

David Keil
Framingham State University

2. Finite automata and regular languages

1. Deterministic finite automata
2. Regular languages and regular expressions
3. Nondeterministic finite automata
4. Proving a language is not regular
5. Lexical analysis

Reading: H-M-U, Ch. 2-4

David Keil Theory of Computing 2. Finite automata 1/12 1

Inquiry

- What problems can be solved by computation?
- What computations can state-transition systems model?

David Keil Theory of Computing 2. Finite automata 1/12 2



Objectives

- 2a. Recognize and construct finite automata with given behavioral features
- 2b. Convert between regular expressions and finite automata
- 2c. Use non-diagonal proof by contradiction, e.g., the Pumping Lemma
- 2d. Use constructive proof to show expressiveness of finite automata
- 2e. Prove that a given automaton accepts a given language

Logic circuits

- A model for the computation of functions $f: \{0,1\}^m \rightarrow \{0,1\}^n$ where m is the arity of f (number of inputs to circuit) and n is the number of outputs
- *Proposition:* for any such function f , a logic circuit that computes f may be constructed from a finite number of \wedge , \vee , or \neg gates
- A logic circuit computes a function on strings of bounded size, i.e., functions with a finite domain

1. Deterministic finite automata

- A *transition system* is a labeled digraph, where vertices denote *state* (memory), edges denote transitions and labels
- 1-state example:  2-state example: 
- DFA differs from *random-access machine* (RAM) model, in which state is a value assignment to a set of variables

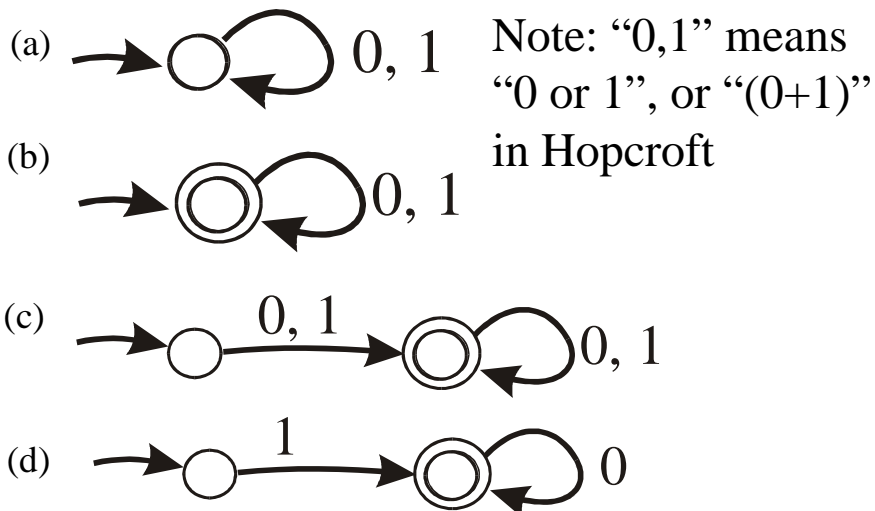
Transition systems

- Components:
 - Alphabet Σ (set of symbols)
 - State set (a state can be anything)
 - Transition function or relation
(rules for going from one state to another)
- Corresponds to a graph, labeled with symbols that trigger transitions
- DFA has finite set of states, unique destination for any transition

Finite transducers

- A finite-state transducer may have output symbols as part of its labels
- FTs are called *Mealy* and *Moore* machines
- On a given input, an FST will output a string of the same length
- Hence it is not an accepter, but performs transduction from one string to another
- See topic 6 (Interaction)

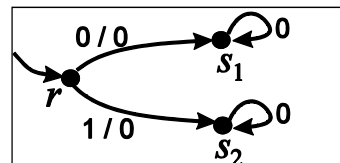
Some simple DFAs



Graph of a transition function

- A *graph* $G = (V, E)$ is a set of vertices and a set of ordered pairs of vertices
- E is a *relation* on V
- If edges are *labeled* with symbols, and each edge from vertex u labeled with symbol a goes to a unique vertex v ...
- ... then the labeled graph denotes a *transition function*

$$\delta : V \times \Sigma \rightarrow V$$



Definition: DFA

- A DFA is a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$ where
 - Q is a set of states
 - Σ is a finite alphabet
 - $\delta: Q \times \Sigma \rightarrow Q$ is a state-transition function
 - $q_0 \in Q$ is the starting state
 - F is the set of accepting states
- We can also define $\delta^* : Q \times \Sigma^* \rightarrow Q$, the *reflexive transitive closure* of δ , which tells what state δ yields for a *string*

Reflexive transitive closure

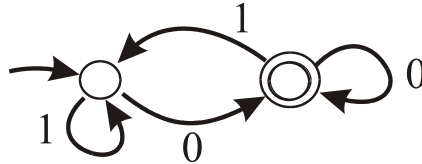
- A *relation* or *function* may be applied over and over; for example, $f(f(f(x)))$
- The reflexive transitive closure of a function or relation is the set of values that can be obtained by applying the function over and over in this way.
- *Example:* the reflexive transitive closure of the addition operation on natural numbers is ... the natural numbers

Language of a DFA

- For DFA M , the language of M , $L(M) = \{ x \in \Sigma^* \mid (\delta^*(q_0, x) = q') \wedge (q' \in F) \}$
- ...where Σ^* is the set of all strings over Σ , and for string x , $\delta^*(q_0, x)$ is the state reached after repeated applications of δ to states and to elements of x
- *Example:* $L(M)$ for M below is all bit strings that have a '1' followed by 0 or more '0's.



Example DFA



- This DFA accepts strings that start with any number of 1's (possibly none), followed by a 0, followed by any number of (10)s, followed by any number of 0s
- Some elements of its language:
0, 10, 00, 100, 110, 010, 1010, 10100

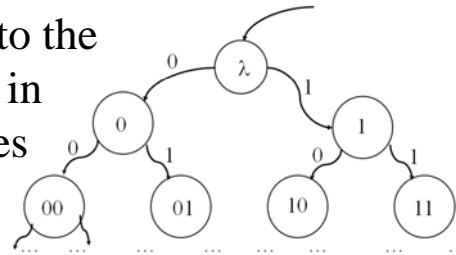
Regular languages

- *Language*: a set of sequences over a finite alphabet of symbols
- *Regular language*: a language whose elements are all recognized by some DFA
- The following decision problems are equivalent:
 - Is sequence x in regular language L ?
 - Is x accepted by a DFA A , where $L(A) = L$?
- Let \mathcal{RL} be the set of regular languages

Finite languages are regular

Proof by construction:

- For a finite language L , construct a DFA whose transition paths form the tree corresponding to all the strings in L
- Each vertex has up to $|\Sigma|$ out-transitions
- Edges corresponding to the last symbol of a word in L go to accepting states



2. Regular languages and regular expressions

- Any RL may be specified by a *regular expression* using any combination of these operations:
 - Concatenation
 - Selection ($|$, $+$, \cup)
 - Iteration ($*$) (binds to immediate preceding symbol)
- *Examples* (what are their DFAs?):
 - $(01)^*$: all strings that repeat the string 01, zero or more times
 - $01^* | 10^*$: all strings that either consist of a 0 followed by zero or more 1's, or consist of a 1 followed by zero or more 0's

Strings

- *Alphabet*: a finite set of symbols, e.g., $\{0,1\}$, $\{0, 1, \dots, 9\}$, $\{a, b, c, \dots, z\}$
- A *string* is a sequence of symbols over a finite alphabet; Σ^* is the set of strings over Σ
 Recursive definition:
 - *Base*: $\lambda \in \Sigma^*$ ($\lambda = \varepsilon = \text{null string}$)
 - *Recursive*: $a \in \Sigma \wedge s \in \Sigma^* \Rightarrow sa \in \Sigma^*$
 - *Restriction*: *only* objects defined as above are strings
- In set notation, $\Sigma^* = \{\lambda\} \cup \{ax \mid a \in \Sigma, x \in \Sigma^*\}$

Formal languages

- *Language*: a set of strings over an alphabet
- Let alphabet Σ be $\{0,1\}$, let λ be the null string
- Then $\Sigma^0 = \{\lambda\}$, $\Sigma^1 = \{(0), (1)\}$
 $\Sigma^2 = \{00, 01, 10, 11\}$
 $\Sigma^k = \text{the set of strings of length } k$
 $\Sigma^* = \bigcup_{k \in \mathbb{N}} \Sigma^k$
- For any language L over Σ , $L \subseteq \Sigma^*$

Operators on languages

- $L_1 L_2$ (concatenation) is the language consisting of strings that are a string in L_1 followed by one in L_2
- $L_1 \mid L_2$ (union) is the language consisting of strings that are either in L_1 or L_2
- L^* (Kleene star) is the language consisting of strings that are sequences of strings in L
- $L_1 \cap L_2$ (intersection) is the language consisting of strings that are in both L_1 and L_2

Operations on languages

- *Concatenation:* $L_1 L_2 = \{xy \mid x \in L_1 \wedge y \in L_2\}$
- *Selection:* $L_1 \cup L_2 = \{x \mid x \in L_1 \vee x \in L_2\}$
- *Iteration:* $L^* = \{xy \mid x \in L \wedge y \in L^*\}$

Regular expressions and languages

- A *regular expression* specifies a language
- The *regular languages* are those languages specified by regular expressions
- Where a is any single symbol in Σ and E, F are regular expressions:
$$RE = \{a\} \cup (E) \cup EF \cup E | F \cup E^*$$
- *Example:* $01^* | 00$ is the regular expression denoting strings beginning 0, followed by any number of 1's, or 0 followed by a single 0

Applications of regular expressions

- Regular expressions identify patterns, such as strings that begin with 'M' or consist of digits, possibly with a decimal point in the middle
- Patterns may guide search in databases, including in bioinformatics, where
 $\Sigma = \{A, C, G, T\}$
- Regular languages are languages accepted by finite automata (see next subtopic)
- Lexical analysis of programs by compilers is guided by regular expressions

Complement of a language

- *Complement* of L is all strings not in L
Examples: Complement of Σ^* is \emptyset ,
complement of 0^* is $(0+1)^* 1 (0+1)^*$
- *Theorem:* The complement of a regular language is regular.
- *Proof:* Construct a DFA that accepts all inputs *not* in L , starting with a DFA that recognizes L ; make all accepting states be non-accepting, all non-accepting states accepting

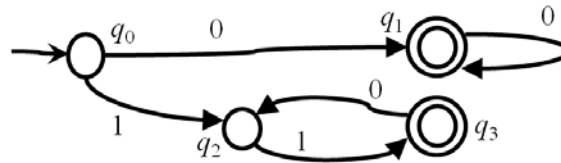
DFA \rightarrow RE

Cases:

- *Sequence* of states with transitions:
Concatenate REs ($E_1 E_2$)
- *Branch* from one state to two or more others:
 $E_1 + E_2 + \dots + E_n$
- *Loop* (transition from one state to the same or a preceding state): Star the RE for the path from destination of transition to its origin (E^*)

A method for DFA \rightarrow RE

- *Observation:* Any state q of a DFA M may be associated with a regular expression that denotes the paths that take M from start state to q



$$L(q_0) = \lambda$$

$$L(q_1) = 00^*$$

$$L(q_2) = 1(01)^*$$

$$L(q_3) = 11(01)^*$$

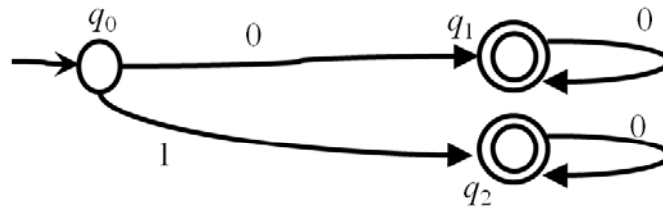
Reg. expr. for this DFA is $L(q_1) \cup L(q_3)$

Minimal DFA for a RL

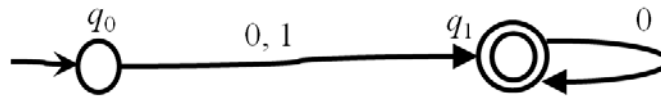
- *Theorem:* For any RL L , there exists a *unique** DFA M s.t. $L(M) = L$ and M has no more states than any other DFA M' where $L(M') = L$ (Myhill-Nerode)
- There exists an algorithm that reduces a DFA to its minimal version
- The algorithm proceeds by finding indistinguishable states and merging them

* unique up to isomorphism (renaming of states)

Minimization example



- In the DFA above, states q_1 and q_2 are indistinguishable
- The DFA below is minimal for the same language



Structural induction

- Many sets, including strings, trees, and formal languages, may be defined recursively
- Method for showing that recursively defined set S has property P :
 1. Show that $P(x)$ for each element of the *base* of S
 2. Show that for each recursive rule, applying the rule to an element that satisfies P yields an object that also satisfies P


Example of structural induction

- Let $S = \{()\} \cup \{(x) \mid x \in S\} \cup \{yz \mid y, z \in S\}$
- *Theorem:* Every element of S has equal numbers of left and right parentheses (property P)
- *Proof:*
 1. *Base:* $()$ has the property P

Induction:

 2. $P(x) \Rightarrow P((x))$ adding one left, one right yields balance
 3. $P(x) \wedge P(y) \Rightarrow P(xy)$ concatenating two strings, each with equal left and right, yields balance

Proving correctness of DFAs

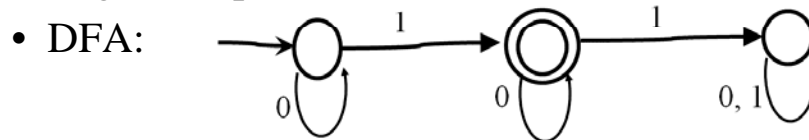
- We use *structural induction* on the size of input strings to show that a given DFA accepts only a certain language
- *Example:* M (right) accepts $L = 10_1^*$
- *Base case:* $1 \in L(M)$ 
- *Induction:* From accept state b , adding a 0 keeps string in L , whereas adding a 1 triggers rejection
- General form of assertion to be proven:

$$(\forall x \text{ s.t. } |x| = n) x \in L \Leftrightarrow x \in L(M) \quad \Rightarrow$$

$$(\forall x \text{ s.t. } |x| = n + 1) x \in L \Leftrightarrow x \in L(M)$$

Example proof of correctness

- Language specification: $n_1(x) = 1$
- Regular expression: 0^*10^*



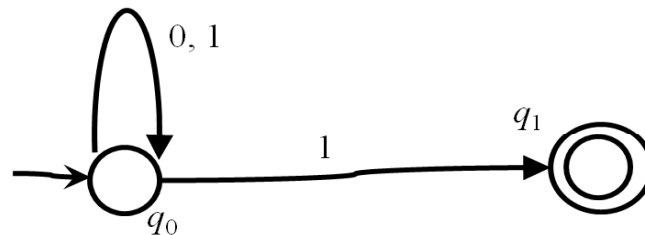
- Proof of correctness:
 - *Base*: $\lambda \notin L(A)$, because left state rejects
 - $1 \in L(A)$, because middle state accepts
 - *Induction*: 1 takes accepted string to reject state; 0 takes any string to same state

3. Nondeterministic finite automata

- *Definition*: NFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$, s.t. $\delta: Q \times \Sigma \rightarrow 2^Q$ permits transitions to any of several states from a given state on a given symbol, and
- δ may have λ -transitions (transitions without input)
- A transition from q to q' on a , with $q, q' \in Q, a \in \Sigma$, denotes that the NFA *can* go from q to q' on a

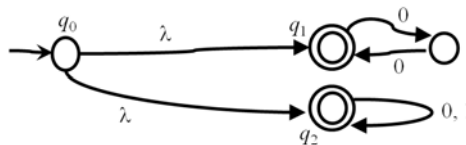
Uses for nondeterminism in theory work

- *Problem:* Some REs are hard to convert to DFAs, e.g., $L = (0 \cup 1)^* 1$
- *NFA solution:*



λ -transitions

- We may consider any state q to have a language, i.e., $L(q) = \{x \in \Sigma^* \mid \delta^*(q, x) \in F\}$
- Now, if q has a λ transition to r , then $L(r) \subset L(q)$, because any string can take M from r to the same states as from q
- Let λ -closure(q) = $\{q\} \cup \lambda$ -closure($\{r \mid r \in \delta(q, \lambda)\}$)
- Solves problem of hard to convert REs, e.g., $(00)^* \mid (0 \mid 1)^*$



Theorem: DFA \rightarrow NFA

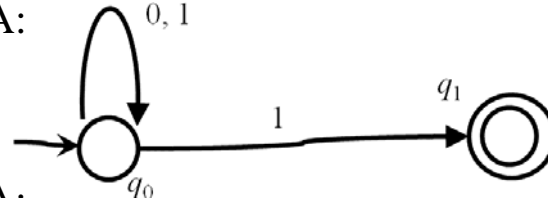
- *Theorem:* Every regular language is recognized by some NFA
- *Proof:* A DFA is by definition an NFA without nondeterministic transitions

RE \rightarrow NFA

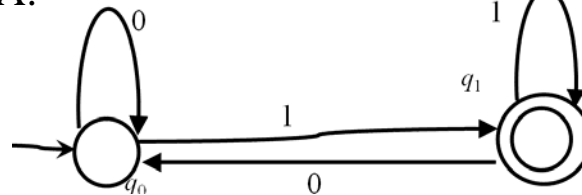
- *Theorem:* For any regular expression, an NFA may be constructed that recognizes the same language
- *Proof (by construction):*
 - for concatenated parts of RE, a sequence of states
 - for “+”, a fork
 - for “*”, a transition to first state in starred part of RE
- *Example:* $(0 | 1)^* 1$ (strings that end in 1)

RE → NFA example

- RE: $(0 | 1)^*1$ (strings that end '1')
- NFA:



- DFA:



NFA → DFA (subset construction)

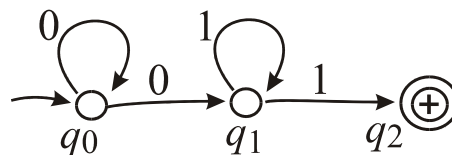
- To construct a DFA $M' = \langle Q', \Sigma, \delta', q_0', F' \rangle$ from an NFA $M = \langle Q, \Sigma, \delta, q_0, F \rangle$, let $Q' = 2^Q$ (set of all subsets of Q).
- Then derive δ' from δ by merging all states with λ -transitions between them...
- and let $\delta'(q, a) = \bigcup_{r \in q} \{ \delta(r, a) \}$ (the state of M' that is the set of all states of M to which there is a transition on a from a state in q)
- F' is the set of states of M' that contain *some* state in F
- $q_0' = \{q_0\}$

Intuition for the subset construction

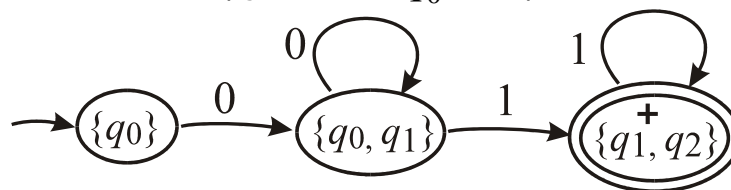
- The DFA has states that are sets of NFA states, because the NFA transition function is from states to sets of states
- λ -transitions mean that states involved in them are in effect equivalent, so they merge in the DFA M'
- $\delta'(q, a)$ of DFA M' is the set of states the NFA M can go to from some state that is in q of DFA M' .

NFA \rightarrow DFA example

NFA $M = \langle Q, \Sigma, \delta, q_0, F \rangle, L(M) = 0^+1^+$

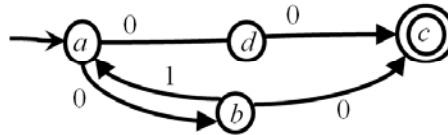


DFA $M' = \langle Q', \Sigma, \delta', q_0', F' \rangle, L(M') = 0^+1^+$

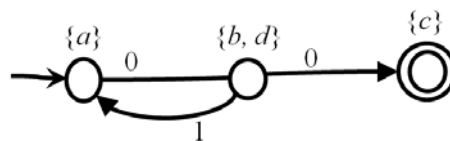


Subset construction example

- NFA $M = \langle Q, \Sigma, \delta, q_0, F \rangle$



- $L(M) = (01)^*00$
- Construct DFA $M' = \langle Q', \Sigma, \delta', q_0', F' \rangle$



Significance of NFA \rightarrow DFA

- Every NFA can be converted to a DFA, and every DFA is an NFA
- This means that NFAs have the same computational power as DFAs (recognize the same languages)
- So the following are equivalent:
 - Some NFA recognizes language L
 - Some DFA recognizes L
 - L is regular
 - L is generated by a regular expression

Theorem: one accept state

- *Thm:* For any DFA, there is an NFA with only one accept state that recognizes the same language
- *Proof:* Construct the NFA from the DFA, keeping one accept state q_{acc} and such that each other acceptance state
 - becomes a reject state, and
 - has a λ transition to q_{acc}

Reverses of RLs are regular

Theorem: If L is regular, then *reverse* of L , L^R (set of elements of L each spelled backwards), is regular

Proof: Construct an NFA that accepts L^R :

1. start with a one-accept-state DFA that recognizes L
2. reverse the directions of all transitions
3. swap accepting and starting states
4. new start state is a new state with λ transitions to each old accept state

Questions?

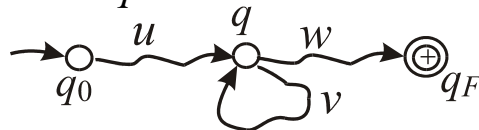
- What are your questions? About DFAs? RLs? REs? NFAs? _____
- How are the exercises?
- Shall we do some?

4. Proving Ls non-regular

- *Lemma:* A property of infinite regular languages L :
 $(x \in L) \Rightarrow (x = uvw)$ s.t. $(\forall k \in \mathbf{N}) uv^k w \in L$,
 $|v| > 0$
- I.e., any string in the language can be expressed as the concatenation of three strings, the middle of which can be “pumped” any number of times without leaving the language
- Proof of this lemma uses Pigeonhole Principle
- This lemma is used to show that certain languages are *not* regular

Pumping Lemma proof

1. Consider RL L and DFA M , s.t. $L = L(M)$, $M = \langle Q, \Sigma, \delta, q_0, F \rangle$, and L is infinite
2. Then for some x in L , $|x| > |Q|$
3. So when M inputs x , M has to pass through some state q more than once



4. Hence there exists $n_0 \leq |Q|$ s.t. if $\delta^*(q_0, u) = q$, $\delta^*(q, w) = q_F$, $uvw \in L$, $|v| > 0$, and $|uvw| \geq n_0$, then $uv^k w \in L$ for any $k \in \mathbf{N}$.

Proving a language is not regular

Theorem: If $L = 0^n 1^n$ then $L \notin \mathcal{RL}$

Proof: By **Pumping Lemma**, if L is regular then $(\forall x \in L)$
 $(\exists u, v, w)$ s.t. $(uvw = x \wedge |v| > 0 \wedge (\forall k) uv^k w \in L)$

Cases:

- If v were all 0's then we could "pump" it with a large enough k so that there are more 0's than 1's so v is not all 0's; similarly v can't be all 1's
- If v is 0's and 1's, then pumping even once produces a string not in L
- Hence no v can exist that satisfies the Pumping Lemma
- Therefore L is not regular

5. Lexical analysis

- *Lexical analysis* (tokenization) is used by compilers and is a precondition for parsing
- To recognize identifiers, numerals, operators, etc., implement a DFA in code
- *State* is an integer variable, δ is a *switch* statement
- Upon recognizing a lexeme, return its lexical class and restart DFA with next character in source code

Lexical analyzers

- Compiler translates from higher-level language to assembler or machine language
- *Lexical analysis*
 - Finds *tokens*, indivisible items of code
 - Tokens are formed by simple rules
 - Examples: literals, operators, keywords, delimiters, identifiers
 - *Lexer* arranges tokens as an ordered list
- *Parsing* applies grammar rules to build tokens into a structure

Lexer example: *identifier*

- $ID = (_ | \text{letter}) (\text{letter} | \text{digit} | _)^*$
- DFA has accepting state for letter or underscore, or for same followed by letter, underscore, or digit
- Pseudocode:


```

i ← 0, q ← 0
      if next char is letter or underscore
          q ← 1
          i ← i + 1
      else
          q ← 2
      while next character is letter, underscore, or digit
          i ← i + 1
      return (q = 1)
      
```

Some lexical categories

- The lexical analysis of a program separates it into *atoms* or lexemes that cannot be further broken down, such as delimiters, keywords, identifiers, operators, etc.
- In *lexer.cpp* and *lexer.c*, these categories are values of the enumerated type *lex_categories*:

```

enum lex_categories { tkNone, tkError, tkID, tkNum,
    tkString, tkCharLit, tkHdrFName, tkLBrace, tkRBrace,
    tkLParen, tkRParen, tkLBracket, tkRBracket, tkExtractor,
    tkInserter, tkSemi, tkComma, tkColon, tkPeriod,
    tkAddop, tkAsgnop, tkMulop, tkNot, tkOr, tkAnd,
    tkRelop, tkScope, tkIncOp, tkPound, tkAmpersand,
    tkLess, tkGrtr, tkInclude, tkInt, tkCase, tkChar,
    tkFloat, tkIf, tkSwitch, tkWhile, tkFor, tkDo,
    tkStruct, tkClass, tkElse, tkPublic, tkPrivate, tkEnum,
    tkConst, tkReturn, tkStatic, tkMain, tkVoid, tkCout,
    tkCin };
  
```

Questions

- What most stayed in your mind in discussing this topic?
- For you, what was the *least* clear concept that you encountered in this topic?

References

Daniel I. I. Cohen. *Introduction to Computer Theory*, 2nd ed. Wiley, 1997.

J. Hopcroft, R. Motwani, J. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 3rd Ed. Addison-Wesley, 2007.

See also *JFLAP* manual.