Umeå University
Institution för Datavetenskap

DV2: Algorithms and Problemsolving 7.5 p
DV169VT16

OU5 Automaton

Submitted 2016-03-21
Author: Lorenz Gerber (`dv15lgr@cs.umu.se` `lozger03@student.umu.se`)
Instructor: Lena Kallin Westin / Erik Moström / Jonathan Westin

Contents

1 Introduction 1

  1.1 Push Down Automata Implementation 1

2 Program Structure 1

  2.1 Own Datatypes 1

3 Discussion 3

4 Conclusions 3

References 3

1 Introduction

The subject of this assignment was to design and construct Push Down Automaton (PDA) as a general datatype and apply it to implement a specific PDA that can process input according to *reverse polish notation* (RPN).

1.1 Push Down Automata Implementation

The lab assignment proposed to use either a representation as table or as a graph. It was also communicated that the implementation shall be finite and non-deterministic.

The PDA was implemented as a struct with the fields **current state**, **current input**, **stack**, **table of states**

2 Program Structure

The program was implemented using a conventional C structure with a lean main function that first defines and initializes variables, then calls for a function that creates and configures the push down automaton and finally the applying the command line argument to the configured pda.

2.1 Own Datatypes

pda - Push Down Automaton

A generic implementation of a push down automaton according to Sipser [2, pp 112-125].

- current state
- current input
- register
- input alphabet dlist with function pointers
- stack
- stack alphabet, dlist with function pointers
- states table
- create
- add input alphabet character
- add stack alphabet character
- add state
- add transition
- run automaton

The datatype *pda* is constructed from a struct. It contains a table (from course datatypes [1], constructed from dynamic list) with *states*, a stack (from course datatypes [1], *stack_1cell*). The *states* table contains the transitions.

Alphabets

The alphabets could be implemented as single chars. Then it is however difficult to define a more generic group of chars such as number or operator. To provide this possibility an alphabet character could also be a list or an array of unsigned chars. A more elegant way to solve this issue would be to define a character by a function, implemented through function pointers. This opens up for very flexible definition of alphabet characters. For the current implementation it was decided to define the alphabet by function pointers and functions.

A general problem in this assignement is the handling of chars and integers. A pda should basically stick to symbols however for the *rpn* calculation convertion to integers is needed.

States and Transitions

The representation of *state* and *transition* for a table based pda can be done in various ways and the distinction between *state* and *transition* is less clear than in a graph based model. Here two different ways were considered: Either states constructed only as a container for transitions, without any reference to the alphabet. This would require a more complex transition datatype. Also the datatype transition becomes less generic as it fits just for one specified pair of input and stack values.

Alternatively, *states* could also be implemented according to the example 2.14 in Sipser [2, p. 114]: The state is represented by triple nested table or an aggregated array where there is a multi column for each letter in the input alphabet with subgroups as the individual column for each letter of the stack alphabet. Implementing a representation for this model could be done with a nested table, an array, where the logic for accessing the different levels is integrated in the code or a tree structure. Such a representation has the advantage that it is directly visible whether a transition for a certain state is already defined or not as it has a unique location in the data structure. When choosing a representation with states as mere containers for transitions, a control mechanism to prevent assignment of duplicate transitions is needed.

A state has an numeric *id* and a table with *id's* of possible transitions to proceed along.

- id
- description
- table with allowed transitions

struct contains a table constructed from *dynamic list*. The table keys are integers with *structs* as data container.

Transition

A transition needs to know whether it matches the current state, it needs to define how to modify the current state and the id of the new state. *struct table* constructed from *dynamic list*.

pseudo example: create automata set input alphabet as list of unsigned chars set stack alphabet as list of unsigned chars set

3 Discussion

4 Conclusions

References

[1] DV169VT16, cambro course homepage, ‘datatypes’. `https://www8.cs.umu.se/kurser/5DV169/datatypes/index.html`, 2016. accessed: 2016-02-28.

[2] Michael Sipser. *Introduction to the Theory of Computation*. Cengage Learning, Boston, USA, 2012.