Sean Sponsler, Evan Walters

Dr. Ravikumar

May 20th, 2023

CS454 Final Project Problem 6

Problem Statement:

This program takes as input a regular expression, and a string. Then there is an output determining if the string is accepted by the regular expression.

Solution Outline:

The program converts the given RE (regular expression) to an epsilon-NFA, which is then converted to an NFA that is then used in conjunction with the input string to determine if the string is accepted. The program loops for continued input.

Here are the main functions that are utilized. (not all functions are here)

Infix-Prefix:

Given a valid regular expression the expression is reconfigured for operators to be left bound and operands to be right bound with binary/unary operators corresponding to two/single operands. This function starts inserting concatenations (‘.’) wherever they are missing. Then the string is reversed (exchanging parenthesis as well). Finally using a character stack a prefix string is built based on the operator priority (* > . > +). Eventually the resulting string is reversed again (void of all parenthesis at this point) and outputted.

Convert:

Convert is a recursive function that is given just the prefix string as input. Then using a helper function (that utilizes the position in the string as an additional input) the string is parsed. Binary operators create two recursively created NFA objects to be combined via the operator, and the unary operator creates a single recursively created NFA object to be mapped via a kleene star. The base case is essentially when an NFA is being created out of a single symbol.

NFA constructors:

There are two main NFA constructors, one utilizes a single symbol the other utilizes an operator and one to two NFAs depending on if the operator is binary or unary. The first NFA constructor creates a 2 state NFA with a single transition using the operand from state 0 to state 1. The second NFA constructor does something different depending on the given operator. In the case of concatenation between NFA M1 and NFA M2, the final state of M1 is mapped with an epsilon transition to the start state of M2. Then only the final state of M2 is in the list of final states for the resulting NFA. In the case of union a new start state is created with two epsilon transitions, one to the start of M1 and one to the start of M2; then, all the final states of M1 and M2 are entered into the list of final states for the resulting NFA. In the case of kleene star only NFA M1 is used and the input of M2 isn't needed/used. This case creates two epsilon transitions: one from all the final states of M1 to the start state of M1, then another epsilon transition from the start state of M1 to all the final states of M1. At the end of this constructor the removeEpsilon function is called.

Remove Epsilon:

This function is an NFA function and works on the current object, to remove all epsilon transitions (Example: an epsilon transition from 0 to 1). To do this we first designate a source state (0) and a destination state (1) for an epsilon transition. Then all transitions from the destination (1) are added to the list of transitions from source (0), this way the epsilon transition is obsolete. After this the epsilon transition is deleted from the source's list of transitions. In the case that the destination had its own epsilon transition added to the source's transition list the algorithm flags a restart. This function has the likelihood of leaving floating states and their transitions that have a high possibility of not being used in the future. There is also the case of an epsilon transition to a final state (like with kleene star); in this case we simply add the source state to the list of final states in addition to the normal mapping changes.

Breadth First Search:

This function takes as input an NFA and a string to test acceptance with by outputting true or false. This function makes use of a queue of pairs, the pairs consisting of a state number and an input symbol. Initially we begin with the start state of the NFA and the first character/symbol in the string. Then in a loop the next pair in the queue is recorded for comparisons then popped. Later in the loop every valid transition from the current state with the first letter in the current string is added to the queue via the result state, and a substring of the current string minus the first character. A check is also done in the loop to determine whether the current state is a final state and that the current string is empty; in this case we return true.

Data Structures and Algorithms:

NFA objects:

NFA objects are stored in the form of a 2-D vector of pairs (kind of like a hash), where each pair is a destination, and a transition. The index of the vector of these transitions refers to the current state for the transitions. For example for the vector of vectors, vector 0 holds all the transitions from state 0, vector 1 holds all the transitions from state 1 and so on. Each NFA object also keeps track of the size, startstate, and a vector of final states (Final states are in ascending order, and searched via a binary search).

Infix to Prefix

A stack is used to keep track of operators and parentheses as the symbols are parsed from the prefix string.

Convert:

Uses a recursive algorithm to create an NFA out of the prefix string. Thought of as creating small NFAs out of the operands then working up creating larger NFAs by combining these with via operators.

Remove Epsilon:

This function uses three for loops, two are nested. The main loop iterates each state (the main vector). The 2nd iterates backwards through the vector of transitions for the current state; here a check for an epsilon transition is made. When found it is deleted and a for loop is initiated, this loop adds every transition from the destination of the deleted epsilon transitions to the list of transitions for the current state. There is also a restart flag that looks for newly added epsilon transitions.

Breadth First Search:

The function uses a queue of pairs, a pair being a current state and string. Then a while loop is entered based on whether the queue isn't empty. The loop starts with a pop of the queue, then all possible routes are added to the queue with a new string that doesn't retain the first character of the current string; from there the loop reiterates adding the next set of possible routes and so on. Each iteration also does a check for whether the string is empty and in a final state. The function has no use for a visited vector as we aren't looking for the shortest route, and that could easily get consuming with implementation of kleene star.

Program operation:

1) Use makefile to generate main executable or use the command “g++ main.cpp NFA.cpp -o main”
    a) Run the program executable
2) The program expects proper syntax from the user, but some accommodations are made for simple errors and mistakes. For the purposes of these examples, 100% proper syntax will be utilized. For the first input, the user is prompted to provide a regular expression following the syntax rules and expected symbols retrieved. This includes the following non-terminal symbols:
    1) + symbol = Union operator
    2) . symbol (period) = Concatenation operator
    3) * symbol = Kleene star
    4) (and) to represent parentheses and context
    5) & is our symbol to represent epsilon

    Terminal symbols are expected as single-character alphanumeric. This means that “a0” will be distinguished as two separated symbols and will instead be represented as “a.0”.

    (Other examples: “aA” → “a.A”, “abc” → “a.b.c”)

    Note: at either input scenarios in the program (when entering RE or when entering string), if the user enters a single ‘%’ symbol, the program will exit. Alternatively, when invalid input is detected, an error message will be outputted and the program will exit.

3) First, the program will expect the user to input a syntactically valid Regular Expression to test a string against. In this example we will use the expression “(a.b.c)*”

```
--------------------------------------------------------
Enter regular expression in a single line with these rules:
1. "+" = union, "." = concatenation, "*" = kleene star, "&" = epsilon/null, "(" and ")" are parentheses
2. Symbols are single characters, can only be upper/lowercase letters or numbers ("a0" will be interpreted as "a.0")
3. Enter % by itself to quit the program.

Enter Regular Expression(% to quit): (a.b.c)*
```

4) Upon entering this input, the program will continue and ask the user to enter a syntactically valid input string. In this example we will use the string “abcabc”

```
Enter Regular Expression(% to quit): (a.b.c)*
Enter string w to be tested for acceptance against given Regular Expression using only alphanumeric characters and epsilon(&)(% to quit): abcabc
```

5) Finally, when the user enters this given input string, the program will run its accepts() overhead function to determine if the given string is a valid form of the given regular expression. In our example this results to:

```
Enter Regular Expression(% to quit): (a.b.c)*
Enter string w to be tested for acceptance against given Regul


The string: abcabc
Is ACCEPTED by the RE: (a.b.c)*
```

6) The program will return to step 3 and repeat.

Time complexity:

The time taken for most examples is exactly 0.001001 seconds. With more complicated inputs we were able to reach time taken of 0.001002 seconds:

```
The string: a
Is ACCEPTED by the RE: (a*)*
Time taken: 0.001001 seconds
-----------------------------------------------------
Enter regular expression in a single line with these rules:
1. "+" = union, "." = concatenation, "*" = kleene star, "&" = epsilon/null, "(" and ")" are parentheses
2. Symbols are single characters, can only be upper/lowercase letters or numbers ("a0" will be interpreted as "a.0")
3. Enter % by itself to quit the program.

Enter Regular Expression(% to quit): (qwertyuiopasdfghjklzxcvbnm)*
Enter string w to be tested for acceptance against given Regular Expression using only alphanumeric characters and epsilon(&)(% to quit): qwertyuiopasdfghjklzxcvbnm


The string: qwertyuiopasdfghjklzxcvbnm
Is ACCEPTED by the RE: (qwertyuiopasdfghjklzxcvbnm)*
Time taken: 0.001002 seconds
```

More examples:

Example 2:

RE: “(a*b*)*”

String: “aaabbb”

```
Enter Regular Expression(% to quit): (a*b*)*
Enter string w to be tested for acceptance against given Regular Expression using only alphanumeric characters and epsilon(&)(% to quit): aaabbb


The string: aaabbb
Is ACCEPTED by the RE: (a*b*)*
```

Example 3:

RE: “(a+b+c)*d”

String: “acbabcabcbabcad” => ACCEPTED

```
Enter Regular Expression(% to quit): (a+b+c)*d
Enter string w to be tested for acceptance against given Regular Expression using only alphanumeric characters and epsilon(&)(% to quit): acbabcabcbabcad


The string: acbabcabcbabcad
Is ACCEPTED by the RE: (a+b+c)*d
```

String: “abc**d**abc” => NOT ACCEPTED

```
Enter Regular Expression(% to quit): (a+b+c)*d
Enter string w to be tested for acceptance against given Regular Expression using only alphanumeric characters and epsilon(&)(% to quit): abcdabc


The string: abcdabc
Is NOT ACCEPTED by the RE: (a+b+c)*d
```

Conclusion:

We were happy to implement this sort of conversion. It really links together how languages can parallel in order to be more easily accessed by code. We understand there are many ways to implement NFAs in code, though it was fun to come up with our own unique way that we could then use to manipulate the NFA based on the regular expressions we tested.