National and Kapodistrian University of Athens

Department of Informatics and Telecommunications

Principles of Programming Languages

Course Project in Haskell

Project implemented by:

CHARALAMPOS MARAZIARIS - 1115201800105
KONSTANTINOS TSIKOURIS - 1115201800201

Contents

1. Brief Introduction

Every task of the project has been implemented, including the 3 basic functions (makeNfa, nfaToDfa, regexFullMatch) and the 3 bonus assignments (Wildcard mathcing, regexPartMatch, README).

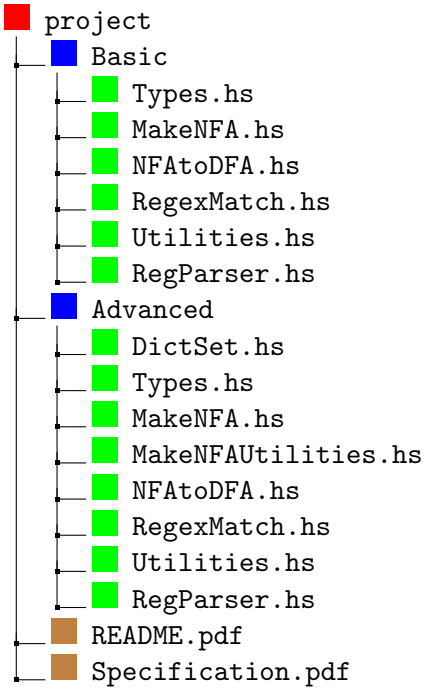
We have included 2 implementations of the same functionality:

- The **Basic**, implementing the algorithms described in the Project's specification PDF.

- The **Advanced**, has the goal of making a more efficient implementation of the desired functionality. This is achieved by implementing a different algorithm for the construction a Nfa without ε-transitions and with the goal of minimizing the number of transitions in general. The paper has the title "**Computing ε-free NFA from regular expression in $O(n(log(n))^2)$ time**" by C HRISTIAN H AGENAH A NCA M USCHOLL. Also, there are differences with the Basic implementation regarding the Bonus assignments which have both upsides and downsides.

2. Submitted Directory Structure

The **zip** file contains the following structure:



3. Run Instructions

```
$ cd <Basic|Advanced>
$ ghci
Prelude> :l RegexMatch.hs
Main> <run makeNfa or nfaToDfa or regexFullMatch or regexPartMatch>
```

4. Basic Implementation

4.1 RegEx to NFA (makeNFA)

We implemented the core of **Thompson's** RegEx to NFA construction algorithm, with 2 slight modifications:

- ε (empty input) is represented as a single-state NFA with an empty list of transitions. The state is also a final state.
- Concatenation of 2 Regular Expressions is done by using an ε-transition connecting the final state of the NFA corresponding to the 1st regex to the initial state of the NFA corresponding to the 2nd regex.

4.2 NFA to DFA

We implemented the algorithm proposed in the presentation of the Project.

4.3 RegEx Full Match

Attempts to pattern-match a given string, by searching for a path in the DFA that fully consumes the input string and also puts the DFA in a final state.

4.4 Bonus: Wildcard Matching

We implemented the Wildcard Matching mechanism by treating **'.'** as a new symbol of our RegEx, thus creating DFAs that include '.' as a transition character. Then, when the *RegEx Full Match* function must consume a character **'c'** from the input string, it can choose whether to follow a **'c'** or a **'.'** transition (if available). In this way, we make sure that we explore every possible path that can lead us to a final state while consuming the input-string. If such path exists, then the Finite Automaton accepts the input string, otherwise it does not.

4.5 Bonus: RegEx Part Match

We implemented an extension to the *RegEx Full Match* function, where for each string we consume, we check if we are in a final state of the DFA. If this is the case, then we add the string we -so far- consumed from the input string to a list and continue by consuming the next character.

5. Advanced Implementation

5.1 RegEx to NFA (makeNFA)

An alternative construction was used, as mentioned above, using an algorithm different from Thompson's construction. The reason this was done is because the number of states and transitions the constructed nfa has will be less compared to the states and transitions of the NFA from the Thompson's construction. Also, the absence of ε-transitions allows the conversion from NFA to DFA using the subset construction easier and faster. Considering the fact that the conversion from NFA to DFA is the most costly procedure in this project, and that it must have a worst-case exponential cost, it is a big advantage for the NFA constructed to have few states and transitions. The algorithm exists in the

MakeNFA.hs module

Also, in the implementation of this new construction, some functionalities where transferred to another module, with file name MakeNFAUtilities.hs

5.2 NFA to DFA

As mentioned before, since the NFA will not have ε-transitions (it is an ε-free NFA), there is no need to compute the epsilon closure of a state. This allows for a less time consuming conversion. Also, there is another change to the subset construction beside the epsilon closure but it will be discussed in the bonus assignment WildCard. The implementation is in the file/module NFAtoDFA.hs

5.3 RegEx Full Match

After constructing the DFA which accepts the given regex, for efficiency purposes, the list of transitions is converted into a dictionary of type **Dict StateId (Dict TransChar StateId)**. Dict refers to a dictionary. The goal of the dictionary is a fast search for the nex state in the transitions given a state current state and a transtion character c. As the type indicates, the dict collects all stateIds which are a first element of a transition. For each such stateId, keep in a dictionary for each transition character c, in which state is leads to (the next state). Also, keep in a **Set StateId** all the final states so that it will be fast to check when a state is also a final state. Of course, for the dictionary and set to be constructed, there will be a cost of worst-case time complexity $O(nlog(n))$. However, comparing that to the fast search of the next state, it is not as significant, especially if there are lots of transitions and the input string is big. The dict and set types and functionalities are defined in the module with file name DictSet.hs.

Besides the change in how the next state is found, the algorithm is the same as that of the Basic implementation. The only other difference is in how the bonus WildCard is implemented which will be discussed below

5.4 Bonus: Wildcard Matching

In the Basic implementation, the wild card '.' is treated as another character with no special characteristics in the NFAtoDFA algorithm. Then, in regexFullMatch, whenever there is a choice of the transition character c or '.', using backtracking, both choices will be checked in the worst case. This means, that if many such choices have to be made during the processing of the input string for acceptance, this will result into an exponential cost, significantly impacting the running time in a negative way.

To avoid this potential issue with the backtracking in the Advanced implementation, a modification in the functionality of the NFAtoDFA algorithm is made. Specifically, in the subset construction, whenever from an set-state a transition character c is about to be read, the next set-state will not be just the the collection of states which can be reached from states in the current set-state using ε-transitions and exactly one transition with transition character c. It will also take into account the transitions with transition character '.'. In other words, instead of demanding the use of exactly one transtition with transition character c, it will demand the use of exactly one transtition with transition character c

or '.'. This way, the NFAtoDFA implements internally in the DFA the bonus WildCard implementation . All that the regexFullMatch has to do then is that when it is about to read a character c, it must first check whether a transition with transition character c exists from the current state. If it does, go to that next state. Otherwise, search for a transition with transition character '.'. If it is found, go that next state, otherwise, the dfa does not accept the input string.

This way, the backtracking is removed and the potential exponential blowup can be avoided. However, this comes at a cost during the construction of the dfa from the nfa. This happens because the modification allows states to reach other states with more ways by allowing reading the character '.'. This results in an increase of states and transitions in the dfa. For comparison, given the regex
"(((a|bcd(.(d*|_).)*a*)ba(e(f|_)|_)*)|_((a|_).e)*((_)*)*(_))|((d.e)*c)", the basic impementation of the NFAtoDFA constructed a dfa with 20 states and 34 transitions. However, the modified functionality of the NFAtoDFA constructs a dfa with 70 states and 430 transitions, a considerable increase.

This means that while backtracking is avoided, a new cost in states and transitions is created. However, because regexFullMatch uses a dictionary instead of a list of transitions, the cost is not as significant while searching for next states, although the initial construction of the dictionary will have a bigger cost. There is one more benefit to this modified functionality of the NFAtoDFA though, and that is that it is compatible with the function regexPartMatch

5.5 Bonus: RegEx Part Match

The modified implementation of the WildCard allows for no significant changes in the algorithm for regexPartMatch, no increase in the complexity. Nothing important needs to be discussed for the difference of the the advanced implementations of regexFullMatch and regexPartMatch, only that in the advanced implementation, regexPartMatch also takes into account the WildCard