CS-4700/5700 Automata, Computability, and Formal Languages
PJ04.docx

Pushdown Automaton Simulator

In this project you will write a program that will read a simplified description of a pushdown automaton, validate it, and then simulate it on each string read from an input text file. Each string that is accepted by the PFA will be echoed to a text file; in addition each machine will have a log file prepared containing specific pieces of information.

Undergraduate students are only responsible for being able to process deterministic machine descriptions, while graduate students must also be able to process nondeterministic machine descriptions. However, all students will receive the same set of machines, so undergraduates will need to be able to determine whether a machine is deterministic or not.

To avoid operating-specific case-sensitivity issues, all filenames will be all lowercase. So that all of the files can be in the same directory, the following naming conventions will be used:

Base name: mxx where xx is a two digit number. The first machine will have a base name of m00 and the remaining machines will be numbered sequentially. This should allow you to write a wrapper program that processes all of the machines in a single run.

Machine description file: basename.pda

Accepted strings: basename.txt

Log file: basename.log

Input file: strings.txt (the same file for all machines)

Machine Description File – basename.fa

Recall that the formal description of a PDA is $M = \{Q, \Sigma, \Gamma, \delta, q_0, F\}$.

Q: States

- The “official” set of states are 0 through 255, inclusive.
- The “nominal” set of states are those states that are reachable from the start state.
- The states need not be numbered sequentially.
- State 255 is an inferred non-accepting trap state.

Σ: Input alphabet

- The “official” input alphabet consists of all of the printing characters (as defined by the C standard library function isprint()) except the grave accent (a.k.a. back tick) character (`), which serves as an ‘epsilon’ character. Note that the ASCII code for the back tick is

96 (0x60). Also note that the space character (ASCII code 32 (0x20)) IS a printing character.

- The "nominal" input alphabet for the machine consists of those characters from the official input alphabet for which at least one transition rule is defined.

Γ: Stack alphabet

- For convenience, the "official" stack alphabet is the same as the input alphabet.
- The "nominal" stack alphabet for the machine consists of those characters from the official stack alphabet for which at least one transition rule is defined.

δ: Transition function ($Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow Q \times \Gamma_\varepsilon$)

- If a transition rules is not defined for a given combination of state, input character, and stack character, then it is inferred that the transition is to the non-accepting trap state (state 255).
- Recall that attempting to read an empty stack is not allowed, thus this equates to a transition to the non-accepting trapstate.

q_0: Start State

- The start state is state 0.

F: Accept states

- This is an explicit set of states.

With this common framework, the description file contents are as follows:

Line #1: Curly-brace enclosed, comma-separated list of accept states.

Line #2+: One transition rule per line in the form:

state, input symbol, pop symbol, new_state, push symbol

Except as noted here, there will not be any whitespace in the file.

- The end of each line is denoted by a newline sequence, which could be either a LF or a CR/LF pair, depending on the operating system used. Your program needs to accommodate both possibilities.
- Since the space character is in the input and stack alphabets, it might be included in the transition rules as a literal character.

CS-4700/5700 Automata, Computability, and Formal Languages
PJ04.docx

A nondeterministic machine description has multiple transition rules for the same state/input/stack combination. Note that the mere presence of an epsilon in a transition rule does NOT necessarily make the machine nondeterministic; nondeterminism is present only if there is at least one point in the machine where at least two distinct options exist for going forward.

Upon reading the machine description file, your program should validate it. There are three possible outcomes from this process: DPDA, NPDA, INVALID. Since the states and alphabet are inferred from the transitions, there are only a limited number of things that can be checked.

- First, the list of accept states should consist of zero or more valid states (namely non-negative integers strictly less than 255).
- Second, each of the transition rules should be valid.

If there are no problems with the format, then the machine is a DPDA provided it has only one transition (including inferred transitions) for each combination of state, input character, and stack character. Should a machine have the same transition rule repeated, you may choose to ignore all but one of them, or you may classify the machine as an NPDA, or you may declare the machine as invalid. This is so that you can essentially let your program deal with this situation in whatever way is natural for it – you can take away from this that such a machine description file will not be included deliberately (but past machine description files have done so inadvertently).

Undergraduates are required to properly identify NPDA descriptions, but do not need to simulate them.

CS-4700/5700 Automata, Computability, and Formal Languages
PJ04.docx

String Input File – strings.txt

There will be a single input file, named strings.txt, that will be run against each machine. This is a simple 8-bit ASCII text file (NOT Unicode). Each line contained in the file is a separate string. The terminating end-of-line (EOL) sequence is NOT part of the string. The EOL sequence could use either the DOS/Windows convention (CR/LF) or the UNIX convention (LF), your program must be able to process either.

There is no limit on the length of a string. Blank lines are the empty string. ALL strings are EOL-terminated, including the last one in the file.

Output File – basename.txt

The strings that are accepted should be echoed to the output file and should be identical to the input string – meaning in particular no missing/extra spaces before or after the string – and should be terminated with an EOL terminator (either CRLF or just LF – you may use whatever your system produces naturally) hence, when opened in a text editor, the cursor should be able to go to the beginning of the line after the last string, but this line should be empty. An empty string is therefore a line containing only and EOL terminator.

Log File – basename.log

After reading the machine description file and processing the input file, the program should log the following information to an ASCII text file (in the following order, one item per line).

Valid: DFA | NFA | INVALID

States: n

This number should include all states that have explicit transitions to/from them plus state 255 only if it has any transitions to it from a used state for any member of the alphabet.

Input Alphabet: abcd

The nominal machine input alphabet, in ASCII order, without delimiters or escape characters of any kind. Note that exactly one space follows the colon after the label. This space is NOT part of the alphabet. Also note that epsilon is NEVER an element in the input alphabet!

Stack Alphabet: abcd

The nominal machine stack alphabet, in ASCII order, without delimiters or escape characters of any kind. Note that exactly one space follows the colon after the label. This space is NOT part of the alphabet. Also note that epsilon is NEVER an element in the stack alphabet!

Accepted Strings: a / m (where a = number accepted; and p = number of strings in input file)

The last lines should be "0 / 0" for any invalid machine (no need to even attempt running the machine) or, for undergraduates, for NFA machines unless you choose to support them.

SUBMISSION

You need to submit a single ZIP file. At the top level should be a single-spaced report (in PDF format) that is no more than five pages long (two to three is preferred). This report should focus on describing the technical approach taken in implementing the simulator. Included in this description should be a short section indicating the programming language and/or development environment that you used and any special configuration settings needed by someone that would like to reproduce your program as well as how to run your program.

There should also be two folders at the top level. The first is named "results" and should contain your output and log files for all of the machines. It does not need to have any other files, including the machine description files or the input file. There should also be a folder named "code" that contains all of the source code for your program. Included should be all files needed by someone to reproduce your program assuming they have the appropriate development environment. This folder can contain subfolders as appropriate.

Keep in mind that the grader is very unlikely to even attempt to run your program – they likely will not know the language you used, let alone have the necessary tools. Thus it is important that your report file describe your approach and algorithm very clearly AND that your code be well organized and commented – the grader WILL review your code.

CS-4700/5700 Automata, Computability, and Formal Languages
PJ04.docx

GRADING CRITERIA

Report/Code: 50%

- 70% Technical merit – algorithm is well explained.
- 10% Programming environment details well explained.
- 10% Code is well organized
- 10% Code is well documented

Machines: 50%

Each machine will count equally toward this portion of the grade.

For EACH machine:

- -20% for mischaracterizing the machine type.
- -5% for each state different than the correct number of states
- -5% for each nominal alphabet symbol in error (either missing or extra)
- -1% for each string acceptance error (either falsely accepted or falsely rejected)

If the alphabet or strings are not in the correct order, this can result in a very low score.

IMPLEMENTATION HINTS

Implementation of a deterministic machine is mostly a straightforward extension of the implementation of a deterministic finite automaton with only the need to implement a stack data structure. However, even a DPDA is allowed to have epsilon transitions and therefore it is possible for such a machine to loop and never reach the end of the input string. For instance, a machine might have a transition rule that does not look at the input string and merely pushes a character onto the stack. In general, detecting that a machine is stuck in an infinite loop is not possible. For our purposes, it is reasonable to impose an upper limit on the number of transition rules applied and declare (with some chance of being wrong) that any machine that exceeds that limit is in a loop and that does not accept the string. For this project, a limit of ten thousand rules should be adequate.

For a non-deterministic machine, the approach taken for a non-deterministic finite automaton is a good starting point provided that it is kept in mind that the stack needs to be cloned as part of the process. The natural data structure to use for the simulation is a queue of machines. At each step the next machine is removed from the queue and clones are made for each available transition rule. The clones are added to the queue and the machine that was removed is discarded.

Remember that the normal way to process a machine is to follow all transitions that read from the input string and then optionally apply any available transitions that don't. For this to work, we also apply any epsilon transitions from the start state before we begin processing the input string at all. When doing this by hand, we generally follow all available epsilon transitions until we determine all of the possible states that we might reach after reaching the next input character. However, if the machine contains an epsilon loop, then could easily get stuck if we take this approach and, unlike with an NFA, it is NOT sufficient to mark the reached states since two machines in the same state are not equivalent unless their stacks are identical, too. Instead, we can take a much simpler approach and simply apply all possible transitions, epsilon and non-epsilon alike, place the resulting clones back in the queue. Each machine should have a lifetime counter that is updated and when that counter reaches the appropriate value the machine is simply discarded.

CLONING A MACHINE

If you are taking an object-oriented approach to the project then you can view a machine as an object that has static elements (the transition function definition and the set of accept states, for instance) and instance elements which consist of the current input string, the current state, and the current stack. We can also include a time-to-live (TTL) counter as an instance data element.

When we create a machine we need to supply all of these elements. For the initial machine the input string is the string read from the input file, the current state is the start state, the current stack is an empty stack, and the TTL value is the maximum number of steps.

When we process a machine we can take the current state and fetch a list of all of the transitions defined for that state (if there aren't any, then that instance of the machine simply dies). We can then walk down that list seeing which, if any, apply to the current machine (based on the next symbol in the input string and the symbol at the top of the stack). Each time one applies, we clone the machine, apply the transition, and add it to the queue. One way to do this is to create a new machine in which the input string is either the current input string or one in which the first character is removed, depending on whether the current rule is an epsilon transition or not, the current state is the new state, the new stack is the current stack as modified by the transition rule's pop/push content, and the TTL value is the current value decremented by one.