Лабораторная работа № 1.3 «Объектно-ориентированный лексический анализатор»

13 марта 2024 г.

Илья Афанасьев, 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

- Числовые литералы: знак «0» либо последовательности знаков «1».
- Строковые литералы: регулярные строки — ограничены двойными кавычками, могут содержать escape-последовательности «\“», «\t», «\n», не пересекают границы строк текста; буквальные строки — начинаются на «@”», заканчиваются на двойную кавычку, пересекают границы строк текста, для включения двойной кавычки она удваивается.

Реализация

Файл `position.hpp`

```
#pragma once

#include <memory>

namespace lexer {

constexpr char kEnd = -1;

class Position final {
 public:
```

```
  Position(std::shared_ptr<const std::string> text) noexcept
      : text_(std::move(text)), line_(1), pos_(1), index_(0) {}

  std::size_t get_line() const noexcept { return line_; }
  std::size_t get_pos() const noexcept { return pos_; }
  std::size_t get_index() const noexcept { return index_; }

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

  Position operator++(int) noexcept;
  Position operator++() noexcept;

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

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

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

}  // namespace lexer

namespace std {

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

}  // namespace std
```

Файл `position.cpp`

```
#include "position.hpp"

namespace lexer {
```

```
char Position::Cp() const noexcept {
  return (index_ == text_->size() ? kEnd : text_->at(index_));
}

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

bool Position::IsWhitespace() const noexcept {
  return (index_ != text_->size() && std::isspace(text_->at(index_)));
}

bool Position::IsNewLine() const noexcept {
  if (index_ == text_->size()) {
    return false;
  }

  if (text_->at(index_) == '\r' && index_ + 1 < text_->size()) {
    return (text_->at(index_ + 1) == '\n');
  }

  return (text_->at(index_) == '\n');
}

void Position::Next() noexcept {
  if (index_ == text_->size()) {
    return;
  }

  if (IsNewLine()) {
    if (text_->at(index_) == '\r') {
      ++index_;
    }

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

  ++index_;
}

Position Position::operator++(int) noexcept {
  auto old = *this;
  Next();
  return old;
}
```

```
Position Position::operator++() noexcept {
  Next();
  return *this;
}

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

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

}  // namespace lexer
```

Файл `fragment.hpp`

```
#pragma once

#include "position.hpp"

namespace lexer {

class Fragment final {
 public:
  Fragment(const Position& starting, const Position& following) noexcept
      : starting_(starting), following_(following) {}

  const Position& get_starting() const& noexcept { return starting_; }
  const Position& get_following() const& noexcept { return following_; }

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

 private:
  Position starting_;
  Position following_;
};

std::ostream& operator<<(std::ostream& os, const Fragment& fragment);

}  // namespace lexer
```

Файл `fragment.cpp`

```
#include "fragment.hpp"
```

```cpp
namespace lexer {

void Fragment::Dump(std::ostream& os) const {
  os << starting_ << "-" << following_;
}

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

}  // namespace lexer
```

Файл message.hpp

```cpp
#pragma once

#include <string>

#include "position.hpp"

namespace lexer {

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 "fragment.hpp"
#include "position.hpp"

namespace lexer {

class Compiler;

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

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

class Token {
 public:
  DomainTag get_tag() const noexcept { return tag_; }
  const Fragment& get_coords() const& noexcept { return coords_; }

  virtual ~Token() {}

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

  DomainTag tag_;
  Fragment coords_;
};

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

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

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

std::ostream& operator<<(std::ostream& os, const Token* const token);

}  // 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 kNumber:
      return "NUMBER";

    case kString:
      return "STRING";
  }
}

std::ostream& operator<<(std::ostream& os, const Token* const token) {
  os << token->get_coords() << " " << ToString(token->get_tag()) << " ";

  switch (token->get_tag()) {
    using enum DomainTag;

    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;
    }
  }

  return os;
}

}  // namespace lexer
```

Файл `compiler.hpp`

```
#pragma once

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

#include "message.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 <cstdint>
#include <regex>
#include <sstream>
#include <string>

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

namespace lexer {

namespace {

const std::string kOverflow = "integral constant is too large";
const std::string kNoOpeningQuote = "missing string opening double quote";
const std::string kNoClosingQuote = "missing string closing double quote";
const std::string kUnexpectedChar = "unexpected character";
```

```
const std::string kBreakLine = "line break is forbidden";
const std::string kBadEscape = "undefined escape sequence";

}  // namespace

std::unique_ptr<Token> Scanner::NextToken() {
  while (cur_.Cp() != kEnd) {
    while (cur_.IsWhitespace()) {
      cur_.Next();
    }

    const auto start = cur_;

    switch (cur_.Cp()) {
      case '0': {
        return std::make_unique<NumberToken>(0, start, ++cur_);
      }

      case '1': {
        std::int64_t val = 1;

        while ((++cur_).Cp() == '1') {
          ++val;
        }

        return std::make_unique<NumberToken>(val, start, cur_);
      }

      case '\"': {
        std::ostringstream oss;

        while ((++cur_).Cp() != '\"' && !cur_.IsEnd()) {
          if (cur_.Cp() == '\\') {
            switch ((++cur_).Cp()) {
              case '\"': {
                oss << '\"';
                break;
              }

              case 't': {
                oss << '\t';
                break;
              }

              case 'n': {
                oss << '\n';
```

```
          break;
        }

        default: {
          compiler_->AddMessage(MessageType::kError, cur_, kBadEscape);
        }
      }
    } else if (cur_.IsNewLine()) {
      compiler_->AddMessage(MessageType::kError, cur_, kBreakLine);
    } else {
      oss << cur_.Cp();
    }
  }

  if (cur_.IsEnd()) {
    compiler_->AddMessage(MessageType::kError, start, kNoClosingQuote);
    return std::make_unique<SpecToken>(DomainTag::kEndOfProgram, cur_);
  }

  return std::make_unique<StringToken>(oss.str(), start, ++cur_);
}

case '@': {
  std::ostringstream oss;

  if ((++cur_).Cp() != '\"') {
    compiler_->AddMessage(MessageType::kError, start, kNoOpeningQuote);
    continue;
  }

  while (true) {
    while ((++cur_).Cp() != '\"' && !cur_.IsEnd()) {
      oss << cur_.Cp();
    }

    if (cur_.IsEnd()) {
      compiler_->AddMessage(MessageType::kError, start, kNoClosingQuote);
      return std::make_unique<SpecToken>(DomainTag::kEndOfProgram, cur_);
    }

    if ((++cur_).Cp() == '\"') {
      oss << '\"';
    } else {
      break;
    }
  }
```

```
        return std::make_unique<StringToken>(oss.str(), start, cur_);
      }

      case kEnd: {
        return std::make_unique<SpecToken>(DomainTag::kEndOfProgram, cur_);
      }

      default: {
        compiler_->AddMessage(MessageType::kError, cur_++, kUnexpectedChar);
      }
    }
  }
}

}  // 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' << token << '\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

```
  111 0
   a 10
 @ @"abc""
  123""_" "123\"\n
  \k(\t)" @"a
```

Вывод на `stdout`

```
TOKENS:
    (1, 3)-(1, 6) NUMBER 3
    (1, 7)-(1, 8) NUMBER 0
    (2, 6)-(2, 7) NUMBER 1
    (2, 7)-(2, 8) NUMBER 0
    (3, 4)-(4, 10) STRING abc"
  123"_
    (4, 11)-(5, 10) STRING 123"
  ( )
    (6, 1)-(6, 1) END_OF_PROGRAM
MESSAGES:
    Error (2, 4): unexpected character
    Error (3, 2): missing string opening double quote
    Error (4, 19): line break is forbidden
    Error (5, 4): undefined escape sequence
    Error (5, 11): missing string closing double quote
```

Вывод

В результате выполнения лабораторной работы я приобрёл навыки реализации лексического анализатора на объектно-ориентированном языке без применения каких-либо средств автоматизации решения задачи лексического анализа. В основе реализации лежит модифицированный лексический анализатор из лабораторной работы 1.2, который был построен в соответствии с анализатором, описанным в лекциях. Анализатор является однопроходным — это свойство естественно реализуется в рамках индивидуального варианта.