Scanner: Lexical Analysis

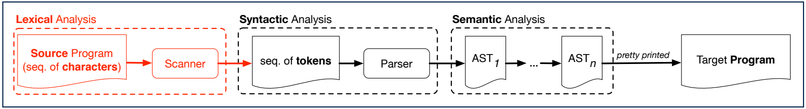
Readings: EAC2 Chapter 2

EECS4302 M:
Compilers and Interpreters
Winter 2020

CHEN-WEI WANG

Scanner in Context

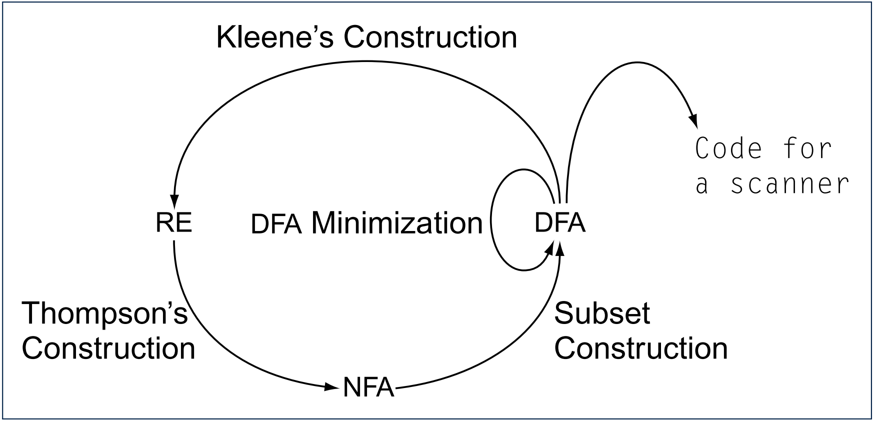
- Recall:



- Treats the input programas as a ***a sequence of characters***
- Applies rules ***recognizing*** character sequences as ***tokens***

  [***lexical*** analysis]
- Upon termination:
  - Reports character sequences not recognizable as tokens
  - Produces a ***a sequence of tokens***
- Only part of compiler touching ***every character*** in input program.
- Tokens *recognizable* by scanner constitute a *regular language*.

Scanner: Formulation & Implementation



Alphabets

- An *alphabet* is a *finite*, *nonempty* set of symbols.
  - The convention is to write Σ, possibly with a informative subscript, to denote the alphabet in question.
    e.g., $\Sigma_{eng} = \{a, b, \dots, z, A, B, \dots, Z\}$ [the English alphabet]
    e.g., $\Sigma_{bin} = \{0, 1\}$ [the binary alphabet]
    e.g., $\Sigma_{dec} = \{d \mid 0 \leq d \leq 9\}$ [the decimal alphabet]
    e.g., Σ_{key} [the keyboard alphabet]
- Use either a *set enumeration* or a *set comprehension* to define your own alphabet.

Strings (1)

- A **string** or a **word** is *finite* sequence of symbols chosen from some *alphabet*.

  e.g., Oxford is a string from the English alphabet Σ_{eng}
  e.g., 01010 is a string from the binary alphabet Σ_{bin}
  e.g., 01010.01 is *not* a string from Σ_{bin}
  e.g., 57 is a string from the binary alphabet Σ_{dec}
- It is not correct to say, e.g., $01010 \in \Sigma_{bin}$ [Why?]
- The **length** of a string w, denoted as $|w|$, is the number of characters it contains.
  - e.g., $|Oxford| = 6$
  - ϵ is the *empty string* ($|\epsilon| = 0$) that may be from any alphabet.
- Given two strings x and y, their **concatenation**, denoted as xy, is a new string formed by a copy of x followed by a copy of y.
  - e.g., Let $x = 01101$ and $y = 110$, then $xy = 01101110$
  - The empty string ϵ is the **identity for concatenation**:

    $\epsilon w = w = w\epsilon$ for any string w

Strings (2)

- Given an *alphabet* Σ, we write Σ^k, where $k \in \mathbb{N}$, to denote the *set of strings of length k from* Σ

$$\Sigma^k = \{w \mid w \texttt{ is from } \Sigma \wedge |w| = k\}$$

  - e.g., $\{0, 1\}^2 = \{00, 01, 10, 11\}$
  - Σ^0 is $\{\epsilon\}$ for any alphabet Σ
- Σ^+ is the set of *nonempty* strings from alphabet Σ

$$\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \ldots = \{w \mid w \in \Sigma^k \wedge k > 0\} = \bigcup_{k>0} \Sigma^k$$

- Σ^* is the set of strings of *all possible lengths* from alphabet Σ

$$\Sigma^* = \Sigma^+ \cup \{\epsilon\}$$

Review Exercises: Strings

1. What is $|\{a, b, \ldots, z\}^5|$?
2. Enumerate, in a systematic manner, the set $\{a, b, c\}^4$.
3. Explain the difference between Σ and Σ^1.

   Σ is a set of *symbols*; Σ^1 is a set of *strings* of length 1.
4. Prove or disprove: $\Sigma_1 \subseteq \Sigma_2 \Rightarrow \Sigma_1^* \subseteq \Sigma_2^*$

Languages

- *A language L over Σ* (where $|\Sigma|$ is finite) is a set of strings s.t.

$$L \subseteq \Sigma^*$$

- When useful, include an informative subscript to denote the *language L* in question.
  - e.g., The language of *valid* Java programs

$$L_{Java} = \{prog \mid prog \in \Sigma_{key}^* \land prog \texttt{ compiles in Eclipse}\}$$

  - e.g., The language of strings with n 0's followed by n 1's ($n \geq 0$)

$$\{\epsilon, 01, 0011, 000111, \dots\} = \{0^n 1^n \mid n \geq 0\}$$

  - e.g., The language of strings with an equal number of 0's and 1's

$$\begin{aligned} & \{\epsilon, 01, 10, 0011, 0101, 0110, 1100, 1010, 1001, \dots\} \\ = \ & \{w \mid \texttt{\# of 0's in } w = \texttt{\# of 1's in } w\} \end{aligned}$$

Review Exercises: Languages

1. Use set comprehensions to define the following languages. Be as *formal* as possible.
   - A language over $\{0, 1\}$ consisting of strings beginning with some 0's (possibly none) followed by at least as many 1's.
   - A language over $\{a, b, c\}$ consisting of strings beginning with some a's (possibly none), followed by some b's and then some c's, s.t. the # of a's is at least as many as the sum of #'s of b's and c's.
2. Explain the difference between the two languages $\{\epsilon\}$ and $\varnothing$.
3. Justify that Σ^*, $\varnothing$, and $\{\epsilon\}$ are all languages over Σ.
4. Prove or disprove: If L is a language over Σ, and $\Sigma_2 \supseteq \Sigma$, then L is also a language over Σ_2.
   Hint: Prove that $\Sigma \subseteq \Sigma_2 \wedge L \subseteq \Sigma^* \Rightarrow L \subseteq \Sigma_2^*$
5. Prove or disprove: If L is a language over Σ, and $\Sigma_2 \subseteq \Sigma$, then L is also a language over Σ_2.
   Hint: Prove that $\Sigma_2 \subseteq \Sigma \wedge L \subseteq \Sigma^* \Rightarrow L \subseteq \Sigma_2^*$

Problems

- Given a *language* L over some *alphabet* Σ, a **problem** is the *decision* on whether or not a given *string* w is a member of L.

$$w \in L$$

Is this equivalent to deciding $w \in \Sigma^*$? [*No*]

- e.g., The Java compiler solves the problem of *deciding* if the *string of symbols* typed in the Eclipse editor is a *member* of L_{Java} (i.e., set of Java programs with no syntax and type errors).

Regular Expressions (RE): Introduction

- *Regular expressions* (RegExp's) are:
  - A type of *language-defining* notation
    - This is *similar* to the equally-expressive *DFA*, *NFA*, and ϵ-*NFA*.
  - *Textual* and look just like a programming language
    - e.g., 01* + 10* denotes $L = \{0x \mid x \in \{1\}^*\} \cup \{1x \mid x \in \{0\}^*\}$
    - e.g., (0*10*10*)*10* denotes $L = \{w \mid w \texttt{ has odd \# of } 1\texttt{'s}\}$
    - This is *dissimilar* to the diagrammatic *DFA*, *NFA*, and ϵ-*NFA*.
    - RegExp's can be considered as a "user-friendly" alternative to *NFA* for describing software components. [e.g., text search]
    - Writing a RegExp is like writing an algebraic expression, using the defined operators, e.g., ((4 + 3) * 5) % 6
- Despite the programming convenience they provide, RegExp's, *DFA*, *NFA*, and ϵ-*NFA* are all *provably equivalent*.
  - They are capable of defining *all* and *only* regular languages.

RE: Language Operations (1)

- Given Σ of input alphabets, the simplest RegExp is $s \in \Sigma^1$.
  - e.g., Given $\Sigma = \{a, b, c\}$, expression a denotes the language consisting of a single string a.
- Given two languages $L, M \in \Sigma^*$, there are 3 operators for building a *larger language* out of them:

1. **Union**

$$L \cup M = \{w \mid w \in L \vee w \in M\}$$

   In the textual form, we write $+$ for union.

2. **Concatenation**

$$LM = \{xy \mid x \in L \wedge y \in M\}$$

   In the textual form, we write either `.` or nothing at all for concatenation.

RE: Language Operations (2)

3. **Kleene Closure** (or **Kleene Star**)

$$L^* = \bigcup_{i \geq 0} L^i$$

where

$$\begin{array}{lcl} L^0 & = & \{\epsilon\} \\ L^1 & = & L \\ L^2 & = & \{x_1 x_2 \mid x_1 \in L \land x_2 \in L\} \\ \dots & & \\ L^i & = & \{ \underbrace{x_1 x_2 \dots x_i}_{i \texttt{ repetations}} \mid x_j \in L \land 1 \leq j \leq i\} \\ \dots & & \end{array}$$

In the textual form, we write $\star$ for closure.

Question: What is $|L^i|$ ($i \in \mathbb{N}$)? [$|L|^i$]

Question: Given that $L = \{0\}^*$, what is L^*? [L]

RE: Construction (1)

We may build *regular expressions* *recursively*:

- Each (*basic* or *recursive*) form of regular expressions denotes a language (i.e., a set of strings that it accepts).
- **Base Case**:
  - Constants ϵ and $\varnothing$ are regular expressions.

$$\begin{array}{lcl} L(\ \epsilon\ ) & = & \{\epsilon\} \\ L(\ \varnothing\ ) & = & \varnothing \end{array}$$

  - An input symbol $a \in \Sigma$ is a regular expression.

$$L(\ a\ ) = \{a\}$$

    If we want a regular expression for the language consisting of only the string $w \in \Sigma^*$, we write w as the regular expression.
  - Variables such as L, M, *etc.*, might also denote languages.

RE: Construction (2)

- **Recursive Case** Given that E and F are regular expressions:
  - The union $E + F$ is a regular expression.

$$L(\ E + F\ ) = L(E) \cup L(F)$$

  - The concatenation EF is a regular expression.

$$L(\ EF\ ) = L(E)L(F)$$

  - Kleene closure of E is a regular expression.

$$L(\ E^*\ ) = (L(E))^*$$

  - A parenthesized E is a regular expression.

$$L(\ (E)\ ) = L(E)$$

RE: Construction (3)

Exercises:

- $\varnothing L$ $\qquad [\ \varnothing L = \varnothing = L\varnothing\ ]$
- $\varnothing^*$

$$
\begin{aligned}
\varnothing^* &= \varnothing^0 \cup \varnothing^1 \cup \varnothing^2 \cup \ldots \\
&= \{\epsilon\} \cup \varnothing \cup \varnothing \cup \ldots \\
&= \{\epsilon\}
\end{aligned}
$$

- $\varnothing^* L$ $\qquad [\ \varnothing^* L = L = L\varnothing^*\ ]$
- $\varnothing + L$ $\qquad [\ \varnothing + L = L = \varnothing + L\ ]$

RE: Construction (4)

Write a regular expression for the following language

$$\{ w \mid w \texttt{ has alternating } 0'\texttt{s and } 1'\texttt{s} \}$$

- Would $(01)^*$ work? [alternating 10's?]
- Would $(01)^* + (10)^*$ work? [starting and ending with 1?]
- $0(10)^* + (01)^* + (10)^* + 1(01)^*$
- It seems that:
  - 1st and 3rd terms have $(10)^*$ as the common factor.
  - 2nd and 4th terms have $(01)^*$ as the common factor.
- Can we simplify the above regular expression?
- $(\epsilon + 0)(10)^* + (\epsilon + 1)(01)^*$

RE: Review Exercises

Write the regular expressions to describe the following languages:

- $\{ w \mid w$ ends with $01 \}$
- $\{ w \mid w$ contains 01 as a substring $\}$
- $\{ w \mid w$ contains no more than three consecutive $1'$s $\}$
- $\{ w \mid w$ ends with $01 \lor w$ has an odd # of $0'$s $\}$
-

$$\left\{ sx.y \;\middle|\; \begin{array}{ll} & s \in \{+, -, \epsilon\} \\ \land & x \in \Sigma_{dec}^{*} \\ \land & y \in \Sigma_{dec}^{*} \\ \land & \neg(x = \epsilon \land y = \epsilon) \end{array} \right\}$$

-

$$\left\{ xy \;\middle|\; \begin{array}{ll} & x \in \{0,1\}^{*} \land y \in \{0,1\}^{*} \\ \land & x \text{ has alternating } 0'\text{s and } 1'\text{s} \\ \land & y \text{ has an odd \# } 0'\text{s and an odd \# } 1'\text{s} \end{array} \right\}$$

RE: Operator Precedence

- In an order of *decreasing precedence*:
  - Kleene star operator
  - Concatenation operator
  - Union operator
- When necessary, use *parentheses* to force the intended order of evaluation.
- e.g.,
  - 10^* vs. $(10)^*$ [10^* is equivalent to $1(0^*)$]
  - $01^* + 1$ vs. $0(1^* + 1)$ [$01^* + 1$ is equivalent to $(0(1^*)) + (1)$]
  - $0 + 1^*$ vs. $(0 + 1)^*$ [$0 + 1^*$ is equivalent to $(0) + (1^*)$]

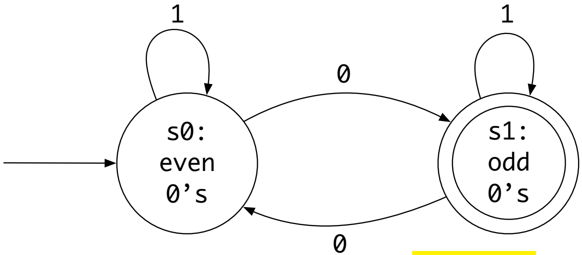
DFA: Deterministic Finite Automata (1.1)

- A *deterministic finite automata (DFA)* is a *finite state machine* (*FSM*) that *accepts* (or recognizes) a pattern of behaviour.
  - For our purpose of this course, we study patterns of *strings* (i.e., how *alphabet symbols* are ordered).
  - Unless otherwise specified, we consider strings in $\{0, 1\}^*$
  - Each pattern contains the set of satisfying strings.
  - We describe the patterns of strings using *set comprehensions*:
    - $\{ w \mid w \texttt{ has an odd number of 0's} \}$
    - $\{ w \mid w \texttt{ has an even number of 1's} \}$
    - $\left\{ w \mid \begin{array}{l} w \neq \epsilon \\ \land \quad w \texttt{ has equal \# of alternating 0's and 1's} \end{array} \right\}$
    - $\{ w \mid w \texttt{ contains } 01 \texttt{ as a substring} \}$
    - $\left\{ w \mid \begin{array}{l} w \texttt{ has an even number of 0's} \\ \land \quad w \texttt{ has an odd number of 1's} \end{array} \right\}$
- Given a pattern description, we design a DFA that accepts it.
  - The resulting DFA can be transformed into an executable program.

DFA: Deterministic Finite Automata (1.2)

The *transition diagram* below defines a DFA which *accepts* exactly the language

$$\{ w \mid w \texttt{ has an odd number of 0's} \}$$

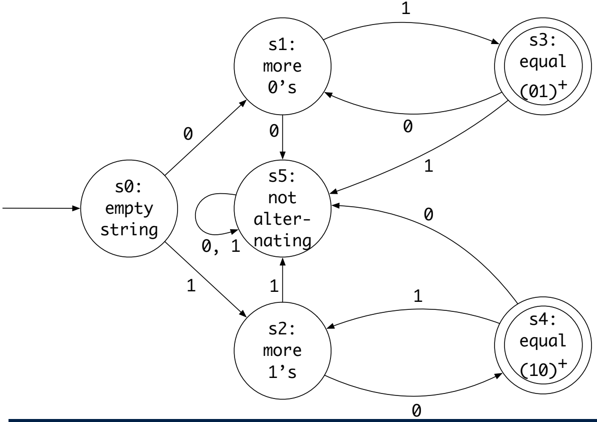


- Each *incoming* or *outgoing* arc (called a *transition*) corresponds to an input alphabet symbol.
- s_0 with an unlabelled *incoming* transition is the *start state*.
- s_3 drawn as a double circle is a *final state*.
- All states have *outgoing* transitions covering $\{0, 1\}$.

DFA: Deterministic Finite Automata (1.3)

The *transition diagram* below defines a DFA which *accepts* exactly the language

$$\left\{ w \;\middle|\; \begin{array}{l} w \neq \epsilon \\ \wedge \;\; w \texttt{ has equal \# of alternating 0's and 1's} \end{array} \right\}$$



Review Exercises: Drawing DFAs

Draw the transition diagrams for DFAs which accept other example string patterns:

- $\{ w \mid w \texttt{ has an even number of 1's} \}$
- $\{ w \mid w \texttt{ contains } 01 \texttt{ as a substring} \}$
- $\left\{ w \mid \begin{array}{l} w \texttt{ has an even number of 0's} \\ \wedge \; w \texttt{ has an odd number of 1's} \end{array} \right\}$

DFA: Deterministic Finite Automata (2.1)

A *deterministic finite automata (DFA)* is a 5-tuple

$$M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$$

- Q is a finite set of *states*.
- Σ is a finite set of *input symbols* (i.e., the *alphabet*).
- $\delta : (Q \times \Sigma) \rightarrow Q$ is a *transition function*

  δ takes as arguments a state and an input symbol and returns a state.
- $q_0 \in Q$ is the *start state*.
- $F \subseteq Q$ is a set of *final* or *accepting states*.

DFA: Deterministic Finite Automata (2.2)

- Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$:
  - We write ***L(M)*** to denote the ***language of M***: the set of strings that M *accepts*.
  - A string is *accepted* if it results in a sequence of transitions: beginning from the *start* state and ending in a *final* state.

$$L(M) = \left\{ \begin{array}{l} a_1 a_2 \dots a_n \mid \\ \quad 1 \le i \le n \land a_i \in \Sigma \land \delta(q_{i-1}, a_i) = q_i \land q_n \in F \end{array} \right\}$$

  - M *rejects* any string $w \notin L(M)$.
- We may also consider ***L(M)*** as *concatenations of labels* from the set of all valid *paths* of *M*'s transition diagram; each such path starts with q_0 and ends in a state in F.

DFA: Deterministic Finite Automata (2.3)

- Given a *DFA* $M = (Q, \Sigma, \delta, q_0, F)$, we may simplify the definition of *L(M)* by extending δ (which takes an input symbol) to $\hat{\delta}$ (which takes an input string).

$$\hat{\delta} : (Q \times \Sigma^*) \rightarrow Q$$

  We may define $\hat{\delta}$ recursively, using δ!

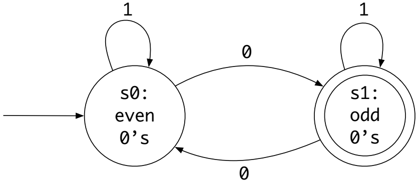
$$\begin{array}{lll} \hat{\delta}(q, \epsilon) & = & q \\ \hat{\delta}(q, xa) & = & \delta(\ \hat{\delta}(q, x), a\ ) \end{array}$$

  where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$
- A neater definition of $L(M)$: the set of strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ is an *accepting state*.

$$L(M) = \{w \mid w \in \Sigma^* \wedge \hat{\delta}(q_0, w) \in F\}$$

- A language L is said to be a *regular language*, if there is some *DFA* M such that $L = L(M)$.

DFA: Deterministic Finite Automata (2.4)



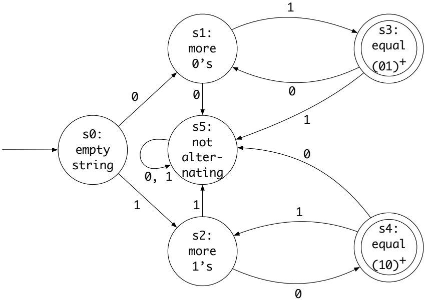
We formalize the above DFA as $M = (Q, \Sigma, \delta, q_0, F)$, where

- $Q = \{s_0, s_1\}$
- $\Sigma = \{0, 1\}$
- $\delta = \{((s_0, 0), s_1), ((s_0, 1), s_0), ((s_1, 0), s_0), ((s_1, 1), s_1)\}$

state \ input	0	1
s_0	s_1	s_0
s_1	s_0	s_1

- $q_0 = s_0$
- $F = \{s_1\}$

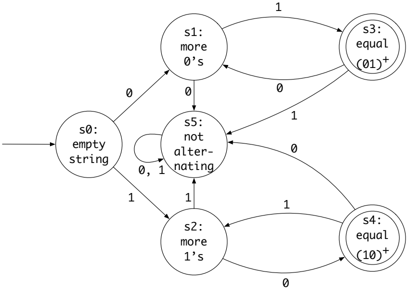
DFA: Deterministic Finite Automata (2.5.1)



We formalize the above DFA as $M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$, where

- $Q = \{s_0, s_1, s_2, s_3, s_4, s_5\}$
- $\Sigma = \{0, 1\}$
- $q_0 = s_0$
- $F = \{s_3, s_4\}$

DFA: Deterministic Finite Automata (2.5.2)



- $\delta =$

state \ input	0	1
s_0	s_1	s_2
s_1	s_5	s_3
s_2	s_4	s_5
s_3	s_1	s_5
s_4	s_5	s_2
s_5	s_5	s_5

Review Exercises: Formalizing DFAs

Formalize DFAs (as 5-tuples) for the other example string patterns mentioned:

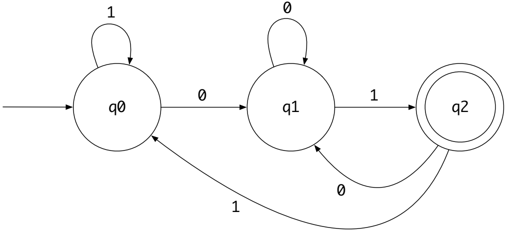
- $\{ w \mid w \texttt{ has an even number of 0's} \}$
- $\{ w \mid w \texttt{ contains } 01 \texttt{ as a substring} \}$
- $\left\{ w \;\middle|\; \begin{array}{l} \quad w \texttt{ has an even number of 0's} \\ \wedge\ w \texttt{ has an odd number of 1's} \end{array} \right\}$

NFA: Nondeterministic Finite Automata (1.1)

Problem: Design a DFA that accepts the following language:

$$L = \{ \; x01 \mid x \in \{0, 1\}^* \; \}$$

That is, L is the set of strings of 0s and 1s ending with 01.



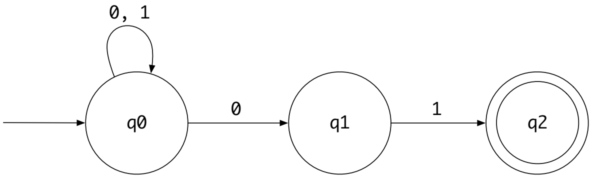
Given an input string w, we may simplify the above DFA by:

- *nondeterministically* treating state q_0 as both:
  - a state *ready* to read the last two input symbols from w
  - a state *not yet ready* to read the last two input symbols from w
- substantially reducing the outgoing transitions from q_1 and q_2

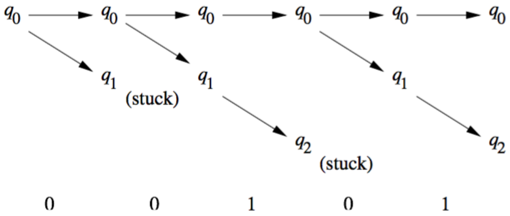
Compare the above DFA with the DFA in slide 39.

NFA: Nondeterministic Finite Automata (1.2)

- A *non-deterministic finite automata (NFA)* that accepts the same language:



- How an NFA determines if an input *00101* should be processed:



NFA: Nondeterministic Finite Automata (2)

- A *nondeterministic finite automata (NFA)*, like a *DFA*, is a *FSM* that *accepts* (or recognizes) a pattern of behaviour.
- An NFA being *nondeterministic* means that from a given state, the *same input label* might corresponds to *multiple transitions* that lead to *distinct states*.
  - Each such transition offers an *alternative path*.
  - Each alternative path is explored independently and in parallel.
  - If **there exists** an alternative path that *succeeds* in processing the input string, then we say the NFA *accepts* that input string.
  - If **all** alternative paths get stuck at some point and *fail* to process the input string, then we say the NFA *rejects* that input string.
- NFAs are often more succinct (i.e., fewer states) and easier to design than DFAs.
- However, NFAs are just as *expressive* as are DFAs.
  - We can **always** convert an NFA to a DFA.

NFA: Nondeterministic Finite Automata (3.1)

- A *nondeterministic finite automata (NFA)* is a 5-tuple

$$M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$$

  - Q is a finite set of *states*.
  - Σ is a finite set of *input symbols* (i.e., the *alphabet*).
  - $\delta : (Q \times \Sigma) \rightarrow \mathbb{P}(Q)$ is a *transition function*

    δ takes as arguments a state and an input symbol and returns a set of states.
  - $q_0 \in Q$ is the *start state*.
  - $F \subseteq Q$ is a set of *final* or *accepting states*.
- What is the difference between a *DFA* and an *NFA*?
  - The transition function δ of a *DFA* returns a *single* state.
  - The transition function δ of an *NFA* returns a *set* of states.

NFA: Nondeterministic Finite Automata (3.2)

- Given a *NFA* $M = (Q, \Sigma, \delta, q_0, F)$, we may simplify the definition of *L(M)* by extending δ (which takes an input symbol) to $\hat{\delta}$ (which takes an input string).

$$\hat{\delta} : (Q \times \Sigma^*) \to \mathbb{P}(Q)$$

We may define $\hat{\delta}$ recursively, using δ!

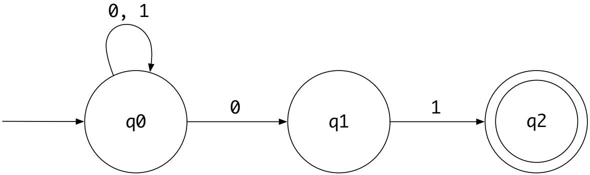
$$\begin{array}{lcl} \hat{\delta}(q, \epsilon) & = & \{q\} \\ \hat{\delta}(q, xa) & = & \bigcup\{\delta(q', a) \mid q' \in \hat{\delta}(q, x)\} \end{array}$$

where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$

- A neater definition of $L(M)$: the set of strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ contains **at least one** *accepting state*.

$$L(M) = \{w \mid w \in \Sigma^* \wedge \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$$

NFA: Nondeterministic Finite Automata (4)



Given an input string 00101:

- **Read 0**: $\delta(q_0, 0) = \{ q_0, q_1 \}$
- **Read 0**: $\delta(q_0, 0) \cup \delta(q_1, 0) = \{ q_0, q_1 \} \cup \varnothing = \{ q_0, q_1 \}$
- **Read 1**: $\delta(q_0, 1) \cup \delta(q_1, 1) = \{ q_0 \} \cup \{ q_2 \} = \{ q_0, q_2 \}$
- **Read 0**: $\delta(q_0, 0) \cup \delta(q_2, 0) = \{ q_0, q_1 \} \cup \varnothing = \{ q_0, q_1 \}$
- **Read 1**: $\delta(q_0, 1) \cup \delta(q_1, 1) = \{ q_0, q_1 \} \cup \{ q_2 \} = \{ q_0, q_1, q_2 \}$

$\because \{ q_0, q_1, q_2 \} \cap \{ q_2 \} \neq \varnothing \therefore$ `00101 is` *accepted*

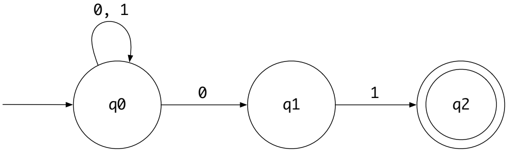
DFA ≡ NFA (1)

- For many languages, constructing an accepting *NFA* is easier than a *DFA*.
- From each state of an *NFA*:
  - Outgoing transitions need **not** cover the entire Σ.
  - An input symbol may *non-deterministically* lead to multiple states.
- In practice:
  - An *NFA* has just as many states as its equivalent *DFA* does.
  - An *NFA* often has fewer transitions than its equivalent *DFA* does.
- In the worst case:
  - While an *NFA* has n states, its equivalent *DFA* has 2^n states.
- Nonetheless, an *NFA* is still just as *expressive* as a *DFA*.
  - Every language accepted by some *NFA* can also be accepted by some *DFA*.

$$\forall N : NFA \bullet (\exists D : DFA \bullet L(D) = L(N))$$

DFA ≡ NFA (2.2): Lazy Evaluation (1)

Given an *NFA*:



Subset construction (with *lazy evaluation*) produces a *DFA* transition table:

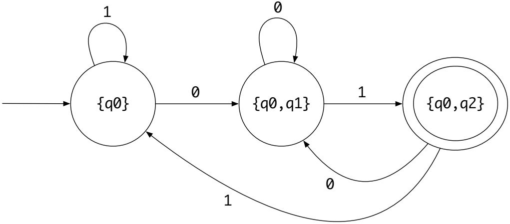
state \ input	0	1
$\{q_0\}$	$\delta(q_0, 0)$ $= \{q_0, q_1\}$	$\delta(q_0, 1)$ $= \{q_0\}$
$\{q_0, q_1\}$	$\delta(q_0, 0) \cup \delta(q_1, 0)$ $= \{q_0, q_1\} \cup \varnothing$ $= \{q_0, q_1\}$	$\delta(q_0, 1) \cup \delta(q_1, 1)$ $= \{q_0\} \cup \{q_2\}$ $= \{q_0, q_2\}$
$\{q_0, q_2\}$	$\delta(q_0, 0) \cup \delta(q_2, 0)$ $= \{q_0, q_1\} \cup \varnothing$ $= \{q_0, q_1\}$	$\delta(q_0, 1) \cup \delta(q_2, 1)$ $= \{q_0\} \cup \varnothing$ $= \{q_0\}$

DFA ≡ NFA (2.2): Lazy Evaluation (2)

Applying *subset construction* (with *lazy evaluation*), we arrive in a *DFA* transition table:

state \ input	0	1
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

We then draw the *DFA* accordingly:



Compare the above DFA with the DFA in slide 31.

DFA ≡ NFA (2.2): Lazy Evaluation (3)

- Given an *NFA* $N = (Q_N, \Sigma_N, \delta_N, q_0, F_N)$, often only a small portion of the $|\mathbb{P}(Q_N)|$ subset states is *reachable* from $\{q_0\}$.

```
ALGORITHM: ReachableSubsetStates
  INPUT: q0 : Q_N      ;      OUTPUT: Reachable ⊆ ℙ(Q_N)
PROCEDURE:
  Reachable   := { {q0} }
  ToDiscover  := { {q0} }
  while (ToDiscover ≠ ∅) {
    choose S : ℙ(Q_N) such that S ∈ ToDiscover
    remove S from ToDiscover
    NotYetDiscovered :=
        ( {δ_N(s,0) | s ∈ S} ∪ {δ_N(s,1) | s ∈ S} ) \ Reachable
    Reachable  := Reachable ∪ NotYetDiscovered
    ToDiscover := ToDiscover ∪ NotYetDiscovered
  }
  return Reachable
```

- RT of *ReachableSubsetStates*? [$O(2^{|Q_N|})$]

ϵ-NFA: Examples (1)

Draw the NFA for the following two languages:

1.

$$\left\{ xy \;\middle|\; \begin{array}{ll} & x \in \{0,1\}^* \\ \wedge & y \in \{0,1\}^* \\ \wedge & x \texttt{ has alternating } 0\texttt{'s and } 1\texttt{'s} \\ \wedge & y \texttt{ has an odd \# } 0\texttt{'s and an odd \# } 1\texttt{'s} \end{array} \right\}$$

2.

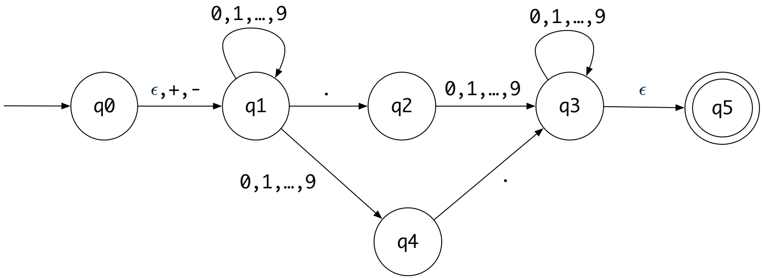
$$\left\{ w : \{0,1\}^* \;\middle|\; \begin{array}{ll} & w \texttt{ has alternating } 0\texttt{'s and } 1\texttt{'s} \\ \vee & w \texttt{ has an odd \# } 0\texttt{'s and an odd \# } 1\texttt{'s} \end{array} \right\}$$

3.

$$\left\{ sx.y \;\middle|\; \begin{array}{ll} & s \in \{+, -, \epsilon\} \\ \wedge & x \in \Sigma_{dec}^* \\ \wedge & y \in \Sigma_{dec}^* \\ \wedge & \neg(x = \epsilon \wedge y = \epsilon) \end{array} \right\}$$

ϵ-NFA: Examples (2)

$$\left\{ sx.y \;\middle|\; \begin{array}{ll} & s \in \{+, -, \epsilon\} \\ \wedge & x \in \Sigma_{dec}^{*} \\ \wedge & y \in \Sigma_{dec}^{*} \\ \wedge & \neg(x = \epsilon \wedge y = \epsilon) \end{array} \right\}$$



From q_0 to q_1, reading a sign is **optional**: a *plus* or a *minus*, or *nothing at all* (i.e., ϵ).

ϵ-NFA: Formalization (1)

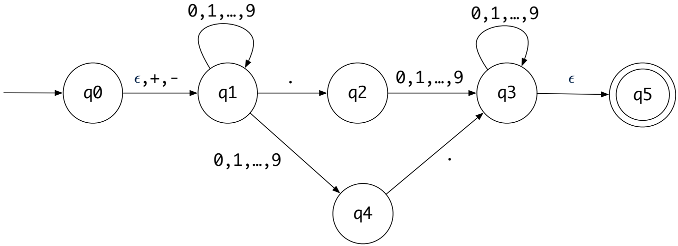
An *ϵ-NFA* is a 5-tuple

$$M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$$

- Q is a finite set of *states*.
- Σ is a finite set of *input symbols* (i.e., the *alphabet*).
- $\delta : (Q \times (\Sigma \cup \{\epsilon\})) \to \mathbb{P}(Q)$ is a *transition function*

  δ takes as arguments a state and an input symbol, or *an empty string* ϵ, and returns a set of states.
- $q_0 \in Q$ is the *start state*.
- $F \subseteq Q$ is a set of *final* or *accepting states*.

ϵ-NFA: Formalization (2)



Draw a transition table for the above NFA's δ function:

	ϵ	+, -	.	$0 .. 9$
q_0	$\{q_1\}$	$\{q_1\}$	$\varnothing$	$\varnothing$
q_1	$\varnothing$	$\varnothing$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_3	$\{q_5\}$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_4	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$
q_5	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

ϵ-NFA: Epsilon-Closures (1)

- Given *ϵ-NFA N*

$$N = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$$

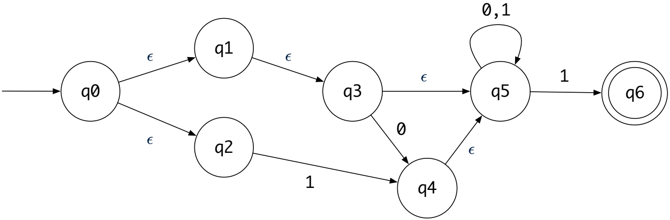
we define the *epsilon closure* (or *ϵ-closure*) as a function

$$\texttt{ECLOSE} : Q \to \mathbb{P}(Q)$$

- For any state $q \in Q$

$$\texttt{ECLOSE}(q) = \{q\} \cup \bigcup_{p \in \delta(q,\epsilon)} \texttt{ECLOSE}(p)$$

ϵ-NFA: Epsilon-Closures (2)



$$
\begin{aligned}
& \text{ECLOSE}(q_0) \\
= \; & \{\delta(q_0, \epsilon) = \{q_1, q_2\}\} \\
& \{q_0\} \cup \text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_2) \\
= \; & \{\mathit{ECLOSE}(q_1),\; \delta(q_1, \epsilon) = \{q_3\},\; \mathit{ECLOSE}(q_2),\; \delta(q_2, \epsilon) = \varnothing\} \\
& \{q_0\} \cup (\; \{q_1\} \cup \mathit{ECLOSE}(q_3) \;) \cup (\; \{q_2\} \cup \varnothing \;) \\
= \; & \{\mathit{ECLOSE}(q_3),\; \delta(q_3, \epsilon) = \{q_5\}\} \\
& \{q_0\} \cup (\; \{q_1\} \cup (\; \{q_3\} \cup \mathit{ECLOSE}(q_5) \;) \;) \cup (\; \{q_2\} \cup \varnothing \;) \\
= \; & \{\mathit{ECLOSE}(q_5),\; \delta(q_5, \epsilon) = \varnothing\} \\
& \{q_0\} \cup (\; \{q_1\} \cup (\; \{q_3\} \cup (\; \{q_5\} \cup \varnothing \;) \;) \;) \cup (\; \{q_2\} \cup \varnothing \;)
\end{aligned}
$$

ϵ-NFA: Formalization (3)

- Given a ϵ-*NFA* $M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$, we may simplify the definition of *L(M)* by extending δ (which takes an input symbol) to $\hat{\delta}$ (which takes an input string).

$$\hat{\delta} : (Q \times \Sigma^*) \rightarrow \mathbb{P}(Q)$$

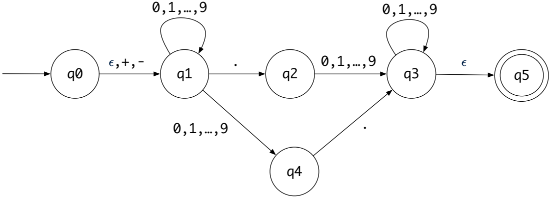
  We may define $\hat{\delta}$ recursively, using δ!

$$\begin{array}{lcl} \hat{\delta}(q, \epsilon) & = & \texttt{ECLOSE}(q) \\ \hat{\delta}(q, xa) & = & \bigcup\{\ \texttt{ECLOSE}(q'') \mid q'' \in \delta(q', a) \land q' \in \hat{\delta}(q, x)\ \} \end{array}$$

  where $q \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$
- Then we define $L(M)$ as the set of strings $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w)$ contains **at least one** *accepting state*.

$$L(M) = \{w \mid w \in \Sigma^* \land \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$$

ϵ-NFA: Formalization (4)



Given an input string 5.6:

$\hat{\delta}(q_0, \epsilon) = \text{ECLOSE}(q_0) = \{q_0, q_1\}$

- **Read 5**: $\delta(q_0, 5) \cup \delta(q_1, 5) = \varnothing \cup \{q_1, q_4\} = \{\ q_1, q_4\ \}$

  $\hat{\delta}(q_0, 5) = \text{ECLOSE}(q_1) \cup \text{ECLOSE}(q_4) = \{q_1\} \cup \{q_4\} = \{q_1, q_4\}$
- **Read .**: $\delta(q_1, .) \cup \delta(q_4, .) = \{q_2\} \cup \{q_3\} = \{\ q_2, q_3\ \}$

  $\hat{\delta}(q_0, 5.) = \text{ECLOSE}(q_2) \cup \text{ECLOSE}(q_3) = \{q_2\} \cup \{q_3, q_5\} = \{q_2, q_3, q_5\}$
- **Read 6**: $\delta(q_2, 6) \cup \delta(q_3, 6) \cup \delta(q_5, 6) = \{q_3\} \cup \{q_3\} \cup \varnothing = \{\ q_3\ \}$

  $\hat{\delta}(q_0, 5.6) = \text{ECLOSE}(q_3) = \{q_3, q_5\}$ [5.6 is *accepted*]

DFA $\equiv$ ϵ-NFA: Subset Construction (1)

Subset construction (with *lazy evaluation* and *epsilon closures*) produces a *DFA* transition table.

	$d \in 0..9$	$s \in \{+, -\}$	.
$\{q_0, q_1\}$	$\{q_1, q_4\}$	$\{q_1\}$	$\{q_2\}$
$\{q_1, q_4\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2, q_3, q_5\}$
$\{q_1\}$	$\{q_1, q_4\}$	$\varnothing$	$\{q_2\}$
$\{q_2\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_2, q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$
$\{q_3, q_5\}$	$\{q_3, q_5\}$	$\varnothing$	$\varnothing$

For example, $\delta(\{q_0, q_1\}, d)$ is calculated as follows: $[d \in 0..9]$

$$\begin{aligned}
& \bigcup\{\texttt{ECLOSE}(q) \mid q \in \delta(q_0, d) \cup \delta(q_1, d)\} \\
= \ & \bigcup\{\texttt{ECLOSE}(q) \mid q \in \varnothing \cup \{q_1, q_4\}\} \\
= \ & \bigcup\{\texttt{ECLOSE}(q) \mid q \in \{q_1, q_4\}\} \\
= \ & \texttt{ECLOSE}(q_1) \cup \texttt{ECLOSE}(q_4) \\
= \ & \{q_1\} \cup \{q_4\} \\
= \ & \{q_1, q_4\}
\end{aligned}$$

DFA $\equiv$ ϵ-NFA: Subset Construction (2)

- Given an ϵ=*NFA* $N = (Q_N, \Sigma_N, \delta_N, q_0, F_N)$, by applying the *extended* subset construction to it, the resulting *DFA* $D = (Q_D, \Sigma_D, \delta_D, q_{D_{start}}, F_D)$ is such that:

$$\begin{array}{lcl} \Sigma_D & = & \Sigma_N \\ Q_D & = & \{ S \mid S \subseteq Q_N \wedge (\exists w : \Sigma^* \bullet S = \hat{\delta}_D(q_0, w)) \} \\ q_{D_{start}} & = & \text{ECLOSE}(q_0) \\ F_D & = & \{ S \mid S \subseteq Q_N \wedge S \cap F_N \neq \varnothing \} \\ \delta_D(S, a) & = & \bigcup \{ \text{ECLOSE}(s') \mid s \in S \wedge s' \in \delta_N(s, a) \} \end{array}$$

Regular Expression to ϵ-NFA

- Just as we construct each complex *regular expression* recursively, we define its equivalent ϵ-*NFA* *recursively*.
- Given a regular expression R, we construct an ϵ-NFA E, such that $L(R) = L(E)$, with
  - Exactly **one** accept state.
  - No incoming arc to the start state.
  - No outgoing arc from the accept state.

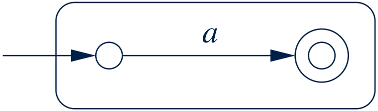
Regular Expression to ϵ-NFA

Base Cases:

- ϵ

- $\varnothing$

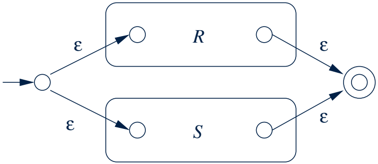
- a $[a \in \Sigma]$



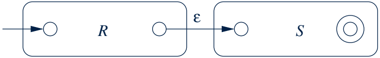
Regular Expression to ϵ-NFA

Recursive Cases: [*R* and *S* are RE's]

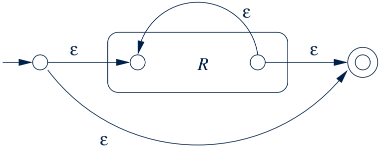
- $R + S$



- RS

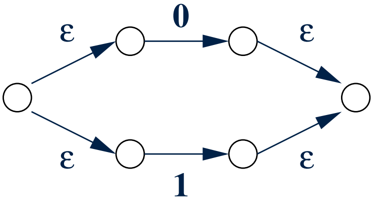


- R^*

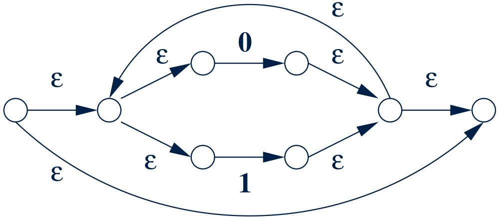


Regular Expression to ϵ-NFA: Examples (1.1)

- $0 + 1$

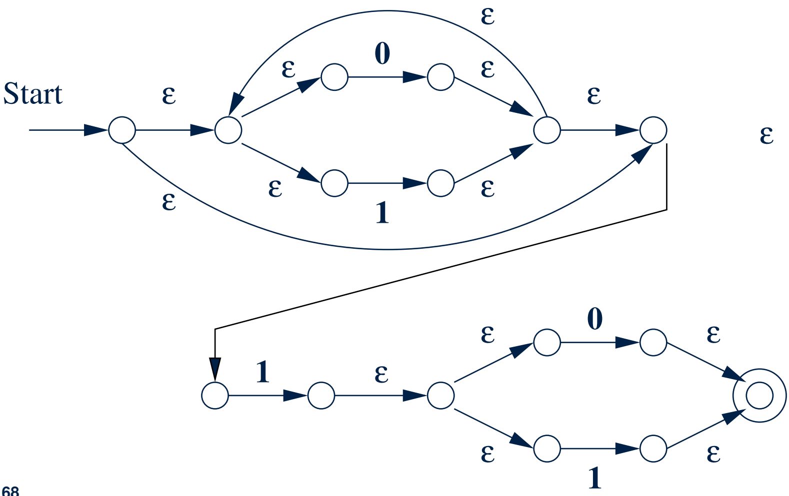


- $(0 + 1)^*$



Regular Expression to ϵ-NFA: Examples (1.2)

- $(0+1)^*1(0+1)$



Minimizing DFA: Motivation

- Recall: Regular Expresion $\longrightarrow$ ϵ-NFA $\longrightarrow$ DFA
- DFA produced by the ***subset construction*** (with ***lazy evaluation***) may **not** be ***minimum*** on its size of state.
- When the required size of memory is sensitive
  (e.g., processor's cache memory),
  the fewer number of DFA states, the better.

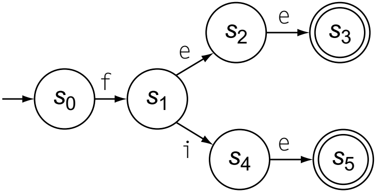
Minimizing DFA: Algorithm

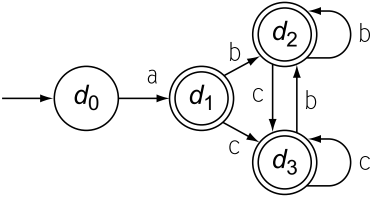
```
ALGORITHM: MinimizeDFAStates
  INPUT: DFA M = (Q, Σ, δ, q0, F)
  OUTPUT: M′ s.t. minimum |Q| and equivalent behaviour as M
PROCEDURE:
  P := ∅ /* refined partition so far */
  T := { F, Q − F } /* last refined partition */
  while (P ≠ T):
    P := T
    T := ∅
    for(p ∈ P s.t. |p| > 1):
      find the maximal S ⊆ p s.t. splittable(p, S)
      if S ≠ ∅ then
        T := T ∪ {S, p − S}
      else
        T := T ∪ {p}
      end
```

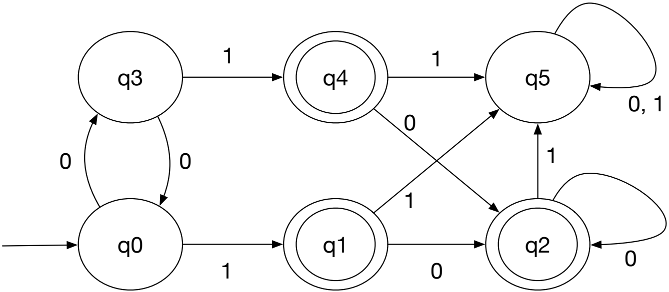
splittable(p, S) holds iff there is $c \in \Sigma$ s.t.

- Transition c leads all $s \in S$ to states in the ***same partition*** $p1$.
- Transition c leads some $s \in p - S$ to a ***different partition*** $p2$ ($p2 \neq p1$).

Minimizing DFA: Examples







Exercises: Minimize the DFA from **here**; Q1 & Q2, p59, EAC2.

Exercise:
Regular Expression to Minimized DFA

Given regular expression `r[0..9]+` which specifies the pattern of a register name, derive the equivalent DFA with the minimum number of states. Show all steps.

Implementing DFA as Scanner

- The source language has a list of ***syntactic categories***:

  e.g., keyword `while` [`while`]

  e.g., identifiers [`[a-zA-Z][a-zA-Z0-9_]*`]

  e.g., white spaces [`[ \t\r]+`]

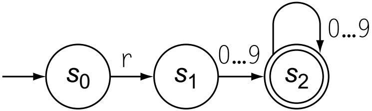
- A compiler's *scanner* must recognize ***words*** from **all** syntactic categories of the source language.
  - Each syntactic category is specified via a ***regular expression***.

$$\underbrace{r_1}_{\texttt{syn. cat. } 1} + \underbrace{r_1}_{\texttt{syn. cat. } 2} + \ldots + \underbrace{r_n}_{\texttt{syn. cat. } n}$$

  - Overall, a scanner should be implemented based on the *minimized DFA* accommodating all syntactic categories.
- Principles of a scanner:
  - Returns one ***word*** at a time
  - Each returned word is the ***longest possible*** that matches a ***pattern***
  - A **priority** may be specified among patterns (e.g., `new` is a keyword, not identifier)

Implementing DFA: Table-Driven Scanner (1)

- Consider the ***syntactic category*** of `register` names.
- Specified as a *regular expression*: `r[0..9]+`
- Afer conversion to ϵ-NFA, then to DFA, then to ***minimized DFA***:



- The following tables encode knowledge about the above DFA:

Classifier (`CharCat`)

r	0, 1, 2, ..., 9	EOF	**Other**
Register	*Digit*	*Other*	*Other*

Transition (δ)

	Register	***Digit***	***Other***
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

Token Type (`Type`)

s_0	s_1	s_2	s_e
invalid	*invalid*	*register*	*invalid*

Implementing DFA: Table-Driven Scanner (2)

The scanner then is implemented via a 4-stage skeleton:

```
NextWord()
  -- Stage 1:  Initialization
  state := s0 ; word := ϵ
  initialize an empty stack s ; s.push(bad)
  -- Stage 2:  Scanning Loop
  while (state ≠ se)
    NextChar(char) ; word := word + char
    if state ∈ F then reset stack s end
    s.push(state)
    cat := CharCat[char]
    state := δ[state, cat]
  -- Stage 3:  Rollback Loop
  while (state ∉ F ∧ state ≠ bad)
    state := s.pop()
    truncate word
  -- Stage 4:  Interpret and Report
  if state ∈ F then return Type[state]
  else return invalid
  end
```

Index (1)

Index (2)

Index (3)

Index (4)

Index (5)

Index (6)