A **non-deterministic finite-state automaton** (NFA) consists of:

- An alphabet Σ of *input symbols*
- A set of *states*, of which we distinguish:
    - a unique *start state*, usually numbered ``0"
    - a set of *final states*
- A transition relation which, for any given state and symbol gives the (possibly empty) set of next states

We can represent a NFA diagrammatically using a (labelled, directed) graph, where states are represented by nodes (circles) and transitions are represented by edges (arrows).

The purpose of an NFA is to model the process of reading in characters until we have formed one of the words that we are looking for.

NFA operates the following way:

- We begin in the start state (usually labelled 0) and read the first character on the input.
- Each time we are in a state, reading a character from the input, we examine the outgoing transitions for this state, and look for one labelled with the current character. We then use this to move to a new state.
    - There may be more than one possible transition, in which case we choose one at random.
    - If at any stage there is an output transition labelled with the empty string, ε, we may take it without consuming any input.
- We keep going like this until we have no more input, or until we have reached one of the final states.
- If we are in a final state, with no input left, then we have succeeded in recognising a pattern.
- Otherwise we must *backtrack* to the last state in which we had to choose between two or more transitions, and try selecting a different one.

Basically, in order to match a pattern, we are trying to find a sequence of transitions that will take us from the start state to one of the finish states, consuming all of the input.

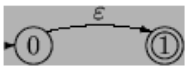
The key concept here is that: *every NFA corresponds to a regular expression*

Moreover, it is fairly easy to convert a regular expression to a corresponding NFA. To see how NFAs correspond to regular expressions, let us describe a conversion algorithm:

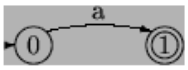
Converting a regular expression to a NFA - Thompson's Algorithm

We will use the rules which defined a regular expression as a basis for the construction:

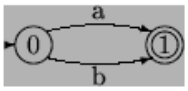
1. The NFA representing the empty string is:



2. If the regular expression is just a character, eg. `a`, then the corresponding NFA is :

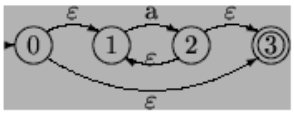


3. The union operator is represented by a choice of transitions from a node; thus `a|b` can be represented as:



4. Concatenation simply involves connecting one NFA to the other; eg. `ab` is:

5. The Kleene closure must allow for taking zero or more instances of the letter from the input; thus `a*` looks like:



James Power 2002-11-29

Constructing a DFA from an NFA (``Subset Construction'')

We merge together NFA states by looking at them from the point of view of the input characters:

- From the point of view of the input, any two states that are connected by an ε-transition may as well be the same, since we can move from one to the other without consuming any character. Thus states which are connected by an ε-transition will be represented by the same states in the DFA.
- If it is possible to have multiple transitions based on the same symbol, then we can regard a transition on a symbol as moving from a state to a set of states (ie. the union of all those states reachable by a transition on the current symbol). Thus these states will be combined into a single DFA state.

To perform this operation, let us define two functions:

- The **ε-closure** function takes a state and returns the set of states reachable from it based on (one or more) ε-transitions. Note that this will always include the state tself. We should be able to get from a state to any state in its ε-closure without consuming any input.
- The function **move** takes a state and a character, and returns the set of states reachable by <u>one</u> transition on this character.

We can generalise both these functions to apply to sets of states by taking the union of the application to individual states.

Eg. If A, B and C are states, move({A,B,C},`a') = move(A,`a') $\cup$move(B,`a') $\cup$move(C,`a').

The Subset Construction Algorithm

1. Create the start state of the DFA by taking the ε-closure of the start state of the NFA.
2. Perform the following for the new DFA state:
   For each possible input symbol:
   1. Apply move to the newly-created state and the input symbol; this will return a set of states.
   2. Apply the ε-closure to this set of states, possibly resulting in a new set.

   This set of NFA states will be a single state in the DFA.

3. Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 does not yield any new states.
4. The finish states of the DFA are those which contain any of the finish states of the NFA.

DFA minimisation

Since we are interested in translating a DFA into a program, we will want to ensure that this program is as efficient as possible. In automata terms, one aspect of this will be to ensure that the constructed DFA has as few states as possible. This is achieved by an algorithm known as **DFA minimisation**.

We take the ``optimistic'' approach: we start by assuming that all states are actually the same, and only distinguish those which we can prove are different. As for subset construction, the states of the minimised DFA will be sets of states from the original DFA; these will be states that we couldn't tell apart!

Two states are different if:

- one is a final state and the other isn't, or
- the transition function maps them to different states, based on the same input character

We base our algorithm on partitioning the states using this criterion.

1. Initially start with two sets of states: the final, and the non-final states.
2. For each state-set created by the previous iteration, examine the transitions for each state and each input symbol. If they go to a different state-set for any two states, then these should be put into different state-sets for the next iteration.
3. We are finished when an iteration creates no new state-sets.