Building Finite Automata From Regular Expressions

We make an FA from a regular expression in two steps:

- Transform the regular expression into an NFA.
- Transform the NFA into a deterministic FA.

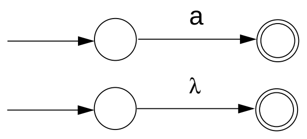
The first step is easy.

Regular expressions are all built out of the *atomic* regular expressions a (where a is a character in Σ) and λ by using the three operations

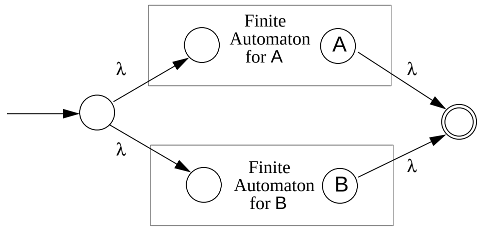
A B and A | B and A^*.

Other operations (like A^+) are just abbreviations for combinations of these.

NFAs for a and λ are trivial:



Suppose we have NFAs for A and B and want one for A | B. We construct the NFA shown below:



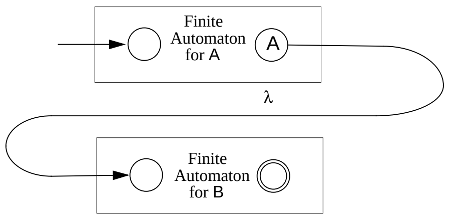
The states labeled A and B were the accepting states of the automata for A and B; we create a new accepting state for the combined automaton.

A path through the top automaton accepts strings in **A**, and a path through the bottom automation accepts strings in **B**, so the whole automaton matches **A | B**.

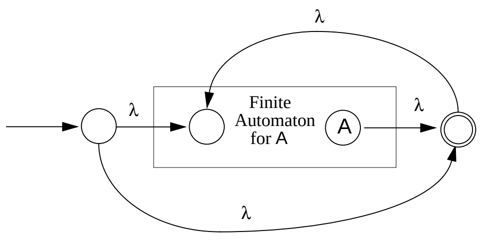
The construction for A B is even easier. The accepting state of the combined automaton is the same state that was the accepting state of B. We must follow a path through **A**'s automaton, then through **B**'s automaton, so overall **A B** is matched.

We could also just merge the accepting state of A with the initial state of B. We chose not to

only because the picture would be more difficult to draw.



Finally, let’s look at the NFA for A^*. The start state reaches an accepting state via λ, so λ is accepted. Alternatively, we can follow a path through the FA for A one or more times, so zero or more strings that belong to A are matched.



Creating Deterministic Automata

The transformation from an NFA N to an equivalent DFA D works by what is sometimes called the *subset construction.*

Each state of D corresponds to a set of states of N.

The idea is that D will be in state {x, y, z} after reading a given input string if and only if N could be in *any* one of the states *x*, *y*, or *z*, depending on the transitions it chooses. Thus D keeps track of *all* the possible routes N might take and runs them simultaneously.

Because N is a *finite* automaton, it has only a finite number of states. The number of subsets of N’s states is also finite, which makes

tracking various sets of states feasible.

An accepting state of D will be any set containing an accepting state of N, reflecting the convention that N accepts if there is *any* way it could get to its accepting state by choosing the “right” transitions.

The start state of D is the set of all states that N could be in without reading any input characters—that
is, the set of states reachable from the start state of N following only λ transitions. Algorithm **`close`** computes those states that can be reached following only λ transitions.

Once the start state of D is built, we begin to create successor states:

We take each state S of D, and each character c, and compute S’s successor under c.

S is identified with some set of N’s states, $\{n_1, n_2, ...\}$.

We find all the possible successor states to $\{n_1, n_2, ...\}$ under c, obtaining a set $\{m_1, m_2, ...\}$.

Finally, we compute $T = \text{CLOSE}(\{ m_1, m_2, ...\})$.

T becomes a state in D, and a transition from S to T labeled with c is added to D.

We continue adding states and transitions to D until all possible successors to existing states are added.

Because each state corresponds to a finite subset of N’s states, the process of adding new states to D must eventually terminate.

Here is the algorithm for λ-closure, called **close**. It starts with a set of NFA states, S, and adds to S all states reachable from S using only λ transitions.

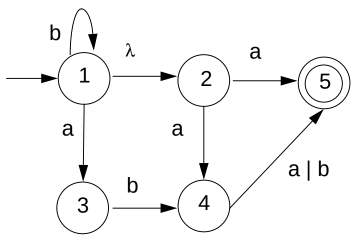
```
void close(NFASet S) {
                        λ
 while (x in S and x → y
       and y notin S) {
          S = S ∪ {y}
}}
```

Using **close**, we can define the construction of a DFA, D, from an NFA, N:

```
DFA  MakeDeterministic(NFA N) {
 DFA  D ; NFASet  T
 D.StartState = { N.StartState }
 close(D.StartState)
 D.States = { D.StartState }
 while (states or transitions can be
        added to D) {
   Choose any state S in D.States
     and any character c in Alphabet
   T = {y in N.States such that
             c
          x → y for some x in S}
   close(T);
   if (T notin D.States) {
        D.States = D.States ∪ {T}
        D.Transitions =
           D.Transitions ∪
                               c
             {the transition S → T}
 } }
 D.AcceptingStates =
  { S in D.States such that an
      accepting state of N in S}
}
```

Example

To see how the subset construction operates, consider the following NFA:



We start with state 1, the start state of N, and add state 2 its λ-successor.

D's start state is {1,2}.

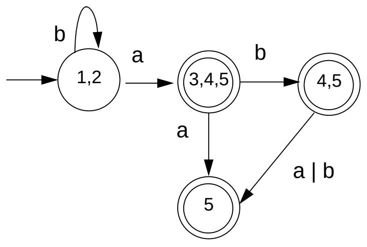
Under a, {1,2}'s successor is {3,4,5}.

State 1 has itself as a successor under b. When state 1's λ-successor, 2, is included, {1,2}'s successor is {1,2}. {3,4,5}'s successors under a and b are {5} and {4,5}.

{4,5}'s successor under b is {5}.

Accepting states of D are those state sets that contain N's accepting state which is 5.

The resulting DFA is:



It is not too difficult to establish that the DFA constructed by **`MakeDeterministic`** is equivalent to the original NFA.

The idea is that each path to an accepting state in the original NFA has a corresponding path in the DFA. Similarly, all paths through the constructed DFA correspond to paths in the original NFA.

What is less obvious is the fact that the DFA that is built can sometimes be *much larger* than the original NFA. States of the DFA are identified with *sets* of NFA states.

If the NFA has n states, there are 2^n distinct sets of NFA states, and hence the DFA may have as many as 2^n states. Certain NFAs actually

exhibit this exponential blowup in size when made deterministic.

Fortunately, the NFAs built from the kind of regular expressions used to specify programming language tokens do not exhibit this problem when they are made deterministic.

As a rule, DFAs used for scanning are simple and compact.

If creating a DFA is impractical (because of size or speed-of-generation concerns), we can scan using an NFA. Each possible path through an NFA is tracked, and reachable accepting states are identified. Scanning is slower using this approach, so it is used only when construction of a DFA is not practical.

Optimizing Finite Automata

We can improve the DFA created by **`MakeDeterministic`.**

Sometimes a DFA will have more states than necessary. For every DFA there is a unique *smallest* equivalent DFA (fewest states possible).

Some DFA's contain *unreachable states* that cannot be reached from the start state.

Other DFA's may contain *dead states* that cannot reach any accepting state.

It is clear that neither unreachable states nor dead states can participate in scanning any valid token. We therefore eliminate all such states as part of our optimization process.

We optimize a DFA by *merging together* states we know to be equivalent.

For example, two accepting states that have no transitions at all out of them are equivalent.

Why? Because they behave exactly the same way—they accept the string read so far, but will accept no additional characters.

If two states, s_1 and s_2, are equivalent, then all transitions to s_2 can be replaced with transitions to s_1. In effect, the two states are merged together into one common state.

How do we decide what states to merge together?

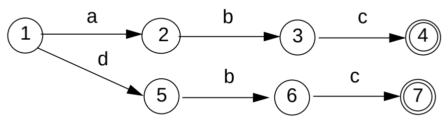
We take a *greedy* approach and try the most optimistic merger of states. By definition, accepting and non-accepting states are distinct, so we initially try to create only two states: one representing the merger of all accepting states and the other representing the merger of all non-accepting states.

This merger into only two states is almost certainly too optimistic. In particular, all the constituents of a merged state must agree on the same transition for each possible character. That is, for character c all the merged states must have no successor under c or they must all go to a single (possibly merged) state.

If all constituents of a merged state do not agree on the

transition to follow for some character, the merged state is *split* into two or more smaller states that do agree.

As an example, assume we start with the following automaton:



Initially we have a merged non-accepting state {1,2,3,5,6} and a merged accepting state {4,7}.

A merger is legal if and only if all constituent states agree on the same successor state for all characters. For example, states 3 and 6 would go to an accepting state given character c; states 1, 2, 5 would not, so a split must occur.

We will add an error state s_E to the original DFA that is the successor state under any illegal character. (Thus reaching s_E becomes equivalent to detecting an illegal token.) s_E is not a real state; rather it allows us to assume every state has a successor under every character. s_E is never merged with any real state.

Algorithm **`Split`**, shown below, splits merged states whose constituents do not agree on a common successor state for all characters. When **`Split`** terminates, we know that the states that remain merged are equivalent in that they always agree on common successors.

```
Split(FASet StateSet) {
 repeat
  for(each merged state S in StateSet) {
    Let S correspond to {$s_1$,…,$s_n$}
    for(each char c in Alphabet){
      Let $t_1$,…,$t_n$ be the successor
       states to $s_1$,…,$s_n$ under c
      if($t_1$,…,$t_n$ do not all belong to
          the same merged state){
           Split S into two or more new
           states such that $s_i$ and $s_j$
           remain in the same merged
           state if and only if $t_i$ and $t_j$
           are in the same merged state}
      }
 until no more splits are possible
}
```

Returning to our example, we initially have states {1,2,3,5,6} and {4,7}. Invoking `Split`, we first observe that states 3 and 6 have a common successor under c, and states 1, 2, and 5 have no successor under c (equivalently, have the error state s_E as a successor).

This forces a split, yielding {1,2,5}, {3,6} and {4,7}.

Now, for character b, states 2 and 5 would go to the merged state {3,6}, but state 1 would not, so another split occurs.

We now have: {1}, {2,5}, {3,6} and {4,7}.

At this point we are done, as all constituents of merged states agree on the same successor for each input symbol.

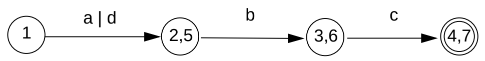
Once `Split` is executed, we are essentially done.

Transitions between merged states are the same as the transitions between states in the original DFA.

Thus, if there was a transition between state s_i and s_j under character c, there is now a transition under c from the merged state containing s_i to the merged state containing s_j. The start state is that merged state containing the original start state.

Accepting states are those merged states containing accepting states (recall that accepting and non-accepting states are never merged).

Returning to our example, the minimum state automaton we obtain is



Properties of Regular Expressions and Finite Automata

- Some token patterns *can't* be defined as regular expressions or finite automata. Consider the set of balanced brackets of the form [[[...]]]. This set is defined formally as
  $\{ [^m]^m \mid m \geq 1 \}$.
  This set is *not* regular.
  No finite automaton that recognizes *exactly* this set can exist.
  Why? Consider the inputs [, [[, [[[, ...
  For two different counts (call them i and j) $[^i$ and $[^j$ must reach the same state of a given FA! (Why?)
  Once that happens, we know that if $[^i]^i$ is accepted (as it should be), the $[^j]^i$ will also be accepted (and that should not happen).

- $\overline{R} = V^* - R$ is regular if R is.
  Why?
  Build a finite automaton for R. Be careful to include transitions to an "error state" s_E for illegal characters.
  Now invert final and non-final states. What was previously accepted is now rejected, and what was rejected is now accepted. That is, $\overline{R}$ is accepted by the modified automaton.

- Not all subsets of a regular set are themselves regular. The regular expression $[^+]^+$ has a subset that isn't regular. (What is that subset?)

- Let R be a set of strings. Define R^{rev} as all strings in R, in reversed (backward) character order.
  Thus if R = {abc, def}
  then R^{rev} = {cba, fed}.
  If R is regular, then R^{rev} is too.
  Why? Build a finite automaton for R. Make sure the automaton has only one final state. Now *reverse* the direction of all transitions, and interchange the start and final states. What does the modified automation accept?

- If R_1 and R_2 are both regular, then $R_1 \cap R_2$ is also regular. We can show this two different ways:
  1. Build two finite automata, one for R1 and one for R2. Pair together states of the two automata to match R1 and R2 simultaneously. The paired-state automaton accepts only if both R1 and R2 would, so $R1 \cap R2$ is matched.
  2. We can use the fact that $R1 \cap R2$ is $= \overline{\overline{R_1} \cup \overline{R_2}}$ We already know union and complementation are regular.