|  |
| --- |
| A close up of a clock  Description automatically generated |
| A close up of a logo  Description automatically generated  Research on executing regular expressions on strings using Thompsons Construction |
| |  |  |  | | --- | --- | --- | | MICHELLE LALLY | 3/31/19 | Graph Threory | |

# Regular Expressions

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

* A Regular Expression is a sequence of characters that define a search pattern. It means, “the specific, standard textual syntax for representing patterns for matching text.”
* Regular Expressions are commonly used in string searching algorithms for find or find and replace functionality
* Their use can be found in text editors, word processors, search engines and for input validation.
* The idea of a regular expression developed around the 1950’s by Stephen Cole Kleene who was an American Mathematician who helped with the founding of recursion theory, also known as computability theory.
* Kleene along with several other mathematicians went on to build the basis of theoretical computer science and would become the interest of his lifelong research.

There are 2 types of characters in a regex:

Special Characters used in Regular Expressions

**Metacharacter:** Has a special meaning.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| [ | ] | \ | ^ | $ | . | | |
| ? | \* | + | { | } | ( | ) |

Examples of literal characters used in Regular Expressions

**Literal Character:** A regular character that has its literal meaning

|  |  |
| --- | --- |
| a-z | A-Z |
| 0-9 | !%&:;@ |

* When used together, metacharacters and literal characters are used to patterns and it can match many different instances of it.
* Most programming languages provide a library with regex capabilities and functions.

<https://www.princeton.edu/~mlovett/reference/Regular-Expressions.pdf>

* A Regular Expression is used to describe a pattern for searching a special string of text.
* When a string is found using a regular expression, it is referred to as a match. A match is just the piece of text that has been found and corresponds to the regular expression.
* Many different programming languages implement regular expressions. They usually have minor differences with syntax but can execute similar processes.

|  |  |
| --- | --- |
| **Expression** | **Matches** |
| regex | The literal text:  regex |
| ^\b[A-Z0-9.\_%+-]+@[A-Z0-9.-]+\.[A-Z]{2,}? | Email addresses that must be:  A series of letters, digits, dots, underscores, percentage signs, and hyphens.  Followed by an at symbol.  Another series of letters, digits, dots, underscores, percentage signs, and hyphens.  Then a single dot and 2 or more letters |
| reg(ular expressions?|ex(p|es)?) | The 5 following words in one search:  regular expression  regular expressions  regex  regexp  regexes |
| ^(19|20)\d\d[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])$ | Dates in the YYYY-MM-DD format:  From 1990-01-01  To 2099-12-31 |

**How to check a valid date in Perl.**

sub isvaliddate {

my $input = shift;

if ($input =~ m!^((?:19|20)\d\d)[- /.](0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])$!){

# At this point, $1 holds the year, $2 the month and $3 the day of the date entered

# 31st of a month with 30 days

if ($3 == 31 and ($2 == 4 or $2 == 6 or $2 == 9 or $2 == 11)) {

return 0;

# February 30th or 31st

} elsif ($3 >= 30 and $2 == 2) {

return 0;

# February 29th outside a leap year

} elsif ($2 == 2 and $3 == 29 and not ($1 % 4 == 0 and ($1 % 100 != 0 or $1 % 400 == 0))) {

return 0;

# Valid date

} else {

return 1;

}

# Not a date

} else {

return 0;

}

}

## <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.

**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

Predefined sets of characters

|  |  |
| --- | --- |
| \d | Matches any decimal digit; this is equivalent to the class [0-9]. |
| \D | Matches any non-digit character; this is equivalent to the class [^0-9]. |
| \s | Matches any whitespace character; this is equivalent to the class [ \t\n\r\f\v]. |
| \S | Matches any non-whitespace character; this is equivalent to the class [^ \t\n\r\f\v]. |
| \w | Matches any alphanumeric character; this is equivalent to the class [a-zA-Z0-9\_]. |
| \W | Matches any non-alphanumeric character; this is equivalent to the class [^a-zA-Z0-9\_]. |

# Stephen Cole Kleene

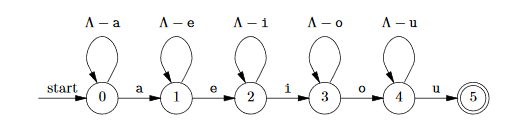
## <https://en.wikipedia.org/wiki/Stephen_Cole_Kleene>

* The inventor of regular expressions had many other contributions to mathematical logic.
* Computability theory is what Kleene is best known for by studying its’ basic objects, computable functions which refer to calculations that can be made using a mechanical calculation device with a procedure outlined by an algorithm.
* There are many mathematical notions which were named in honour of his work that solved problems that have made the modern computing we have today possible.
* Some of these concepts that may be heard of today; Kleene hierarchy, Kleene star, Kleene recursion theorem, Kleene algebra.
* He worked alongside some of the other big names known for mathematics such as [Alonzo Church](https://en.wikipedia.org/wiki/Alonzo_Church), [Alan Turing](https://en.wikipedia.org/wiki/Alan_Turing) and [Rózsa Péter](https://en.wikipedia.org/wiki/R%C3%B3zsa_P%C3%A9ter).

# 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(){

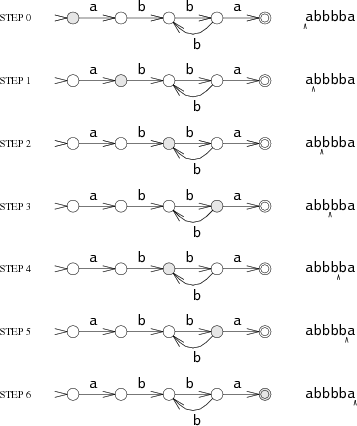
printf("%d\n", testWord("abstemious"));

}

## <https://swtch.com/~rsc/regexp/regexp1.html>

* Finite Automate are another way to describe sets of character strings. They are also known as state machines.
* The machine reads an input string one character at a time, following arrows corresponding to the input to move from state to state
* There are 2 types of machines that can be used to represent Finite Automata.

|  |  |
| --- | --- |
| **Deterministic (DFA)** | **Non-Deterministic (DFA)** |
| Only possible to be in 1 state at a time | Machines that must choose between multiple possible next states |
| Each input letter leads to at least one new state | To always let the machine guess which state it thinks is correct |
|  | NFA for a(bb)+a |

**An example of a DFA that**

**recognizes the set of string matched by**

**abbbba**

* When the DFA reads the first letter of the string, the a, it is in the start state s0.
* It follows the a arrow to state s1.
* This process repeats as the DFA reads the rest of the string: b to s2, b to s3, b to s2, b to s3, and finally a to s4
* The DFA ends in *s*4, a matching state, so it matches the string
* If the machine ends in a non-matching state, it does not match the string

NFA’s & DFA’s are a powerful way of matching Regular Expressions. Every regular expression has an equivalent NFA, they can match the same strings.

There have been many ways described on how to convert regular expressions into NFA’s, Thompsons Construction for example is a method used which has been proven to speed up the matching process to make 1,000,000 times faster.

# 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 . or a | pop 2 fragments from the stack
* Where there is a \*, ? or + pop 1 fragment from the stack

**Algorithm**

* Read the regular expression from left to right and build up 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 1 or 2 automata off the stack
* Put each automaton to the left of the first automata
* Build a new, bigger automaton to add to the stack
* At the end, there should only be one automaton

**Examples of the smaller NFA’s that will be built at a time after each character is read**

|  |  |
| --- | --- |
| A single literal character  Single-character NFA | The concatenation of e1 and e2  Concatenation NFA |
| The choice of e1 or e2  Alternation NFA | The choice of e or the empty string  Zero or one NFA |
| The choice of zero of more of e  Zero or more NFA | Having only one or more of e  One or more NFA |

# 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