Regular Language Library

Grant Clark (December 1, 2022)

Contents

Chapter 1

The STL Helper

1.1 Extended std::to_string Functions

This regular language library relies heavily on extending functionality of the std::to_string functions, and within STL_Helper.h, there are many new std::to_string functions defined. There are defenitions for the following types:

- char
- std::string
- std::pair
- std::unordered_map
- std::unordrered_set
- std::vector

1.2 Extended Hash Structs

This regular language library also relies on extending the structs used to hash items within standard unordrered sets and maps. These hash structs are the reason why it was necessary to extend the usage of the std::to_string functions, as the std::string class already has a defined hash struct. All of my hash structs use the std::to_string functions to hash its values. There are hash structs defined for the following types:

- std::pair

- std::unordered_map
- std::unordered_set

1.3 The "helper" Namespace

The helper namespace was created for the defenition of my std::unordered_set merge function. I used a separate namespace so that the std::merge function can still work, but the version of C++ I was using did not have access to that function, so I put it into its own namespace so that a user can also utilize the std::merge function.

Chapter 2

The DFA Class

2.1 Construction

This class behaves like a deterministic finite-state automata and can compute acceptance for constructed DFAs. To construct a DFA object, you need to provide the constructor with two template types:

- The sigma type, the type that you will use as characters for tranisions, referred to as S_t.
- The state type, the type you will use to represent states, referred to as Q_t.

Note: Both given types **MUST** have an std::to_string function overwritten to work, so if you want to use a class that does not already have an std::to_string function written for it, create one for your classes in the STL_Helper.h file.

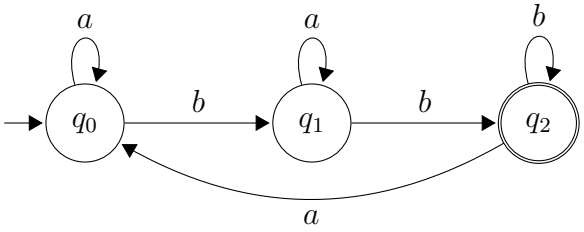
Next, the constructor must be provided with the following parameters in this order:

- std::unordered_set< S_t >, which is your sigma (set of valid transitioning characters)
- std::unordered_set< Q_t >, which is the set of your states
- Q_t, your initial state
- std::unordered_set< Q_t >, which is the set of accepting states (subset of the set of states)
- std::unordered_map< std::pair< Q_t, S_t >, Q_t > which will act as your

delta, or transition function.

This looks rather confusing, so to clarify we will use an example. We will take the following DFA:

$$\Sigma = a, b$$



and represent it using the DFA class, using std::string as both the sigma and state type:

```
#include <iostream>
#include "RegLang.h"

typedef std::string S_t;
typedef std::string Q_t;

int main()
{
    S_t a = "a";
    S_t b = "b";
    Q_t q0 = "q0";
    Q_t q1 = "q1";
    Q_t q2 = "q2";

    std::unordered_set< S_t > S {a, b};
    std::unordered_set< Q_t > Q {q0, q1, q2};
    std::unordered_set< Q_t > F {q2};

    std::unordered_map< std::pair< Q_t, S_t >, Q_t > delta;
    delta[{q0, a}] = q0;
    delta[{q0, b}] = q1;
    delta[{q1, a}] = q1;
    delta[{q1, b}] = q2;
    delta[{q2, a}] = q0;
    delta[{q2, b}] = q2;

    DFA< S_t, Q_t > M(S, Q, q0, F, delta);

    return 0;
}
```

You can also construct a DFA using the copy constructor like so:

```
DFA< S_t, Q_t > M(S, Q, q0, F, delta);
DFA< S_t, Q_t > M_2(M); //M_2 is now a copy of DFA M.
```

You can also use the assignment operator to make a DFA into a copy of another DFA:

```
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

// Assuming S_, Q_, q0_, F_, and delta_ are all valid.
DFA< S_t, Q_t > M_2(S_, Q_, q0_, F_, delta_);

M_2 = M; //M_2 is now a cppy of DFA M.
```

2.2 Accessing Members

The DFA object has 5 methods used to access members, however, once a DFA is created, you are not allowed to alter the contents of its members. i.e., once a DFA is made, it is meant to be immutable (the exception to this is the assignment operator). Each of the following methods returns that given member variable as a constant reference:

- sigma()
- states()
- initial_state()
- accept_states()
- delta()

2.3 Computing Acceptance

A DFA object can be used to compute acceptance by providing the operator() method with a string of characters. This string is represented represented by an **std::vector< S_t >**, and currently is the only accepted input for acceptance. Using the example DFA we created above, we can compute the following:

```
// S, Q, q0, F, and delta are all the same
// from the previous example.
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

M({a, b, a, a, b}); // True
M({b, b, a, b});    // False
M({a, a, a, a, a}); // False
M({b, b, a, b, b}); // True
M({a, b, b, b, b}); // True
```

Errors associated with operator():

- **DFA_Invalid_Sigma_Character_Error**, This error is thrown if you give the operator() method an std::vector ofcharacters to compute that contains a character that is not within its given set of characters Sigma.

2.4 Instantaneous Descriptions of Acceptance

You can view the process of acceptance visually using the **IDs()** method. This function takes in an std::vector of characters in sigma to compute acceptance, but instead of returning a boolean, it returns a, std::vector< std::string > which contains an instantaneous description of each step in the computation. Using the given example DFA above, this code:

```
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

std::vector< S_t > w {a, b, a, a, b};

std::vector< std::string > ids = M.IDs(w);
for (const std::string & str : ids)
    std::cout << str << std::endl;
```

will produce this output:

```
[a,b,a,a,b]
(q0, [a,b,a,a,b])
(q0, [b,a,a,b])
(q1, [a,a,b])
(q1, [a,b])
(q1, [b])
(q2, [])*
```

The first string in the vector of strings returned by IDs() is always just the string itself, and every subsequent line is a computational step, with the first computational step always being a tuple of the initial state and the initial string given. Each subsequent step uses the first character in the string and moves to the next state using delta until there are no characters left in the

string. If the final state you have reached is an accepting state, the final tuple will have a * character printed after it to denote that the string was accepted.

NOTE: All errors associated with operator() and how they are thrown are the same for the IDs() function.

2.5 Validating a DFA

This class includes a method, **valid()**, which returns a boolean that denotes whether the given DFA is a valid DFA. For the DFA to be considered valid, it must meet the following criteria:

- Set of states must be non-empty.
- All accepting states must be present within the set of states.
- Initial state must be within the set of states.
- All states and characters within delta transitions must be within the set of states and set of characters sigma respectively.
- All states transitioned to within delta must be within the set of states.
- Each state must have exactly one transition for each character in the set of characters sigma.
- Each state must be reachable.

If all of the above criteria are met, the **valid()** function will return true.

2.6 DFA Compliment

The **compliment()** method returns a DFA< S_t, Q_t > object in which every accept state of the initial DFA is now no longer an accept state, and every state in the initial DFA which was not an accept state is now an accept state. Using the acceptance examples, we can do the following:

```
// S, Q, q0, F, and delta are all the same
// from the previous example.
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

M({a, b, a, a, b}); // True
M({b, b, a, b});    // False
M({a, a, a, a, a}); // False
M({b, b, a, b, b}); // True
M({a, b, b, b, b}); // True

DFA< S_t, Q_t > M_comp(M.compliment());

// Acceptances are now flipped.
M_comp({a, b, a, a, b}); // False
M_comp({b, b, a, b});    // True
M_comp({a, a, a, a, a}); // True
M_comp({b, b, a, b, b}); // False
M_comp({a, b, b, b, b}); // False
```

2.7 DFA Minimization

the **minimal()** method returns a minimized DFA< S_t, Q_t > using Hopcroft's algorithm for DFA minimization.

```
// S, Q, q0, F, and delta are all the same
// from the previous example.
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

M = M.minimal(); // M has now been minimized.
```

2.8 is_accepting() Method

This method takes in a state object (Q_t) and returns true if it is within the set of accepting states.

2.9 DFA to NFA Conversion

the **to_nfa()** method returns an NFA< S_t, Q_t >. This method requires that the user provide an S_t value to be used as the epsilon character in the character set sigma of the returned NFA.

```
// assuming all given parameters are valid.
DFA< S_t, Q_t > M(S, Q, q0, F, delta);

//In this specific instance, we are assuming that
//S_t is std::string.
S_t epsilon = "";

NFA< S_t, Q_t > N(M.to_nfa(epsilon));
```

2.10 DFA Intersection

This library comes with a function that will take two DFA objects and return a new DFA created from the intersection of the given DFAs. The returned DFA will be in this form: DFA< S_t, std::pair< Q_t0, Q_t1 > > where Q_t0 is the state type of the first DFA, and Q_t1 is the state type of the second DFA. This function takes three templated parameters, the first being the sigma type of both DFA, the second being the state type of the first DFA, and the third being the sigma type of the second DFA. The two parameters it takes in are the two DFA in which the intersection will be performed on.

NOTE: This function only works on two DFA who share the same S_t type. Also, do **NOT** use this function on two DFA who share any state names, as it will result in unintended behavior. Additionally, the two DFA sent in **MUST** have exactly the same sigma, or set of accepted characters, otherwise an intersection does not make sense to perform.

```
// assuming all given parameters are valid and the sets
// of states Q0 and Q1 do not share any state names.
// NOTE: Sigma is shared.
DFA< S_t, Q_t0 > M0(S, Q0, q00, F0, delta0);
DFA< S_t, Q_t1 > M1(S, Q1, q10, F1, delta1);

DFA< S_t, std::pair< Q_t0, Q_t1 > >
    intersection = dfa_intersection< S_t, Q_t0, Q_t1 >(M0, M1);
```

Errors associated with dfa_intersection():

- **DFA_Sigma_Mismatch_Intersection_Error**, This error is thrown if you call the intersection function on two DFA who do not share the same exact set of characters sigma.

Chapter 3

The NFA Class

3.1 Construction

This class bahaves like a non-deterministic finite-state automata and can compute acceptance for constructed NFAs. To do so, it constructs a DFA object that is equivalent to the NFA you describe using the constructor parameters, as computation with a DFA is much faster than trying to compute with an NFA. The creation of the computational DFA uses powerset construction but does not create any unreachable states. To construct an NFA object, you will need to provide the constructor with two template types, similar to the DFA object:

- The sigma type, the type that you will use as characters for tranisions, referred to as S_t.
- The state type, the type you will use to represent states, referred to as Q_t.

Note: Just the same as the DFA object, both given types **MUST** have an std::to_string function overwritten to work, so if you want to use a class that does not already have an std::to_string function written for it, create one for your classes in the STL_Helper.h file.

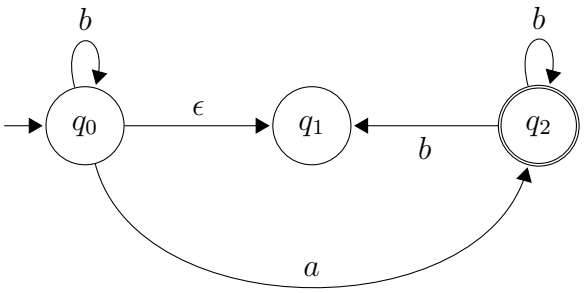
Next, the constructor must be provided with the following parameters in this order:

- std::unordered_set< S_t >, which is your sigma (set of valid transitioning characters)
- std::unordered_set< Q_t >, which is the set of your states

- Q_t, your initial state
- std::unordered_set< Q_t >, which is the set of accepting states (subset of the set of states)
- std::unordered_map< std::pair< Q_t, S_t >, std::unordered_set<Q_t> >, which will act as your delta, or transition function.
- S_t, a character that will be treated as epsilon. (MUST BE IN SIGMA)

This is rather confusing to look at, so we will take the following NFA:

$$\Sigma = a, b, \epsilon$$



and represent it using the NFA class, using std::string as both the sigma and state type:

```
#include <iostream>
#include "RegLang.h"

typedef std::string S_t;
typedef std::string Q_t;

int main()
{
    S_t a = "a";
    S_t b = "b";
    S_t ep = ""; //Epsilon
    Q_t q0 = "q0";
    Q_t q1 = "q1";
    Q_t q2 = "q2";

    std::unordered_set< S_t > S {a, b, ep};
    std::unordered_set< Q_t > Q {q0, q1, q2};
    std::unordered_set< Q_t > F {q1};

    D_t delta;
    delta[{q0, a}] = {q2};
    delta[{q0, b}] = {q0};
    delta[{q0, ep}] = {q1};
    delta[{q2, b}] = {q2, q1};

    NFA< S_t, Q_t > N(S, Q, q0, F, delta, ep);

    return 0;
}
```

You can also construct an NFA using the copy constructor like so:

```
NFA< S_t, Q_t > N(S, Q, q0, F, delta, ep);
NFA< S_t, Q_t >N_2(N); //N_2 is now a copy of NFA N.
```

You can also use the assignment operator to make an NFA into a copy of another NFA:

```
NFA< S_t, Q_t > N(S, Q, q0, F, delta, ep);

// Assuming S_, Q_, q0_, F_, delta_, and ep_are all valid.
NFA< S_t, Q_t > N_2(S_, Q_, q0_, F_, delta_, ep_);

N_2 = N; //N_2 is now a cppy of NFA N.
```

The main difference between DFA and NFA construction besides epsilon is the delta construction. Due to the fact that an NFA can have multiple transitions

with the same sigma character, we represent the delta transitions as returning sets of states instead of a singular state, so the set of states at the location of the std::pair of your state and sigma character holds every state that this character leads to with that state.

3.2 Accessing Members

The NFA object has 5 methods used to access members, however, once an NFA is created, you are not allowed to alter the contents of its members. i.e., once an NFA is made, it is meant to be immutable (the exception to this is the assignment operator). Each of the following methods returns that given member variable as a constant reference:

- sigma()
- states()
- initial_state()
- accept_states()
- delta()
- epsilon()

3.3 Computing Acceptance

An NFA object can be used to compute acceptance by providing the operator() method with a string of characters. This string is represented represented by an **std::vector< S_t >**, and currently is the only accepted input for acceptance. Using the example NFA we created above, we can compute the following:

```
// S, Q, q0, F, delta, and epsilon are all the same
// from the previous example NFA.
NFA< S_t, Q_t > N(S, Q, q0, F, delta, epsilon);

N({a, b});             // True
N({a, a});             // False
N({b, b, a});          // False
N({b, b, ep, ep, ep}); // True
N({});                 // True
```

Errors associated with operator():

- **NFA_Invalid_Sigma_Character_Error**, This error is thrown if you give the NFA an std::vector of characters to compute that contains a character that is not within its given set of characters Sigma.

3.4 Validating an NFA

This class includes a method, **valid()**, which returns a boolean that denotes whether the current NFA is a valid NFA. For the NFA to be considered valid, it must meet the following criteria:

- Set of states must be non-empty.
- All accepting states must be present within the set of states.
- Initial state must be within the set of states.
- All states and characters within delta transitions must be within the set of states and set of characters sigma respectively.
- All states within the set of states transitioned to within delta must be within the set of states.

If all of the above criteria are met, the **valid()** function will return true.

3.5 Kleene Star Method

the **kleene_star()** method returns a new NFA that is the kleene star of the current NFA, which means that a new initial state is made that has one epsilon transition to the current initial state, and all accepting states are given an epsilon transition to the old initial state if they did not already have one. This method takes one parameter:

- Q_t, the new initial state of the NFA.

This new initial state is needed because the NFA state type Q_t is a templated type so it is unreasonable to try and create a state name that does not already exist. **Errors associated with kleene_star():**

- **NFA_Invalid_Kleene_Star_Initial_State_Error**, This error is thrown if you give the kleene_star() method a new initial state parameter that is already a state within the NFA.

3.6 is_accepting() Method

This method takes in a state object (Q_t) and returns true if it is within the set of accepting states.

3.7 NFA Union Function

This function returns a new NFA< S_t, Q_t > which is the union of the two given NFA. This is done by making the given new initial state into the returned NFA's initials state, and this new NFA's initial state has two epsilon transitions, one to each initial state of the two given NFA.

The **nfa_union()** function is a templated function that takes two templated parameters in this order:

- S_t Sigma type of both NFA.
- Q_t State type of both NFA.

Note: both NFA must also share the exact same sigma set, and must both share the same S_t and Q_t types.

The parameters of the function are in order as follows:

- NFA< S_t, Q_t >, the first NFA.
- NFA< S_t, Q_t >, the second NFA.
- Q_t, the new initial state.

WARNING: Providing this function with either

- Two NFA that share any state names
- A new initial state that is already in the set of states for either provided NFA

will result in unintended behavior.

Here is an example of using the nfa_union() function:

```
// Assuming that Q0 and Q1 have no overlapping state names.
NFA< S_t, Q_t > N0(S, Q0, q00, F0, delta0, ep);
NFA< S_t, Q_t > N1(S, Q1, q01, F1, delta1, ep);

// Assuming that Q_t is std::string and that "qi" is not
// a state name in either Q0 or Q1.
NFA< S_t, Q_t > N2 = nfa_union< S_t, Q_t >(N0, N1, "qi");
```

Errors associated with nfa_union():

- **NFA_Union_Sigma_Mismatch_Error**, This error is thrown if you give the nfa_union() function two NFA parameters that do not share all values in sigma.

3.8 NFA Concatenation Function

The NFA concatenation function takes two NFA that share an S_t and Q_t type and returns a new NFA that is the concatenation of the two NFA. This is done by taking the first parameter NFA and "attatching" the second one onto the end of it by linking all accept states in the first NFA to the initial state of the second NFA using epsilon transitions. This also makes all accepting states in the first NFA no longer accepting states, and only the accepting states in the second NFA will be accepting states in the returned NFA. The initial state of the returned NFA will be the same intitial state as in the first NFA.

The **nfa_concat()** function is a templated function that takes two template parameters in this order:

- S_t Sigma type of both NFA.
- Q_t State type of both NFA.

Note: both NFA must also share the exact same sigma set, and must both share the same S_t and Q_t types.

The parameters of the function are in order as follows:

- NFA< S_t, Q_t >, the first NFA.

- NFA< S_t, Q_t >, the second NFA.

WARNING: Providing this function with NFA that share any state names will result in unintended behavior.

Here is an example of using the nfa_concat() function:

```
// Assuming that Q0 and Q1 have no overlapping state names.
NFA< S_t, Q_t > N0(S, Q0, q00, F0, delta0, ep);
NFA< S_t, Q_t > N1(S, Q1, q01, F1, delta1, ep);

NFA< S_t, Q_t > N2 = nfa_concat< S_t, Q_t >(N0, N1);
```

Errors associated with nfa_concat():

- **NFA_Concatenation_Sigma_Mismatch_Error**, This error is thrown if you give the nfa_concat() function two NFA parameters that do not share all values in sigma.

3.9 NFA to DFA Conversion

The NFA to DFA conversion method, **to_dfa()**, is extremely simple as the DFA equivalent for this NFA already exists, as it is used to compute acceptance for this NFA. As such, this method simply returns a DFA< S_t, std::unordered_set< Q_t > > that is a copy of this computation DFA within the NFA class. The state type of the DFA is an std::unordered_set of the state type used by the NFA because this DFA was created using powerset construction, which in turn uses the epsilon closure's of states, which are sets of states.

Here is an example of using the to_dfa() method:

```
// Assuming that all parameters are valid.
NFA< S_t, Q_t > N(S, Q, q0, F, delta, ep);

DFA< S_t, std::unordered_set< Q_t > > M = N.to_dfa();
```

3.10 NFA to Regex Conversion

The **to_regex()** method uses Kleene's Theorem to convert an NFA into a Regular Expression by first converting the NFA into a generalized NFA using two given state names that are not already in the NFA, and then moves through and removes all states within the NFA and turning transitions into regular expressions until there is only one regular expression remaining that moves from the first given state name to the second given state name.

This method requires three parameters in this order:

- Q_t, the first given state
- Q_t, the second given state
- S_t, a sigma character representation that you want to use to represent the emptyset for the created Regex object.

The emptyset representation required as a parameter is in case the NFA evaluates to the emptyset, so it can have a valid Regex object.

Here is an example of using the to_regex() method:

```
// Assuming that all parameters are valid.
NFA< S_t, Q_t > N(S, Q, q0, F, delta, ep);

// Assuming that S_t and Q_t are both std::string and
// neither "qi" nor "qa" are state names in the NFA,
// and "EMPTYSET" is not a character in sigma.
Regex R = N.to_regex("qi", "qa", "EMPTYSET");
```

Errors associated with to_regex():

- **NFA_To_Regex_Invalid_qi_Error**, This error is thrown if the first Q_t parameter you sent to the method is already in the set of states.
- **NFA_To_Regex_Invalid_qa_Error**, This error is thrown if the second Q_t parameter you sent to the method is already in the set of states.
- **NFA_To_Regex_Invalid_Emptyset_Error**, This error is thrown if the S_t parameter you sent to the method is already within the set of characters sigma.

Chapter 4

The Regex Class

4.1 Construction

This class acts like a regular expression, and uses the Thompson's construction algorithm to create an equivalent NFA that performs all computations, which then uses its own DFA to perform computations. Say we wanted to take the following regular expression:

$$aab*a$$

and represent it as a Regex object. This is very easy, and uses std::string for construction:

```
#include <iostream>
#include "RegLang.h"

int main()
{
    // Both are valid.
    Regex R0("aab*a");
    Regex R1 = "aab*a";

    // R2 is now a copy of R0.
    Regex R2(R0);

    return 0;
}
```

You can also use the assignment operator like so:

```
Regex R0 = "ab";
Regex R1 = "ba";

// R0 is now the same Regex as R1.
R0 = R1;

// R0 is now a Regex representing the
// regular expression "bbba".
R0 = "bbba";
```

These constructors assumes you are using an epsilon that is the empty string, "", and an emptyset that is the string of the null character "\0". Commonly epsilon is the empty string, and it is very uncommon for a regular expression to actually use the emptyset as a character so I made it something I assume would not be frequently used, the null character. The emptyset character is mainly in place just in case you use the NFA's to_regex() method and that specific NFA evaluated to an emptyset regular expression. You may specify these parameters upon construction if you wish:

```
// regular expression of "aab*a"
// epsilon character of "ep"
// empty set character of "EMPTYSET"
Regex R("aab*a", "ep", "EMPTYSET");
```

4.2 Valid Regex Operations

This section will go over all valid operations that can be used within a Regex's regular expression. **NOTE:** the ' ' space character IS A VALID CHARACTER.

$$\therefore \text{"}a\ b*\text{"} \not\equiv \text{"}ab*\text{"}$$

4.2.1 Concatenation

Since concatenation is implied by characters in regex being next to one another, there is no concatenation operator.

$$a \cdot a \equiv aa$$

and since this is true, just use "aa" as your regular expression.

4.2.2 Parenthases

Parenthases will determine if operators will be used on singular characters or parenthasized regex expressions, with an exception being the union operator, '|'. For example, in the following regex:

$$"ab*a"$$

the kleene star operator only applies to the character 'b'. But in this Regex:

$$"(ab)*a$$

the kleene star operator applies to the sub-regex "ab". This is consistent with all operators except for union, as mentioned before.

4.2.3 Kleene Star

The kleene star operator has normal kleene star functionality. Examples:

b*	accepts: "", "b", "bb", "bbb", "bbbb", ...
a(ba)*	accepts: "a", "aba", "ababa", "abababa", ...
(a|b|c)*	accepts: all strings of the characters 'a', 'b', or 'c' of any length and order.

Errors associated with the kleene star operator:

- **Regex_NFA_Construction_Error**, This error is thrown if you have two or more '*' or '+' operators appear next to each other, and because this makes no sense, it throws an error.

4.2.4 Union

The union operator in this Regex class is the '|' character. It also does not follow traditional operator rules, and applies to everything behind it until it finds and uneven parenthases (meaning it is within parenthases) or another concatenation operator. Some examples:

a|b	accepts: "a", "b".
abc(abb)|c	accepts: "abcabb", "c".
(abc(12(3)|456)	accepts: "abc123", "abc456".
a(b*|(cd)*e)	accepts: "a", "ae", "ab", "acde", "abb", "acdcde", ...
a|	accepts: "", "a".

The final example above functions in that way due fact that if the other side of a union operator has no expression, it will assume epsilon. This would also work with custom epsilon characters, as they are removed before creating a computational NFA so that the construction is consistent.

This rule of not only applying to the previous expression, be it a character or a parenthasized expression, is consistent with all other extended operators in this Regex class, as they are all converted into the regular kleene star and union operators before conversion into an NFA.

4.2.5 Powers

You can apply powers onto singular characters or parenthasized expressions to avoid typing out long stretches of the same expressions. Examples:

a{3}	accepts: "aaa".
a{0}	accepts: "".
(ab|bc){2}	accepts: "abab", "abbc", "bcab", "bcbc".

This also works for ranges of powers, and the range is separated by a comma.

a{0,4}	accepts: "", "a", "aa", "aaa", "aaaa".

This also works if you want a minimum power or more by giving the power a

number that will serve as the minimum power and then following it with only a comma like so:

a{2,}	accepts: "aa", "aaa", "aaaa", "aaaaa", ...
(a|b){2,}	accepts: "aa", "ab", "bb", "ba", "aba", "abb", ...

This all works because in the first initial scan of the Regex string before the NFA conversion, it takes all of the "fancy" operators like powers and converts them into regular operators like so:

$$"a\{4\}" \rightarrow "(aaaa)"$$
$$"a\{2,5\}" \rightarrow "(aa|aaa|aaaa|aaaaa)"$$
$$"a\{4,\} \rightarrow "(aaaa)(a)*"$$

Errors associated with powers:

- **Regex_Invalid_Power_Error**, This error is thrown if you incorrectly construct a power operator.

4.2.6 Ranges

These are not operators that perform actions on existing expressions, rather these allow for the easy and fast creation of otherwise large expressions. These ranges allow for any alphanumeric characters and basically serve as large chains of unions. **Note:** do not put any spaces within the ranges, it will throw a regex range error. The accepted characters for each character type is as follows:

$$\begin{aligned} \text{Lowercase Letters: } & a, b, c, d, e, ..., x, y, z \\ \text{Upperase Letters: } & A, B, C, D, E, ..., X, Y, Z \\ \text{Numbers: } & 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \end{aligned}$$

Any characters not listed above are not valid within a range.

If you want individual characters, you can do the following:

[1a4cL]	accepts: "1", "a", "4", "c", "L"

As you can see, the characters can be input in any order. You are even allowed to put in one character more than once, but there is no point in doing so.

[aaaab] accepts: "a", "b"

Now for the purpose which gives them their name, you can input ranges of characters. **Note:** The ranges go off of the character's numeric ascii values, so the first and last characters in a range must be of the same alphanumeric type, so lowercase letters must be paired with other lowercase letters, numbers must be paired with numbers, and uppercase letters must be paired with other numbers. You must also make sure that the first character in a range comes before the second character in the ascii table. If all of these rules are follwed, the expression created will accept all characters between the start and end character including both the start and end character.

[0-9] accepts: any single digit number.

[a-z] accepts: any lowercase letter.

[A-Z] accepts: any uppercase letter.

[0-9a-zA-Z] accepts: any alphanumeric character.

[1-9][0-9] accepts: any non-zero two digit number.

As you can see above, to use multiple ranges in a single range operation, simply put them right next to eachother with no spaces. You can also mix and match them with single characters:

[0-9az] accepts: any single digit number, "a", "z".

[a-z02468A-Z] accepts: any letter or an even single digit number.

This all works because in the first pass of the Regex string given to the Regex object in either the constructor or the assignment operator will first take all "special" operations and macros such as the powers and these ranges, and convert them into regular expressions. Some examples:

$$\text{"[a-d]"} \rightarrow \text{"}(a|b|c|d)\text{"}$$
$$\text{"[axz0-4G-J]"} \rightarrow \text{"}(a|x|z|0|1|2|3|4|G|H|I|J)\text{"}$$

Errors associated with ranges:

- **Regex_Invalid_Range_Error**, This error is thrown if you incorrectly construct a range.

4.2.7 Optional Operator

The optional operator '?' will take a character or parenthasized expression and make it to where it will accept if that character/expression is present and it will accept if it is not. In other words, it will accept that expression or epsilon. Examples:

abb?c	accepts: "abc", "abbc".
abc(def)?	accepts: "abc", "abcdef"

This works because because in the first pass of the Regex string, it converts all of these optional operators and their related expressions into a union of that expression and epsilon. Examples:

$$\text{"a?"} \rightarrow \text{"}(a|)\text{"}$$
$$\text{"(a*b)?"} \rightarrow \text{"}((a * b)|)\text{"}$$

WARNING: Using the optional operator with the kleene star and positive operators will cause an error to be thrown. An error will also be thrown if an optional operator is applied to a power that accepts a minimum power, and for all other types of powers it will result in unintended behavior. To remedy this, simply enclose any kleene stars, positives, and powers in parenthases.

$$\text{"}a*?\text{"} \rightarrow \text{"}(a*)?\text{"}$$
$$\text{"}a+?\text{"} \rightarrow \text{"}(a+)?\text{"}$$
$$\text{"}(abc)\{2,4\}?\text{"} \rightarrow \text{"}((abc)\{2,4\})?\text{"}$$

Errors associated with the optional operator:

- **Regex_NFA_Construction_Error**, This error is thrown if you apply an optional operator directly after a kleene star, positive, and minimum power operators.

4.2.8 Positive Operator

The positive operator is simply the kleene star operator but it requires that the expression must exist at least once. Examples:

a+	accepts: "a", "aa", "aaa", "aaaa", "aaaaa", ...
a(bc)+	accepts: "abc", "abcbc", "abcbcbc", ...

This works because because in the first pass of the Regex string, it converts all of these positive operators and their related expressions into the expression itself followed by the expression with a kleene star. Examples:

$$\text{"a+"} \rightarrow \text{"}aa*\text{"}$$
$$\text{"ab+c"} \rightarrow \text{"}abb*c\text{"}$$
$$\text{"(abcd)+"} \rightarrow \text{"}(abcd)(abcd)*\text{"}$$

Errors associated with the positive operator:

- **Regex_NFA_Construction_Error**, This error is thrown if you have two or more '*' or '+' operators appear next to each other, and because this makes no sense, it throws an error.

4.2.9 Escape Character

From the above operators that I have included, we have now lost the ability to use the following characters in regular expressions:

'{', '}', '[', ']', '(', ')', '*', '+', '?'

And because of this, I have included an escape character to be able to use these special symbols as characters in a regex. This character is the forward slash '/' character. It also worsk on itself, so you can use the escape character as well. Examples:

//	accepts: "/"
/+*	accepts: "", "+", "++", "+++", ...

This also works on normal characters, but there is no real advantage to doing this at all.

/a/b	accepts: "ab"
/a*	accepts: "", "a", "aa", "aaa", ...

NOTE: ALL characters that do not require escape charcters can be used freely. This includes the space character ' ', at sign character '@', equals character '=', etc...

Errors associated with the escape character:

- **Regex_Invalid_Escape_Character_Error**, This error is thrown if you misplace an escape character, i.e. put an escape character somewhere it does not make sense to be.

4.3 Accessing Members

The Regex object has 4 methods used to access members. Each of the following methods returns that given member variable a copy of the member variable (all of these methods return std::string):

- **expression()**, returns the expression given to the Regex object during either construction or assignment.
- **regular_expression()**, returns the expression that is the result of converting all of the "fancy" operations into regular expressions and making the regular expression that is then turned into the computational NFA.
- **epsilon()**, returns the string representation of epsilon.
- **emptyset()**, returns the string representation of the emptyset.

4.4 Computing Acceptance

A Regex object can be used to compute acceptance by providing the operator() method with a string of characters. This string is represented represented

by an **std::vector< std::string >**, and can also be provided with only an **std::string** as well.

```
std::string a = "a";
std::string b = "b";

Regex R("a*b");

R({a, b});              // True
R({b});                 // True
R({a, a, a, b, a});     // False
R("ab");                // True
R("b");                 // True
R("aaaba")              // False
```

Unlike the other acceptance methods on the NFA and DFA classes, this one will not throw an error if given a character that was not specified in the language of the regular expression. The language, or "sigma" of a Regex is always just that of any characters present in the regex that are not specified as operators or special symbols. If the computational NFA in the Regex were to throw an error trying to compute a character that does not exist within the NFA, the Regex will catch it and return false.

4.5 Regex to NFA Conversion

The Regex class contains a method **to_nfa()** which returns a copy of the computational NFA that is created during construction, so this conversion method does not need to even create a new NFA, as it already exists. The NFA will ALWAYS be of type NFA< std::string, std::string >. Example:

```
Regex R("a*b");

NFA< std::string, std::string > N = R.to_nfa();
```