A lexer generator converts a lexical specification consisting of a list of regular expressions and corresponding actions into code that breaks the input into tokens. In this lecture we examine how this is done.

We can think of the lexical specification as a big regular expression $R_1 \mid R_2 \mid \dots R_n$ where the R_i are the descriptions of each of the token.

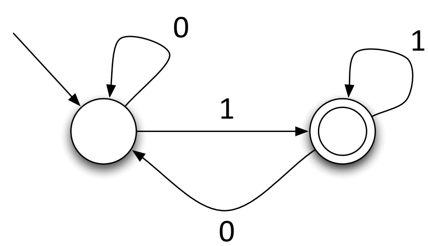
A lexer generator works by converting this regular expression into a deterministic finite automaton (DFA). This is done in a couple of steps. First, the regular expression is converted into a nondeterministic finite automaton (NFA). The NFA is converted into a DFA, which then becomes the basis for a table-driven lexer.

1 DFAs

We start by reviewing DFAs. A DFA is a abstract machine:

- The machine reads an input stream of symbols $x \in \Sigma$, where Σ is the *alphabet* of the DFA.
- It has a finite set of states q_i.
- There is a distinguished *initial state* q_0 in which the machine begins reading its input.
- As the machine reads each symbol, it changes its state according to a *transition function* δ. On reading symbol x in state q, it changes to the new state q' where $q' = \delta(q, x)$.
- It has a set of *final* or *accept* states F. The machine *accepts* the input if it arrives at the end of the input in a final state $q \in F$.

A DFA can be drawn as a labeled graph in which states are nodes, the initial state q_0 is indicated by an incoming edge from outside, other edges are labeled with the corresponding input symbol, and final states in F are marked by nodes with double circles. For example, consider the following DFA, which accepts only odd numbers expressed in binary, corresponding to the regular expression $(0|1)^*1$:



We can model illegal characters by adding a non-final *error state* to the DFA, which we may not bother to draw in such a diagram. Every state has transitions to the error state on every symbol that cannot lead to a final state. Therefore δ is total.

We can describe the transition function δ as a table, which hints at how we might implement the DFA:

	0	1
q_0	q_0	q_1
q_1	q_0	q_1

Pseudo-code for implementing a DFA that reads an input of length n, where `input[i]` is the ith input character, looks roughly like this:

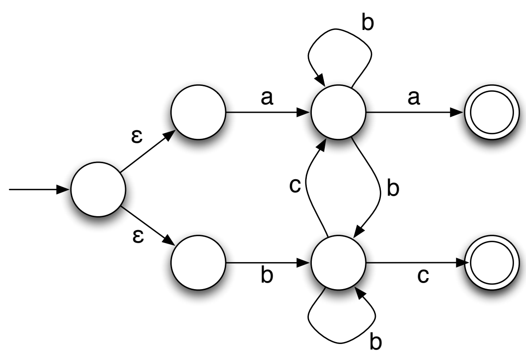
```
start := i
q := q0
while (i ≤ n) {
    q := δ(q, input[i])
    i := i + 1
}
if (q ∈ F) return accept
else return fail
```

Now the question is how to obtain the table δ from a regular expression.

2 NFA

The first step is to convert the regular expression into a *nondeterministic* finite automaton. An NFA differs from a DFA in that each state can transition to zero or more other states on each input symbol, and a state can also transition to others without reading a symbol. In the diagram representation, multiple exiting edges can be labeled with the same symbol. Edges corresponding to not reading a symbol are labeled with ε.

For example, the following is an NFA:

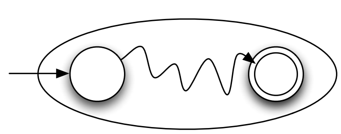


Given an input stream, the NFA accepts if there is *any way* to reach a final state. That is, it has *angelic nondeterminism*. We imagine there is an angel or oracle telling it which transitions to take. If the machine above receives the input "aba", it can reach a final state by choosing the upper ε-transition, and staying within the top three states. Therefore the machine accepts this input. It does not accept "ac", however, because there is no way to reach a final state while reading that input. (Can you write a regular expression that describes exactly the strings that this NFA accepts?)

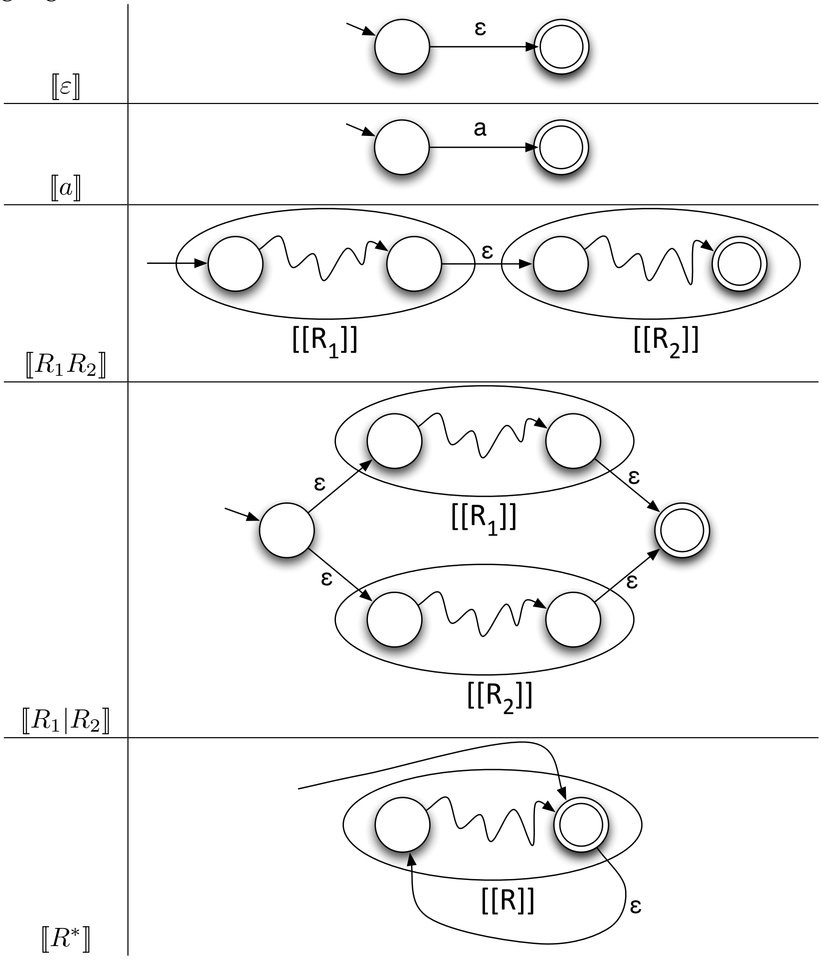
3 RE to NFA

We show how to translate a regular expression to an equivalent NFA by induction on the structure of the regular expression. That is, given that we know how to convert the subexpressions of a regular expression, we show how to use the NFAs produced by those translations to produce the NFA for the full expression.

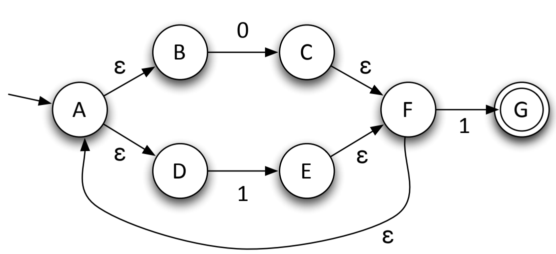
In each case, the result of translating a regular expression will be an NFA with a single accept state, which we represent with the following diagram:



Let us write $\llbracket R \rrbracket$ to mean the translation of regular expression R to an NFA that accepts exactly the language of R. We define it as follows:



By working bottom up, we can use these translations to construct an NFA for any regular expression. For example, the odd number regular expression above, $(0|1)^*1$, translates to the following NFA, which clearly accepts the same strings. (The states in this diagram are labeled with names A–G for later use).



4 NFA to DFA

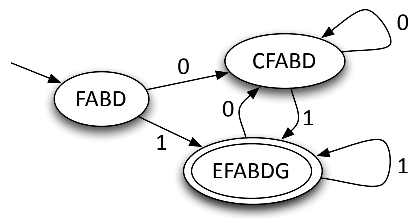
Although an NFA can do anything a DFA can, the reverse is also true. We can convert an arbitrary NFA into a DFA (though the DFA may in general be exponentially larger than the NFA). The intuition is that we make a DFA that simulates all possible executions of the NFA. At any given point in the input stream, the NFA could be in some set of states. For each *set* of states the NFA could be in during its execution, we create a state in the DFA. There is even in general a state $\emptyset$ in the DFA, to describe the case in which no NFA state is reachable using the input seen up to a certain point.

Since ε transitions can be taken at any time, it is useful to have the concept of the *ε-closure* of a state q. It is the set of all states reachable from q using zero or more ε-transitions. Similarly, we can can take the ε-closure of a *set* of states by finding all states reachable from any state in the set using only ε-transitions.

For example, in the odd-number NFA above, the ε-closure of F is the set $\varepsilon\text{-closure}(()F) = \{F, A, B, D\}$. The ε-closure of $\{E, G\}$ is $\varepsilon\text{-closure}(E) \cup \varepsilon\text{-closure}(G) = \{E, F, A, B, D, G\}$.

Now let us discover which set of states are reachable in this NFA and construct the corresponding DFA.

The initial state of the DFA is the ε-closure of the start state of the NFA: that is, $\varepsilon\text{-closure}(F) = \{F, A, B, D\}$. From that set of states we can take a transition on either 0 or 1. A transition on 0 can only happen from state B to state C, so the DFA state reached is $\varepsilon\text{-closure}(C) = \{C, F, A, B, D\}$. From either of these two DFA states, we can transition on 1 to reach states E and G, so the final DFA state is $\varepsilon\text{-closure}(\{E, G\}) = \{E, F, A, B, D, G\}$. The full DFA looks as follows:



5 DFA minimization

In general the DFA generated by this procedure may have more states than necessary. John Hopcroft showed that it is possible to *minimize* a DFA by merging states. Let us write $q_1 \not\approx q_2$ if merging states q_1 and q_2 would change the language accepted by the DFA; in this case we say that q_1 and q_2 are distinguishable.

Clearly, two states are distinguishable if one of them is final and one of them is non-final. We can express this idea as the following reasoning rule:

$$\frac{q_1 \in F \quad q_2 \notin F}{q_1 \not\approx q_2} \text{ (Rule 1)}$$

Two states are also distinguishable if following the the same symbol from each of them leads to distinguishable states:

$$\frac{q_1' = \delta(q_1, x) \quad q_2' = \delta(q_2, x) \quad q_1' \not\approx q_2'}{q_1 \not\approx q_2} \text{ (Rule 2)}$$

If we can use these two rules to infer that two states are distinguishable, they must be distinguishable. Conversely, if we can't infer that two states are distinguishable by these rules, then merging the states will not change which strings the DFA accepts.

Algorithmically, we keep track of whether each pair of states q_i and q_j are distinguishable, starting from the supposition that they are not distinguishable. We mark all final/non-final pairs distinguishable, by Rule 1. We then apply Rule 2. We follow similarly-labeled edges backward from all distinguishable states to identify additional pairs of states that are distinguishable. Eventually no more distinguishable pairs can be identified.

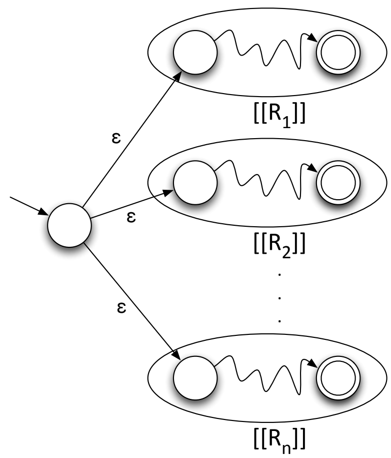
For the odd-number DFA, the result is as shown in the following table:

	FABD	CFABD
CFABD		
EFABDG	$\not\approx$	$\not\approx$

By Rule 1, states FABD and CFABD must both be distinguishable from EFABDG, as indicated by the $\not\approx$ in the table. Rule 2 cannot be applied to either of these pairs of distinguishable states, so we are done. Since FABD and CFABD were not distinguishable, they can be merged, giving us exactly the 2-state DFA shown at the beginning of the notes.

6 Building an efficient lexer

Now we can construct an efficient DFA for an arbitrary lexical specification $R_1 \mid R_2 \mid \dots R_n$. We construct an alternation NFA in which we keep the final states distinct so they can be associated with the appropriate lexer action.



We convert this to a DFA but continue to mark each final state with the corresponding action.

Recall that we are implementing the longest-matching token rule. So we when we hit an accept state in the DFA, we remember that it was encountered and keep reading ahead. We only stop when we get to the DFA error state $\emptyset$, meaning that there is no way to read more symbols to build a longer token. At that point we rewind the state of the input back to the last final state and construct a token out of the symbols seen to that point. To implement this backtracking, we assume our input stream has an `unread(c)` operation that lets us put a character back into the stream, in addition to the `peek()` operation that allows inspecting the new character.

To find the last final state, we keep track of the states we have seen by pushing them onto a stack. (It might seem harmless by unnecessary to remember the non-final states seen along the way, but we will use this for an optimization shortly.)

```
start := i
q := q0
// read ahead until stuck
while (true) {
    input[i] = read()
    if (input[i] = EOF or δ(q, input[i] = ∅)) break
    if (q∈F) clear the stack
    push q
    q := δ(q, input[i])
    i := i + 1
}
// backtrack to last final state
while (q ∉ F) {
    if (stack is empty) fail
    q := pop()
    unread(token[i])
    i := i - 1
}
return input[start..i-1]
```

For most lexical specifications, this algorithm will be fairly efficient and take time linear in the number of characters on the input. In the worst case, it can be quadratic because of backtracking. For example, consider the lexical specification abc | (abc)*d, with input consisting of these n characters: "abcabcabc...abc". The correct result is a sequence of abc tokens, but the lexer must read all the way to the end of the input to find whether there is a d. The above algorithm will backtrack $n/3$ times on this input, taking $\Theta(n^2)$ time.

To make the algorithm more efficient, we memoize hopeless lexer states. If during backtracking we see that some non-final state q was encountered at position i, there is no reason to try finding a token again from that state and position. We add a memoization table hopeless[q,i] to record such scanner states, and modify the algorithm above to update and use this information:

```
start := i
q := q0
// read ahead until stuck
while (true) {
    if (hopeless[q,i]) break
    input[i] = read()
    if (input[i] = EOF or δ(q, input[i] = ∅)) break
    if (q∈F) clear the stack
    push q
    q := δ(q, input[i])
    i := i + 1
}
// backtrack to last final state
while (q ∉ F) {
    hopeless[q,i] := true
    if (stack is empty) fail
    q := pop()
    i := i - 1
    unread(input[i])
}
return input[start..i-1]
```

In the example of lexing "abcabcabc...abc", the modified lexer reads all the input to find the first token, but on the second and following tokens does not read past the characters that make up each of the abc tokens.

This algorithm, due to Tom Reps, ensures lexical analysis takes linear time.

7 More reading

Hopcroft and Ullman. "Introduction to automata theory, languages, and computation," Chapter 2. Addison-Wesley, 1979.

Thomas Reps. " 'Maximal-munch' tokenization in linear time". ACM Transactions on Programming Languages and Systems, 20(2):259–273, March 1998.