# Course Description

**Weekly Overview**

This week focuses on formal languages and regular expressions. This is important for two reasons. First, it is the very basics of how a computer processes information, and second because it provides background for the material on Turing Machines. Students will be introduced to formal and regular languages and to regular expressions. They will go back and forth between English descriptions of a regular language and the regular expression that describes the language. They will also be introduced to deterministic finite automata.

# Institutional Learning Outcomes

**Main Objectives**

* Define formal language.
* Define regular language and regular expression.
* Convert between English descriptions of languages and their regular expression.
* Build DFAs that represent regular expressions.
* Understand Kleene’s Theorem

# Discipline Specific Outcomes

# Student Readings

None

**Daily Outline**

Day 1: Formal Languages and Regular Expressions

Day 2: Regular Expressions (continued)

Day 3: The DFA

Day 4: DFAs, NFAs, and Kleene’s Theorem

Day 5: Flexible

**Included Resources**

Lecture Notes: Formal Languages and Regular Expressions

Homework: Regular Expressions

Lecture Notes: In-Class Practice with Regular Expressions

In-Class Exercises: Regular Expressions

Handout: The Java Pattern Class

Homework: More Regular Expressions

Lecture Notes: The DFA

Homework: DFAs

Lecture Notes: More DFAs, NFAs, and Kleene’s Theorem

In-Class Exercises: DFAs

Homework: More DFAs

**Lecture Notes: Formal Languages and Regular Expressions**

**Bell Work (5 minutes)**

Complete the following code that prints true if the String s has a vowel, and false otherwise:

//String s = some string

for(int i=0; i<s.length(); i++){

//remember that s.charArt(i) will give the character

//at index i

}

***Answer:***

//String s = some string

boolean vowel = false;

for(int i=0; i<s.length(); i++){

if(s.chartAt(i)==’a’ || s.chartAt(i)==’e’ || s.chartAt(i)==’i’ ||

s.chartAt(i)==’o’ || s.chartAt(i)==’u’){

vowel = true;

i = s.length();

}

}

System.out.println(vowel);

**Main Lecture – Part 1: Formal Languages (10 minutes)**

*N.B. Much of this material is taken from* [*https://introcs.cs.princeton.edu/java/51language/*](https://introcs.cs.princeton.edu/java/51language/)*.*

*Note to Teacher*: There are three reasons we study regular expressions and formal languages. (1) They are necessary prerequisites for DFAs, which are themselves prerequisites for Turing Machines, and the limits of computation (a main goal of the course) come from an understanding of a Turing Machine. (2) Regular expressions are used quite a bit in computer code across a variety of languages. (3) Even though we do not address this specifically, the study of a formal language provides some background for programming languages themselves (compilers, etc.)

Begin with the motivating observation that there are lots of situations that involve String descriptions.

1. The numerical system (just number) is a collection of Strings involving the symbols {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, .}. (This would give us negative and decimals.
2. The “mathematical expression” system would include also the symbols +, /, \*, and lots of others.
3. The English language involves letters and punctuation.
4. Genetic codes are a combination of A, T, C, and G.
5. Protein codes are a combination of A, C, D, E, G, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, and Y.
6. Telephone numbers are a combination of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, –, (, and ).

The individual symbols for each example are called ***symbols***, and the entire universe of symbols for each example is an ***alphabet***. The set of languages over the strings in each example is called a ***formal language***.

There are other formal languages that are much harder to describe or write down:

1. Legal Java programs.
2. Java programs that halt (are not caught in an infinite loop).
3. The set of all works of Shakespeare.
4. The set of all genetic codes that exist in a living human being.

Notice that a formal languages is almost always smaller than “all possibilities.” In other words, the formal language that is “Telephone Numbers” includes the string “(201)-602-1889”, but does *not* include the string “9-2-)8143(“. Likewise, the formal language that is “English words” includes “dog”, but not “gdoasdadas”, and the formal language “English Sentences” includes “This is a sentence.” but not “Run dog my with I.”

**Class-Exercise**

Have the students spend five minutes answering the following questions:

1. Describe the formal language that is of all possible social security numbers.
2. Describe the formal language that is all numbers divisible by 5.
3. Is the empty string a string in the language that is “all palindromes.”

Answers:

1. Three digits followed by a hyphen followed by two digits followed by a hyphen followed by three digits.
2. A non-zero digit followed by a finite sequence of any number of digits followed by a 0 or a 5 … or the number 5. (This is hard.)
3. Yes, but this might depend on how palindrome is defined.

These examples illustrate two important problems in the study of formal languages.

*The Specification Problem:* How do we completely and precisely describe a formal language. Sometimes our English descriptions are sufficient (“all numbers divisible by 5”), but sometimes they are very much not (“all legal sentences”).

*The Recognition Problem:* Given a language *L* and a string *x*, can we decide if *x* is in *L*?

In general, these are *very hard* problems. However, there is a subset of languages for which they are much easier. This subset is called *Regular Languages*. In fact, one could say that the very definition of a regular language is one for which the Specification and Recognition Problems are solvable. In other words, regular languages were developed to be able to describe these “solvable” languages.

**Main Lecture – Part 2: Regular Expressions (15 minutes)**

A regular language of course has a starting alphabet. What defines a regular language is that is can be specified from a *regular expression*.

A regular expression uses an alphabet, plus the symbols |, \*, (, and ).

**Definition:** A regular expression (RE) is a string of symbols that specifies a formal language. It consists of symbols from an alphabet that are “glued” together using the following operations. If *R*  and *S* are strings in the language:

1. Union: *R* | *S*.
2. Concatenation: *RS* (*R* followed by *S*)
3. Closure: *R*\* (*R* repeated zero or more times)
4. Parenthesis: (*R*)

For example, A(B | C)D\* is a regular expression.

As a regular expression, it specifies a language. What language? “A followed by B or C followed by zero or more D’s.”

**Class Exercises:**

Which of the following strings is in the language specified by A(B | C)D\*?

1. A
2. AD
3. ABDD
4. AC
5. ABCDDD
6. B
7. AACDDDDDD
8. ACDDDDDDDDD

Answer: 3, 4, and 8

**Definition:** A language is a *formal language* if and only if it can be specified by a regular expression.

**Class Exercise:**

1. Write a regular expression that specifies the language “all strings that contain only G and H and that end in G.”
2. Write a regular expression that specifies the language “all strings consisting of REPEAT at least once, but repeated as many times as you want.” (For example, the following strings are in the language: REPEAT, REPEATREPEAT, REPEATREPEATREPEATREPEAT, but the empty string is NOT in the language.)

Answers: (G|H)\*G, REPEAT(REPEAT\*)

**Class Exercise:**

1. Describe in English the language specified by the regular expression (1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)(1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9)\*0.
2. Describe in English the language specified by the regular expression (a | b)\*a(a | b)(a | b) (a | b).

Answers: all positive integers divisible by 10, all strings of a’s and b’s where the fourth to last character is an a.

**Main Lecture – Part 3: Generalized Regular Expressions (15 minutes)**

The definition of regular expression is the standard one, and only consists of those basic operations (union, concatenation, closure [repeated], and grouping). However, many computer scientists and mathematicians use some concepts so often, that they have developed additional symbols to save time. These are effectively shorthand notations. It is important to note that they do not make the expression “more powerful”. In other words, any language that can be specified by one of these “expanded” or “generalized” REs can also be specified by a simple (standards) RE. Some of the most common generalizations are:

1. A period (.), or the wild card, representing any character in the alphabet. In other words, for the alphabet of lower case letters: . = (a | b | c | d | … | x | y | z).
2. A list or range of symbols (if there is an understood “order” in the alphabet). As a list, [c–f] is all letters between and including c and f. In other words [c–f] = (c | d | e | f), and [4–9] = (4 | 5 | 6 | 7 | 8 | 9). As a range, [abcde] = (a | b | c | d | e)
3. If the symbol ^ is in braces, then it refers to the symbols *not* in the range. For example, [^4–7] = (0 | 1 | 2 | 3 | 8 | 9). It can also be used on lists: [^agh] is any character that is not a, g, or h.
4. The symbol + means “one or more”. For example, a+ = a(a\*).
5. The symbol ? means “zero or one”. For example, colou?r = (colour | color)
6. The combination of symbols {n} means “exactly n of these things”. For example, a{4} = aaaa
7. {m, n} means “between m and n of these things, inclusive”. For example, a{3,5} = (aaa | aaaa | aaaaa)

Now, because we have added other symbols, there is an increased risk that these symbols themselves may be part of an alphabet. For example, if we want a regular expression to represent the set of all possible numbers with exactly two decimal places we might think it is “([0–9])\*. ([0–9])([0–9])”. However, the period means “any character in the alphabet.” Here, we actually want a period. How do we deal with this? We deal with it the same way that Java code deals with it, by using a “\” is indicate an escape character. The symbol for a period is “\.”.

Therefore, / is used to indicate an escape character, for use when you need a character that “means something else” in the RE: /(, /), /\*, /^, /|, /{, /}, /[, /[, and of course //.

**Class Exercise:**

1. Write a generalized regular expression to specify the language that is “all gmail addresses where the user name is only letters.”
2. What is the language specified by the generalized regular expression \([0-9]{3}\) [0-9]{3}-[0-9]{4}?

Answers: [a-zA-z]+@(gmail)\.(com), all U.S. phone numbers.

**Homework:** Regular Expressions Worksheet

**Homework: Regular Expressions**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Give a regular expression that specifies the following languages. Do not use generalized regular expressions.

1. All strings consisting of only x and y where the first character is an x and the last character is a y.
2. All strings consisting of only a and b but contain the substring abba.
3. All positive integers.
4. All linear equations of the form y = mx + b (m and b positive). This contains things like y = 3x + 6 and y = 134x + 486.
5. All positive integers divisible by 5.
6. All English words (only lowercase) that contain the substring “spb” (such as raspberry).
7. All words that can be typed using only the top row of the keyboard.

Describe the language specified by the following regular expressions.

1. (1)(0 | 1)\*
2. a\*ba\*ba\*
3. (a\*ba\*ba\*)\*
4. a\*b\*c\*
5. (0|1)((0|1)(0|1))\*

Give a generalized regular expression that specifies the following languages.

1. All email addresses where the use name consists only of lowercase letters following by a single digit (smith4) and had one of the three extensions: con, net, org
2. Dollar amounts, e.g. $253.12
3. All positive integers that have exactly seven 5’s in a row.
4. All three letter words using lower case letters.

Describe the languages specified by the following generalized REs.

1. (Mr)(s?)\. [A-Z][a-z]\*
2. –[^0][0-9]\* (the alphabet is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9})
3. \\(\_+)/
4. (\(740\) …-…) | (…-…) (the alphabet is {0, 1, 2, 3, 4, 5, 6, 7, 8, 9})

**Lecture Notes: More Regular Expressions**

**Bell Work (5 minutes)**

Write a regular expression that represents all strings that contain a vowel. The alphabet is lower case letters.

***Answer:***

(a | b | … | z)\*(a | e | i | o | u) (a | b | … | z)\* or [a-z]\*[aeiou][a-z]\*

**In-Class Exercises: Regular Expressions (40 minutes)**

The bulk of this lesson is in-class practice with regular expressions. You may either hand out the “In-Class Exercises: Regular Expressions” worksheet and have students work in groups to solve then, or you may use the worksheet to go through examples with the class as a whole.

**Introducing the Java Pattern Class (10 minutes)**

Pass out the API handout for the Java Pattern class.

Spend this time talking through the class with the students. They will need this on their lab, even though the lab is not for a few days. Do not spend much time right now on the complications with the backslash. You may mention it, but they have enough to think about right now.

**Homework:** More Regular Expressions Worksheet

**In-Class Exercises: Regular Expressions**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Give a regular expression that specifies the following languages.

1. The set of license plates that start with 4 digits and end with two upper case letters.
2. The set of all IP addresses of the form a.b.c.d, where a, c, and d are three digits, but b is two digits.
3. The set of all IP addresses of the form a.b.c.d, where a, b, c, and d can be either one, two, or three digits.
4. The set of all strings that do not contain the substring bbb. The alphabet only {a, b}.
5. The set of all strings that have exactly one occurrence of the substring aaa, e.g. abaaabbba is in the language, but abaaabbaaaba is not, and neither is aaaa. The alphabet is only {a, b}.
6. The set of all strings where the number of a’s is even. The alphabet is only {a, b}.
7. The set of all strings where the number of a’s is odd. The alphabet is only {a, b}.
8. The set of all strings where the number of a’s is a multiple of 3. The alphabet is only {a, b}.
9. The set of all positive integers that do not begin with a 7 and do not contain a 0.
10. (Hard) The set of all quadratic polynomials with positive coefficients where the exponent is given by “^”, e.g. 3x^2 + 2x +178. Use a generalized RE, and remember that the symbols for ^ is \^ because ^ is reserved for a special meaning.
11. All genes. A gene must start with ATG and must end with TAG, TAA, or TGA, and the number of letters must be (1) a multiple of 3, and (2) at least 9. The alphabet is {A, T, G, C}.
12. Sequences of upper case letters that are in “strict” alphabetical order. “Strict” means that consecutive letters cannot be the same. So, for example, “ADXY” is in the language, but “ADDXY” is not.
13. The set of all strings that contain at least three 1s in a row, where the alphabet is only {0, 1, 2}.
14. The set of all strings that have at least two 0s but no consecutive 0s. The alphabet is only {0, 1}.
15. The set of all words that have at least 5 letters but at most 7 letters.
16. The set of all strgins such that the number of a’s plus the number of b’s is even. The alphabet is only {a, b}.
17. The set of all possible files names in Mac OSX. File names can contain any character other than a colon but cannot begin with a period. Assume the alphabet is all characters.
18. The set of strings that contain two or more occurrences of the tetranucleotide GATA. The alphabet is {A, T, G, C}.
19. The set of all Java strings that have a tab character.
20. The set of all calendar dates, in the form 3/5/19 for the year 2019.

**Handout; java.util.regex.Pattern**

The Pattern class is a compiled representation of a regular expression. A regular expression, specified as a String, must first be compiled into an instance of this class. The resulting pattern can then be used to create a Matcher object that can match arbitrary character sequences against the regular expression. All of the state involved in performing a match resides in the Matcher, so many matchers can share the same pattern. A typical invocation sequence is thus

Pattern p = Pattern.[compile](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#compile(java.lang.String))("a\*b");

Matcher m = p.[matcher](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#matcher(java.lang.CharSequence))("aaaaab");

boolean b = m.[matches](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html#matches())();

The following statement is equivalent to the three statements above, though for repeated matches it is less efficient since it does not allow the compiled pattern to be reused:

boolean b = Pattern.matches("a\*b", "aaaaab");

**Summary of Regular Expression Constructs**

The regular expressions in Java are “generalized” regular expressions. They include the formal constructs, but also more complicated ones for convenience. Remember: these more complicated constructs do not make the generalized system more powerful than the formal one.

|  |  |
| --- | --- |
| **Construct (most common)** | **Matches** |
| **Characters** | |
| x | The character x |
| \\ | The backslash character |
| \t | The tab character |
| \n | The newline character |
| **Character Classes** | |
| [abc] | a, b, or c (simple class), so [abc] = (a|b|c) |
| [^abc] | Any character except a, b, or c (negation) |
| [a-d] | a through d, inclusive (range) |
| [a-dm-p] | a through d or m through p (union) |
| [a-z&&[^bc]] | a through z, except for b and c (intersection) equivalent to [ad-z] |
| **Predefined Character Classes** | |
| . | Any character |
| \d | A digit, equivalent to [0-9] |
| \D | A non-digit, equivalent to [^0-9] |
| \s | A whitespace character, equivalent to [ \t\n\x0B\f\r] |
| \S | A non-whitespace character, equivalent to [^\s] |
| \w | A word character, equivalent to [a-zA-z\_0-9] |
| \W | A non-word character, equivalent to [^\w] |
| **Quantifiers** | |
| *X*? | *X*, once or not at all |
| *X*\* | *X*, zero or more times |
| *X*+ | *X*, one or more times |
| *X*{*n*} | *X*, exactly *n* times |
| *X*{*n*,} | *X*, at least *n* times |
| *X*{*n*,*m*} | *X*, at least *n*, but not more than *m* times |
| **Logical Operators** | |
| *XY* | *X* followed by *Y* |
| *X*|*Y* | *X* or *Y* |
| (*X*) | *X* as a group |
| **General Note** | |
| \(, \), \\*, \., \?, \{, \}, \|, \^, etc. | The backslash is necessary for indicating a literal character that is already assigned a special function. In other words, \( is necessary for indicating a parenthesis because ( is reserved for indicating a grouping. |

**A Note on Backslashes and Actually Using Pattern and Strings in Code**

Because the backslash is an escape character for both Strings (in Java code) and for regular expressions, its use can be confusing. Often, multiple backslashes are required.

For example, the regular expression \(.\*\) represents a left parenthesis followed by any number of characters followed by a right parenthesis. Note that in a regular expression, the ( and ) characters are reserved for grouping, which is why the \ is required for both. In fact, the regular expression (.\*) represents simply all strings, equivalent to .\*

The problem occurs when we try to define the code for matching:

Pattern p = Pattern.[compile](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#compile(java.lang.String))("\(.\*\)");

Matcher m = p.[matcher](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#matcher(java.lang.CharSequence))("(a1b2sss)");

boolean b = m.[matches](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html#matches())();

The first line will throw a Java compile error because the \ is interpreted as an escape character *for Strings*. The problem is, in Strings, there is no escape character \(. What we want is an *actual* backslash in the regular expression. To get a backslash in a String, we need \\. Therefore, the first line should be:

Pattern p = Pattern.[compile](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#compile(java.lang.String))("\\(.\*\\)");

In other words, the three-fold interpretation/conversion becomes:

**English Description**

All strings beginning with ( and ending with )

**Regular Expression**

\(.\*\)

**String**

"\\(.\*\\)"

Note that the second line of code does not use escape characters at all. This is because ( is a perfectly valid character for a String, and this String is not a regular expression – it is merely a String to be checked against the regular expression.

As a matter of either academic exercise or obnoxiousness, consider the code for checking a String against a regular expression representing all Strings that has a backslash somewhere in the String. The regular expression symbol for a backslash is \\. Therefore, the regular expression is .\*\\.\*

However, once this is put into a String in Java, the symbol for a backslash is \\. Therefore, each of the two backslashes in the RE .\*\\.\* needs an additional backslash once it is put into a String in Java code. Therefore, the String that represents that regular expression .\*\\.\* is “.\*\\\\.\*”. The code for compiling the RE and checking the input String abc\xyz is:

Pattern p = Pattern.[compile](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#compile(java.lang.String))(".\*\\\\.\*");

Matcher m = p.[matcher](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#matcher(java.lang.CharSequence))("abc\\xyz");

boolean b = m.[matches](https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html#matches())();

The code returns true. Notice that in the first example, the ( and the ) did not need a backslash in the second line of code. This is because parentheses are valid characters in Java Strings. In this example, the backslash needs a backslash. This is because a backslash is *not* a valid character in a Java String, but indicates an escape character is coming. In this case, the escape character itself is a backslash. If you understand this example, then you will have no problem working with REs and the Pattern class in Java.

**Homework: More Regular Expressions**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Write a regular expression that specifies that following languages.

1. The set of all words that do not contain the substring “ing.”
2. The set of all polynomials with positive coefficients where the exponent is given by “^”, e.g. 3x^19 + 2x^7 +178. Use a generalized RE, and remember that the symbols for ^ is \^ because ^ is reserved for a special meaning.
3. (Bonus) All positive integers divisible by 9.

**Lecture Notes: The DFA**

**Bell Work (5 minutes)**

Write a regular expression that represents all integers for which the numbers are in non-increasing order. For example, 9872 is in the language, as is 98665111.

***Answer:***

9\*8\*7\*6\*5\*4\*3\*2\*1\*0\*

**Main Lecture (50 minutes)**

What we are working towards is a conceptual understanding of what an *algorithm* is, which is in effect was a *computer* can do. Regular expressions provide some preliminaries toward this understanding. But the first major step is understanding a ***deterministic finite-state automata***, or a ***DFA***.

We can go into all sorts of formal definitions for this, but the best place to start is that a DFA is any system that has a current state and a way of moving from one state to the next. On top of that, certainly states are “accepted” while others are not.

A simple mechanical example of a DFA would be a combination lock. As certain inputs are fed into the lock, the state of the lock moves. You might think of the states as “no numbers correct”, “first number correct”, “second number correct”, and “third number correct.” Only the third state is “accepted” (in which case the lock opens). And the inputs (numbers) move the lock from one state to the next. For example, if the first “correct” number in the combination is 3, then a 3 moves the lock from “no numbers correct” to “first number correct.” Any other number moves “no number correct” back to itself. For simplicity, let’s suppose that the only possible numbers are 1, 2, and 3, and that the correct combination is 3,3, 1.

A picture of this might be:

No numbers correct

NO

First number correct

NO

Third number correct

YES

Second number correct

NO

3

3

1

1, 2

2, 3

1, 2

Notice that only the correct combination lets us arrive at the “Accept State” of “YES”. We also should specify that the starting state is “No Numbers correct.” Upon entering an incorrect number, we must start the combination over.

Which sequence of numbers will satisfy this DFA? It is not only “331”, but also “11222331”. Actually, any sequence that ends in “331” will satisfy the lock. (Yes, I know that a lock must be “reset” upon entering an incorrect combination. But we are simplifying this a bit to make it easier to understand.

**Class Exercise:**

What RE describes the strings accepted by this DFA?

Answer: (1 | 2 | 3)\*331, or [123]\*331

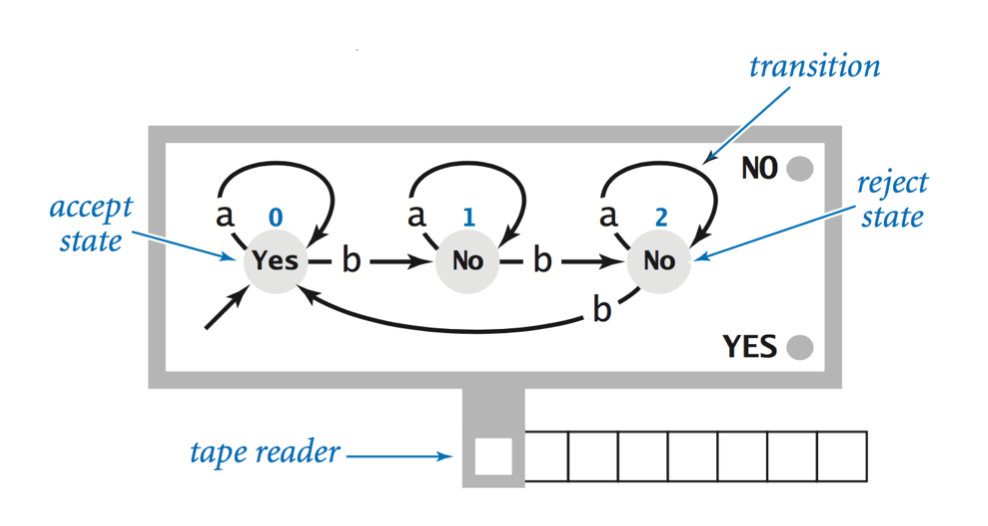
We can use this example to produce a formal definition of a DFA:

**Definition:** A **deterministic finite-state automata**, or **DFA**, is an abstract machine that consists of:

1. A finite number of “states”, each of which is designed as either an *accept* state or a *reject* state.
2. A set of *transitions* that specify how the machine changes state. Each state has one transition for each symbol in the alphabet.
3. A *tape reader* initially positioned at the first symbol of an input string and capable of reading a symbol and moving to the next symbol.

Finally, by definition the DFA begins at state #0. When the input is exhausted, the DFA halts. If it halts on an accept state, the input it said to be “accepted”. Otherwise, the input is said to be “rejected.”

We typically “describe” DFAs using a diagram:



Notice the state numbers are 0, 1, and 3, and the alphabet is apparently {a, b}. One of the states is an accept state (state 0) and the other two are reject states.

The “job” of the DFA is to decide which strings on the input tape are ultimately accepted, and which are rejected.

Take the students through the tape consisting of the following three examples so that they can see the DFA “work”:

1. aaababaaaaaba (accepted)
2. aaababa (rejected)
3. abaababaabaabab (accepted)

**Class Exercise**

1. What strings are accepted by the DFA above?
2. Write an RE to characterize the language that is accepted by the DFA.

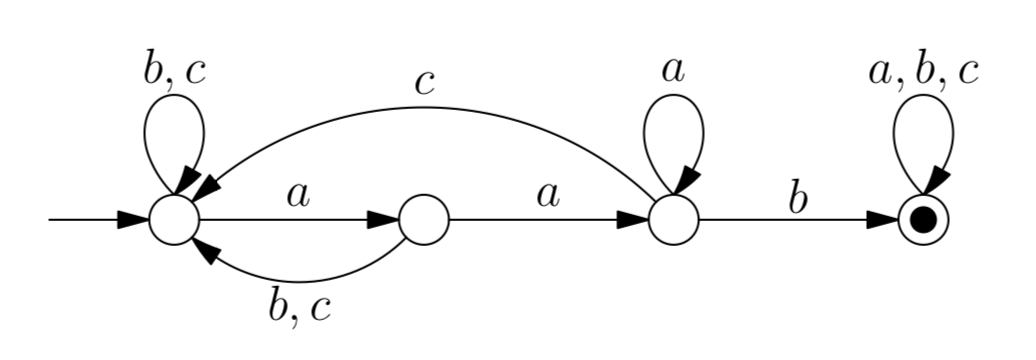
Answers: all strings that have a number of b’s divisible by 3; a\* | (a\*ba\*ba\*ba\*)\*

**Class Exercise**

Draw a DFA that accepts any string that contains the substring aab where the alphabet is {a, b, c}.

Let the class work on this before drawing the solution together.

Answer:



[Teacher note: the notation here is a bit different. The closed circle is the “accept” state, and the left-most arrow indicates where the DFA begins, what we have called state 0. Feel free to draw it the other way or to adopt this simplified diagram/notation.]

**Homework:** DFAs Worksheet

**Homework: DFAs**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

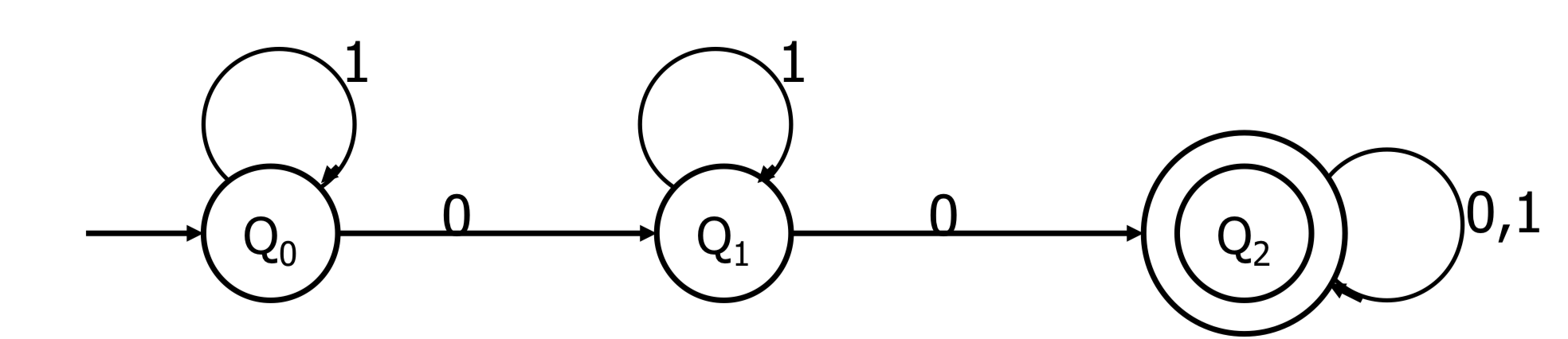
Draw a DFA that accepts the following languages.

1. All positive integers starting with a 1.
2. All positive integers starting with the digits 123.
3. The strings aab and bba only. The alphabet is {a, b}.
4. All strings over {a, b} starting with aa or with bb.
5. All strings over the alphabet {0, 1} that have an odd number of 1s.

**Lecture Notes: More DFAs, NFAs, and Kleene’s Theorem**

**Bell Work (5 minutes)**

What language does the following DFA specify?



[Teacher note: again, forgive the difference in notation. The double circle is the accept state.]

***Answer:***

All strings over {0, 1} that have at least two 0s.

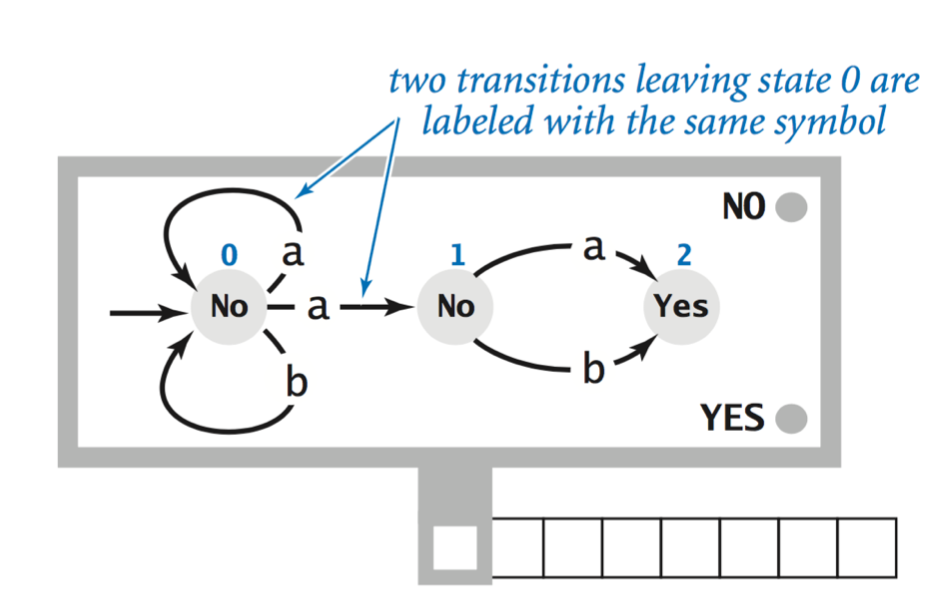
**In-Class Exercises: DFAs (30 minutes)**

The first half of this lesson is in-class practice with DFAs expressions. You may either hand out the “In-Class Exercises: Regular DFAs” worksheet and have students work in groups to solve then, or you may use the worksheet to go through examples with the class as a whole.

*N.B. If the teacher wants to flip the lecture on NFAs and Kleene’s Theorem with the in-class exercises, that will also work.*

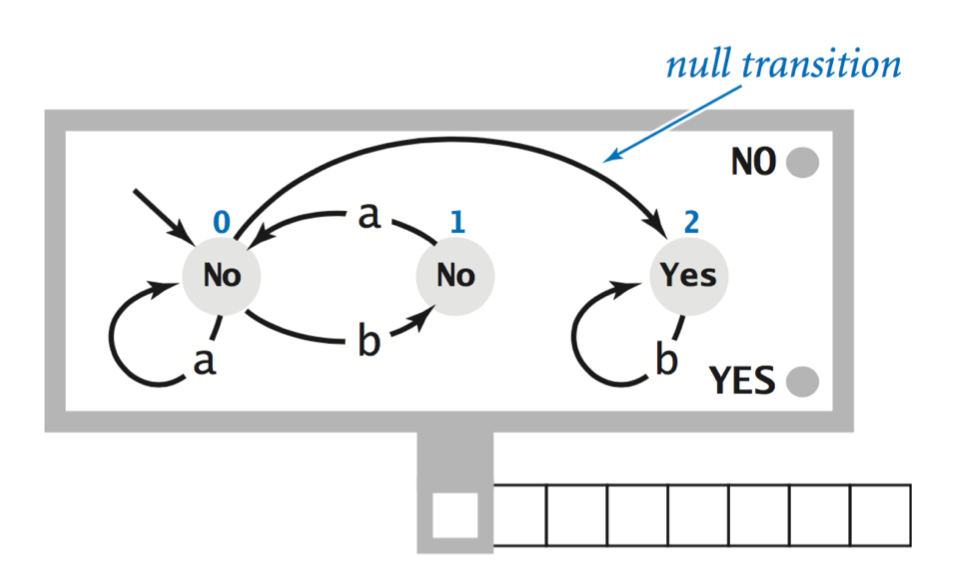
**Main Lecture: NFAs and Kleene’s Theorem (20 minutes)**

There are two ways that we can make a DFA more general. One is that we could have more than one transition coming out of the same state for the same tape-read (input):



Note that coming out of the first state (#0), there are two paths for “a”. As an observer of this machine, *you* can “choose” which a to pick. Note that this machine, while not really “deterministic”, still specifies a language. A string is in the language if there is *any* way that it can land on a Yes state when the tape is exhausted.

Another way is to allow for *null* transitions, or transitions that “jump” from one state to the next without consuming the tape:



Note that when we are in state #0, and the tape reads “b”, we can either consume the b and move to state #1, or jump to state 2 without reading/consuming the b. This also makes the machine non-deterministic.

It seems that these “NFAs” (non-deterministic finite automata) are more powerful than the DFAs. However, there is an amazing and very important theorem, which also shows what we have been suspecting for a while: a connection between REs and DFAs.

**Kleene’s Theorem.** REs, DFAs, and NFAs are equivalent models – they all characterize the same languages: the regular languages.

In other words, none of the three are more powerful than the other. (As a challenge exercise, take the two examples – the one with multiple transitions and the other with a null transition – and try to turn them into a DFA.) This is another way of stating that neither has more “power” than the other. What one can do, the others can do. It also helps computer science answer some questions about what is possible with REs. Sometimes DFAs are easier to work with, so if we know something is impossible with a DFA, then we also know it is impossible with an RE.

Two famous “impossible” tasks are:

1. There is no RE (or DFA, or NFA) that specifies the language where the number of a’s is equal to the number of b’s (even if the alphabet is only {a, b}).
2. There is no RE (or DFA, or NFA) that specifies the language of palindromes, regardless of the alphabet (so long as the alphabet has at least two characters).

(It is a hard thing to prove this, but perhaps some interested students can start thinking about why a DFA cannot exist to do these two things.)

**Homework:** More DFAs Worksheet

**In-Class Exercises: DFAs**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. String over {0, 1, 2} that are at least of length 3.
2. Strings over {a, b, c} where the characters are in alphabetical order.
3. a(a | b)\*b+
4. Social Security Numbers
5. Zip Codes
6. Words containing the substring “ph”, where the alphabet is lowercase letters.
7. Positive integers less than 100.
8. Email addresses with “.com” extensions, domains with only lower case letters, and user names that have only lower case letters followed by a single digit, e.g. smith5@somthing.com.
9. Positive integers divisible by 10.
10. Strings over {a, b} that do *not* end in the same letter.

**Homework: More DFAs**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Draw a DFA that specifies each of the following.

1. The language over {a, b} that has no consecutive b’s.
2. (a | b)\*a\*
3. Any string except bb or bbb.
4. Starts and ends with the same symbol. (The alphabet is {c, d}.)
5. The strings a, aab, and abab only.