Лабораторная работа № 1.5 «Порождение лексического анализатора с помощью flex»

20 марта 2024 г.

Илья Афанасьев, ИУ9-61Б

Цель работы

Целью данной работы является изучение генератора лексических анализаторов flex.

Индивидуальный вариант

- Химические вещества: последовательности латинских букв и цифр, начинающиеся с заглавной буквы, при этом после цифры не может следовать строчная буква (атрибут: строка). Примеры: «CuSO4», «CH3CH2OH», «Fe2O3».
- Коэффициенты: последовательности десятичных цифр. Между коэффициентом и веществом пробел может отсутствовать.
- Операторы: «+», «->».

Реализация

Файл `position.hpp`:

```
#pragma once

#include <memory>
#include <string>

namespace lexer {

struct Position final {
  std::size_t line = 1;
  std::size_t pos = 1;
  std::size_t index = 0;
```

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

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.index < rhs.index;
  }
};

}  // namespace std
```

Файл `position.cpp`:

```
#include "position.hpp"

#include <ostream>

namespace lexer {

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 {

struct Fragment final {
  Position starting;
```

```
  Position following;

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

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

}  // namespace lexer
```

Файл `fragment.cpp`:

```
#include "fragment.hpp"

#include <ostream>

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`:

```
#pragma once

#include <string>

#include "position.hpp"

namespace lexer {

enum class MessageType {
  kError,
  kWarning,
};

std::ostream& operator<<(std::ostream& os, const MessageType type);

struct Message final {
  MessageType type;
  std::string text;
```

```
};

}  // namespace lexer
```

Файл `message.cpp`:

```
#include "message.hpp"

namespace lexer {

std::ostream& operator<<(std::ostream& os, const MessageType type) {
  switch (type) {
    case MessageType::kError: {
      os << "Error";
      break;
    }

    case MessageType::kWarning: {
      os << "Warning";
      break;
    }
  }

  return os;
}

}  // namespace lexer
```

Файл `token.hpp`:

```
#pragma once

#include "fragment.hpp"

namespace lexer {

enum class DomainTag {
  kArrow,
  kEndOfProgram,
  kCoefficient,
  kPlus,
  kSubstance,
};

std::ostream& operator<<(std::ostream& os, const DomainTag tag);

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 Fragment& coords) noexcept
      : tag_(tag), coords_(coords) {}

  DomainTag tag_;
  Fragment coords_;
};

class SubstanceToken final : public Token {
 public:
  SubstanceToken(const std::string& str, const Fragment& coords) noexcept
      : Token(DomainTag::kSubstance, coords), str_(str) {}

  SubstanceToken(std::string&& str, const Fragment& coords) noexcept
      : Token(DomainTag::kSubstance, coords), str_(std::move(str)) {}

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

 private:
  std::string str_;
};

class CoefficientToken final : public Token {
 public:
  CoefficientToken(const std::int64_t value, const Fragment& coords) noexcept
      : Token(DomainTag::kCoefficient, coords), value_(value) {}

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

 private:
  std::int64_t value_;
};

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

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

```
}  // namespace lexer
```

Файл token.cpp:

```
#include "token.hpp"

#include <ostream>

namespace lexer {

std::ostream& operator<<(std::ostream& os, const DomainTag tag) {
  switch (tag) {
    case DomainTag::kArrow: {
      os << "ARROW";
      break;
    }

    case DomainTag::kEndOfProgram: {
      os << "END_OF_PROGRAM";
      break;
    }

    case DomainTag::kCoefficient: {
      os << "COEFFICIENT";
      break;
    }

    case DomainTag::kPlus: {
      os << "PLUS";
      break;
    }

    case DomainTag::kSubstance: {
      os << "SUBSTANCE";
      break;
    }
  }

  return os;
}

std::ostream& operator<<(std::ostream& os, const Token& token) {
  const auto tag = token.get_tag();
  const auto& coords = token.get_coords();

  os << coords << " " << tag;
```

```
  if (tag == DomainTag::kCoefficient) {
    const auto& coeff = static_cast<const CoefficientToken&>(token);
    os << ' ' << coeff.get_value();

  } else if (tag == DomainTag::kSubstance) {
    const auto& substance = static_cast<const SubstanceToken&>(token);
    os << ' ' << substance.get_str();
  }

  return os;
}

}  // namespace lexer
```

Файл compiler.hpp:

```
#pragma once

#include <map>

#include "message.hpp"

namespace lexer {

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

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

 private:
  std::map<Position, Message> messages_;
};

}  // namespace lexer
```

Файл compiler.cpp:

```
#include "compiler.hpp"

namespace lexer {

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

```
}

}  // namespace lexer
```

Файл ‘scanner.hpp’:

```
#pragma once

#include <variant>

#include "message.hpp"

#ifndef YY_DECL
#define YY_DECL                                                          \
  lexer::DomainTag lexer::Scanner::lex(lexer::Attribute& yylval, \
                                       lexer::Fragment& yylloc)
#endif

#ifndef yyFlexLexer
#include <FlexLexer.h>
#endif

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

namespace lexer {

using Attribute = std::variant<std::uint64_t, std::unique_ptr<std::string>>;

class Scanner : public yyFlexLexer {
 public:
  Scanner(std::shared_ptr<Compiler> compiler, std::istream& is = std::cin,
          std::ostream& os = std::cout);

  DomainTag lex(Attribute& yylval, Fragment& yylloc);

 private:
  void AdjustCoords(Fragment& yylloc) noexcept;

  DomainTag HandleSubstance(Attribute& yylval) const;
  DomainTag HandleCoefficient(Attribute& yylval) const;

  std::shared_ptr<Compiler> compiler_;
  Position cur_;
};

}  // namespace lexer
```

Файл `scanner.l`:

```
%{

#include "scanner.hpp"

#define yyterminate() return lexer::DomainTag::kEndOfProgram

#define YY_USER_ACTION AdjustCoords(yylloc);

using lexer::DomainTag;
using lexer::MessageType;

%}

%option c++
%option debug
%option noyywrap

WHITESPACE      [ \t\r\n]
LETTER          [a-zA-Z]
CAPITAL_LETTER  [A-Z]
DIGIT           [0-9]
SUBSTANCE       {CAPITAL_LETTER}{LETTER}*({CAPITAL_LETTER}{LETTER}*|{DIGIT}+)*
COEFFICIENT     {DIGIT}+
PLUS            \+
ARROW           \-\>

%%

{WHITESPACE}+  /* pass */

{SUBSTANCE}    { return HandleSubstance(yylval); }

{COEFFICIENT}  { return HandleCoefficient(yylval); }

{PLUS}         { return DomainTag::kPlus; }

{ARROW}        { return DomainTag::kArrow; }

.              { compiler_->AddMessage(MessageType::kError, yylloc.starting,
                                       "unexpected character"); }

%%

namespace lexer {
```

```
Scanner::Scanner(std::shared_ptr<Compiler> compiler, std::istream& is,
                 std::ostream& os)
    : yyFlexLexer(is, os), compiler_(std::move(compiler)) {}

void Scanner::AdjustCoords(Fragment& yylloc) noexcept {
  yylloc.starting = cur_;

  for (std::size_t i = 0; i < yyleng; ++i) {
    if (yytext[i] == '\n') {
      ++cur_.line;
      cur_.pos = 1;
    } else {
      ++cur_.pos;
    }

    ++cur_.index;
  }

  yylloc.following = cur_;
}

DomainTag Scanner::HandleSubstance(Attribute& yylval) const {
  yylval = std::make_unique<std::string>(yytext);
  return DomainTag::kSubstance;
}

DomainTag Scanner::HandleCoefficient(Attribute& yylval) const {
  yylval = std::stoull(yytext);
  return DomainTag::kCoefficient;
}

}  // namespace lexer

#ifdef yylex
#undef yylex
#endif

int yyFlexLexer::yylex() {
  return 0;
}
```

Файл `main.cpp`:

```
#include <FlexLexer.h>

#include <fstream>
```

```cpp
#include <iostream>
#include <memory>
#include <vector>

#include "scanner.hpp"

int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cerr << "Usage: lab1-5 <filename>\n";
    return 1;
  }

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

  auto compiler = std::make_shared<lexer::Compiler>();
  auto scanner = std::make_unique<lexer::Scanner>(compiler, file);

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

  lexer::DomainTag tag;
  lexer::Fragment coords;
  lexer::Attribute attr;

  do {
    tag = scanner->lex(attr, coords);

    if (tag == lexer::DomainTag::kSubstance) {
      auto& str = std::get<std::unique_ptr<std::string>>(attr);
      tokens.push_back(
          std::make_unique<lexer::SubstanceToken>(std::move(*str), coords));

    } else if (tag == lexer::DomainTag::kCoefficient) {
      auto& value = std::get<std::uint64_t>(attr);
      tokens.push_back(
          std::make_unique<lexer::CoefficientToken>(value, coords));

    } else {
      tokens.push_back(std::make_unique<lexer::SpecToken>(tag, coords));
    }
  } while (tag != lexer::DomainTag::kEndOfProgram);

  std::cout << "TOKENS:\n";
```

```
  for (const auto& token : tokens) {
    std::cout << '\t' << *token << '\n';
  }

  std::cout << "MESSAGES:\n";
  for (const auto& [pos, msg] : compiler->get_messages()) {
    std::cout << '\t' << msg.type << pos << ": " << msg.text << '\n';
  }
}
```

Тестирование

Входные данные

```
2Fe2O3+6errC -> 2 Fe + 3 CO2
```

Вывод на `stdout`

```
TOKENS:
    (1, 3)-(1, 4) COEFFICIENT 2
    (1, 4)-(1, 9) SUBSTANCE Fe2O3
    (1, 9)-(1, 10) PLUS
    (1, 10)-(1, 11) COEFFICIENT 6
    (1, 14)-(1, 15) SUBSTANCE C
    (1, 16)-(1, 18) ARROW
    (1, 19)-(1, 20) COEFFICIENT 2
    (1, 21)-(1, 23) SUBSTANCE Fe
    (1, 24)-(1, 25) PLUS
    (1, 26)-(1, 27) COEFFICIENT 3
    (1, 28)-(1, 31) SUBSTANCE CO2
    (1, 31)-(2, 1) END_OF_PROGRAM
MESSAGES:
    Error(1, 11): unexpected character
    Error(1, 12): unexpected character
    Error(1, 13): unexpected character
```

Вывод

В результате выполнения лабораторной работы я научился использовать генератор лексических анализаторов flex. Эта лабораторная работа была, наверное, самой увлекательной среди всех работ первого модуля (разве что жаль, что мне выпал крайне простой для реализации вариант). Изучив документацию мне стало интересно попробовать flex-API для языка C++: вместе с самой утилитой распространяется заголовочный файл `FlexLexer.h`, в котором определены классы, инкапсулирующие поля и методы распознавателя. Наследование от `yyFlexLexer` позволяет переопределить функцию анализа

`yylex`, добавлять собственные поля и методы для более лаконичного написания `lex`-файла. Во втором модуле было бы интересно попробовать flex в связке с bison, использующимся для автоматизации синтаксического анализа.