$ cannot be pumped. The string s can be divided into s=uvxyz.
  - vxy can neither be at the beginning of the string s or at the end because s is symmetric and hence uv^2xy^2z cannot be pumped.
  - vxy straddles in the midpoint of s: $0^p1^{p-1}\underline{10}0^{p-1}1^p$ in this case, uv^2xy^2z is not in C.

but not in the correct order. Hence it cannot be a member of B. Contradiction

- We get a contradiction with the pumping lemma, thus C is not a CFL

Chap 3 : The Church-Turing thesis

TURING MACHINES

- Similar to a finite automaton but with an unlimited and unrestricted memory, a Turing machine is a much more accurate model of a general purpose computer. A Turing machine can do everything that a real computer can do. Nonetheless, even a Turing machine cannot solve certain problems. In a very real sense, these problems are beyond the theoretical limits of computation.
- The following list summarizes the differences between finite automata and Turning machines:
  1. A Turing machine can both write on the tape and read from it.
  2. The read – write head can move both to the left and to the right.
  3. The tape is infinite.
  4. The special states for rejecting and accepting take immediate effect.
- In actuality we almost never give formal descriptions of Turing machines because they tend to be very big.
- Definition: A Turing machine is a 7 – tuple $(Q, \Sigma, \Gamma, d, q_0, q_{accept}, q_{reject})$, where Q,Σ,Γ are all finite sets and:
  1. Q is the set of states.
  2. Σ is the input alphabet not containing the special blank symbol _
  3. Γ is the tape alphabet, where $\_\in\Gamma$ and $\Sigma\subseteq\Gamma$
  4. $d : Q\times\Gamma\rightarrow Q\times\Gamma\times\{L, R\}$ is the transition function
  5. $q_0\in Q$ is the start state
  6. $q_{accept}\in Q$ is the accept state
  7. $q_{reject}\in Q$ is the reject state, where $q_{reject}\neq q_{accept}$

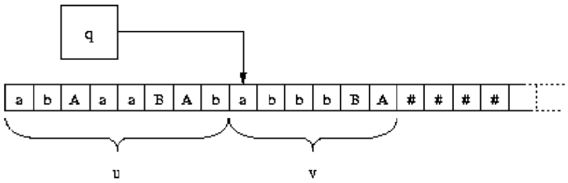
- For a Turing machine, d takes the form: $d : Q\times\Gamma\rightarrow Q\times\Gamma\times\{L, R\}$. That is, when the machine is in a certain state $q1$ and the head is over a tape square containing the symbol a, and if $d(q1, a) = (q2, b, L)$, the machine writes the symbol b replacing the a, and goes to state $q2$
- For a Turing machine, δ takes the form: $\delta : Q\times\Gamma\rightarrow Q\times\Gamma\times\{L, R\}$. That is, when the machine is in a certain state q_1 and the head is over a tape square containing a symbol a, and if $\delta(q_1, a) = (q_2, b, L)$, the machine writes the symbol b replacing the a, and goes to state q_2
- The collection of strings that M accepts is the language of M, denoted L(M).
- Call a language Turing – recognizable if some Turing machine recognizes it.
- When we start a TM on an input, three outcomes are possible. The machine may accept, reject, or loop. By loop we mean that the machine simply does not halt. It is not necessarily repeating the same steps in the same way forever as the connotation of looping may suggest. Looping may entail any simple or complex behaviour that never leads to a halting state.
- We prefer Turing machines that halt on all inputs; such machines never loop. These machines are called deciders because they always make a decision to accept or reject. A decider that recognizes some language also is said to decide that language.
- Call a language Turing – decidable or simply decidable if some Turing machine decides it.
- Every decidable language is Turing – recognizable but certain Turing – recognizable languages are not decidable.

CONFIGURATIONS OF TM

- Initially M receives its input $w = w_1 w_2 \ldots w_n \in\Sigma^*$ on the leftmost n squares of the tape, and the rest of the tape is blank.
- As a Turing machine computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a configuration of the Turing machine.
- A configuration is represented as uqv where u and v in Γ* and q in Q
  - The current state is q
  - The current tape content is uv
  - The current head of the tape is the first symbol of v
- Suppose that a, b, and c in Γ, and u and v in Γ*.
  - uaqibv yields uqjacv if δ(qi , b)=(qj, c, L)
  - uaqibv yields uacqjv if δ(qi , b)=(qj, c, R)
  - qibv yields qjcv if δ(qi , b)=(qj, c, L) (head on the left hand end)
  - qibv yields cqjv if δ(q1 , b)=(q2, c, R) (head on the left hand end)
  - uaqi is equivalent to uaqi Ñ if the head is on the right hand end

- A Turing machine M accepts input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists where
  1. C_1 is the start configuration of M on input w
  2. Each C_i yields C_{i+1}
  3. C_k is an accepting configuration.

Configuration of abAaaBAbqabbbBA :



Properties of configurations :

- The start configuration is q_0w
- In accepting configuration the state is q_{accept}
- In rejecting configuration the state is q_{reject}
- Accepting and rejecting configurations are halting configurations

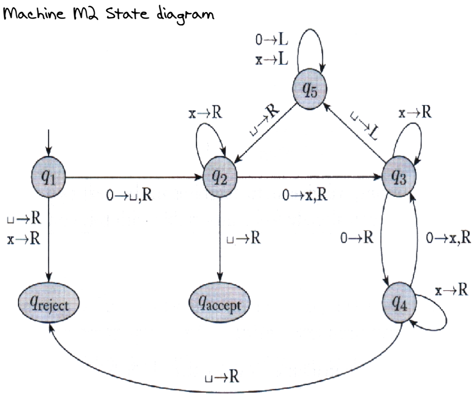
- A Turing machine M accepts input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists where
  - C_1 is the start configuration of M on input w
  - Each C_i yields C_{i+1}, and
  - C_k is an accepting configuration

EXAMPLES OF TURING MACHINES

M2 decides $A = \{0^{2^n} \mid n\geq 0\}$

M2 = "On input string w:
1. Sweep left to right across the tape, crossing off every other 0
2. If in stage 1 the tape contained a single 0, accept.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, reject.
4. Return the head to the left-hand end of the tape.
5. Go to stage 1."

Machine M2 State diagram



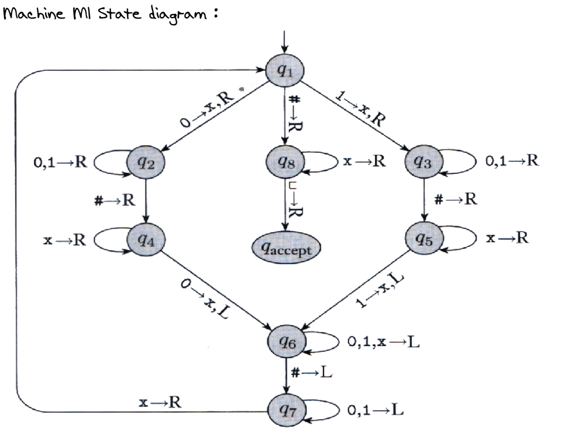
M3 decides $C = \{a^ib^jc^k \mid i * j = k \text{ and } i,j,k\geq 1\}$

M3 = "On input string w:
1. Scan the input from left to right to determine whether it is a member of a+b+c+ and reject if it is not.
2. Return the head to the left-hand end of the tape.
3. Cross off an a and scan to the right until a b occurs. Shuttle between the b's and c's, crossing off one of each until all b's are gone. If all c's have been crossed off and some b's remain, reject.
4. Restore all the crossed off b's and repeat stage 3 if there is another a to cross off. If all a's have been crossed of, determine whether all c's also have been crossed off. If yes, accept; otherwise reject.

Machine M2 configuration example for input 0000

$q_1 0000$	$\sqcup q_5 x 0 x \sqcup$	$\sqcup x q_5 x x \sqcup$
$\sqcup q_2 000$	$q_5 \sqcup x 0 x \sqcup$	$\sqcup q_5 x x x \sqcup$
$\sqcup x q_3 00$	$\sqcup q_2 x 0 x \sqcup$	$q_5 \sqcup x x x \sqcup$
$\sqcup x 0 q_4 0$	$\sqcup x q_2 0 x \sqcup$	$\sqcup q_2 x x x \sqcup$
$\sqcup x 0 x q_3 \sqcup$	$\sqcup x x q_3 x \sqcup$	$\sqcup x q_2 x x \sqcup$
$\sqcup x 0 q_5 x \sqcup$	$\sqcup x x x q_3 \sqcup$	$\sqcup x x q_2 x \sqcup$
$\sqcup x q_5 0 x \sqcup$	$\sqcup x x q_5 x \sqcup$	$\sqcup x x x q_2 \sqcup$
		$\sqcup x x x \sqcup q_{accept}$

Machine M1 State diagram :



M1 decides $B = \{w\#w \mid w\in\{0,1\}^*\}$

M1 = "On input string w:
1. Zig-Zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.
2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, reject; otherwise, accept."

PROPERTIES OF TURING-X'ABLE LANGUAGES

The collection of Turing-decidable languages is closed under union, concatenation, start, complementation and intersection

The collection of Turing-recognizable languages is closed under union, concatenation, start and intersection (NOT complementation)

VARIANTS OF TURING MACHINES

- The original TM model and its reasonable variants all have the same power – they recognize the same class of languages.
- To show that two models are equivalent we simply need to show that we can simulate one by the other.
- A multitape TM is like an ordinary Turing machine with several tapes. Each tape has its own head for reading and writing. Initially the input appears on tape 1, and the others start out blank.
- Two machines are equivalent if they recognize the same language.
- Every multitape Turing machine has an equivalent single tape Turing machine.
- A language is Turing – recognizable if and only if some multitape Turing machine recognizes it.
- A nondeterministic Turing machine is defined in the expected way. At any point in a computation the machine may proceed according to several possibilities. The transition

- The computation of a nondeterministic Turing machine is a tree whose branches correspond to different possibilities for the machine. (If you want to simulate a nondeterministic TM with a "normal" TM you have to perform a breadth – first search through the tree, because with depth – first you can lose yourself in a infinite branch of the tree and miss the same accept state). If some branch of the computation leads to the accept state, the machine accepts its input.
- Every nondeterministic Turing machine has an equivalent deterministic Turing machine.
- A language is Turing – recognizable if and only if some nondeterministic Turing machine recognizes it.
- We call a nondeterministic Turing machine a decider if all branches halt on all inputs.
- A language is decidable if and only if some nondeterministic TM decides it.
- Loosely defined, an enumerator is a Turing machine with an attached printer.
- A language is Turing – recognizable if and only if some enumerator enumerates it.

function for a nondeterministic Turing machine has the form: $\delta : Q \times \Gamma \rightarrow P(Q \times \Gamma \times \{L, R\})$

THE CHOMSKY HIERACHY of LANGUAGES

Grammar	Languages	Autoamaton	Productions	Grammar	Languages	Autoamaton	Productions
Type-0	Turing -recognizable (Recursively enumerable)	Turing machine	$\alpha \rightarrow \beta$ (no restrictions)	Type-2	Context-free	(Non-deterministic) Pushdown automaton	$A \rightarrow \gamma$
Type-1	Context-sensitive	(LBA) Linear-Bounded non deterministic Turing machine	$A A\beta \rightarrow \alpha\gamma\beta$	Type-4	Regular	Finite State Automaton	$A \rightarrow a$ $A \rightarrow aB$

THE DEFINITION OF AN ALGORITHM

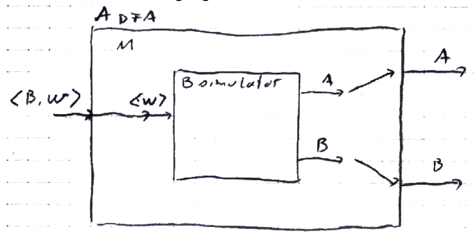
- Informally, an algorithm is a collection of simple instructions for carrying out some task.
- Alonzo Church used a notational system called the λ- calculus to define algorithms. Turing did it with his "machines". These two definitions were shown to be equivalent. This connection between the informal notion of algorithm and the precise definition has come to be called the Church – Turing thesis
- The Church–Turing thesis: Intuitive notion of algorithm is equal to Turing machine algorithms.
- Our notation for the encoding of an object O into its representation as a string is <O>. If we have several objects $O_1, O_2, \ldots, O_k$ we denote their encoding into a single string by $<O_1, O_2, \ldots, O_k>$.
- An algorithm always stops.

Chap 4 : Decidability

DECIDABLE LANGUAGES

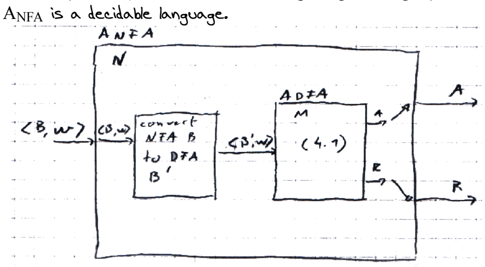
Acceptance problem expressed as languages for regular expressions:

$A_{DFA} = \{<B,w> \mid B$ is a DFA that accepts input string $w\}$
A_{DFA} is a decidable language.



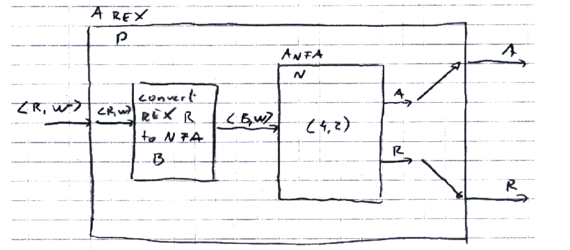
TM M decides A_{DFA}. The input of M is a pair <B,w> where B is a DFA and w is a string.
1. M simulates the DFA B on the input w.
2. If the simulation ends in an accept state, accept; otherwise, reject.

$A_{NFA} = \{<B,w> \mid B$ is an NFA that accepts input string $w\}$
A_{NFA} is a decidable language.



TM N decides A_{NFA} and behaves as follows on an input <B,w> where B is a DFA and w is a string.
1. N converts the NFA B into equivalent DFA C
2. N runs the TM M built for A_{DFA}

$A_{REX} = \{<R,w> \mid R$ is a regular expression that generates string $w\}$
A_{REX} is a decidable language.

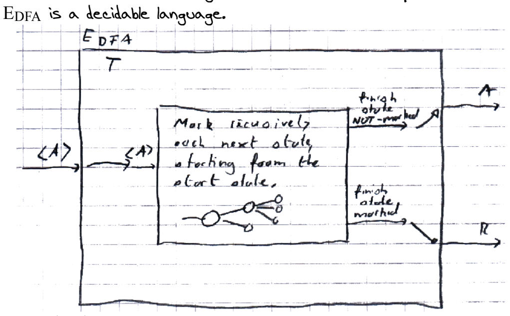


TM P decides A_{REX} and behaves as follows on an input <R,w> where R is a regular expression and w is a string.
1. N converts the reg exp R into equivalent NFA B
2. N runs the TM M built for A_{NFA}

The problem of testing whether a DFA B accepts an input w is the same as the problem of testing whether <B,w> is a member of the language DFA A. Similarly, we can formulate other computational problems in term of testing membership in a language. Showing that the language is decidable is the same as showing that the computational problem is decidable.

Emptiness testing for regular expressions:

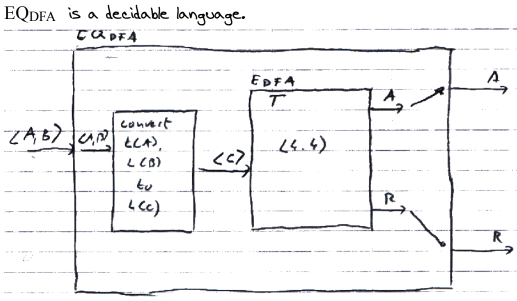
$E_{DFA} = \{<A> \mid A$ is a DFA and $L(A) = \varnothing\}$
This means, that no string exists that DFA A accepts.
E_{DFA} is a decidable language.



TM T decides E_{DFA}. The input of T is a singleton <A> where A is a DFA.
1. Mark recursively each next state, starting from the start state
2. If the finish state is NOT marked, accept; otherwise, reject.

The next theorem states that testing whether two DFAs **recognize the same language is decidable:**

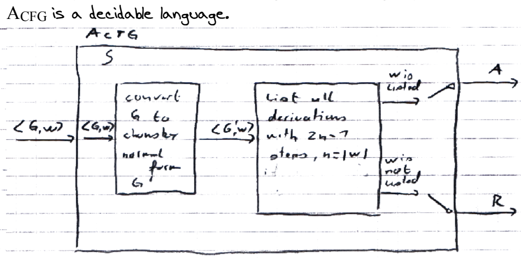
$EQ_{DFA} = \{<A, B> \mid A$ and B are DFAs and $L(A) = L(B)\}$
EQ_{DFA} is a decidable language.



TM Q decides EQ_{DFA} and behaves as follows on an input <A,B> where both A and B are DFA's:
1. I construct a DFA C with language L(C) defined as: $L(C) = (L(A) \cap \overline{L(B)}) \cup (\overline{L(A)} \cap L(B))$
2. N runs the TM M built for E_{DFA}

Acceptance problem expressed as languages for context – free languages:

$A_{CFG} = \{< G,w > \mid G$ is a CFG that generates string $w\}$
A_{CFG} is a decidable language.



TM S for A_{CFG} behaves as follows for an input <G,w> where G is a CFG and w is a string:
1. Convert G in an equivalent grammar in Chomsky normal form.
2. List all derivations with $2m - 1$ steps, where n is the length of w, except if $n = 0$, then instead list all derivations with 1 step
3. If any of them generates w, accept; if not, reject;

Language inclusion testing for regular language

$INC_{DFA} = \{<A,B> \mid A$ and B are DFAs and $L(B) \subseteq L(A)\}$ / INC_{DFA} is decidable
Idea : Show that $L(B) \cap L(A) = L(B)$
Build DFA C that accepts L(C) and use machine built for EQ_{DFA}

Emptiness testing for context – free grammars:

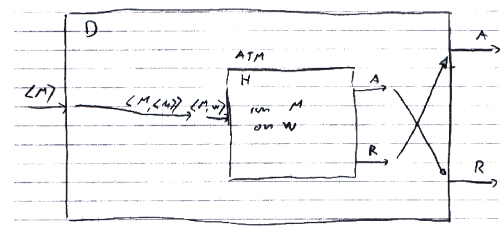
$E_{CFG} = \{<G> \mid G$ is a CFG and $L(G) = \varnothing\}$
E_{CFG} is a decidable language.
TM R decides E_{CFG}. On input <G> where G is a CFG, G does :
1. Mark all terminal symbols on G
2. Repeat step 3 until no new variable get marked :
3. Mark any variable A where G has a rule $A \rightarrow U_1U_2\ldots U_k$ and each symbol $U_1U_2\ldots U_k$ has already been marked.
4. If the start variable is not marked, accept; otherwise reject.

Equivalence testing for context – free grammars:

$EQ_{CFG} = \{< G,H > \mid G$ and H are CFLs and $L(G) = L(H)\}$
EQ_{CFG} is NOT decidable !!
Proving EQ_{CFG} the way we proved EQ_{DFA} is not feasible because ContextFree languages are NOT closed under intersection and complementation.
We will see later the technique used to prove that EQ_{CFG} is not decidable.

Universal language for regular language

$ALL_{DFA} = \{<A> \mid A$ is a DFA and $L(A) = \Sigma^*\}$ / ALL_{DFA} is a decidable language
Idea : Consider $\overline{ALL_{DFA}} = \{<A> \mid A$ is a DFA and $L(A) = \varnothing\}$ that can be decided using E_{DFA}
Then the complement of a decidable language is decidable.



The Halting problem

$A_{TM} = \{< M,w > \mid M$ is a TM and M accepts $w\}$
Diagonalization :

	<M1>	<M2>	<M3>	...	<D>
M1	A	R	R	...	R
M2	R	R	A	...	A
M2	A	A	R	...	A
D	R	A	A	...	?

A_{TM} is undecidable but it is Turing-recognizable hence A_{TM} is sometimes called the halting problem.

Suppose H is a decided for ATM. On input <M,w> where M is a TM and w is a string, H halts and accepts if M accepts w. Machine D has H as subroutine. D calls H to determine what it does and does the contrary. Here D cannot the contrary as itself, hence we have a contradiction, Hence A_{TM} is not decidable (See Diagonalization above)

TURING DECIDABLE / TURING RECOGNIZABLE

- Cantor observed that two finite sets have the same size if the elements of one set can be paired with the elements of the other set.
- Assume A and B are two (infinite) sets. If then exists a bijektive function f between the two sets, they have the same size.
- A set B is countable if either it is finite or it has the same size as the natural numbers N.
- Q (rational numbers) and N have the same size.
- R (real numbers) is uncountable.
- It shows that some languages are not decidable or even Turing – recognizable, for the reason that there are uncountable many languages yet only countably many Turing machines. Because each Turing machine can recognize a single language and there are more languages than Turing machines, some languages are not recognizable by any Turing machine.
- Some languages are not Turing – recognizable.
- The following theorem shows that, if both a language and its complement are Truing – recognizable, the language is decidable. Hence, for any undecidable language, either it or its complement is not Truing – recognizable. We say that a language is co – Turing – recognizable if it is the complement of a Turing – recognizable language.
- A language is decidable ⇔ it is both Turing – recognizable and co – Turing – recognizable.
- A language is decidable ⇔ its complement is decidable.
- A_{TM} is not Turing – recognizable.

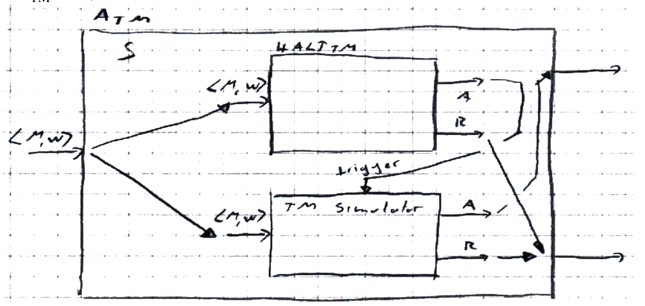
Chap 5 : Reducibility

UNDECIDABLE PROBLEMS FROM LANGUAGE THEORY

- In this chapter we examine several additional unsolvable problems. In doing so we introduce the primary method for proving that problems are computationally unsolvable. It is called reducibility.
- A reduction is a way of converting one problem into another problem I such a way that a solution to the second problem can be used to solve the first problem.
- When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives a solution to A. In terms of computability theory, if A is reducible to B and B is decidable, A also is decidable. Equivalently, if A is undecidable and reducible to B, B is undecidable. This last version is key to proving that various problems are undecidable.
- Rice's theorem: Testing any property of the languages recognized by a TM is undecidable.

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

$HALT_{TM}$ is undecidable



We operate a reduction from A_{TM} to $HALT_{TM}$. We build a TM S deciding A_{TM} as follows, on input <M,w> where M is a TM and w a string :

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 accepted, accept; otherwise, reject.

$E_{TM} = \{<M> \mid M \text{ is a TM and } L(M) = \varnothing\}$

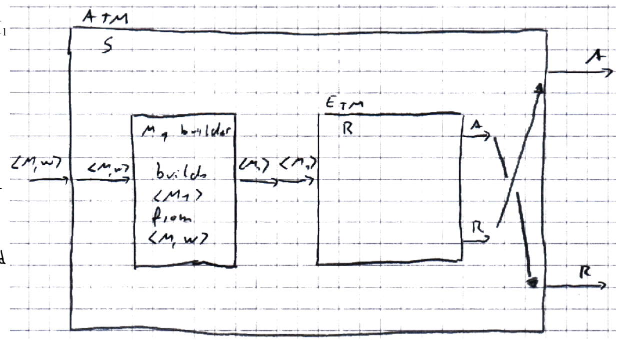
E_{TM} is undecidable.

We need as special machine M_1 which on input x, behaves as follows.

1. If $x \neq w$, reject.
2. If $x = w$, run M on input w and accept if M does.

Assuming that a TM R decides E_{TM}, we construct TM S that decides A_{TM}. On input <M,w>, S behaves as follows.

1. Use the description of M and w to construct M_1.
2. Run R on input $<M_1>$.
3. If R accepts, reject; if R reject, accept



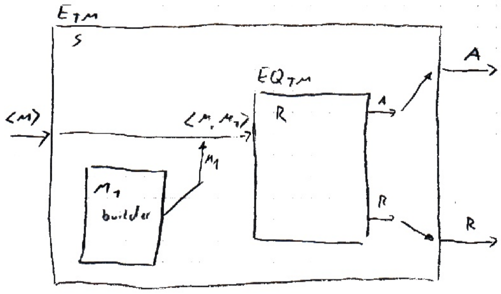
According the above construction, we observe that S decides A_{TM}, which is a contradiction. Hence, E_{TM} is undecidable.

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

EQ_{TM} is undecidable.

We assume that TM R decides EQ_{TM} and we construct TM M_1 that rejects all inputs (ie. it accepts the empty language $\varnothing$). We construct a decider S that runs R on $<M,M_1>$ where M is the input of E_{TM}. If R accepts, accept; otherwise, rejects.

=> Contrad =>Thus EQ_{TM} not decidable



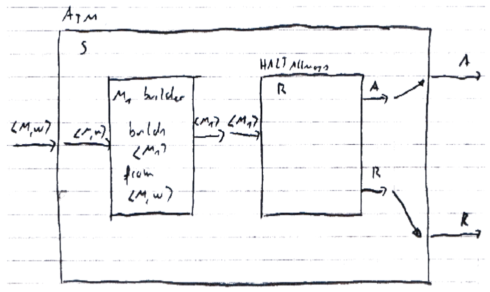
$HALT_{ALLWAYS} = \{<M> \mid M \text{ is a TM that halts on any input}\}$

M_1 builder build M_1, M_1 on input X

- if $x \neq w$, reject
- if $x = w$, run M on w and accept if M accepts w (if M loops or rejects, undefined)

Hence

- if M accepts w => M_1 always halts
- If M does not accept w => M_1 does not always halt



Notes on E_{TM}:

$E_{TM} = \{<M> \mid M \text{ is a TM and } L(M) = \varnothing\}$

$\overline{E_{TM}} = \{<M> \mid M \text{ is a TM and } L(M) \neq \varnothing\}$

$\overline{E_{TM}}$ is Turing-recognizable (idea : we build an enumerator for Σ^* and we give all the words it builds to M. We stop when a word is finally accepted)

Hence E_{TM} is not turing-recognizable, see proof below :

MAPPING REDUCIBILITY

- Roughly speaking, being able to reduce problem A to problem by using a mapping reducibility means that a computable function exists that converts instances of problem A to instances of problem B. If we have such a conversion function, called a reduction, we can solve A with a solver for B.
- A function $f : \Sigma^* \rightarrow \Sigma^*$ is a computable function if some Turing machine M, on every input w, halts with just f(w) on its tape.
- Language A is mapping reducible to language B, written $A \leq_m B$, if there is a computable function $f : \Sigma^* \rightarrow \Sigma^*$, where for every w, $w \in A \Leftrightarrow f(w) \in B$
- The notion of mapping reduction is slightly different from the reduction we have been doing before :
- The function f is called the reduction of A to B.
- If $A \leq_m B$ and B is decidable, then A is decidable.
- If $A \leq_m B$ and A is undecidable, then B is undecidable.
- If $A \leq_m B$ and B is Turing – recognizable, then A is Turing – recognizable.
- If $A \leq_m B$ and A is not Turing – recognizable, then B is not Turing – recognizable.
- If $A \leq_m B$ then $\overline{A} \leq_m \overline{B}$ with the very same function.

EQTM and $\overline{EQTM}$ are not Turing-recognizable

To prove that B is not Turing-recognizable we may show that $A_{TM} \leq_m B$.

First we show that EQ_{TM} is not Turing-recognizable by means of a reduction from A_{TM} to $\overline{EQ_{TM}}$.

To show that $\overline{EQ_{TM}}$ is not Turing-recognizable we provide a reduction from A_{TM} to the complement of $\overline{EQ_{TM}}$, namely EQ_{TM}.

Chap 6 : Time Complexity

MEASURING COMPLEXITY

- Even when a problem is decidable and thus computationally solvable in principle, it may not be solvable in practice if the solution requires an inordinate amount of time or memory. In this final part of the book we introduce computational complexity theory – an investigation of the time, memory, or other resources required for solving computational problems.
- For simplicity we compute the running time of an algorithm purely as a function of the length of the string representing the input and don't consider any other parameters. In worst – case analysis, the form we consider here, we consider the longest running time of all inputs of a particular length.
- Definition:
  Let M be a deterministic Turing machine that halts on all inputs. The running time or time complexity of M is the function $f : N \rightarrow N$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine.
- Because the exact running time of an algorithm often is a complex expression, we usually just estimate is. In one convenient form of estimation, called asymptotic analysis, we seek to understand the running time of the algorithm when it is run on large inputs.
- Definition: (Big-O notation)
  o Let f and g be to functions f, $g : N \rightarrow R+$. Say that $f(n) = O(g(n))$ if positive integers c and n0 exist so that for every integer $n \geq n0$ then $f(n) \leq c \cdot g(n)$
  o When $f(n) = O(g(n))$ we say that g(n) is an upper bound for f(n), or more precisely, that g(n) is an asymptotic upper bound for f(n), to emphasize that we are suppressing constant factors.
- Frequently we derive bounds of the form n^c for c greater than 0. Such bounds are called polynomial bounds. Bounds of the form $2^{(n^\delta)}$ are called exponential bounds when δ is a real number greater than 0.
- Big – O notation has a companion called small – o notation. Big – O notation gives a way to say that one function is asymptotically no more than another. To say that one function is asymptotically less than another we use small – o notation. The difference between the big – O and small – o notation is analogous to the difference between $\leq$ and $<$.
- Definition: (Small-o notation)
  o Let f and g be two functions $f, g : N \rightarrow R^+$. Say that $f(n) = o(g(n))$ if $\lim_{n\rightarrow\infty}(f(n) / g(n)) = 0$
  o In other words, $f(n) = o(g(n))$ means that, for any real number $\chi > 0$, a number n_0 exists, where $f(n) < c \cdot g(n)$ for all $n \geq n_0$.
- Definition: (Time complexity class)
  o Let $t : N \rightarrow N$ be a function. Define the time complexity class, TIME(t(n)), to be $TIME(t(n)) = \{ L \mid L \text{ is a language decided by an } O(t(n)) \text{ time Turing machine}\}$
- Any language that can be decided in $o(n \cdot \log n)$ time on a single – tape Turing machine is regular.
- This discussion highlights an important difference between complexity theory and computably theory. In computability theory, the Church – Turing thesis implies that all reasonable models of computation are equivalent, that is, they all decide the same class of languages. In complexity theory, the choice of the model affects the time complexity of languages.

COMPLEXITY RELATIONSHIPS AMONG MODELS

- Let $t(n)$ be a function, where $t(n) \geq n$. Then every t(n) time multitape Turing machine has an equivalent $O(t^2(n))$ time single – tape Turing machine.
- Definition: Let N be a nondeterministic Turing machine that is a decider. The running time of N is the function $f : N \rightarrow N$, where f(n) is the maximum number of steps that N uses on any branch f its computation on any input of length n.
- Let t(n) be a function, where $t(n) \geq n$. Then every t(n) time nondeterministic single – tape Turing machine has an equivalent $2^{O(t(n))}$ time deterministic single tape Turing machine.

THE CLASS P

- Exponential time algorithms typically arise when we solve problems by searching through a space of solutions, called brute – force search.
- P is the class of languages that are decidable in polynomial time on a deterministic single – tape Turing machine. In other words: $P = \bigcup_k TIME(n^k)$

The class P plays a central role in our theory and is important because

- P is invariant for all models of computation that are polynomially equivalent to the deterministic single – tape Turing machine.
- P roughly corresponds to the class of problems that are realistically solvable on a

- Every context – free language is a member of P
- P is closed under union, concatenation and complement

Examples of problems in P :

- PATH = {<G,s,t> | G dir. graph with dir. Path from s to t} is in P
  A polynomial time algorithm for PATH is (TM M with input <G,s,t> behaves as
  1. Place a mark on node s
  2. Repeat the stage 3 until no additional nodes are marked :
  3. Scan all edges of G. If an edge (a,b) is found going from a marked node a to an unmarked node b, mark b
  4. If it is marked, accept. Otherwise, reject.

computer.

- RELPRIME = {<x,y>| x and y are relatively prime} is in P
- CONNECTED = {<G> | G is a connected undirected graph} is in P
  Proof idea : Same demonstration as PATH but with an additional step looking for unmarked nodes.
- TRIANGLE (3-CLIQUE), 4-CLIQUE and 5-CLIQUE are in P. But Caution : k-CLIQUE (n > 5) **is in NP !!!** Proof idea for triangle : Select all sub-sets of 3 vertices and check if they are connected. Graphe of k on n => $O(n! / (n-k)! * k!) = O(n^3)$

THE CLASS NP

- NP means Non-Deterministic Polynomial
- Hamilton – Path: HAMPATH = {<G,s,t> | G is a directed graph with a Hamiltonian path from s to t}
- The HAMPATH problem does have a feature called polynomial verifiability.
- Some problems may not be polynomial verifiable. For example, take $\overline{\text{HAMPATH}}$, the complement of the HAMPATH problem. Even if we could determine (somehow) that a graph did not have a Hamiltonian path, we don't know of a way for someone else to verify its non-existence without using the same exponential time algorithm for making the determination in the first place.
- A verifier for a language A is an algorithm V, where A = {w | V accepts <w,c> for some string c}
- We measure the time of a verifier only in terms of the length of w, so a polynomial time verifier runs in polynomial time in the length of w.
- NP is the class of languages that have polynomial time verifiers.
- A verifier uses additional information, represented by the symbol c, to verify that a string w is a member of A. This information is called a certificate, or proof, of membership in A. Observe that, for polynomial verifiers, the certificate has polynomial length (in the length of w) because that is all the verifier can access in its time bound.
- A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.
- NTIME(t(n)) = {L | L is decided by a O(t(n)) time nondeterministic Turing machine}
- $NP = U_k NTIME(n^k)$
- P is the class of languages that are decidable in polynomial time on a non-deterministic Turing machine.
- The best method known for solving problems in NP deterministically uses exponential time algorithm : $NP \subseteq EXPTIME = U_k TIME(2^{n^k})$

Examples of problems in NP :

CLIQUE = { <G,k> | G is an undirected graph with a k – clique } / CLIQUE is in NP.

Verifier for CLIQUE : On input <<G,k>,c>
1. Test whether c is a set of nodes in G
2. Test whether G contains all edges connecting nodes in c
3. If both pass, accept; otherwise reject.

OR we construct a non-deterministic polynomial TM that decides CLIQUE : On input <G,k> :
1. Nondeterministically select a subset c of k nodes of G
2. test whether G contains all edges connecting nodes in c
3. if yes, accept; otherwise reject.

SUBSET-SUM = { <S,t> | S = {$x_1,...,x_k$} and for some {$y_1,...,y_j$} ⊆ {$x_1,...,x_k$}, we have $\Sigma y_i = t$}
SUBSET-SUM is in NP.

Verifier for SUBSET-SUM: : On input <<S,t>,c>
1. Test whether c is a collection of numbers that sum to t
2. Test whether S contains all numbers in c
3. If both path, accept; otherwise reject.

ISO = {<G, H> | G and H are isomoprhic graphs} is in NP. Proof idea :
1. Check the numbers of edges and vertices are the same
2. Select a bijection of the edges of both graphs non-deterministically
3. Check whether they have the same connections

Notes on NP :

P versus NP

- P = class of languages for which membership can be decided quickly
- NP = class of languages for which membership can be verified quickly
- We don't know if P = NP (one of the greatest unsolved problem). We have not been able to prove that a language in NP is not in P. We *think* however that P ≠ NP
- Obviously P ⊆ NP since deterministic TMs are special cases of non-deterministic TMs

NP and Co-NP

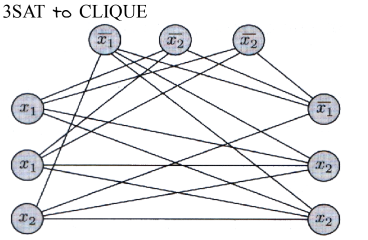
- The class co-NP contains the languages for which the complement is in NP
- $\overline{\text{HAMPATH}}$, $\overline{\text{CLIQUE}}$ and $\overline{\text{SUBSET-SUM}}$ are not obviously also in NP. Verifying that something is not present seems to be more difficult than verifying that it is present.
- We believe that NP ≠ co-NP, but clearly P = co-P.

NP-COMPLETENESS

- SAT = { <Φ> | Φ is a satisfiable Boolean formula }
- Cook – Levin theorem: SAT ∈ P ⇔ P = NP
- We have already defined the concept of reducing one problem to another. When problem A reduces to problem B, a solution to B can be used to solve A. Now we define a version of reducibility that takes the efficiency of computation into account. When problem A is efficiently reducible to problem B, an efficient solution to B can be used to solve A efficiently.
- Definition: A function f : Σ*→Σ* is a polynomial time computable function if some polynomial time Turing machine M exists that halts with just f(w) on its tape, when started on any input w.
- Definition: language A is polynomial time mapping reducible, or simply polynomial time reducible, to language B, written $A \leq_p B$, if a polynomial time computable function f : Σ*→Σ* exists, where for every w, w∈ A⇔ f (w)∈ B
  The function f is called the polynomial time reduction of A to B. If $A \leq_p B$ and B∈ P, then A∈ P.
- 3SAT = { <Φ> | Φ is a satisfiable 3-CNF-formula}
- 3SAT is polynomial time reducible to CLIQUE. This means, if CLIQUE is solvable in polynomial time, so is 3SAT.
- Definition: A language B is NP – complete if it is satisfies two conditions:
  1. B is in NP
  2. Every A in NP is polynomial time reducible to B.
- If B is NP – complete and B∈ P , then P = NP. No one has been found so far.
- If B is NP – complete and $B \leq_p C$ for C in NP, then C is NP – complete.
- SAT is NP – complete. (Cook-Levin theorem)
- 3SAT is NP – complete.

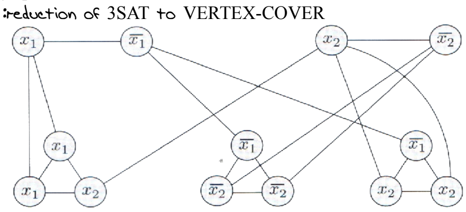
- CLIQUE is NP – complete. Proof idea :for the reduction of 3SAT to CLIQUE

The nodes of G are organized in *k* groups of three nodes each called triples. Each triple corresponds to one of the clauses in Φ, and each note in a triple correspond to a literal in the clause. Label each node of G with its corresponding literal in Φ The edges of G connect all but 2 types of pairs of nodes : No edge is present between nodes in the same triple and no edge is present between 2 nodes with contradictory labels, as in X2 and $\overline{\text{X2}}$. The the clause is satisfiable if the graph has a k-CLIQUE.



- If G is an undirected graph, a vertex cover of G is a subset of the nodes where every edge of G touches one of those nodes. The vertex cover problem asks for the size of the smallest vertex cover.
- VERTEX-COVER = { <G,k> | G is an undirected graph that has a k – node vertex cover }
- VERTEX-COVER is NP – complete. Proof idea :reduction of 3SAT to VERTEX-COVER

1. Each variable x in Φ produces two nodes labelled x and $\overline{x}$. 2. Each clause produces three nodes labelled as the literal, which are all connected together. 3. Each node x produced in 1. is connected to all nodes in G. 4. Each node $\overline{x}$ produced in 1. is connected to all nodes in G. There are 2m + 3l nodes in G (Φ has m variables and l clauses) We look for a k = m + 2l vertex cover.



- HAMPATH is NP – complete. HAMPATH $\leq_p$ HAMCIRRCUIT (new vertex added to the graph)
- SUBSET-SUM is NP – complete. HAMCIRCUIT $\leq_p$ TSP (HAMCIRCUIT is a specific case of TSP)