Automating Construction of Lexers
Regular Expression to Programs

- Not all regular expressions are simple.
- How can we write a lexer for \((a^*b \mid aaa)\)?
- Tokenizing \(aaaab\) Vs \(aaaaaa\)
Finite State Automaton
(Finite State Machine)

- $A = (\Sigma, Q, q_0, \delta, F)$
- $\delta \subseteq Q \times \Sigma \times Q$,
  $q_0 \in Q$,
  $F \subseteq Q$
  $q_0 \in Q$
  $q_1 \subseteq Q$
  $\delta = \{ (q_0, a, q_1), (q_0, b, q_0),$
  $(q_1, a, q_1), (q_1, b, q_1), \}$

- $\Sigma$ - alphabet
- $Q$ - states (nodes in the graph)
- $q_0$ - initial state (with ‘->’ sign in drawing)
- $\delta$ - transitions (labeled edges in the graph)
- $F$ - final states (double circles)
Numbers with Decimal Point

digit digit* . digit digit*

What if the decimal part is optional?
Kinds of Finite State Automata

- **DFA:** \( \delta \) is a function: \((Q, \Sigma) \rightarrow Q\)
- **NFA:** \( \delta \) could be a relation

- In NFA there is no unique next state. We have a set of possible next states.
Remark: Relations and Functions

• **Relation** \( r \subseteq B \times C \)
  \[ r = \{ ..., (b,c_1) , (b,c_2) , ... \} \]

• **Corresponding function:** \( f : B \rightarrow 2^C \)
  \[ f = \{ ... (b,\{c_1,c_2\}) ... \} \]
  \[ f(b) = \{ c \mid (b,c) \in r \} \]

• Given a state, next-state function returns the set of new states
  - for deterministic automaton, the set has exactly 1 element
Allowing Undefined Transitions

• Undefined transitions lead to a sink state from where no input can be accepted
Allowing Epsilon Transitions

- Epsilon transitions:
  - traversing them does not consume anything

- Transitions labeled by a word:
  - traversing them consumes the entire word
Interpretation of Non-Determinism

• A word is accepted if there is a path in the automaton that leads to an accepting state on reading the word

Eg.

• Does the automaton accept ‘a’?
  - yes
Exercise

• Construct a NFA that recognizes all strings over \{a,b\} that contain "aba" as a substring
Running NFA (without epsilons)

```scala
def δ(a : Char)(q : State) : Set[States] = { ... }
def δ'(a : Char, S : Set[States]) : Set[States] = {
    for (q1 <- S, q2 <- δ(a)(q1)) yield q2 // S.flatMap(δ(a))
}
def accepts(input : MyStream[Char]) : Boolean = {
    var S : Set[State] = Set(q0) // current set of states
    while (!input.EOF) {
        val a = input.current
        S = δ'(a,S) // next set of states
    }
    !(S.intersect(finalStates).isEmpty)
}
```
NFA Vs DFA

• For every NFA there exists an equivalent DFA that accepts the same set of strings

• But, NFAs could be exponentially smaller (succinct)

• There are NFAs such that every DFA equivalent to it has exponentially more number of states
Theorem:
If L is a set of words, it is describable by a regular expression iff (if and only if) it is the set of words accepted by some finite automaton.

Algorithms:
• regular expression → automaton (important!)
• automaton → regular expression (cool)
Recursive Constructions

• Union

• Concatenation
Recursive Constructions

- Star
Exercise: \((aa)^* \mid (aaa)^*\)

- Construct an NFA for the regular expression
NFAs to DFAs (Determinisation)

- keep track of a set of all possible states in which the automaton could be

- view this finite set as one state of new automaton
NFA to DFA Conversion

Possible states of the DFA: $2^Q$

\[
\{ \{ \} , \{ 0 \}, \ldots, \{12\}, \{0, 1\}, \ldots, \{0, 12\}, \ldots, \{12, 12\}, \{0, 1, 2\} \ldots, \{0, 1, 2\ldots, 12\} \} \]
NFA to DFA Conversion
NFA to DFA Conversion

• DFA: \((\Sigma, 2^Q, q'_0, \delta', F')\)

• \(q'_0 = E(q_0)\)

• \(\delta'(q', a) = \bigcup_{\exists q_1 \in q', \delta(q_1, a, q_2)} E(q_2)\)

• \(F' = \{ q' | q' \in 2^Q, q' \cap F \neq \emptyset \}\)
NFA to DFA Conversion
NFA to DFA Example
Clarifications

• what happens if a transition on an alphabet ‘a’ is not defined for a state ‘q’?

• $\delta'(\{q\}, a) = \emptyset$

• $\delta'(\emptyset, a) = \emptyset$

• Empty set represents a state in the NFA

• It is a trap/sink state: a state that has self-loops for all symbols, and is non-accepting.
Minimizing DFAs to Keep Them Small

• First, throw away all unreachable states: those for which there is no path to them from the initial state
Minimizing DFAs: Procedure

- Write down all pairs of state as a table
- Every cell in the table denotes whether the corresponding states are equivalent

<table>
<thead>
<tr>
<th></th>
<th>q1</th>
<th>q2</th>
<th>q3</th>
<th>q4</th>
<th>q5</th>
</tr>
</thead>
<tbody>
<tr>
<td>q1</td>
<td>x</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>q2</td>
<td>x</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>q3</td>
<td>x</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>q4</td>
<td></td>
<td>x</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
<tr>
<td>q5</td>
<td></td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
Minimizing DFAs: Procedure

• Initialize cells \((q_1, q_2)\) to false if one of them is final and other is non-final

• Make the cell \((q_1, q_2)\) false, if \(q_1 \rightarrow q_1'\) on some alphabet symbol and \(q_2 \rightarrow q_2'\) on ‘a’ and \(q_1'\) and \(q_2'\) are not equivalent

• Iterate the above process until all non-equivalent states are found
Properties of Automata

Complement:
• Given a DFA $A$, switch accepting and non-accepting states in $A$ gives the complement automaton $A^c$
• $L(A^c) = (\Sigma^* \setminus L(A))$

Note this does not work for NFA

Intersection: $L(A') = L(A_1) \cap L(A_2)$

$-A' = (\Sigma, Q_1 \times Q_2, (q^1_0, q^2_0), \delta', F_1 \times F_2)$

$-\delta'( (q_1, q_2), a ) = \delta(q_1, a) \times \delta(q_2, a)$

Emptiness of language, inclusion of one language into another, equivalence – they are all decidable
Exercise 0.1: on Equivalence

Prove that \((a^*b^*)^*\) is equivalent to \((a|b)^*\)
Sequential Circuits are Automata

\[ A = (\Sigma, Q, q_0, \delta, F) \]

Q – states of flip-flops, registers, etc.
\( \delta \) – combinational circuit that determines next state