6.005 Elements of Software Construction
Fall 2008

6.005 Elements of Software Construction | Fall 2008
Problem Set 1: The State Machine Paradigm

The purpose of this problem set is to give you practice in the basic techniques of the state machine paradigm. You'll construct state transition diagrams and grammars for a variety of small problems, and code them using standard patterns.

State Machine Reasoning

Milk and Tea Puzzle. Here's a well-known puzzle. You have a glass of tea and an equal-sized glass of milk. You pour some milk into the tea, mix it up throughly, and then transfer the same amount of liquid back to the milk. This process is repeated some number of times. Is there now more milk in the tea or more tea in the milk?

The answer is that there is exactly the same amount. To prove this, (1) construct a state machine to model the problem; (2) define an invariant; and (3) show that the invariant is true in the initial state and is preserved by the state machine's actions.

Hint: Treat each glass as consisting of a discrete number of particles, and define an action that does to-and-fro transfers in a single step. Also, note that, as specified, the machine is already non-deterministic, since it permits an arbitrary quantity of milk to be poured initially. Can you make it even more non-deterministic and still preserve the invariant?

State Diagram Modeling

Three-way Light. A 'three-way' lighting circuit consists of two switches and a single lamp; toggling either switch toggles the state of the lamp. Draw a state machine representing the behavior of this system, with events `up_1` and `down_1` for moving switch 1 up and down (and analogously for switch 2). Show the state of the lamp by marking the states of the machine appropriately.

Cruise Control. Consider a cruise-control system with these events: `on` (turns the system on); `set` (records the current speed and attempts to maintain it); `brake` (stops attempting to maintain speed); `resume` (attempts to return to and maintain last recorded speed); and `off`

(turns the system off and clears any memory of a recorded speed). Draw a state transition diagram that shows which event sequences are accepted (assuming that on is accepted only when the system is off, `resume` only when the system is on, etc), and which of three modes the system is in: `OFF` (not operating); `ON` (operating but not actively controlling speed); and `ACTIVE` (actively controlling speed). A mode may correspond to one or more states. Then, using this diagram as a basis, construct a second diagram that includes three additional events that are produced as outputs: save (which saves the current speed to a register), `control` (commanding the engine to maintain the speed in the register) and `relinquish` (relinquishing control of the engine to the driver).

Alternating Bit Protocol. One way to compensate for a lossy communications channel is to have the receiver acknowledge each received message and have the sender retransmit if no acknowledgment is obtained. But if an acknowledgment arrives late, and a resend has already occurred, messages and their acknowledgments get out of step. To avoid this, you can attach a sequence number to each message so that the receiver can indicate which message is being acknowledged. The simplest version of this scheme is the *Alternating Bit Protocol* which uses sequence numbers of just one bit. Here's how it works. The sender marks outgoing messages with a zero, then a one, then a zero, etc. The receiver acknowledges each message with the same bit that the message was marked with. If either the sender or the receiver gets a message with the wrong bit, it resends its previous message. The sender also resends a message if it receives no acknowledgment at all after some unspecified elapsed time. Model this protocol as two state transition diagrams, one for the sender and one for the receiver, accompanied by a list of events and their definitions. Draw a third diagram to model the behavior of an unreliable channel between them that can hold up to two messages at a time, and can drop but never corrupt or reorder messages.

Grammar Modeling

Multi-Unit Calculator. It's often convenient to use different units in the same computation. For example, to figure out how many lines of twelve-point type fit in a six-inch column, it would be nice to have a calculator that accepted the expression `"6in/12pt"`. Construct two grammars for such a language: first, a lexical grammar that breaks the sequence of characters into numbers, unit specifiers, operators, and left/right parentheses, filtering out spaces and tabs (but not requiring them as delimiters); and second, a syntactic grammar that groups expressions appropriately. Your grammar should allow nesting of expressions using parentheses (eg, `"1in + (2in + 3in)"`), and it should allow unit conversions (eg, `"30pt in"` to show 30 points in inches). *Note*: You need only handle inches and points, and you can assume that every expression denotes a length or a dimensionless scalar (and not, for example, a product of lengths). There are 12 points in a pica, 6 picas in an inch, and thus 72

points in an inch.

Cruise Control. Model the cruise control system described above but using grammars instead of state transition diagrams.

Implementing with Patterns

Multi-unit Calculator. Implement the multi-unit calculator described above, with a class whose main method takes a single string as an argument and returns a string as a result. You should split your program into two separate phases, one for 'lexing' (breaking the string into lexical tokens), one for 'parsing' (grouping the tokens syntactically). Your code for these phases should correspond directly to the two grammars you designed. Provide at least three JUnit test cases for your code. *Hint*: You can perform evaluation as part of the parsing phase or in a separate subsequent phase.

Alternating Bit Protocol. Implement the alternating bit protocol as described above, including the unreliable communications channel. Each of the three components should be implemented as (at least) one class with a method `step` which, when called, causes the component to take a step -- such as sending, receiving or transmitting a message. Provide at least three JUnit test cases for your code that include a case in which the protocol handles dropped messages.

Infrastructure

No code is provided for this problem set. A directory called `pset1` will be created for you in your personal repository, containing a copy of this file. In addition to your code and test cases, you should commit your solutions to the exercises as a single PDF file.