RW771 Tutorial 1

Shaun Schreiber
16715128

9 February 2014

1 Problem Description

Write a program that can parse and simulate any DFA that is in the correct EBNF[1], the EBNF is show in the last section. The parser needs to be able to do the following: detect incorrect format, detect inconsistencies and extract a DFA 5-tuple[2]. The inconsistencies that should be detected are as follow. The transition function specifies a mapping between two states and one of those states do not exist, when a symbol is used to signal a transition between two states that is not part of the specified alphabet. The program needs to take in two arguments. The first argument specifies the file where the DFA is located and the second argument specifies the input to that DFA. When a DFA executes the input and it finishes in an accept state the word "accept" should be printed else the word "reject" should be printed. If any error occurs, the correct error message should be printed, followed by the word "reject".

2 Design and Implementation

The program design is divide up into 3 section. A scanner, parser and simulator.

2.1 Scanner

The task of this section is to build up tokens that is recognized by the specified EBNF. This will be achieved in the follow way.

- Read a character.
- Is the character a letter? If so build up a string of characters until a white space is encountered and start again. If not continue to the next step.
- Is the character a number? If so create a symbol token that has the value of the number and start again.
- Is the character one of the following symbols, { } = ( ) - and ,
- In the case where a - symbol is read, read and check if the next symbol is a > sign if so, create the appropriate token and start again else throw an error.

[1] Extended Backus-Naur Form
[2] (states, alphabet, transition function, starting state, accepting states)

- If the EOF character is read, stop test if the current symbol is valid. If so, create the appropriate token and stop else if not error.
- If the character is not identified stop processing the file and throw an error.

2.2 Parser

The task of this section is to Analyse the syntax of the specified DFA to make sure it is in the correct format and extract the DFA 5-tuple. This is achieved by implementing a recursive descend parser in the following manner. Variables[3] become functions. The left-hand side of the variables are then checked per symbol. If the left-hand side contains a variable , the appropriate function must be called. The 5-tuple that a DFA consists of is described in the first production[4] of the EBNF thus as each part of the top production is finished. The result can be stored into the corresponding variable.

2.3 Simulator

The task of this section is to simulate a DFA. This is achieve by using the following algorithm.

- `Step 1` set currentState = startingState
- `Step 2` Read a character from the input and put it equal to currentCharacter.
- `Step 3` Concatenate the currentState and the currentCharacter to form a key.
- `Step 4` set currentState = hashmap(key). If there are no more characters goto step 5 else got to step 2.
- `Step 5` Check if the currentState is indeed a accepting state if so display "accept" else "reject".

3 Test Cases

3.1 Test Case 1

Test: This Test case tests if the starting state is not the final state and there is no input string.
DFA:automaton = ({q1,q2}, {0, 1},{(q1,0) -> q1, (q1,1) -> q2}, q1,{q2})
Input String:
Result:reject

[3]The words to the left of the equals sign that doesn't equate to only one token.
[4]A production looks like a mathematical equation

3.2 Test Case 2

Test: This Test case tests if the starting state is the final state and there is no input string.
DFA:automaton = ({q1,q2}, {0, 1},{(q1,0) -> q1, (q1,1) -> q2}, q2,{q2})
Input String:
Result:accept

3.3 Test Case 3

Test: This Test case tests if the solution can handle more than two states.
DFA:automaton = ({s,q1,q2,r1,r2}, {a, b},{(s,a) -> q1, (s,b) -> r1, (q1,a) -> q1, (q1,b) -> q2, (q2,b) -> q2, (q2,a) -> q1, (r1,b) -> r1 , (r1,a) -> r2, (r2,a) -> r2, (r1,b) -> r1}, s,{q1, r1})
Input String:bbaabab
Result:accept

3.4 Test Case 4

Test: This Test case tests to see if the program gives the correct error when symbols that are not in the alphabet are used.
DFA:automaton = ({q1,q2}, {0, 1},{(q1,a) -> q1, (q1,1) -> q2}, q2,{q2})
Input String: 100101
Result(s):IllegalDFAFormatException
reject

3.5 Test Case 5

Test: This Test case tests if the program detects an error when a state is used that wasn't defined.
DFA:automaton = ({q1,q2}, {0, 1},{(I,2) -> q1, (q1,1) -> q2}, q2,{q2})
Input String: 100101
Result(s):IllegalDFAFormatException
reject

3.6 Test Case 6

Test: Test if the program gives the correct error when an invalid string is given as input. DFA:automaton = ({q1,q2}, {0, 1},{(q1,0) -> q1, (q1,1) -> q2}, q2,{q2})
Input String:123
Result:IllegalInputException: 2
reject

3.7 Test Case 7

Test: Test if the program picks up invalid state names. DFA:automaton = ({q1,q2}, {0, 1},{(q1,0) -> q1, (q1,1) -> q2}, q2,{q2})
Input String: 10010101
Result:Expected: ID But found: SYMBOL
IllegalDFAFormatException
reject

4 DFA EBNF

$$
\begin{aligned}
automaton &= \text{“}DFA\text{”}\ \text{“}=\text{”}\ \text{“(”}states\text{“,”}alphabet\text{“,”}tfunction\text{“,”}start\text{“,”}accept\text{“)”}. \\
states &= idset. \\
alphabet &= \text{“\{”}symbol\{\text{“,”}symbol\}\text{“\}”}. \\
start &= id. \\
tfunction &= \text{“\{”}map\{\text{“,”}map\}\text{“\}”}. \\
map &= \text{“(”}id\text{“,”}symbol\text{“)”}\ \text{“}->\text{”}id. \\
accept &= idset. \\
idset &= \text{“\{”}id\{\text{“,”}id\}\text{“\}”}. \\
id &= letter\{letter|digit\}. \\
symbol &= letter|digit. \\
letter &= \text{“}a\text{”}|...|\text{“}z\text{”}|\text{“}A\text{”}|...|\text{“}Z\text{”}. \\
digit &= \text{“0”}|...|\text{“9”}.
\end{aligned}
\tag{1}
$$