Task documentation DKA: Determinization of a finite automaton in PHP 5 for IPP 2016/2017
Name and surename: Martina Grzybowská
Login: xgrzyb00

1 Assignment

The objective of this assignment is to create a script in PHP 5 that processes text entry of a finite automaton. Subsequently, using pre-selected algorithms, it is to convert this automaton representation into various forms determined by the choice of command line arguments.

2 Data processing

2.1 Processing of command line arguments

This script allows for various arguments, by which it is necessary to meet prescribed requirements and combination conditions. In the case of detecting an unknown argument, multiple occurrence of one argument or illegal combination of two arguments, the script is terminated with error code `OPT_CODE` and error quote `OPT_QUOTE`. If the arguments have been successfully parsed, the script sets flags which determine its subsequent behaviour.

2.2 Processing of input file

The text from the input file is loaded into a variable as a string using a built-in function `file_get_content()`. If arguments `-i` or `--case-insensitive` are set, the string is directly converted into lowercase. This string is being split by one character a time using user-defined function `getToken()`, in which this character is subsequently sent to be analysed.

The analysis of syntax is carried out in descending order according to a set of rules created on the basis of prescribed sequence of prospective characters. If the input character does not comply with these syntactic rules, the script is terminated with an error code `LEX_SYN_ERR` and error quote `LEX_SYN_QUOTE`.

1 : *<start>* ⟶ ({*<pStates>*} , {*<pSyms>*} , {*<pRules>*} , *id* , {*<pStates>*})
2 : *<pState>* ⟶ ϵ
3 : *<pState>* ⟶ *id <pNextState>*
4 : *<pNextState>* ⟶ , *id <pNextState>*
5 : *<pNextState>* ⟶ ϵ
6 : *<pSyms>* ⟶ *'id' <pNextSym>*
7 : *<pNextSym>* ⟶ , *'id' <pNextSym>*
8 : *<pNextSym>* ⟶ ϵ
9 : *<pRules>* ⟶ ϵ
10 : *<pRules>* ⟶ *id 'id' –> id <pNextRule>*
11 : *<pNextRule>* ⟶ , *id 'id' –> id <pNextRule>*
12 : *<pNextRule>* ⟶ ϵ

Consequently, if the current character is in syntactic terms correctly evaluated as `LEFT_CURLY` (left curly bracket), the context is switched to one of the functions ensuring processing of inner contents and semantic inspection. Selection of one particular function from the set is determined by the current location in the descent of rules. In the event of the current character being evaluated as `COMMA`, these functions are called recursively, in which the terminating condition is based on recognising a character evaluated as `RIGHT_CURLY` that is not enclosed by apostrophes. If the end of string is reached instead, the script is terminated with an error code, as it does not comply with the syntactical rules.

Ignoring of all white characters and single line comments is ensured by calling a user-defined function `cutSpaces()` in all places where an occurrence of such a character would be syntactically correct.

2.3 Data storage

States and input symbols are stored in one-dimensional arrays. However, a single array would not be sufficient for effectively storing all rule components. As such, each newly created rule is an instance of the class called `InputRules` with `initState`, `inputSymbol` and `targetState` as its properties. Each one of these objects is then stored in an array of rules' objects.

3 Algorithms implementation

3.1 Elimination of epsilon transitions

First of all, in order for all subsequent implementations of algorithms to work properly, it is necessary to eliminate possible epsilon transitions by editing rules containing them. This was achieved by firstly creating an epsilon closure of each state and secondly altering every rule affected, using several nested loops. These loops iterate through arrays of states, symbols and rules alike and store the intermediate results into two auxiliary one-dimensional arrays.

3.2 Determinization and elimination of inaccessible states

To convert the mathematical interpretation of the algorithm for finite automaton determinization without inaccessible states to programming code, it was necessary to create two auxiliary one-dimensional arrays for iterating through loops and one array of arrays for possible necessity of merging of states.

Each such state, from which it is possible to proceed via currently analysed input symbol to the currently analysed state stored in a one-dimensional array called `s_iterator`, is, in turn, stored in a one-dimensional array called `s_pusher`. Such an array of stored states is then compared with all arrays stored in an array of arrays called `states_storage`. If no match is found, `s_pusher` is pushed into `states_storage` for subsequent analysis.

Following this, a new object of a rule is created with the current state from `s_iterator` as `initState`, current input symbol from the input alphabet as `inputSymbol` and a state consisting of merged components from the array `s_pusher` as `targetState`. If such a rule is not yet part of the rules array, it is pushed into it. If there is an intersection between one of the components in the array `s_iterator` and the array of final states, these components are merged and pushed into the final states array. This algorithm iterates until the `states_storage` array becomes empty.

4 Extension

4.1 WSA

The essence of this extension was a follow-up algorithm for the elimination of nonterminating states and creation of a well-specified finite automaton implemented on top of the existing implementation for the determinization without inaccessible states.

The very first step is the elimination of nonterminating states, for which it is necessary to create an auxiliary array of states containing all final states. Using multiple loops we search for rules, whose `targetState` belongs to this auxiliary array and if such a rule is found, its `initState` is also pushed into the auxiliary array. In the end we loop through rules again and search and eliminate those, whose `initState`, `targetState`, or both do not belong in the auxiliary array.

The second step of this algorithm is to create an artificial state called `qFALSE`, representing a trap. We create new rules by directing all combinations of states and input symbols, which were not contained in the original automaton, to the `qFALSE` state. If the `startingState` does not belong to the auxiliary array, `qFALSE` becomes the new `startingState`.