CSE 30151 Spring 2014
Thursday, April 17th, 2014
Nicholas LaRosa and Eric Krakowiak

Project 3 Report

This project was completed in the Python programming language (version 2.6) and made executable via a small shell script. All testing was performed on the Fitzpatrick Linux machines with various test input. Debugging was completed using simple print() statements to monitor the state of variables and data structures within member functions and object instances.

This program was approached from an object-oriented standpoint; a Turing Machine is represented by an object (defined in TM.py) that contains a list of states, a list of input symbols (input alphabet), a list of tape symbols (tape alphabet), a list of dictionaries containing transition rules, a string containing the start state, a string containing an accept state, and finally, a string containing a reject state. These data structures make up the definition of the TM, and are initialized via the object member functions addStates(), addInputAlphabet(), addTapeAlphabet(), addTransition(), addStartState(), and addAccRejStates(). These functions are implemented in such a way that a TM's input and tape alphabets and state list must be established before any transitions, accept/reject states, or the start state can be defined. This is because transitions can only occur from valid states on valid input and tape alphabet symbols, and the start, accept, and reject states must be contained within the state list. Additionally, the only two possible symbols for tape head direction were 'L' for left and 'R' for right.

Once the user has instantiated a TM object, they begin defining the TM using the functions mentioned above. With the state list and the input and tape alphabets being contained within Python lists, lookup and access is made easy due to the fact that any type of variable can be appended to a Python list. Slightly more complicated is the data structure required to contain all transition rules. Because a transition occurs from a start state on a tape symbol to result in another state (at which point the tape symbol is replaced with another) and the tape head lastly moves left or right, each transition was contained within a clearly defined dictionary. For each transition, a dictionary contains the clearly-labeled keys 'startState', 'currTapeSymbol', 'resultState', 'writTapeSymbol', and 'headDirection', with corresponding values defined by the transition configuration line. Although representing the transitions in this way may not be the most efficient, it allows for ease-of-programming.

For example, it would have been more efficient to represent all transitions within a single dictionary keyed by start state. The corresponding values could then have been other dictionaries keyed by tape symbol, with values being a list containing the result state, the tape symbol to be written, and the tape head direction. This definitely would have allowed for quicker access of transitions on the same start state and same tape symbol (instead of having to search the entire list of transition dictionaries when these transitions are desired). However, this simple and clear approach provided a constant reminder of what transition values were being compared during the programming process, which turned out to be valuable in terms of debugging and will hopefully be helpful for those reviewing our code.

After the user has provided a TM configuration file and the mentioned functions have established the state list, input and tape alphabets, transition list, start state, accept state, and reject state, it is necessary to verify that the given configuration is indeed deterministic. Firstly, we know that no duplicate transitions exist due to our checking against the list of transitions for every new transition pending. Thus, the TM verification involves the function testTransitions(), being called after all transitions have been added from the configuration. The function returns true if no transitions on the same start state and tape symbol exist, meaning that the path at each state is deterministic, so we can ensure that the configuration file contains a deterministic TM before we begin accepting input.

Per project guidelines, the TM definition is provided via an input file. Thus, the processFile() member function will read an input file and define the corresponding TM instance using the previously mentioned add…() functions. Once this is complete, the user can have the DPDA accept tape input via the getInput() function. Taking the number of tape inputs as its first input from standard out, the getInput() will subsequently call processInputLine() for that provided number. This function keeps track of the current state after each input by attempting a transition on the current start state and tape symbol. The tape, tape head, and current state is modified accordingly if a transition is found, otherwise the reject state is implicitly entered and the TM stops execution. If the accept state is entered, the TM also stops execution. The tape was implemented via a Python list and the pop and append (ie. push) functions. Lastly, if over 1,000 transitions are taken without entered either the accept or reject states, the TM stops execution and a DID NOT HALT flag appears. No recursion is necessary in the case of a TM because only one state path is possible.

Test programs:	**Tape inputs:**
tests/tm1.txt	tests/tm1-input.txt
tests/tm2.txt	tests/tm2-input1.txt
	tests/tm2-input2.txt

Usage:

```
./tm tm-definition.txt < tm-input.txt > tm-output.txt
```

Turing Machine Programs

Machine 1: Integer Division

L = { $a^i b^j c^k$ | |k| = |i| / |j| and i, j, k ε Z }

This Turing Machine computes integer division on the number of a's divided by the number of b's. The answer is the number of c's, with any remainder truncated.

Computations can be seen by loading the configuration in **machines/division.txt** and inputting a tape of only a's and b's. The resulting stack with contain the result of the division as the number of c's.

Machine 2: String Comparison

L = { x#y | x, y ε {0,1}* and x = y }

This Turing Machine performs a string comparison between two strings separated by a hash, accepting strings that contain the two similar hash-separated strings.

Comparisons can be seen by loading the configuration in **machines/strcmp.txt** and inputting a tape of two strings containing 0s and 1s separated by a hash. If the result is ACCEPT, the strings are the same. Otherwise, they are unique.

Machine 3: Bitwise AND

L = { x#y#z | x, y, z ε {0,1}* and z = x & y }

This Turing Machine performs a bitwise AND between the first two hash-separated bit strings, accepting strings that contain three hash-separated bit strings, with the third string being the result of a bitwise AND between the first and second bit strings.

Computations can be seen by loading the configuration in **machines/bitwiseAND.txt** and inputting a tape of two bit strings, each followed by a hash. Upon execution, the machine will accept and contain the resulting bit string as a third hash-separated string in the tape (to the right). However, if the two bit strings are of different length, the machine rejects and does not continue the computation.