# Lab 4

Dragos-Vlad, Tatar

The following link will open github.com, where you can find the files for the laboratory: Link to github

## 1 Symbol Table implementation explanation

In order to create the symbol table I used a binary tree that is made of pointers to nodes, which are a struct containing: an index of type size_t, a value of type string, a left pointer to another node that has a value smaller than the current one, a right pointer that has a value greater than the current one.

```
struct Node {
    size_t index;
    std::string value;
    Node* left;
    Node* right;
};
```

When I want to add a symbol I first check if the symbol table is empty and if it is empty I add a node as the tree head and return 0(first index in symbol table). If it is not empty I check if the symbol is already in the table. If it is, I return the index of the symbol. Otherwise, I search for the position of the symbol in the binary tree, add it to the corresponding position and return the index of the symbol added.

```
class SymbolTable {
public:
    SymbolTable();
    ~SymbolTable();

    size_t add(std::string symbol);
    std::pair<bool, size_t> find(std::string symbol);
    void remove(Node* node);
private:
    Node* m_tree;
    size_t m_current_index;
};
```

## 2 Pif implementation

In order to create the Pif I used a vector of pairs representing the name of the symbol as a string and the symbol table position also as a string. Pif contains the push function that adds an entity to the end of the vector.

```
class Pif
{
public:
    Pif() = default;
    ~Pif() = default;

    const std::vector<std::pair<std::string, std::string>>&
          get_pif() const;

    void push(const std::string& key,
              const std::string& value);

private:
    std::vector<std::pair<std::string, std::string>>
         m_pif;
};
```

## 3 Scanner implementation

The Scanner contains a symbol table and a pif. It has a try_parse function that checks if the file given is lexically correct and returns a pair containing the result of the parse. The parse function writes the symbol table and pif contents to SY.out and Pif.out and prints to screen if the file is lexically correct or not.

```
class Scanner
{
public:
    Scanner(const std::string& path,
            const std::string& tokens_file);
    ~Scanner() = default;

    std::pair<bool, std::pair<int, std::string>>
         try_parse();
    std::string parse();

    const Pif& get_pif() const;
    const SymbolTable& get_symTable() const;

private:
    std::string m_path;
    Pif m_pif;
    SymbolTable m_symTable;
    std::vector<std::string> m_tokens;
};
```

## 4 Transition

The Transition represents a transition of a Finite Automata.

```
class Transition
{
public:
    Transition(const std::string& source,
               const std::string& symbol,
               const std::string& destination);

    std::string get_string() const;

    friend bool operator< (const Transition& el,
                           const Transition& other);

    const std::string& source() const;
    const std::string& symbol() const;
    const std::string& destination() const;

private:
    std::string m_source;
    std::string m_symbol;
    std::string m_destination;
};
```

## 5 Fa

The Fa class represents a finite automata. It contains a set of states, a set of alphabet literals, a set of transitions, an initial state, and a set of final states.

```
class Fa
{
public:
    Fa() = default;
    ~Fa() = default;

    void parse(const std::string& file_path);

    const std::set<std::string>& get_states() const;
    const std::set<std::string>& get_alphabet() const;
    const std::set<Transition>& get_transitions() const;
    const std::string& get_initial_state() const;
    const std::set<std::string>& get_final_states() const;

    bool isDFA() const;

    bool check(const std::string& word);

private:
    std::set<std::string> m_states;
    std::set<std::string> m_alphabet;
    std::set<Transition> m_transitions;
    std::string m_initial_state;
    std::set<std::string> m_final_states;
    bool m_isDFA = true;
};
```

## 6 Used regexes

```
^([a-zA-Z]|_)[a-zA-Z_0-9]*$ -> for variables

^-?[1-9][0-9]*|0|$ -> for numbers

^\'[a-zA-Z0-9]?\'$ -> for characters

^\"[a-zA-Z0-9\{\}_]*\"$ -> for strings

^-?[1-9][0-9]*|0\.\.([a-zA-Z]|_)[a-zA-Z_0-9]*$ -> for ranges
```

## 7 BNF for FA.in

```
letter       ::= "a" | "b" | ... | "z"
digit        ::= "0" | "1" |...| "9"
character    ::= digit | letter
qualifier    ::= letter character*
transition   ::= "(" qualifier "," character "->" qualifier ")"
initialState ::= "q0 = " qualifier
finalStates  ::= "F = {" (qualifier "\n")+ "}"
alphabet     ::= "E = {" (character "\n")+ "}"
states       ::= "Q = {" (qualifier "\n")+ "}"
transitions  ::= "T = {" (transition "\n)"+ "}"
fa           ::= states "\n"+ alphabet "\n"+ transitions "\n"+
                 initialState "\n"+ finalStates
```