Formal Models of Computation

March 21

Here's a big question:

- Given an arbitrary specification, are we guaranteed that there is a program that meets the specification?

To answer this question, we need to say what we mean by "specification" and "program". This is the topic of formal models of computation.

1 Specification/Problems/Languages

In this class, we'll take a very simple notion of specification: a specification (or "problem" or "language") is just a set of strings of characters from a finite set (called the "alphabet"). I'll use Σ to denote the alphabet and Σ^* to denote the set of finite length strings. We'll use ϵ to denote the empty string.

For example, here are some specifications:

- The set of all strings (this is implemented by the program that always says "yes")
- The empty set (implemented by the program that always says "no")
- The set of all strings of balanced parentheses
- The set of all strings of the form "n,m,k" where $n = mk$.

We'll consider programs that simply output "yes" or "no". A program meets a specification L if it outputs "yes" on input x if and only if $x \in L$. We'll refer to the set of strings on which a program P outputs "yes" as the "language of P", denoted $L(P)$. Formally:

$$L(P) = \{x \in \Sigma \mid P \text{ outputs "yes" on input } x\}$$

We say that P "recognizes" $L(P)$

2 First answer to the computability question

I haven't yet specified what a "program" is. For a moment, let's pick a concrete and intuitively appealing definition: let's say that a "program" is just a string containing Java source code for a function that takes a string as input and returns a boolean. For simplicity, let's interpret any string that is not valid Java source code as a program that says "no" on every input - thus every string can be interpreted as a program.

We can now state the "big question" asked earlier more formally, and easily provide an answer. We asked if every language L has a program P that recognizes it.

The set of languages is just the set of sets of strings:

$$\mathcal{L} = \mathbb{P}(\Sigma^*)$$

The set of programs is the set of strings:

$$\mathcal{P} = \Sigma^*$$

We have the function $L(\cdot)$ that takes a program and gives the language of the program:

$$L(\cdot) : \Sigma^* \to \mathbb{P}(\Sigma^*)$$

Asking whether every specification has a program is the same as asking if $L(\cdot)$ is onto.

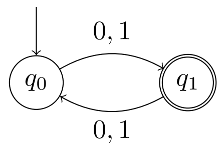
The answer is clearly no, because $|\mathbb{P}(\Sigma^*)| > |\Sigma^*|$, so there are no onto functions from Σ^* to $\mathbb{P}(\Sigma^*)$.

3 Deterministic Finite Automata (DFAs)

This is a very interesting fact with an unenlightening proof. What languages don't have corresponding programs? Can we characterize the ones that do somehow?

We'll start by studying a very simple model of computation (much simpler than Java!). Instead of strings containing Java text, we'll talk about programs that are built using state machines that process strings one character at a time in a very simple way.

These machines will have a collection of "states", and each time they process a character they will "transition" to a new state. Here is a picture of a machine that recognizes strings of odd length:



When processing a string, the machine starts in state q_0 (as indicated by the arrow pointing at q_0. When it is in a given state, it reads a character from the input, and follows the edge corresponding to that character from the state it is in to find a new state.

It will say "yes" on the string x if it is in the accept state (designated by the double circle) after processing x completely.

If we run this machine on the input "010", it will start in q_0. It will read "0" which causes it to transition to q_1. It will read "1", causing it to transition back to q_0. Finally, it will read "0", and transition back to q_1. Since q_1 is an accept state, it will say "yes".

4 Formalism

We can encode all of the parts of the picture symbolically as follows. A machine M consists of the following:

- A finite set Q of states ($\{q_0, q_1\}$ in the example)
- An initial state $q_0 \in Q$. (q_0 in the example)
- A set of "accepting states" $A \subseteq Q$ ($\{q_1\}$ in the example)
- A transition function $\delta : Q \times \Sigma \to Q$. In the example, $\delta(q_0, 0) = \delta(q_0, 1) = q_1$ and $\delta(q_1, 0) = \delta(q_1, 1) = q_0$.

Using this encoding, we can give a crisp definition of the informal process described above. We can use δ to define a function $\hat{\delta} : Q \times \Sigma^* \to Q$ that gives the state of the machine after processing an entire string. For the machine above, $\hat{\delta}(010) = q_1$.

$\hat{\delta}$ is defined inductively. For any state q,

- $\hat{\delta}(q, \epsilon) = q$
- $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$

Then M accepts x if $\delta(q_0, x) \in A$.

5 Nondeterministic Finite Automata (NFAs)

It is also useful to loosen the restrictions on building machines to allow multiple transitions on the same character, or no transitions at all. This allows us to more easily express some constructions, but turns out not to add any computational power (as we will see later).

We will also add "ϵ transitions," allowing the machine to transition from one state to another without consuming any input.

A machine will accept the input x if there is ANY path from the start state to an accepting state that is labeled by x.

Here is an example NFA:

$$\begin{tikzcd}
& q_1 \arrow[r, bend left, "\epsilon"] \arrow[d, bend left, "1"] & q_3 \arrow[d, "1"] \\
{} \arrow[r] & q_0 \arrow[u, bend left, "0"] \arrow[ur, "1"] \arrow[r, "1"'] & q_2 \arrow[loop right, "0"]
\end{tikzcd}$$

On the input 010, this machine will operate as follows. It begins in state q_0. After processing the first 0, it can transition to state q_1, and it can also take the ϵ transition to state q_3. So $\hat{\delta}(q_0, 0) = \{q_1, q_3\}$.

It will then process the 1, allowing it to transition to state q_0 (from state q_1) or to state q_2 (from state q_3). Finally it will process the second 0, allowing it to end in state q_1, q_2, or q_3. Since at least one of these is an accept state, the string 010 will be accepted.

Another way to look at the process is that we can insert some ϵs into the string 010 so that it gives a path from the start state to the accepting state: we can expand it to $0\epsilon10$, and take the path $q_0 \to q_1 \to q_3 \to q_2 \to q_2$. Since q_2 is an accepting state, we accept.

6 NFA Formalism

We can extend the formal definitions we gave for DFAs. We need to modify them in two ways:

- δ now returns a set of possible states: $\delta : Q \times \Sigma \to \mathbb{P}(Q)$
- We must encode the epsilon transitions. We do this using a function $\epsilon : Q \to \mathbb{P}(Q)$.

This means it's a little more effort to define the extended transition function, but it's not too bad[1]:

- $\hat{\delta}$ on the empty string is allowed to take epsilon transitions:

$$\hat{\delta}(q, \epsilon) = \{q\} \cup \epsilon(q)$$

- To get to a state q''' from state q on input ax, you can take an epsilon transition to get to q', then process a to arrive at q'' then process x to get to q'''.

$$\hat{\delta}(q, ax) = \{q''' \in \hat{\delta}(q'', x) \mid q'' \in \delta(q', a) \text{ and } q' \in \epsilon(q)\}$$

The machine will accept the input x if there is *any* final state $q \in A$ reachable from the start state:

$$L(M) = \{x \in \Sigma \mid \exists q \in \hat{\delta}(q_0, x) \text{ such that } q \in A\}$$

[1]this is slightly different then the definition I gave in lecture, which had a small bug

7 Summary of notation and definitions

There's a lot of notation in this lecture. Here's a summary for quick reference.

7.1 Languages

- Σ is a finite alphabet; Σ^* is the set of finite strings of Σ
- ϵ is the empty string.
- a language is a subset of Σ^*
- a program P recognizes the language

$$L(P) = x \in \Sigma^* | P \text{ outputs "yes" when run on input } x$$

7.2 DFAs

- DFA stands for "deterministic finite automaton"
- A DFA M is represented by a tuple (Q, A, q_0, δ) where
  - Q is a finite set of states
  - $q_0 \in Q$ is an initial state
  - $A \subseteq Q$ is a set of final states
  - $\delta : Q \times \Sigma \to Q$ is the transition function
- there is an extended transition function $\hat{\delta} : Q \times \Sigma^* \to Q$ defined as follows:
  - $\hat{\delta}(q, \epsilon) = q$
  - $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$
- $M = (Q, A, q0, \delta)$ outputs "yes" on x if $\hat{\delta}(q_0, x) \in A$

7.3 NFAs

- NFA stands for "nondeterministic finite automaton"
- An NFA is allowed to transition to multiple states on a given input (or no states)
- An NFA is represented by a tuple $(Q, A, q_0, \delta(\cdot), \epsilon(\cdot))$ where
  - Q, A, q_0 are the same as for a DFA
  - $\delta : Q \times \Sigma \to \mathbb{P}(Q)$ defines the set of transitions out of q
  - $\epsilon : Q \to \mathbb{P}(Q)$ defines the ϵ transitions
- $\hat{\delta} : Q \times \Sigma \to P(Q)$ is like $\hat{\delta}$ for DFAs but accounts for ϵ transitions and nondeterminism:
  - $\hat{\epsilon}(S) = \cup_{q \in S}\{q\} \cup \epsilon(q)$
  - $\hat{\delta}(q, ax) = \{q''' \in \hat{\delta}(q'', x) \mid q'' \in \delta(q', a) \text{ and } q' \in \epsilon(q)\}$