Finite Automata

Extra slide material for interested students. Not included in the regular course.

Why automata models?

- Automaton: Strongly limited computation model compared to ordinary computer programs
- A weak model (with many limitations)...
  - allows to do static analysis
    - e.g. on termination (decidable for finite automata)
    - which is not generally possible with a general computation model
  - is easy to implement in a general-purpose programming model
    - e.g. scanner generation/coding, parser generation/coding
    - source code generation from UML statecharts
- Generally, we are interested in the weakest machine model (automaton model) that is still able to recognize a class of languages.

Finite Automaton / Finite State Machine

- Given by quintuple \((\Sigma, S, s_0, F, \delta)\)

Computation of a Finite Automaton

- Initial configuration:
  - current state := start state \(s_0\)
  - read head points to first symbol of the input string
- 1 computation step:
  - read next input symbol, \(t\)
  - look up \(\delta\) for entry \((current\ state, t, new\ state)\)
  - current state := new state
  - move read head forward to next symbol on tape
  - if all symbols consumed and new state is a final state: accept and halt
  - otherwise repeat

NFA and DFA

- NFA (Nondeterministic Finite Automaton)
  - "empty moves" (reading \(\epsilon\)) with state change are possible, i.e. entries \((s_i, \epsilon, s_j)\) may exist in \(\delta\)
  - ambiguous state transitions are possible, i.e. entries \((s_i, t, s_j)\) and \((s_i, t, s_k)\) may exist in \(\delta\)
  - NFA accepts input string if there exists a computation (i.e., a sequence of state transitions) that leads to "accept and halt"

- DFA (Deterministic Finite Automaton)
  - No \(\epsilon\)-transitions, no ambiguous transitions \((\delta\ is\ a\ function)\)
  - Special case of a NFA

DFA Example

- DFA with Alphabet \(\Sigma = \{0, 1\}\)
  - State set \(S = \{s_0, s_1\}\)
  - Initial state: \(s_0\)
  - Final states: \(F = \{s_1\}\)
  - Transition function \(\delta\):
    \(\delta = \{(s_0, 0, s_0), (s_0, 1, s_1), (s_1, 0, s_1), (s_1, 1, s_0)\}\)
- recognizes (accepts) strings containing an odd number of 1s

Computation for input string 10110:

- \(s_0\) read 1
- \(s_1\) read 0
- \(s_0\) read 1
- \(s_1\) read 1
- \(s_1\) accept
From regular expression to code

4 Steps:
- For each regular expression \( r \) there exists a NFA that accepts \( L_r \) [Thompson 1968 - see whiteboard]
- For each NFA there exists a DFA accepting the same language
- For each DFA there exists a minimal DFA (min. #states) that accepts the same language
- From a DFA, equivalent source code can be generated. [
Theorem: For each regular expression \( r \) there exists an NFA that accepts \( L_r \) [Thompson 1968]

Proof: By induction, following the inductive construction of regular expressions

- Divide-and-conquer strategy to construct NFA(\( r \)):
  1. decompose \( r \) into its constituent subexpressions \( r_1, r_2, \ldots \)
  2. recursively construct NFA(\( r_1 \)), NFA(\( r_2 \)), ...
  3. compose these to NFA(\( r \)) according to decomposition of \( r \)

2 base cases:
- Case 1: \( r = \epsilon \):
  \[ \text{NFA}(\epsilon) = \]
  recognizes \( L(\epsilon) = \{ \epsilon \} \).
- Case 2: \( r = a \) for \( a \in \Sigma \): \[ \text{NFA}(a) = \]
  recognizes \( L(a) = \{ a \} \).

(continues)

4 recursive decomposition cases:
- Case 3: \( r = r_1 | r_2 \):
  By ind.-hyp. exist NFA(\( r_1 \)), NFA(\( r_2 \))
  \[ \text{NFA}(r) = \]
  recognizes \( L(r_1 | r_2) = L(r_1) U L(r_2) \)
- Case 4: \( r = r_1 . r_2 \):
  By ind.-hyp. exist NFA(\( r_1 \)), NFA(\( r_2 \))
  \[ \text{NFA}(r) = \]
  recognizes \( L(r_1 . r_2) = L(r_1) . L(r_2) \)
- Case 5: \( r = r_1^* \):
  By ind.-hyp. exists NFA(\( r_1 \))
  \[ \text{NFA}(r) = \]
  recognizes \( L(r_1^*) = (L(r_1))^* \).
  (similarly for \( r = r_1^+ \))
- Case 6: Parentheses: \( r = (r_1) \)
  \[ \text{NFA}(r) = \]
  (no modifications).

The theorem follows by induction.