Лабораторная работа № 1.2. «Лексический анализатор на основе регулярных выражений»

5 марта 2025 г.

Александр Старовойтов, ИУ9-61Б

Цель работы

Целью данной работы является приобретение навыка разработки простейших лексических анализаторов, работающих на основе поиска в тексте по образцу, заданному регулярным выражением.

Индивидуальный вариант

- Открывающий тег: последовательность букв и цифр, окружённая «<» и «>».
- Закрывающий тег: последовательность букв и цифр, окружённая «</» и «>».
- Пробел (значимый токен): любая последовательность пробельных символов.
- Ключевое слово: «<», «>», «&».
- Символ: любой печатный символ, кроме «<», «>» и «&».

Реализация

https://github.com/stewkk/iu9-compilers/pull/1

```
#pragma once

#include <array>
#include <string>

namespace stewkk::lexer {

enum class DomainType { kOpeningTag, kClosingTag, kWhitespace, kLt, kGt, kAmp, kSymbol };

std::string ToString(DomainType type);
```

```
struct Domain {
  std::string pattern;
  DomainType type;
};

static const std::array<Domain, 7> kDomainPatterns{{
    {R"((<(?:\w|\d)+>))", DomainType::kOpeningTag},
    {R"((<\/(?:\w|\d)+>))", DomainType::kClosingTag},
    {R"((\s+))", DomainType::kWhitespace},
    {R"((&lt;))", DomainType::kLt},
    {R"((&gt;))", DomainType::kGt},
    {R"((&amp;))", DomainType::kAmp},
    {R"(([^&<>]))", DomainType::kSymbol},
}};

}  // namespace stewkk::lexer
```

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

namespace stewkk::lexer {

std::string ToString(DomainType type) {
  switch (type) {
    case DomainType::kOpeningTag:
      return "OTAG";
    case DomainType::kClosingTag:
      return "CTAG";
    case DomainType::kWhitespace:
      return "WS";
    case DomainType::kLt:
      return "LT";
    case DomainType::kGt:
      return "GT";
    case DomainType::kAmp:
      return "AMP";
    case DomainType::kSymbol:
      return "SYM";
  }
  throw "unreachable";
}

}  // namespace stewkk::lexer
```

```
#pragma once

#include <cstddef>
```

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

namespace stewkk::lexer {

struct Position {
  std::size_t line;
  std::size_t column;

  bool operator==(const Position&) const = default;
};

struct Token {
  DomainType domain;
  Position position;
  std::string text;

  bool operator==(const Token&) const = default;
};

}  // namespace stewkk::lexer
```

```cpp
#pragma once

#include <exception>

#include <stewkk/lexer/token.hpp>

namespace stewkk::lexer {

class LexerError : public std::exception {
public:
  explicit LexerError(Position pos);
  Position GetPosition() const;

private:
  Position pos_;
};

}  // namespace stewkk::lexer
```

https://github.com/stewkk/iu9-compilers/pull/1

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

namespace stewkk::lexer {

LexerError::LexerError(Position pos) : pos_(std::move(pos)) {}
```

```
Position LexerError::GetPosition() const { return pos_; }

}  // namespace stewkk::lexer
```

```
#pragma once

#include <optional>
#include <string_view>

#include <stewkk/lexer/token.hpp>

namespace stewkk::lexer {

struct Match {
  Token match;
  std::string_view rest;

  bool operator==(const Match&) const = default;
};

class Matcher {
public:
  std::optional<Match> NextMatch(std::string_view text, Position pos);
};

}  // namespace stewkk::lexer
```

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

#include <re2/re2.h>
#include <range/v3/algorithm/max.hpp>
#include <range/v3/view/transform.hpp>
#include <range/v3/view/zip.hpp>

namespace stewkk::lexer {

namespace {

struct TextMatch {
  std::string_view rest;
  std::string content;
};

std::optional<TextMatch> TryMatch(std::string_view text, const std::string& pattern) {
  std::string content;
  if (!RE2::Consume(&text, pattern, &content)) {
```

```
    return std::nullopt;
  }

  return TextMatch{
      .rest = text,
      .content = content,
  };
}

}  // namespace

std::optional<Match> Matcher::NextMatch(std::string_view text, Position pos) {
  const auto matches = kDomainPatterns
                       // Find matches in text
                       | ranges::views::transform(
                           [&text](const auto& domain) { return TryMatch(text, domain.patter

  auto [match, domain] = ranges::max(ranges::views::zip(matches, kDomainPatterns), std::less
                                     [](const auto& match_with_domain) {
                                       const auto& [match, _] = match_with_domain;
                                       if (!match.has_value()) {
                                         return 0ul;
                                       }
                                       const auto& content = match.value().content;
                                       return content.size();
                                     });

  if (!match.has_value()) {
    return std::nullopt;
  }

  return Match{
      .match = Token{
        .domain = domain.type,
        .position = pos,
        .text = std::move(match).value().content,
      },
      .rest = std::move(match).value().rest,
    };
}

}  // namespace stewkk::lexer
```

```
#pragma once

#include <iterator>
```

```cpp
#include <string>
#include <string_view>

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

namespace stewkk::lexer {

class Lexer {
public:
  explicit Lexer(std::string text);

  struct Iterator {
    using difference_type = std::ptrdiff_t;
    using value_type = Token;

    Iterator();
    Iterator(std::string_view rest, Matcher m);

    Token operator*() const;

    Iterator& operator++();

    Iterator operator++(int);

    bool operator==(const Iterator& other) const;

    Token token_;
    Position pos_;
    std::string_view rest_;
    Matcher m_;
  };
  static_assert(std::forward_iterator<Iterator>);

  const Iterator begin();
  const Iterator end();

private:
  std::string text_;
  Matcher m_;
};

void OutputTokens(const std::string& text);

}  // namespace stewkk::lexer
```

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

#include <algorithm>
#include <format>
#include <iostream>

#include <stewkk/lexer/error.hpp>

namespace stewkk::lexer {

using Iterator = Lexer::Iterator;

namespace {

Position NextPosition(Position pos, const std::string& token) {
  auto it = token.begin();
  while (true) {
    auto prev = it;
    it = std::find(it, token.end(), '\n');
    ;
    if (it == token.end()) {
      pos.column += it - prev;
      return pos;
    }
    pos.line += 1;
    pos.column = 1;
    it++;
  }
}

}  // namespace

Lexer::Lexer(std::string text) : text_(text), m_() {}

Iterator::Iterator() : token_(), rest_() {}

Iterator::Iterator(std::string_view rest, Matcher m)
    : token_(), rest_(std::move(rest)), m_(std::move(m)), pos_(Position{1, 1}) {
  if (rest_.empty()) {
    return;
  }
  auto match = m_.NextMatch(rest_, pos_);
  if (!match.has_value()) {
    throw LexerError{std::move(pos_)};
  }
  token_ = std::move(match).value().match;
```

```
  rest_ = std::move(match).value().rest;
  pos_ = NextPosition(pos_, token_.text);
}

Token Iterator::operator*() const { return token_; }

Iterator& Iterator::operator++() {
  if (rest_.empty()) {
    token_ = Token{};
    return *this;
  }

  auto match = m_.NextMatch(rest_, pos_);
  if (!match.has_value()) {
    throw LexerError{std::move(pos_)};
  }

  token_ = std::move(match).value().match;
  rest_ = std::move(match).value().rest;
  pos_ = NextPosition(pos_, token_.text);
  return *this;
}

Iterator Iterator::operator++(int) {
  auto tmp = *this;
  ++*this;
  return tmp;
}

bool Iterator::operator==(const Iterator& other) const { return rest_ == other.rest_; }

const Iterator Lexer::begin() { return Iterator(text_, m_); }

const Iterator Lexer::end() { return Iterator(); }

void OutputTokens(const std::string& text) {
  Lexer l(text);
  try {
    for (const auto token : l) {
      std::cout << std::format("{} ({},{}): {}", ToString(token.domain), token.position.line
                               token.position.column, token.text)
                << std::endl;
    }
  } catch (const LexerError& e) {
    std::cerr << std::format("syntax error ({},{})", e.GetPosition().line, e.GetPosition().c
              << std::endl;
```

```
  }
}

}  // namespace stewkk::lexer
```

```
#include <iostream>
#include <string>

#include <stewkk/lexer/lexer.hpp>

int main() {
  std::string input((std::istreambuf_iterator<char>(std::cin)), std::istreambuf_iterator<cha
  stewkk::lexer::OutputTokens(input);
  return 0;
}
```

Тестирование

Тесты

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

#include <string_view>

#include <stewkk/lexer/domain.hpp>
#include <stewkk/lexer/error.hpp>
#include <stewkk/lexer/lexer.hpp>
#include <stewkk/lexer/matcher.hpp>
#include <stewkk/lexer/token.hpp>

using ::testing::Eq;
using ::testing::Optional;

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

namespace stewkk::lexer {

TEST(MatcherTest, MatchesOpeningTag) {
  auto text = "<div> </div>"sv;
  Matcher m;

  auto match = m.NextMatch(text, Position{1, 1});

  ASSERT_THAT(match, Optional(Match{
      Token{
```

```
        .domain = DomainType::kOpeningTag,
        .position = Position{
          .line = 1,
          .column = 1,
        },
        .text = "<div>"s,
      },
      " </div>"sv}));
}

TEST(MatcherTest, MatchesClosingTag) {
  auto text = "</div>"sv;
  Matcher m;

  auto match = m.NextMatch(text, Position{1, 1});

  ASSERT_THAT(match, Optional(Match{
      Token{
        .domain = DomainType::kClosingTag,
        .position = Position{
          .line = 1,
          .column = 1,
        },
        .text = "</div>"s,
      },
      ""sv}));
}

TEST(MatcherTest, MatchesSpaces) {
  auto text = "  \n <div>"sv;
  Matcher m;

  auto match = m.NextMatch(text, Position{1, 1});

  ASSERT_THAT(match, Optional(Match{
      Token{
        .domain = DomainType::kWhitespace,
        .position = Position{
          .line = 1,
          .column = 1,
        },
        .text = "  \n "s,
      },
      "<div>"sv}));
}
```

```
TEST(MatcherTest, MatchesLt) {
  auto text = "&lt;  \n <div>"sv;
  Matcher m;

  auto match = m.NextMatch(text, Position{1, 1});

  ASSERT_THAT(match, Optional(Match{
      Token{
        .domain = DomainType::kLt,
        .position = Position{
          .line = 1,
          .column = 1,
        },
        .text = "&lt;"s,
      },
      "  \n <div>"sv}));
}

TEST(MatcherTest, MatchesSymbol) {
  auto text = "lt;  \n <div>"sv;
  Matcher m;

  auto match = m.NextMatch(text, Position{1, 1});

  ASSERT_THAT(match, Optional(Match{
      Token{
        .domain = DomainType::kSymbol,
        .position = Position{
          .line = 1,
          .column = 1,
        },
        .text = "l"s,
      },
      "t;  \n <div>"sv}));
}

TEST(LexerTest, IteratesOverTokens) {
  Lexer l("<div> ");
  auto it = l.begin();

  ASSERT_THAT(*it, Eq(
      Token{
        .domain = DomainType::kOpeningTag,
        .position = Position{
          .line = 1,
          .column = 1,
```

```
        },
        .text = "<div>"s,
      }));

  it++;

  ASSERT_THAT(*it, Eq(
      Token{
        .domain = DomainType::kWhitespace,
        .position = Position{
          .line = 1,
          .column = 6,
        },
        .text = " "s,
      }));

  it++;

  ASSERT_THAT(it, Eq(l.end()));
}

TEST(LexerTest, HandlesEOL) {
  Lexer l("<div> \n </div>");
  auto it = l.begin();
  it++;
  it++;

  ASSERT_THAT(*it, Eq(
      Token{
        .domain = DomainType::kClosingTag,
        .position = Position{
          .line = 2,
          .column = 2,
        },
        .text = "</div>"s,
      }));
}

TEST(LexerTest, HandlesTwoEOL) {
  Lexer l("<div> \n \n</div>");
  auto it = l.begin();
  it++;
  it++;

  ASSERT_THAT(*it, Eq(
      Token{
```

```
        .domain = DomainType::kClosingTag,
        .position = Position{
          .line = 3,
          .column = 1,
        },
        .text = "</div>"s,
      }));
}

}  // namespace stewkk::lexer
```

Вывод

В рамках данной лабораторной работы я приобрел навыки разработки простейших лексических анализаторов, работающих на основе поиска в тексте по образцу, заданному регулярным выражением. А также навыки реализации простого итератора для своего класса на языке C++.