

CSE6390 3.0 Special Topics in AI & Interactive Systems II
Introduction to Computational Linguistics
Instructor: Nick Cercone – 3050 CSEB – nick@cse.yorku.ca
Tuesdays, Thursdays 10:00-11:30 – South Ross 104
Fall Semester, 2010

Regular Expressions, Finite State Machines and The Pumping Lemma

Regular Expressions

A regular expression (regex), often called a [pattern](#), is an expression that describes a set of strings. They are usually used to give a concise description of a [set](#), without having to list all elements. For example, the set containing the three strings "*Handel*", "*Händel*", and "*Haendel*" can be described by the pattern $H(\ddot{a}|ae?)ndel$ (or alternatively, it is said that the pattern [matches](#) each of the three strings). In most formalisms, if there is any regex that matches a particular set then there are an infinite number of such expressions. Most formalisms provide the following operations to construct regular expressions.

Abbreviated History: The origins of regular expressions lie in [automata theory](#) and [formal language theory](#), both of which are part of theoretical computer science, which study models of computation (automata) and ways to describe and classify formal languages. In the 1950s, mathematician Steven Kleene described these models using his mathematical notation called [regular sets](#). The SNOBOL language was an early implementation of [pattern matching](#), but not identical to regular expressions. Ken Thompson built Kleene's notation into the editor [QED](#) as a means to match patterns in text files. He later added this capability to the Unix editor [ed](#), which eventually led to the popular search tool [grep](#)'s use of regular expressions ("grep" is a word derived from the command for regular expression searching in the ed editor: $g/re/p$ where *re* stands for regular expression). Since that time, many variations of Thompson's original adaptation of regular expressions have been widely used in Unix and Unix-like utilities including [expr](#), [AWK](#), [Emacs](#), [vi](#), and [lex](#).

[Perl](#) and [Tcl](#) regular expressions were derived from a regex library written by Henry Spencer though Perl later expanded on Spencer's library to add many new features. Philip Hazel developed [PCRE](#) (Perl Compatible Regular Expressions), which attempts to closely mimic Perl's regular expression functionality, and is used by many modern tools including [PHP](#) and [Apache HTTP Server](#). Part of the effort in the design of [Perl 6](#) is to improve Perl's regular expression integration, and to increase their scope and capabilities to allow the definition of [parsing expression grammars](#). The result is a mini-language called [Perl 6 rules](#), which are used to define Perl 6 grammar as well as provide a tool to programmers in the language. These rules maintain existing features of Perl 5.x regular expressions, but also allow [BNF](#)-style definition of a [recursive descent parser](#) via sub-rules.

The use of regular expressions in structured information standards for document and database modeling started in the 1960s and expanded in the 1980s when industry standards like [ISO SGML](#) (precursor was ANSI "GCA 101-1983") consolidated. The kernel of the [structure specification language](#) standards are regular expressions.

Formally: Definition: Regular expressions can be defined in [formal language theory](#). Regular expressions consist of constants and operators that denote sets of strings and operations over these sets, respectively. The following definition is standard, and found as such in most textbooks on formal language theory. Given a finite [alphabet](#) Σ , the following constants are defined:

- (*empty set*) Φ denoting the set
- (*empty string*) ϵ denoting the "empty" string, with no characters at all.
- (*literal character*) a in Σ denoting a character in the language.

The following operations are defined:

- (*concatenation*) RS denoting the set $\{\alpha\beta \mid \alpha \text{ in } R \text{ and } \beta \text{ in } S\}$. For example $\{\text{"ab"}, \text{"c"}\}\{\text{"d"}, \text{"ef"}\} = \{\text{"abd"}, \text{"abef"}, \text{"cd"}, \text{"cef"}\}$.
- (*alternation*) $R \mid S$ denoting the **set union** of R and S . For example $\{\text{"ab"}, \text{"c"}\} \mid \{\text{"ab"}, \text{"d"}, \text{"ef"}\} = \{\text{"ab"}, \text{"c"}, \text{"d"}, \text{"ef"}\}$.
- (*Kleene star*) R^* denoting the smallest **superset** of R that contains ϵ and is **closed** under string concatenation. This is the set of all strings that can be made by concatenating zero or more strings in R . For example, $\{\text{"ab"}, \text{"c"}\}^* = \{\epsilon, \text{"ab"}, \text{"c"}, \text{"abab"}, \text{"abc"}, \text{"cab"}, \text{"cc"}, \text{"ababab"}, \text{"abcab"}, \dots\}$.

To avoid parentheses it is assumed that the Kleene star has the highest priority, then concatenation and then set union. If there is no ambiguity then parentheses may be omitted. For example, $(ab)c$ can be written as abc , and $a|(b(c^*))$ can be written as $a|bc^*$. Many textbooks use the symbols $*$, $+$, or $|$ for alternation instead of the vertical bar.

Examples:

- $a|b^*$ denotes $\{\epsilon, a, b, bb, bbb, \dots\}$
- $(a|b)^*$ denotes the set of all strings with no symbols other than a and b , including the empty string: $\{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$
- $ab^*(c|\epsilon)$ denotes the set of strings starting with a , then zero or more b s and finally optionally a c : $\{a, ac, ab, abc, abb, abbc, \dots\}$

Expressive power and compactness: The formal definition of regular expressions is purposely parsimonious and avoids defining the redundant quantifiers $?$ and $+$, which can be expressed as follows: $a+ = aa^*$, and $a? = (a|\epsilon)$. Sometimes the **complement** operator is added; R^c denotes the set of all strings over Σ^* that are not in R . In principle, the complement operator is redundant, as it can always be circumscribed by using the other operators. However, the process for computing such a representation is complex, and the result may require expressions of a size that is double exponentially larger.

Regular expressions in this sense can express the **regular languages**, exactly the class of languages accepted by **deterministic finite automata**. There is, however, a significant difference in compactness. Some classes of regular languages can only be described by deterministic finite automata whose size grows exponentially in the size of the shortest equivalent regular expressions. The standard example are here the languages L_k consisting of all strings over the alphabet $\{a, b\}$ whose k^{th} -last letter equals a . For example, a regular expression describing L_4 is given by $(a|b)^* a(a|b)(a|b)(a|b)$.

It is known that every deterministic finite automaton accepting the language L_k must have at least $2k$ many states. Luckily, there is a simple mapping from regular expressions to the more general **nondeterministic finite automata** (NFAs) that does not lead to such a blowup in size; for this reason NFAs are often used as alternative representations of regular languages. NFAs are a simple variation of the type-3 grammars of the **Chomsky hierarchy**.

Finally, it is worth noting that many real-world "regular expression" engines implement features that cannot be described by the regular expressions in the sense of formal language theory.

Some Definitions

Finite state machine

Definition: A *model of computation* consisting of a set of *states*, a *start state*, an input *alphabet*, and a *transition function* that maps input symbols and current states to a *next state*. Computation begins in the start state with an input string. It changes to new states depending on the transition function. There are many variants, for instance, machines having actions (outputs) associated with transitions (*Mealy machine*) or states (*Moore machine*), multiple start states, transitions conditioned on no input symbol (a null) or more than one transition for a given symbol and state (*nondeterministic finite state machine*), one or more states designated as *accepting states* (*recognizer*), etc. Also known as *finite state automaton*.

Turing machine

Definition: A *model of computation* consisting of a *finite state machine* controller, a read-write head, and an unbounded sequential tape. Depending on the current *state* and symbol read on the tape, the machine can change its state and move the head to the left or right. Unless otherwise specified, a Turing machine is *deterministic*.

Deterministic finite state machine

Definition: A *finite state machine* with at most one transition for each symbol and *state*. Also known as *DFA*, *deterministic finite automaton*.

Nondeterministic finite state machine

Definition: A *finite state machine* whose *transition function* maps inputs symbols and *states* to a (possibly empty) set of *next states*. The transition function also may map the null symbol (no input symbol needed) and states to next states. Also known as *NFA*, *nondeterministic finite automaton*.

Mealy machine

Definition: A *finite state machine* which produces an output for each *transition*.

Moore machine

Definition: A *finite state machine* that produces an output for each *state*.

Markov chain

Definition: A *finite state machine* with probabilities for each transition, that is, a probability that the next state is s_j given that the current state is s_i .

Hidden Markov model

Definition: A variant of a *finite state machine* having a set of *states*, Q , an output *alphabet*, O , transition probabilities, A , output probabilities, B , and initial state probabilities, Γ . The current state is not observable. Instead, each state produces an output with a certain probability (B). Usually the states, Q , and outputs, O , are understood, so an HMM is said to be a triple, (A, B, Γ) .

Formal Definition:

- $A = \{a_{ij} = P(q_j \text{ at } t+1 \mid q_i \text{ at } t)\}$, where $P(a \mid b)$ is the conditional probability of a given b , $t \geq 1$ is time, and $q_i \in Q$.
Informally, A is the probability that the next state is q_j given that the current state is q_i .
- $B = \{b_{ik} = P(o_k \mid q_i)\}$, where $o_k \in O$.
Informally, B is the probability that the output is o_k given that the current state is q_i .
- $\Gamma = \{p_i = P(q_i \text{ at } t=1)\}$.
Also known as *HMM*.

Equivalence of finite state machines and regular expression languages

A language is given by a regular expression if and only if it is a language of some finite state machine. If a language is given by one of these two ways, we can always convert to the other if this is more convenient.

- regular expressions can be easier to specify textually
- it is easier to check whether a string is accepted by a finite state machine

Some finite state machine accepts every language given by a regular expression. By induction on the structure of regular expressions, we construct the finite state machine that accepts this language.

For the base case, observe that we can easily construct finite state machines for empty language ϵ , and a finite state machine for a singleton language a for $a \in \Sigma$.

For the inductive step, we use closure properties of finite state machines for the cases of union, concatenation, and iteration.

Every language accepted by a finite state machine is given by some regular expression

Finite state machines with regular expression labels. We first generalize the notion of a finite state automaton so that we can label its edges not only with elements of Σ as in the standard definition of finite state machine and with epsilon transitions as in finite state machine with epsilon transitions, but with arbitrary regular expressions.

An accepting execution for such a generalized finite state machine is a sequence of states and regular expressions $q_0, r_1, r_2, \dots, r_n, q_n$ with $q \in F$, and the accepted strings of that execution are all strings in the union of $L(r_1, r_2, \dots, r_n)$ over all such accepting sequences.

Note that if all regular expressions are elements of Σ , the definition reduces to the standard definition of finite state machine.

Preparing for conversion to regular expression: To convert a finite state machine into a regular expression, we first view it as a generalized finite state machine, and then eliminate states of the state

machine one by one. We start the process by creating a fresh initial state q_i and fresh final state q_f and expressing all final states in F by transitions to q_f . We let $\{q_f\}$ be the new set of final states. For any pair of states, we then ensure that there is exactly one edge between them:

- if there was no edge, we introduce an edge with the label is \varnothing ;
- if there are multiple edges with labels a_1, \dots, a_n , we introduce instead one edge whose label is the regular expression $a_1 / \dots / a_n$.

Elimination step: We show how to eliminate a state q (we assume q is not initial and not a final state).

- let the self-loop edge labelled q be r
- for every two states q_1 and q_2 (possibly equal), distinct from q :
- suppose we have these regular expressions on edges:

$$q_1 \xrightarrow{r_1} q \xrightarrow{r_1} q_2$$

- extend label from q_1 to q_2 with $r_1 r^* r_1$

At the end, we are left with one non-empty edge from q_i to q_f , whose label is the desired regular expression.

Exercise

Consider alphabet $\Sigma = \{a, b\}$. A string $s \in \Sigma^*$ is *desperate* if it contains 'aaa' as a substring. Construct a regular expression that describes the set of all strings that are **not** desperate.

One solution: $((\varepsilon/a/aa)b)^ (\varepsilon/a/aa)$*

What, in simple terms, is the pumping lemma?

To put in layman's terms is difficult, but basically regular expressions should have a non-empty substring within the expression that can be repeated as many times as you wish while the entire "new formed word" remains valid for the language.

In practice, pumping lemmas are not sufficient to *prove* a language correct, but rather as a way to do a proof by contradiction to show a language does not fit in the class of languages (Regular or Context-Free) by showing the pumping lemma does not work for the language.

The answer above is acceptable, but the answer does not *feel* like it explains the *purpose* of the pumping lemma.

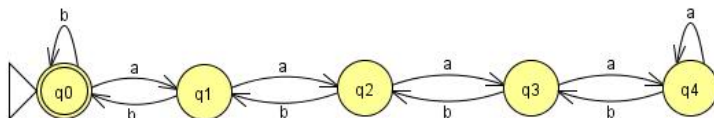
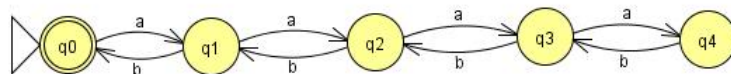
The pumping lemma is a simple proof to show that a language is not regular, meaning that a Finite State Machine cannot be built for it. The canonical example is the language $(a^n)(b^n)$. This is the simple language which is just any number of a's, followed by the same number of b's. So the strings

ab aabb aaabbb aaaabbbb etc. are in the language, but

aab bab aaabbbbb etc. are not.

It's simple enough to build a FSM for these examples:

This one will work all the way up to $n=4$. The problem is that our language didn't put any constraint on n , and Finite State Machines have to be, well, finite. No matter how many states I add to this machine, someone can give me an input where n equals the number of states plus one and my machine will fail. So if there can be a machine built to read this language, there must be a loop somewhere in there to keep the number of states finite. With these loops added:



with (a^*) representing any all of the strings in our language will be accepted, but there is a problem. After the first four as, the machine

loses count of how many a's have been input because it stays in the same state. That means that after four, I can add as many a's as I want to the string, without adding any b's, and still get the same return value. This means that the strings

aaaa(a*)bbbb

number of a's, will all be accepted by the machine even though they obviously aren't all in the language. In this context, we would say that the part of the string (a*) can be pumped. The fact that the Finite State Machine is finite and n is not bounded, guarantees that any machine which accepts all strings in the language *must* have this property. The machine must loop at some point, and at the point that it loops the language can be pumped. Therefore no Finite State Machine can be built for this language, and the language is not regular.

Remember that regular expressions and finite state machines are equivalent, then replace a and b with opening and closing Html tags which can be embedded within each other, and you can see why it is not possible to use regular expressions to parse Html

Another try

By definition regular languages are those recognized by a finite state automaton. Think of the FSM as a labyrinth: states are rooms, transitions are one-way corridors between rooms, there's an initial room, and an exit (final) room. As the name 'finite state automaton' says, there is a finite number of rooms. Each time you travel along a corridor, you jot down the letter written on its wall. A word can be recognized if you can find a path from the initial to the final room, going through corridors labelled with its letters, in the correct order.

The pumping lemma says that there is a maximum length (the pumping length) for which you can wander through the labyrinth without ever going back to a room through which you have gone before. The idea is that since there are only so many distinct rooms you can walk in, past a certain point, you have to either exit the labyrinth or cross over your tracks. If you manage to walk a longer path than this pumping length in the labyrinth, then you are taking a detour : you are inserting a(t least one) cycle in your path that could be removed (if you want your crossing of the labyrinth to recognize a smaller word) or repeated (pumped) indefinitely (allowing to recognize a super-long word).

There is a similar lemma for context-free languages. Those languages can be represented as word accepted by pushdown automata, which are finite state automata that can make use of a stack to decide which transitions to perform. Nonetheless, since there is still a finite number of states, the intuition explained above carries over, even through the formal expression of the property may be slightly *more complex*.

Sleep on it. An experiment conducted by German neurologists and documented in *Nature*, volume 427 (2004), p. 352 hypothesizes that students who get more sleep are able to solve tricky problems better than students who are sleep deprived. The problem they used involves a string consisting of the three digits 1, 4, and 9. "Comparing" two digits that are the same yields the original digit; comparing two digits that are different yields the missing digit. For example $f(1, 1) = 1$, $f(4, 4) = 4$, $f(1, 4) = 9$, $f(9, 1) = 4$. Compare the first two digits of the input string, and then repeatedly compare the current result with the next digit in the string. Given a specific string, what number do you end up with? For example, if the input is string 11449494, you end up with 9.

1 1 4 4 9 4 9 4
1 9 1 4 4 1 9

Postscript: Parsing Html The Cthulhu Way

*Cthulhu is a fictional cosmic entity created by horror author H. P. Lovecraft in 1926, first appearing in the short story "The Call of Cthulhu" when it was published in *Weird Tales* in 1928.*

Among programmers of any experience, it is generally regarded as A Bad Idea™ to attempt to parse HTML with regular expressions. How bad of an idea? It apparently drove one Stack Overflow user to the brink of madness:

You can't parse HTML with regex. Because HTML can't be parsed by regex. Regex is not a tool that can be used to correctly parse HTML. As I have answered in HTML-and-regex questions here so many times before, the use of regex will not allow you to consume HTML.

Regular expressions are a tool that is insufficiently sophisticated to understand the constructs employed by HTML. HTML is not a regular language and hence cannot be parsed by regular expressions. Regex queries are not equipped to break down HTML into its meaningful parts. so many times but it is not getting to me. Even enhanced irregular regular expressions as used by Perl are not up to the task of parsing HTML. You will never make me crack. HTML is a language of sufficient complexity that it cannot be parsed by regular expressions.

Even Jon Skeet cannot parse HTML using regular expressions. Every time you attempt to parse HTML with regular expressions, the unholy child weeps the blood of virgins, and Russian hackers pwn your webapp. Parsing HTML with regex summons tainted souls into the realm of the living. HTML and regex go together like love, marriage, and ritual infanticide. The `<center>` cannot hold it is too late. The force of regex and HTML together in the same conceptual space will destroy your mind like so much watery putty. If you parse HTML with regex you are giving in to Them and their blasphemous ways which doom us all to inhuman toil for the One whose Name cannot be expressed in the Basic Multilingual Plane, he comes.



That's right, if you attempt to parse HTML with regular expressions, you're succumbing to the temptations of the dark god Cthulhu's ... er ... code.

This is all good fun, but the warning here is only partially tongue in cheek, and it is born of [a very real frustration](#).

I have heard this argument before. Usually, I hear it as justification for seeing something like the following code:

```
# pull out data between <td> tags
($stable_data) = $html =~ /<td>(.*?)</td>/gis;
```

"But, it works!" they say "It's easy!" - "It's quick!" - "It will do the job just fine!"

I berate them for not being lazy. You need to be lazy as a programmer. *Parsing HTML is a solved problem. You do not need to solve it. You just need to be lazy.* Be lazy, use CPAN and use [HTML::Sanitizer](#). It will make your coding easier. It will leave your code more maintainable. You won't have to sit there hand-coding regular expressions. Your code will be more robust. You won't have to bug fix every time the HTML breaks your crappy regex

For many novice programmers, there's something unusually seductive about parsing HTML the Cthulhu way instead of, y'know, using a library like a sane person. Which means this discussion gets reopened almost every single day on Stack Overflow. The above post from five years ago could be a discussion from yesterday. I think we can forgive a momentary lapse of reason under the circumstances.

Like I said, this is a well understood phenomenon in most programming circles. However, I was surprised to see a few experienced programmers [in metafilter comments](#) actually *defend the use of regular expressions to parse HTML*. I mean, they've heeded the [Call of Cthulhu](#) ... and liked it. Many programs will neither need to, nor should, anticipate the entire universe of HTML when parsing. In fact, designing a program to do so may well be a completely wrong-headed approach, if it changes a program from a few-line script to a bullet-proof commercial-grade program which takes orders of magnitude more time to properly code and support. Resource expenditure should always (oops, make that very frequently, I about overgeneralized, too) be considered when creating a programmatic solution.

In addition, hard boundaries need not always be an HTML-oriented limitation. They can be as simple as "work with these sets of web pages", "work with this data from these web pages", "work for 98% users 98% of the time", or even "OMG, we have to make this work in the next hour, do the best you can".

We live in a world full of newbie PHP developers doing the first thing that pops into their collective heads, with more born every day. What we have here is an ongoing education problem. The real enemy isn't regular expressions (or, for that matter, [goto](#)), but ignorance. The only crime being perpetrated is not knowing what the alternatives are.

So, while I may attempt to parse HTML using regular expressions **in certain situations**, I go in knowing that:

- It's generally a bad idea.
- Unless you have discipline and put very strict conditions on what you're doing, matching HTML with regular expressions rapidly devolves into madness, just how Cthulhu likes it.
- I had what I thought to be good, rational, (semi) defensible reasons for choosing regular expressions in this specific scenario.

It's considered good form to demand that regular expressions be considered verboten, totally off limits for processing HTML, but I think that's just as wrongheaded as demanding every trivial HTML processing task be handled by a full-blown parsing engine. It's more important to understand the tools, and their strengths and weaknesses, than it is to knuckle under to knee-jerk dogmatism.

So, yes, generally speaking, it is a bad idea to use regular expressions when parsing HTML. We should be teaching neophyte developers that, absolutely. Even though it's an apparently never-ending job. But we should also be teaching them the very real difference between **parsing HTML** and the simple expedience of processing a few strings. And how to tell which is the right approach for the task at hand.

Whatever method you choose -- just don't leave the <cthulhu> tag open, for humanity's sake.

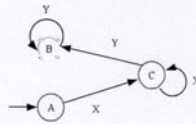


Selected readings:

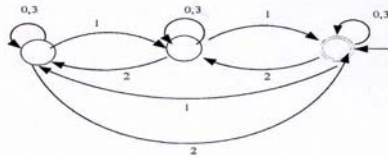
- J E Hopcroft and J D Ullman: *Introduction to Automata Theory, Languages and Computation* (Addison-Wesley, 1979)
- M D Davis and E.J.Weyuker: *Computability, Complexity and Languages* (Academic Press, 1983).
- T A Sudkamp: *Languages and Machines* (Addison-Wesley, 1988)
- J G Brookshear: *Theory of Computation: Formal Languages, Automata and Complexity* (Benjamin-Cummings, 1989)
- J. Carroll and D. Long: *Theory of finite automata* (Prentice Hall, 1989).
- J. M. Howie: *Automata and Languages* (Oxford Science Publications, 1991).
- M D Davis, R. Sigal, and E.J.Weyuker: *Computability, Complexity and Languages* (Academic Press, 1994).

Appendix 1: Regular Expressions and FSAs

A Finite State Automation (FSA) has four components: an input alphabet (those letters or strings which are legal inputs); a set of transition rules to advance from state to state (given a current state and an element of the input alphabet, what is the next state); a unique start state; and one or more final states. We can draw the FSA, as shown below, by representing each state as a circular node; the final state as a double circle; the start state as the only node with an incoming arrow; and the transition rules by the strings on the edges connecting the nodes. When labels are assigned to states, they appear inside the circle representing the state.



If there is a path from the start state to a final state by which an input string can be parsed, then the input string is said to be "accepted" by the FSA. The FSA above will accept strings composed of one or more x 's followed by one or more y 's (e.g., xy , xyx , $xxxxyy$, $xyyy$). A more complicated example is given below. The input alphabet is the digits 0, 1, 2 and 3. This FSA accepts those strings whose base 4 value is a multiple of 3. It does this by summing the value of the digits: when in the left node, the running sum of the digits has a value, modulo 3, of "1"; in the middle node, "2", and in the right node, "0". A base four number, like a base ten number, is a multiple of 3 only when its digits sum to a multiple of 3.



Just like Boolean Algebra is a convenient algebraic representation of Digital Electronic Circuits, a regular expression is an algebraic representation of an FSA. For example, the regular expression corresponding to the first FSA given above is xx^*y^* . The regular expression for the second FSA is extremely complex! The following simple FSAs and REs illustrate the correspondence:



The rules for forming a regular expression (RE) are as follows:

- [1] The null string (λ) is a RE.
- [2] If the string a is in the input alphabet, then it is a RE.
- [3] if the strings a and b are both REs, then so are the strings built up using the following rules:
 - [3a] CONCATENATION. " ab " (a followed by b).
 - [3b] UNION. " $a \cup b$ " (a or b).
 - [3c] CLOSURE. " a^* " (a repeated zero or more times).

If we have a regular expression, then we can mechanically build an FSA to accept the strings which are generated by the regular expression. Conversely, if we have an FSA, we can mechanically develop a regular expression which will describe the strings which can be parsed by the FSA. For a given FSA or regular expression, there are many others which are equivalent to it. A "most simplified" regular expression or FSA is not always well defined.

Identities for regular expressions appear below. The order of precedence for regular expression operators is: Kleene Star, concatenation; and then union. The Kleene Star binds from right to left; concatenation and union both bind from left to right.

Typical problems in the category will include: translate an FSA to/from a regular expression; simplify an FSA or regular expression (possibly to minimize the number of states or transitions); create an FSA to accept certain types of strings.

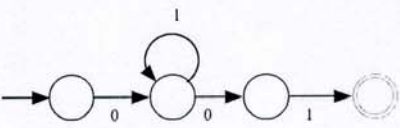
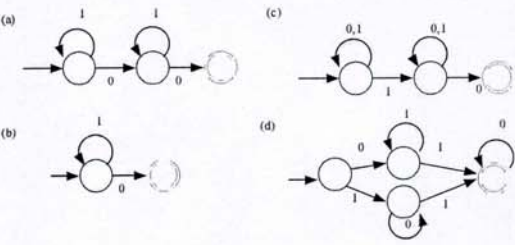
References

FSAs are usually covered in books on compilers. Sedgewick's *Algorithms* covers this topic rather nicely in the context of Pattern Matching and Parsing (Chapters 20 and 21).

Basic Identities

- | | |
|-------------------------|--------------------------------|
| 1. $(a^*)^* = a^*$ | 5. $a(ba)^* = (ab)^*a$ |
| 2. $aa^* = a^*a$ | 6. $(aU^*b)^* = (a^*U^*b^*)^*$ |
| 3. $aa^*U\lambda = a^*$ | 7. $(aU^*b)^* = (a^*b^*)^*$ |
| 4. $a(bU^*c) = abU^*ac$ | 8. $(aU^*b)^* = a^*(ba)^*$ |

Sample Problems

<p>Find a simplified regular expression for the following FSA:</p> 	<p>The expression 01^*01 is read directly from the FSA. It is in its most simplified form.</p>
<p>List all the following FSAs which represent 1^*01^*0.</p> 	<p>Only choice (a) is correct: the other FSAs correspond to the following regular expressions: (b) 1^*0 (c) $(0U1)^*1(0U1)^*0$ (d) $01^*10^*U10^*10^*$ Note that choices (c) and (d) can be rewritten using various identities. For example, an equivalent regular expression corresponding to FSA (d) is $(01^*U10^*)10^*$.</p>
<p>Which, if any, of the following Regular Expressions are equivalent?</p> <p>A. $(aU^*b)(ab^*)(b^*Ua)$ B. $(aab^*Ubab^*)a$ C. $aab^*Ubab^*UaabaUbab^*a$ D. $aab^*Ubab^*Uaab^*aUbab^*a$ E. $a^*U^*b^*$</p>	<p>On first inspection, B can be discarded because it is the only RE whose strings must end in an <i>a</i>. D can also be discarded since it is the only RE that can accept a null string. C and D are not equal by inspection. After expanding A, we must compare it to C and D. It is equal to D, but not to C.</p>

Appendix 2: COS 126 Lecture 15: Finite-State Automata

slide 1 - Regular Expression

Here's a recursive definition of what a regular expressions is:

i.) base cases:

- the empty set
- a single character in the alphabet (In most examples we'll look at, we'll use the binary digits 0 and 1 as our alphabet.)

ii.) if a and b are regular expressions (for our alphabet), then so are:

- (a) parentheses can be used to group regular expressions
- ab concatenation of two regular expressions
- a+b union: if both a and b are regular expressions, then so is 'a or b'
- a* zero or more of regular expression 'a'

Regular expressions define a language (not uniquely.) Given a regular expression, you can ask whether a particular string is in that language. For each of the given examples, I've listed a few of the strings which are in the language and a few that are not.

- $(10)^*$
 - YES: the empty string, 10, 1010, 101010, ...
 - NO: 0, 1, 00, 11, 0101, 0110, and any other strings not like the above pattern!
- $(0+011+101+110)^*$
 - YES: the empty string, 0, 00, 000, 011, 101, 110, 0011, 0101, 0110, 1010, 1100, 011101, 011110, and many more!
 - NO: 1, 00, 01, 10, 11, 111, 1111, 001, 010, 100, and others
- $(01^*01^*01^*)^*$
 - YES: the empty string, 000, 0100, 0010, 0001, 000000, 0000111010111
 - NO: any strings that start with 1, any strings which don't have a multiple-of-3 0's, and others??

slide 2 - Formal Languages

For each of the examples given, how would you write the corresponding regular expression?

1. all bit strings that begin with 0 and end with 1
2. all bit strings whose number of 0's is a multiple of 5
3. all bit strings with more 1's than 0's
4. all bit strings with no consecutive 1's

answers

1. $0(0+1)^*1$
2. $(1^*01^*01^*01^*01^*01^*)^*$
3. you can't do this with a regular expression. How can you tell? the clue is that this language requires counting. Regular expressions can't 'count.'
4. This one is hard - I wouldn't expect you to get it. My answer probably isn't the best one, either: $(0^* + (01)^*)^* + 1(0^* + (01)^*)^*$
(The first part represents strings that start with 0, (and the empty string), while the second represents those strings which start with 1)

slide 3 - Finite State Automata

In precept, I referred to a finite state machine. This is the same thing: an FSA. Each FSA has a fixed number of states (hence the 'finite' part of the name.) There is always a start state. One or more states may be designated as 'accept' states. (The start state may be an accept state.) An FSA accepts a given input

string if and only if you are 'in' an accept state when you finish reading the string. How do you use an FSA to read a string?

You read the input string one character at a time (left to right) and start in the 'start' state of the FSA. That start state will have arrows labeled with different characters in the alphabet. (Let's assume the FSA is "deterministic" - I'll explain what this means, shortly.) In a deterministic FSA, there will be exactly one arrow for each possible input character. So, for the binary alphabet we've been using, there will be two arrows coming out of each state - one labeled '0,' the other labeled '1.' For concreteness, let's assume that the first digit of the string is a 0. We'll read this 0 and move to the state the '0' arrow points to. Now, read the next character in the input string and follow the corresponding arrow to the next state. Continue like this until you have read all the characters in the input string. If you 'end up' in an accept state, we say that the FSA recognizes the input string, or, equivalently, that the input string is part of the language described by the FSA. (This is just like saying that a string is in a language defined by a regular expression.

If you look at the examples in this lecture slide, you can see the order of the states encountered (or 'traveled through') as we read the input string. The states in this example are numbered. By default, state 0 is the start state. Accept states are indicated by double circles.

You should learn to figure out what language a particular FSA represents. The first example recognizes the language of all strings with the pattern 10 (10, 1010, 101010,...) The second example recognizes the language with an odd number of 0's. Trying out a few example input strings and looking for a pattern is a good way to start if you're stuck. Also, always be sure to check whether the empty set is in the language.

slide 4 - An application

This slide demonstrates the computational power of an FSA. On each arrow, the first number represents a character read in the input string. The second number (after the /) is the output generated. The chart on the right shows the output generated when the FSA is applied to the input string: 01010101 - the string 00011101 is generated. Isolated 0's and 1's within the string are eliminated. Notice that this FSA doesn't really have an accept state. It's not recognizing a language; rather, it's performing a computation. This example demonstrates how powerful even a simple FSA can be.

Pay particular attention to the state interpretations chart. If you're asked (on an exam, for example) to draw an FSA representing a given language, it's good to try to formulate the meaning of each state.

In the first example on slide 3, here's the interpretation of each state:

- 0: nothing read so far
- 1: a 1 was just read
- 2: two consecutive 1's read, OR two consecutive 0's read OR a 0 was read initially
- 3: a 0 was read after state 1
 - i.e., a 0 was read after a 1 was read

How do the state interpretations help you understand the FSA as a whole? State 3 is the accept state, because only strings with the pattern $(10)^*$ will end up in this state. State 2 has arrows ("transitions" is the correct terminology) for both 0 and 1 returning to it. This corresponds to the fact that once you've reached state 2, you've read input characters which are not in the language represented by the regular expression $(10)^*$. So, you can never get to the accept state. In state 1, you've read strings of the form $(10)^*1$. For an input string to be in the language, you must read a 0 next, and go to state 3.

Interpreting states can help you find mistakes, too. In this example, you should notice that the empty string is in the language of strings $(10)^*$. So, state 0 should be an accept state, too. The FSA drawn actually represents all strings of the form $(10)(10)^*$. To fix the FSA, just make state 0 an accept state, too. (Remember: you can have more than 1 accept state!)

Question: what are the interpretations of the states in the second example on slide 3?

slide 5 - C program to Simulate FSAs

This C program simulates FSAs that operate on the *binary* alphabet: 0 and 1. The C program opens a file representing the FSA. (The file name is passed as a command line argument. See K&R section 5.10) The format of the file is a list of states, one line per state. For each state (line of the file), the destination state is given for each of the two possible inputs, 0 and 1. So, each line in the file will have two numbers. The first number corresponds to the '0 arrow.' The second number corresponds to the '1 arrow.' The value of the numbers is the number of the state the arrow points to. For the example given, the file will look like this:

```
0 1
2 0
1 2
```

Then, an input string is read from standard input. To run this program, you would type something like:

```
a.out FSA_filename 11101010
```

In each iteration of the while loop, the value of state is updated based on the current state and whether a 0 or a 1 is read. At the end, if the state is 0 (the accept state), Accepted is printed. Otherwise, Rejected is printed. You should try tracing through the code while tracing through the FSA to make sure you understand it.

slide 6 - A language that is not regular

It's good to remember that regular languages can't count. (It's likely to show up on a multiple choice question.) A proof by contradiction is given for the simple case of counting the 0's and 1's for equality.

slide 7 - Pushdown Automata

Think of a Pushdown Automata (PDA) as an FSA + a stack. The stack stores extra information - it is the "memory" added to the FSA. In an FSA, the next state depended only on the current state and whether the input bit was a zero or a one. In a PDA, the next state will depend on the current state and the input bit AND the value of the bit on the top of the stack. Also, in addition to 'moving' to the next state, we can also update the stack by either POPping the top item off the stack or PUSHing the current input bit onto the stack.

Each arrow on the PDA has a 3 part value on it instead of just the input bit (like in an FSA.) The first part is the input bit (exactly the same as the FSA.) The second part is the value of the bit on the top of the stack. Instead of an ACCEPT state, we accept if the stack is empty, when we finish reading the string. If we ever try to POP an empty stack, we *crash*, meaning that the string is not in the language (rejected.)

In the example given, we use the stack to hold the extra 0's or 1's. If, for example, we read 10111..., we'll use the stack to hold the extra 1's until enough 0's come along to POP them off the stack.

slide 8 - Nondeterministic Machines

A defining feature of the FSAs we've looked at so far is that they've all been deterministic. The American Heritage Dictionary defines determinism as "The philosophical doctrine that every event, act, and decision is the inevitable consequence of antecedents that are independent of the human will." In computer science, we use the adjective "deterministic" to describe machines, or automata in which decisions are forced. There is no real choice. In the FSA we've looked at, we never had to choose which state to go to next: we just followed the only arrow (transition) which was designated for the given input bit. In a nondeterministic FSA, instead of one arrow for each input bit, there may be 0 or more arrows. If there are more than one arrow, we need to 'choose' which arrow to follow. How can we interpret nondeterministic FSA?

The basic concept is the same. We read the input string one bit at a time and move to the next state accordingly. If there is no arrow for the input bit, we have 'crashed.' The input string is rejected. If there is

only one arrow for the input bit, follow it to the next state (just like in a deterministic FSA.) If there is more than one, just choose one, follow it and continue with the next input bit. Here's the tricky part: we say that the string is recognized by the FSA (accepted) if there is *any* way to get to an accept state. Let's look at the example given on the lecture slide:

01 - we start in state 0. then, we have the choice of going to state 1 or state 2. If we choose state 2, we then stay in state 2 when we read the '1.' (result=rejected.) However, if we choose state 1, we move to state 3 when we read the 1, so the result is ACCEPT. Since there is a way to end up in the accept state, 01 is recognized by this FSA

011110101 - the sequence of states that leads to the accept state is: 0,1,3,2,2,2,2,0,0,1,3

01000010110 - the sequence of states that leads to the accept state is: 0,1,3,1,1,1,1,3,1,3,2,3

01000 - there is no sequence of states that leads to the accept state. We start in state 0, stay in state 0, and then if we move to state 1, we'll stay there and never leave. If we instead choose to move to state 2, we still won't end up in state 3...

slide 9 - Nondeterminism doesn't help in FSAs

This slide demonstrates a method for converting a nondeterministic FSA to a deterministic one. This is the procedure:

If there are N states, list all the N digit binary numbers. Each binary number represents some combination of the states. 0000 represents none of the states. 0110 represents states 1 and 2 (not 0 and 3.) 1110 represents states 0 and 1 and 2. etc. Each of these binary permutations will represent a state in our deterministic FSA (but we might not use all of them.) For each new state, you need to determine its 'arrows.' That is, you need to determine what the next state will be for a '0' input and a '1' input. To make it easier, we'll use the decimal representation of each of our binary numbers to indicate the number of the corresponding state.

It's not as complicated as it sounds, but it does take a while. Let's start constructing the table in the example:

0001 - in the n-FSA, state 3 had a 0-arrow going to state 1, and a 1-arrow going to state 2. The binary representation of state 1 is 0100. The binary representation of state 2 is 0010. So, the 0-arrow for our state 1 (0001) will go to 4 (0100) and the 1-arrow will go to 2 (0010).

0010 - in the n-FSA, state 2 has a 0-arrow going to states 0 (1000) and 3 (0001), and a 1-arrow going to state 2 (0010). So, in the deterministic FSA, our state 2 (0010) will have a 0-arrow going to state 9 (1001), and a 1-arrow going to state 2. Notice that state 9 represents the combination of states 0 and 3 - (1001).

0011 - this state (3) represents the combination of states 2 and 3. So, we need to look at the transitions (arrows) of both states in the n-FSA. For the 0-arrow, the destinations are states 1 and 0 and 3. For the 1-arrow, the destinations are 2 for both states. In the deterministic FSA, then, for state 3, we'll have a 0-arrow going to state 13 (1101) (wrong on the slide) and the 1-arrow going to state 2 (0010).

If any state doesn't have any arrows for a particular input bit in the n-FSA, we'll make the corresponding arrow go to state 0 in the deterministic FSA.

Notice that each state in the deterministic FSA will have exactly 2 arrows going from it - one for 0 and one for 1. If any states in the deterministic FSA we create don't have any arrows going to it (besides the start state), we can eliminate those states.

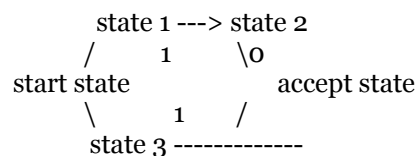
One more thing: we haven't specified which states are the start and accept states. The start state will remain the same. If state 0 was the start state, then state 8 will be the new start state (1000). Any states in the new FSA which include an accept state as part of their combinations of states will be an accept state. In our example, if state 3 (in the n-FSA) is the accept state, then the accept states will be 1 (0001), 3 (0011), 5 (0101), 7 (0111), 9 (1001), 11 (1011), 13 (1101), and 15 (1111).

You can probably see that this would take a long time to do (especially on an exam) even for such a simple FSA. In precept, I can show you how I convert an n-FSA to a deterministic one with a different method. (It would take too long to try to explain without pictures, and I don't have time to create the graphics for it right now...)

slide 10 - FSAs are equivalent to REs

This slide is an overview of a proof demonstrating that FSAs and REs are equivalent. The proof that NFSA and FSAs are equivalent is by construction: given any NFSA, we can construct an equivalent FSA. This slide shows that for any RE, we can construct an NFSA. One thing I haven't mentioned yet, which is important for understanding this slide is that in N-FSA you can have transitions between states which correspond to the null character (neither 0 or 1). You have the choice of following these arrows before reading the next character in the input string. The connecting lines in the rules listed are these null-arrows.

For example, for the simple regular expression: $10 + 1$



OK, let me explain the picture. Coming out of the start state are two arrows (pretend), neither of which has any input bit associated with it. Initially, you can just choose which one to follow. Going the top route, you need to read a 1 followed by a 0 to get to the accept state. Going the bottom direction, you need to read a 1 to get to the accept state. Anything more than this simple example would be too hard to draw.

The 'rules' on this slide can help you construct a FSA from a given regular expression. Basically, a + in a RE corresponds to 'branching' in your FSA. And basically, a * in a RE corresponds to 'loops' in your FSA.

slides 11 & 12 - Nondeterminism does help in PDAs & Turing Machines

The main thing to remember is that non-deterministic FSAs and deterministic FSAs are equivalent. Non-deterministic PDAs are more powerful than deterministic ones; that is, you can recognize a broader class of languages with a nondeterministic PDA. Turing machines can recognize an even more broad class of languages.

Turing machines are the same type of thing that FSAs and PDAs are. You can still think of reading an input string and 'moving' through the states accordingly. Like the PDAs, you have additional information to consider. Instead of a stack, you have a 'tape.' Think of the tape as a long strip of paper onto which you can write. The initial input is written on the tape so that you can read it. Depending on your current state and what you read, you can write a bit, move left or right, and move to a new state.

A Turing Machine has no restrictions on the tape. You can even think of having multiple tapes, if you need them (but you don't.) This type of automata is so powerful that non-determinism doesn't make it any more powerful.