# References/Research

## Regular Expressions

### <https://en.wikipedia.org/wiki/Regular_expression>

* RE is a sequence of characters that define a search pattern

<https://github.com/ianmcloughlin/slides-regular-expressions/raw/master/slides.pdf>

* RE are strings that represent patterns of text
* Brackets are used to group the characters together
* They are used to search other strings for patterns

**Special Characters**

|  |  |  |
| --- | --- | --- |
| **.** | To concatenate | a.b means a followed by b |
| **|** | Means or | a|b means a or b |
| **\*** | Zero or more times | a\* means 0 or more a’s |

**Precedence**

* \* comes first
* . after \* but before |
* | comes last
* Treat brackets as individual charcters

### <https://docs.python.org/2/library/re.html>

* Regular expressions can be concatenated to form new regular expressions; if A and B are both regular expressions, then AB is also a regular expression. In general, if a string p matches A and another string q matches B, the string pq will match AB.

This logic can be used in the project to build small NFA’s for the regular expression

**Characters that have a special meaning when using the re library**

|  |  |
| --- | --- |
| **Dot (.)** | Any character except a newline |
| **Caret (^)** | The start of a string |
| **Dollar ($)** | The end of a string |
| **Asterisk (\*)** | Match 0 or more repetitions as possible of the RE |
| **Plus (+)** | Match 1 or more repetitions as possible of the RE |
| **Question (?)** | Match 1 or 0 or more repetitions as possible of the RE |
| **Square Brackets ([])** | A set of characters |
| **Pipe (|)** | Either or |
| **Back slash (\)** | Special sequence |
| **Round Brackets ()** | A group |

**Functions used in the re library**

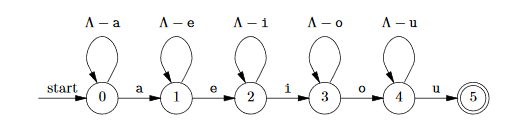
|  |  |
| --- | --- |
| **findall** | Returns list of all matches |
| **search** | Returns an object that if there’s a match in the string |
| **split** | Returns list where the string in split at each match |
| **sub** | Replaces one or more matches with a string |

I will keep these symbols and functions in mind when building my program

## Finite Automata

### <http://infolab.stanford.edu/~ullman/focs/ch10.pdf>

* The finite automaton is a graph-based way of specifying patterns. These come in two varieties, deterministic automata and nondeterministic automata
* Regular expressions are an algebra for describing the same kinds of patterns that can be described by automata
* Regular expressions can be converted to automata and vice versa



**Program written in C which examines a sequence of letters that contain aeiou which represents the Automaton above**

#include <stdio.h>

#define TRUE 1

#define FALSE 0

typedef int BOOLEAN;

BOOLEAN findChar(char \*\*pp, char c){

(while (\*\*pp != c && \*\*pp != ’\0’)

(\*pp)++;

if (\*\*pp == ’\0’)

return FALSE;

else {

(\*pp)++;

return TRUE;}

}

BOOLEAN testWord(char \*p){

/\* state 0 \*/

if (findChar(&p, ’a’))

/\* state 1 \*/

if (findChar(&p, ’e’))

/\* state 2 \*/

if (findChar(&p, ’i’))

/\* state 3 \*/

if (findChar(&p, ’o’))

/\* state 4 \*/

if (findChar(&p, ’u’))

/\* state 5 \*/

return TRUE;

return FALSE;

}

main(){

printf("%d\n", testWord("abstemious"));

}

## Thompsons Construction Algorithm

## <https://github.com/ianmcloughlin/slides-thompson/raw/master/slides.pdf>

* An algorithm to construct an NFA from a regular expression
* The NFA will recognize the same language as the regular expression
* Using fragments which are a smaller NFA inside the overall NFA which are on the stack
* normal characters are pushed to the stack
* Special characters are popped from and pushed to the stack

Types of characters

* Normal, non-special characters including the empty string, push the fragment to the stack
* Where there is a ., pop 2 fragments from the stack
* Where there is a |, pop 2 fragments from the stack
* Where there is a \*, pop 1 fragments from the stack
* Read the regular expression from left to right and build up smaller NFA’s one character at a time.
* When you come across a special character, used NFA’s previously created to create one with the special character to build one NFA
* Pop 2 automata off the stack
* Put each automaton to the left of the first automata

## Postfix and Infix

### [http://interactivepython.org/runestone/static/pythonds/BasicDS/InfixPrefixandPostfixExpres ions.html](http://interactivepython.org/runestone/static/pythonds/BasicDS/InfixPrefixandPostfixExpres%20ions.html)

* An arithmetic expression such as B \* C is a type of notation is referred to as **infix** since the operator is in between the two operands that it is working on
* Each operator has a **precedence** level. Operators of higher precedence are used before operators of lower precedence
* The only thing that can change that order is the presence of parentheses. The precedence order for arithmetic operators’ places multiplication and division above addition and subtraction.
* Remember that computers need to know exactly what operators to perform and in what order

**Expression Examples**

|  |  |
| --- | --- |
| A + B \* C | B and C are multiplied first, and A is then added to that result |
| (A + B) \* C | Parentheses would force the addition of A and B to be done first before the multiplication. |
| A + B + C | By precedence the leftmost + would be done first. |

## Shunting Yard Algorithm

<https://web.microsoftstream.com/video/a29536d4-e975-4172-a470-40b4fe28866e>

* Translates infix to postfix
* Postfix puts the operator (|, \*, .) after the operands (a, b)
* Precedence needs to be taken into consideration when translating

**Precedence**

1. **\* Kleene star**
2. **. Concatenate**
3. **| Or**

**Expression Examples**

|  |  |
| --- | --- |
| **Infix** | **Postfix** |
| a.b | ab. |
| a|b | ab| |
| a\* | a\* |

* Using the stack data structure, you can translate infix to postfix
* Each time you open a bracket, put the operator inside that bracket onto the stack
* Each time you close a bracket, take the operator inside that bracket off the stack in order of whichever is on top.
* If the operator that is about to be added to stack has lower or equal precedence than the operator at the top of the stack, take it off the stack and added to the expression
* Each time an operator is taken off the stack and the brackets has been closed; the stack is then cleared. Start fresh

**Elaborated Expression Example**

|  |  |
| --- | --- |
| **Infix** | **Postfix** |
| (a|b).(a\*|b\*) | ab|a\*b\*|. |

**Steps:**

1. Put ( onto the stack
2. Put a into the expression
3. Put | onto the stack
4. Put b into the expression
5. Take | off the stack and put into the expression because the brackets have closed. The stack is now empty
6. Put . onto the stack
7. Put ( onto the stack
8. Put a into the expression
9. Put \* onto the stack
10. Put | onto the stack
11. Take \* off the stack because it has higher precedence than | and put into the expression
12. Put b into the expression
13. Put \* onto the stack
14. Take \* off the stack and put into the expression because the brackets have closed and it has higher precedence than |
15. Take . off the stack. The stack is now empty
16. The expression has all been read into postfix notation