Review Automata Theory

[**quant67**]
Harbin Engineering University

December 31, 2014

1 Why Study the Theory of Computation?

. . .

No reason for me.

2 Languages and Strings

2.1 Strings

A string is a finite sequance, possibly empty, of symbols drawn from alphabet Σ. Note that ε is an empty string and that Σ^* is the set of all possible strings over an alphabet Σ.

2.1.1 Functions on Strings

$|s|$: the length of a string s.

$\#_c(s)$: the number of times that the symbol c occurs in s.

st: the concatenation of two strings s and t, also written $s||t$.

w^i: the replication of string w. $w^0 = \varepsilon, w^{i+1} = w^i w$.

w^R: string reversal. $(wx)^R = x^R w^R$.

2.1.2 Relations on Strings

A string s is a **substring** of a string t iff s occurs contiguously as part of t.

A string s is a **proper substring** of a string t iff s is a substring of t and $s \neq t$.

A string s is a **prefix** of t iff $\exists x \in \Sigma^*(t = sx)$. A string s is a **proper prefix** of a string t iff s is a prefix of t and $s \neq t$.

A string s is a **suffix** of t iff $\exists x \in \Sigma^*(t = xs)$. A string s is a **proper suffix** of a string t iff s is a suffix of t and $s \neq t$.

2.2 Languages

A **language** is a (finite or infinite) set of strings over a finite alphabet Σ.

If $\Sigma \neq \phi$ then Σ^* is countably infinite.

If $\Sigma \neq \phi$ then the set of languages over Σ is uncountably infinite.

$L_1L_2 = \{w \in \Sigma^* : \exists s \in L_1(\exists t \in L_2(w = st))\}$. $(L_1L_2)L_3 = L_1(L_2L_3)$.

$L^* = \{\varepsilon\} \cup \{w \in \Sigma^* : \exists k \geq 1(\exists w_1, w_2, \cdots, w_k \in L(w = w_1w_2\cdots w_k))\}$.

$L^+ = LL^*$.

$L^R = \big\{w \in \Sigma^* : w = x^R \ for\ some\ x \in L\big\}$, $(L_1L_2)^R = L_2^R L_2^R$.

3 The Big Picture: A Language Hierarchy

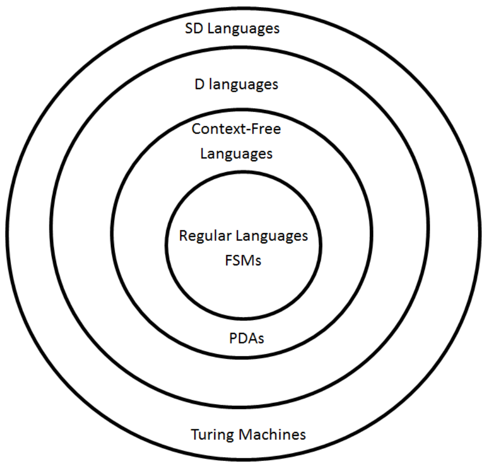

Figure 1: A hierarchy of language classes

4 Computation

Nothing to care about here, or it is hard to say.

5 Finite State Machines

5.1 Deterministic Finite State Machines

A **deterministic FSM**(or **DFSM**) M is a quintuple $(K, \Sigma, \delta, s, A)$, where:

- K is a finite set of states,
- Σ is the input alphabet,

- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states, and
- δ is the transition function. It maps from $K \times \sigma \; to \; K$.

M accepts w iff $(s, w)| -_M^* (q, \varepsilon)$, for some $q \in A_M$, Any configuration (q, ε), for some $q \in A_M$, is called an accepting configuration of M.

M rejects w iff $(s, w)| -_M^* (q, \varepsilon)$, for some $q \notin A_M$, Any configuration (q, ε), for some $q \notin A_M$, is called a rejecting configuration of M.

5.2 The Regular Languages

We define the set of **regular languages** to be exactly those that can be accepted by some DFSM.

5.3 Nondeterministic FSMs

also called **NDFSM**.

5.3.1 What Is a NDFSM

A NDFSM M is a quintuple $(K, \Sigma, \Delta, s, A)$, where:

- K is a finite set of states,
- Σ is an alphabet,
- Δ is the transition relation. It is a finite subset of:$(K \times (\Sigma \cup \{\varepsilon\})) \times K$.
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of final states, and

M accepts w iff at least one of its computations accepts.
M rejects w iff none of its computations accepts.

5.3.2 Analyzing Nondeterministic FSMs

Handling $\varepsilon-$Transitions

$$eps(q) = \{p \in K : (q, w)| -_M^* (p, w)\}.$$

The following algorithm computes *eps*:

$eps(q : state) =$

1. result=q.
2. While there exists some p$\in$ result and some r$\notin$ result and some transition $(p, \varepsilon, r) \in \Delta$ do: Insert r into result.
3. Return result.

A simulation Algorithm

$ndfsmsimulate(M : NDFSM, w : string) =$

```
1. current-state = eqs(s).
2. While any input symbols in w remain to be read do:
   (a) c = get-next-symbol(w).
   (b) next-state=$\phi$.
   (c) For each state q in current-state do:
           For each state p such that (q, c, p)$\in \Delta$ do:
               next-state = next-state$\cup eps(p)$.
   (d) current-state=next-state.
3. If current-state contains any states in A, accept. Else reject.
```

5.3.3 The Equivalence of Nondeterministic and Deterministic FSMs

If There is a DFSM for L, There is an NDFSM for L.

If there is an NDFSM $M = (K, \Sigma, \Delta, s, A)$ that accepts some language L, there exists an equivalent DFSM that accepts L.

The following algorithm computes M' given M:
$ndfsmtodfsm(M : NDFSM) =$

```
1. For each state q in K do:
       Compute eps(q).
2. $s' = eps(s)$.
3. Compute $\delta'$:
   (a) active-states=$s'$
   (b) $\delta' = \phi$.
   (c) While there exists some element Q of active-states for which $\delta'$ has
       not yet been computed do:
           For each character c in $\Sigma$ do:
               new-state= $\phi$.
               For each state q in Q do:
                   For each state p such that $(q, c, p) \in \Delta$ do:
                       new-state=new-state$\cup eps(p)$.
                   Add the transition (Q, c, new-state) to $\delta'$
                   If new-state $\notin$ active-states then insert it into active-
       states.
4. $K' = active = states$.
5. $A' = \{Q \in K' : Q \cap A \neq \phi\}$.
```

5.4 Minimizing FSMs

We will say that x and y are indistinguishable with respect to L, which we will weite as $x \approx_L y$ iff:

$$\forall z \in \Sigma^*(either\ both\ xz\ and\ yz \in L\ or\ neither\ is).$$

Let L be a regular language and let $M = (K, \Sigma, \delta, s, A)$ be a DFSM that accepts L, The number of states in M is greater than or equal to the number of equivalence classes of $\approx_L$.

Let L be a regular language over some alphabet Σ. Then there is a DFSM M that accepts L and that has precisely n states where n is the number of equivalence classes of $\approx_L$. Any other DFSM that accepts L must either have more states than M or it must be equivalent to M except for state names.

A language is regular iff the number of equivalence classes of $\approx_L$ is finite.
$minDFSM(M : DFSM) =$

1. classes=$A, K - A$.
2. Repeat untill a pass at which no change to classes has been made:
   (a) $newclass = \phi$.
   (b) For each equivalence class e in classes, if e contains more than one state, see if it needs to be split:
   For each state q in e do:
   For each character c in Σ do:
   Determine which element of classes q goes to if c is read.
   If there are any two states p and q such that there is any character c such taht when c is read, p goes to one element of classes and q goes to another, then p and q must be split. Create as many new equivalence classes as are necessary so that no state remains in the same class with a state whose behavior difffers from its. Insert those chasses into newclasses.
   If there are no states whose behavior differs, no splitting is necessary. Insert e into newclasses.
   (c) classes = newclasses.
3. Return $M' = (classes, \Sigma, \delta, [s_M], \{[q : the\ elements\ of\ q\ are\ in\ A_M]\})$, where $\delta_{M'}$ is constructed as follows:

$$if\ \delta_M(q, c) = p, then\ \delta_{M'}([q], c) = [p].$$

5.5 A Canonical Form for Regular Languages

$buildFSMcanonicalform(M : FSM) =$

1. $M' = ndfsmtodfsm(M)$.

2. $M_{\#} = minDFSM(M')$.

3. Create a unique assignment of names to the states of $M_{\#}$ as follows:

   (a) Call the start state q_0.

   (b) Define an order on the elements of Σ.

   (c) Until all states have been named do:

      Select the lowest numbered named state that has not yet been selected. Call it q.

      Create an ordered list of the transitions out of q by the order imposed on their labels.

      Create an ordered list of the as yet unnamed states that those transitions enter by doing the following: If the first transition is (q, c_1, p_1), then put p_1 first. If the second transition is (q, c_2, p_2) and p_2 is not already on the list, put it next. If it is already on the list, skip it. Continue until all transitions have been considered. Remove from the list any states that have already been named.

      Name the states on the list that was just created: Assign to the first one the name q_k where k is the smallest index that hasn't yet been used. Assign the next name to the next state and so forth until all have been named.

4. Return $M_{\#}$.

5.6 Finite State Transducers

A **Moore machine** M is a seven-tuple $(K, \Sigma, O, \delta, D, s, A)$, where:

- K is a finite set of states,
- Σ is an input alphabet,
- O is an output alphabet,
- $s \in K$ is the start state,
- $A \subseteq K$ is the set of accepting states,
- δ is the transition function. It is function from $(K \times \Sigma)$ to (O^*).
- D is the display or output function. It is a function from (K) to O^*.

A **Mealy machine** M is a six-tuple $(K, \Sigma, O, \delta, s, A)$, where:

- K is a finite set of states,
- Σ is an input alphabet,
- O is an output alphabet,

- $s \in K$ is the start state,
- $A \subseteq K$ is the set of acceptiong states, and
- δ is the transition function. It is a function from $(K \times \Sigma)$ to $(K \times O^*)$.

...

6 Regular Expressions

A **regular expression** is a string that can be formed according to the following rules

- ϕ is a regular expression.
- ε is a regular expression.
- Every element in Σ is a regular expression.
- Given two regular expressions α and β, $\alpha\beta$ is a regular expression.
- Given two regular expressions α and β, $\alpha \cup \beta$ is a regular expression.
- Given a regular expression α, α^* is a regular expression.
- Given a regular expression α, α^+ is a regular expression.
- Given a regular expression α, (α) is a regular expression.

Any language that can be define with a regular expression can be accepted be some FSM and so is regular.

7 Regular Grammars

A **regular grammar** G is a quadruple (V, Σ, R, S), where:

- V is the rule alphabet, which contains nonterminals and terminals,
- Σ (the set of terminals) is a subset of V,
- R (the set of rules) is a finite set of rules of the form $X \to Y$, and
- S (the start symbol) is a nonterminal.

In a regular grammar, all rules in R must:

- have a left-hand side that is a single nonterminal, and
- have a right-hand side that is ε or a single terminal or a single terminal followed by a single nonterminal.

The class of languages that can be defined with regular grammars is exactly the regular languages.

If L is a regular language then:

$$\exists k \geq 1(\forall \; strings \; w \in L, \; where \; |w| \geq k(\exists x, y, z \\ (w = xyz, |xy| \leq k, y \neq \varepsilon, \; and \; \forall q \geq 0(xy^q z \in L)))).$$

8 Context-Free Grammars

A **context-free grammar** G is a quadruple (V, Σ, R, S), where:

- V is the rule alphabet, which contains nonterminals and terminals,
- Σ (the set of terminals) is a subset of V,
- R (the set of rules) is a finite set of rules of the form $X \rightarrow Y$, is a subset of $(V - \Sigma) \times V^*$ and
- S (the start symbol) can be any element of $V - \Sigma$.

9 Pushdown Automata

A **pusdown automata** (or **PDA**) M is a sex-tuple $(K, \Sigma, \Gamma, \Delta, s, A)$, where:

- K is a finite set of states,
- Σ is the input alphabet,
- Γ is the stack alphabet,
- $s \in K$ is the start state,
- $A \subset K$ is the set of accepting states, and
- Δ is the transition relation. It is a finite subset of

$$(K \times (\Sigma \cup \{\varepsilon\}) \times \Gamma^*) \times (K \times \Gamma^*).$$

If L is a context-free language, then:

$$\exists k \geq 1(\forall strings \; w \in L, where \; |w| \geq k(\exists u, v, x, y, z(w = uvxyz, \\ vy \neq \varepsilon, |vxy| \leq k, and \; \forall q \geq 0(uv^q xy^q z \; is \; in \; L)))).$$

10 Turing Machines

A **Turing machine** M is a six-tuple $(K, \Sigma, \Gamma, \delta, s, H)$, where:

- K is a finite set of states,
- Σ is the input alphabet, which does not contain $\square$,
- Γ is the tape alphabet, which must, at a minimum, contain $\square$ and have Σ as a subset,
- $s \in K$ is the start state,
- $H \subseteq K$ is the set of halting states, and
- δ is the transition function, It maps from:

$$(K - H) \times \Gamma \ to \ K \times \Gamma \times \{\rightarrow, \leftarrow\}.$$

$\ldots\ldots$
I am hungry and tired of typing$\cdots$