CSE 30151 Spring 2014
Sunday March 30th, 2014
Nicholas LaRosa

**Project 2 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 student00.cse.nd.edu machine with various test input. Debugging was completed using simple print() statements to monitor that state of variables and data structures within member functions and object instances.

This program was approached from an object-oriented standpoint; a DPDA is represented by an object (defined in DPDA.py) that contains a list of states, a list of input symbols (input alphabet), a list of stack symbols (stack alphabet), a list of dictionaries containing transition rules, a string containing the start state, and finally, a list of accept states. These data structures make up the definition of the DPDA, and are initialized via the object member functions addStates(), addInputAlphabet(), addStackAlphabet(), addTransition(), addStartState(), and addAcceptStates(). These functions are implemented in such a way that a DPDA's input and stack alphabets and state list must be established before any transitions, accept states, or the start state can be defined. This is because transitions can only occur from valid states on valid input and stack alphabet symbols, and the start state and accept states must be contained within the state list.

Once the user has instantiated a DPDA object, they begin defining the DPDA using the functions mentioned above. With the state list, input and stack alphabets, and accept states 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 an input and stack alphabet symbol to result in another state (at which a stack symbol may be pushed), each transition was contained within a clearly defined dictionary. For each transition, a dictionary contains the clearly-labeled keys 'startState', 'inputSymbol', 'stackSymbolPop', 'resultState', and 'stackSymbolPush', 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, something that my first project lacked.

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 input symbol, with values being dictionaries keyed by popped stack symbol, with values being a list containing the result state and stack symbol to be pushed. This definitely would have allowed for quicker access of transitions on the same start state, same input symbol, and/or same stack 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 my code.

After the user has provided a DPDA configuration file and the mentioned functions have established the state list, input and stack alphabets, transition list, start state, and accept states, it is necessary to verify that the given configuration is indeed deterministic (after removing any duplicate transitions via removeDuplicateTransitions()). This DPDA verification involves the functions testDPDA_Rule1(), testDPDA_Rule2(), testDPDA_Rule3(), and testDPDA_Rule4(), invoked simultaneously via isValidTransition(), being called on each transition. The isValidTransition() function returns any return value from any of the four testing functions, with a non-zero value corresponding to the rule violated. As expected, these four functions test the four requirements for a DPDA outlined in lecture: (1) there can be at most one transition on a given input character and stack character, (2) if a transition doesn't read the stack, all others cannot read (the stack but not input) and other transitions cannot read the same input and the stack, (3) if a transition reads the stack but not input, other transitions cannot read (input but not stack) and other transitions that read the same stack symbol cannot read input, and (4) if a transition reads no input and no stack, no other transitions are allowed. By calling these functions with all transitions compared to all other transitions that occur on the same start state, the program successfully detects determinism in all test cases.

Per project guidelines, the DPDA definition is provided via an input file. Thus, the processFile() member function will read an input file and define the corresponding DPDA 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 first attempting a transition on the empty string, and if no transition occurs, next attempting a transition on the input symbol. The stack was implemented via a Python list and the pop and append (ie. push) functions. If no such transition exists, the machine cannot proceed and the input is rejected. On the other hand, if all inputs result in a transition and the final state is an accept state, the input is accepted. Accordingly, if the final state is not an accept state, the input is rejected. No recursion is necessary in the case of a DPDA because only one state path is possible.

| **Test programs:** | **Tape inputs:** |
|---|---|
| tests/dpda1.txt | tests/dpda-input1.txt |
| tests/dpda4.txt | tests/dpda-input4.txt |

**Usage:**

./dpda dpda-definition.txt < dpda-input.txt > dpda-output.txt