*Computer Science 52*

# *Models of Computation*

*Spring Semester, 2017*

Computer scientists use mathematical models to study the nature of computation. In these notes, we look at two sorts of models, finite automata and Turing machines.

## *I Deterministic finite automata*

Start with a finite alphabet, $\mathcal{A}$. In examples, we often use an alphabet with only two symbols, like a and b or 0 and 1. In "real life," the alphabet is the full alphabet or a computer system's character set. A *deterministic finite automaton,* abbreviated DFA, consists of

- a finite set $Q$ of states, one of which is designated as the start state $q_0$;
- a subset $F$ of $Q$ of final states; and
- a transition function $\delta : Q \times \mathcal{A} \to Q$.

The idea is that our automaton processes a string of alphabet symbols, one at a time. It begins in the start state, and each time it processes a character, it moves to a (possibly different) state as determined by the transition function. The "deterministic" part is that the new state is determined solely by the current state and the symbol.

A string is *accepted* if, after processing the string, the machine is in a state in $F$. Otherwise, the string is *rejected*. The *language* accepted by the DFA is the set of all accepted strings.

We can represent a DFA by a diagram with arrows labeled by alphabet letters, like the one in Figure 1. Starting in the start state, the one with a triangle, we process a string of characters from left to right. At each step, we follow the arrow labeled by the current character. If, after following an arrow for each character, we end up in a final state, we accept the string. The diagram shows a DFA which accepts the language of strings that start and end with a.

More formally, let $\mathcal{A}^*$ be the set of all strings composed of letters from the alphabet $\mathcal{A}$. It includes the empty string, denoted $\lambda$. The transition function $\delta$ takes states and characters into states. We can define a new function $\delta^*$ from states and *strings* into states using recursion; it is easy to imagine transforming this definition into an

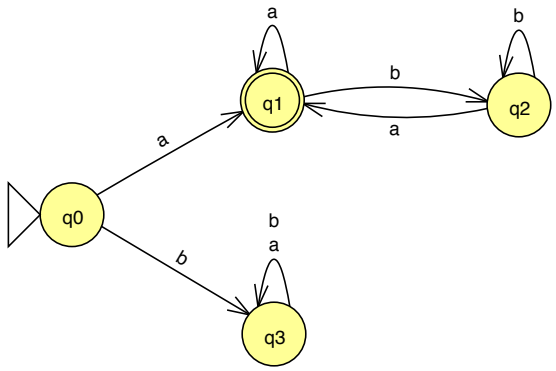


Figure 1: A DFA over the two-letter alphabet accepting strings that begin and end with a. The start state is $q_0$; the only final state is $q_1$.

SML function. Consider a string to be empty or to consist of a first character $c$ and a "rest of string" $s$.

$$\begin{aligned}\delta^*(q,\lambda) &= q\\ \delta^*(q,c\,s) &= \delta^*(\delta(q,c),s)\end{aligned}$$

Then the language accepted by the DFA with transition function $\delta$ is

$$\{t \in \mathcal{A}^* \mid \delta^*(q_0,t) \in F\}\,.$$

## *II Nondeterministic automata*

A nondeterministic finite automaton, abbreviated NFA, is one that may make "guesses" as it processes characters. More precisely, its transition function gives not a single new state but a set of possibilities. There may be several arrows out of a state, all labeled with the same character. Or there may be a state with no arrow labeled with a particular character. Let $\mathcal{P}(Q)$ be the set of subsets of $Q$, and consider a transition function of the form

$$\delta : Q \times \mathcal{A} \to \mathcal{P}(Q).$$

If the automaton is in state $q$ and encounters a symbol $c$, then $\delta(q,c)$ is the set of possible next states. One can think of the new state as being chosen randomly from the set. The set may have just one element, in which case there is no choice; or the set may even be empty, in which case there is no new state and the automaton freezes up. We can define a function that gives us the *set* of possible states of the automaton after processing a string. It is a function that takes a set of states and a string and produces a set of states.

$$\begin{aligned}\delta^*(R,\lambda) &= R\\ \delta^*(R,c\,s) &= \textstyle\bigcup_{q\in R} \delta^*(\delta(q,c),s)\end{aligned}$$

An NFA accepts the string $t$ if $\delta^*(\{q_0\},t)$ contains an element of $F$. A string is accepted if there is one sequence of guesses that results in a

final state. It is rejected if there is no way to get into a final state after processing the string.

Nondeterministic automata are useful because they are often simpler and more compact. The NFA in Figure 2 accepts the language of all strings that end with `bbb`.

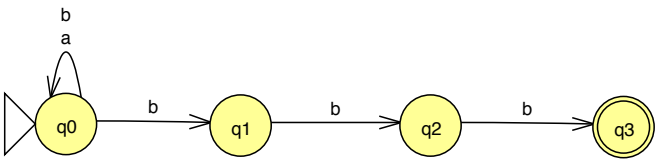


Figure 2: An NFA over the two-letter alphabet accepting strings that end with `bbb`. A DFA accepting the same language would have more states or more transitions.

A DFA is a special kind of NFA, so a language that can be accepted by a DFA is also accepted by a DFA. It is also true that a language that is accepted by a NFA is accepted by a DFA. To see that, transform transform a nondeterministic automaton into a deterministic one that accepts the same states. Given an NFA, construct a DFA whose states are sets of the NFA's states, and define a transition function similar to the one above.

## *III Regular and non-regular languages*

A language (that is, a set of strings over some alphabet) is *regular* if it is accepted by some finite automaton. We have seen that a language accepted by an NFA is also accepted by a DFA, so we can say that a language is regular if it is accepted by a DFA.

Are there languages that are not regular? Yes. A common example is the language of balanced parentheses. The string `()(()())()` is in that language, while `)()` is not. Most programming languages are not regular because they insist, among other things, that parentheses be balanced. Another example of a non-regular language is given in one of the assignments.

To see how one might determine that a language is not regular, suppose that we have a language and a DFA that accepts it. Consider how the DFA processes strings. If two strings, $t$ and $u$, end up in the same state and we append the same suffix $x$ to each of them, then the new strings $tx$ and $ux$ are also in the same state. We know that the two strings $t$ and $u$ are in *different* states if we can find a suffix $x$ such that one of $tx$ and $ux$ is accepted and the other is rejected. When this occurs, we say that we can "distinguish" between $t$ and $u$.

Notice that "distinguishing" depends only on the language being accepted and is independent of the particular DFA. Given a language

$\mathcal{L}$, we say that a pair of strings $t$ and $u$ is *distinguishable* if there is another string $x$ such that $tx$ is in $\mathcal{L}$ and $ux$ is not, or *vice versa.* We have the following theorem.

**Regular Language Theorem.** A language $\mathcal{L}$ is regular if and only if there is no infinite set of pairwise distinguishable strings.

Here is a sketch of one direction of the proof: If a language is regular, it is accepted by some DFA with $n$ states. Suppose that we have a set of strings with more than $n$ elements. Each string is processed by the DFA into *some* state, and since there are more strings than states, two strings must end up in the same state. (This is an application of the famous Pigeon Hole Principle.) Those two strings are indistinguishable. Therefore, any pairwise indistinguishable set can have at most $n$ elements and must be finite.

The proof of the converse requires equivalence relations, which are covered in another class.

We can now see why the language of balanced parentheses is not regular. Let $p_j$ be the string consisting of exactly $j$ left parentheses. If $j \neq k$, then $p_j$ and $p_k$ are distinguishable because, when $r$ is the string of $j$ right parentheses, $p_j r$ is in the language and $p_k r$ is not. The set of all $p_j$'s is an infinite pairwise distinguishable set.

## *IV Context-free Grammars*

Earlier in the course, we saw the EBNF formalism for defining languages. It is equivalent to context-free grammars, a topic that some of you may have encountered in a linguistics course. The language of balanced parentheses is described by a context-free grammar, as are most programming languages.

In a later course, you will study context-free grammars and discover that they can be recognized by a more powerful machine model called a *push down automaton.*

## *V Turing Machines*

Turing machines are the "ultimate" computing machines. Anything that can be computed can be computed on a Turing machine. The machines were invented by Alan Turing in the 1930's to study the nature of computation.

A Turing machine is like a DFA, except that it has a "memory" in the form of a paper tape. Here is a quick summary of the structure of the machine. A *Turing machine* consists of the following components:

- a finite input alphabet $\mathcal{A}$;
- a finite tape alphabet $\mathcal{T}$ which contains all the symbols in $\mathcal{A}$ plus (at least) a blank symbol $\text{␣}$;
- a finite set of states $Q$ with three especially-designated states—a start state $q_0$, an accepting state $q_A$, and a rejecting state $q_R$;
- a tape, divided into cells, each holding exactly one character;
- a head that scans one cell of the tape; and
- a finite set of possible transitions of the form $(a, q\,;\, b, D, r)$.

The tape is potentially infinite in both directions. By "potentially" we mean that we only use finitely much of it at any given time, but we can always extend the tape in either direction when we need to. All tape cells, except the ones that have been explicitly changed, contain the blank symbol $\text{␣}$. When the machine begins, it is in the start state, there is a string of symbols from the input alphabet on the tape, and the tape head is scanning the leftmost symbol of the input string.

A transition specifies one step in the computation. It is determined by the current state of the machine and the contents of the cell under the head. The transition $(a, q\,;\, b, D, r)$ applies when the tape head is scanning a character $a$ and the machine is in state $q$. The action of the transition is to replace the character by $b$ (which may be the same as $a$), to move the head as specified by $D$, and to enter state $r$. The machine is then ready for the next step in the computation. The head movement $D$ is either $L$, $R$, or $S$, standing for move left one square, move right one square, or stay put.

A string is *accepted* if the Turing machine enters its accepting state at some point along the computation. A string is *rejected* if it is not accepted. For simplicity, it is best to design the machine so that it enters the explicit rejecting state whenever possible, but this requirement cannot always be enforced. There may be situations in which no transition is applicable; the machine freezes and implicitly rejects its input in those cases.

Unlike a DFA or NFA, a Turing machine need not examine its entire input string. It may also go backwards over its input, or replace part of the input with other characters.

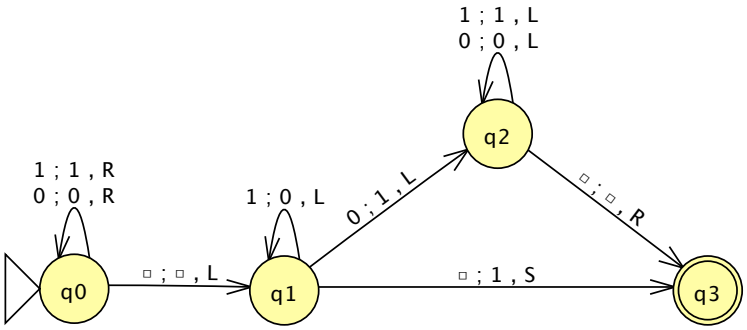


Figure 3: A Turing machine that adds one in binary.

Turing machines are not limited to accepting and rejecting strings.

They can also carry out computations to produce results. The Turing machine in Figure 3 adds one to a binary number. The labels on the arrows are

> old-character ; new-character, direction

The idea is that the machine (in its start state $q_0$) moves to the right of its input. It then enters its "carry state" $q_1$ and moves to the left, changing `1`'s to `0`'s. When it encounters a `0`, it exits the "carry state" and preserves the input until it halts on the left bit of the result.

Turing machines can be non-deterministic. That is, there may be situations in which two or more transitions are applicable. You can think of the Turing machine as making a random choice of which transition to apply in those cases. A string is accepted if *some* legal sequence of steps leads to an accepting state. Perhaps surprisingly, the class of languages accepted by non-deterministic Turing machines is the same as the class of languages accepted by deterministic ones.

As already mentioned, Turing machines are thought to be "universal" in the sense that any computable process can be carried out on a Turing machine. It would not be difficult to write in your favorite programming language a program that simulates a Turing machine. It is equally possible, but tedious, to design a Turing machine that simulates an ordinary computer program. Turing machines are valuable because they capture the nature of computation in a simple model and can therefore be used to study the limits of computation.

## *VI The Church-Turing Thesis*

The value of a Turing machine is not that we actually carry out computations on them, but that we use them to study computation. In this course you will design a few Turing machines to do simple computational tasks. If you had more time (as perhaps you will in a later course), you will create Turing machines with more complicated behavior. You might, for example, build a Turing machine that executed SML programs.

After some experience with Turing machines, people often come to a conclusion:

> Any computable process can be carried out on a Turing machine.

This assertion is known as the Church-Turing Thesis, or more simply, Church's Thesis. It is named for the mathematicians Alonzo Church and Alan Turing, both pioneers in the theory of computation.

The Church-Turing Thesis is a statement of confidence. It cannot ever be proved, because we do not have a clear definition of "computable process." One might, in principle at least, discover a process that everyone agrees is computable but cannot be carried out on a Turing machine. It is highly unlikely that anyone will do that, and the Church-Turing Thesis is usually taken as the *definition* of a computable process.

## *VII Self-reproducing Programs*

To see an unusual example of a computable process, we take a short digression into self-reproducing programs. We seek a program that takes no input and produces its own code as its output. Here is an example in SML.

```
val q="\"";
val s=";";
(fn x => x^q^x^q^s)"(fn x => x^q^x^q^s)";
```

This program is unsatisfying because the declarations of the values `q` and `s` are "outside" the program. We can put it inside by complicating the function a bit. The code below generates a self-reproducing expression in SML. We use ASCII values for the characters to avoid having to quote the quotation mark.

```
val q=str(chr 34);
val s=str(chr 59);
(fn x => x^q^x^q^s)
    "val q=str(chr 34);val s=str(chr 59);(fn x => x^q^x^q^s)";
```

This program is still not quite self-reproducing, because the code above contains line breaks and spaces that do not appear in the output. But the output itself is *exactly* a self-reproducing program!

The Java program below is another example of a self-reproduction. It comes from `www.nyx.net/~gthompso/self_java.txt` where it is attributed to Bertram Felgenhauer. A cursory web search will uncover literally hundreds of self-reproducing programs in all possible programming languages.

A computer virus is a slightly more sophisticated version of a self-reproducing program. Instead of merely writing its own source code, the virus makes a copy of itself and sends it on to another computer.

```
class S {
  public static void main(String[]a){
    String s="class S{public static void main(String[]a){
              String s=;char c=34;
              System.out.println(s.substring(0,52)+
                                 c+s+c+s.substring(52));}}";
    char c=34;
```

```
      System.out.println(s.substring(0,52)+c+s+c+s.substring(52));
    }
  }
```

Running these programs is a computational process. Therefore, if we accept the Church-Turing Thesis, there must be a Turing machine that does the same thing. By "the same thing," we might mean the trivial act of printing the above SML code or Java code, but in fact one can design a Turing machine that will print its own description. Try it before reading further!

If we accept the Church-Turing Thesis, then we have an appealing shortcut to designing Turing machines. We can informally describe a computational process and then use the Church-Turing Thesis to assert that a corresponding Turing machine must exist. It is only a shortcut, however. If we were pressed, we would have to verify an instance of the thesis by specifying the Turing machine down to the last transition. It is usually not a difficult task, but it is tedious and distracting. For our purposes now, we will rely on the Church-Turing Thesis and be satisfied with the higher-level descriptions.

## *VIII Universal Turing Machines*

If someone gives you a description of a Turing machine and an input string appropriate for that machine, you can step through the computation. With pencil and paper, you can keep track of the contents of the tape, the position of the head, and the state of the machine. It is a simple mechanical—or computational—process to move from one configuration to the next.

It follows that simulating the behavior of a Turing machine is itself a computational process. By the Church-Turing thesis, therefore, there must be a Turing machine that simulates other Turing machines.

More precisely, there is a Turing machine $U$, called a *universal Turing machine.* Suppose that $\langle M \rangle$ is the description of a Turing machine $M$ (suitably translated into the input alphabet of $U$) and $w$ is an input string for $M$. When presented with ta tape containing $\langle M \rangle$ and $w$, U does the same thing (accepts, rejects explicitly, or runs forever) as $M$ does on input $w$. We have a programmable computer!

The idea that a program, $\langle M \rangle$ in this case, is just another form of data was a crucial observation in the development of computers. The mathematician John von Neumann is usually credited with that insight.

The indistinguishability of program and data leads to interesting cases of self-refeence. One such example is the self-reproducing computer. Let $C_w$ be the Turing machine that erases its input, prints $w$ on the tape, and then halts. Creating $C_w$ from $w$ is a computational process, so there is a Turing machine $D$ that on input $w$ prints a description of $C_w$. Now let $E$ be the Turing machine that behaves as follows.

> On input $w$, $E$ begins by making a second copy of $w$. It simulates $D$ on one copy to obtain $C_w$. It then treats the second copy of $w$ as a Turing machine description and "composes" it with $C_w$: first $C_w$, then $w$. The Turing machine $E$ prints a description of the composite machine and then halts.

Notice that $E$ halts on all inputs, because it is only manipulating the *descriptions* of Turing machines.

Let $e$ be a description of $E$, and let $s$ be the Turing machine description produced by running $E$ with input $e$. Let $S$ be the Turing machine described by $s$. What does $S$ do? It runs $C_e$ and writes $e$ on the tape. It passes that input to $E$, which in turn produces $s$. Therefore, $S$ writes its own description.

## *IX The Halting Problem*

Here is a problem that programmers confront frequently: Given a program and an input, does the program with the given input halt? It is a question that can be asked in our ordinary language about computation. And it can often have an easy answer. All of us have found, and corrected, infinite loops in our code. But is there a general algorithm that will always answer the question?

Simulating the program on the input will give us only one possible answer. If the simulation halts, then we can say "Yes, the program halts." But if the simulation does not halt, we will never get to the poitn where we can say "No, the program runs for ever."

It turns out that there is no algorithm to decide whether a program halts on a given input value. In our language of Turing machines, there is no Turing machine $H$ with the following properties.

- An input for $H$ is a pair $(\langle M \rangle, w)$ consisting of a Turing machine description and an input.
- If $M$ with input $w$ halts, then $H$ with input $(\langle M \rangle, w)$ halts in an accepting state.
- If $M$ with input $w$ fails to halt, then $H$ with input $(\langle M \rangle, w)$ halts in an rejecting state.

The result is troubling. We have a simple question about computation that has no computable answer.

The proof that there is no Turing machine $H$ is similar to the construction of a self-reproducing program. Suppose that we have a Turing machine $H$ that halts on all inputs. We will find a Turing machine $M$ and an input $w$ for which $H$ gives the "wrong answer" for $(\langle M\rangle, w)$. Let $D$ be a Turing machine that, on input $x$, does the following.

i. creates the pair $(x, x)$,
ii. simulates $H$ on $(x, x)$, and
iii. accepts $x$ if $H$ rejects and loops forever if $H$ accepts.

This is a symbolic activity that follows explicit rules and uses a bounded amount of resource. Therefore, by the Church-Turing Thesis, it can be carried out on a Turing machine. Let $\langle D\rangle$ be a description of this machine. What does $D$ do on input $\langle D\rangle$?

- If $H$ accepts $(\langle D\rangle, \langle D\rangle)$, then $D$ loops forever on $\langle D\rangle$.
- If $H$ rejects $(\langle D\rangle, \langle D\rangle)$, then $D$ accepts $\langle D\rangle$—and halts.

Either way, $H$ gives the "wrong answer" for $(\langle D\rangle, \langle D\rangle)$, and $H$ is not the "halting checker" that we were seeking.

The problem of finding an algorithm that will tell us whether $M$ halts on input $w$ is called the *Halting Problem.* The proof just sketched shows that there is no computable solution to the Halting problem. It is an example of a problem that has no computable solution. There is, of course, a non-computable solution to the Halting Problem. The definition

$$\mathcal{H}(\langle M\rangle, w) = \begin{cases} 0 & \text{if } M \text{ halts on } w, \text{ and} \\ 1 & \text{otherwise} \end{cases}$$

is a perfectly well-defined mathematical function. It is just not a *computable* function.

Thnegative solution to the Halting Problem has important consequences. For example, we cannot decide in a computational manner if a Turing-machine-and-input pair enters a specific state, because if we could, we could apply that ability to the halting state and determine whether another computation halts. Translated to programs, we cannot decide—in an algorithmic manner—whether a program enters a particular block of code. This limits the ability of a compiler to eliminate "dead code." There are many similar "impossibility" results.