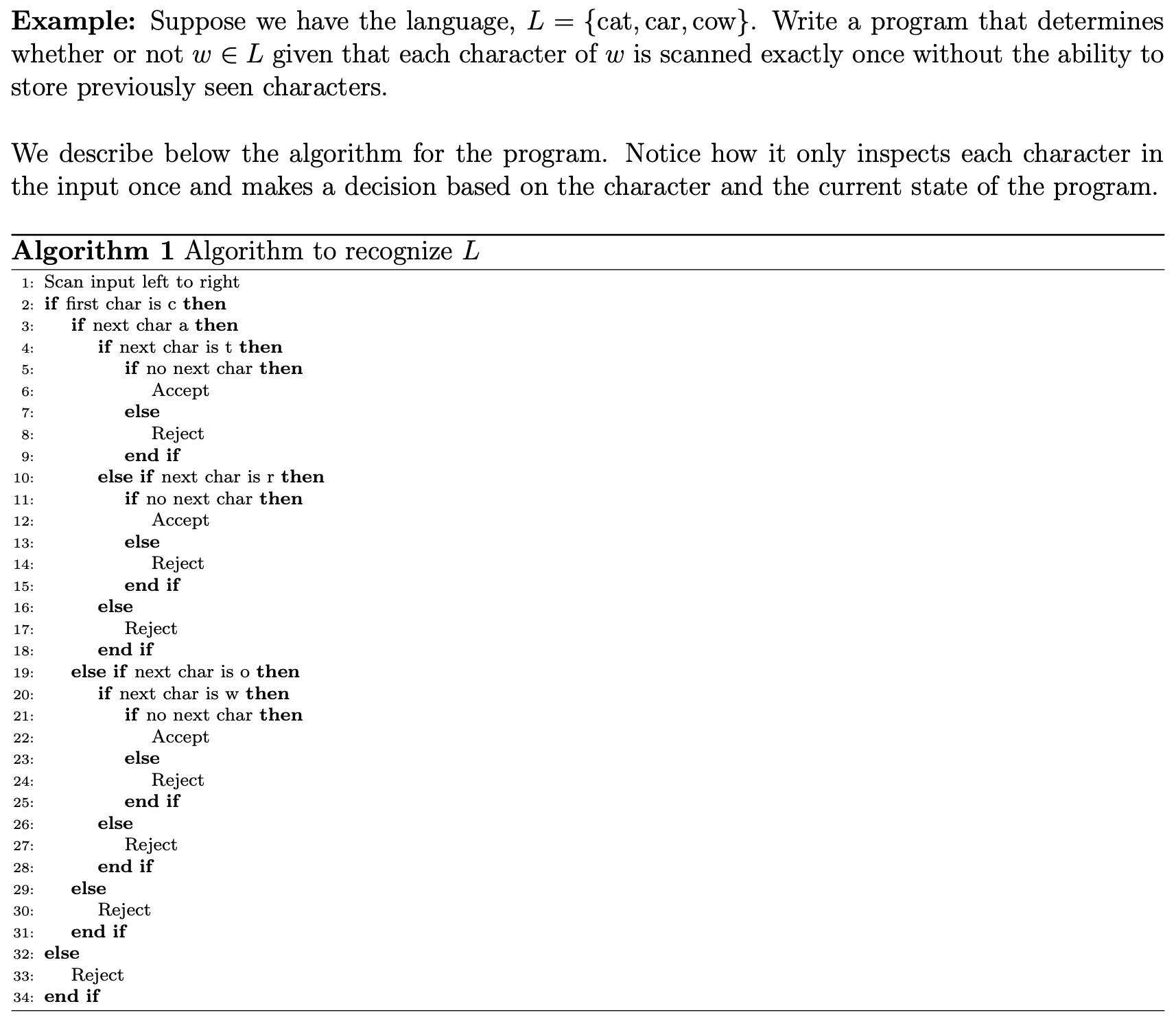
Formal Languages

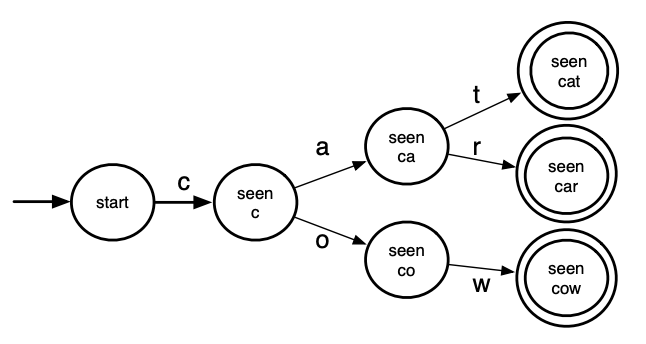
* an alphabet is a non-empty, finite set of symbols, often denoted by Σ (capital sigma)
  + like a lexicon/dictionary
  + e.g. Σ = {WHILE, IF, ELSE, VAR, NUM, EQUALS, BECOMES, PLUS, ...} alphabet of elements in a programming language
  + e.g. Σ = {0, 1} alphabet of binary digits
* a string/word w is a finite sequence of symbols chosen from Σ
  + set of all words over an alphabet Σ is denoted by Σ\*
  + length of word is denoted by |w|
* ε (epsilon) is the empty string
  + assume Σ will never contain ε since it’s just a notational convention for the empty string
* language is set of strings
  + e.g. L = ∅ or {}, the empty language
    - size = 0
  + e.g. L = {} the language consisting of the empty string
    - size = 1
  + e.g. L = {abna : n ∈ N} the set of strings over the alphabet Σ = {a, b} consisting of an a followed by 0 or more b characters followed by an a
* a language may be finite or infinite
* a language L over alphabet Σ is by definition a subset of Σ\*
  + Σ\* is also a language itself
* a recognition algorithm is a decision algorithm that takes the specification of a language and an input word, then answers whether the word satisfies the specification (i.e. whether the word is in the language)
  + a compiler does recognition, but then also translates the program into a different output language
* in order of relative difficulty on how hard it is to recognize languages: finite, regular, context-free, context-sensitive, recursive/decidable, and undecidable

Finite Languages

* since a finite language is a finite set of words, one way to specify the language is to list all the words but more efficient ways exist
* example of efficient recognition algorithm for finite language below:



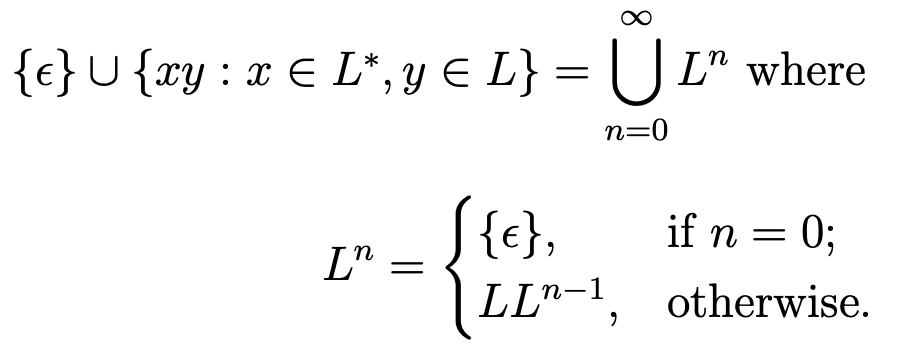
* + this can be represented more concisely as a state diagram



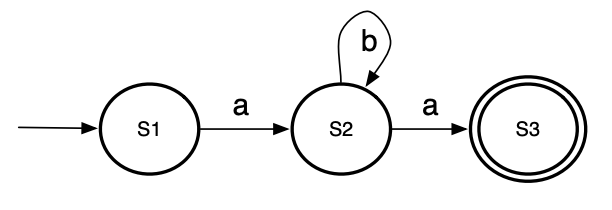
* + - each bubble rep a state: configuration of program based on input seen/scanned so far
    - there’s unique start state, denoted by arrow into it
    - arrows from state to state are labelled and rep transitions based on the next input symbol
    - if transition isn’t defined for the next input symbol, state transitions to implicit error state
    - accepting states have 2 circles

Regular Languages

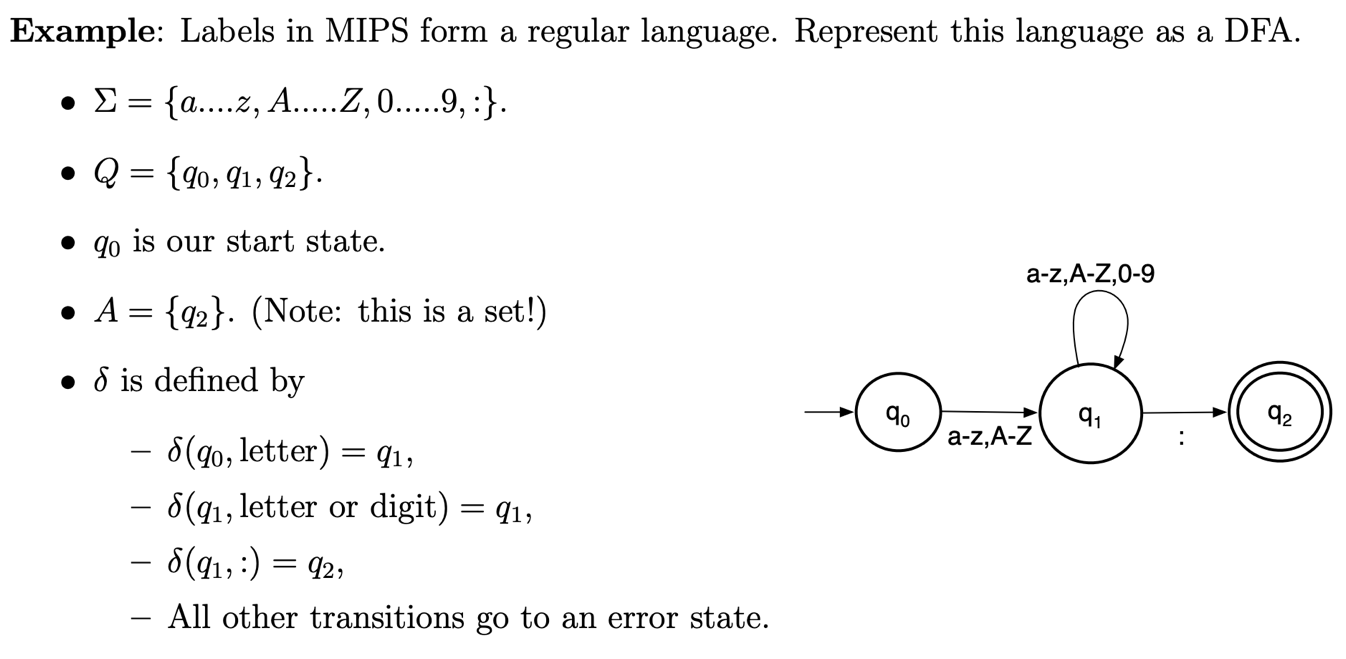
* language over an alphabet Σ is a regular language if
  + it’s the empty language, {}, or the language consisting of the empty word, {ε}
  + it’s a language of the form {a} for some a ∈ Σ
  + it’s a language built using the union, concatenation or Kleene star of 2 regular languages
* let L1, L2, and L3 be regular languages; the following are also regular languages:
  + union: L1 ∪ L2 = {x : x ∈ L1 or x ∈ L2}
  + concatenation: L1L2 = {xy : x ∈ L1, y ∈ L2}
* Kleene star: L\* =



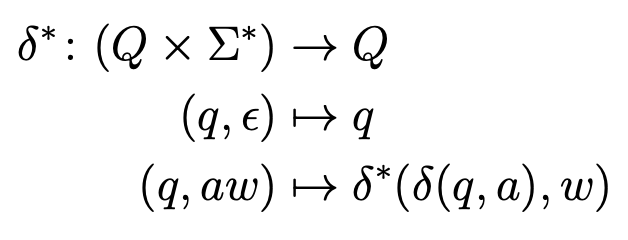
* + L\* is set of all strings consisting of 0 or more occurrences of strings from L concatenated together
* e.g. suppose L1 = {up, down}, L2 = {hill, load}, and L = {a, b} over appropriate alphabets:
  + L1 ∪ L2 = {up, down, hill, load}
  + L1L2 = {uphill, downhill, downhill, download}
  + L\* = {ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, ...}
* regular expressions are a way of expressing regular languages
* with notations, we drop set notation:
  + {ε} becomes ε
  + L1 ∪ L2 becomes L1 | L2
  + concatenation is still • or implicit
  + empty language is still ∅
* order of operations is \*, •, then |
* e.g. L = {abna : n ∈ N} is written as ab\*a as regular expression
* can use state diagrams to pictorially rep regular languages
  + since regular languages can be infinite, self-loops are introduced into state diagrams
* state diagrams are called deterministic finite automata (DFA)
  + transitions are deterministic, meaning for each state, there’s only one transition on a given symbol (i.e. can’t define multiple transitions from one state on the same symbol)



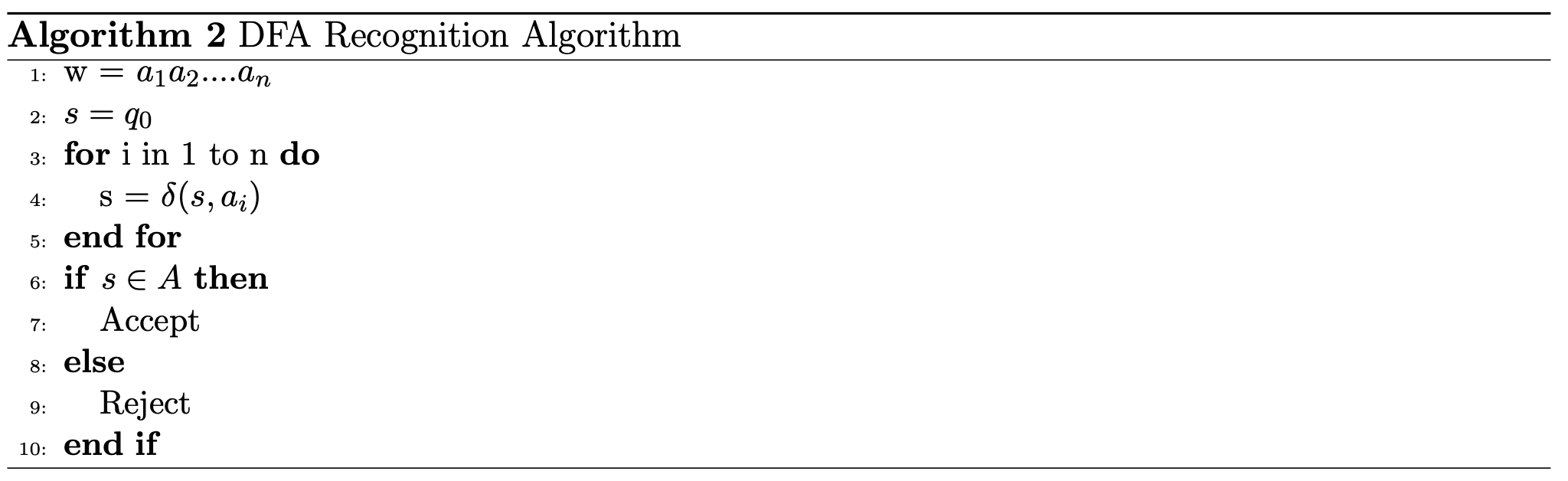
* a DFA is a 5-tuple (Σ, Q, q0, A, δ):
  + Σ is a finite non-empty set (alphabet)
  + Q is a finite non-empty set of states
  + q0 ∈ Q is a start state
  + A ⊆ Q is a set of accepting states.
  + δ : (Q × Σ) → Q is our [total] transition function (i.e. given a state and a symbol of our alphabet, which state to go to)
* e.g. DFA for L = ab\*a, Σ = {a,b}, Q = {S1,S2,S3}, q0 = S1, A = {S3}, and δ(S1, a) = S2, δ(S2, b) = S2 and δ(S2, a) = S3
* another example of a DFA:



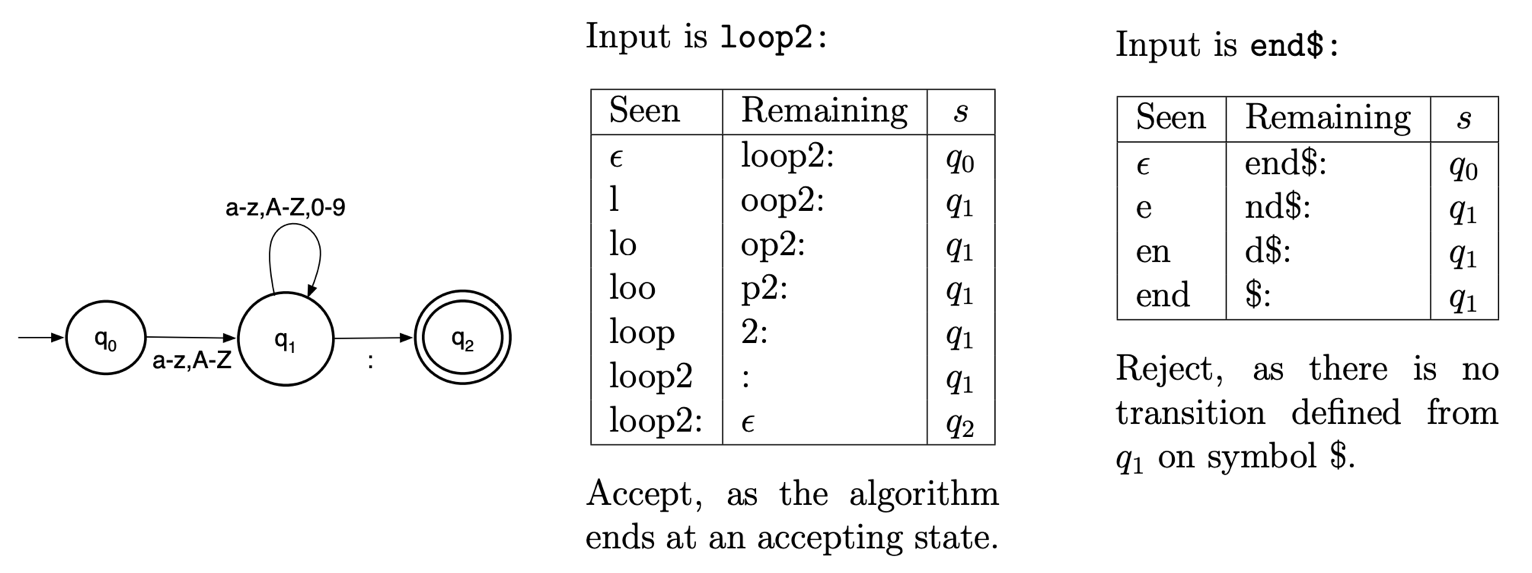
* to define a recognition algorithm for a word w in a regular language using a DFA, first extend definition δ : (Q × Σ) → Q recursively to a function δ\* : (Q × Σ\* ) → Q by doing:



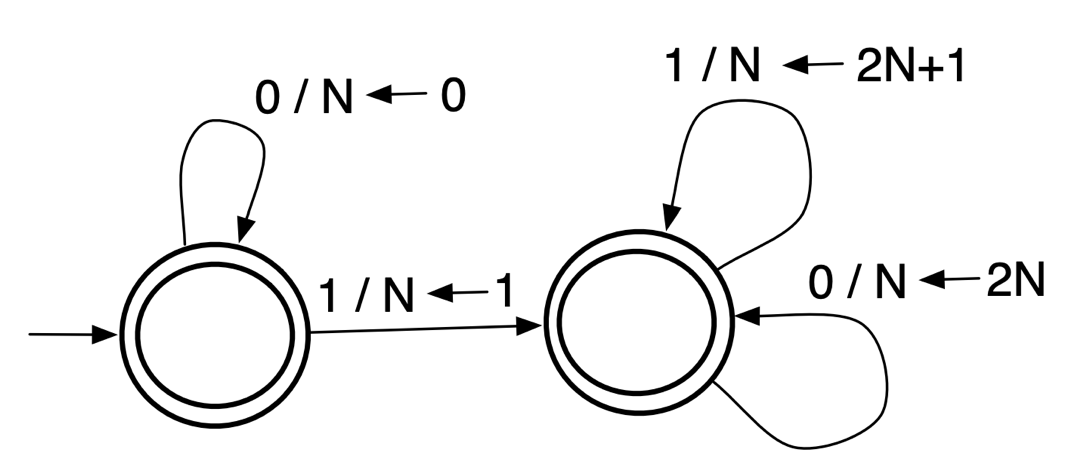
* + a ∈ Σ and w ∈ Σ\*
  + δ\* processes a string by processing the first character, then processing the rest of the string recursively
* DFA given by M = (Σ, Q, q0, A, δ) accepts a string w if and only if δ\*(q0, w) ∈ A
  + to recognize a word w=a1a2a3...an, start at DFA’s start state q0 and transition on first symbol a1 to end up at some qi
  + then, transition on a2 from qi
  + repeat until we have transitioned through all symbols a1...an
  + if resulting state is accepting, we accept and reject otherwise
  + with implicit error states, if transition is undefined, reject right away
  + as an algorithm:



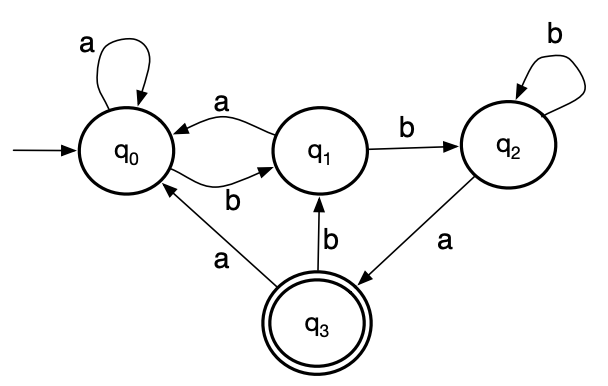
* e.g. using the above DFA for MIPS labels, we can run the recognition algorithm:



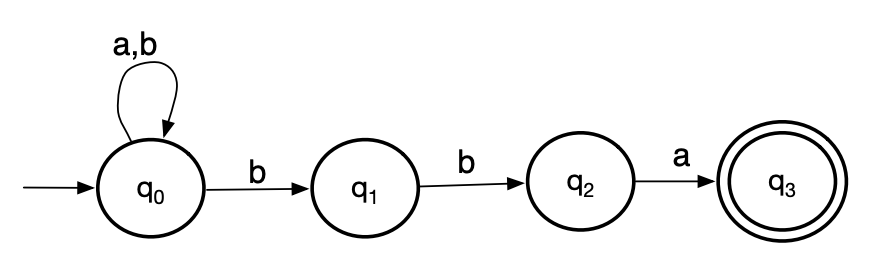
* the language of a DFA M, denoted L(M), is set of all strings accepted by M: L(M) = {w : M accepts w}
* theorem 1 (Kleene): L is regular iff L = L(M) for some DFA M
  + i.e. regular languages are precisely the languages accepted by DFAs
* can extend DFAs by attaching actions to each transition
  + aka finite transducers
  + e.g. a DFA that simultaneously detects if word is a subset of the language of binary numbers without leading zeroes and prints out the decimal rep



* + - read 1/N ← 2N + 1 as the left 1 corresponding to a DFA transition and the N ← 2N + 1 changes N to be 2N + 1
* DFAs are used in Scanning stage of compiler, where input is sequence of characters rep program and output is sequence of tokens
* non-determinism is when the machine can choose which path to go on
* e.g. consider DFA for L = {w : w ends with bba} over the alphabet Σ = {a, b}



* a non-deterministic finite automata (NFA) can be used to simplify the DFA
  + e.g. for above DFA, an NFA chooses to stay at q0 until the ending bba



* + an NFA allows for more than one transition from a state on the same symbol
* NFA is a 5-tuple (Σ, Q, q0, A, δ):
  + Σ is a finite non-empty set (alphabet)
  + Q is a finite non-empty set of states
  + q0 ∈ Q is a start state
  + A ⊆ Q is a set of accepting states
  + δ : (Q × Σ) → 2Q is our [total] transition function. Note that 2Q denotes the power set of Q, that is, the set of all subsets of Q
* only difference between DFA and NFA is how transition function is defined
  + in NFA, a state and symbol combination can lead to a set of states instead of a single new state
  + all DFAs are NFAs
* to get recognition algorithm, extend the definition of δ: (Q × Σ) → 2Q to a function δ\* : (2Q × Σ\* ) → 2Q by doing:

Text, letter

Description automatically generated

* + a ∈ Σ
  + δ\* processes a string by processing the first character, then processing the rest of the string recursively
  + since the input to δ is a single state and our result is a set of states, we must make a union of all the states as separate inputs
    - as long as one end state is accepting, the input is valid
* an NFA given by M = (Σ, Q, q0, A, δ) accepts a string w if and only if δ\* ({q0}, w) ∩ A != ∅
  + to recognize a word w=a1a2…an, start at NFA’s start state q0 and transition on first symbol a1 to end up in set of states Si
  + then, transition on each state qi from Si on symbol a2 to new set of states
  + repeat until we have transitioned on all of the symbols
  + if the resulting set of states contains an accepting state, we accept and reject otherwise

Graphical user interface, text, application, email

Description automatically generated

* e.g. using above NFA for recognizing if a word ends on bba, we can run recognition algorithm on input word abbba:

Diagram

Description automatically generated

* the language of an NFA M to be the set of all strings accepted by M: L(M) = {w: M accepts w}
* NFAs are not more powerful than DFAs because even the power-set of a set of states is finite
  + rep set of states in NFA as single states in DFA
* to run regular languages, convert it to NFA and then convert it to DFA
* to convert from NFA to DFA:
  + start with state S = {q0}
  + from this state, go to each NFA and determine what happens on each symbol a ∈ Σ for each state q ∈ S
    - set of resulting states should become its own state in DFA
  + repeat for each new state until we have exhausted every possibility/combination of states
  + accepting states are any states that included an accepting state in the original NFA
* example of converting NFA to DFA:

Diagram

Description automatically generated

* ε-Non-Deterministic Finite Automata permits state transitions without consuming a symbol
  + ε-transitions make it easy to construct concatenation of two NFAs and Kleene star of NFA
  + e.g. for the language L = {abc} ∪ {w: w ends with cc}, create an ε-NFA by creating a new start state and connecting it to the NFAs for {abc} and {w: w ends with cc} using ε transitions

Diagram

Description automatically generated

* an ε-NFA is a 5-tuple (Σ, Q, q0, A, δ):
  + Σ is a finite non-empty set (alphabet) that does not contain the symbol ε
  + Q is a finite non-empty set of states
  + q0 ∈ Q is a start state
  + A ⊆ Q is a set of accepting states
  + δ: (Q × Σ ∪ { ε }) → 2Q is our [total] transition function
    - 2Q denotes the power set of Q (i.e. the set of all subsets of Q)
* epsilon closure E(S) is set of all states reachable from S in 0 or more ε-transitions
* to get recognition algorithm, extend the definition of δ: (Q × Σ ∪ { ε }) → 2Q to a function δ\* : (2Q × Σ\* ) → 2Q by doing:

Text, letter

Description automatically generated

* + a ∈ Σ
* an ε-NFA given by M = (Σ, Q, q0, A, δ) accepts a string w if and only if δ\* (E{q0}, w) ∩ A != ∅
  + recognition algorithm is same as NFA except that for each set of states, including the start state, we compute its epsilon closure

Graphical user interface, text, application

Description automatically generated with medium confidence

* e.g. trace 2 below inputs on previous ε-NFA for L = {abc} ∪ {w: w ends with cc}

Diagram

Description automatically generated

* an ε-NFA exists for every regular language

Diagram

Description automatically generated

Scanning

* Scanning stage is conversion of sequence of characters into tokens
* in C, keywords, identifiers, literals, operators, and comments are regular expressions
  + finite automata can do the scanning/tokenization
* maximal munch scanning algorithm is greedy algorithm
  + i.e. it attempts to consume the max number of characters it can
  + e.g. consider input string s = ababca with Σ = {a, b, c}, L = {a, b, abca} and the DFA as shown below:

Diagram

Description automatically generated

* + - first consumes ‘a’ and flags q1 as a state because it’s accepting
    - then, consumes ‘b’ and moves on to state q2
    - algorithm is stuck because it’s not at an accepting state and there’s no transition on the symbol ‘a’
    - so, it backtracks to last accepting state and un-consumes the previous input since that last state
    - since last seen accepting state is q1 where only ‘a’ has been consumed, algorithm outputs that token and resets to the start state q0
    - resumes consuming ‘b’ and q5 as last seen accepting state
    - algorithm becomes stuck and since current state is accepting, ‘b’ is outputted and resets to q0
    - keeps going until the entire string has been consumed
* simplified maximal munch algorithm is same as maximal munch, but it doesn’t include the backtracking
* difference between the 2:
  + maximal munch: consume characters until there’s no valid transition; if there’s still characters left to consume, backtrack to the last valid accepting state and resume
  + simplified maximal munch: consume characters until there’s no valid transition; if current state is accepting, produce the token and proceed; otherwise go to an error state
* e.g. for input string s = ababca with Σ = {a, b, c}, L = {a, b, abca} and DFA as shown above, simplified algorithm would reject the token
  + can use whitespace to account for these limitations and have the input string be w = a b abca

Shape

Description automatically generated

* algorithm for simplified maximal munch (assuming input machine is DFA (Σ, Q, q0, A, δ)

Graphical user interface, text, application, email

Description automatically generated

* + assumes function peek(w) returns next symbol without consuming it and consume(w) consumes and returns next symbol from input (while mutating input string)