CENG 242

Programming Language Concepts

Spring 2023-2024

Programming Exam 4

Due date: May 5 2023, Sunday, 23:59

Introduction

In this assignment, you are going to complete an implementation for **Nondeterministic Finite Automata** (NFA) using C++11 with an emphasis on **operator overloading**.

Brief reminder for the notation and concepts: Finite automata are one of the simplest abstractions for computation. Mathematically, an NFA M is defined as $M = \{K, \Sigma, \Delta, s, F\}$ where K is the set of states of the automata, Σ is the alphabet that contains the input alphabet, Δ is the transition relation which is a subset of $K \times (\Sigma \cup \{e\}) \times K$, $s \in K$ is the starting state, and $F \subseteq K$ is the set of final states. Given a string s, which is an ordered tuple of symbols from the alphabet Σ, an NFA processes s and determines whether s is accepted or rejected by the formal definition that the machine encapsulates.

1 Overview

In this assignment, you are going to work with the following classes:

1. `Alphabet`,
2. `Rule`,
3. `TransitionTable`,
4. `ComputationBranch`,
5. `NFA`.

States are not implemented as a separate class, instead, the implementation holds state information as `strings`. The class `NFA` leverages the other classes to implement a properly functioning NFA. It further implements the unary **complement** operation in case a given NFA is essentially a DFA, in which case $M_2 = M_1^c$ ensures $\mathcal{L}(M_2) = \Sigma^* - \mathcal{L}(M_1)$, and the binary **union** operation such that $M_3 = M_1 \cup M_2$ ensures $\mathcal{L}(M_3) = \mathcal{L}(M_1) \cup \mathcal{L}(M_2)$.

2 Class definitions and expected implementations

2.1 Alphabet

This is an implementation for an alphabet that holds the symbols (implemented as a `char` variable) using the standard library's `set` class. For this class, you will implement

1. a validation function, `is_valid(string)`, that returns `true` if the string is composed only of symbols in the alphabet and returns `false` otherwise.

2. a getter function, `get_symbols()`, that returns the symbols contained in the object.
3. an overloading for `operator+=(Alphabet)` which updates the symbols of the current `Alphabet` object such that it contains the symbols of the passed argument as well. You will use this operator when taking the union of two NFAs.
4. an overloading for `operator<<(ostream, Alphabet)` such that the `ostream` object will contain a string representation for the object. This function returns the reference to the `ostream` object. The desired string representation lists the symbols contained in the alphabet in a single line in a single line with a single space separation.

Examples — **Alphabet**

If an alphabet `sigma1` contains the symbols `a,b` and another alphabet `sigma2` contains the symbols `b,c`

- `sigma1.is_valid('abc');` returns `false`
- `cout << sigma;` should print (the order of printed symbols are not important)

  `a␣b`

  where the little cup symbols indicate blank spaces.
- `sigma1 += sigma2; cout << sigma1;` prints

  `a␣b␣c`

Important: The symbol `'e'` is reserved for indicating the empty transition; thus, is not included in the set of symbols for any of the test cases that your code will be graded on.

2.2 Rule

This class encapsulates a single transition rule for NFA with the variables `string initial_state`, `char symbol`, and `string final_state`. For this class, you will implement

1. a constructor `Rule(string, char, string)`.
2. the function `get_final_state` that returns the `final_state` of the rule.
3. the function `applies_to(string init, char s)` that returns whether the rule applies to the passed initial state and symbols.
4. the function `update_state_name(string old_name, string new_name)` that updates the name of a state. This will be needed when you are implementing the union of two NFAs.
5. an overloading for `operator<<(ostream, Rule)`.

Examples — **Rule**

If the variable `rule` is defined via `Rule rule("q0", 'a', "q1");` then

- `cout << rule.get_final_state();` prints `q1`
- `rule.applies_to("q0", 'a');` returns `true` whereas `rule.applies_to("q1", 'a');` returns `false`
- `rule.update_state_name("q1", "qq1"); cout << rule.get_final_state();` prints `qq1`. Passing an irrelevant state name, such as `rule.update_state_name("q2", "qq1");` will have no effect.
- `cout << rule;` should print (the ordering matters)

  `q0␣a␣q1`

2.3 TransitionTable

This class will hold all of the transition rules for an NFA in the variable `rules` which is a `vector` of `Rule`s. For this class, you will implement

1. the function `add_rule(string initial_state, char symbol, string final_state)` which creates and pushes the corresponding rule to the variable `rules`.
2. the function `update_state_name(string old_name, string new_name)` that calls the same-named function on all of the `Rule`s contained in `rules`.
3. an overloading for `operator()`, which we will use for indexing. The overloaded function takes two inputs, `string initial_state` and `char symbol`, and returns the set of **all** possible next states.
4. an overloading for `operator+=` that will simply append the rules from the passed `TransitionTable` object (`other.rules`) to the current object's rules (`this->rules`),
5. an overloading for `operator<<(ostream, TransitionTable)` which prints all the `Rule`s in the table, **each** followed by a line break (`std::endl`).

Examples — TransitionTable

Let `transitions1` be a `TransitionTable` object that initially contains a single rule `Rule("q0", 'a', "q1")`, and `transitions2` contain the rules `Rule("qq0", 'a', "qq1")` and `Rule("qq0", 'a', "qq2")`.

- `transitions1.add_rule(Rule("q0", 'b', "q0"); cout << transitions1;` should print (the ordering of the lines do not matter)

```
q0␣a␣q1
q0␣b␣q0
```

- if `transitions1.update_state_name("q0", "q3");` is called after the above call, then `cout << transitions1;` prints

```
q3␣a␣q1
q3␣b␣q3
```

- `transitions2("qq0", 'a')` returns a **set** that includes the strings `"qq1"` and `"qq2"` in it.
- if `transitions1 += transitions2;` is called after the above calls, then `cout << transitions1;` prints

```
q3␣a␣q1
q3␣b␣q3
qq0␣a␣qq1
qq0␣a␣qq2
```

2.4 ComputationBranch

Since NFAs are nondeterministic, we are going to keep track of device configurations from different branches via this class. Instead of just storing the current configuration, this class will keep track of the all of the configurations leading to the current state. A configuration for an NFA is just the tuple composed of the NFA's current state and the remaining part of the input string. For this assignment, this tuple is represented as a pair of strings (`pair<string,string>`), and the configuration history (which is the only variable of this class) is of type `vector<pair<string,string>>`. For this class, you will implement

1. the function `push_config(string state, string input)` that appends the corresponding configuration tuple to the configuration history.

2. the getter `get_last_config()` that returns the last configuration from the current computation branch.

3. an overloading for `operator<<(ostream, ComputationBranch)`. In the string representation of the computation branch, starting from the initial configuration, you will list the configurations separated by `␣:-␣`. To indicate the empty string you are **required to** use the letter `e`.

Examples — **ComputationBranch**

Let the variable `branch` be an instance of `ComputationBranch` and currently hold a single configuration, namely, `make_pair<"s", "ab">`.

- `branch.push_config("q0", "b"); cout << branch;` should print (the ordering matters)

  `(s,␣ab)␣:-␣(q0,␣b)`

- `branch.get_last_config()` returns the pair equivalent to `make_pair<"q0", "b">` if it is called after the above `push_config` call.

- if `branch.push_config("q3", ""); cout << branch;` is executed following the above calls, it prints

  `(s,␣ab)␣:-␣(q0,␣b)␣:-␣(q3,␣e)`

  Again, be careful to represent the empty string as `e`.

2.5 NFA

This class unites all of the pieces together to yield a functioning NFA. For this class, you will implement

1. the state-checking function `has_state(string state_name)` which returns `true` if the set `all_states` has a member named as `state_name` and returns `false` otherwise.

2. the function `is_final_state(string state_name)` which returns `true` if `final_states` includes a state named as `state_name` and returns `false` otherwise.

3. the function `is_DFA()` which returns `true` if the current NFA object is a DFA[1], returns `false` otherwise.

4. the function `update_state_name(string old_name, string new_name)` that updates the name of a state. Specifically, this function checks the variables `all_states`, `starting_state`, and `final_states` and replaces all occurrences of `old_name` with `new_name`. After this, it invokes the same-named function on the variable `transitions`.

5. the function `process(string input)` which processes the `input` based on the specifications of the NFA. If it accepts the string it returns `true` and returns `false` otherwise. Since NFAs are nondeterministic, you are expected to keep track of each computation branch, for which you can use a `queue`.

   - if the input is not valid (*i.e.* contains symbols that are not present in the alphabet), the function prints `Invalid string` and returns `false`,
   - if an accepting branch is found, the function prints the corresponding `ComputationBranch` object, and in a separate line prints `Accept`,
   - if the `input` is rejected, then the last `ComputationBranch` object (not unique) that was checked is printed, and in a separate line `Reject` is printed

[1]Recall that if the transition relation Δ is a **function** with the domain $K \times \Sigma$ and the range K, then the NFA is in fact a DFA.

The following is a pseudocode for a possible implementation to help you with the structure of the function.

```
function process(input):
    if input is invalid:
        print("Invalid string\n")
        return False
    Q = init_queue(starting_state, input)
    while Q is not empty:
        branch = Q.dequeue()
        if branch is accepting:
            print(branch)
            print("Accept\n")
            return True
        set_NFA_configuration(branch)
        check_e_transitions_and_push_new_configs()
        check_symbol_transitions_and_push_new_configs()
    print(branch)
    print("Reject\n")
    return False
```

6. an overloading of the unary `operator!(void)` that returns an instance of the class NFA. If the current object is not a DFA, the overloaded function simply prints `Not a DFA` and returns an NFA that is equivalent to the current object. If it is a DFA, it returns the **complement** of the current object such that the returned NFA accepts all the strings (over the same alphabet) rejected by the current object, and rejects all the strings accepted by the current object.

   - Remark that the complement of a DFA can be acquired by simply replacing the set of final states F by $K - F$. That is, for $M = \{K, \Sigma, \Delta, s, F\}$, the complement is defined as $M^c = \{K, \Sigma, \Delta, s, K - F\}$.

7. an overloading of the binary `operator+(NFA)` that returns an instance of the class NFA. The returned NFA instance is a **union** of the current object and the passed argument. That is, the returned object is an NFA that accepts a string if and only if any of the operands accept it, and reject the string otherwise.

   - Remark that the union of two NFAs, $M_1 = \{K_1, \Sigma_1, \Delta_1, s_1, F_1\}, M_2 = \{K_2, \Sigma_2, \Delta_2, s_2, F_2\}$, can be defined as $M = M_1 \cup M_2 = \{K, \Sigma, \Delta, s, F\}$ such that[2]

$$
\begin{aligned}
K &= K_1 \cup K_2 \cup \{s\} \\
\Sigma &= \Sigma_1 \cup \Sigma_2 \\
F &= F_1 \cup F_2 \\
\Delta &= \Delta_1 \cup \Delta_2 \cup \{(s, e, s_1), (s, e, s_2)\}
\end{aligned}
$$

     where we assumed $s \notin K_1$ and $s \notin K_2$.

   - In this implementation, states are represented as strings, and the two machines might have states with the same name. To properly implement the union operation, you are expected to update the name of each state in the **operand** (*i.e.* `other`) by inserting the letter **"q"** at the beginning as much as needed **if** the state name is shared by both machines. Similarly, you are expected to update the name of the newly introduced state by inserting the letter **"s"** at the beginning as much as needed should the names "s", "ss", ... be already in use. These cases are illustrated in the examples below.

Note: The implementation for `operator<<(ostream, NFA)` is provided in the `.hpp` file, however it will need the proper overloading of the `operator<<` for the other classes to function correctly.

[2]We slightly generalize the case in the CENG280's textbook (Lewis & Papadimitriou) by allowing the two alphabets to be different.

Examples NFA

Let $M_1 = \{K_1, \Sigma_1, \Delta_1, s_1, F_1\}$ where $K_1 = \{q_0, q_1\}$, $\Sigma_1 = \{a, b\}$, $s_1 = q_0$, $F_1 = \{q_1\}$, and $\Delta_1 = \{(q_0, a, q_1), (q_0, b, q_0), (q_1, a, q_0), (q_1, b, q_1)\}$, and $M_2 = \{K_2, \Sigma_2, \Delta_2, s_2, F_2\}$ where $K_2 = \{q_0, q_2\}$, $\Sigma_2 = \{b, c\}$, $s_2 = q_0$, $F_2 = \{q_2\}$, and $\Delta_2 = \{(q_0, b, q_2), (q_2, c, q_2)\}$, and `M1` and `M2` be the corresponding NFA instantiations that implement these machines by adhering to the naming of the states. Then,

- `M1.has_state("q0")` returns `true`, while `M1.has_state("q2")` returns `false`
- `M1.is_final_state("q0")` returns `false`, while `M1.is_final_state("q1")` returns `true`
- `M1.is_DFA()` should return `true`, while `M2.is_DFA()` returns `false`
- `M2.update_state_name("q2", "q3"); cout << M2;` prints (the ordering of the alphabet, states, and rules are not essential)

  ```
  b␣c
  q0␣q3
  q0
  q3
  q0␣b␣q3
  q3␣c␣q3
  ```

- after the above calls, `M2.process("cbb"); cout << "--" << endl; M2.process("bcc"); cout << "--" << endl; M2.process("abc");` prints

  ```
  (q0,␣cbb)
  Reject
  --
  (q0,␣bcc)␣:-␣(q3,␣cc)␣:-␣(q3,␣c)␣:-␣(q3,␣e)
  Accept
  --
  Invalid␣string
  ```

- `NFA M3(!M1); cout << M3 << "--" << endl; M3 = !M2; cout << "--" << endl << M3;` if called after the above calls, prints

  ```
  a␣b
  q0␣q1
  q0
  q0
  q0␣a␣q1
  q0␣b␣q0
  q1␣a␣q0
  q1␣b␣q1
  --
  Not␣a␣DFA
  --
  b␣c
  q0␣q3
  q0
  q3
  q0␣b␣q3
  q3␣c␣q3
  ```

  Note that since `M2` is not a DFA, its complement is not calculated. Consequently, after the call `M3 = !M2`, `M3` is equivalent to `M2`.

Examples (continued) NFA

- If `NFA M4(M1+M2); cout << M4; M4 = M4 + M2; cout << "--" << endl << M4;` is called after the above calls, the expected output is

```
a␣b␣c
q0␣q1␣q3␣qq0␣s
s
q1␣q3
qq0␣b␣q3
q3␣c␣q3
q0␣a␣q1
q0␣b␣q0
q1␣a␣q0
q1␣b␣q1
s␣e␣qq0
s␣e␣q0
--
a␣b␣c
q0␣q1␣q3␣qq0␣qq3␣qqq0␣s␣ss
ss
q1␣q3␣qq3
qqq0␣b␣qq3
qq3␣c␣qq3
qq0␣b␣q3
q3␣c␣q3
q0␣a␣q1
q0␣b␣q0
q1␣a␣q0
q1␣b␣q1
s␣e␣qq0
s␣e␣q0
ss␣e␣qqq0
ss␣e␣s
```

Note that in the description of first machine, `q0` of `M2` is renamed to `qq0` while the same-named state of `M1` kept its name. Also observe that `q3` of `M2` retained its name since no state of `M1` shares this name. Similarly, after the second union call, state names of the second argument of the `operator+`, *i.e.* `M2`, is updated while the state names of the first argument, *i.e.* `M4`, are kept as is. Lastly, the name of the newly introduced state is updated to `ss` since a state named `s` is already present in the machine. If we were to call `M4+M2` once again, the new starting state would be named as `sss`.

3 Specifications and Notes

1. **Implementation and Submission:** The template files are available in the Virtual Programming Lab (VPL) activity called "PE4" on odtuclass. You can download them and work on your local device, or directly work on the VPL's editor. The last saved versions of your `nfa.cpp` and `components.cpp` files will be used for final grading. Make sure that your `.cpp` files are compatible with the provided `.hpp` files. Ensure your code compiles using the following command.

   ```
   >> g++ -std=c++11 main.cpp nfa.cpp components.cpp utils.cpp
   ```

2. Do not edit or introduce anything to the `.hpp` files or to `utils.cpp`; your changes to these files will be **discarded** during grading.

3. **Cheating: We have zero tolerance policy for cheating.** People involved in cheating (any kind of code sharing and codes taken from internet included) will be punished according to the university regulations.

4. **Evaluation:** Your program will be evaluated automatically using "black-box" technique so make sure to obey the specifications. No erroneous input will be used. Also, none of the inputs will include empty-transition loops. Thus, you **do not need to** detect and eliminate computation loops; you can safely assume all string processing will take a finite amount of time.

5. The given sample inputs are only to ease your debugging process and are **not extensive**. Furthermore, it is not guaranteed that they cover all the cases for required functions. As a programmer, it is **your responsibility** to consider such extreme cases for the functions. Your implementations will be evaluated on a more comprehensive set of test cases to determine your **final** grade after the deadline.