Finite State Machines: Motivating Examples

Greg Plaxton
Theory in Programming Practice, Fall 2005
Department of Computer Science
University of Texas at Austin

The Wolf-Goat-Cabbage Puzzle

- A shepherd arrives at the left bank of a river with a wolf, a goat, and a cabbage
- There is a boat that can carry the shepherd and at most one other item
- The shepherd cannot leave the wolf and goat together unattended
- The shepherd cannot leave the goat and cabbage together unattended
- Can the shepherd cross the river and bring along the wolf, goat, and cabbage?
- More generally, what approach can be used to solve puzzles of this kind?

Wolf-Goat-Cabbage: Key Insight

- We can model the wolf-goat-cabbage puzzle (in various ways) as a system with a finite number of "admissible states"
  - The initial state is an example of an admissible state
  - Any state in which the goat and cabbage are left together unattended is inadmissible
- For each admissible state s, some number of admissible states are reachable in "one step" from s by executing some "primitive action"
  - Each primitive action needs to be simple enough that it is clear how to carry it out; for example, we would not consider moving everything on the left bank to the right bank to be a primitive action
  - A sequence of primitive actions is admissible if, starting from the initial state, it visits only admissible states
  - The set of primitive actions must be rich enough so that any strategy for the puzzle can be modeled as an admissible sequence of primitive actions

Wolf-Goat-Cabbage: A Finite Model

- Note that there is an interplay between the choice of state space and the choice of the set of primitive actions
- For example, if we model the position of the boat as being on the left bank, on the right bank, or in the middle of the river, then we would probably not want to consider moving the boat from the left bank to the right bank to be a primitive action
  - Rather, the boat would move from the left to the middle, and then from the middle to the right
- It is generally beneficial to choose the state space and associated set of primitive actions so that they are as small as possible
  - But recall that it should be clear how to carry out each primitive action within the system we intend to model
- What are some examples of such finite models for the wolf-goat-cabbage puzzle?

Wolf-Goat-Cabbage: A Finite State Machine

- The “admissible states” referred to previously correspond to the states of the FSM
  - In a diagram of an FSM, each such state is depicted as a circle labeled with some encoding of the state
- The set of primitive actions that can be applied to a given admissible state correspond to the set of transitions associated with that state
  - Each such transition is depicted by an arrow between states, labeled with some encoding of the associated primitive action
- The FSM has a distinguished start state (with an arrow pointing to it labeled “start”)
  - In this case, it is the state in which everything is on the left bank
- The FSM has a set of accepting states (indicated by a double circle)
  - In this case, there is one accepting state in which everything is on the right bank

A Traffic Light

- Many everyday artifacts can be viewed as finite state machines
- Normally a traffic light changes from green to yellow, from yellow to red, from red to green, et cetera
- Assume that when an emergency vehicle such as an ambulance needs to cross through the intersection, it can change the light to red regardless of the current state
- How can we model such a traffic light as an FSM?

Pattern Matching

- Suppose we want to write a program to recognize whether a given string satisfies a certain property, for example:
  - Whether the five vowels — a, e, i, o, u — appear in alphabetical order within the string (e.g., "facetious" or "sacrilegious", but not "tenacious")
  - Whether the string denotes an integer
  - Whether the string denotes a floating point number
- Each of the above problems can be solved by designing a suitable FSM
  - To determine whether a string is accepted by an FSM, we use the string to determine the sequence of transitions to perform (beginning from the start state), and check whether we end up in an accepting state

Lexical Analysis

- A more elaborate version of the kind of pattern matching problems discussed on the preceding slide is addressed within the lexical analyzer of a compiler
- The input to the lexical analyzer is the source code, and the output is a stream of "tokens"
  - White space and comments are eliminated
  - Each keyword or other predefined string (such as arithmetic and logical operators) is reduced to a corresponding token
  - All occurrences of a given identifier or constant are mapped to the same token (unique to that identifier/constant)
- Direct implementation of a lexical analyzer is tedious and error-prone
- In part by exploiting the connection to FSMs, tools (e.g., lex, flex) have been developed that greatly simplify the task of creating a lexical analyzer for a given language