Computer Science 711
Tutorial 1

Arné Esterhuizen
15367940

13 February 2011

Abstract

This report deals with the implementation of a simulator for a deterministic finite automaton (DFA).

1 Introduction

A DFA is a 5-tuple consisting of a set of states, an alphabet, a transition function, a start state, and a set of accept states. For this tutorial, a DFA simulator was implemented. The simulator reads an input file containing the specifications of a DFA. Having done that, the simulator then runs the specified DFA on an input string and either accepts or rejects it. Section 2 deals with the DFA simulator's general design, whilst Section 3 deals with details of its implementation.

2 DFA Simulator

The DFA simulator must be able to read the specifications of a DFA from an input file. To enable it to do so, the explanations and examples as provided by the practical notes were closely followed. The simulator loosely consists of three parts, a scanner, a parser, and the simulator itself. The scanner reads characters from the input file and uses these to form tokens. The parser uses the tokens obtained from the scanner and ensures that they specify a valid DFA. The rules used by the parser to validate the DFA are given in the practical notes in the form of an EBNF. The design of the parser closely mimics the structure of this EBNF. Whilst analysing the tokens obtained from the scanner, the parser builds up the necessary datastructures needed by the simulator to run the DFA on an input string. The transition function is represented internally using a transition matrix.

3 Implementation

The DFA simulator was implemented in Python. It requires no external libraries and was tested on both Windows and Linux machines.

3.1 Scanner

The scanner was implemented as a class that must be instantiated. The parser creates a scanner object with the input file as a parameter, and calls the scanner object's `get_token()` method in order to obtain the next token. The scanner itself only reads a single character at a time from the input file, and skips over whitespace characters. The scanner also keeps track of the line number of the input file to help find errors, should there be any. The scanner will raise an exception if it encounters any unexpected characters.

3.2 Tokens

The scanner returns tokens to the parser, and a token is also implemented as a class. A token has three fields: the token type, a number value if it is a digit, and a string if it is text. The type field is always set and describes what type of token this is. These types are defined as enums in `tokens.py`, along with the Token class. Python does not actually have support for enums, but a work around for this is to simply define a class that acts like enums.

3.3 Parser

The parser and simulator are not easily separable as they both use the same data structures. The parser builds the data structures, while the simulator uses the information stored in these data structures to simulate the DFA. They are thus implemented together in `dfa_simulator.py`. The parser will raise an exception if it encounters an illegal DFA specification.

3.4 Data Structures

The parser stores all the states of the DFA in a dictionary which maps state names (keys) to integers (values), starting at zero. This is so that state names from the input strings may be easily translated to index values for the transition matrix. Similarly, the DFA's alphabet is also stored in a dictionary. The transition matrix is implementd as a 2-dimensional Python list, and contains only integer values. The accept states are simply stored as their integer representations (obtained from the dictionary of stored states) as a Python list.

4 Conclusion

This DFA simulator follows the rules and definitions of DFA's as closely as possible, and can sucessfully validate input strings based on DFA input specifications. Several test DFA's are included in the practical submission. Together with these, the programs `test_scanner.py` and `test_dfa_simulator.py` demonstrate that the scanner and parser/simulator do indeed function correctly.
To run the DFA simulator:

```
./dfa_simulator.py <file name> <input string>
```

where `<filename>` is the input file containing the specification of the DFA, and `<input string>` is the string to be validated.
To run `test_scanner.py`:

```
./test_scanner.py
```

To run `test_dfa_simulator.py`:

```
./test_dfa_scanner.py
```