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

13 марта 2024 г.

Илья Афанасьев, 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

- Строковые литералы: ограничены обратными кавычками, могут занимать несколько строчек текста, для включения обратной кавычки она удваивается.
- Числовые литералы: десятичные литералы представляют собой последовательности десятичных цифр, двоичные — последовательности нулей и единиц, оканчивающиеся буквой «b».
- Идентификаторы: последовательности десятичных цифр и знаков «?», «*» и «|», не начинающиеся с цифры.

Реализация

Файл `position.hpp`

```
#pragma once

#include <memory>

namespace lexer {

class Position final {
 public:
  Position(std::shared_ptr<const std::string> text) noexcept
      : text_(std::move(text)), it_(text_->cbegin()), line_(1), pos_(1) {}
```

```
  std::string::const_iterator get_it() const noexcept { return it_; }
  std::size_t get_line() const noexcept { return line_; }
  std::size_t get_pos() const noexcept { return pos_; }

  bool IsEnd() const noexcept;
  bool IsWhitespace() const noexcept;
  bool IsNewLine() const noexcept;
  void Next() noexcept;

  void Dump(std::ostream& os) const;

 private:
  std::shared_ptr<const std::string> text_;
  std::string::const_iterator it_;
  std::size_t line_;
  std::size_t pos_;
};

std::ostream& operator<<(std::ostream& os, const Position& p);

}  // namespace lexer

namespace std {

template <>
struct less<lexer::Position> {
  bool operator()(const lexer::Position& lhs,
                  const lexer::Position& rhs) const noexcept {
    return (lhs.get_line() < rhs.get_line() ||
            lhs.get_line() == rhs.get_line() && lhs.get_pos() < rhs.get_pos());
  }
};

}  // namespace std
```

Файл `position.cpp`

```
#include "position.hpp"

namespace lexer {

bool Position::IsEnd() const noexcept { return (it_ == text_->end()); }

bool Position::IsWhitespace() const noexcept { return std::isspace(*it_); }

bool Position::IsNewLine() const noexcept {
```

```
  if (*it_ == '\r') {
    const auto suc = it_ + 1;
    return (suc != text_->end() && *suc == '\n');
  }

  return (*it_ == '\n');
}

void Position::Next() noexcept {
  if (IsNewLine()) {
    if (*it_ == '\r') {
      ++it_;
    }

    ++line_;
    pos_ = 1;
  } else {
    ++pos_;
  }

  ++it_;
}

void Position::Dump(std::ostream& os) const {
  os << "(" << line_ << ", " << pos_ << ")";
}

std::ostream& operator<<(std::ostream& os, const Position& p) {
  p.Dump(os);
  return os;
}

}  // namespace lexer
```

Файл `message.hpp`

```
#pragma once

#include <string>

#include "position.hpp"

namespace lexer {

const std::string kSyntaxError = "syntax error";

enum class MessageType {
```

```
  kError,
  kOther,
  kWarning,
};

std::string_view ToString(const MessageType type) noexcept;

class Message final {
 public:
  Message() noexcept : type_(MessageType::kOther) {}
  Message(const MessageType type, const std::string& text) noexcept
      : type_(type), text_(text) {}

  MessageType get_type() const noexcept { return type_; }
  const std::string& get_text() const& noexcept { return text_; }

 private:
  MessageType type_;
  std::string text_;
};

void Print(std::ostream& os, const Message& message, const Position& position);

}  // namespace lexer
```

Файл message.cpp

```
#include "message.hpp"

namespace lexer {

std::string_view ToString(const MessageType type) noexcept {
  switch (type) {
    using enum MessageType;

    case kError:
      return "Error";

    case kOther:
      return "Other";

    case kWarning:
      return "Warning";
  }
}

void Print(std::ostream& os, const Message& message, const Position& position) {
```

```
  os << ToString(message.get_type()) << " " << position << ": "
     << message.get_text();
}

}  // namespace lexer
```

Файл token.hpp

```
#pragma once

#include "compiler.hpp"
#include "position.hpp"

namespace lexer {

class Compiler;

enum class DomainTag {
  kEndOfProgram,
  kIdent,
  kNumber,
  kString,
};

std::string_view ToString(const DomainTag tag) noexcept;

class Token {
 public:
  DomainTag get_tag() const noexcept { return tag_; }
  const Position& get_starting() const& noexcept { return starting_; }

  virtual ~Token() {}

 protected:
  Token(const DomainTag tag, const Position& starting) noexcept
      : tag_(tag), starting_(starting) {}

  DomainTag tag_;
  Position starting_;
};

class IdentToken final : public Token {
 public:
  IdentToken(const std::size_t code, const Position& starting) noexcept
      : Token(DomainTag::kIdent, starting), code_(code) {}

  std::size_t get_code() const noexcept { return code_; }
```

```
 private:
  std::size_t code_;
};

class NumberToken final : public Token {
 public:
  NumberToken(const std::int64_t value, const Position& starting) noexcept
      : Token(DomainTag::kNumber, starting), value_(value) {}

  std::int64_t get_value() const noexcept { return value_; }

 private:
  std::int64_t value_;
};

class StringToken final : public Token {
 public:
  StringToken(const std::string& str, const Position& starting) noexcept
      : Token(DomainTag::kString, starting), str_(str) {}

  StringToken(std::string&& str, const Position& starting) noexcept
      : Token(DomainTag::kString, starting), str_(std::move(str)) {}

  const std::string& get_str() const& noexcept { return str_; }

 private:
  std::string str_;
};

class SpecToken final : public Token {
 public:
  SpecToken(const DomainTag tag, const Position& starting) noexcept
      : Token(tag, starting) {}
};

void Print(std::ostream& os, const Token& token, const Compiler& compiler);

}  // namespace lexer
```

Файл `token.cpp`

```
#include "token.hpp"

namespace lexer {

std::string_view ToString(const DomainTag tag) noexcept {
```

```
  switch (tag) {
    using enum DomainTag;

    case kEndOfProgram:
      return "END_OF_PROGRAM";

    case kIdent:
      return "IDENT";

    case kNumber:
      return "NUMBER";

    case kString:
      return "STRING";
  }
}

void Print(std::ostream& os, const Token& token, const Compiler& compiler) {
  os << token.get_starting() << " " << ToString(token.get_tag()) << " ";

  switch (token.get_tag()) {
    using enum DomainTag;

    case kIdent: {
      const auto ident = static_cast<const IdentToken* const>(&token);
      const auto& name = compiler.GetName(ident->get_code());
      os << name;
      break;
    }

    case kNumber: {
      const auto number = static_cast<const NumberToken* const>(&token);
      os << number->get_value();
      break;
    }

    case kString: {
      const auto str = static_cast<const StringToken* const>(&token);
      os << str->get_str();
      break;
    }
  }
}

}  // namespace lexer
```

Файл compiler.hpp

```
#pragma once

#include <map>
#include <unordered_map>
#include <vector>

#include "message.hpp"
#include "position.hpp"
#include "scanner.hpp"
#include "token.hpp"

namespace lexer {

class Scanner;

class Compiler final {
 public:
  const std::map<Position, Message>& get_messages() const& noexcept {
    return messages_;
  }

  std::size_t AddName(const std::string& name);
  const std::string& GetName(const std::size_t code) const&;

  void AddMessage(const MessageType type, const Position& p,
                  const std::string& text);

 private:
  std::map<Position, Message> messages_;
  std::unordered_map<std::string, std::size_t> name_codes_;
  std::vector<std::string> names_;
};

std::unique_ptr<Scanner> GetScanner(
    const std::shared_ptr<Compiler>& compiler,
    const std::shared_ptr<const std::string>& program) noexcept;

}  // namespace lexer
```

Файл compiler.cpp

```
#include "compiler.hpp"

#include "message.hpp"
```

```
namespace lexer {

std::size_t Compiler::AddName(const std::string& name) {
  if (const auto it = name_codes_.find(name); it != name_codes_.cend()) {
    return it->second;
  }

  const auto code = names_.size();
  names_.push_back(name);
  name_codes_[name] = code;
  return code;
}

const std::string& Compiler::GetName(const std::size_t code) const& {
  return names_.at(code);
}

void Compiler::AddMessage(const MessageType type, const Position& p,
                          const std::string& text) {
  messages_[p] = Message(type, text);
}

std::unique_ptr<Scanner> GetScanner(
    const std::shared_ptr<Compiler>& compiler,
    const std::shared_ptr<const std::string>& program) noexcept {
  return std::make_unique<Scanner>(program, compiler);
}

}  // namespace lexer
```

Файл scanner.hpp

```
#pragma once

#include <memory>

#include "compiler.hpp"
#include "position.hpp"
#include "token.hpp"

namespace lexer {

class Compiler;
class Token;

class Scanner final {
 public:
```

```
  Scanner(std::shared_ptr<const std::string> program,
          std::shared_ptr<Compiler> compiler) noexcept
      : program_(std::move(program)),
        compiler_(std::move(compiler)),
        cur_(program_) {}

  std::unique_ptr<Token> NextToken();

 private:
  std::shared_ptr<const std::string> program_;
  std::shared_ptr<Compiler> compiler_;
  Position cur_;
};

}  // namespace lexer
```

Файл scanner.cpp

```
#include "scanner.hpp"

#include <boost/regex.hpp>
#include <string>

#include "message.hpp"
#include "position.hpp"
#include "token.hpp"

namespace lexer {

namespace {

const std::string kBinary = "BINARY";
const std::string kDecimal = "DECIMAL";
const std::string kIdent = "IDENT";
const std::string kString = "STRING";

std::unique_ptr<Token> GetToken(Compiler& compiler, const Position& cur,
                                const boost::smatch& matches) {
  const auto str = matches[0].str();

  if (matches[kDecimal].matched) {
    const auto value = std::stoi(str);
    return std::make_unique<NumberToken>(value, cur);

  } else if (matches[kBinary].matched) {
    const auto value = std::stoi(str, nullptr, 2);
    return std::make_unique<NumberToken>(value, cur);
```

```
  } else if (matches[kIdent].matched) {
    const auto code = compiler.AddName(str);
    return std::make_unique<IdentToken>(code, cur);

  } else if (matches[kString].matched) {
    static const boost::regex double_back_quote("``");
    static constexpr std::string_view back_quote("`");

    std::ostringstream t;
    std::ostream_iterator<char> oi(t);
    boost::regex_replace(oi, str.begin() + 1, str.end() - 1, double_back_quote,
                         back_quote);
    return std::make_unique<StringToken>(t.str(), cur);

  } else {
    throw std::runtime_error("scanner.cpp: undefined named subexpression");
  }
}

}  // namespace

std::unique_ptr<Token> Scanner::NextToken() {
  // clang-format off
  static const boost::regex regex(
      "(?<" + kBinary + ">[01]+b)|"
      "(?<" + kDecimal + ">\\d+)|"
      "(?<" + kIdent + ">[\\?\\*\\|]([\\?\\*\\|]|\\d)*)|"
      "(?<" + kString + ">`([\\x00-\\x5F\\x61-\\x7F]|``)*`)");
  // clang-format on

  while (!cur_.IsEnd() && cur_.IsWhitespace()) {
    cur_.Next();
  }

  if (cur_.IsEnd()) {
    return std::make_unique<SpecToken>(DomainTag::kEndOfProgram, cur_);
  }

  boost::smatch matches;
  if (!boost::regex_search(cur_.get_it(), program_->cend(), matches, regex)) {
    compiler_->AddMessage(MessageType::kError, cur_, kSyntaxError);
    return std::make_unique<SpecToken>(DomainTag::kEndOfProgram, cur_);
  }

  const auto& match = matches[0];
```

```
  if (cur_.get_it() != match.first) {
    compiler_->AddMessage(MessageType::kError, cur_, kSyntaxError);

    do {
      cur_.Next();
    } while (cur_.get_it() != match.first);
  }

  auto token = GetToken(*compiler_, cur_, matches);

  while (cur_.get_it() != match.second) {
    cur_.Next();
  }

  return token;
}

}  // namespace lexer
```

Файл `main.cpp`

```
#include <algorithm>
#include <fstream>
#include <iostream>

#include "compiler.hpp"

int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cerr << "Usage: lab1_3 <filename>\n";
    return 1;
  }

  std::ifstream file(argv[1]);
  if (!file.is_open()) {
    std::cerr << "Cannot open file " << argv[1] << "\n";
    return 1;
  }

  const auto program = std::make_shared<const std::string>(
      std::istreambuf_iterator<char>(file), std::istreambuf_iterator<char>());

  auto compiler = std::make_shared<lexer::Compiler>();
  auto scanner = lexer::GetScanner(compiler, program);

  std::vector<std::unique_ptr<lexer::Token>> tokens;
```

```
  do {
    tokens.push_back(scanner->NextToken());
  } while (tokens.back()->get_tag() != lexer::DomainTag::kEndOfProgram);

  std::cout << "TOKENS:\n";
  for (const auto& token : tokens) {
    std::cout << '\t';
    lexer::Print(std::cout, *token, *compiler);
    std::cout << '\n';
  }

  std::cerr << "MESSAGES:\n";
  for (const auto& [position, message] : compiler->get_messages()) {
    std::cout << '\t';
    lexer::Print(std::cout, message, position);
    std::cout << '\n';
  }
}
```

Тестирование

Входные данные:

```
  42*101b
 110
    `a_?
10b`` *
``` a ||
11b?``
```

Вывод на `stdout`:

```
TOKENS:
    (1, 3) NUMBER 42
    (1, 5) IDENT *101
    (2, 2) NUMBER 110
    (3, 5) STRING a_?
10b` *
`
    (5, 7) IDENT ||
    (6, 1) NUMBER 3
    (6, 4) IDENT ?
    (6, 5) STRING
    (7, 1) END_OF_PROGRAM
MESSAGES:
```
```

```
Error (1, 9): syntax error
Error (5, 5): syntax error
```

Вывод

В результате выполнения лабораторной работы я получил навыки разработки простейшего лексического анализатора, работающего на основе поиска в тексте по образцу, заданному регулярным выражением. В работе использовалась библиотека Boost.Regex, предоставляющая больше возможностей для работы с регулярными выражениями в сравнении со стандартной библиотекой C++ (в частности, использование именованных групп). До сих пор работать с Boost.Regex не приходилось, поэтому использование библиотеки было новым и полезным опытом.

Архитектура проекта выстраивалась с опорой на лексический анализатор, описанный в лекциях. Делалось это с заделом на лабораторную работу 1.3, хотя и выглядело местами “из пушки по воробьям”. Тем не менее, заниматься реализацией опираясь на образцовый анализатор было увлекательно и познавательно.