1 Trabajos Futuros

1. Extender las pruebas unitarias.
2. 'Factorizar' los metodos (y responsabilidades) en NFA y DFA a una clase `FiniteAutomata`
3. Agregar la clase State y extender las clases correspondientes. Muy útil (probablemente) porque seria mas elegante disponer de for(const auto& transition: state), LabelType label = state.getLabel(), etc.
4. Como se exige que el operador concatenación sea explicito se puede extender el concepto de símbolo del alfabeto de caracteres char a cadena de caracteres std::string. O directamente se crea una clase adecuada, Symbol.
5. Extender las expresiones regulares permitiendo el uso del operador 'one-or-more', donde $a+ = a.a*$, y del operador 'zero-or-one', donde $a? = (\epsilon|a)$. Sus prioridades serian la misma del operador Kleene.

2 Punto de Entrada

Se indica el flujo mas superficial del sistema. Utiliza las bibliotecas `ShuntingYard::` y `Automata::`. Se encarga principalmente de obtener los datos desde el teclado y procesarlos.

```
#include <iostream>
#include <string>
#include "gtest/gtest.h"
#include "Hopcroft/Hopcroft.h"
#include "ShuntingYard/SimpleAlgorithm.h"
#include "Powerset/Powerset.h"
#include "Thompson/Thompson.h"
#include "DFA.h"

const std::string exitToken = "$";

void printIntroduction()
{
  //std::string input = "(a|b)*.a.b.b";
  //std::string input = "a.(a|b)*.a.b.b";
  //std::string input = "f.e.d.e.r.i.c.o";
  std::cout << "Available operators: " << std::endl;
  std::cout << "1. Parenthesis, <(> and <)>" << std::endl;
  std::cout << "2. Kleene, <*>" << std::endl;
  std::cout << "3. Alternative, <|>" << std::endl;
  std::cout << "4. Concatenation, <.>" << std::endl;
  std::cout << std::endl;
  std::cout << "Some examples: " << std::endl;
  std::cout << "(a|b)*.a.b.b -- All strings ending with abb" << std::endl;
  std::cout << "a.(a|b)*.b -- All string starting with a and ending with b" << std::endl;
  std::cout << "f.e.d.e.r.i.c.o -- Only the string federico" << std::endl;
  std::cout << std::endl;
  std::cout << "Some observations:" << std::endl;
  std::cout << "Alphabet will be calculated from the input" << std::endl;
  std::cout << "The concatenation operator is *explicitly* symbolized with <.>" << std::endl;
  std::cout << "The string <" << exitToken << "> must not belong to the alphabet" << std::endl;
  std::cout << std::endl;
}

void printConclution()
{
  std::cout << "Exiting..." << std::endl;
}

void testDFA(const Automata::DFA& dfa)
{
  Automata::DFARunner runner(dfa);

  std::string input;
  do
  {
    std::cout << "Input string (" << exitToken << " to exit, " << Automata::Epsilon << " to test empty
    string): " << std::endl;
    std::cin >> input;
    if(input == Automata::Epsilon)
      input = "";
    if(input != exitToken)
    {
```

```
      if(runner.run(input))
        std::cout << "Input <" << input << "> acepted." << std::endl;
      else
        std::cout << "Input <" << input << "> rejected." << std::endl;
    }

  }while(input != exitToken);
}

bool proccessACase()
{
  std::cout << "RE (" << exitToken << " to exit): " << std::endl;

  std::string input;
  std::cin >> input;
  if(input == exitToken)
      return false;

  std::cout << "Input: " << input << std::endl;

  const auto postfixInput = ShuntingYard::SimpleAlgorithm::apply(input);
  std::cout << "Postfix Expression: " << postfixInput << std::endl;

  const auto nfa = Automata::Thompson::apply(postfixInput);
  std::cout << "Resulting NFA from Thompson's Construction: " << std::endl << nfa << std::endl;

  const auto dfa = Automata::Powerset::apply(nfa);
  std::cout << "Resulting DFA from Subset Construction: " << std::endl << dfa << std::endl;

  const auto minDfa = Automata::Hopcroft::apply(dfa);
  std::cout << "Resulting Minimal DFA from Hopcroft's Algorithm: " << std::endl << minDfa << std::endl;

  testDFA(minDfa);

  return true;
}

int main(int argc, char* argv[]) {
  ::testing::InitGoogleTest(&argc, argv);
  const bool runTests = false;

  printIntroduction();

  while(proccessACase());

  printConclution();

  if(runTests)
    return RUN_ALL_TESTS();
  else
    return EXIT_SUCCESS;
}
```

"source/CompilersTP1.cpp"

3 Biblioteca `ShuntingYard::`

La evaluación de las expresiones se realizan mediante el Algorithmo Shunting-Yard. La biblioteca consta de 3 archivos: `SimpleAlgorithm.h` y `SimpleAlgorithm.cpp` indican la cabezera y el archivo de codigo, `SimpleAlgorithm_test.cpp` simplemente almacena algunas pruebas unitarias contra la biblioteca.

```
#ifndef SHUNTINGYARD_SIMPLEALGORITHM_H_
#define SHUNTINGYARD_SIMPLEALGORITHM_H_
#include <string>
#include <vector>
#include <sstream>
#include <stack>
#include <queue>
#include <iostream>

namespace ShuntingYard {

class SimpleAlgorithm {
public:
```

```
  static std::string apply(const std::string&);

protected:
  using TokenType = std::string;
  struct OperatorData
  {
    const TokenType token;
    const unsigned int precedence;
    const bool associativity;
  };
  using ContainerType = std::vector<TokenType>;
  using OperatorsDataType = std::vector<OperatorData>;
  using OperatorStackType = std::stack<TokenType>;
  using OutputQueueType = std::queue<TokenType>;

  static const OperatorsDataType operatorsData;
  static const TokenType leftParenthesis;
  static const TokenType rightParenthesis;

  static ContainerType run(const ContainerType&);
  static void processLeftParenthesis(OperatorStackType&);
  static void processRightParenthesis(OperatorStackType&, OutputQueueType&);
  static void processOperator(const TokenType&, OperatorStackType&, OutputQueueType&);
  static bool isOperator(const TokenType&);
  static bool isLeftAssociative(const TokenType&);
  static unsigned int getPrecedence(const TokenType&);
  static bool hasGreaterPrecedence(const TokenType&, const TokenType&);
  static bool hasEqualPrecedence(const TokenType&, const TokenType&);
};

} /* namespace ShuntingYard */

#endif /* SHUNTINGYARD_SIMPLEALGORITHM_H_ */
```

"source/SimpleAlgorithm.h"

```
#include "SimpleAlgorithm.h"

namespace ShuntingYard {

const SimpleAlgorithm::OperatorsDataType SimpleAlgorithm::operatorsData({{"*", 500, true}, {".", 400, true
    }, {"|", 300, true}});
const SimpleAlgorithm::TokenType SimpleAlgorithm::leftParenthesis("(");
const SimpleAlgorithm::TokenType SimpleAlgorithm::rightParenthesis(")");

std::string SimpleAlgorithm::apply(const std::string& input)
{
  ContainerType inputTokens;
  inputTokens.push_back(leftParenthesis);
  for(const auto& c: input)
    inputTokens.push_back(TokenType(1, c)); // TODO Find a better way to build a TokenType from char
  inputTokens.push_back(rightParenthesis);

  ContainerType outputTokens = run(inputTokens);

  std::ostringstream oss;
  for(const auto& token: outputTokens)
    oss << token;

  return oss.str();
}

SimpleAlgorithm::ContainerType SimpleAlgorithm::run(const ContainerType& tokens)
{
  OperatorStackType operators;
  OutputQueueType output;

  for(const auto& token: tokens)
  {
    if(leftParenthesis == token) processLeftParenthesis(operators);
    else if(rightParenthesis == token) processRightParenthesis(operators, output);
    else if(isOperator(token)) processOperator(token, operators, output);
    else output.push(token);
  }

  while(!operators.empty())
```

```
40      output.push(operators.top()), operators.pop();

42    ContainerType outputTokens;
      while(!output.empty())
44      outputTokens.push_back(output.front()), output.pop();

46    return outputTokens;
}
48
void SimpleAlgorithm::processLeftParenthesis(OperatorStackType& operators)
50{
    operators.push(leftParenthesis);
52}

54void SimpleAlgorithm::processRightParenthesis(OperatorStackType& operators, OutputQueueType& output)
{
56    while(operators.top() != leftParenthesis)
        output.push(operators.top()), operators.pop();
58    operators.pop(); // pop leftParenthesis
}
60
void SimpleAlgorithm::processOperator(const TokenType& operatorToken, OperatorStackType& operators,
    OutputQueueType& output)
62{
    TokenType topOperator = operators.top();
64    while(leftParenthesis != topOperator &&
        ( hasGreaterPrecedence(topOperator, operatorToken) || (hasEqualPrecedence(topOperator, operatorToken
    ) && isLeftAssociative(topOperator)) )
66    )
        output.push(operators.top()), operators.pop(), topOperator = operators.top();
68    operators.push(operatorToken);
}
70
bool SimpleAlgorithm::isOperator(const TokenType& token)
72{
    for(const auto& data: operatorsData)
74      if(data.token == token)
          return true;
76    return false;
}
78
bool SimpleAlgorithm::isLeftAssociative(const TokenType& operatorToken)
80{
    for(const auto& data: operatorsData)
82      if(data.token == operatorToken)
          return data.associativity;
84    throw std::invalid_argument("Token does not correspond to a registered operator");
}
86
unsigned int SimpleAlgorithm::getPrecedence(const TokenType& operatorToken)
88{
    for(const auto& data: operatorsData)
90      if(data.token == operatorToken)
          return data.precedence;
92    throw std::invalid_argument("Token does not correspond to a registered operator");
}
94
bool SimpleAlgorithm::hasGreaterPrecedence(const TokenType& operatorA, const TokenType& operatorB)
96{
    return getPrecedence(operatorA) > getPrecedence(operatorB);
98}

100bool SimpleAlgorithm::hasEqualPrecedence(const TokenType& operatorA, const TokenType& operatorB)
{
102    return getPrecedence(operatorA) == getPrecedence(operatorB);
}
104
} /* namespace ShuntingYard */
```

"source/SimpleAlgorithm.cpp"

```
1 #include "../gtest/gtest.h"
  #include "SimpleAlgorithm.h"
3
  TEST(ShuntingYardTests, OneOrLessOperators)
5 {
```

```
  ASSERT_EQ("a", ShuntingYard::SimpleAlgorithm::apply("a"));
  ASSERT_EQ("a", ShuntingYard::SimpleAlgorithm::apply("(a)"));
  ASSERT_EQ("a*", ShuntingYard::SimpleAlgorithm::apply("a*"));
  ASSERT_EQ("ab|", ShuntingYard::SimpleAlgorithm::apply("a|b"));
  ASSERT_EQ("ab.", ShuntingYard::SimpleAlgorithm::apply("a.b"));
}

TEST(ShuntingYardTests, OperatorsRelations)
{
  ASSERT_EQ("ab|*", ShuntingYard::SimpleAlgorithm::apply("(a|b)*"));
  ASSERT_EQ("ab.*", ShuntingYard::SimpleAlgorithm::apply("(a.b)*"));
  ASSERT_EQ("a**", ShuntingYard::SimpleAlgorithm::apply("a**"));
  ASSERT_EQ("ab|c|", ShuntingYard::SimpleAlgorithm::apply("a|b|c"));
  ASSERT_EQ("ab.c.*", ShuntingYard::SimpleAlgorithm::apply("(a.b.c)*"));
  ASSERT_EQ("a*b*|", ShuntingYard::SimpleAlgorithm::apply("a*|b*"));
  ASSERT_EQ("a*b*.", ShuntingYard::SimpleAlgorithm::apply("a*.b*"));
  ASSERT_EQ("ab*|", ShuntingYard::SimpleAlgorithm::apply("a|b*"));

}

TEST(ShuntingYardTests, LeftAssociativity)
{
  ASSERT_EQ("ab|c|", ShuntingYard::SimpleAlgorithm::apply("a|b|c"));
  ASSERT_NE("abc||", ShuntingYard::SimpleAlgorithm::apply("a|b|c"));
  ASSERT_EQ("ab.c.", ShuntingYard::SimpleAlgorithm::apply("a.b.c"));
  ASSERT_NE("abc..", ShuntingYard::SimpleAlgorithm::apply("a.b.c"));

}

TEST(ShuntingYardTests, ComplexOnes)
{
  ASSERT_EQ("ab|*a.b.b.", ShuntingYard::SimpleAlgorithm::apply("(a|b)*.a.b.b"));
  ASSERT_EQ("abc*|.", ShuntingYard::SimpleAlgorithm::apply("a.(b|c*)"));
  ASSERT_EQ("abc|*.", ShuntingYard::SimpleAlgorithm::apply("a.(b|c)*"));
  ASSERT_EQ("aa.b|*abb.|*.", ShuntingYard::SimpleAlgorithm::apply("(a.a|b)*.(a|b.b)*"));

}
```

"source/SimpleAlgorithm_test.cpp"

Biblioteca `Automata::`

Esta biblioteca engloba las clases relacionadas con la Teoría de Automatas además de los algoritmos que trabajan sobre ellos, por ejemplo `TransitionTable`, `NFA`, `DFA`, `Thompson`, etc.

3.1 Datos comunes a todos los bloques

Almacena los tipos personalizados para aumentar la legibilidad del codigo (claramente StateSetType describe mas que std::set<unsigned int>. Además de algunas funciones auxiliares.

```
#ifndef COMMON_H_
#define COMMON_H_

#include <set>
#include <string>
#include <iostream>
#include <sstream>

namespace Automata
{
  using StateType = unsigned int;
  using StateSetType = std::set<StateType>;
  using SymbolType = std::string;
  using SymbolSetType = std::set<SymbolType>;
  using AlphabetType = SymbolSetType;

  const SymbolType Epsilon = "#";

  template <class T>
  std::ostream& operator<<(std::ostream& os, const std::set<T>& s)
  {
    os << "{";
    if(!s.empty())
```

```
24      {
26        auto iter = std::begin(s);
          os << *iter;
28        ++iter;
          while(iter != std::end(s))
30        {
            os << ", " << *iter;
32          ++iter;
          }
34      }
        return os << "}";
36    }

      template <class T>
38    std::string to_string(const T& x)
      {
40      std::ostringstream oss;
        oss << x;
42      return oss.str();
      }
44  }

46

  #endif /* COMMON_H_ */
```

"source/Common.h"

3.2 Clase `TransitionTable`

Su responsabilidad es almacenar los datos de un Automata (Deterministico o no). Las clases internas son iteradores sobre los datos.

```
1   #ifndef TRANSITIONTABLE_H_
    #define TRANSITIONTABLE_H_
3
    #include <vector>
5   #include <set>
    #include <string>
7   #include <map>
    #include <iostream>
9   #include <iomanip>

11  #include "Common.h"

13  namespace Automata
    {
15
    class TransitionTable
17  {
    private:
19    using TableType = std::vector<std::vector<StateSetType>>;
      using SymbolToIndexType = std::map<SymbolType, size_t>;
21
      TableType _table;
23    SymbolSetType _symbols;
      SymbolToIndexType _symbolsMapping;
25
    public:
27    TransitionTable();
      TransitionTable(const TransitionTable&);
29    TransitionTable& operator=(TransitionTable);
      StateType addState();
31    void addSymbol(const SymbolType&);
      void addTransition(const StateType&, const SymbolType&, const StateType&);
33    const StateSetType& getTransition(const StateType&, const SymbolType&) const;
      StateSetType& getTransition(const StateType&, const SymbolType&);
35    bool isValidState(const StateType&) const;
      bool isValidSymbol(const SymbolType&) const;
37    ~TransitionTable() = default;
      // Iterators
39    class StateIterator
      {
41    private:
        size_t _maxIndex;
43      size_t _currentIndex;
```

```
45    public:
      StateIterator(size_t);
47    StateIterator(size_t, size_t);
      StateIterator(const StateIterator&);
49    StateIterator& operator=(StateIterator);
      StateType operator*() const;
51    void operator++();
      bool operator==(const StateIterator&) const;
53    bool operator!=(const StateIterator&) const;
      ~StateIterator() = default;
55  };
    class TransitionIterator
57  {
    private:
59    const TransitionTable* _transitionTable;
      StateType _state;
61    SymbolSetType::const_iterator _symbolIterator;

63  public:
      TransitionIterator(const TransitionTable&, const StateType&);
65    TransitionIterator(const TransitionTable&, const StateType&, const SymbolSetType::const_iterator&);
      TransitionIterator(const TransitionIterator&);
67    TransitionIterator& operator=(TransitionIterator);
      std::pair<SymbolType, StateSetType> operator*() const;
69    void operator++();
      bool operator==(const TransitionIterator&) const;
71    bool operator!=(const TransitionIterator&) const;
      ~TransitionIterator() = default;
73  };
    class TransitionIteratorTag
75  {
    public:
77    const TransitionTable& _transitionTable;
      const StateType& _state;
79    TransitionIteratorTag(const TransitionTable& transitionTable, const StateType& state)
      :_transitionTable(transitionTable), _state(state)
81    {}
    };
83  TransitionIteratorTag getTransitions(const StateType&) const;
    // Friends functions
85  friend std::ostream& operator<<(std::ostream&, const TransitionTable&);
    friend void swap(TransitionTable& lhs, TransitionTable& rhs);
87  friend StateIterator begin(const TransitionTable&);
    friend StateIterator end(const TransitionTable&);
89  friend TransitionIterator begin(const TransitionIteratorTag&);
    friend TransitionIterator end(const TransitionIteratorTag&);
91 };

93 } /* namespace Automata */

95 #endif /* TRANSITIONTABLE_H_ */
```

"source/TransitionTable.h"

```
1  #include "TransitionTable.h"

3  namespace Automata {

5  TransitionTable::TransitionTable()
   :_table(), _symbols({Epsilon})
7  {
     _symbolsMapping.insert(std::make_pair(Epsilon, 0));
9  }

11 TransitionTable::TransitionTable(const TransitionTable& other)
   :_table(other._table), _symbols(other._symbols), _symbolsMapping(other._symbolsMapping)
13 {
   }
15
   // Use pass-by-value to use copy-elision optimization
17 TransitionTable& TransitionTable::operator=(TransitionTable other)
   {
19   swap(other, *this);
     return *this;
21 }
```

```
23 StateType TransitionTable::addState()
24 {
25   _table.emplace_back();
26   for(size_t i = 0; i < _symbols.size(); i++)
27     _table.back().emplace_back();
28   return StateType(_table.size()-1);
29 }
30
31 void TransitionTable::addSymbol(const SymbolType& symbol)
32 {
33   if(symbol.empty())
34     throw std::invalid_argument("Empty symbol is not valid");
35
36   if(_symbols.find(symbol) == std::end(_symbols))
37   {
38     _symbols.insert(symbol);
39     _symbolsMapping.insert(std::make_pair(symbol, _symbolsMapping.size()));
40     for(auto& row: _table)
41       row.emplace_back();
42   }
43 }
44
45 void TransitionTable::addTransition(const StateType& start, const SymbolType& symbol, const StateType& end
       )
46 {
47   if(!isValidState(start) || !isValidState(end))
48     throw std::invalid_argument("Invalid state");
49   if(!isValidSymbol(symbol))
50     throw std::invalid_argument("Invalid symbol");
51   const size_t column = _symbolsMapping[symbol];
52   _table.at(start).at(column).insert(end);
53 }
54
55 const StateSetType& TransitionTable::getTransition(const StateType& start, const SymbolType& symbol) const
56 {
57   if(!isValidState(start))
58     throw std::invalid_argument("Invalid state");
59   if(!isValidSymbol(symbol))
60     throw std::invalid_argument("Invalid symbol");
61   const size_t column = _symbolsMapping.find(symbol)->second;
62   return _table.at(start).at(column);
63 }
64
65 StateSetType& TransitionTable::getTransition(const StateType& start, const SymbolType& symbol)
66 {
67   return const_cast<StateSetType&>(static_cast<const TransitionTable&>(*this).getTransition(start, symbol)
       );
68 }
69
70 bool TransitionTable::isValidState(const StateType& state) const
71 {
72   return 0 <= state && state < _table.size();
73 }
74
75 bool TransitionTable::isValidSymbol(const SymbolType& symbol) const
76 {
77   return _symbols.find(symbol) != std::end(_symbols);
78 }
79
80 TransitionTable::TransitionIteratorTag TransitionTable::getTransitions(const StateType& state) const
81 {
82   return TransitionIteratorTag(*this, state);
83 }
84
85
86 // StateIterator
87 TransitionTable::StateIterator::StateIterator(size_t maxIndex)
88 :_maxIndex(maxIndex), _currentIndex(0)
89 {
90 }
91
92 TransitionTable::StateIterator::StateIterator(size_t maxIndex, size_t currentIndex)
93 :_maxIndex(maxIndex), _currentIndex(currentIndex)
94 {
95 }
```

```
97  TransitionTable::StateIterator::StateIterator(const StateIterator& other)
    :_maxIndex(other._maxIndex), _currentIndex(other._currentIndex)
99  {
    }
101
    TransitionTable::StateIterator& TransitionTable::StateIterator::operator=(StateIterator other)
103 {
      std::swap(_maxIndex, other._maxIndex);
105   std::swap(_currentIndex, other._currentIndex);
      return *this;
107 }

109 StateType TransitionTable::StateIterator::operator*() const
    {
111   return static_cast<StateType>(_currentIndex);
    }
113
    void TransitionTable::StateIterator::operator++()
115     {
      _currentIndex++;
117     }

119 bool TransitionTable::StateIterator::operator==(const StateIterator& other) const
        {
121   return _currentIndex == other._currentIndex;
      }
123
    bool TransitionTable::StateIterator::operator!=(const StateIterator& other) const
125     {
      return !operator==(other);
127     }

129 // TransitionIterator
    TransitionTable::TransitionIterator::TransitionIterator(const TransitionTable& transitionTable, const
        StateType& state)
131 :_transitionTable(&transitionTable), _state(state), _symbolIterator(std::begin(transitionTable._symbols))
    {
133 }

135 TransitionTable::TransitionIterator::TransitionIterator(const TransitionTable& transitionTable, const
        StateType& state,
        const SymbolSetType::const_iterator& iterator)
137 :_transitionTable(&transitionTable), _state(state), _symbolIterator(iterator)
    {
139 }

141 TransitionTable::TransitionIterator::TransitionIterator(const TransitionIterator& other)
    :_transitionTable(other._transitionTable), _state(other._state), _symbolIterator(other._symbolIterator)
143 {
    }
145
    TransitionTable::TransitionIterator& TransitionTable::TransitionIterator::operator=(TransitionIterator rhs
        )
147 {
      std::swap(_transitionTable, rhs._transitionTable);
149   std::swap(_state, rhs._state);
      std::swap(_symbolIterator, rhs._symbolIterator);
151   return *this;
    }
153
    std::pair<SymbolType, StateSetType> TransitionTable::TransitionIterator::operator*() const
155 {
      const SymbolType symbol = *_symbolIterator;
157   const StateSetType states = _transitionTable->getTransition(_state, symbol);
      return std::make_pair(symbol, states);
159 }

161 void TransitionTable::TransitionIterator::operator++()
    {
163   ++_symbolIterator;
    }
165
    bool TransitionTable::TransitionIterator::operator==(const TransitionIterator& rhs) const
167 {
      return _transitionTable == rhs._transitionTable && _state == rhs._state && _symbolIterator == rhs.
```

```
    _symbolIterator;
}

bool TransitionTable::TransitionIterator::operator!=(const TransitionIterator& rhs) const
{
  return !(*this == rhs);
}

// Friend methods
std::ostream& operator<<(std::ostream& os, const TransitionTable& tt)
{
  size_t maxStringLength = 0;
  std::vector<std::vector<std::string>> body;
  for(const auto& row: tt._table)
  {
    body.emplace_back();
    auto& body_row = body.back();
    for(const auto& transition: row)
    {
      const auto transitionString = to_string(transition);
      maxStringLength = std::max(maxStringLength, transitionString.size());
      body_row.push_back(transitionString);
    }
  }

  std::map<size_t, SymbolType> symbolMappingInverse;
  for(const auto& p: tt._symbolsMapping)
    symbolMappingInverse[p.second] = p.first;

  for(const auto& p: symbolMappingInverse)
  {
    os << std::setw(maxStringLength) << std::setfill(' ') << p.second;
  }
  os << std::endl;

  StateType state = 0;
  for(const auto& row: body)
  {
    os << std::setw(3) << std::setfill(' ') << state++;
    for(const auto& transition: row)
    {
      os << std::setw(maxStringLength) << std::setfill(' ') << transition;
    }
    os << std::endl;
  }

  return os;
}

void swap(TransitionTable& lhs, TransitionTable& rhs)
{
  std::swap(lhs._table, rhs._table);
  std::swap(lhs._symbols, rhs._symbols);
  std::swap(lhs._symbolsMapping, rhs._symbolsMapping);
}

TransitionTable::StateIterator begin(const TransitionTable& transitionTable)
{
  return TransitionTable::StateIterator(transitionTable._table.size());
}

TransitionTable::StateIterator end(const TransitionTable& transitionTable)
{
  const auto& numberOfStates = transitionTable._table.size();
  return TransitionTable::StateIterator(numberOfStates, numberOfStates);
}

TransitionTable::TransitionIterator begin(const TransitionTable::TransitionIteratorTag&
    transitionIteratorTag)
{
  const TransitionTable& transitionTable = transitionIteratorTag._transitionTable;
  const StateType& state = transitionIteratorTag._state;
  return TransitionTable::TransitionIterator(transitionTable, state);
}

TransitionTable::TransitionIterator end(const TransitionTable::TransitionIteratorTag&
```

```
    transitionIteratorTag)
{
  const TransitionTable& transitionTable = transitionIteratorTag._transitionTable;
  const StateType& state = transitionIteratorTag._state;
  return TransitionTable::TransitionIterator(transitionTable, state, std::end(transitionTable._symbols));
}

} /* namespace Automata */
```

"source/TransitionTable.cpp"

```
#include "gtest/gtest.h"
#include "TransitionTable.h"

#include <iostream>

using Automata::Epsilon;
using Automata::StateType;
using Automata::StateSetType;
using Automata::SymbolType;
using Automata::TransitionTable;


TEST(TransitionTableTest, AddState)
{
  TransitionTable tt;
  StateType newState = tt.addState();
  ASSERT_TRUE(tt.isValidState(newState));
}

TEST(TransitionTable, AddSymbolAndIsValidSymbol)
{
  TransitionTable tt;
  tt.addSymbol(SymbolType("a"));
  ASSERT_TRUE(tt.isValidSymbol("a"));
  ASSERT_FALSE(tt.isValidSymbol("b"));
}

TEST(TransitionTable, ExceptionOnEmpty)
{
  TransitionTable tt;
  ASSERT_ANY_THROW(tt.addSymbol(""));
}

TEST(TransitionTable, AddTransitionAndGetTransition)
{
  TransitionTable tt;
  const auto source = tt.addState();
  const auto end = tt.addState();
  const auto symbol1 = "a";
  const auto symbol2 = "b";
  tt.addSymbol(symbol1);
  tt.addSymbol(symbol2);

  ASSERT_ANY_THROW(tt.addTransition(123, symbol1, end));
  ASSERT_ANY_THROW(tt.addTransition(source, "c", end));
  ASSERT_ANY_THROW(tt.addTransition(source, symbol1, 123));

  tt.addTransition(source, symbol1, end);
  tt.addTransition(source, symbol2, end);

  ASSERT_EQ(tt.getTransition(source, symbol1), StateSetType({end}));
  ASSERT_EQ(tt.getTransition(source, symbol2), StateSetType({end}));
  ASSERT_ANY_THROW(tt.getTransition(source, "c"));
}

TEST(TransitionTableTest, Iterator)
{
  TransitionTable tt;
  StateSetType states;
  for(int i = 0; i < 10; i ++)
    states.insert(tt.addState());

  for(const auto& state: tt)
    ASSERT_TRUE(states.find(state) != std::end(states));
}
```

"source/TransitionTable_test.cpp"

3.3 Clase `NFA`

Su responsabilidad es almacenar un Automata No Deterministico. Es inmutable. Dispone de 2 clases iteradoras, una sobre los estados y otra sobre las transitiones de un estado. Tambien se dispone de las clases NFARunner, para alimentar una cadena al automata, NFABuilder, para la construcción de un NFA, y EpsilonClosure, para calcular las cerraduras de los estados.

```
#ifndef NFA_H_
#define NFA_H_

#include <queue>
#include <iostream>

#include "TransitionTable.h"
#include "Common.h"

namespace Automata {

class EpsilonClosure;
template <class LabelType> class NFABuilder;

class NFA
{
private:
  const StateType _initialState;
  const StateSetType _finalStates;
  const TransitionTable _transitions;

public:
  NFA(const NFA&);
  NFA(const TransitionTable&, const StateType&, const StateSetType&);
  StateType getInitialState() const;
  const StateSetType& getFinalStates() const;
  StateSetType getFinalStates();
  const StateSetType& move(const StateType&, const SymbolType&) const;
  StateSetType move(const StateType&, const SymbolType&);
  TransitionTable::TransitionIteratorTag getTransitions(const StateType&) const;
  ~NFA() = default;
  // Friend classes
  friend class EpsilonClosure;
  template <class LabelType> friend class NFABuilder;
  // Friend methods
  friend std::ostream& operator<<(std::ostream&, const NFA&);
  friend TransitionTable::StateIterator begin(const NFA&);
  friend TransitionTable::StateIterator end(const NFA&);
  friend TransitionTable::TransitionIterator begin(const TransitionTable::TransitionIteratorTag&);
  friend TransitionTable::TransitionIterator end(const TransitionTable::TransitionIteratorTag&);
};

template <class LabelType>
class NFABuilder
{
protected:
  using TransitionType = std::tuple<LabelType, SymbolType, LabelType>;

  std::vector<TransitionType> _transitions;
  LabelType _initialStateLabel;
  bool _initialStateLabelSet;
  typename std::set<LabelType> _finalStatesLabels;

public:
  NFABuilder():_transitions(), _initialStateLabel(), _initialStateLabelSet(false), _finalStatesLabels() {}
  NFABuilder(const NFABuilder&) = delete;
  NFABuilder& operator=(const NFABuilder&) = delete;
  void setInitialStateLabel(const LabelType& initialStateLabel)
  {
    _initialStateLabel = initialStateLabel;
    _initialStateLabelSet = true;

  }
  void addFinalStateLabel(const LabelType& finalStateLabel)
```

```
65    {
66      _finalStatesLabels.insert(finalStateLabel);
67    }
68    void addTransition(const LabelType& startLabel, const SymbolType& symbol, const LabelType& endLabel)
69    {
70      _transitions.push_back(std::make_tuple(startLabel, symbol, endLabel));
71    }
72    NFA build()
73    {
74      if(_initialStateLabelSet == false)
75        throw std::invalid_argument("Initial state label was not defined");
76
77      const auto alphabet = buildAlphabet();
78
79      TransitionTable transitionTable;
80      const auto labelStateMapping = buildMapping(transitionTable);
81
82      for(const auto& symbol: alphabet)
83        transitionTable.addSymbol(symbol);
84
85      for(const auto& t: _transitions)
86      {
87        const auto startLabel = std::get<0>(t);
88        const auto startState = labelStateMapping.find(startLabel)->second;
89        const auto endLabel = std::get<2>(t);
90        const auto endState = labelStateMapping.find(endLabel)->second;
91        const auto symbol = std::get<1>(t);
92
93        transitionTable.addTransition(startState, symbol, endState);
94      }
95
96      StateSetType finalStates;
97      for(const auto& finalStateLabel: _finalStatesLabels)
98      {
99        const auto iter = labelStateMapping.find(finalStateLabel);
100       if(iter != std::end(labelStateMapping))
101         finalStates.insert(iter->second);
102     }
103
104     StateType initialState = labelStateMapping.find(_initialStateLabel)->second;
105
106     return NFA(transitionTable, initialState, finalStates);
107   }
108   ~NFABuilder() = default;
109
110 protected:
111   AlphabetType buildAlphabet() const
112   {
113     AlphabetType alphabet;
114     for(const auto& t: _transitions)
115     {
116       const SymbolType symbol = std::get<1>(t);
117       if(symbol != Epsilon)
118         alphabet.insert(symbol);
119     }
120     return alphabet;
121   }
122   std::map<LabelType, StateType> buildMapping(TransitionTable& transitionTable) const
123         {
124     std::map<LabelType, StateType> mapping;
125     mapping[_initialStateLabel] = transitionTable.addState();
126
127     for(const auto& t: _transitions)
128     {
129       const auto startLabel = std::get<0>(t);
130       if(mapping.find(startLabel) == std::end(mapping))
131         mapping[startLabel] = transitionTable.addState();
132
133       const auto endLabel = std::get<2>(t);
134       if(mapping.find(endLabel) == std::end(mapping))
135         mapping[endLabel] = transitionTable.addState();
136     }
137
138     return mapping;
139         }
140 };
```

```
141
142 class EpsilonClosure
143 {
144 private:
145   using MappingType = std::map<StateType, StateSetType>;
146
147   MappingType _mapping;
148
149 public:
150   EpsilonClosure(const NFA&);
151   const StateSetType& getClosure(const StateType&) const;
152   StateSetType getClosure(const StateType&);
153   StateSetType getClosure(const StateSetType&) const;
154
155   static StateSetType calculateClosure(const NFA&, const StateType&);
156 };
157
158 class NFARunner
159 {
160 private:
161   const NFA _nfa;
162
163 public:
164   NFARunner(const NFA&);
165   NFARunner(const NFARunner&) = delete;
166   NFARunner& operator=(NFARunner) = delete;
167   bool run(const std::string&);
168
169 private:
170   bool finalStateReached(const StateSetType&, const StateSetType&) const;
171 };
172
173 } /* namespace Automata */
174
175 #endif /* NFA_H_ */
```

"source/NFA.h"

```
1 #include "NFA.h"
2
3 namespace Automata {
4
5 NFA::NFA(const NFA& nfa)
6 : _initialState(nfa.getInitialState()), _finalStates(nfa.getFinalStates()), _transitions(nfa._transitions)
7 {
8 }
9
10 NFA::NFA(const TransitionTable& transitions, const StateType& initialState, const StateSetType&
      finalStates)
11 : _initialState(initialState), _finalStates(finalStates), _transitions(transitions)
12 {
13 }
14
15 StateType NFA::getInitialState() const
16 {
17   return _initialState;
18 }
19
20 const StateSetType& NFA::getFinalStates() const
21 {
22   return _finalStates;
23 }
24
25 StateSetType NFA::getFinalStates()
26 {
27   return const_cast<StateSetType&>(static_cast<const NFA&>(*this).getFinalStates());
28 }
29
30 const StateSetType& NFA::move(const StateType& startState, const SymbolType& symbol) const
31 {
32   return _transitions.getTransition(startState, symbol);
33 }
34
35 StateSetType NFA::move(const StateType& startState, const SymbolType& symbol)
36 {
37   return _transitions.getTransition(startState, symbol);
```

```
}

TransitionTable::TransitionIteratorTag NFA::getTransitions(const StateType& state) const
{
  return this->_transitions.getTransitions(state);
}

// Friend functions

std::ostream& operator<<(std::ostream& os, const NFA& nfa)
{
  os << nfa._transitions << std::endl;
  os << "Initial State: " << nfa.getInitialState() << std::endl;
  os << "Final States: " << nfa.getFinalStates() << std::endl;
  return os;
}

TransitionTable::StateIterator begin(const NFA& nfa)
{
  return begin(nfa._transitions);
}

TransitionTable::StateIterator end(const NFA& nfa)
{
  return end(nfa._transitions);
}


// EpsilonClosure
EpsilonClosure::EpsilonClosure(const NFA& nfa)
{
  for(const auto& state: nfa)
  {
    const StateSetType closure = calculateClosure(nfa, state);
    _mapping[state] = closure;
  }
}

const StateSetType& EpsilonClosure::getClosure(const StateType& state) const
{
  if(_mapping.find(state) == std::end(_mapping))
    throw std::invalid_argument("Invalid state");
  return _mapping.find(state)->second;
}

StateSetType EpsilonClosure::getClosure(const StateType& state)
{
  return const_cast<StateSetType&>(static_cast<const EpsilonClosure&>(*this).getClosure(state));
}

StateSetType EpsilonClosure::getClosure(const StateSetType& states) const
{
  StateSetType closures;
  for(const auto& state: states)
  {
    const auto closure = getClosure(state);
    for(const auto& state: closure)
      closures.insert(state);
  }

  return closures;
}

StateSetType EpsilonClosure::calculateClosure(const NFA& nfa, const StateType& initialState)
{
  StateSetType alreadyProcessed;

  StateSetType closure;
  closure.insert(initialState);

  std::queue<StateType> unprocessedStates({initialState});
  while(!unprocessedStates.empty())
  {
    const StateType state = unprocessedStates.front();
    unprocessedStates.pop();
    alreadyProcessed.insert(state);
```

```
    for(const auto& nextState: nfa.move(state, Epsilon))
    {
      if(alreadyProcessed.find(nextState) == std::end(alreadyProcessed))
        {
        closure.insert(nextState);
        unprocessedStates.push(nextState);
        }
    }
  }
  return closure;
}

NFARunner::NFARunner(const NFA& nfa)
:_nfa(nfa)
{
}

bool NFARunner::run(const std::string& input)
{
  const StateSetType& finalStates = _nfa.getFinalStates();
  EpsilonClosure closure(_nfa);

  StateSetType currentStates = closure.getClosure(_nfa.getInitialState());

  for(const char& c: input)
  {
    const std::string symbol(1, c);
    StateSetType nextStates;
    for(const auto& currentState: currentStates)
    {
      try
      {
        const StateSetType transition = _nfa.move(currentState, static_cast<SymbolType>(symbol));
        nextStates.insert(std::begin(transition), std::end(transition));
        for(const auto& nextState: transition)
        {
          const StateSetType nextStateClosure = closure.getClosure(nextState);
          nextStates.insert(std::begin(nextStateClosure), std::end(nextStateClosure));
        }
      }
      catch(std::invalid_argument& e)
      {
        return false;
      }
    }
    currentStates = nextStates;
  }

  return finalStateReached(currentStates, finalStates);
}

bool NFARunner::finalStateReached(const StateSetType& currentStates, const StateSetType& finalStates)
    const
{
  for(const auto& finalState: finalStates)
    if(currentStates.find(finalState) != std::end(currentStates))
      return true;
  return false;
}

} /* namespace Automata */
```

"source/NFA.cpp"

```
#include "gtest/gtest.h"
#include "NFA.h"

using namespace Automata;

TEST(EpsilonClosure, kleeneLike)
{
  NFABuilder<int> builder;

  builder.addTransition(0, Epsilon, 1);
  builder.addTransition(1, "a", 2);
  builder.addTransition(2, Epsilon, 3);
```

```
13    builder.addTransition(0, Epsilon, 3);
      builder.setInitialStateLabel(0);
15    builder.addFinalStateLabel(3);

17    const NFA nfa = builder.build();
      EpsilonClosure closure(nfa);
19
      ASSERT_EQ(StateSetType({0, 1, 3}), closure.getClosure(nfa.getInitialState()));
21    ASSERT_EQ(StateSetType({1}), closure.getClosure(1));
      ASSERT_EQ(StateSetType({2, 3}), closure.getClosure(2));
23    ASSERT_EQ(StateSetType({3}), closure.getClosure(3));
}
25
TEST(EpsilonClosure, withCycles)
27{
      NFABuilder<int> builder;
29
      builder.addTransition(0, Epsilon, 0);
31    builder.addTransition(0, Epsilon, 1);
      builder.addTransition(1, Epsilon, 2);
33    builder.addTransition(0, Epsilon, 2);
      builder.addTransition(2, Epsilon, 0);
35    builder.addTransition(0, "a", 3);
      builder.setInitialStateLabel(0);
37    builder.addFinalStateLabel(3);

39    const NFA nfa = builder.build();
      EpsilonClosure closure(nfa);
41
      ASSERT_EQ(StateSetType({0, 1, 2}), closure.getClosure(0));
43    ASSERT_EQ(StateSetType({0, 1, 2}), closure.getClosure(1));
      ASSERT_EQ(StateSetType({0, 1, 2}), closure.getClosure(2));
45    ASSERT_EQ(StateSetType({3}), closure.getClosure(3));
}
47
TEST(EpsilonClosure, multipleSources)
49{
      NFABuilder<int> builder;
51
      builder.addTransition(0, Epsilon, 0);
53    builder.addTransition(0, Epsilon, 1);
      builder.addTransition(1, Epsilon, 2);
55    builder.addTransition(0, Epsilon, 2);
      builder.addTransition(2, Epsilon, 0);
57    builder.addTransition(0, "a", 3);
      builder.setInitialStateLabel(0);
59    builder.addFinalStateLabel(3);

61    const NFA nfa = builder.build();
      EpsilonClosure closure(nfa);
63
      ASSERT_EQ(StateSetType({0, 1, 2}), closure.getClosure(0));
65    ASSERT_EQ(StateSetType({0, 1, 2}), closure.getClosure(1));
      ASSERT_EQ(StateSetType({0, 1, 2}), closure.getClosure(2));
67    ASSERT_EQ(StateSetType({3}), closure.getClosure(3));

69    ASSERT_EQ(closure.getClosure({0, 1}), StateSetType({0, 1, 2}));
      ASSERT_EQ(closure.getClosure({0, 3}), StateSetType({0, 1, 2, 3}));
71    ASSERT_EQ(closure.getClosure({0, 1, 2}), StateSetType({0, 1, 2}));
      ASSERT_EQ(closure.getClosure({3}), StateSetType({3}));
73}

75TEST(NFARunner, simple1)
{
77    // a*
      NFABuilder<int> builder;
79    builder.addTransition(0, Epsilon, 1);
      builder.addTransition(1, "a", 2);
81    builder.addTransition(2, Epsilon, 3);
      builder.addTransition(0, Epsilon, 3);
83    builder.addTransition(2, Epsilon, 1);
      builder.setInitialStateLabel(0);
85    builder.addFinalStateLabel(3);

87    NFARunner runner(builder.build());
```

```
  const std::set<std::string> correctInputs({"", "a", "aa", "aaa"});
  for(const auto& input: correctInputs)
    ASSERT_TRUE(runner.run(input));

  const std::set<std::string> incorrectInputs({"b", "c", "bc"});
  for(const auto& input: incorrectInputs)
    ASSERT_FALSE(runner.run(input));
}

TEST(NFARunner, simple2)
{
  // a?
  NFABuilder<int> builder;
  builder.addTransition(0, Epsilon, 1);
  builder.addTransition(1, "a", 2);
  builder.addTransition(2, Epsilon, 3);
  builder.addTransition(0, Epsilon, 3);
  builder.setInitialStateLabel(0);
  builder.addFinalStateLabel(3);

  NFARunner runner(builder.build());

  const std::set<std::string> correctInputs({"", "a"});
  for(const auto& input: correctInputs)
    ASSERT_TRUE(runner.run(input));

  const std::set<std::string> incorrectInputs({"aa", "aaa", "b", "c", "bc"});
  for(const auto& input: incorrectInputs)
    ASSERT_FALSE(runner.run(input));
}
```

"source/NFA_test.cpp"

3.4 Clase `DFA`

Clase análoga a NFA, sin embargo las firmas de los métodos son especificos (por ejemplo, `.move` retorna `StateType`, no `StateSetType`). Tambien dispone de algunas clases análogas a NFA, como DFARunner y DFABuilder.

```
#ifndef DFA_H_
#define DFA_H_

#include <algorithm>
#include <iostream>

#include "Common.h"
#include "TransitionTable.h"
#include "NFA.h"

namespace Automata {

class DFARunner;

class DFA
{
protected:
  const NFA _nfa;

public:
  DFA(const DFA&);
  DFA(const NFA&);
  StateType getInitialState() const;
  StateSetType getFinalStates() const;
  StateType move(const StateType&, const SymbolType&) const;
  TransitionTable::TransitionIteratorTag getTransitions(const StateType& state) const;
  // Friend classes
  friend class DFARunner;
  // Friend methods
  friend std::ostream& operator<<(std::ostream&, const DFA&);
  friend TransitionTable::StateIterator begin(const DFA&);
  friend TransitionTable::StateIterator end(const DFA&);
  friend TransitionTable::TransitionIterator begin(const TransitionTable::TransitionIteratorTag&);
  friend TransitionTable::TransitionIterator end(const TransitionTable::TransitionIteratorTag&);
};

template <class LabelType>
```

```
38 class DFABuilder
40 {
   protected:
     using TransitionType = std::tuple<LabelType, SymbolType, LabelType>;
42
     std::vector<TransitionType> _transitions;
44   NFABuilder<LabelType> _nfaBuilder;

46 public:
     DFABuilder() = default;
48   DFABuilder(const DFABuilder&) = delete;
     DFABuilder& operator=(const DFABuilder&) = delete;
50   void setInitialStateLabel(const LabelType& initialStateLabel){ _nfaBuilder.setInitialStateLabel(
       initialStateLabel); }
     void addFinalStateLabel(const LabelType& finalStateLabel){ _nfaBuilder.addFinalStateLabel(
       finalStateLabel); }
52   void addTransition(const LabelType& startLabel, const SymbolType& symbol, const LabelType& finalLabel)
     {
54     if(symbol == Epsilon)
         throw std::invalid_argument("Epsilon can not be used as a transition symbol.");
56     for(const auto& t: _transitions)
         if(std::get<0>(t) == startLabel && std::get<1>(t) == symbol && std::get<2>(t) != finalLabel)
58         throw std::invalid_argument("Same source and same symbol can not go to different targets.");

60     const auto transition = std::make_tuple(startLabel, symbol, finalLabel);
       if(std::find(std::begin(_transitions), std::end(_transitions), transition) == std::end(_transitions))
62       _transitions.push_back(transition);
     }
64   DFA build()
     {
66     for(const auto& t: _transitions)
         _nfaBuilder.addTransition(std::get<0>(t), std::get<1>(t), std::get<2>(t));
68     return DFA(_nfaBuilder.build());
     }
70 };

72 class DFARunner
   {
74 protected:
     NFARunner _runner;
76
   public:
78   DFARunner(const DFA&);
     DFARunner(const DFARunner&) = delete;
80   DFARunner& operator=(const DFARunner&) = delete;
     bool run(const std::string&);
82 };

84 } /* namespace Automata */

86 #endif /* DFA_H_ */
```

"source/DFA.h"

```
#include "DFA.h"
2
namespace Automata {
4
DFA::DFA(const DFA& dfa)
6 :_nfa(dfa._nfa)
{
8 }

10 DFA::DFA(const NFA& nfa)
   :_nfa(nfa)
12 {
   }
14
   StateType DFA::getInitialState() const
16 {
     return _nfa.getInitialState();
18 }

20 StateSetType DFA::getFinalStates() const
   {
22   return _nfa.getFinalStates();
```

```
}

StateType DFA::move(const StateType& from, const SymbolType& symbol) const
{
  auto targets = _nfa.move(from, symbol);
  if(targets.empty())
    throw std::invalid_argument("No transition from " + std::to_string(from) + " with symbol " + symbol);
  return *_nfa.move(from, symbol).begin();
}

TransitionTable::TransitionIteratorTag DFA::getTransitions(const StateType& state) const
{
  return _nfa.getTransitions(state);
}

DFARunner::DFARunner(const DFA& dfa)
:_runner(dfa._nfa)
{
}

std::ostream& operator<<(std::ostream& os, const DFA& dfa)
{
  os << dfa._nfa;
  return os;
}

TransitionTable::StateIterator begin(const DFA& dfa)
{
  return begin(dfa._nfa);
}

TransitionTable::StateIterator end(const DFA& dfa)
{
  return end(dfa._nfa);
}

bool DFARunner::run(const std::string& input)
{
  return _runner.run(input);
}



} /* namespace Automata */
```

"source/DFA.cpp"

```
#include "gtest/gtest.h"
#include "DFA.h"

using namespace Automata;

TEST(DFABuilder, simple1)
{
  DFABuilder<int> builder;
  builder.addTransition(0, "a", 1);
  builder.addTransition(1, "b", 2);
  builder.addTransition(0, "b", 3);
  builder.addTransition(1, "a", 3);
  builder.addTransition(3, "a", 3);
  builder.addTransition(3, "b", 3);
  builder.addTransition(2, "b", 3);
  builder.addTransition(2, "a", 1);

  builder.setInitialStateLabel(0);
  builder.addFinalStateLabel(2);

  const DFA dfa = builder.build(); // (a.b)+

  DFARunner runner(dfa);

  ASSERT_TRUE(runner.run("ab"));
  ASSERT_TRUE(runner.run("abab"));
  ASSERT_TRUE(runner.run("ababab"));
  ASSERT_FALSE(runner.run("aba"));
  ASSERT_FALSE(runner.run("abaa"));
```

```
30    ASSERT_FALSE(runner.run("a"));
      ASSERT_FALSE(runner.run("bbb"));
32    ASSERT_FALSE(runner.run("baba"));
      ASSERT_FALSE(runner.run(""));
34  }

36  TEST(DFARunner, simple2)
    {
38    DFABuilder<int> builder;
      builder.addTransition(0, "a", 1);
40    builder.addTransition(1, "b", 2);
      builder.addTransition(2, "b", 3);
42    builder.addTransition(0, "b", 0);
      builder.addTransition(1, "a", 1);
44    builder.addTransition(2, "a", 1);
      builder.addTransition(3, "a", 1);
46
      builder.setInitialStateLabel(0);
48    builder.addFinalStateLabel(3);

50    const DFA dfa = builder.build(); //(a|b)*.a.b.b

52    DFARunner runner(dfa);

54    ASSERT_TRUE(runner.run("abb"));
      ASSERT_TRUE(runner.run("aabb"));
56    ASSERT_TRUE(runner.run("ababb"));
      ASSERT_TRUE(runner.run("bbabb"));
58    ASSERT_TRUE(runner.run("abbabb"));
      ASSERT_FALSE(runner.run("abba"));
60    ASSERT_FALSE(runner.run("abbb"));
      ASSERT_FALSE(runner.run("ba"));
62    ASSERT_FALSE(runner.run("a"));
      ASSERT_FALSE(runner.run("b"));
64    ASSERT_FALSE(runner.run("bb"));
      ASSERT_FALSE(runner.run("ab"));
66
    }
```

"source/DFA_test.cpp"

3.5 Clase `Thompson` y ayudantes

Su responsabilidad es implementar la Construccion de Thompson a partir de una expresion postfija produciendo un NFA. Además se dispone de las clases ayudantes análogas a cada una de las construcciones de Thompson.

```
1   #ifndef THOMPSON_H_
    #define THOMPSON_H_
3
    #include <string>
5   #include <stack>

7   #include "../Common.h"
    #include "../NFA.h"
9
    namespace Automata {
11
    class Trivial
13  {
    public:
15    Trivial() = delete;
      static Automata::NFA apply(const SymbolType&);
17  };

19  class Concatenation
    {
21  public:
      Concatenation() = delete;
23    static Automata::NFA apply(const NFA&, const NFA&);
    };
25
    class Alternative
27  {
    public:
29    Alternative() = delete;
```

```
  static Automata::NFA apply(const NFA&, const NFA&);
};

class Kleene
{
public:
  Kleene() = delete;
  static Automata::NFA apply(const NFA&);
};

void copyTransitions(NFABuilder<std::string>&, const NFA&, const std::string& = "");

class Thompson {
public:
  Thompson() = delete;
  static Automata::NFA apply(const std::string&);
  static bool isConcatenationOperator(const char& c){return c == '.';}
  static bool isAlternativeOperator(const char& c){return c == '|';}
  static bool isKleeneOperator(const char& c){return c == '*';}
};

} /* namespace Automata */

#endif /* THOMPSON_H_ */
```

"source/Thompson.h"

```
#include "Thompson.h"

namespace Automata {

Automata::NFA Trivial::apply(const SymbolType& symbol)
{
  Automata::NFABuilder<int> builder;

  builder.addTransition(0, symbol, 1);
  builder.setInitialStateLabel(0);
  builder.addFinalStateLabel(1);

  return builder.build();
}

Automata::NFA Concatenation::apply(const NFA& first, const NFA& second)
{
  const std::string firstPrefix("_first");
  const std::string secondPrefix("_second");

  const StateType firstStartState = first.getInitialState();
  const StateType firstEndState = *std::begin(first.getFinalStates());
  const StateType secondStartState = second.getInitialState();
  const StateType secondEndState = *std::begin(second.getFinalStates());

  Automata::NFABuilder<std::string> builder;

  copyTransitions(builder, first, firstPrefix);

  builder.addTransition(firstPrefix+std::to_string(firstEndState), Epsilon, secondPrefix+std::to_string(
    secondStartState));

  copyTransitions(builder, second, secondPrefix);

  builder.setInitialStateLabel(firstPrefix + std::to_string(firstStartState));
  builder.addFinalStateLabel(secondPrefix + std::to_string(secondEndState));

  return builder.build();
}

Automata::NFA Alternative::apply(const NFA& first, const NFA& second)
{
  const std::string firstPrefix("_first");
  const std::string secondPrefix("_second");

  const std::string startStateLabel = "_begining";
  const std::string finalStateLabel = "_final";
  const StateType firstStartState = first.getInitialState();
  const StateType firstEndState = *std::begin(first.getFinalStates());
```

```
49    const StateType secondStartState = second.getInitialState();
      const StateType secondEndState = *std::begin(second.getFinalStates());
51
      Automata::NFABuilder<std::string> builder;
53
      builder.addTransition(startStateLabel, Epsilon, firstPrefix + std::to_string(firstStartState));
55    copyTransitions(builder, first, firstPrefix);
      builder.addTransition(firstPrefix + std::to_string(firstEndState), Epsilon, finalStateLabel);
57
      builder.addTransition(startStateLabel, Epsilon, secondPrefix + std::to_string(secondStartState));
59    copyTransitions(builder, second, secondPrefix);
      builder.addTransition(secondPrefix + std::to_string(secondEndState), Epsilon, finalStateLabel);
61
      builder.setInitialStateLabel(startStateLabel);
63    builder.addFinalStateLabel(finalStateLabel);

65    return builder.build();
  }
67
  Automata::NFA Kleene::apply(const Automata::NFA& nfa)
69{
      const std::string startStateLabel = "_begining";
71    const std::string finalStateLabel = "_final";
      const StateType nfaStartState = nfa.getInitialState();
73    const StateType nfaEndState = *std::begin(nfa.getFinalStates());

75    NFABuilder<std::string> builder;

77    copyTransitions(builder, nfa);

79    builder.addTransition(startStateLabel, Epsilon, std::to_string(nfaStartState));
      builder.addTransition(startStateLabel, Epsilon, finalStateLabel);
81    builder.addTransition(std::to_string(nfaEndState), Epsilon, finalStateLabel);
      builder.addTransition(std::to_string(nfaEndState), Epsilon, std::to_string(nfaStartState));
83
      builder.setInitialStateLabel(startStateLabel);
85    builder.addFinalStateLabel(finalStateLabel);

87    return builder.build();
  }
89
  void copyTransitions(Automata::NFABuilder<std::string>& builder, const NFA& nfa, const std::string& prefix
      )
91{
      for(const auto& stateId: nfa)
93    {
        const std::string startStateLabel = prefix + std::to_string(stateId);
95      for(const auto& p: nfa.getTransitions(stateId))
        {
97        const auto symbol = p.first;
          const auto transition = p.second;
99        for(const auto& endStateId: transition)
          {
101         const std::string endStateLabel = prefix + std::to_string(endStateId);
            builder.addTransition(startStateLabel, symbol, endStateLabel);
103       }
        }
105   }
  }
107
  Automata::NFA Thompson::apply(const std::string& postfix)
109{
      std::stack<NFA> output;
111
      for(const auto& c: postfix)
113   {
        if(isConcatenationOperator(c))
115     {
          const NFA b = output.top(); output.pop();
117       const NFA a = output.top(); output.pop();
          output.push(Concatenation::apply(a, b));
119     }
        else if(isAlternativeOperator(c))
121     {
          const NFA b = output.top(); output.pop();
123       const NFA a = output.top(); output.pop();
```

```
      output.push(Alternative::apply(a, b));
    }
    else if(isKleeneOperator(c))
    {
      const NFA a = output.top(); output.pop();
      output.push(Kleene::apply(a));
    }
    else
    {
      output.push(Trivial::apply(std::string(1, c)));
    }
  }

  return output.top();
}

} /* namespace Automata */
```

"source/Thompson.cpp"

```
#include "../gtest/gtest.h"
#include "Thompson.h"

using namespace Automata;

TEST(TrivialConstruction, simple1)
{
  const SymbolType symbol("a");

  const NFA nfa = Trivial::apply(symbol);

  NFARunner runner(nfa);

  ASSERT_TRUE(runner.run(symbol));
  ASSERT_FALSE(runner.run(""));
  ASSERT_FALSE(runner.run("aa"));
  ASSERT_FALSE(runner.run("ab"));
  ASSERT_FALSE(runner.run("b"));
}

TEST(TrivialConstruction, Epsilon)
{
  const NFA nfa = Trivial::apply(Epsilon);

  NFARunner runner(nfa);

  ASSERT_TRUE(runner.run(""));
  ASSERT_FALSE(runner.run("a"));
  ASSERT_FALSE(runner.run("aa"));
  ASSERT_FALSE(runner.run("ab"));
  ASSERT_FALSE(runner.run("b"));
}

TEST(ConcatenationConstrction, simple1)
{
  const SymbolType symbol1("a");
  const SymbolType symbol2("b");

  const NFA nfa = Concatenation::apply(Trivial::apply(symbol1), Trivial::apply(symbol2));
  NFARunner runner(nfa);

  ASSERT_TRUE(runner.run(symbol1 + symbol2));
  ASSERT_FALSE(runner.run(symbol1));
  ASSERT_FALSE(runner.run(symbol2));
  ASSERT_FALSE(runner.run(symbol2 + symbol1));
  ASSERT_FALSE(runner.run("ccab"));
  ASSERT_FALSE(runner.run("abcc"));
  ASSERT_FALSE(runner.run(symbol1+symbol2+symbol1+symbol2));
}

TEST(ConcatenationConstruction, Epsilon1)
{
  const SymbolType symbol1("a");

  const NFA nfa = Concatenation::apply(Trivial::apply(Epsilon), Trivial::apply(symbol1));
  NFARunner runner(nfa);
```

```
58    ASSERT_TRUE(runner.run(symbol1));
      ASSERT_FALSE(runner.run(""));
60    ASSERT_FALSE(runner.run("aa"));
  }
62
  TEST(ConcatenationConstruction, Epsilon2)
64  {
      const SymbolType symbol1("a");
66
      const NFA nfa = Concatenation::apply(Trivial::apply(symbol1), Trivial::apply(Epsilon));
68    NFARunner runner(nfa);

70    ASSERT_TRUE(runner.run(symbol1));
      ASSERT_FALSE(runner.run(""));
72    ASSERT_FALSE(runner.run("aa"));
  }
74
  TEST(AlternativeConstruction, simple1)
76  {
      const SymbolType symbol1("a");
78    const SymbolType symbol2("b");

80    const NFA nfa = Alternative::apply(Trivial::apply(symbol1), Trivial::apply(symbol2));
      NFARunner runner(nfa);
82
      ASSERT_TRUE(runner.run(symbol1));
84    ASSERT_TRUE(runner.run(symbol2));
      ASSERT_FALSE(runner.run(symbol2 + symbol1));
86    ASSERT_FALSE(runner.run("ccab"));
      ASSERT_FALSE(runner.run("abcc"));
88    ASSERT_FALSE(runner.run(symbol1+symbol2+symbol1+symbol2));
  }
90
  TEST(AlternativeConstruction, epsilon)
92  {
      const SymbolType symbol1("a");
94
      const NFA nfa = Alternative::apply(Trivial::apply(symbol1), Trivial::apply(Epsilon));
96    NFARunner runner(nfa);

98    ASSERT_TRUE(runner.run(symbol1));
      ASSERT_TRUE(runner.run(""));
100   ASSERT_FALSE(runner.run("ccab"));
      ASSERT_FALSE(runner.run("abcc"));
102   ASSERT_FALSE(runner.run("aa"));
  }
104
  TEST(KleeneConstruction, simple1)
106 {
      const SymbolType symbol("a");
108
      const NFA nfa = Kleene::apply(Trivial::apply(symbol));
110   NFARunner runner(nfa);

112   ASSERT_TRUE(runner.run(""));
      ASSERT_TRUE(runner.run(symbol));
114   ASSERT_TRUE(runner.run(symbol+symbol));
      ASSERT_TRUE(runner.run(symbol+symbol+symbol));
116   ASSERT_FALSE(runner.run(symbol+"b"+symbol));
      ASSERT_FALSE(runner.run(symbol+symbol+"b"));
118 }

120 TEST(KleeneConstruction, epsilon)
  {
122   const NFA nfa = Kleene::apply(Trivial::apply(Epsilon));
      NFARunner runner(nfa);
124
      ASSERT_TRUE(runner.run(""));
126   ASSERT_FALSE(runner.run("a"));
      ASSERT_FALSE(runner.run("aa"));
128 }

130 TEST(ThompsonConstruction, simple1)
  {
132   const std::string postfixExpression = "ab.";
```

```
  const NFA nfa = Thompson::apply(postfixExpression);
  NFARunner runner(nfa);

  ASSERT_TRUE(runner.run("ab"));
  ASSERT_FALSE(runner.run("a"));
  ASSERT_FALSE(runner.run("aa"));
  ASSERT_FALSE(runner.run("aba"));
}
```

"source/Thompson_test.cpp"

3.6 Clase `Powerset`

Su responsabilidad es implementar el Algoritmo de Subconjuntos para producir un DFA a partir de un NFA.

```
#ifndef POWERSET_H_
#define POWERSET_H_

#include <map>
#include <queue>
#include "../NFA.h"
#include "../DFA.h"

namespace Automata {

class Powerset
{
public:
  Powerset() = delete;
  static DFA apply(const NFA&);
private:
  static StateSetType moveOverSet(const NFA&, const StateSetType&, const SymbolType&);
  static AlphabetType getAlphabet(const NFA&);
  static StateSetType multipleMove(const NFA&, const StateSetType&, const SymbolType&);
  static bool containsAFinalState(const StateSetType&, const StateSetType&);
};

} /* namespace Automata */

#endif /* POWERSET_H_ */
```

"source/Powerset.h"

```
#include "Powerset.h"

namespace Automata {

DFA Powerset::apply(const NFA& nfa)
{
  using DFAStatesSet = std::set<StateSetType>;
  using DFAStatesQueue = std::queue<StateSetType>;

  const AlphabetType alphabet = getAlphabet(nfa);

  DFABuilder<StateSetType> builder;
  EpsilonClosure closure(nfa);

  const StateSetType sourceClosure = closure.getClosure(nfa.getInitialState());

  DFAStatesSet dfaStates;
  dfaStates.insert(sourceClosure);

  DFAStatesQueue dfaStatesToProcess;
  dfaStatesToProcess.push(sourceClosure);
  while(!dfaStatesToProcess.empty())
  {
    const auto dfaState = dfaStatesToProcess.front();
    dfaStatesToProcess.pop();

    StateSetType sources;
    for(const auto& symbol: alphabet)
    {
      const auto nextDFAState = closure.getClosure(multipleMove(nfa, dfaState, symbol));
      if(dfaStates.find(nextDFAState) == std::end(dfaStates))
      {
```

```
33          dfaStates.insert(nextDFAState);
            dfaStatesToProcess.push(nextDFAState);
35        }
          builder.addTransition(dfaState, symbol, nextDFAState);
37      }
    }
39
    builder.setInitialStateLabel(sourceClosure);
41
    for(const auto& dfaState: dfaStates)
43    if(containsAFinalState(dfaState, nfa.getFinalStates()))
        builder.addFinalStateLabel(dfaState);
45
    return builder.build();
47
}
49
StateSetType Powerset::moveOverSet(const NFA& nfa, const StateSetType& sources, const SymbolType& symbol)
51{
    StateSetType targets;
53  for(const auto& source: sources)
    {
55    const StateSetType singleSourceTransitions= nfa.move(source, symbol);
      for(const auto& transition: singleSourceTransitions)
57      targets.insert(transition);
    }
59
    return targets;
61}
63AlphabetType Powerset::getAlphabet(const NFA& nfa)
{
65  AlphabetType alphabet;

67  for(const auto& state: nfa)
      for(const auto& p: nfa.getTransitions(state))
69    {
        const SymbolType symbol = p.first;
71      if(symbol != Epsilon)
          alphabet.insert(symbol);
73    }

75  return alphabet;
}
77
StateSetType Powerset::multipleMove(const NFA& nfa, const StateSetType& sources, const SymbolType& symbol)
79{
    StateSetType targets;
81  for(const auto& state: sources)
    {
83    const auto singleSourceTargets = nfa.move(state, symbol);
      for(const auto& state: singleSourceTargets)
85      targets.insert(state);
    }
87  return targets;
}
89
bool Powerset::containsAFinalState(const StateSetType& states, const StateSetType& finalStates)
91{
    for(const auto& finalState: finalStates)
93    if(states.find(finalState) != std::end(states))
        return true;
95  return false;
}
97
} /* namespace Automata */
```

"source/Powerset.cpp"

```
#include "../gtest/gtest.h"
2#include "Powerset.h"

4using namespace Automata;

6TEST(Powerset, allABstringWithABBsufix)
{
```

```
  NFABuilder<int> builder;
  builder.addTransition(0, Epsilon, 1);
  builder.addTransition(0, Epsilon, 7);
  builder.addTransition(1, Epsilon, 2);
  builder.addTransition(1, Epsilon, 4);
  builder.addTransition(2, "a", 3);
  builder.addTransition(3, Epsilon, 6);
  builder.addTransition(4, "b", 5);
  builder.addTransition(5, Epsilon, 6);
  builder.addTransition(6, Epsilon, 1);
  builder.addTransition(6, Epsilon, 7);
  builder.addTransition(7, "a", 8);
  builder.addTransition(8, "b", 9);
  builder.addTransition(9, "b", 10);
  builder.setInitialStateLabel(0);
  builder.addFinalStateLabel(10);
  const NFA nfa = builder.build();

  NFARunner nfaRunner(nfa);
  ASSERT_TRUE(nfaRunner.run("abb"));
  ASSERT_TRUE(nfaRunner.run("aabb"));
  ASSERT_TRUE(nfaRunner.run("babb"));
  ASSERT_TRUE(nfaRunner.run("ababb"));
  ASSERT_TRUE(nfaRunner.run("aaababababb"));
  ASSERT_FALSE(nfaRunner.run(""));
  ASSERT_FALSE(nfaRunner.run("a"));
  ASSERT_FALSE(nfaRunner.run("bb"));
  ASSERT_FALSE(nfaRunner.run("abab"));

  const DFA dfa = Powerset::apply(nfa);

  DFARunner dfaRunner(dfa);
  ASSERT_TRUE(dfaRunner.run("abb"));
  ASSERT_TRUE(dfaRunner.run("aabb"));
  ASSERT_TRUE(dfaRunner.run("babb"));
  ASSERT_TRUE(dfaRunner.run("ababb"));
  ASSERT_TRUE(dfaRunner.run("aaababababb"));
  ASSERT_FALSE(dfaRunner.run(""));
  ASSERT_FALSE(dfaRunner.run("a"));
  ASSERT_FALSE(dfaRunner.run("bb"));
  ASSERT_FALSE(dfaRunner.run("abab"));
}
```

"source/Powerset_test.cpp"

3.7 Clase `Hopcroft`

Su responsabilidad es implementar el Algoritmo de Hopcroft para producir un DFA Mínimo a partir de un DFA.

```
#ifndef HOPCROFT_HOPCROFT_H_
#define HOPCROFT_HOPCROFT_H_

#include "../Common.h"
#include "../DFA.h"

namespace Automata {

class Hopcroft {
private:
  using IdType = int;
  struct Group
  {
    StateSetType states;
    IdType id;
    bool operator<(const Group& rhs) const { return id < rhs.id; }
    bool operator==(const Group& rhs) const { return states == rhs.states; }
    bool operator!=(const Group& rhs) const { return !operator==(rhs); }
  };
  using PartitionType = std::set<Group>;

public:
  Hopcroft() = delete;
  static DFA apply(const DFA&);

private:
```

```
  static PartitionType initialPartition(const DFA&);
  static PartitionType improvePartition(const DFA&, const PartitionType&, const AlphabetType&);
  static AlphabetType getAlphabet(const DFA&);
  static IdType findGroupId(const PartitionType&, const StateType&);
  static void printPartition(const PartitionType&);
  static void printGroup(const Group&);

};

} /* namespace Automata */

#endif /* HOPCROFT_HOPCROFT_H_ */
```

"source/Hopcroft.h"

```
#include "Hopcroft.h"

namespace Automata {

DFA Hopcroft::apply(const DFA& dfa)
{
  const auto alphabet = getAlphabet(dfa);
  PartitionType partition = initialPartition(dfa);

  PartitionType newPartition = improvePartition(dfa, partition, alphabet);
  while(newPartition != partition)
  {
    partition = newPartition;
    newPartition = improvePartition(dfa, partition, alphabet);
  }

  DFABuilder<IdType> builder;
  // Add transitions
  for(const auto& group: partition)
  {
    const auto& representativeState = *group.states.begin();
    const auto& sourceLabel = group.id;
    for(const auto& symbol: alphabet)
    {
      const auto& targetState = dfa.move(representativeState, symbol);
      const auto& targetLabel = findGroupId(partition, targetState);

      builder.addTransition(sourceLabel, symbol, targetLabel);
    }
  }
  // Find group with source state and set it to start state into the min DFA
  for(const auto& group: partition)
    if(group.states.find(dfa.getInitialState()) != std::end(group.states))
    {
      builder.setInitialStateLabel(group.id);
      break;
    }
  // Find all groups with at least one final state and add them into the min DFA
  const auto finalStates = dfa.getFinalStates();
  for(const auto& group: partition)
    for(const auto& finalState: finalStates)
      if(group.states.find(finalState) != std::end(group.states))
        builder.addFinalStateLabel(group.id);

  return builder.build();
}

Hopcroft::PartitionType Hopcroft::initialPartition(const DFA& dfa)
{
  StateSetType acceptedStates = dfa.getFinalStates();
  StateSetType nonAcceptedStates;
  for(const auto& state: dfa)
    if(acceptedStates.find(state) == std::end(acceptedStates))
      nonAcceptedStates.insert(state);

  PartitionType partition;
  partition.insert({acceptedStates, 0});
  partition.insert({nonAcceptedStates, 1});

  return partition;
}
```

```
62
Hopcroft::PartitionType Hopcroft::improvePartition(const DFA& dfa, const PartitionType& partition, const
    AlphabetType& alphabet)
64 {
  IdType groupIdCounter = 0;
66   PartitionType newPartition;

68   for(const auto& group: partition)
  {
70     bool copyOriginalGroupToNewPartition = true;
    for(const auto& symbol: alphabet)
72     {
      std::map<IdType, StateSetType> newGroups;
74
      for(const auto& state: group.states)
76       {
        const auto nextState = dfa.move(state, symbol);
78         const auto groupId = findGroupId(partition, nextState);
        newGroups[groupId].insert(state);
80       }

82       if(newGroups.size() > 1)
      {
84         for(const auto& p: newGroups)
        {
86           const auto& states = p.second;
          const auto& groupId = groupIdCounter++;
88           newPartition.insert({states, groupId});
        }
90         copyOriginalGroupToNewPartition = false;
        break;
92       }
    }
94     if(copyOriginalGroupToNewPartition)
      newPartition.insert({group.states, groupIdCounter++});
96   }

98   return newPartition;
}
100
AlphabetType Hopcroft::getAlphabet(const DFA& dfa)
102 {
  AlphabetType alphabet;
104   for(const auto& state: dfa)
  {
106     for(const auto& transition: dfa.getTransitions(state))
    {
108       const auto symbol = transition.first;
      if(symbol != Epsilon)
110         alphabet.insert(symbol);
    }
112   }
  return alphabet;
114 }

116 Hopcroft::IdType Hopcroft::findGroupId(const PartitionType& partition, const StateType& state)
{
118   for(const auto& group: partition)
    for(const auto& groupState: group.states)
120       if(state == groupState)
        return group.id;
122   throw std::invalid_argument("State is not contained in this partition.");
}
124
} /* namespace Automata */
```

"source/Hopcroft.cpp"

```
#include "../gtest/gtest.h"
#include "Hopcroft.h"
#include "../DFA.h"

using namespace Automata;

TEST(Hopcroft, minDFAasInput)
{
```

```
  DFABuilder<int> builder;
  builder.addTransition(0, "a", 1);
  builder.addTransition(0, "b", 0);
  builder.addTransition(1, "b", 2);
  builder.addTransition(1, "a", 1);
  builder.addTransition(2, "a", 1);
  builder.addTransition(2, "b", 3);
  builder.addTransition(3, "a", 1);
  builder.addTransition(3, "b", 0);
  builder.setInitialStateLabel(0);
  builder.addFinalStateLabel(3);
  const DFA dfa = builder.build();

  DFARunner dfaRunner(dfa);
  ASSERT_TRUE(dfaRunner.run("abb"));
  ASSERT_TRUE(dfaRunner.run("abbabb"));
  ASSERT_TRUE(dfaRunner.run("ababb"));
  ASSERT_TRUE(dfaRunner.run("babb"));
  ASSERT_TRUE(dfaRunner.run("ababbabb"));
  ASSERT_FALSE(dfaRunner.run("ab"));
  ASSERT_FALSE(dfaRunner.run("bab"));
  ASSERT_FALSE(dfaRunner.run("b"));
  ASSERT_FALSE(dfaRunner.run("a"));
  ASSERT_FALSE(dfaRunner.run("abbaaba"));

  const DFA minDfa = Hopcroft::apply(dfa);

  DFARunner minDfaRunner(dfa);
  ASSERT_TRUE(minDfaRunner.run("abb"));
  ASSERT_TRUE(minDfaRunner.run("abbabb"));
  ASSERT_TRUE(minDfaRunner.run("ababb"));
  ASSERT_TRUE(minDfaRunner.run("babb"));
  ASSERT_TRUE(minDfaRunner.run("ababbabb"));
  ASSERT_FALSE(minDfaRunner.run("ab"));
  ASSERT_FALSE(minDfaRunner.run("bab"));
  ASSERT_FALSE(minDfaRunner.run("b"));
  ASSERT_FALSE(minDfaRunner.run("a"));
  ASSERT_FALSE(minDfaRunner.run("abbaaba"));
}

TEST(Hopcroft, allABStringStartingWithB)
{
  DFABuilder<int> builder;
  builder.addTransition(0, "a", 1);
  builder.addTransition(0, "b", 2);
  builder.addTransition(1, "b", 1);
  builder.addTransition(1, "a", 1);
  builder.addTransition(2, "a", 2);
  builder.addTransition(2, "b", 2);
  builder.setInitialStateLabel(0);
  builder.addFinalStateLabel(2);
  const DFA dfa = builder.build();

  DFARunner dfaRunner(dfa);
  ASSERT_TRUE(dfaRunner.run("babb"));
  ASSERT_TRUE(dfaRunner.run("babbabbbaa"));
  ASSERT_TRUE(dfaRunner.run("bababab"));
  ASSERT_TRUE(dfaRunner.run("babb"));
  ASSERT_TRUE(dfaRunner.run("b"));
  ASSERT_FALSE(dfaRunner.run(""));
  ASSERT_FALSE(dfaRunner.run("ababa"));
  ASSERT_FALSE(dfaRunner.run("aab"));
  ASSERT_FALSE(dfaRunner.run("a"));
  ASSERT_FALSE(dfaRunner.run("abbaaba"));

  const DFA minDfa = Hopcroft::apply(dfa);

  DFARunner minDfaRunner(dfa);
  ASSERT_TRUE(dfaRunner.run("babb"));
  ASSERT_TRUE(dfaRunner.run("babbabbbaa"));
  ASSERT_TRUE(dfaRunner.run("bababab"));
  ASSERT_TRUE(dfaRunner.run("babb"));
  ASSERT_TRUE(dfaRunner.run("b"));
  ASSERT_FALSE(dfaRunner.run(""));
  ASSERT_FALSE(dfaRunner.run("ababa"));
  ASSERT_FALSE(dfaRunner.run("aab"));
```

```
85      ASSERT_FALSE(dfaRunner.run("a"));
        ASSERT_FALSE(dfaRunner.run("abbaaba"));
87  }
```

"source/Hopcroft_test.cpp"