Formal Languages

We'll use the *English* language as a running example.

Definitions.

Examples.

- A *string* is a *finite set* of *symbols*, where each *symbol* belongs to an *alphabet* denoted by Σ.

- The *set* of *all strings* that can be constructed from an *alphabet* Σ is Σ^*.

- If x, y are two strings of *lengths* $|x|$ and $|y|$, then:
  - xy or $x \circ y$ is the *concatenation* of x and y, so the length, $|xy| = |x| + |y|$
  - $(x)^R$ is the *reversal* of x
  - the k^{th}-*power* of x is

$$x^k = \begin{cases} \epsilon & \text{if } k = 0 \\ x^{k-1} \circ x, & \text{if } k > 0 \end{cases}$$

  - *equal, substring, prefix, suffix* are defined in the expected ways.
  - Note that the language $\emptyset$ is *not* the same language as ϵ.

Operations on Languages

Suppose that L_E is the *English* language and that L_F is the *French* language over an alphabet Σ.

- **Complementation**: $\overline{L} = \Sigma^* - L$

  $\overline{L}_E$ is the set of all words

- **Union:** $L_1 \cup L_2 = \{x : x \in L_1 \text{ or } x \in L_2\}$

  $L_E \cup L_F$ is the set

- **Intersection:** $L_1 \cap L_2 = \{x : x \in L_1 \text{ and } x \in L_2\}$

  $L_E \cap L_F$ is the set

- **Concatenation:** $L_1 \circ L_2$ is the set of all strings xy such that $x \in L_1$ and $y \in L_2$

  Q: What is an example of a string in $L_E \circ L_F$?

  Q: What if L_E or L_F is $\emptyset$? What is $L_E \circ L_F$?

- **Kleene star:** L^*. Also called the *Kleene Closure* of L and is the *concatenation* of *zero* or *more strings* in L.

  Recursive Definition

  – **Base Case**: $\epsilon \in L$

  – **Induction Step**: If $x \in L^*$ and $y \in L$ then $xy \in L^*$

- **Language Exponentiation** Repeated *concatenation* of a language L.

$$L^k = \begin{cases} \{\epsilon\} & \text{if } k = 0 \\ L^{k-1} \circ L, & \text{if } k > 0 \end{cases}$$

- **Reversal** The language *Rev(L)* is the language that results from *reversing all strings in L*.

Q: How do we *define* the strings that belong to a *language* such as *English, French, Java, arithmetic,* etc.

Example: For the *language of arithmetic*, $\mathcal{LA}$:

Define $\Sigma = \{\mathbb{N}\} \cup \{+, -, =, (,)\}$ then

$$\text{“)((2(+4(= ”} \in \Sigma^*$$

but

$$\text{“)((2(+4(= ”} \notin \mathcal{LA}.$$

Regular Expressions

A *regular expression* over an alphabet Σ consists of

1. *Symbols* in the *alphabet*
2. The symbols $\{+, (,),^*\}$ where $+$ means OR and * means *zero* or *more times*.

Recursive Definition.

Let the set $\mathcal{RE}$ of *ALL regular expressions*, be the smallest set such that:

- **Basis**: $\emptyset, \epsilon, a \in RE, \forall a \in \Sigma$
- **Inductive Step**: if R and S are *regular expressions* $\in \mathcal{RE}$, then so are: $(R+S), (RS), R^*$

Examples: Let $\Sigma = \{0, 1\}$:

Regular Expression	Corresponding Language
$(0+1)^*$	all binary strings.
$((0+1)(0+1)^*)$	all non-empty binary str.
$((0+1)(0+1))^*$	all even lengthed binary strings.
$\epsilon + 0 + 0(0+1)^*0$	all strings not starting or ending with 1.
$11(0+11)^*$	all strings with 1's in pairs and starting with 11.

Relating Regular Expressions to Languages

Let $\mathcal{L}(\mathcal{R})$ represent the *language* constructed by the *regular expression* R.

We define $\mathcal{L}(\mathcal{R})$ *inductively* as follows:

Base Case:

- $\mathcal{L}(\emptyset) = \emptyset$
- $\mathcal{L}(\epsilon) = \{\epsilon\}$
- For any $a \in \Sigma$, $\mathcal{L}(a) = \{a\}$

Induction Step: If R is a *regular expression*, then by definition of R,

- $R = ST$, or
- $R = S + T$, or
- $R = S^*$

where S and T are *regular expressions* and by *induction*, $\mathcal{L}(\mathcal{S})$ and $\mathcal{L}(\mathcal{T})$ have been defined.

We can define the language denoted by R, ie., $\mathcal{L}(\mathcal{R})$ as follows:

- $\mathcal{L}((\mathcal{S} + \mathcal{T}))$ = L(S) union L(T)
- $\mathcal{L}((\mathcal{S}\mathcal{T}))$ = L(S) ∘ L(T)
- $\mathcal{L}(\mathcal{S}^*)$ = (L(S))*

Q: Why is this definition important?

Use these definitions when proving languages are equivalent to each other.

Example

Q: What is a *regular expression* R_A to denote the language of strings consisting of only an *even number* of a's?

e.g., $aa, aaaa, aaaaaaaa$ etc.

L (aa)*

Q: What is a *regular expression* $\mathcal{R}_\mathcal{B}$ for the language of strings consisting of *1* or *more triples* of b's? e.g., $bbb, bbbbbb, bbbbbbbbb$.

(bbb)(bbb)*

Q: What is a regular expression, $\mathcal{R}_{\mathcal{AB}}$, for the language of strings consisting of an *even number* of a's *sandwiched* between 1 or more *triples* of b?

eg., $bbbaabbb$, or $bbbaaaaaabbb$ $R_B R_A R_B$.

Equivalence. We say that two regular expressions R and S are *equivalent* if they *describe* the *same language*.

In other words, if $\mathcal{L}(\mathcal{R}) = \mathcal{L}(\mathcal{S})$ for two regular expressions R and S then $R = S$.

Examples.

- Are R and S *equivalent*?

$$R = a^*(ba^*ba^*)^* \text{ and } S = a^*(ba^*b)^*a^*$$

b baabb   bb

Q: Why?   bbaabb $\in R$

$\notin S$.

- Are $R = (a(a+b)^*)$ and $S = (a(a+b))^*$ *equivalent*?

$\varepsilon \notin R$, $\varepsilon \in S$.

Regular Expression Equivalences

There exist *equivalence axioms* for *regular expressions* that are very similar to those for *predicate/propositional logic*.

Equivalences for Regular Expressions

- Commutativity of union:
- Associativity of union:
- Associativity of concatenation:
- Left distributivity:
- Right distributivity:
- Identity of Union:
- Identity of Concatenation:
- Annihilator for concatenation:
- Idempotence of Kleene star:

Theorem (Substitution) *If two substrings R and R' are equivalent then if R is a substring of S then replacing R by R' constructs a new regular expression equivalent to S.*

Equivalent Regular Expressions

Q: How can we determine whether two *regular expressions* denote the *same language*?

Examples.

Prove that

$$(0110 + 01)(10)^* \equiv 01(10)^*$$

Proof.

$$
\begin{aligned}
(0110 + 01)(10)^* &\equiv \\
&\equiv \\
&\equiv \\
&\equiv \\
&\equiv \\
&\equiv \\
&\equiv
\end{aligned}
$$

Another Example.

Prove that R denotes the *language* L of all strings that contain an *even number* of *0*s.

$$R = 1^*(01^*01^*)^*$$

Equivalently,

$$x \in L \Leftrightarrow x \in \mathcal{L}(R)$$

Proof.

($\Rightarrow$)

- Let $x \in \mathcal{L}(R)$.
- Then $x \in$
- Let $x = y(zw)^*$ then
- Therefore, y has
- Therefore, w has
- Therefore, z has
- So, $x = y(zw)^*$ has

($\Leftarrow$)

- Suppose that x is an *arbitrary* string in L.

- $\Rightarrow$ x has an *even* number of *0*s. Denote by $2k$ for some $k \in \mathbb{N}$.

- How can we rewrite x consisting of *0*s and *1*s?

- Let $x = y_0, y_1, y_2, \ldots, y_k$, so

- So $x = y_0 y_1 \ldots y_k \in \mathcal{L}(1^*)(\mathcal{L}(01^*01^*))^* = \mathcal{L}(1(01^*01^*)^*)$.

Q: Can *every* possible type of string be *represented* by a *regular expression*?

To answer this, we turn to *Finite State Machines*.

String Matching and Finite State Machines

- Given *source code* (say in Java)
- Find the *comments* – may need to remove comments for *software transformations*

```
public class QuickSort {
    private static long comparisons = 0;
    private static long exchanges   = 0;

   /***********************************************************************
    *  Quicksort code from Sedgewick 7.1, 7.2.
    ***********************************************************************/
    public static void quicksort(double[] a) {
        shuffle(a);                             // to guard against worst-case
        quicksort(a, 0, a.length - 1);
    }
    public static void quicksort(double[] a, int left, int right) {
        if (right <= left) return;
        int i = partition(a, left, right);
        quicksort(a, left, i-1);
        quicksort(a, i+1, right);
    }

    private static int partition(double[] a, int left, int right) {
        int i = left - 1;
        int j = right;
        while (true) {
            while (less(a[++i], a[right]))      // find item on left to swap
                ;                               // a[right] acts as sentinel
            while (less(a[right], a[--j]))      // find item on right to swap
                if (j == left) break;           // don't go out-of-bounds
            if (i >= j) break;                  // check if pointers cross
            exch(a, i, j);                      // swap two elements into place
        }
        exch(a, i, right);                      // swap with partition element
        return i;
    }
```

Q. What *patterns* are we looking for?

Q. What do we know if we see a `/` followed by a

`*`

`/`

`text`

Q. What do we know if we see `/*` followed by a

`*`

`/`

`text`

Let's represent these ideas with a *diagram*.

Deterministic Finite State Automata (DFSA or DFA)

A *DFA* consists of:

- Q.
- Σ.
- $s \in Q$.
- $F \subseteq Q$.
- δ.

Comment Example.

- Q =
- Σ =
- $s =$
- $F =$
- δ

Example cont...

δ(state, input)	/	*	text	\nl
start				
/				
//				
/*				
accept				

Q: What if we want to know which state the input "**//" ends at if we *begin* at *start*?

Two Options.

1. Compute:

2. Define δ^*.

Formal definition of $\delta^*(q, x)$ (reading left to right):

$$\delta^*(q,x) = \begin{cases} q & if \quad x = \epsilon \\ \delta(\delta^*(q,z),a) & if \quad x = za, a \in \Sigma, z \in \Sigma^* \end{cases}$$

Regular Expressions and DFA

- The set of strings *accepted* by an *automaton* defines a *language*.
- For *automaton* M the language M accepts is $\mathcal{L}(M)$.
- Given *regular expression* R, find M such that

$$\mathcal{L}(R) = \mathcal{L}(M).$$

Examples.

Let *regular expression* $R_1 = (1 + 00)^*$.

Q. Which strings belong to $\mathcal{L}(R_1)$?

Q: What is a *DFA* M_1 such that $\mathcal{L}(M_1) = \mathcal{L}(R_1)$?

DFSA Conventions

- Strings ending at a *final state* are *accepted* (if we want to accept/reject).
- Drop *dead* states.
- Group *elements* that go from and to the *same states*.

Examples cont.

Let *regular expression* $R_2 = 1(1 + (01))^*$.

Q. Which strings belong to $\mathcal{L}(R_2)$?

Q: What is a *DFA* M_2 such that $\mathcal{L}(M_2) = \mathcal{L}(R_2)$?

$\delta:$ $\delta(q_0, 0) =$ $\delta(q_0, 1) =$ $\delta(q_1, 0) =$

$\delta(q_1, 1) =$ $\delta(q_2, 0) =$ $\delta(q_2, 1) =$

$\delta(d_1, 0 \text{ } or \text{ } 1) =$

Q: How do we know that our *machine* M is *correct*?

We can show this by proving that $\delta^*(q_0, x)$ *only accepts those strings in* $\mathcal{L}(R_2)$.

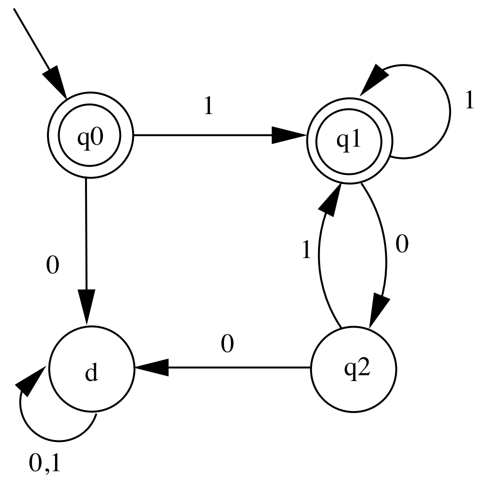
Q: What might be a *good* way to do this?

Proving a DFA is Correct

Q: What should we do *induction* on?

Q: What should our $S(x)$ include?

Proof that $\mathcal{L}(M_2) = \mathcal{L}(R_2)$:



$\mathcal{L}(R_2) = \{x \in \{0, 1\} \mid$ every 0 is sandwiched between 1s $\}$

$$S(x): \quad \delta^*(q_0, x) = \begin{cases} q_0 \\ q_1 \\ q_2 \\ d_1 \end{cases}$$

RTP $S(x)$ for all $x \in \Sigma^*$.

Base Case. $x = \epsilon$:

IS. Assume that $S(y)$ holds for $y \in \Sigma^*$ and consider $x = ya$ where $a \in \Sigma$.

Two cases: *Case 1.* $a = 1$. and *Case 2:* $a = 0$.

Case 1. $a = 1$. Then $\delta^*(q_0, y1)$ = by definition of δ^*.

$$\delta^*(q_0, y1) = \begin{cases} \delta(q_0, 1) & \text{if} \\ \delta(q_1, 1) & \text{if} \\ \delta(q_2, 1) & \text{if} \\ \delta(d_1, 1) & \text{if} \end{cases}$$

Q. Why can we write this?

We can rewrite in terms of x to get:

$$\delta^*(q_0, y1) = \begin{cases} \delta(q_0, 1) \\ \delta(q_1, 1) \\ \\ \delta(q_2, 1) \\ \\ \delta(d_1, 1) \end{cases}$$

from the *definition* of δ (or the diagram):

$$\delta^*(q_0, y1) = \begin{cases} \end{cases}$$

Case 2: $a = 0$ Then $\delta^*(q_0, y0)$ = by definition of δ^*.

$$= \begin{cases} \delta(q_0, 0) & \text{if } y \text{ is empty, } y \in \mathcal{L}(R_2) \\ \delta(q_1, 0) & \text{if all 0s in } y \text{ sandwiched by 1s} \\ \delta(q_2, 0) & \text{if } y \text{ ends in 0} \\ \delta(d_1, 0) & \text{if } y \text{ has a 0 not preceded by a 1} \end{cases}$$

because x ends with a 0,

$$= \begin{cases} \delta(q_0, 0) & \text{if } x \text{ has a 0 that is not preceded by a 1, } x \notin \mathcal{L}(R_2) \\ \delta(q_1, 0) & \text{if } x \text{ ends in a 0, so } x \notin \mathcal{L}(R_2) \\ \delta(q_2, 0) & \text{if } x \text{ ends in 00, so } x \notin \mathcal{L}(R_2) \\ \delta(d_1, 0) & \text{if } x \text{ has a 0 not preceded by a 1, , so } x \notin \mathcal{L}(R_2) \end{cases}$$

from the state diagram and definition of δ

$$= \begin{cases} d_1 & \text{if } x \text{ has a 0 that is not preceded by a 1, } x \notin \mathcal{L}(R_2) \\ q_2 & \text{if } x \text{ ends in a 0 but all other 0s sandwiched by 1s} \\ d_1 & \text{if } x \text{ has a 0 not preceded by a 1, } x \notin \mathcal{L}(R_2) \\ d_1 & \text{if } x \text{ has a 0 not preceded by a 1, , } x \notin \mathcal{L}(R_2) \end{cases}$$

Therefore, our *DFA* satisfies the *invariant* $S(x)$.

Non-Deterministic Finite State Automata (NFA or NFSA)

Q: What does *deterministic* mean?

NFSA. A *non-deterministic finite state automata* (*NFSA*) extends *DFSA* by *allowing* choice at each state.

Differences between *DFSA* and *NFSA*:

- **NFSA.** Given a state q_i and an input x there can be *more* than *one* possible *transition*, i.e,

$$\delta^*(q, x) = \{set\ of\ q_i\}$$

- **NFSA.** Given state q_i, we can have an ϵ *transition*.

$$\delta^*(q_i, \epsilon) = q_j$$

  This means we can spontaneously *jump* from q_i to q_j.

Q: How do we know if a *string* is *accepted* by an *NFSA*?

Examples of NFSA

Consider the strings that are *represented* by the *regular expression*: $(\mathbf{010} + \mathbf{01})^*$

NFSA:

DFSA:

Formally. An *NFSA*, is a machine $M = (Q, \Sigma, \delta, s, F)$ where

- each of Q, Σ, s, F are as for a *DFSA*.
- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ ($\mathcal{P}(Q)$ is a *set* of states).
- $\delta^* : Q \times \Sigma^* \to \mathcal{P}(Q)$.

Limitations of DFSA and NFSA

Q: Can *every* set of strings be *recognized* by a *DFSA*? an *NFSA*?

Q: How much more *powerful* is an *NFSA* over a *DFSA*?

Detailed answers.

- Only strings representable by *regular expressions* can be *recognized* by an *NFSA* or *DFSA*.
- There exists an *algorithm* to *convert* between *deterministic* and *non-deterministic* machines.

Theorem. *If L is a regular language then the following are all equivalent:*

1. L is denoted by a *regular expression*
2. L is accepted by a *deterministic FSA*
3. L is accepted by a *non-deterministic FSA*

(See the course text for the proof.)

Closure Properties of FSA-accepted Languages

Q: What do we mean by *closure*?

Theorem Every *regular* language L is *closed* under *complementation, union, intersection, concatenation* and the *Kleene star* operation.

Q: *What does this mean?*

Proof of $L \cup L'$.

- Let M be a *NFSA* that accepts L.
- Let M' be a *NFSA* that accepts L'.

Q: How can we *construct* $M_{\cup}$ that will accept *either* language?

Proof of L^*.

Given M accepting L, how can we build a new *NFSA* to *accept* L^*?

Regular Languages

Q: How can we prove that a *language* L is *regular*?

Q: How can we prove that a $\mathcal{L}$ is *not* regular?

- Any *FSA* has a *finite* number of *states*, say n.
- Therefore if L is *infinite*, then L has strings with $> n$ symbols.
- **Q:** *What does this imply about at least one state of the FSA?*
- Repeating this *cycle* an *arbitrary number* of times must yield another *string* in L.
- **Q:** *What does this mean?*
- **Q:** *How does this help us?*