19 Theory of Computation

"You are about to embark on the study of a fascinating and important subject: the theory of computation. It comprises the fundamental mathematical properties of computer hardware, software, and certain applications thereof. In studying this subject we seek to determine what can and cannot be computed, how quickly, with how much memory, and on which type of computational model."
- Michael Sipser, "Introduction to the Theory of Computation", one of the driest computer science textbooks ever

In which we go back to the basics with arcane, boring, and irrelevant set theory and counting.

This week's material is made challenging by two orthogonal issues. There's a whole bunch of stuff to cover, and 95% of it isn't the least bit interesting to a physical scientist. Nevertheless, being the starry-eyed optimist that I am, I'll forge ahead through the mountainous bulk of knowledge. In other words, expect these notes to be long! But skim when necessary; if you look at the homework first (and haven't entirely forgotten the lectures), you should be able to pick out the important parts. I understand if this type of stuff isn't exactly your cup of tea, but bear with me while you can.

In some sense, most of what we refer to as computer science isn't computer science at all, any more than what an accountant does is math. Sure, modern economics takes some crazy hardcore theory to make it all work, but underlying the whole thing is a solid layer of applications and industrial moolah. Similarly, the vast bulk of computer science is driven in one way or another by dot-com millionaires leading high-powered lives from the seclusion of their 50th-story penthouse offices. But underneath it all lies a layer of *real* computer science as far removed from pets.com as Galois theory is from a balanced checkbook.

Theory of computation is the broad label applied to a portion of computer science that has not much to do with programming and a lot more to do with discrete mathematics. It can be roughly broken down into two areas: complexity analysis (which we've encountered earlier) and computability. The former actually deals with things like "numbers" and "algorithms", which tend to be a little too concrete for the latter. Computability, the portion of computational theory that we get to discuss here, really delves down into abstract math to describe the limits of what any computer can do, regardless of its speed or the runtime of the algorithm.

To do away with all of this programming and instruction nonsense, we'll limit our definition of an algorithm to a set of strings; that is, a language. In terms of theory of computation, a string is an ordered sequence of zero or more arbitrary symbols, and a language is a set (in the mathematical sense) of strings (either finite or infinite). This seems like it must limit your computational power a bit; after all, it's tough to play WoW when all you've got are sets of strings, no graphics, no networking, no nothing. On the other hand, you could probably come up with some clever ways to encode things. If you have a five-by-two pixel screen, the string ".o.o.._/." might be meaningful (well, marginally). The language $\{1, 2, 4, 8, 16, 32, ...\}$ only uses ten different symbols, but it's awfully interesting. It's all about posing the question correctly.

If algorithmic problems are to be posed as languages, then, the fundamental problem of computation is to determine whether a string is in a given language. "Is 128 a power of two?" is equivalent to testing whether the string 128 is in that language we just saw. "Is 8675309 prime?" is equivalent to testing whether that string is in some set $\{2, 3, 5, 7, 11, 13, ...\}$ Given a full-blown computer, we can write a program to test these things - but the task of theory of computation is to develop a mathematical formalism, *also* just using set theory, that can determine what languages are recognizeable and which are just impossible gibberish. Let's look at some potential models of computation that could capture real computers mathematically; we'll avoid most of the set theory here, but keep in mind that everything we discussed can be reduced to a surprisingly small collection of simple mathematical operations.

19.1 Regular Expressions

"Kleene, Kleene, give me your closure do,
I'm have crazy all for the use of you.
You repeat my statements over,
And over and over and over and over and over and over and over.
You're so neat, and make languages complete,
With the closure that's named for you." - Curtis Huttenhower, written for Computer Science 1 (sung to the tune of Daisy, Daisy)

In that whirlwind tour of computational theory, we established that any algorithmic problem can be modeled as a language and that a language was just a set of strings. Without a lot more time to spend on this, that might not make a lot of sense (and it might not seem very interesting even if it does), but if you happen to be in a mathy sort of mood, it opens the door to a lot of interesting analyses on what computers can and can't do - some of which you've already seen in lecture.

Metaphysical questions about the nature of computation and thought aside, though, it also provides some motivation for developing a convenient way of describing whole sets of strings at once. If you want to describe the language, "Every string consisting of zero or more 0 symbols and nothing else," or the (more practical) language, "Every string of length exactly 16 containing any combination of the letters A, G, C, and T," it's nice not to have to write out big long sentences every time. Besides, as we established with our little exercise in language modeling, the computer's not going to understand complete sentences anyhow! What does it take to formally, mathematically, and concisely describe a whole set of strings using some sort of shorthand notation?

Hmm. Sets of strings. If those are the two important words, what does it take to build a set or a string in isolation?

- Strings are just sequences of characters, zero or more in a row, with ordering and repetition. Whether we're defining a single string or a whole set of them, the characters they contain must be drawn from some alphabet, which is just a set of arbitrary symbols. The string "This is a test" is made up of symbols drawn from the alphabet $A = \{T, h, i, s, \ , a, t, e\}$, as are the strings "", "ThisThis", and "tastie state at that".
- The empty string "" is sort of special, so it's often referred to using the symbol ϵ (since writing out empty quotes like that looks really weird after a while and gets tough to read).
- Sets are just collections of entities over which the familiar union $\cup$ and intersection $\cap$ operators do their things.
- The empty set $\{\}$ is similarly special, and we often use the symbol $\emptyset$ to refer to it.

These four statements pretty much define everything important we can do using strings and sets. If we want to formalize a representation of sets of strings, why don't we just combine their forces Captain Planet style and see what we get?

- A language is a set of strings. It can be finite (e.g. "The set of all strings over the alphabet $A = \{0, 1\}$ containing exactly three symbols.") or infinite ("The set of all strings over $\{0, 1\}$ containing exactly one 0.").
- The empty set $\emptyset$ is a set containing only strings (that is, it contains nothing that isn't a string), so it must be a language. We'll give the empty language the special name $\Omega = \{\}$.
- The set $\{\epsilon\}$ is a set containing only strings (it contains exactly one string, the empty string $\epsilon = ""$), so it must be a language. We'll give the language containing only the empty string the special name $\Delta = \{\epsilon\}$.
- If S is a language and T is a language, then $S \cup T$ must be a language (since the union of two sets of strings will result in a new set containing only strings). For languages, since we're lazy computer scientists, we'll drop the union symbol and just write $S + T$ to represent a choice between any string in the language S or any string in the language T.
  - As a quick example, consider $S = \{"00", "01"\}, T = \{"11", "10"\}$, and $S + T = \{"00", "01", "11", "10"\}$.
  - Also note that union is both associative and commutative, just as it is for any sets. $(A + (B + C)) = ((A + B) + C)$, and $A + B = B + A$.
- If S is a language and T is a language, then we'll define the concatenation operator $S \times T$ as the cross product of the sets S and T. In practical terms, $S \times T$ generates a set containing every combination of a string from S followed by a string from T. For example, if $S = \{00, 01\}$ and $T = \{11, 10\}$, then $S \times T = \{0011, 0010, 0111, 0110\}$. Note that:
  - Because we're still lazy computer scientists (even though I've had some coffee between typing the last bullet point and this one), we've dropped the quotes surrounding strings in our languages. Typing that many quotes gets annoying real fast! They're still assumed to be there - they're just invisible. The emporor's new quotes.

  - Similarly, the concatenation operation $S \times T$ is often abbreviated ST.
  - Concatenation is associative but not commutative. $((AB)C) = (A(BC))$, but $AB \neq BA$. There's an obvious difference between {fire}{truck} = {firetruck} and {truck}{fire} = {truckfire}.
  - Finally, consider that $|ST| = |S||T|$ (hence the times symbol $\times$ making some sense to use there), $S\Delta = \Delta S = S$, and $S\Omega = \Omega S = \Omega$.

- Is S is a language, we define the Kleene closure operator S^* as... well, something complicated. Mathematically, suppose that we define S^n as S concatenated with itself n times. For example:

$$S = \{0, 1\}$$
$$S^0 = \Delta$$
$$S^1 = S = \{0, 1\}$$
$$S^2 = SS = \{0, 1\}\{0, 1\} = \{00, 01, 10, 11\}$$
$$S^3 = SSS = \{0, 1\}\{00, 01, 10, 11\} = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

  Then the Kleene closure operator can be defined as:

$$S^* = \bigcup_{i=0}^{\infty} S^i = S^0 \cup S^1 \cup S^2 \cup ...$$

  Thus for our example of $S = \{0, 1\}$, $S^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, ...\}$. As before, there are a bunch of mathematical properties of the closure operator that arise directly from the definition:

  - $\epsilon \in S^*$ for any S, even $S = \Omega$.
  - $\Omega^* = \Delta^* = \Delta$.
  - $|S^*|$ is unbounded for any language other than Ω or Δ.

- As a final laziness abbreviation, languages containing single strings are pretty much always abbreviated using the string itself. Thus 0 actually means $\{0\}$, and 010 actually means $\{0\}\{1\}\{0\} = \{010\}$. This lets you write shorthand statements like $(0 + 1)^*(00 + 11)$ to mean complicated things like $(\{"0"\} + \{"1"\})^*(\{"0"\}\{"0"\} + \{"1"\}\{"1"\})$. Try typing that expression the short way and the long way and you'll see why the shorthand is useful!

- In addition to these three operators (union, concatenation, and closure), you can always add parentheses to clarify the order of operations. Without parentheses, the precedence is $*$ before $\times$ before $+$ (just like exponentiation comes before multiplication comes before addition in normal arithmetic). For $S = \{00, 01\}$ and $T = \{10, 11\}$, $SS + T = (SS) + T$ describes a very different language than $S(S + T)$!

By now, the obvious question is, "Why on Earth should I care about any of this?" The less-than-obvious answer is that every language is not just a set of strings, it's also implicitly a pattern. Given some input alphabet, a language represents a set of strings that all match some pattern. Over the alphabet $\{0, 1\}$, you might be interested in the language, "All strings containing the pattern 010," represented by $(0 + 1)^*010(0 + 1)^* = \{010, 0010, 0100, 1010, 0101, 00010, ...\}$. You might be interested in, "All strings starting with a 1," represented by $1(0 + 1)^* = \{1, 10, 11, 100, 101, ...\}$. Over the alphabet $\{A, C, G, T\}$, you might be interested in the eminently more practical language, "All sequences upstream of a gene containing the string TATATAAA". Although languages as set-theoretical constructs probably aren't too interesting to the physical scientists among you, as pattern matchers they have, in the words of Half-Life, limitless potential.

Those of you familiar with Perl (or, for that matter, any advanced text editor's search-and-replace feature) might see where I'm going with this. The formal mechanism that we've just developed for describing languages also happens to be the foundation of regular expressions (also referred to as REs), a pattern matching tool popularized in its most horrifically complex form by Perl and spread in less intimidating embodiments to text editors and programming languages everywhere. Even Java gets in on the act - check out the documentation for the java.util.regex package!

Before we go too far down this route, I'm going to spend a while giving examples of regular expressions and rambling about what they represent. If you feel like you already Get It, feel free to skip ahead. If not, hopefully you (like me) learn well by examples! All of these REs are over the alphabet $\{a, b\}$, although this is pretty much equivalent to the alphabet $\{0, 1\}$ that I used above, and of course you can define REs over any alphabet you'd like. Alphabets of size two are just convenient because A) they allow us to express binary numbers and B) the number of possible combinations of n symbols, as we've seen, grows exponentially. If your alphabet $A = \{0, 1, 2, 3\}$, A^4 is really big, whereas $\{0, 1\}^4$ is at least relatively easy to write. In any case... on to the good stuff.

- $(a + b)^*$
  We've seen this guy above: any string containing zero or more copies of either a or b in any order, or $\{\epsilon, a, b, aa, ab, ba, bb, ...\}$. You can think of the Kleene star operator as allowing us to repeatedly pick something from $(a + b)$ over and over, as many times as we want.

- $(a^* + b^*)$
  This language means, "anything that's zero or more copies of a or anything that's zero or more copies of b." The first half of that includes ϵ, a, aa, aaa, and so on; the latter half includes ϵ (as does anything to which a Kleene star is applied), b, bb, bbb, and so forth. So the whole set of strings looks like $\{\epsilon, a, b, aa, bb, aaa, bbb, ...\}$

- $aba(a + b)^*$
  We can read this as, "The string aba followed by anything from all strings over $\{a, b\}$." Each string in the language has to start with a followed by b followed by a, but then we see the familiar pattern $(a + b)^*$ afterwards, which pretty much just means anything. So the language will look like $\{aba, abaa, abab, abaaa, abaab, ababa, ababb, abaaaa, ..$
  Note that ϵ is not in this language because (to think of it in English) ϵ doesn't start with aba.

- $(a + b)^*aba$
  This is pretty much the same deal, except that now each string in the language has to end with aba. We can read this as, "any string over $\{a, b\}$ followed by aba." So this means $\{aba, aaba, baba, aaaba, ababa, baaba, bbaba, aaaaba, ...\}$. Again, no ϵ.

- $(a + b)^*aba(a + b)^*$
  This is getting more interesting. Now we have any string over $\{a, b\}$, followed by aba, followed by any string again. A good English way to say this would be, "Any string over $\{a, b\}$ containing the substring aba." We're allowed to begin and end with whatever we want, so long as each string in the language contains aba somewhere within it. So the set will look like $\{aba, aaba, abaa, baba, abab, aaaba, abaaa, abaab, ...\}$

- $(a + b)(a + b)(a + b)$
  We've seen this guy before in a sightly different guise (think 0s and 1s). Like I said the first time it came up, languages represented by REs don't have to be infinite! This can be read in English as, "Any three characters from $\{a, b\}$," or more procedurally, "Pick any one character from $\{a, b\}$ followed by any one character from $\{a, b\}$ followed by any one character from $\{a, b\}$." This will generate a finite language: $\{aaa, aab, aba, abb, baa, bab, bba, bbb\}$. Note that we can count the number of strings in the language by filling three slots, each with two choices: $2 * 2 * 2 = 2^3 = 8$.

- $(a + ba)^*(b + \Delta)$
  In real life (and in the assignment), you'll pretty much never see regular expressions written using Δ or Ω. But if you check back allll the way up at the top of the notes, you'll see that this is still a legal RE as we defined them, so it must mean something. We can break it down to something that sounds like, "Any string consisting of zero or more combinations of a and ba followed by b or nothing." The first part makes the set $\{\epsilon, a, ba, aa, aba, baa, baba, ...\}$. The second part is trickier; formally, $b + \Delta = \{"b"\} \cup \{\epsilon\} = \{"b", \epsilon\}$.
  We've figured out how to generate the set of strings for this language... but there's a hidden English explanation that makes things much more obvious. In English, it's easy to say, "The set of all strings that don't contain *bb*." But REs only allow us to define what strings a set *does* contain, not what strings it doesn't. To formally describe all strings not containing *bb*, we have to figure out what pattern it is that they all *do* share. In this case, we allow any combination of a and ba any number of times - which is a way of always guaranteeing that a b will be preceded and/or followed by an a. This precludes any occurrences of *bb* in the first part of the string. But it also prevents the string from ending in b, which is not what we want. So we allow either b or ϵ to be concatenated onto the end, the former to allow strings like b or $aaaab$, and the latter to allow strings like a or $aaaa$. And thus the Δ appears in our RE because that's how we say, "The language containing just ϵ."

- $a^*b(a + b)^*$ or $(a + b)^*b(a + b)^*$
  These two REs prove by example something you might have suspected by now: you can represent the same language using more than one different regular expression. We could read the first RE as, "Any string starting with zero or more *a*s followed by one b and then anything." We can read the second one as, "Any string starting and ending with anything and containing one b in the middle." Or we can read both REs as either, "Any string containing at least one b," or, "Any string containing a b somewhere in it."
  The second RE is a little more intuitive than the first one, but the first one is definitely more concise. Think about it (the first one) this way. If a string over $\{a, b\}$ contains at least one character, it has to start with either an a or a b. If it starts with a b, we're done, and it can contain anything it wants after that. This is the case

that occurs when the a^* in the first RE expands to zero copies of a. However, if the string starts with an a instead, we eventually have to hit a b for the string to match our pattern. As soon as we hit a b, anything else is allowed, and until we hit that first b, we can keep generating as many copies of a as we want. And that's what happens when a^* expands to one or more copies of a; once we hit the first b, no matter what comes before it, we're allowed to include anything afterwards.

- $ab + ba$
  This RE clearly represents the language containing all palindromic band names (you'd be amazed at the weird results you get on Google if you search for "palindromic band names"). Note that this should not be confused with $(U + 2)^*$, the set containing all good bands with digits in their titles; $(BO)^*(YZ)^*(2 + MEN)$, the set containing all bad bands with digits in their titles; or , the set of all artists formerly known as some combination of characters from the English alphabet.

Regular expressions make nifty theoretical tools for describing sets of strings, and as a theoretical construct, it's useful to keep them as minimalist as possible: three operations plus an alphabet, end of story. Out in the real world, regular expressions are *the* way of performing pattern matching, and they get extended left and right with more syntax, operators, bells, whistles, and constructs that actually make "regular expressions" in the wild nothing at all like regular expressions in their theoretical incarnation. Nevertheless, these are the beasties you're actually interested in (or should be!), so I'll quickly cover some of the ways REs are implemented in languages like Perl or Java.

- First of all, regular expressions as a theoretical construct always match the entirety of every string in a language. In the language described by a^*b^*, every string is entirely of the form, "Zero or more as followed by zero or more bs," with no extra characters floating around. In the real world, regular expressions are often used as patterns to match part of a string. In Perl, the regular expression `a*b*` matches the strings `a`, `b`, `aaab`, `abbb`, and so forth as expected, but it also matches `woonkwoonkaboompaloompa` (or even just `woonkwoonkoompaloompa`, since a^*b^* includes ϵ). In Java, most regular expression operators force a match of the entire string, so this shouldn't be an issue.

- In a similar vein, the syntax for using regular expressions in Java is:

```
Pattern ptrn = Pattern.compile( "a*b*" );
Matcher mchr = ptrn.matcher( "aaab" );
mchr.matches( );
```

  Here, `"a*b*"` is the regular expression and `"aaab"` is the input string to be tested. In this case, `mchr.matches( )` will return true; if we had generated `mchr` from the input string `"baaa"`, if would return false. The `Matcher` class includes a plethora of methods for manipulating a matched (or unmatched) RE/input pair; documentation on `Matcher` and `Pattern` are found on the usual java.sun.com site in the package `java.util.regex`.

- Real-world regular expression syntax includes the closure, concatenation, and union operators just like we've seen here. Kleene closure is represented using * as expected, and concatenation is represented using nothing (just like we've written it here). Just to be confusing, though, union is represented using the | (bar or pipe) operator. Thus, in Java, $(a + b)^*$ would be written as `(a|b)*`. Why two different standards developed for theory versus practice, I'm not sure, but it makes life difficult...

- ...because the + operator means something completely different in Java. Consider that Kleene closure * means, "zero or more copies of the preceding thing." In those words, the plus operator means, "one or more copies of the preceding thing." Thus `(a+b)*` will match the strings `""`, `"ab"`, `"abab"`, `"aab"`, `"aabab"`, and so forth.

- Similarly, the ? operator matches exactly zero or one copies of the preceding thing; you can think of it as an optional operator (the preceding thing might appear once, but it's optional). `(a?b)*` will match the strings `""`, `"b"`, `"ab"`, `"bb"`, `"abab"`, `"abb"`, and so forth.

- Java (and Perl) also includes many (many many) pieces of syntax not directly mappable to theoretical regular expressions, often due to the fact that real-world REs don't generally operate over a defined alphabet. This allows the creation of character class operators that match any character in a large (or infinite) set. The dot character . is possibly the most popular, matching any one character at all. This lets you use the quick RE `.*` to represent any amount of anything; `.*abc.*` will match any string containing the letters `"abc"`. `f.l.l.` will match the strings `"falala"`, `"falelo"`, `"fFlLlL"`, or anything else containing f, l, and l in the appropriate positions.

- You can define your own (finite) character classes using the bracket operators []. Characters within a pair of square brackets are treated as one big union; in other words, the Java regular expression `[abc]` is equivalent to the theoretical RE $a + b + c$. This is most useful in combination with the - range operator, which lets you write things like `[a-z]` to represent any lower case letter, `[a-fA-F0-9]` to represent any hexadecimal numeral (without respect to case), or `[aeiou0-4]` to represent any vowel or digit below five. If a character class starts with ^, it is complemented and represents any character *not* specified in the class. Thus `[^fvsz]` represents (roughly) anything that's not a lower case fricative English consonant, and (to be a little less fancy) `[^0-9]` represents any one character that's not a digit.
- Java (and Perl) include several useful predefined character classes. `\d` represents any one digit (and thus `\d+` matches any unsigned decimal integer, for example), `\s` represents any whitespace, and `\w` represents any "word" character (alphanumeric or underscore). The complements of these classes are also built-in as `\D` (any non-digit character), `\S` (any non-whitespace character; very useful!), and `\W` (any non-word character).
- There are also a bunch of regular expression characters that don't match characters at all - they match the spaces between characters, called edges or boundaries. The only ones of these that show up very often are the ^ and $ characters matching, respectively, the beginning and end of a line. This means that the RE `^.*[aeiou].*$` matches any complete line containing at least one lower case vowel, and `money$` matches any line ending with the lower case string `"money"`.
- Finally, parentheses serve more than one purpose in real-world regular expressions. In addition to performing their normal service as precedence indicators, they also function as capturing groups. This is possibly the most useful behavior of real-world regular expressions, particularly when completing your homework! Any portion of a regular expression contained between parentheses is captured and saved for later retrieval by index; the first capture group is given index one, the second two, and so forth (unlike arrays/strings/etc., capture groups start at index one!) If I execute the following code:

  ```
  Matcher mchr = Pattern.compile( "(i+)(c+)(u+)" ).matcher( "iiiccu" );
  mchr.matches( );
  for( int i = 1; i <= 3; ++i )
      System.out.println( mchr.group( i ) );
  ```

  I'll get the output `"iii"`, then `"cc"`, then `"u"`. If I match the same regular expression against the input `"icccuu"` instead, I'll get (predictably) the output `"i"`, `"ccc"`, `"uu"`. Capture groups can be used for a million and one things, and they're actually responsible for part of what makes real-world REs very very much not at all like theoretical REs (example: you can represent the language ww using the Java RE `(.*)\1`, but ww isn't a regular language and can't be represented theoretically). Normally, *any* use of parentheses will create a capture group with its own sequentially increasing index; if you want to use parentheses for grouping or precedence marking without creating a capture, use the construct `(?:stuff)` in place of the usual `(stuff)`.

I could keep creating bullet points for another dozen pages without even scratching the surface of Java regular expressions, and Perl contains even more crazy junk than that. Most of the important points are covered on the `Pattern` class documentation page:

`http://java.sun.com/j2se/1.5.0/docs/api/java/util/regex/Pattern.html`

For the somewhat more curious, Sun's written a Java regular expressions tutorial, although I've always found it a little circuitous and inscrutable:

`http://java.sun.com/docs/books/tutorial/extra/regex/index.html`

For the completely insane and not-nearly-occupied-enough, there's a series of Perl documentation pages that provide something of the canonical-and-exhaustive regular expressions reference:

`http://www.perl.com/doc/manual/html/pod/perlre.html`

I encourage you to read these to your heart's content, but I'm hoping that your heart is content with just the `Pattern` class page; reading through the entirety of the Perl documentation would probably take a significant bite out of the time you have available to do things like, say, your homework.

19.2 Deterministic Finite Automata

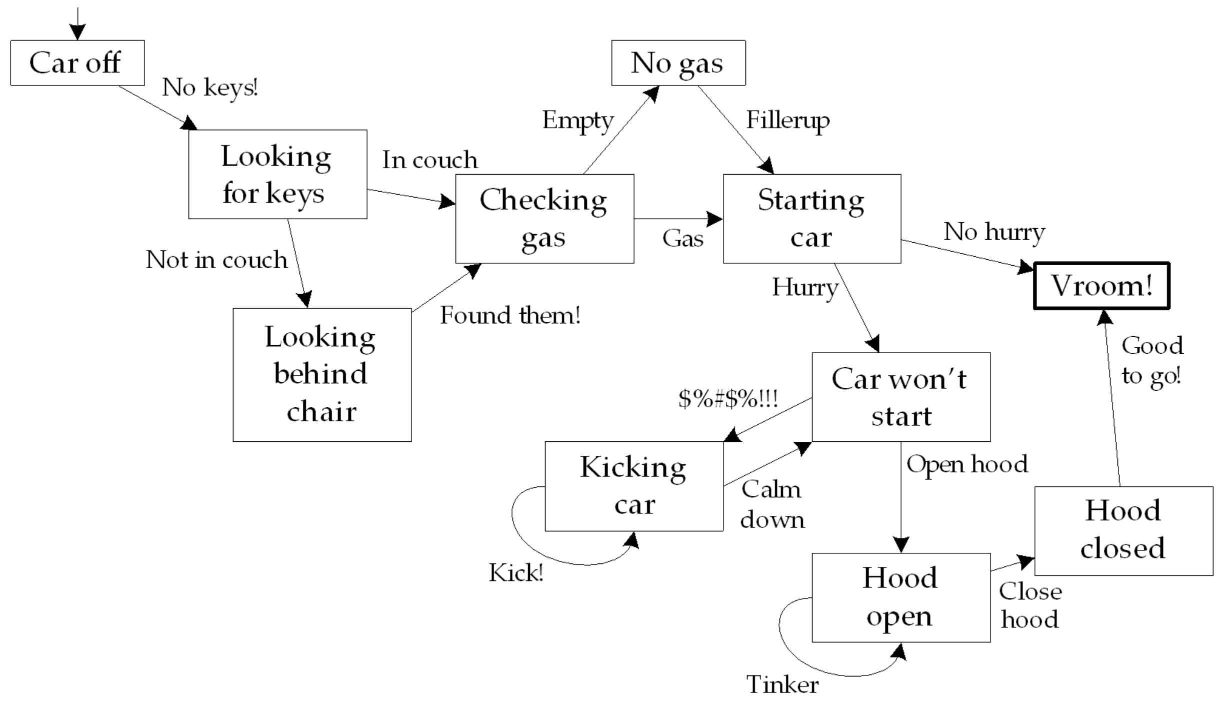
"Democracy For America is dedicated to grassroots activism and supporting candidates at all levels of government.
Dimensional Fund Advisor applies academic research to the practical world of investing.
The Department of Foreign Affairs - Republic of the Philippines
DFA Capital Management Inc. is the first software vendor to enable integrated Enterprise Risk Management (ERM) for Insurance companies.
Death From Above 1979
Arkansas Department of Finance and Administration provides information and assistance for state residents in complying with tax, child support, and licensing laws."
- results of a Google search for DFA, none of the first twenty of which have anything to do with computer science

One way to start this section would be to ask the question, "If I were a machine, how would I try to process regular expressions?" But that would probably give too much away, so we won't ask that question. Instead, we'll ask the far more important question, "How does a car work?"

Well, that's a big question, so we'll break it down into some simple rules:

- When we first get to our car, it always starts turned off.
- We always have to look for our keys first.
  - Most often, they're in the couch.
  - Otherwise, the cat hid them behind a chair.
- Once we get to the car, if we have plenty of time, it starts up and we drive off.
- If we're in a hurry, the car won't start.
  - We can kick the car any number of times to make it start, but it won't work.
  - We can open the hood.
    - The car won't ever start while the hood's open!
    - We can tinker as much as we want and then close the hood again.
- If the fuel tank is empty, the car won't start.
  - Unless we add fuel, in which case we have the same problems as before.

I warned you that cars were complex pieces of machinery! Fortunately, we're allowed to draw pictures, so we can summarize the entire workings of our car in the following diagram:



We always start over on the left, where the arrow goes in; this is the start state, and there is always exactly one (just like a Markov model). Then we can perform any number of actions or receive any number of inputs to move between states (also just like a Markov model). The arrows with labels (generally called transitions) are the actions and inputs; for example, "Open hood" would be an action, while "Empty" would be an input (we've received the information that the gas tank is empty). Similarly, the boxes all represent states; states tend to represent things that are ongoing or currently true about the world (such as the ongoing "Kicking car" or the currently true "Hood open" and "Hood closed"). The accept or final state is the one in bold labeled "Vroom!" That's where we want to get to - and unlike the start state, we can have more than one of these. Suppose that tinkering didn't always fix the car; maybe then we'd have a, "Give up, go home, and call in to work sick," state that was also a final state. The major difference relative to Markov models is that our transitions are *not* random; in fact, for any given input, we always have exactly one transition available that we must take.

Note that it is possible to enter this process and not reach a final state. What if we got to the "No gas" state and didn't have any gas sitting around? Or what if we forgot to open the hood and tried tinkering while it was still closed? We'd fall outside of our machine then - we'd have no knowledge about the situation, and (especially given how bad I am with cars) we'd be guaranteed never to reach an accept state. It's not clear exactly what would happen in such a situation, but it's definitely not something successful (of course, I highly recommend public transportation or self-propulsion like bicycling or walking; I don't like cars, but I do like biking).

If you stretch your imagination a bit, you might think of this diagram as describing successful patterns of actions and inputs. If I look for my keys and check the gas and put in more if it needs it and don't hurry too much etc. etc. etc., I'm guaranteed that my car will start (wouldn't it be nice if this were true in real life?) If I do something that's outside of my successful pattern - trying to kick the car before I even find my keys, for example - I'm pretty much stuck with a large, mostly metal paperweight in my driveway. In fact, if you were a clever theory of computation student, you might even say that this diagram defines a pattern for including certain inputs in a set of successful outputs.

And of course, that's what it is. Regular expressions were one way of representing sets of strings - formal languages - by describing a pattern that they had to match. If an input string matches, it's in the output set, and thus the language; if it doesn't match, it's still a valid string over some alphabet, but it's not in our current language of interest. Diagrams like the one we saw above are called deterministic finite automata, or DFAs for short (of course, since "automata" is already the plural of "automataon", DFAs isn't really a very good TLA, is it?), and they're just another way of describing patterns that input strings must match to be included in a formal language.

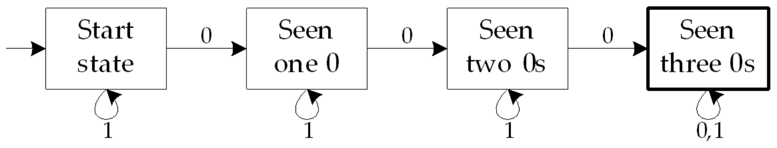
When describing a DFA formally, the states are still referred to as states (big surprise), and the labels on the transition arrows are always inputs - single symbols from our alphabet. For any given input string, we read it from left to right and take exactly one transition per symbol. If we ever hit a wall - that is, have no possible outgoing transitions

- the string is immediately rejected. Otherwise, when we get to the end of the string, we check to see if we're in an accepting state. If we are, the string is in our language, and otherwise, it's not (when this happens, the string is rejected).

None of this stuff ever makes sense when I just write down a description, so let's look at an example instead. Suppose we want to draw a DFA that accepts any string over $\{0,1\}$ containing at least three 0s. There are lots of ways we could write an RE for this, but one of the most intuitive might be:

$$1^*0(0+1)^*0(0+1)^*01^*$$

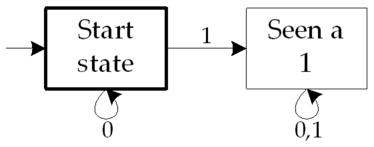
Let's draw a DFA that does the same thing:



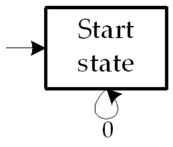
Another simple example is the DFA for, "All strings over $\{0,1\}$ containing no 1s." The RE is simple:

$$0^*$$

wnd the DFA isn't much worse:



which can be further abbreviated as:



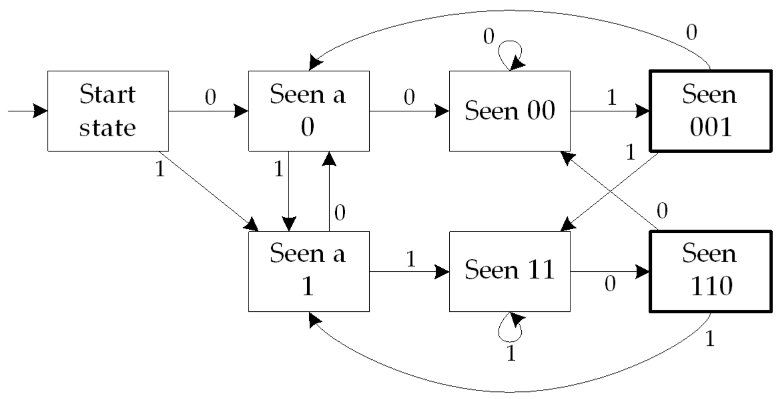
I would like, if I may (You may! How strange was it?), to point out a few rules being used to draw these DFAs:

- Each state has exactly one outgoing transition for each symbol in our alphabet.
- If a state is drawn with no outgoing transitions for some symbol, this is an abbreviation for running into the wall. The transition would go to a dead state, one from which no possible input could ever produce an acceptance. This is seen above in the "Seen a 1" state.
- If two or more symbols' transitions would go to the same state, you can draw one arrow and label it with multiple symbols separated by commas. Pretend that there are separate arrows, one for each symbol.
- The start state can also be an accept state (this is necessary for languages which include ϵ, such as 0^* above).
- You can have multiple accept states, but only one start state!

To illustrate this last point, let's see one more example. Suppose we want a DFA that accepts any string ending in 001 or in 110. As an RE, this is (again) not too bad:

$$(0+1)^*(001+110)$$

But this time, the DFA is a little trickier. Try the following on for size:



My word, that took forever to draw. And it's complicated! This is pretty much as tricky as a (reasonable) DFA should ever get. It's easy to see that it rejects anything shorter than three characters (including ϵ) and that it accepts 001 and 110. But what happens for longer strings, say, 11001? This input will follow the path "Start state", "Seen a 1", "Seen 11", "Seen 110", "Seen 00", and "Seen 001" to end in an accept state - because it ends with 001. Suppose instead we add one more character to get 110011. Now our path is the same, except that we proceed one state further to "Seen 11". This is *not* an accept state, so this string is not in our language. A string can pass through an accept state during processing; it is only accepted if it *ends* in an accept state.

19.3 Nondeterministic Finite Automata

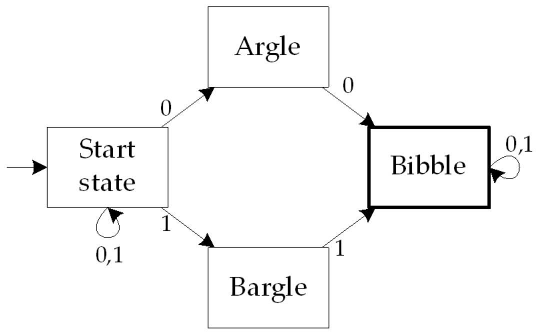
"Through it all threaded the realization that her son was the Kwisatz Haderach, the one who could be many places at once. He was the fact out of the Bene Gesserit dream. And the fact gave her no peace." - Frank Herbert, "Dune"

DFAs seem to be pretty much a sort of quasi-physical representation of a string classification machine. We could imagine some series of physical states (gears and levers and whatnot) that receive input and, eventually, do something if the string is accepted; in our very first car example, the "something" was "the car starts up". If you're careful, DFAs can also be represented entirely mathematically, and that this representation lends itself to some interesting proofs based on set theory and formal languages.

Interesting. Very interesting. And as all good children when given a new toy, our first question should be, "Can we break it?" For that matter, our zeroeth question should be, "Is it already broken?" Are DFAs It? Are they all that could ever be hoped for in string processing? A veritable Nirvana of symbol acceptance and rejection? Or, to put things a little more mathematically, suppose we have some alphabet A. A language L over A is just some subset of A^*. For some A, does there exist a language $L \subseteq A^*$ that cannot be described by some DFA over A (technically, a DFA isn't over an alphabet A, it's associated with the alphabet A; only strings can be over an alphabet, but everyone says it this way anyhow)?

There are a million ways to prove that yes, there are plenty of languages you can't represent using DFAs (arbitrarily many, in fact), but since I'm already trying to teach an entire semester's worth of course in one precept, I'll skip the proof here (consider the language $L = \{w | w = 0^n 1^n, n \in \mathbb{N}\}$). This answers one of our questions: DFAs are, in fact, already broken. We cannot represent any language which requires us to "remember" more than a specific, finite number of symbols in a row; anything that requires counting arbitrarily many symbols, for any reason, will eventually pump our DFA full and overflow. This means we can revise our first (remember, the other one was the zeroeth) question from, "Can we break it?" to, "Can we fix it? " We should suspect the answer to this already - computers, after all, are clearly able to recognize palindromes, addition, and other problems requiring counting - but the question (as usual) is how.

If by some chance you happen to remember our original definition of DFAs, you might notice that we only allow one transition per symbol per state - in fact, we *require* exactly on transition per symbol per state. Part of the deficiency of DFAs as described above involved the whole transitions-between-states thing, so one option would be to let each state have multiple transitions using the same symbol. This would let us draw something like:



This is easy to draw, but it's a lot harder to interpret. What does such a beast represent? What language might it encode (you should be able to figure this one out, at least; can you think of more descriptive names for each state?) We know each transition in a DFA represented, "Consume one character from the input string and move to the next step," but that doesn't seem to apply any more. How can we consume a 0 in the Start state and simultaneously move to Start and to Argle?

By doing exactly that! There are at least four equivalent ways to think of this process; I'll pick the one I like best first. Imagine that this machine is consuming some input string. In the Start state, if we see a 0, the string copies itself, places one copy in the Start state, and places one copy in the Argle state. Then each of these copies continues at the same time. Let's do an example using the string 0011:

- We start with the input 0011 in the Start state.
- We consume the first character, 0; since there are two possible destinations, the remainder of the string is *copied*, placing 011 in Start and 011 in Argle.
- We consume the second character, 0, from *both copies* simultaneously.
  - The copy of 011 in Start produces two more copies, a 11 in Start and a 11 in Argle.
  - The copy of 011 in Argle produces a 11 in Bibble.
- We consume the third character, 1, from all three strings simultaneously.
  - The copy of 11 in Start produces a 1 in Start and a 1 in Bargle.
  - The copy of 11 in Argle dies - if any copy of any string goes to a dead state, we can consider it to disappear completely.
  - The copy of 11 in Bibble produces a 1 in Bibble.
- We consume the fourth and last character, 1, from all three remaining copies.
  - The copy of 1 in Start produces an ϵ (and thus ends) in Start and in Bargle.
  - The copy of 1 in Bargle ends in Bibble.
  - The copy of 1 in Bibble ends in Bibble.

If *any* copy ends (or, equivalently, has zero characters left to consume, or, equivalently, produces an ϵ) in an accepting state, then the *whole string* is considered to be accepted. If we end up making 100 copies of a string, 99 of them die or end in non-accepting states, and one last copy ends in an accepting state, then the whole string is still considered to be in our language. Another way to think of this is as a union or an or - if the first copy or the second copy or the third copy or etc. etc. etc. is accepted, then the original string is accepted.

I told you that there were at least four ways to think about this duplication or simultaneous processing. Copying strings is the first. The second is copying the machine. Whenever a particular symbol has to go to two (or more) target states at the same time, pretend that the entire automaton copies itself - string and all. Then all of the copies of the machine keep running in parallel. If any machine ends in an accept state, then the original string is accepted (and thus in our language).

The third option is to think of the automaton as trying all possible paths in sequence. If there is ever a choice between two different target states, the automaton picks one and tries it. It repeats this over and over until it runs out of input characters. If it's in a final state, it's done - the string is accepted. If it's not, it "backs up" to the last decision it had to make and tries another option. It does this over and over until it tries all possible choices once; if any of them end in an accept state, then the string is accepted, and otherwise (if every possible combination of decisions results in rejection) it is not in our language.

The fourth way to think about this process is to think of the automaton as being "smart". If there is any possible combination of choices that will accept some input, that's the path that the string takes. Any time a choice presents itself, the machine takes only the path that will eventually result in the string being accepted; if there is no such path, then it just picks some path at random.

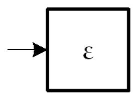
All four of these methods are equivalent. They are just different ways of thinking about a process called nondeterminism, which is a description of any process in which all choices are considered simultaneously. We called DFAs deterministic because in any state, we could specifically determine which state would be entered next. With these new automata, which we'll call NFAs (for "nondeterministic finite automata", of course), we cannot always make such a determination - multiple choices are possible, and every choice gets made eventually (or simultaneously, depending on how you think about it).

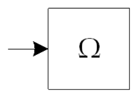
I'm trying to find a way to avoid having the remainder of the necessary material take up another 15 pages... you don't need any of the following for your assignment, so I'll compress it down as much as possible. If you're interested in theory, though, this is where all of the good stuff happens!

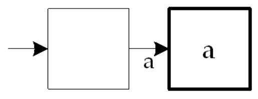
- You can add another trick to NFAs, ϵ-transitions, for free. As is, NFAs allow duplication of any input string arbitrarily many times per character consumed. ϵ-NFAs include ϵ-transitions that consume no characters; that is, they allow duplication of any input string arbitrarily many times without consuming any characters at all. ϵ-NFAs are strict supersets of NFAs (which are strict supersets of DFAs), so ϵ-NFAs can do at least as much as NFAs. This gives us DFAs $\subseteq$ NFAs $\subseteq$ ϵ-NFAs.

- The free part comes from the fact that you can show that this adds no extra power beyond normal nondeterministic transitions and that nondeterministic transitions add no extra power beyond deterministic transitions. The quick proof: suppose you have an ϵ-NFA with states $Q = \{q_0, q_1, ..., q_n\}$, start state q_0, and final states $F \subseteq Q$. Any input string can occupy at most $2^{|Q|}$ states, and each possible combination is just some member of the power set of Q, $P(Q)$ (the power set is the set of all subsets, so $P(Q) = \{\emptyset, \{q_0\}, \{q_1\}, ..., \{q_0, q_1\}, ..., \{q_0, q_1, ...,$ The power set of Q is exponentially larger, but it's still finite. Replace each nondeterministic transition between states in Q with a deterministic transition between states in $P(Q)$, and you've turned any ϵ-NFA (and thus any NFA) into an equivalent DFA (that is, one representing exactly the same language). This gives us ϵ-NFAs $\subseteq$ DFAs and NFAs $\subseteq$ DFAs, so DFAs $=$ NFAs $=$ ϵ-NFAs.

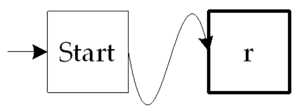
- We can now show that every RE has an equivalent DFA by showing that any regular expression has an equivalent ϵ-NFA. The simplest regular expressions are all singletons: Ω, Δ, or any symbol a from our alphabet. These base cases can be represented by the following three ϵ-NFAs, respectively:



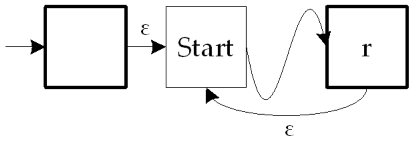


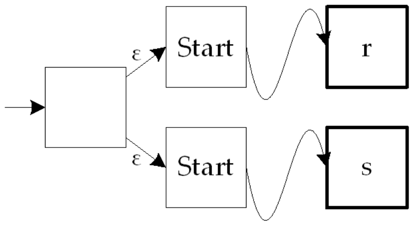


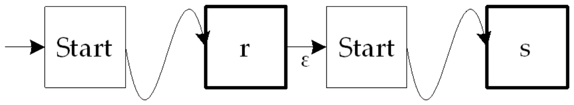
Given these base cases, you can build ϵ-NFAs representing each of the three regular expression operators inductively. Suppose that the following ϵ-NFA accepts the same language as some regular expression r:



Anything could happen between the start and final states; we just assume that the NFA is correct. Given such an r and a similar s, I propose that the Kleene closure r^*, union $r + s$, and concatenation rs operators are encoded by ϵ-NFAs as follows:

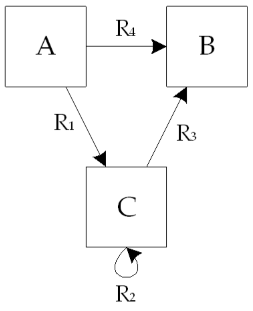




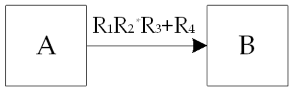


Assuming you believe this quick/dirty/sketchy/handwavy proofish type thing, this gives us REs $\subseteq$ DFAs.

- The proof that every DFA has an equivalent RE is even longer, so the version seen here will be even more questionable. First, let's define a generalized nondeterministic finite automaton or GNFA as an NFA in which every transition can contain not just one letter but any regular expression. Because we know REs are no more powerful than DFAs (based on our preivous proof), this doesn't add any power to NFAs. Similarly, every NFA is trivially a GNFA (modulo some other modifications traditionally included in the proof that I'm omitting here for brevity). This means that you can turn any DFA into an NFA into an ϵ-NFA into a GNFA. Now, given a GNFA with $n > 2$ states, we can always turn it into a GNFA with $n - 1$ states. Such a GNFA must contain a widget of the form:

  where R_1, R_2, R_3, and R_4 are all arbitrary regular expressions. Given such a widget, you can always eliminate state C by producing an equivalent widget of the form:

  where $R_1 R_2^* R_3 + R_4$ is just some longer regular expression made by combining the R_is. Given any GNFA at all, you can repeat this process until it contains exactly two states, one start state and one final state. At this point, the regular expression on the single remaining transition will be equivalent to the original GNFA! This gives us DFAs $\subseteq$ REs and thus REs = DFAs. Whew!

The upshot of this is that all of our various flavors of finite automata and regular expressions represent exactly the same class of languages. We refer to any language that can be represented by a regular expression as a regular language. And based on just the things we've done so far, we know a surprising number of cool things about regular languages:

- We can union regular expressions to get another RE, so the class of regular languages is closed under union.
- We can concatenate REs to get another RE, so the class of regular languages is closed under concatenation (or Cartesian product).
- We can repeat REs using the star operator to get another RE, so the class of regular languages is closed under Kleene closure (that sounds redundant, doesn't it?)
- We can complement D/NFAs using a fairly simple process that I often assign as a homework problem and thus haven't described here (insert evil laughter here), so the class of regular languages is closed under complementation. In other words $A^* - L$ is regular for any regular language L over any alphabet A.
- We can reverse REs and D/NFAs, so the class of regular languages is closed under reversal.

I could keep going, but by now, you're probably wondering, "Oh no! If everything we do produces more regular languages, and regular languages can't even count, how do we get out? How do we do better?" Or at least if you aren't wondering this, you should be! We originally defined REs as a formal language for representing formal languages - that is, a syntactical way of manipulating symbols in any formal language representing anything. DFAs were supposed to be more machine-like representations of REs - things that were like computer hardware. If we can't handle counting, we can't even add, let alone do all of the other crazy stuff a computer does. Yipe!

Fret not. There is a solution. The universe remains a consistent place, and we'll find a way to model nonregular languages. For starters, consider that our original problem with DFAs (and thus REs or NFAs) was that they ran out of memory - they had no way to count past some finite point roughly equivalent to the number of states. We tried modifying them to use nondeterministic choices - and that didn't help. Next, we'll try some ways of giving them a better memory instead (I want to modify *me* to have a better memory; I only have three neurons, so I can only remember to eat, sleep, and teach precept)... and maybe, just maybe, that'll help us make some progress.

19.4 Turing Machines

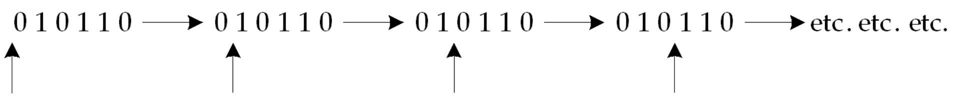
"Turing, your teeth are big and green,
Turing, we loved learning about your computing machines,
Turing, your proofs enduring,
We loved learning about you in CPS1." - Mark Ruben, written for Computer Science 1 (sung to the tune of the theme from The Bridge On The River Kwai)

Suppose that one day you were bored. Really bored. So bored that you were walking down the street, minding your own business, thinking, "I'm so bored!" And all of a sudden, some crazy logician wielding a pocket protector and a giant beard jumped out of the bushes. He rushes up to you and demands that you design a model capable of accepting the (apparently simple) language ww, for $w \in \{0, 1\}^*$. "After all," he says, "These dumb notes have been talking about it for the last few dozen pages. When are you going to walk the walk rather than just talk the talk?"

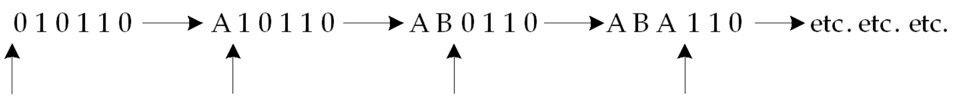
And with that, he disappears in a puff of logic.

Well, you being so bored and all, you decide to try it. I've already told you that a regular expression won't do it, and that should be fairly easy to believe; regular expressions themselves don't have any way to refer back to something they saw earlier, and DFAs can only remember a finite amount of what they saw earlier (and w might be arbitrarily large). There's actually a really elegant derivation combining a limited type of memory with a finite automaton (called a pushdown automaton) that shows that you can't represent this language well using linearly accessible memory.

However, since the lectures have already given away the punch line (and it would just take even more time to work up to it accurately here), we'll skip straight to the heart of things. Imagine, if you will, an old-style victrola attached to a large roll of toilet paper (I bet you never thought toilet paper was at the heart of things). A very, very large roll of toilet paper. In fact, you'd go so far as to claim that this roll contains infinitely many little paper squares. When we started out with plain old boring DFAs, we thought of them as machines that read through input strings. Sumewhere hidden behind the states and transitions was a little cursor reading through an input string, one symbol at a time:



Suppose we still have only one string of characters with one cursor on it, but we can write to it as well as read from it. This mysterious device starts out with our input on it, but we can change what's there as we run:



Think of each character as living in its own little square of toilet paper. We can wind back and forth on the roll, erasing and rewriting characters as we choose. We still have states, but when we make transitions, we read from exactly one location (the current tape position) and (optionally) write to exactly one location (the current tape position).

You should have many questions at this point. The first should be, "Where can I find that much toilet paper, and can I afford the storage space for infinitely much toilet paper? What if I sell it on Ebay? How much is an arbitrarily large quantity of wood pulp worth? Won't that devalue the dollar - for that matter, won't that devalue every currency on Earth? Has this toilet paper sparked an economic revolution?"

Well, those were the first several, I suppose. The next should be, "Wait a minute. Machine tape read write forth back and what?" As my default attempt to clarify these extremely difficult concepts, let's look at an example. I'm allowed to include the following in my new machine:

- A single storage space with a distinct left side that extends infinitely far to the right.
- The leftmost n characters of the storage tape begin containing our input string.
- Every space on the tape after that contains some "blank" symbol β.
- A bunch of states, just like any old DFA/NFA/PDA/AAA/FALALA.
- A read/write head on the tape (much like the imaginary read-only head that consumed input for a DFA).
- On each transition, we have...
  - Input: The current (source) state (just like a DFA's state).
  - Input: The character on the tape at the head's current position (just like a DFA's transition label).
  - Output: The new (target) state (just like a DFA transition's target).
  - Output: The character to be written to the tape at the head's current position (this one's new!)
  - Output: The direction to move the head, left or right (also new).
- We represent each transition as an arrow, just as before, labeled with $a \to b, c$ to represent, "read the symbol a from the tape, write the symbol b to the tape, and move in the direction c." c can be either L or R, and b can be equal to a if we want to leave the tape unmodified.
- Since we no longer have some distinct finite input string we're reading in, we require exactly one explicit accept state and one explicit reject state. If we ever enter either of these states, we do as we're told (accept or reject) and stop processing immediately. Nondeterminism is right out; we assume (for now) that we can always move exactly once per state per character, and the only way to stop is by entering one of these two states.

That little explanation should have succeeded in making the previously clear-as-mud situation into one with approximately the phototransmission properties of your average brick wall. On to that example that I promised would clear everything up. In lieu of ww, we'll tackle the slightly simpler language $w2w$, for $w \in \{0,1\}^*$. Let's describe textually the approximate process we'd like our machine to follow:

- We start with an input tape containing our string followed by infinitely many βs (blanks). Our read/write head is on the first character.

  $\underline{0}\ 0\ 1\ 0\ 2\ 0\ 0\ 1\ 0\ \beta\ \beta$ ...

  The underline here (and from now on) represents our read/write head.

- We mark the first character, scan right to the 2, and then mark the first matching character afterwards (i.e. if we don't match, we reject).

  $\#\ 0\ 1\ 0\ 2\ \underline{\#}\ 0\ 1\ 0\ \beta\ \beta$ ...

- We scan back to the left, past the 2, to the #. Then we move one to the right:

  $\#\ \underline{0}\ 1\ 0\ 2\ \#\ 0\ 1\ 0\ \beta\ \beta$ ...

- Now we repeat the process. We mark the current character, scan right to the 2, scan right to the first non-# character, and mark it (assuming it matches; otherwise, we reject).

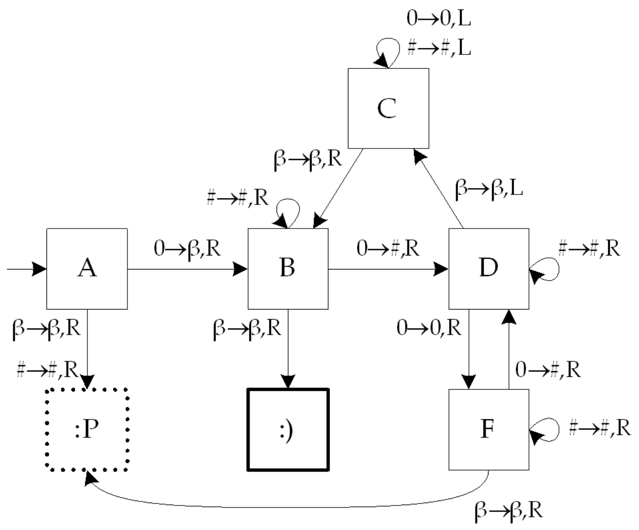
  $\#\ \#\ 1\ 0\ 2\ \#\ \underline{\#}\ 1\ 0\ \beta\ \beta$ ...

- We repeat this process until we've marked everything to the left of the 2.

  $\#\ \#\ \#\ \#\ \underline{2}\ \#\ \#\ \#\ \#\ \beta\ \beta$ ...

- We move right until we run out of #s. If we're on a β and not some extra 0s or 1s or something, we accept.

Ok, so that looks plausible. To get back to the pretty pictures, which I much prefer, consider the unary language 0^{2^n}, or all strings of 0s with length a power of two (e.g. 0, 00, 0000, 00000000, etc.) We can show this language isn't regular, and I'm fairly certain that it's not context free, although I don't have a proof for that handy. In any case, it's a language, and we might as well build a toilet paper machine for it. Because after all, what good is computer science if you can't build toilet paper doobobs out of it?



This machine isn't actually all that different from what you'd want to handle $w2w$, but... let's just say that this is simpler. This is pretty much the simplest toilet paper machine I know of, for that matter, and it's still crazy complicated. We can run some strings through it, and that might sort of help...

Suppose we start with 0000 on the tape. We can describe the entire sequence of <u>configurations</u> of the machine, including the current state, head position, and tape contents:

A: $\underline{0}000\beta$	C: $\underline{\beta}\#0\#\beta$	C: $\underline{\beta}\#\#\#\beta$
B: $\beta\underline{0}00\beta$	B: $\beta\underline{\#}0\#\beta$	B: $\beta\underline{\#}\#\#\beta$
D: $\beta\#\underline{0}0\beta$	B: $\beta\#\underline{0}\#\beta$	B: $\beta\#\underline{\#}\#\beta$
F: $\beta\#0\underline{0}\beta$	D: $\beta\#\#\underline{\#}\beta$	B: $\beta\#\#\underline{\#}\beta$
D: $\beta\#0\#\underline{\beta}$	D: $\beta\#\#\#\underline{\beta}$	B: $\beta\#\#\#\underline{\beta}$
C: $\beta\#0\underline{\#}\beta$	C: $\beta\#\#\underline{\#}\beta$	:)
C: $\beta\#\underline{0}\#\beta$	C: $\beta\#\underline{\#}\#\beta$	
C: $\beta\underline{\#}0\#\beta$	C: $\beta\underline{\#}\#\#\beta$	

Suppose we start with 000 on the tape:

A: $\underline{0}00\beta$
B: $\beta\underline{0}0\beta$
D: $\beta\#\underline{0}\beta$
F: $\beta\#0\underline{\beta}$
:P

Well, that seems to work pretty much the way we expect it to. Thank goodness for small blessings. But we still don't really know what sort of beast we have here!

A brief historical interlude. As some of you may know, somewhere ranging from about the mid-1930s until the lateish 1940s, the United States was involved in a little scuffle with pretty much every other national power worldwide. The aptly named World War II resulted in an astonishing number of things, including velcro (named as a combination of the words "velour" and "crochet", or approximately (in translation) "fabric that's been twisted together") and frisbees. Pretty much anyone who was someone was involved in the war effort, but a lot of the important someones were classified for some, all, or a large superset of the war. A lot of names you probably recognize worked on some sort of New York housing project, including Oppenheimer, Fermi, Feynman, Dyson, and a whole collection of other physicists. A lot of names you probably don't recognize worked on some sort of British cricket project, including Welchman and Turing.

Well, that one you probably recognize, at least from lecture. He's been popularized lately by such ride-the-geek-wave authors as Neil Stephenson. Alan Turing gained most of his current recognition when his role in World War II became (mostly) declassified in the 1970s. You see, the Germans didn't really want everyone else to know where they were moving troops, planning to attack, what they were having for dinner, listening in to phone calls to their parents, that sort of thing. But pretty much every one else in the world *did* want to eavesdrop on the Axis forces.

Turing was the one who succeeded, along with a surprisingly gigantic (several thousand, all told) team of engineers, operators, mathematicians, and logicians. The short version is that he made an excellent theoretical study of encrypted German text and developed an electrical gidgy to decode it known as the Enigma machine. The long version is told in any cryptography class worth its salt, so I'll skip it here.

Because we don't really care about Turing saving civilization as we know it. We care about his young, carefree days as a graduate student right here at Princeton. Between jaunts around campus and through the surrounding woods (you should check out the walking trails down by the Institute for Advanced Study), Turing published a paper called, "On Computable Numbers".

Now this is interesting. Today, that's not so thrilling; everyone does computation with numbers. In *1936*, though, it was pretty unique. This paper introduced the idea of a Turing machine, a 100% theoretical construct that was mathematically rigorously defined and could provably generate all sorts of interesting behavior. It described formal languages not acceptable by simpler mathematical constructs, and it was (soon) shown to be equivalent to another algorithmic formalism referred to as the λ-calculus. This was another logical representation of calculus created not many years earlier by Turing's advisor, Alonzo Church.

Now Alonzo and Alan, upon generating these equivalently powerful formalisms, made the following claim, known as the Church-Turing Thesis: any process that can be performed by a finite number of "mechanical" or "syntactic" steps can be performed by a Turing machine (or, equivalently, by the λ-calculus). The Thesis is often phrased in such a way as to extend to a sort of philosophical belief: any process that can be performed by any rational being in a finite amount of time can be performed by a Turing machine. We're significantly more concerned with the former, formal version, but the latter makes for some interesting philosophical conversations.

One big difference between a Turing machine and our earlier automata is the way that input is handled. Just as we saw above, there is no separate "input string" for a Turing machine the way there is for a D/NFA. Instead, we start with the input on the tape. In fact, we start such that:

- For input s, the leftmost $|s|$ symbols on the tape are drawn from the language alphabet A.
- All symbols starting at $|s|$ and extending infinitely far to the right are β, the blank symbol.
- The read/write marker starts at the leftmost (first) tape cell.

Since we're not longer advancing a separate marker through symbols in the input string, we need to redefine acceptance and rejection. We can no longer simply stop if there are no characters left in the input string, since now we can move the tape marker around at will. So we define the following:

- A Turing machine accepts an input string as soon as it enters its unique accept state. All further processing stops at this point, the string is included in the TM's language, and the TM is said to have halted.
- A Turing machine rejects an input string as soon as it enters its unique reject state. All further processing stops, the string is not included in the TM's language, and the TM is said to have halted.

But there's a *third* option! What about a Turing machine that somehow, say, just keeps moving to the right and writing #s onto the tape? It'll never repeat the same action twice - it'll never write in the same tape cell twice. But it will also never accept and never reject. This Turing machine is said to be looping; it's not necessarily repeating exactly the same thing over and over, but it's not stopping, either. Because of this third option, we have to be careful how we discuss the languages described by Turing machines:

- A language is decidable if we can define a Turing machine to describe it that will halt (either accept or reject) for all inputs.
- A language is recognizable if we can define a Turing machine to describe it that will halt for input strings in the language and either halt or loop for input strings not in the language.

So if a language is decidable, it is also recognizable. But a language can be recognizable without being decidable! Suppose we want some language that describes integer roots of polynomials. That is, the polynomial

$$6x^3yz^2 + 3xy^2 - x^3 - 10 = 0$$

has integer roots $x = 5$, $y = 3$, and $z = 0$. If we ask a Turing machine, "Does the equation $6x^3yz^2+3xy^2-x^3-10 = 0$ have integer roots?" we'll get an answer (yes) and it will halt. But in the general case, there are polynomials for which the TM will not halt; good polynomials (ones with integer roots, i.e. ones in our language) are accepted, but bad polynomials will sometimes make the TM run forever. This language is recognizable but not decidable.

There are a hundred and one different things you can do once you've defined Turing machines like this. According to the Church-Turing thesis, *nothing* can do better than a Turing machine as we've described it. It's surprisingly easy to show that many potential modifications to Turing machines add no power: doubly infinite tapes, multiple tapes, multiple accept/reject states, and even nondeterminism all fail to describe additional languages (although they do change the amount of work necessary to represent a language, just like NFAs are potentially exponentially smaller than their corresponding DFAs). Similarly, you can show that just about any useful "program" can be encoded as a language to be decided by a Turing machine. Consider the set of strings:

010100
0110
0100010000
000100100000
0000100010000000

Look like it might be useful for a computer to decide that language? We can do pretty much the same thing for any arithmetic concept, or for graphs or graphics or natural languages or computer programs or *anything*; if you're careful, you can conceive of a language that encodes nearly any problem you'd like to solve.

Taken as a philosophical question about human cognition, the Church-Turing thesis is even more interesting, because Turing machines have limits! In the example above, I may have convinced you that finding integral roots of polynomials (in more than one variable) is hard for a Turing machine. It's certainly a hard problem for humans, and correspondingly, it's recognizable *but not decidable* for Turing machines. This conflict - decidability versus recognizability versus neither - is extremely important in describing the limits of computer and, if you believe the broader version of the Church-Turing thesis, of human thought as well. So even if you don't care about computers and theory that much, it can't hurt to think about thought a little!