2. Finite automata

1. Deterministic finite automata
2. Regular expressions
3. Nondeterministic finite automata
4. Non-regular languages
5. Lexical analysis

Inquiry

• What problems can be solved by computation?
• How does UNIX grep work?
• How do compilers work?
• What problems can simple state-transition systems solve?

*Stephen Kleene, 1909-1994*
Topic objective

Explain the relation between finite automata and regular expressions, proving related expressiveness and correctness results

Subtopic objectives

2.1a Construct a finite automaton**
2.1b Prove that an automaton accepts a given language**
2.1c Define a finite-state model other than the DFA
2.2 Give a regular expression for a FA*
2.3 Show expressiveness of finite automata**
2.4 Prove that a language is not regular
2.5 Describe how DFAs are used in compilation
1. Deterministic finite automata

• What is a language?
• A computation?
• A state?
• An automaton?
• What computations can looping state-transition systems model?

Subtopic objectives

2.1a Construct a finite automaton**
2.1b Prove that an automaton accepts a given language**
2.1c Define a finite-state model other than the DFA
Transition systems

- A transition system is a labeled digraph, where vertices denote state (memory), and labeled edges denote transitions

- Components:
  - Alphabet $\Sigma$ (set of symbols)
  - State set (a state can be anything)
  - Transition function or relation (rules for going from one state to another)

- Models a process such as a computation

UML state diagrams

- Depict a process that transitions from state to state; e.g., academic courses
Some language-decision problems

- Some languages:
  - $1 \mid 0 \quad \{0, 1\}$
  - $(1 \mid 0) \quad \{11, 01\}$
  - $1^* \quad \{\lambda, 1, 11, \ldots\}$
  - $1^*0 \quad \{0, 10, 110, 1110, \ldots\}$
  - $0(1|0)^* \quad \{0, 00, 01, 000, 001, 010, 011, \ldots\}$

Q: Can a transition system accept strings in an infinite language?

---

DFAs are transition systems

- 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
Some simple DFAs

(a) \[ \quad \]

(b) \[ \quad \]

(c) \[ \quad \]

(d) \[ \quad \]

Note: “0,1” means “0 or 1”, or “(0+1)” in Hopcroft

JFLAP

- An environment for designing and testing automata
- Useful in topics 2-4 of this course
- Free download: http://www.jflap.org/
Theorem: Some DFAs accept infinite languages

- Proof: The DFA at right accepts \{0,1\}*, the infinite set of all strings over \{0,1\}

- \textit{Proof:} For any string \( x \) accepted by the DFA, \( x0 \) and \( x1 \) are accepted as well (just follow the transition back to the accepting state)

Formal definition of DFA

- A DFA is a 5-tuple \( \langle Q, \Sigma, \delta, q_0, F \rangle \) where
  - \( Q \) is a finite set of states
  - \( \Sigma \) 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

- If the repeated application of \( \delta \) to the (state, symbol) pairs in a computation leaves the DFA in a state in \( F \), \textit{accept} the input; else \textit{reject}
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 $\delta : V \times \Sigma \rightarrow V$

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 in this way.
- Examples:
  - the reflexive transitive closure of $\mathbb{N}$ under the addition operation is $\mathbb{N}$
  - $\delta^* : Q \times \Sigma^* \rightarrow Q$ maps from (state, string)
Language of a DFA

- For DFA $M$, the language of $M$, $L(M) = \{ x \in \Sigma^* | (\delta^*(q_0, x) = q') \land (q' \in F) \}$
- $M$ accepts strings in $L(M)$, rejects all others
- …where $\Sigma^*$ is the set of all strings over $\Sigma$, and for string $x$, $\delta^*(q_0, x)$ is the state reached after repeated applications of $\delta$ 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
- That is, any string that ends in ‘0’
- 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**: one whose elements are each accepted by some DFA (cf. subtopic 1.1)

- The following decision problems are equivalent:
  - Is sequence \( x \) in regular language \( L \)?
  - Is \( x \) accepted by a DFA \( A \), where \( L(A) = L \)?

- We write \( \mathcal{R}L \) for the set of regular languages

**Theorem: Finite languages are regular**

*Proof by construction:*

- For a finite language \( L \), construct a DFA in which each vertex has up to \( |\Sigma| \) out-transitions

- Transition paths form the tree corresponding to all the strings in \( L \)

- Edges corresponding to the last symbol of a word in \( L \) go to accepting states
Expressiveness of DFAs

- *Implication of previous theorem:* DFA model is *more expressive* than logic-circuit model, since some regular languages are infinite

- We will show a model later that has *equivalent* expressiveness (computing power) to DFAs, and other models that are *more* expressive

---

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.

Finite transducers

- A finite-state transducer may have output symbols as part of its labels.
- We assume that input is fixed before computation starts.
- FTs are called Mealy and Moore machines.
- On a given input string, an FST outputs a string of the same length.
- Hence it is not an accepter, but performs transduction from one string to another.
Mealy machines

- **Definition:** a 6-tuple \( \langle Q, \Sigma, \Gamma, \delta, \text{out}, q_0 \rangle \) s.t.
  - \( Q \) is a set of states, \( q_0 \) is start state
  - \( \Sigma, \Gamma \) are finite input, output alphabets
  - \( \delta \) is a transition function \( Q \times \Sigma \to Q \)
  - \( \text{out} \) is an output function \( Q \times \Sigma \to \Gamma \)

- **Example (right)**
- **Question:** What functions on strings can Mealy machines compute?

Transducers and accepters

*Theorem:* Any computation of a function \( f: \{0, 1\}^* \to \{0, 1\}^n \) can be done by \( n \) accepters

*Proof:*
1. For \( i \leq n \), let \( L_i = \{ x \mid \text{bit } i \text{ of } f(x) \text{ is } 1 \} \)
2. Construct accepters \( M_{1..i} \) for \( L_{1..i} \) as follows.
3. \( M_i \) decides the \( i \)th bit of \( f(x) \) for any string \( x \).
4. Let \( P_i \) be the predicate computed by \( M_i \)
5. So \( f(x) = \text{Concat}_{i \leq n}(P_i(x)) \)
6. Hence \( f(x) \) is computable by \( M_{1..n} \)
Kripke structures

- Unlabeled transition system, used to diagram and reason about reactive systems
- To a Kripke structure corresponds an infinite computation tree reflecting all possible paths through the system

Markov models

- A Markov state machine or chain is a system with a finite number of observable states, and with probabilistic transitions between states
- Defined by initial state $s_0$, transition model $T(s, a, s')$, and reward function $R(s)$
- Example: weather at any location
Example: Weather

- Let states be \{sunny, cloudy, rainy\}
- Let transition probabilities be in labels
- First-order Markov model:

![Graphical representation of a Markov model with states sunny, cloudy, and rainy, and transition probabilities between them.]

2. Regular expressions

- What is *syntax*?
- Do RLs have a common syntax?
- How does wild-card search work?
- Can we prove that a given DFA accepts a given language?
2.2 Give a regular expression for a finite automaton*

Regular expressions

- Any regular language may be specified by a \textit{regular expression} using any combination of these operations:
  - Concatenation
  - Selection ($\mid$, $+$, $\cup$)
  - Iteration ($\ast$) (binds to immediate preceding symbol)
- Where $a$ is any single symbol in $\Sigma$, and $E$, $F$ are regular expressions, these are REs:
  $a \quad (E) \quad EF \quad E \mid F \quad E^\ast$
Example RE

• The language accepted by the DFA below is generated by the regular expression 1(0 | 1)*

• Note: Infinite languages are specified by * in REs and recognized using loops in DFAs

Regular expressions and languages

• A regular expression generates a language

• 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 a 1 followed by zero or more 0’s

• 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 $\Sigma = \{A, C, G, T\}$ (amino acids)
- 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 $\Sigma^*$ 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 accepts $L$; make all accepting states be non-accepting, all non-accepting states accepting
DFA → RE conversion

Cases:

- **Sequence** of states with transitions: Concatenate REs \((E_1E_2)\)
- **Branch** from one state to two or more others: \(E_1 + E_2 + \ldots + E_n\)
- **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 \((E^*)\)

Algorithm for DFA → 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 \(L(q_1) \cup L(q_3)\)
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

For any graph $G = (V, E)$, $\Sigma_{v \in V} \delta(v) = 2|E|$ (sum of the degrees of vertices in a graph is twice number of edges.)

Proof:

- **Base**: $G = (\emptyset, \emptyset)$: $\Sigma_{v \in V} \delta(v) = 2|E| = 0$
- **Inductive**: To show that if for all graphs s.t. $|V| = n$, $\Sigma_{v \in V} \delta(v) = 2|E|$, then same where $|V| = n + 1$.
- For any graph, adding one vertex alone (with no edges) changes neither $\Sigma_{v \in V} \delta(v)$ nor $E$.
- Adding 1 edge, yielding $G' = (V, E')$, increases the degree of a participating vertex by one: $|E'| = |E| + 2$. 
Proving correctness of DFAs

- We may use structural induction on the size of input strings to show that a given DFA accepts precisely a certain language.
- Example: $M$ (right) accepts $L = 10^*$
- 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:
$$(\forall x \text{ s.t. } |x| = n) \ x \in L \iff x \in L(M) \Rightarrow$$
$$(\forall x \text{ s.t. } |x| = n+1) \ x \in L \iff x \in L(M)$$

Example proof of correctness

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

Proof of correctness:
- $Base$: (a) $\lambda \not\in L(A)$, because left state rejects;
  (b) $1 \in L(A)$, because middle accepts
- $Induction$: 1 takes accepted string to reject state; 0 takes any string to same state.
3. Nondeterministic finite automata

- What is nondeterminism?
- Does it matter if a process is nondeterministic?

Subtopic objective

2.3 Show expressiveness of finite automata**
Definition of NFA model

- 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
- \( \delta \) may have \( \lambda \)-transitions (i.e., 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 \( a \)

NFA example

<table>
<thead>
<tr>
<th>( \delta )</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>{q_0, q_1}</td>
<td>{q_3}</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>{q_3}</td>
<td>{q_1, q_2}</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>{q_3}</td>
<td>{q_3}</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>{q_3}</td>
<td>{q_3}</td>
</tr>
</tbody>
</table>

Note: state \( q_3 \) (dungeon) is implicit
**\(\lambda\)-closure**

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

---

**Uses for nondeterminism in theory work**

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

- *Theorem:* Every regular language is recognized by some NFA
- *Proof:* A DFA is by definition an NFA without nondeterministic transitions
- This is called an *immediate* or *trivial* proof

**RE → 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:* \((0 \mid 1)^* 1\) (strings that end in 1)
RE → NFA example

- RE: \((0 \mid 1)^*1\) (strings that end ‘1’)
- NFA:
  ![NFA Diagram]
- DFA:
  ![DFA Diagram]

NFA → DFA (subset construction)

- To construct a DFA \(M' = \langle Q', \Sigma, \delta', q_0', F' \rangle\) from an NFA \(M = \langle Q, \Sigma, \delta, q_0, F \rangle\) as follows:
  - let \(Q' = 2^Q\) (set of all subsets of \(Q\));
  - let \(q_0' = \{q_0\}\);
  - let \(F' = \) the set of states of \(M'\) that contain some state in \(F\)
- Then derive \(\delta'\) from \(\delta\):
  - merge all states with \(\lambda\)-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 some state in \(q\))
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.
- \( \lambda \)-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 → DFA example

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

- \( \delta(q_0, 0) = q_0 \)
- \( \delta(q_0, 1) = q_1 \)
- \( \delta(q_1, 1) = q_2 \)

DFA \( M' = \langle Q', \Sigma, \delta', q'_0, F' \rangle \), \( L(M') = 0^+1^+ \):

- \( \delta'({q_0}, 0) = \{ q_0, q_1 \} \)
- \( \delta'({q_0, q_1}, 1) = \{ q_1, q_2 \} \)
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 or expressiveness 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
2. Finite automata

Theorem: one accept state

- *Theorem:* For any DFA, there is an NFA with only one accepting state that recognizes the same language.
- *Proof:* Construct the NFA from the DFA, adding an accept state that has $\lambda$ transitions from each accepting state of the DFA; make the DFA’s accepting states non-accept in the NFA.

RLs: closed under concatenation

- For languages $L_1$ and $L_2$, the concatenation $L_1L_2$ is $\{xz | x \in L_1, z \in L_2\}$
- *Theorem:* $(L_1, L_2 \in RL) \Rightarrow (L_1L_2 \in RL)$
- *Proof:* DFAs $M_1$ and $M_2$ may be concatenated using $\lambda$-transitions as follows:

![Diagram of NFA concatenation](image)

- This NFA recognizes the languages of $M_1$ and $M_2$, concatenated, $L(M_1) L(M_2)$
RLs: closed under union and star

Theorem: For regular languages, $L$, $L_1$ and $L_2$

a. $(L_1 \mid L_2)$ is regular (union)

b. $(L)^*$ is regular (Kleene star)

Proofs:

RLs: closed under intersection

• Theorem: $\text{Reg}(L_1, L_2) \Rightarrow \text{Reg}(L_1 \cap L_2)$

• Proof:

1. $L_1 \cap L_2 = \sim(\sim L_1 \cup \sim L_2)$ (De Morgan’s Law)

2. Let $M_1$ be a DFA that accepts $\sim L_1$

3. Let $M_2$ be a DFA that accepts $\sim L_2$

4. An NFA, $M_3$, accepts $(\sim L_1 \cup \sim L_2)$ (see previous slide)

5. Let $M_4$ be an NFA that accepts the complement of the language of $M_3$

6. $M_4$ accepts $L_1 \cap L_2$
Reverses of RLs are regular

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

Proof: Construct an NFA:
1. Start with a one-accept-state DFA that accepts $L$
2. Reverse the directions of all transitions
3. Swap accepting and starting states
4. This NFA accepts $L^R$

4. Non-regular languages

- Are all languages regular?
- Can the DFA compute any predicate?
- What would we learn about the DFA model, if we found a non-regular language that a Java program could accept?
2. Finite automata

Subtopic objective

2.4 Prove that a language is not regular

The pumping lemma

- A property of infinite regular languages $L$: 
  
  $$(\exists n_0) \left( \forall x \text{ s.t. } |x| > n_0 \right) (x \in L) \Rightarrow 
  (\exists u, w \in \Sigma^*, v \in \Sigma^+)(x = uvw) \land (\forall k \in \mathbb{N})
  uv^kw \in L$$

- That is, any string in $L$ can be expressed as the concatenation of three strings, the middle of which can be “pumped” any number of times with the result also in the language

- Example: In $0(01)^*1$, a $(01)$ may be pumped

- Proof of lemma uses Pigeonhole Principle
Proof of Pumping Lemma

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 $uv^kw \in L$ for any $k \in \mathbb{N}$

Example: $0^n1^n$ is not regular

Theorem: If $L = 0^n1^n$ then $L \not\in \mathcal{RL}$

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

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

• What is a *lexicon*?
• What does a compiler do?
• How does one work?

Subtopic objective

2.5 Describe the use of DFAs in compilation
Regular expressions and UNIX

- Standard REs are extended by character classes, e.g., “.” (dot for any character); [a-z]; [abcd]
- UNIX search command `grep` (Global search for Regular Expression and Print) uses regular expressions
- UNIX has a lexical-analyzer generator, `lexx`

Lexical analysis

- Used by compilers, as part of syntactic analysis, to prepare for parsing
- AKA tokenization
- To recognize identifiers, numerals, operators, etc., implement a DFA in code
- State is stored in an integer variable, δ is implemented as 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

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 = (\_\_ | letter) (letter | digit | \_)*
- DFA accepts for letter or underscore, or same followed by letter, underscore, or digit
- **Pseudocode:**
  
  \[
  i \gets 0, \quad q \gets 0
  \]
  
  if next char is letter or underscore
  
  \[
  q \gets 1 \quad i \gets i + 1
  \]
  
  else
  
  \[
  q \gets 2
  \]
  
  while next character is letter, underscore, or digit
  
  \[
  i \gets i + 1
  \]
  
  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, IDs, operators, etc.
- These categories may be made an enumerated type:

  ```
  enum lex_categories {
    tkID, tkNum, tkStringLit, tkCharLit, tkLBrace, tkRBrace, 
    tkLParen, tkRParen, tkLBracket, tkRBracket, tkSemi, tkComma, 
    tkColon, tkPeriod, tkAddop, tkAsgnop, tkMulop, tkNot, tkOr, 
    tkAnd, tkRelop, tkScope, tkLess, tkGrtr, tkFinal, tkInt, 
    tkCase, tkChar, tkDouble, tkString, tkIf, tkSwitch, 
    tkWhile, tkFor, tkDo, tkElse, tkClass, tkPublic, tkEnum, 
    tkReturn, tkStatic, tkMain, tkVoid, };
  ```
References


See also *JFLAP* manual.