Convert this DFA $M$ to equivalent r.e.

Language: even number of $a$'s or an odd number of $b$'s.

$L(M) = L(M1) \cup L(M2) \cup L(M3)$

$L(M2) = L(M1)^* L(M4)$

$L(M4) = aL(M5) \cup bL(M6)$
Algorithm

**Given a DFA** \( M = (Q, A, \phi, i, T) \).

**Construct a regular expression for** \( L(M) \).

Recursive: express solution in terms of solutions to simpler instances of this problem.

Base case: \( M \) has no labelled transitions to an accepting state.
- If \( i \in T \) then \( L(M) = \{ \epsilon \} \)
- Else \( L(M) = \emptyset \)

The algorithm (continued)

Recursive case: there’s a transition to accepting state
- If \( T = \{ q_1, \ldots, q_k \} \) where \( k > 1 \)
- Then \( L(M) = \bigcup_{j=1}^{k} L(M_j) \)
  - where \( M_j = (Q, A, \phi, i, \{ q_j \}) \)

Evaluations of regular expressions for \( L(M_j) \) are done recursively.
The algorithm (continued)

Else /* one accepting state */
Let \( T = \{ t \} \) (i.e. \( M = (Q, A, \phi, i, \{ t \}) \))
If \( t = i \) Then \( L(M) = L(M')^* \)
where \( M' = (Q \cup \{ t' \}, A, \phi', i, \{ t' \}) \)
and \( \phi'(q, a) = t' \) whenever \( \phi(q, a) = i \)
otherwise \( \phi'(q, a) = \phi(q, a) \).

Regular expression for \( L(M') \) is found recursively.

The algorithm (continued)

Else /* \( T = \{ t \}, t \neq i */
If \( M \) has labelled transitions to \( i \) Then \( L(M) = L(M')^* L(M'') \)
where \( M' = (Q, A, \phi, i, \{ i \}) \), \( M'' \) is like \( M \) but without the labelled transitions to \( i \)
Else /* no labelled transitions to \( i */
\( L(M) = \bigcup_{a \in A} a L(M_a) \)
where \( M_a \) is like \( M \) but without transition from \( i \) labelled by \( a \) (don’t include \( M_a \) if no such transition is in \( M \))
Initial state of \( M_a \) is \( \phi(i, a) \).

Regular expressions for \( L(M') \), \( L(M'') \), \( L(M_a) \) etc are found recursively.

Justifying the DFA to r.e. algorithm

The algorithm expresses the language accepted by a given DFA \( M \) in terms of languages accepted by other DFAs - recursive
Want to verify that:

1. \( L(M) \) is the same as the language accepted by the combinations of DFAs constructed in the recursive calls
2. The algorithm terminates.

Regarding (1): We saw that in all the cases that arose the languages are the same
Regarding (2): Define "simpler" such that the algorithm decomposes a task into "simpler" tasks, until we reach base case (the "simplest" tasks where it’s obvious what to do)

Proving that the algorithm terminates

Say what we mean by “simpler”
Verify that the algorithm always expresses a language in terms of languages accepted by "simpler" machines.
Prove that any sequence of machines \( M_1, M_2, M_3, \ldots \) where \( M_i \) is simpler than \( M_{i-1} \), is finite.
That will prove that the depth of recursion is finite.
Comparing finite automata $M_1$ and $M_2$

Define the number of transitions of $M$ to be the number of pairs $(q, a)$ for which $\phi(q, a)$ is defined.

1. If $M_1$ and $M_2$ have differing numbers of transitions, the one with the smaller number is simpler.
2. Failing that, the one with the smaller number of accepting states is simpler.
3. If one (but not both) of $M_1$, $M_2$ has no transitions to initial state, that one is simpler.
4. If one (but not both) of $M_1$, $M_2$ has initial state accepting, the other one is simpler.

Check that in each case given in the algorithm, the machines constructed for the recursive calls are simpler.

Kleene’s Theorem (overview)

1. General method for converting from reg expr to DFA
2. General method for converting from DFA to reg expr (note: current version is different from these slides)

Method 1 also needs general method for converting NFA to DFA (given earlier). (Due to the fact that the constructions for closure and concatenation of languages accepted by DFAs may create a NFA.)

Note that an expression/finite automaton may (typically) be much larger than smallest possible.
"short cuts" are possible if you don’t follow the general method.
(The point of the general method is that it always works)
e.g. if a DFA has a “bottleneck” (a transition that accepting paths
must use) that may suggest its language is the concatenation of 2
languages accepted by machines each side of that transition. (And,
maybe the general algorithm would start by expressing the DFA as
the union of several DFAs that are only slightly smaller...)

Usually easy to describe a programming language’s tokens using
regular expressions (often a natural way of expressing a description
symbolically). But it’s obvious how to implement a DFA (giving
algorithm to scan tokens)

Example
A token is \( <, <=, =, >=, =, <>, (, \), +, −, *, /, :=, ;, \) or an
identifier, keyword, constant or literal.

identifier = \( (A)\langle A \rangle \cup \langle D \rangle \)^*
keyword = \{ for, to, if, ... \}
constant = \( \langle D \rangle \)^*
literal = "\( (\langle D \rangle \cup \langle A \rangle \cup \{<, =, \rangle \})\)"
(any string of symbols in single quotes)
comment = \(/ \ast (\langle A \rangle \cup \langle D \rangle \cup \{<, =, \})\)^* /

observations, exercise

Recall: any formal language is a well-defined set of words over
some given alphabet.
A regular language is one that is given by some regular expression
or accepted by some DFA (or NFA)
A regular language will have many different but equivalent DFAs or
reg expres that represent it.

Which of the following regular expressions over alphabet \{ a, b \} are
equivalent:
\( (a^*)b, \ b \cup ba(a^*)b, \ ba(a^*), \ \{b, \epsilon\}a(a^*), \ \{ba, a\}a^* \)
Let $R$, $S$ and $T$ denote regular expressions. We can note some general rules governing equivalence of regular expressions such as

$$R \cup S = S \cup R$$
$$R \cup (S \cup T) = (R \cup S) \cup T$$
$$R(ST) = (RS)T$$
$$R(S \cup T) = (RS) \cup RT$$

### Example proof

**Prove that** $(R^*)^* = R^*$.  

Suppose $w \in (R^*)^*$.  

Then $w = w_1w_2 \ldots w_n$ for $w_i \in R^*$.  

$w_i \in R^*$ means that $w_i = w_{i,1}w_{i,2} \ldots w_{i,n(i)}$ for $w_{i,j} \in R$.  

Hence $w$ is a concatenation of words that belong to $R$, so $w \in R^*$.

Then, prove that if $w \in R^*$ then $w \in (R^*)^*$.  

If $w \in R^*$ then it follows immediately that $w \in (R^*)^*$, since the closure of a language contains all words in that language and possibly more.