Лабораторная работа № 1.3 «Объектно-ориентированный лексический анализатор»

18 марта 2025 г.

Александр Старовойтов, ИУ9-61Б

Цель работы

Целью данной работы является приобретение навыка реализации лексического анализатора на объектно-ориентированном языке без применения каких-либо средств автоматизации решения задачи лексического анализа.

Индивидуальный вариант

Строковые литералы: ограничены двойными кавычками, не могут пересекать границы строк текста, содержат escape-последовательности «\n», «\“», «\t» и «\\».

Целые числа: последовательности десятичных знаков и знаков «_», начинающиеся с цифры (прочерк не влияет на значение числа).

Идентификаторы: состоят из латинских букв, цифр и знаков «_», «$», «@», не могут начинаться на цифру.

Реализация

Реализован однопроходный и чистый лексический анализатор на C++.

Для поддержки функционального стиля использована библиотека mach7 для паттерн матчинга на C++, а также библиотека immer для иммутабельных структур данных в стиле Clojure.

Кроме того, использован паттерн `std::variant` в связке с `std::visit`.

Файл `main.cpp`

```
#include <iostream>
```

```cpp
#include <utf8.h>
#include <optional>

#include <stewkk/lexer/lexer.hpp>
#include <stewkk/lexer/token.hpp>

int main() {
  stewkk::lexer::TokenizerState state{stewkk::lexer::Whitespace{stewkk::lexer::TokenizerStat
  immer::flex_vector<stewkk::lexer::Token> tokens;
  std::optional<stewkk::lexer::Message> message;
  auto b = std::istreambuf_iterator<char>(std::cin);
  auto e = std::istreambuf_iterator<char>();

  while (true) {
    const char32_t next = utf8::next(b, e);
    std::tie(state, tokens, message) = stewkk::lexer::TokenizeWithEof(next, state);

    for (auto token : tokens) {
      std::cout << ToString(token) << std::endl;
    }
    if (message.has_value()) {
      std::cout << message.value() << std::endl;
    }
    if (b == e) {
      break;
    }
  }

  std::tie(state, tokens, message)
      = stewkk::lexer::TokenizeWithEof(stewkk::lexer::kEofMarker, state);
  for (auto token : tokens) {
    std::cout << ToString(token) << std::endl;
  }
  if (message.has_value()) {
    std::cout << message.value() << std::endl;
  }

  return 0;
}
```

Файл `lexer.hpp`

```cpp
#pragma once

#include <cwctype>
#include <string>
#include <variant>
```

```cpp
#include <tuple>

#include <immer/flex_vector.hpp>
#include <immer/map.hpp>
#include <strong_type/strong_type.hpp>

#include <stewkk/lexer/token.hpp>

namespace stewkk::lexer {

struct TokenizerStateData {
  immer::flex_vector<char32_t> token_prefix;
  Position token_start;
  Position prev;
  Position current;
  immer::map<std::string, std::size_t> ident_to_index;
  immer::flex_vector<std::string> index_to_ident;

  bool operator==(const TokenizerStateData& other) const = default;

  TokenizerStateData DiscardPrefix() const;
  TokenizerStateData AddToPrefix(char32_t c) const;
  TokenizerStateData MovePositionBy(char32_t c) const;
  TokenizerStateData SetTokenStart(Position p) const;
  TokenizerStateData AddIdentIfNotExists(std::string ident) const;
};

template <typename Tag> using StateType = strong::type<TokenizerStateData, Tag, strong::equa

using Whitespace = StateType<struct whitespace_>;
using Str = StateType<struct str_>;
using Escape = StateType<struct escape_>;
using Number = StateType<struct number_>;
using Ident = StateType<struct ident_>;
using Eof = StateType<struct eof_>;

using TokenizerState = std::variant<Whitespace, Str, Escape, Number, Ident, Eof>;

std::string GetName(const Whitespace&);
std::string GetName(const Str&);
std::string GetName(const Escape&);
std::string GetName(const Number&);
std::string GetName(const Ident&);
std::string GetName(const Eof&);

using Message = std::string;
```

```
using Tokens = immer::flex_vector<Token>;
using Messages = immer::flex_vector<Message>;

using TokenizerOutput = std::tuple<TokenizerState, std::optional<Token>, std::optional<Messa
using TokenizerStringOutput = std::tuple<TokenizerState, Tokens, Messages>;

TokenizerOutput Tokenize(
    char32_t code_point, TokenizerState state);

std::tuple<TokenizerState, immer::flex_vector<Token>, std::optional<Message>> TokenizeWithEo
    char32_t code_point, TokenizerState state);

TokenizerStringOutput Tokenize(std::string input, TokenizerState state);

std::string ToString(immer::flex_vector<char32_t> s);

constexpr const int kEofMarker = -1;

}  // namespace stewkk::lexer
```

Файл `lexer.cpp`:

```
#include <stewkk/lexer/lexer.hpp>

#include <stdexcept>
#include <functional>

#include <utf8.h>
#include <mach7/type_switchN-patterns.hpp> // Support for N-ary Match statement on patterns
#include <mach7/patterns/predicate.hpp>    // Support for predicate patterns
#include <mach7/patterns/constructor.hpp>  // Support for constructor patterns

namespace stewkk::lexer {

namespace {

Position NextPosition(Position pos, char32_t code_point) {
  return code_point == '\n' ? Position{.line = pos.line + 1, .column = 0}
                            : Position{.line = pos.line, .column = pos.column + 1};
}

bool IsSpace(char32_t c) { return std::iswspace(c); }

bool IsDigit(char32_t c) { return std::iswdigit(c); }

bool IsAlpha(char32_t c) { return std::iswalpha(c); };
```

```
bool Contains(std::u32string s, char32_t c) {
  return std::find(std::begin(s), std::end(s), c) != std::end(s);
}

template <typename Predicate>
concept CharPredicate =
requires(Predicate p, char32_t c) {
    { p(c) } -> std::same_as<bool>;
};

template <CharPredicate... Predicates>
bool Any(char32_t c, Predicates... predicates) {
  return (... || predicates(c));
}

bool IsIdentFirst(char32_t c) {
  return Any(c, std::bind(Contains, U"_$@", std::placeholders::_1), IsAlpha);
}

TokenizerOutput HandleState(
    char32_t code_point, const Whitespace& state) {
  const auto [token_prefix, token_start, prev, current, ident_to_index, index_to_ident]
      = state.value_of();

  Match(code_point) {
    Case(IsSpace) return {
        Whitespace(
            state.value_of().DiscardPrefix().MovePositionBy(code_point)),
        std::nullopt, std::nullopt};
    Case(IsDigit) return {Number(state.value_of()
                                     .AddToPrefix(code_point)
                                     .MovePositionBy(code_point)
                                     .SetTokenStart(current)),
                          std::nullopt, std::nullopt};
    Case('"') return {
        Str(state.value_of().DiscardPrefix().MovePositionBy(code_point).SetTokenStart(curren
        std::nullopt, std::nullopt};
    Case(IsIdentFirst) return {Ident(state.value_of()
                                         .AddToPrefix(code_point)
                                         .MovePositionBy(code_point)
                                         .SetTokenStart(current)),
                               std::nullopt, std::nullopt};
    Case(kEofMarker) return {
        Eof(state.value_of().DiscardPrefix().MovePositionBy(code_point).SetTokenStart(curren
        std::nullopt, std::nullopt};
```

```
    Otherwise() return {Whitespace(state.value_of().MovePositionBy(code_point)), std::nullop
                        std::format("Unknown symbol at ({}:{}): {}", current.line+1, current
                                    ToString({code_point}))};
  }
  EndMatch throw std::logic_error{"unreachable"};
}

TokenizerOutput HandleState(char32_t code_point, const Str& state) {
  const auto [token_prefix, token_start, prev, current, ident_to_index, index_to_ident]
      = state.value_of();

  Match(code_point) {
    Case('\\') return {Escape(state.value_of().MovePositionBy(code_point)), std::nullopt,
                       std::nullopt};
    Case('"') return {Whitespace(state.value_of()
                                      .DiscardPrefix()
                                      .SetTokenStart(NextPosition(current, code_point))
                                      .MovePositionBy(code_point)),
                      StringLiteralToken(Coords{token_start, current}, ToString(token_prefix
                      std::nullopt};
    Case(kEofMarker) return {Eof(state.value_of()
                                      .DiscardPrefix()
                                      .SetTokenStart(current)
                                      .MovePositionBy(code_point)),
                      StringLiteralToken(Coords{token_start, prev}, ToString(token_prefix)),
                      std::format("Expected closing \" at ({}:{})", current.line+1, current.
    Otherwise() return {Str(state.value_of().AddToPrefix(code_point).MovePositionBy(code_poi
                        std::nullopt, std::nullopt};
  }
  EndMatch throw std::logic_error{"unreachable"};
}

TokenizerOutput HandleState(char32_t code_point, const Escape& state) {
  const auto [token_prefix, token_start, prev, current, ident_to_index, index_to_ident]
      = state.value_of();

  Match(code_point) {
    Case('n') return {Str(state.value_of().AddToPrefix('\n').MovePositionBy(code_point)),
                      std::nullopt, std::nullopt};
    Case('"') return {Str(state.value_of().AddToPrefix('\"').MovePositionBy(code_point)),
                                      std::nullopt, std::nullopt};
    Case('\\') return {Str(state.value_of().AddToPrefix('\\').MovePositionBy(code_point)),
                                      std::nullopt, std::nullopt};
    Case('t') return {Str(state.value_of().AddToPrefix('\t').MovePositionBy(code_point)),
                      std::nullopt, std::nullopt};
    Case(kEofMarker) return {
```

```
        Eof(state.value_of().DiscardPrefix().MovePositionBy(code_point).SetTokenStart(curren
        StringLiteralToken(Coords{token_start, prev}, ToString(token_prefix)),
        std::format("Expected literal symbol and closing \" at ({}:{})", current.line+1,
                    current.column+1)};
    Otherwise() return {Str(state.value_of().AddToPrefix(code_point).MovePositionBy(code_poi
                        std::nullopt,
                        std::format("Unknown escape sequence at ({}:{}): \\{}", current.line
                                    current.column+1, ToString({code_point}))};
  }
  EndMatch throw std::logic_error{"unreachable"};
}

bool IsIdentFirstNotUnderscore(char32_t c) {
  return Any(c, IsAlpha, std::bind(Contains, U"$@", std::placeholders::_1));
}

TokenizerOutput HandleState(char32_t code_point, const Number& state) {
  const auto [token_prefix, token_start, prev, current, ident_to_index, index_to_ident]
      = state.value_of();

  Match(code_point) {
    Case(IsDigit) return {
        Number(state.value_of().AddToPrefix(code_point).MovePositionBy(code_point)),
        std::nullopt, std::nullopt};
    Case('_') return {
        Number(state.value_of().MovePositionBy(code_point)),
        std::nullopt, std::nullopt};
    Case(IsSpace) return {
        Whitespace(state.value_of().DiscardPrefix().MovePositionBy(code_point).SetTokenStart
        IntegerToken(Coords{token_start, prev}, std::stoll(ToString(token_prefix))), std::nu
    Case(IsIdentFirstNotUnderscore) return {
        Ident(state.value_of().AddToPrefix(code_point).MovePositionBy(code_point)),
        IntegerToken(Coords{token_start, prev}, std::stoll(ToString(token_prefix))), std::nu
    Case(kEofMarker) return {
        Eof(state.value_of().DiscardPrefix().MovePositionBy(code_point).SetTokenStart(curren
        IntegerToken(Coords{token_start, prev}, std::stoll(ToString(token_prefix))), std::nu
    Otherwise() return {Number(state.value_of().MovePositionBy(code_point)), std::nullopt,
                        std::format("Unknown symbol at ({}:{}): {}", current.line+1, current
                                    ToString({code_point}))};
  }
  EndMatch throw std::logic_error{"unreachable"};
}

bool IsIdent(char32_t c) {
  return Any(c, IsAlpha, IsDigit, std::bind(Contains, U"@_$", std::placeholders::_1));
}
```

```
std::size_t GetIdentIndex(immer::map<std::string, std::size_t> ident_to_index, std::string i
  const auto it = ident_to_index.find(ident);
  return it == nullptr ? ident_to_index.size() : *it;
}

TokenizerOutput HandleState(
    char32_t code_point, const Ident& state) {
  const auto [token_prefix, token_start, prev, current, ident_to_index, index_to_ident]
      = state.value_of();

  Match(code_point) {
    Case('"') return {Str(state.value_of()
                              .DiscardPrefix()
                              .SetTokenStart(current)
                              .MovePositionBy(code_point)
                              .AddIdentIfNotExists(ToString(token_prefix))),
                      IdentToken(Coords{token_start, prev},
                                 GetIdentIndex(ident_to_index, ToString(token_prefix))),
                      std::nullopt};
    Case(IsSpace) return {Whitespace(state.value_of()
                                         .DiscardPrefix()
                                         .SetTokenStart(current)
                                         .MovePositionBy(code_point)
                                         .AddIdentIfNotExists(ToString(token_prefix))),
                          IdentToken(Coords{token_start, prev},
                                     GetIdentIndex(ident_to_index, ToString(token_prefix))),
                          std::nullopt};
    Case(IsIdent) return {
        Ident(state.value_of().AddToPrefix(code_point).MovePositionBy(code_point)), std::nul
        std::nullopt};
    Case(kEofMarker) return {Eof(state.value_of()
                                     .DiscardPrefix()
                                     .MovePositionBy(code_point)
                                     .SetTokenStart(current)
                                     .AddIdentIfNotExists(ToString(token_prefix))),
                             IdentToken(Coords{token_start, prev},
                                        GetIdentIndex(ident_to_index, ToString(token_prefix)
                             std::nullopt};
    Otherwise() return {Ident(state.value_of().MovePositionBy(code_point)), std::nullopt,
                        std::format("Unknown symbol at ({}:{}): {}", prev.line+1, prev.colum
                                    ToString({code_point}))};
  }
  EndMatch throw std::logic_error{"unreachable"};
}
```

```
TokenizerOutput HandleState(char32_t code_point, const Eof& state) {
  throw std::logic_error{"HandleState called from EOF state"};
}

std::pair<char32_t, std::string> NextCodePoint(std::string s) {
  auto b = std::begin(s);
  auto e = std::end(s);

  const char32_t next = utf8::next(b, e);

  return {next, std::string(b, e)};
}

}  // namespace

TokenizerOutput Tokenize(
    char32_t code_point, TokenizerState state) {
  return std::visit([&code_point](const auto& state) { return HandleState(code_point, state)
                    state);
}

std::tuple<TokenizerState, immer::flex_vector<Token>, std::optional<Message>> TokenizeWithEo
  char32_t code_point, TokenizerState state) {
  Position pos = std::visit([](const auto& state_v) {
    return state_v.value_of().current;
  }, state);
  const auto [new_state, token, message] = Tokenize(code_point, state);
  if (code_point == kEofMarker && token.has_value()) {
    return {new_state, immer::flex_vector<Token>{token.value(), EofToken(Coords{pos, pos}, '
  }
  if (code_point == kEofMarker) {
    return {new_state, immer::flex_vector<Token>{EofToken(Coords{pos, pos}, ' ')}, message};
  }
  if (token.has_value()) {
    return {new_state, immer::flex_vector<Token>{token.value()}, message};
  }
  return {new_state, {}, message};
}

TokenizerStringOutput Tokenize(std::string s, TokenizerState state) {
  return s.empty() ? std::make_tuple(state, Tokens{}, Messages{})
                   : [&state, &s] {
                       const auto [code_point, rest] = NextCodePoint(s);

                       const auto [next_state, token, message] = Tokenize(code_point, state)
                       const auto [res_state, tokens, messages] = Tokenize(rest, next_state)
```

```
                      const auto res_tokens = token.has_value() ? tokens.push_front(token.v
                      const auto res_messages = message.has_value() ? messages.push_front(m
                      return std::make_tuple(res_state, res_tokens, res_messages);
                    }();
}

std::string GetName(const Whitespace&) {
  return "WS";
}

std::string GetName(const Str&) {
  return "STR";
}

std::string GetName(const Escape&) {
  return "ESCAPE";
}

std::string GetName(const Number&) {
  return "INTEGER";
}

std::string GetName(const Ident&) {
  return "IDENT";
}

std::string GetName(const Eof&) {
  return "EOF";
}

TokenizerStateData TokenizerStateData::DiscardPrefix() const {
  return TokenizerStateData{
      .token_prefix = {},
      .token_start = token_start,
      .prev = prev,
      .current = current,
      .ident_to_index = ident_to_index,
      .index_to_ident = index_to_ident,
  };
}

TokenizerStateData TokenizerStateData::AddToPrefix(char32_t c) const {
  return TokenizerStateData{
      .token_prefix = token_prefix.push_back(c),
      .token_start = token_start,
```

```
      .prev = prev,
      .current = current,
      .ident_to_index = ident_to_index,
      .index_to_ident = index_to_ident,
  };
}

TokenizerStateData TokenizerStateData::MovePositionBy(char32_t c) const {
  return TokenizerStateData{
      .token_prefix = token_prefix,
      .token_start = token_start,
      .prev = current,
      .current = NextPosition(current, c),
      .ident_to_index = ident_to_index,
      .index_to_ident = index_to_ident,
  };
}

TokenizerStateData TokenizerStateData::SetTokenStart(Position p) const {
  return TokenizerStateData{
      .token_prefix = token_prefix,
      .token_start = p,
      .prev = prev,
      .current = current,
      .ident_to_index = ident_to_index,
      .index_to_ident = index_to_ident,
  };
}

TokenizerStateData TokenizerStateData::AddIdentIfNotExists(std::string ident) const {
  return ident_to_index.find(ident) == nullptr ? TokenizerStateData{
      .token_prefix = token_prefix,
      .token_start = token_start,
      .prev = prev,
      .current = current,
      .ident_to_index = ident_to_index.set(ident, index_to_ident.size()),
      .index_to_ident = index_to_ident.push_back(ident),
  } : TokenizerStateData{
      .token_prefix = token_prefix,
      .token_start = token_start,
      .prev = prev,
      .current = current,
      .ident_to_index = ident_to_index,
      .index_to_ident = index_to_ident,};
}
```

```
std::string ToString(immer::flex_vector<char32_t> s) {
  std::string res;
  for (const auto& sym : s) {
    utf8::append(sym, res);
  }
  return res;
}

}  // namespace stewkk::lexer
```

Файл `token.cpp`:

```
#include <stewkk/lexer/token.hpp>

#include <mach7/type_switchN-patterns.hpp> // Support for N-ary Match statement on patterns
#include <mach7/patterns/constructor.hpp>  // Support for constructor patterns

namespace stewkk::lexer {

std::string GetName(const StringLiteralToken&) {
    return "STRING";
}

std::string GetName(const IntegerToken&) {
    return "INTEGER";
}

std::string GetName(const IdentToken&) {
    return "IDENT";
}

std::string GetName(const EofToken&) {
    return "EOF";
}

std::string ToString(Token token_variant) {
  return std::visit(
      [](const auto& token) {
        const auto& token_value = token.value_of();
        return std::format("{} ({}:{})-({}:{}): {}", GetName(token), token_value.coords.star
                           token_value.coords.start.column+1, token_value.coords.end.line+1,
                           token_value.coords.end.column+1, token_value.attr);
      },
      token_variant);
}
```

```
}  // namespace stewkk::lexer
```

Тестирование

Тесты:

```
#include <gmock/gmock.h>

#include <iostream>
#include <variant>

#include <utf8.h>

#include <stewkk/lexer/token.hpp>
#include <stewkk/lexer/lexer.hpp>

using ::testing::Eq;

using std::string_literals::operator""s;

namespace stewkk::lexer {

TokenizerStringOutput PrintState(const TokenizerStringOutput out) {
  const auto [state_variant, tokens, messages] = out;
  std::visit(
      [](const auto& state) {
        const auto& state_value = state.value_of();
        std::cout << GetName(state) << std::endl;
        std::cout << std::format("    prefix: '{}'\n", ToString(state_value.token_prefix));
        std::cout << std::format("    token start: {} {}\n", state_value.token_start.line,
                                 state_value.token_start.column);
        std::cout << std::format("    prev: {} {}\n", state_value.prev.line, state_value.pre
        std::cout << std::format("    current: {} {}\n", state_value.current.line, state_val
      },
      state_variant);
  for (const auto& token : tokens) {
    std::cout << "    " << ToString(token) << std::endl;
  }
  for (const auto& message : messages) {
    std::cout << "    " << message << std::endl;
  }
  return out;
}

TokenizerState GetStartState() {
```

```
  return Whitespace(TokenizerStateData{
      .token_prefix = immer::flex_vector<char32_t>{},
      .token_start = Position{0, 0},
      .prev = Position{0, 0},
      .current = Position{0, 0},
      .ident_to_index = immer::map<std::string, std::size_t>{},
      .index_to_ident = immer::flex_vector<std::string>{},
  });
}

TEST(LexerTest, TokenizeSymbol) {
  ASSERT_THAT(Tokenize('0', GetStartState()),
              Eq(std::make_tuple(TokenizerState(Number(TokenizerStateData{
                                     .token_prefix = immer::flex_vector<char32_t>{'0'},
                                     .token_start = Position{0, 0},
                                     .prev = Position{0, 0},
                                     .current = Position{0, 1},
                                 })),
                                 std::nullopt, std::nullopt)));
}

TEST(LexerTest, TokenizeString) {
  ASSERT_THAT(Tokenize("  0"s, GetStartState()),
              Eq(std::make_tuple(TokenizerState(Number(TokenizerStateData{
                                     .token_prefix = immer::flex_vector<char32_t>{'0'},
                                     .token_start = Position{0, 2},
                                     .prev = Position{0, 2},
                                     .current = Position{0, 3},
                                 })),
                                 Tokens{}, Messages{})));
}

TEST(LexerTest, TokenizeQuote) {
  ASSERT_THAT(Tokenize("  \""s, GetStartState()),
              Eq(std::make_tuple(TokenizerState(Str(TokenizerStateData{
                                     .token_prefix = immer::flex_vector<char32_t>{},
                                     .token_start = Position{0, 2},
                                     .prev = Position{0, 2},
                                     .current = Position{0, 3},
                                 })),
                                 Tokens{}, Messages{})));
}

TEST(LexerTest, TokenizeIdent) {
  ASSERT_THAT(Tokenize("  Y"s, GetStartState()),
              Eq(std::make_tuple(TokenizerState(Ident(TokenizerStateData{
```

```
                                    .token_prefix = immer::flex_vector<char32_t>{'Y'},
                                    .token_start = Position{0, 2},
                                    .prev = Position{0, 2},
                                    .current = Position{0, 3},
                                })),
                                Tokens{}, Messages{})));
}

TEST(LexerTest, TokenizeFullString) {
  ASSERT_THAT(
      Tokenize("  \"oa\\to \\\" a\" "s, GetStartState()),
      Eq(std::make_tuple(TokenizerState(Whitespace(TokenizerStateData{
                              .token_prefix = immer::flex_vector<char32_t>{},
                              .token_start = Position{0, 14},
                              .prev = Position{0, 14},
                              .current = Position{0, 15},
                          })),
                          Tokens{StringLiteralToken(Coords{{0, 2}, {0, 13}}, "oa\to \" a")},
}

TEST(LexerTest, TokenizeInteger) {
  ASSERT_THAT(Tokenize("  123_999 "s, GetStartState()),
              Eq(std::make_tuple(TokenizerState(Whitespace(TokenizerStateData{
                                    .token_prefix = immer::flex_vector<char32_t>{},
                                    .token_start = Position{0, 9},
                                    .prev = Position{0, 9},
                                    .current = Position{0, 10},
                                })),
                                Tokens{IntegerToken(Coords{{0, 2}, {0, 8}}, 123999)}, Messa
}

TEST(LexerTest, TokenizeIdentFull) {
  ASSERT_THAT(
      Tokenize("  Y1@_123 "s, GetStartState()),
      Eq(std::make_tuple(TokenizerState(Whitespace(TokenizerStateData{
                              .token_prefix = immer::flex_vector<char32_t>{},
                              .token_start = Position{0, 9},
                              .prev = Position{0, 9},
                              .current = Position{0, 10},
                              .ident_to_index = immer::map<std::string, std::size_t>{{"Y1@_12
                              .index_to_ident = immer::flex_vector<std::string>{{"Y1@_123"}},
                          })),
                          Tokens{IdentToken(Coords{{0, 2}, {0, 8}}, 0u)}, Messages{})));
}

TEST(LexerTest, TokenizeError) {
```

```
  ASSERT_THAT(Tokenize("  * "s, GetStartState()),
              Eq(std::make_tuple(TokenizerState(Whitespace(TokenizerStateData{
                                      .token_prefix = immer::flex_vector<char32_t>{},
                                      .token_start = Position{0, 0},
                                      .prev = Position{0, 3},
                                      .current = Position{0, 4},
                                  })),
                                  Tokens{},
                                  Messages{
                                      "Unknown symbol at (1:3): *",
                                  })));
}

TEST(LexerTest, TokenizeUnicodeString) {
  ASSERT_THAT(
      Tokenize("\"я русский!\""s, GetStartState()),
      Eq(std::make_tuple(TokenizerState(Whitespace(TokenizerStateData{
                              .token_prefix = immer::flex_vector<char32_t>{},
                              .token_start = Position{0, 12},
                              .prev = Position{0, 11},
                              .current = Position{0, 12},
                          })),
                          Tokens{StringLiteralToken(Coords{{0, 0}, {0, 11}}, "я русский!")},
}

TEST(LexerTest, TokenizeEOF) {
  const auto [state, tokens, messages] = Tokenize("123"s, GetStartState());
  ASSERT_THAT(Tokenize(kEofMarker, state),
              Eq(std::make_tuple(TokenizerState(Eof(TokenizerStateData{
                                      .token_prefix = immer::flex_vector<char32_t>{},
                                      .token_start = Position{0, 3},
                                      .prev = Position{0, 3},
                                      .current = Position{0, 4},
                                  })),
                                  Token{IntegerToken(Coords{{0, 0}, {0, 2}}, 123)}, std::null
}

}  // namespace stewkk::lexer
```

Вывод

В рамках данной лабораторной работы, я приобрел навыки реализации лексического анализатора на объектно-ориентированном языке без применения каких-либо средств автоматизации решения задачи лексического анализа.

Был реализован однопроходный, чистый функциональный парсер на C++.