Computing 'nullable' for regular expressions

If e is regular expression (its syntax tree), then $L(e)$ is the language denoted by it.
For $L \subseteq A^*$ we defined $nullable(L)$ as $\varepsilon \in L$
If e is a regular expression, we can compute $nullable(e)$ to be equal to $nullable(L(e))$, as follows:

$$
\begin{array}{rcl}
nullable(\emptyset) & = & false \\
nullable(\varepsilon) & = & true \\
nullable(a) & = & false \\
nullable(e_1|e_2) & = & nullable(e_1) \vee nullable(e_2) \\
nullable(e^*) & = & true \\
nullable(e_1 e_2) & = & nullable(e_1) \wedge nullable(e_2)
\end{array}
$$

Computing 'first' for regular expressions

For $L \subseteq A^*$ we defined: $\mathit{first}_L(L) = \{a \in A \mid \exists v \in A^*.\ av \in L\}$.
If e is a regular expression, we can compute $\mathit{first}(e)$ to be equal to $\mathit{first}(L(e))$, as follows:

$\mathit{first}(e) = \mathit{first}_L(L(e))$

$$
\begin{aligned}
\mathit{first}(\emptyset) &= \emptyset \\
\mathit{first}(\varepsilon) &= \emptyset \\
\mathit{first}(a) &= \{a\},\ \text{for } a \in A \\
\mathit{first}(e_1|e_2) &= \mathit{first}(e_1) \cup \mathit{first}(e_2) \\
\mathit{first}(e*) &= \mathit{first}(e) \\
\mathit{first}(e_1 e_2) &= \begin{cases} \mathit{if}\,(\mathit{nullable}(e_1))\ \mathit{then}\ \mathit{first}(e_1) \cup \mathit{first}(e_2) \\ \mathit{else}\ \ \mathit{first}(e_1) \end{cases}
\end{aligned}
$$

$\varepsilon \in L(e_1)$

Clarification for first of concatenation

Let e be $\mathbf{a}^*\mathbf{b}$. Then $L(e) = \{b, ab, aab, aaab, \ldots\}$
$\mathit{first}(L(e)) = \{a, b\}$

$e = e_1 e_2$ where $e_1 = a^*$ and $e_2 = b$. Thus, $\mathit{nullable}(e_1)$.

$$\mathit{first}(e_1 e_2) = \mathit{first}(e_1) \cup \mathit{first}(e_2) = \{a\} \cup \{b\} = \{a, b\}$$

It is *not correct* to use $\mathit{first}(e) =^? \ \mathit{first}(e_1) = \{a\}$.
Nor is it correct to use $\mathit{first}(e) =^? \ \mathit{first}(e_2) = \{b\}$.
We must use their union.

Converting Simple Regular Expresssions into a Lexer

regular expression	*lexercode*
$a \quad (a \in A)$	*if* $(current = a)$ *next else ...*
$r_1 r_2$	$code(r_1);\ code(r_2)$
$r_1 \| r_2$	*if* $(current \in first(r_1))$ $\quad code(r_1)$ *else* $code(r_2)$
$r*$	*while* $(current \in first(r))$ $\quad code(r)$

More complex cases

In other cases, a few upcoming characters ("lookahead") are not sufficient to determine which token is coming up.

Examples:
A language might have separate numeric literal tokens to simplify type checking:

- integer constants: *digit digit**
- floating point constants: *digit digit** . *digit digit**

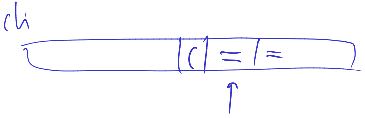
10112301.21

Floating point constants must contain a period (e.g., Modula-2).

Division sign begins with same character as // comments.
Equality can begin several different tokens.

In such cases, we process characters and store them until we have enough information to make the decision on the current token.

Example of a part of a lexical analyzer



```
ch.current match {
  case '(' ⇒ {current = OPAREN; ch.next; return}
  case ')' ⇒ {current = CPAREN; ch.next; return}
  case '+' ⇒ {current = PLUS; ch.next; return}
  case '/' ⇒ {current = DIV; ch.next; return}
  case '*' ⇒ {current = MUL; ch.next; return}
  case '=' ⇒ { // more tricky because there can be =, ==
    ch.next
    if (ch.current == '=') {ch.next; current = CompareEQ; return}
    else {current = AssignEQ; return}
  }
  case '<' ⇒ { // more tricky because there can be <, <=
    ch.next
    if (ch.current == '=') {ch.next; current = LEQ; return}
    else {current = LESS; return}
  }
}
```

==

x= 3;

Example of a part of a lexical analyzer

<=

<= =

```
ch.current match {
  case '(' => {current = OPAREN; ch.next; return}
  case ')' => {current = CPAREN; ch.next; return}
  case '+' => {current = PLUS; ch.next; return}
  case '/' => {current = DIV; ch.next; return}
  case '*' => {current = MUL; ch.next; return}
  case '=' => { // more tricky because there can be =, ==
    ch.next
    if (ch.current == '=') {ch.next; current = CompareEQ; return}
    else {current = AssignEQ; return}
  }
  case '<' => { // more tricky because there can be <, <=
    ch.next
    if (ch.current == '=') {ch.next; current = LEQ; return}
    else {current = LESS; return}
  }
}
```

What if we omit ch.next?

Example of a part of a lexical analyzer

```
ch.current match {
  case '(' => {current = OPAREN; ch.next; return}
  case ')' => {current = CPAREN; ch.next; return}
  case '+' => {current = PLUS; ch.next; return}
  case '/' => {current = DIV; ch.next; return}
  case '*' => {current = MUL; ch.next; return}
  case '=' => { // more tricky because there can be =, ==
    ch.next
    if (ch.current == '=') {ch.next; current = CompareEQ; return}
    else {current = AssignEQ; return}
  }
  case '<' => { // more tricky because there can be <, <=
    ch.next
    if (ch.current == '=') {ch.next; current = LEQ; return}
    else {current = LESS; return}
  }
}
```

What if we omit ch.next?

Lexer could generate a non-existing equality token!

White spaces and comments

$x \overbrace{\sqcup\sqcup\sqcup} + \quad y$ | xy $x\sqcup y$

Whitespace can be defined as a token, using space character, tabs, and various end of line characters. Similarly for comments.

In most languages (Java, ML, C) white spaces and comments can occur between any two other tokens have no meaning, so parser does not want to see them.

$(32 \mid 13 \mid 10 \mid 9)^*$ // ... /* /**/ */

Convention: the lexical analyzer removes those "tokens" from its output. Instead, it always finds the next non-whitespace non-comment token.

Other conventions and interpretations of new line became popular to make code more concise (sensitivity to end of line or indentation). Not our problem in this course! Tools that do formatting of source also must remember comments. We ignore those.

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
    while (!isEOL && !isEOF) {
      ch.next
    }
  } else {
```

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
    while (!isEOL && !isEOF) {
      ch.next
    }
  } else {
    ch.current = DIV
  }
}
```

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
    while (!isEOL && !isEOF) {
      ch.next
    }
  } else {
    ch.current = DIV
  }
}
```

Nested comments: this is a single comment:

/* foo /* bar */ baz */

Solution:

Skipping simple comments

```
if (ch.current='/') {
  ch.next
  if (ch.current='/') {
    while (!isEOL && !isEOF) {
      ch.next
    }
  } else {
    ch.current = DIV
  }
}
```

Nested comments: this is a single comment:

/* foo /* bar */ baz */

Solution: use a counter for nesting depth

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	$<=$
LT:	$<$
EQ:	$=$

How can we split this input into subsequences, each of which in a token:

$$\textit{interpreters} <= \textit{compilers}$$

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	$<=$
LT:	$<$
EQ:	$=$

How can we split this input into subsequences, each of which in a token:

interpreters $<=$ *compilers*

Some solutions:

ID(interpreters) LE ID(compilers) - OK, longest match rule

ID(inter) ID(preters) LE ID(compilers)

ID(interpreters) LT EQ ID(compilers)

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	$<=$
LT:	$<$
EQ:	$=$

How can we split this input into subsequences, each of which in a token:

interpreters $<=$ *compilers*

Some solutions:

ID(interpreters) LE ID(compilers) - OK, longest match rule

ID(inter) ID(preters) LE ID(compilers)

- not longest match: ID(inter)

ID(interpreters) LT EQ ID(compilers)

Longest match (maximal munch) rule

Lexical analyzer is required to be greedy: always get the longest possible token at this time. Otherwise, there would be too many ways to split input into tokens!

Consider language with the following tokens:

ID:	letter(digit | letter)*
LE:	<=
LT:	<
EQ:	=

How can we split this input into subsequences, each of which in a token:

interpreters <= *compilers*

Some solutions:

ID(interpreters) LE ID(compilers) - OK, longest match rule

ID(inter) ID(preters) LE ID(compilers)

- not longest match: ID(inter)

ID(interpreters) LT EQ ID(compilers)

- not longest match: LT

Longest match rule is greedy, but that's OK

Consider language with ONLY these three operators:

LT:	$<$
LE:	$<=$
IMP:	$=>$

For sequence:

$$<=>$$

lexer will first return LE as token, then report unknown token $>$.
This is the behavior that we expect.

This is despite the fact that one could in principle split the input into $<$ and $=>$, which correspond to sequence LT IMP. But a split into $<$ and $=>$ would not satisfy longest match rule, so we do *not* want it. Reporting error is the right thing to do here.

This behavior is not a restriction in practice: programmers we can insert extra spaces to stop maximal munch from taking too many characters.

Token priority

token (<= token class

identifiers

What if our token classes intersect?

Longest match rule does not help, because the same string belongs to two regular expressions

Examples:

- a keyword is also an identifier
- a constant that can be integer or floating point

Solution is priority: order all tokens and in case of overlap take one earlier in the list (higher priority).

Examples:

- if it matches regular expression for both a keyword and an identifier, then we define that it is a keyword.
- if it matches both integer constant and floating point constant regular expression, then we define it to be (for example) integer constant.

Token priorities for overlapping tokens must be specified in language definition.

Automating Construction of Lexers
by converting
Regular Expressions to Automata

Regular Expression to Programs

- How can we write a lexer that has these two classes of tokens:
  - a*b
  - aaa

$\overbrace{aaaa}^{n \geq 0}b$

- Consider run of lexer on: **aaaab** and on: **aaaaaa**

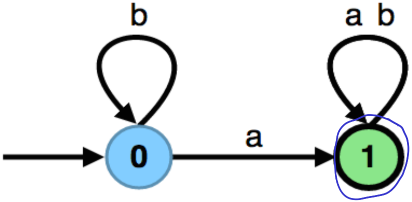
Regular Expression to Programs

- How can we write a lexer that has these two classes of tokens:
  - a*b
  - aaa
- Consider run of lexer on: **aaaab** and on: **aaaaaa**
- A general approach:

Finite Automaton (Finite State Machine)

$M\ \cancel{A} = (\Sigma, Q, q_0, \delta, F)$

$$\delta \subseteq Q \times \Sigma \times Q,$$
$$q_0 \in Q,$$
$$F \subseteq Q$$



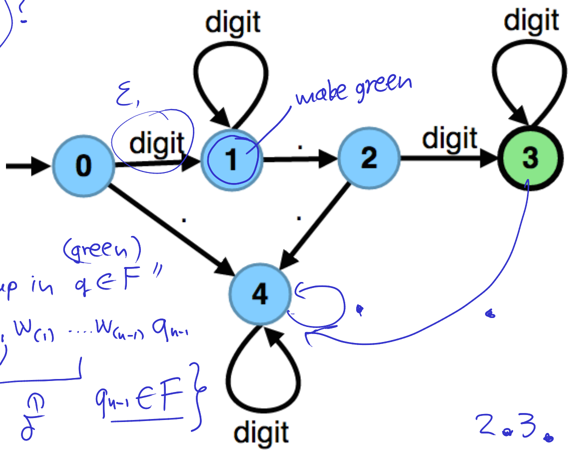
$$q_0 \in Q$$
$$q_1 \subseteq Q$$
$$\delta = \{ (q_0, a, q_1), (q_0, b, q_0), (q_1, a, q_1), (q_1, b, q_1), \}$$

- Σ - alphabet
- Q - states (nodes in the graph)
- q_0 - initial state (with ‘->' sign in drawing)
- δ - transitions (labeled edges in the graph)
- F - final states (double circles)

Numbers with Decimal Point

digit digit* (. digit digit*)?

$\Sigma = \{\bullet, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$



$w \in L(M)$ $\quad (e)^? \equiv e \mid \varepsilon$

$L(M) = \{ w \mid$ "if we run w we end up in $q \in F$" (green)

if $\exists$ path $\underbrace{q_0\, w_{(0)}\, q_1}_{\delta}\, w_{(1)} \dots w_{(n-1)}\, q_{n-1}$ $\quad \underline{q_{n-1} \in F} \}$

~~n=|w|~~

$q_0 s_0 q_1 s_1 \dots s_n\ q_{n-1}$

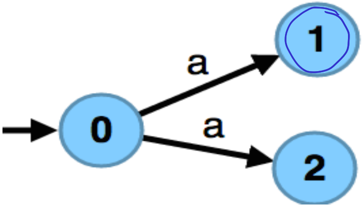
$s_0 s_1 \dots s_{n-1} = w$

What if the decimal part is optional?

Kinds of Finite State Automata

- DFA: δ is a function : $(Q, \Sigma) \mapsto Q$
- NFA: δ could be a relation $\delta \subseteq Q \times \Sigma \times Q$

- In NFA there is no unique next state. We have a set of possible next states.



$a \in L$

Remark: Relations and Functions

- Relation $r \subseteq \overbrace{B}^{Q \times \Sigma} \times \overbrace{C}^{Q}$

  $\delta \subseteq (Q \times \Sigma) \times Q$

  r = { …, (b,c1) , (b,c2) ,… }

  $r = \{ (b,c) \mid c \in f(b) \}$

- Corresponding function: f : B -> 2^C

  f = { … (b,{c1,c2}) … }

  $\delta_f : Q \times \Sigma \to 2^Q$

f(b) = { c | (b,c) ∈ r }

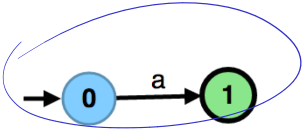
- Given a state, next-state function returns a **set** of new states

  for *deterministic* automaton, set has *exactly 1* element

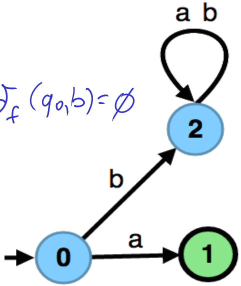
Allowing Undefined Transitions

$\Sigma = \{a, b\}$

no $(q_0, b, -)$

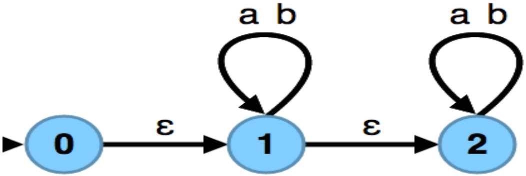


$\delta_f(q_0, b) = \emptyset$



- Undefined transitions are equivalent to transition into a sink state (from which one cannot recover)

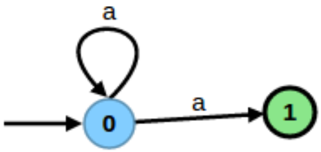
Allowing Epsilon Transitions



- Epsilon transitions:
  - traversing them does not consume anything
- Transitions labeled by a word:
  - traversing them consumes the entire word

When Automaton Accepts a Word

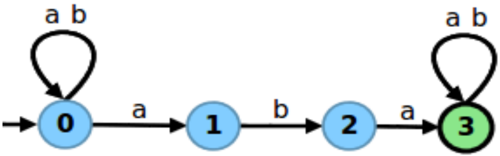
Automaton accepts a word *w* iff there **exists a path** in the automaton from the starting state to some accepting state such that concatenation of words on the path gives *w*.



- Does the automaton accept the word ***a*** ?

Exercise

- Construct a NFA that recognizes all strings over {a,b} that contain "aba" as a substring $= \{ w_1 \, aba \, w_2 \mid w_1, w_2 \in \{a,b\}^* \}$

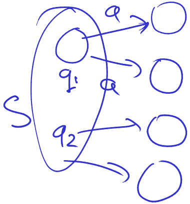


$a \, aba$

Running NFA (without epsilons)

$\delta_f : (State \times \Sigma) \to 2^{State}$

```
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

- Every DFA is also a NFA (they are a special case)
- 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

Regular Expressions and Automata

Theorem:

Let L be a language. There exists a regular expression that describes it if and only if there exists a finite automaton that accepts it.

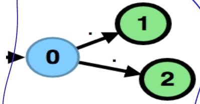
Algorithms:

- regular expression → automaton (important!)
- automaton → regular expression (cool)

Recursive Constructions

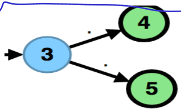
r_1

A_1:



r_2

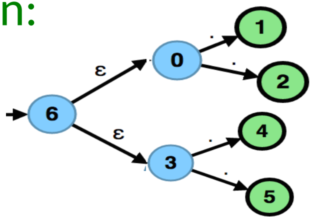
A_2:



$\boxed{r_1 \mid r_2}$

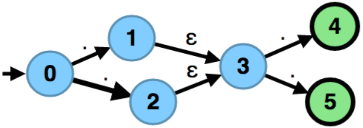
A_1 A_2

Union:



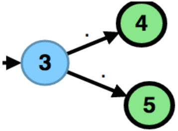
$r_1 \mid r_2$

Concatenation:

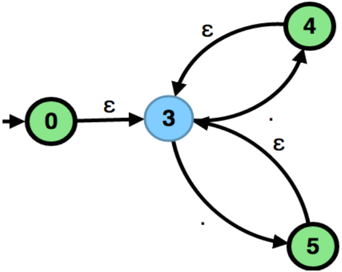


$r_1 \ r_2$

Recursive Constructions

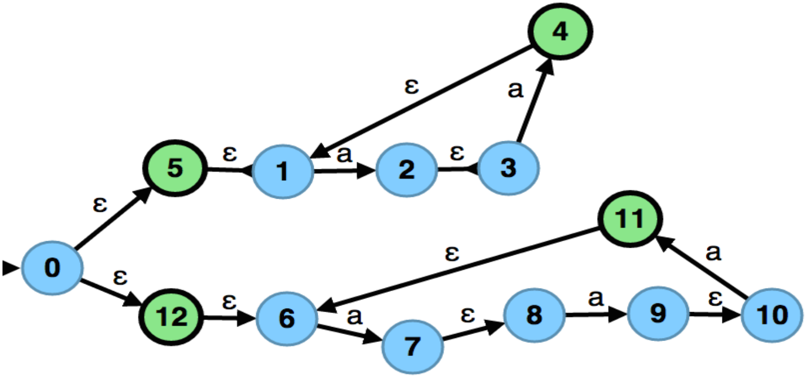


Star:



Exercise: (aa)* | (aaa)*

- Construct an NFA for the regular expression

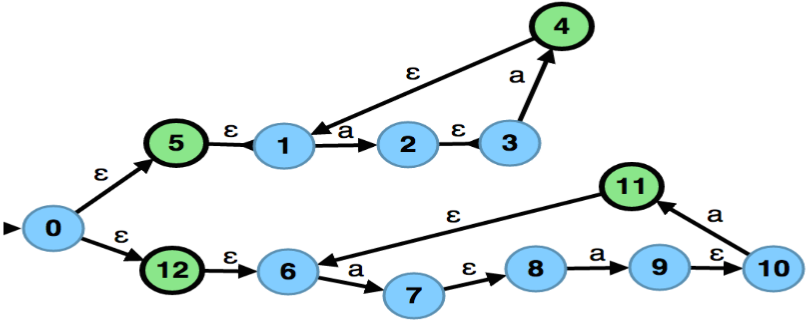


NFAs to DFAs (Determinization)

- 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

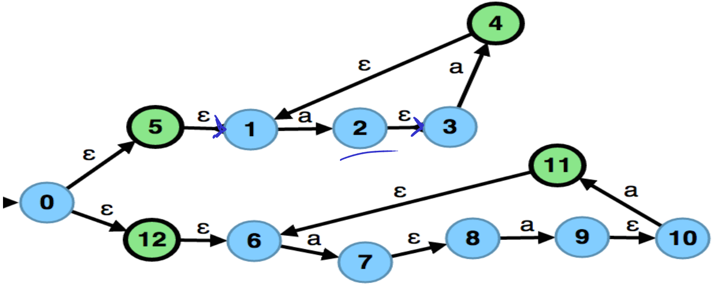
NFA to DFA Conversion



Possible states of the DFA: 2^Q

{ { } , { 0},...{12}, {0,1}, ...,{0,12}, ...{12, 12},
{0,1,2} ...., { 0,1,2...,12 } }

NFA to DFA Conversion



Epsilon Closure

–All states reachable from a state through epsilon

– q $\in E(q)$

– If $q_1 \in E(q)$ and $\delta(q_1, \epsilon, q_2)$ then $q_2 \in E(q)$

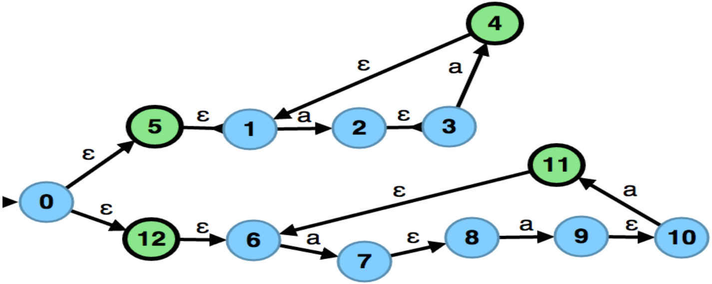
E(0) = {0,5,12}   E(1) = {1}   E(2) = {2,3}

NFA to DFA Conversion

$\delta' : (2^Q) \times \Sigma \to (2^Q)$

- 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 through Examle



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: Procedure

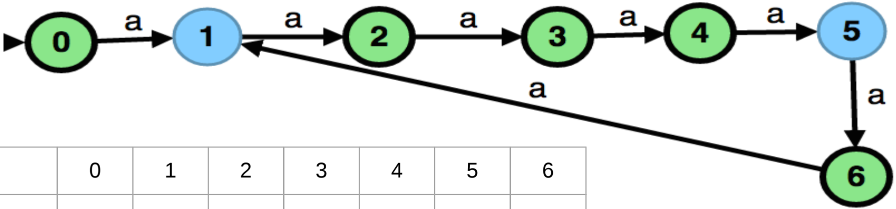
- Write down all pairs of state as a table
- Every cell in the table denotes whether the corresponding states are equivalent

	q1	q2	q3	q4	q5
q1	x	?	?	?	?
q2		x	?	?	?
q3			x	?	?
q4				x	?
q5					x

Minimizing DFAs: Procedure

- Inititalize cells (q1, q2) to false if one of them is final and other is non-final
- Make the cell (q1, q2) false, if q1 → q1’ on some alphabet symbol and q2 → q2’ on ‘a’ and q1’ and q2’ are not equivalent
- Iterate the above process until all non-equivalent states are found

Minimizing DFAs: Illustration



	0	1	2	3	4	5	6
0	x						
1		x					
2			x				
3				x			
4					x		
5						x	
6							x

Properties of Automata

Complement:

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

Note this does not work for NFA

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

$-A' = (\Sigma, Q_1 \times Q_2, (q_0^1, q_0^2), \delta', F_1 \times F_2)$

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

Emptiness of language, inclusion of one language into another, equivalence – they are all decidable