Лабораторная работа № 2.3 «Синтаксический анализатор на основе предсказывающего анализа»

10 апреля 2024 г.

Илья Афанасьев, ИУ9-61Б

Цель работы

Целью данной работы является изучение алгоритма построения таблиц предсказывающего анализатора.

Индивидуальный вариант

```
# ключевые слова
# начинаются с кавычки

F  -> "n" 'or "(" E ")" 'end
T  -> F T1 'end
T1 -> "*" F T1 'or 'epsilon 'end
'axiom E  -> T E1 'end
E1 -> "+" T E1 'or 'epsilon 'end
```

Реализация

Неформальное описание синтаксиса входного языка

Программа `Program` состоит из произвольного числа правил `Rule`:

```
Program ::= Rule*
```

Правило `Rule` состоит из левой `RuleLHS` и правой `RuleRHS` частей, разделяющихся оператором `OP_ARROW`:

```
Rule ::= RuleLHS OP_ARROW RuleRHS
```

Левая часть правила `RuleLHS` является нетерминалом `NONTERMINAL`. Если данное правило — аксиома грамматики, то нетерминал предваряется ключевым словом

KW_AXIOM.

```
RuleLHS ::= KW_AXIOM? NONTERMINAL
```

Правая часть RuleRHS правила содержит выражение Expr и маркер окончания правила KW_END.

```
RuleRHS ::= Expr KW_END
```

Выражение Expr состоит из альтернатив, разделямых ключевым словом KW_OR, являющихся конкатенацией символов Symbol.

```
Expr ::= Symbol+ (KW_OR Symbol+)*
```

Символ Symbol может являться терминалом TERMINAL, нетерминалом NONTERMINAL или пустым словом KW_EPSILON.

```
Symbol ::= TERMINAL | NONTERMINAL | KW_EPSILON
```

Лексическая структура

```
WHITESPACE  ::= [ \t\n\r]+
COMMENT     ::= #.*
NONTERMINAL ::= [a-zA-Z][a-zA-Z0-9]*
TERMINAL    ::= \"[^\n"]+\"
OP_ARROW    ::= ->
KW_AXIOM    ::= 'axiom
KW_EPSILON  ::= 'epsilon
KW_OR       ::= 'or
KW_END      ::= 'end
```

Грамматика языка

```
Program ::= Rules
Rules ::= Rule Rules | ε
Rule ::= RuleLHS OP_ARROW RuleRHS
RuleLHS ::= KW_AXIOM NONTERMINAL | NONTERMINAL
RuleRHS ::= Expr KW_END
Expr ::= Term Expr1
Expr1 ::= KW_OR Term Expr1 | ε
Term ::= Symbol Term1 | KW_EPSILON
Term1 ::= Symbol Term1 | ε
Symbol ::= TERMINAL | NONTERMINAL
```

Программная реализация

Файл position.h:

```
#pragma once
```

```cpp
#include <ostream>

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.cc`:

```cpp
#include "position.h"

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.h`:

```cpp
#pragma once
```

```
#include "position.h"

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.cc`:

```
#include "fragment.h"

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.h`:

```
#pragma once

#include <ostream>

namespace lexer {

enum class MessageType {
  kError,
};

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

struct Message final {
```

```
  MessageType type;
  std::string text;
};

std::ostream& operator<<(std::ostream& os, const Message& message);

}  // namespace lexer
```

Файл `message.cc`:

```
#include "message.h"

namespace lexer {

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

  return os;
}

std::ostream& operator<<(std::ostream& os, const Message& message) {
  os << message.type << " " << message.text;
  return os;
}

}  // namespace lexer
```

Файл `token.h`:

```
#pragma once

#include "fragment.h"

namespace lexer {

enum class DomainTag {
  kNonTerminal,
  kTerminal,
  kOpArrow,
  kKwAxiom,
  kKwEpsilon,
  kKwOr,
  kKwEnd,
```

```
  kEndOfProgram,
};

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

class Token {
 public:
  virtual ~Token() {}

  virtual void OutputAttr(std::ostream& os) const = 0;

  DomainTag tag() const noexcept { return tag_; }
  const Fragment& coords() const& noexcept { return coords_; }

 protected:
  Token(const DomainTag tag, const Fragment& coords) noexcept
      : tag_(tag), coords_(coords) {}

  DomainTag tag_;
  Fragment coords_;
};

class NonTerminalToken final : public Token {
  std::string str_;

 public:
  template <typename String>
  NonTerminalToken(String&& str, const Fragment& coords) noexcept
      : Token(DomainTag::kNonTerminal, coords),
        str_(std::forward<String>(str)) {}

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

  void OutputAttr(std::ostream& os) const override { os << str_; }
};

class TerminalToken final : public Token {
  std::string str_;

 public:
  template <typename String>
  TerminalToken(String&& str, const Fragment& coords) noexcept
      : Token(DomainTag::kTerminal, coords), str_(std::forward<String>(str)) {}

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

```
  void OutputAttr(std::ostream& os) const override { os << str_; }
};

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

  void OutputAttr(std::ostream&) const override {}
};

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

}  // namespace lexer
```

Файл token.cc:

```
#include "token.h"

#include <ostream>

namespace lexer {

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

    case DomainTag::kTerminal: {
      os << "TERMINAL";
      break;
    }

    case DomainTag::kOpArrow: {
      os << "OP_ARROW";
      break;
    }

    case DomainTag::kKwAxiom: {
      os << "KW_AXIOM";
      break;
    }

    case DomainTag::kKwEpsilon: {
      os << "KW_EPSILON";
```

```
      break;
    }

    case DomainTag::kKwOr: {
      os << "KW_OR";
      break;
    }

    case DomainTag::kKwEnd: {
      os << "KW_END";
      break;
    }

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

  return os;
}

std::ostream& operator<<(std::ostream& os, const Token& token) {
  os << token.coords() << " " << token.tag() << ": ";
  token.OutputAttr(os);

  return os;
}

}  // namespace lexer
```

Файл `compiler.h`:

```
#pragma once

#include <map>

#include "message.h"
#include "position.h"

namespace lexer {

class Compiler final {
 public:
  auto MessagesCbegin() const& noexcept { return messages_.cbegin(); }
  auto MessagesCend() const& noexcept { return messages_.cend(); }
```

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

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

}  // namespace lexer
```

Файл compiler.cc:

```
#include "compiler.h"

namespace lexer {

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

}  // namespace lexer
```

Файл scanner.h:

```
#pragma once

#ifndef YY_DECL
#define YY_DECL                                                    \
  lexer::DomainTag lexer::Scanner::Lex(lexer::Attribute& attr, \
                                       lexer::Fragment& coords)
#endif

#include <memory>
#include <vector>

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

#include "compiler.h"
#include "fragment.h"
#include "token.h"

namespace lexer {

using Attribute = std::unique_ptr<std::string>;

class IScanner {
```

```
 public:
  virtual std::unique_ptr<Token> NextToken() = 0;

  virtual ~IScanner() = default;
};

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

  auto CommentsCbegin() const& noexcept;
  auto CommentsCend() const& noexcept;

  std::unique_ptr<Token> NextToken() override;

 private:
  DomainTag Lex(Attribute& attr, Fragment& coords);

  void AdjustCoords(Fragment& coords) noexcept;

  DomainTag HandleNonTerminal(Attribute& attr) const;
  DomainTag HandleTerminal(Attribute& attr) const;

  std::shared_ptr<Compiler> compiler_;
  std::vector<Fragment> comments_;
  Position cur_;
};

}  // namespace lexer
```

Файл `scanner.l`:

```
%{
#include "scanner.h"

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

#define YY_USER_ACTION AdjustCoords(coords);

using lexer::DomainTag;
using lexer::MessageType;
%}

%option c++
%option debug
%option noyywrap
```

```
WHITESPACE  [ \t\r\n]
COMMENT     #.*
NONTERMINAL [A-Za-z][A-Za-z0-9]*
TERMINAL    \"[^"\n]+\"
OP_ARROW    ->
KW_AXIOM    'axiom
KW_EPSILON  'epsilon
KW_OR       'or
KW_END      'end

%%

{WHITESPACE}+  /* pass */
{NONTERMINAL}  { return HandleNonTerminal(attr); }
{TERMINAL}     { return HandleTerminal(attr); }
{OP_ARROW}     { return DomainTag::kOpArrow; }
{KW_AXIOM}     { return DomainTag::kKwAxiom; }
{KW_EPSILON}   { return DomainTag::kKwEpsilon; }
{KW_OR}        { return DomainTag::kKwOr; }
{KW_END}       { return DomainTag::kKwEnd; }
{COMMENT}      { comments_.emplace_back(coords.starting, coords.following); }
.              { compiler_->AddMessage(MessageType::kError, coords.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)) {}

auto Scanner::CommentsCbegin() const& noexcept {
  return comments_.cbegin();
}

auto Scanner::CommentsCend() const& noexcept {
  return comments_.cend();
}

std::unique_ptr<Token> Scanner::NextToken() {
  Fragment coords;
  Attribute attr;

  const auto tag = Lex(attr, coords);
```

```
  switch (tag) {
    case DomainTag::kNonTerminal: {
      return std::make_unique<NonTerminalToken>(std::move(*attr), coords);
    }

    case DomainTag::kTerminal: {
      return std::make_unique<TerminalToken>(std::move(*attr), coords);
    }

    default: {
      return std::make_unique<SpecToken>(tag, coords);
    }
  }
}

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

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

    ++cur_.index;
  }

  coords.following = cur_;
}

DomainTag Scanner::HandleNonTerminal(Attribute& attr) const {
  attr = std::make_unique<std::string>(yytext);
  return DomainTag::kNonTerminal;
}

DomainTag Scanner::HandleTerminal(Attribute& attr) const {
  attr = std::make_unique<std::string>(yytext + 1, yyleng - 2);
  return DomainTag::kTerminal;
}

}  // namespace lexer
```

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

Файл node.h:

```
#pragma once

#include <iostream>
#include <memory>
#include <vector>

#include "token.h"

namespace parser {

enum class NonTerminal {
  kProgram,
  kRules,
  kRule,
  kRuleLHS,
  kRuleRHS,
  kExpr,
  kExpr1,
  kTerm,
  kTerm1,
  kSymbol,
  kDummy,
};

std::ostream& operator<<(std::ostream& os, const NonTerminal non_terminal);

class Node {
 public:
  virtual void Output(std::ostream& os = std::cout,
                      const std::string& indent = std::string()) const = 0;

  virtual ~Node() {}
};

class InnerNode final : public Node {
 public:
  InnerNode(const NonTerminal non_terminal) noexcept
      : non_terminal_(non_terminal) {}

  std::vector<std::unique_ptr<Node>>& Children() noexcept { return children_; }
  NonTerminal non_terminal() const noexcept { return non_terminal_; }
```

```
  Node& AddChild(std::unique_ptr<Node>&& node);

  void Output(std::ostream& os, const std::string& indent) const override;

 private:
  NonTerminal non_terminal_;
  std::vector<std::unique_ptr<Node>> children_;
};

class LeafNode final : public Node {
 public:
  LeafNode(std::unique_ptr<lexer::Token>&& token) noexcept
      : token_(std::move(token)) {}

  void Output(std::ostream& os, const std::string& indent) const override;

 private:
  std::unique_ptr<lexer::Token> token_;
};

}  // namespace parser
```

Файл node.cc:

```
#include "node.h"

#include <algorithm>

namespace parser {

const auto kIndent = ".  ";

std::ostream& operator<<(std::ostream& os, const NonTerminal non_terminal) {
  switch (non_terminal) {
    case NonTerminal::kProgram: {
      os << "Program";
      break;
    }

    case NonTerminal::kRules: {
      os << "Rules";
      break;
    }

    case NonTerminal::kRule: {
      os << "Rule";
```

```
      break;
    }

    case NonTerminal::kRuleLHS: {
      os << "RuleLHS";
      break;
    }

    case NonTerminal::kRuleRHS: {
      os << "RuleRHS";
      break;
    }

    case NonTerminal::kExpr: {
      os << "Expr";
      break;
    }

    case NonTerminal::kExpr1: {
      os << "Expr1";
      break;
    }

    case NonTerminal::kTerm: {
      os << "Term";
      break;
    }

    case NonTerminal::kTerm1: {
      os << "Term1";
      break;
    }

    case NonTerminal::kSymbol: {
      os << "Symbol";
      break;
    }

    case NonTerminal::kDummy: {
      os << "Dummy";
      break;
    }
  }

  return os;
}
```

```
void InnerNode::Output(std::ostream& os, const std::string& indent) const {
  os << indent << non_terminal_ << " {\n";

  std::for_each(
      children_.cbegin(), children_.cend(),
      [&os, &indent](auto&& child) { child->Output(os, indent + kIndent); });

  os << indent << "}\n";
}

void LeafNode::Output(std::ostream& os, const std::string& indent) const {
  os << indent << token_->tag() << ": ";
  token_->OutputAttr(os);
  os << "\n";
}

Node& InnerNode::AddChild(std::unique_ptr<Node>&& node) {
  children_.push_back(std::move(node));
  return *children_.back();
}

}  // namespace parser
```

Файл `parser.h`:

```
#pragma once

// clang-format off
#include <boost/functional/hash.hpp>
// clang-format on

#include "node.h"
#include "scanner.h"
#include "token.h"

namespace parser {

using Symbol = std::variant<NonTerminal, lexer::DomainTag>;

class SententionalForm final {
 public:
  SententionalForm(std::initializer_list<Symbol> il) : symbols_(il) {}

  auto Crbegin() const& noexcept { return symbols_.crbegin(); }
  auto Crend() const& noexcept { return symbols_.crend(); }
```

```
 private:
  std::vector<Symbol> symbols_;
};

class AnalyzerTable final {
 public:
  static const AnalyzerTable& Instance();

  AnalyzerTable(const AnalyzerTable&) = delete;
  AnalyzerTable& operator=(const AnalyzerTable&) = delete;

  auto Find(const NonTerminal non_terminal, const lexer::DomainTag tag) const&;
  auto Cend() const& noexcept { return um_.cend(); }

 private:
  using Key = std::pair<NonTerminal, lexer::DomainTag>;
  using SententionalFormRef = std::reference_wrapper<const SententionalForm>;

  AnalyzerTable();

  std::vector<SententionalForm> sfs_;
  std::unordered_map<Key, SententionalFormRef, boost::hash<Key>> um_;
};

class Parser final {
 public:
  std::unique_ptr<Node> TopDownParse(lexer::IScanner& scanner);

 private:
  const AnalyzerTable& table_ = AnalyzerTable::Instance();
};

}  // namespace parser
```

Файл `parser.cc`:

```
#include "parser.h"

#include <stack>

#include "node.h"
#include "token.h"

namespace parser {

namespace {
```

```
template <typename T>
concept Printable = requires(std::ostream& os, T&& t) {
  os << t;
};

template <Printable T>
void ThrowParseError(const lexer::Token& token, T&& t) {
  std::ostringstream err;
  err << token.coords() << ": expected " << t << ", got " << token.tag();
  throw std::runtime_error(err.str());
}

}  // namespace

const AnalyzerTable& AnalyzerTable::Instance() {
  static AnalyzerTable table{};
  return table;
}

using lexer::DomainTag;

AnalyzerTable::AnalyzerTable()
    : sfs_({
          {NonTerminal::kRules},                        // 0
          {NonTerminal::kRule, NonTerminal::kRules},    // 1
          {/* ε */},                                    // 2
          {NonTerminal::kRuleLHS, DomainTag::kOpArrow,
           NonTerminal::kRuleRHS},                                          // 3
          {DomainTag::kKwAxiom, DomainTag::kNonTerminal},                   // 4
          {DomainTag::kNonTerminal},                                        // 5
          {NonTerminal::kExpr, DomainTag::kKwEnd},                          // 6
          {NonTerminal::kTerm, NonTerminal::kExpr1},                        // 7
          {DomainTag::kKwOr, NonTerminal::kTerm, NonTerminal::kExpr1},      // 8
          {NonTerminal::kSymbol, NonTerminal::kTerm1},                      // 9
          {DomainTag::kTerminal},                                           // 10
          {DomainTag::kKwEpsilon},                                          // 11
      }),
      um_({
          {{NonTerminal::kProgram, DomainTag::kNonTerminal}, sfs_[0]},
          {{NonTerminal::kProgram, DomainTag::kKwAxiom}, sfs_[0]},
          {{NonTerminal::kProgram, DomainTag::kEndOfProgram}, sfs_[0]},
          {{NonTerminal::kRules, DomainTag::kNonTerminal}, sfs_[1]},
          {{NonTerminal::kRules, DomainTag::kKwAxiom}, sfs_[1]},
          {{NonTerminal::kRules, DomainTag::kEndOfProgram}, sfs_[2]},
          {{NonTerminal::kRule, DomainTag::kNonTerminal}, sfs_[3]},
          {{NonTerminal::kRule, DomainTag::kKwAxiom}, sfs_[3]},
```

```
          {{NonTerminal::kRuleLHS, DomainTag::kNonTerminal}, sfs_[5]},
          {{NonTerminal::kRuleLHS, DomainTag::kKwAxiom}, sfs_[4]},
          {{NonTerminal::kRuleRHS, DomainTag::kNonTerminal}, sfs_[6]},
          {{NonTerminal::kRuleRHS, DomainTag::kTerminal}, sfs_[6]},
          {{NonTerminal::kRuleRHS, DomainTag::kKwEpsilon}, sfs_[6]},
          {{NonTerminal::kExpr, DomainTag::kNonTerminal}, sfs_[7]},
          {{NonTerminal::kExpr, DomainTag::kTerminal}, sfs_[7]},
          {{NonTerminal::kExpr, DomainTag::kKwEpsilon}, sfs_[7]},
          {{NonTerminal::kExpr1, DomainTag::kKwOr}, sfs_[8]},
          {{NonTerminal::kExpr1, DomainTag::kKwEnd}, sfs_[2]},
          {{NonTerminal::kTerm, DomainTag::kNonTerminal}, sfs_[9]},
          {{NonTerminal::kTerm, DomainTag::kTerminal}, sfs_[9]},
          {{NonTerminal::kTerm, DomainTag::kKwEpsilon}, sfs_[11]},
          {{NonTerminal::kTerm1, DomainTag::kNonTerminal}, sfs_[9]},
          {{NonTerminal::kTerm1, DomainTag::kTerminal}, sfs_[9]},
          {{NonTerminal::kTerm1, DomainTag::kKwOr}, sfs_[2]},
          {{NonTerminal::kTerm1, DomainTag::kKwEnd}, sfs_[2]},
          {{NonTerminal::kSymbol, DomainTag::kNonTerminal}, sfs_[5]},
          {{NonTerminal::kSymbol, DomainTag::kTerminal}, sfs_[10]},
      }) {}

auto AnalyzerTable::Find(const NonTerminal non_terminal,
                         const DomainTag tag) const& {
  return um_.find({non_terminal, tag});
}

std::unique_ptr<Node> Parser::TopDownParse(lexer::IScanner& scanner) {
  using StackItem = std::pair<Symbol, std::reference_wrapper<InnerNode>>;

  auto dummy = std::make_unique<InnerNode>(NonTerminal::kDummy);

  auto stack = std::stack<StackItem>{};
  stack.push({{DomainTag::kEndOfProgram}, *dummy});
  stack.push({{NonTerminal::kProgram}, *dummy});

  auto token = scanner.NextToken();
  const auto cend = table_.Cend();

  do {
    auto& [symbol, parent] = stack.top();

    if (auto* tag = std::get_if<DomainTag>(&symbol)) {
      if (token->tag() != *tag) {
        ThrowParseError(*token, *tag);
      }
```

```
      stack.pop();
      parent.get().AddChild(std::make_unique<LeafNode>(std::move(token)));
      token = scanner.NextToken();
    } else {
      const auto non_terminal = std::get<NonTerminal>(symbol);

      if (auto it = table_.Find(non_terminal, token->tag()); it != cend) {
        stack.pop();
        auto& child = static_cast<InnerNode&>(
            parent.get().AddChild(std::make_unique<InnerNode>(non_terminal)));

        const auto& sf = it->second.get();
        for (auto it = sf.Crbegin(), end = sf.Crend(); it != end; ++it) {
          stack.push({*it, child});
        }
      } else {
        ThrowParseError(*token, non_terminal);
      }
    }
  } while (!stack.empty());

  return std::move(dummy->Children().front());
}

}  // namespace parser
```

Файл `main.cc`:

```
#include <exception>
#include <fstream>
#include <iostream>
#include <memory>

#include "parser.h"
#include "scanner.h"

int main(int argc, char* argv[]) {
  if (argc != 2) {
    std::cerr << "Usage: lab2-3 <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 = lexer::Scanner(compiler, file);
  auto parser = parser::Parser();

  try {
    const auto root = parser.TopDownParse(scanner);
    root->Output();
  } catch (const std::exception& e) {
    std::cerr << e.what() << std::endl;
    return 1;
  }
}
```

Тестирование

Входные данные

```
# ключевые слова
# начинаются с кавычки

F  -> "n" 'or "(" E ")" 'end
T  -> F T1 'end
T1 -> "*" F T1 'or 'epsilon 'end
'axiom E  -> T E1 'end
E1 -> "+" T E1 'or 'epsilon 'end
```

Вывод на `stdout`

```
Program {
.  Rules {
.  .  Rule {
.  .  .  RuleLHS {
.  .  .  .  NONTERMINAL: F
.  .  .  }
.  .  .  OP_ARROW:
.  .  .  RuleRHS {
.  .  .  .  Expr {
.  .  .  .  .  Term {
.  .  .  .  .  .  Symbol {
.  .  .  .  .  .  .  TERMINAL: n
.  .  .  .  .  .  }
.  .  .  .  .  .  Term1 {
.  .  .  .  .  .  }
.  .  .  .  .  }
.  .  .  .  .  Expr1 {
```

```
. . . . . .  KW_OR:
. . . . . .  Term {
. . . . . . .  Symbol {
. . . . . . .  .  TERMINAL: (
. . . . . . .  }
. . . . . . .  Term1 {
. . . . . . .  .  Symbol {
. . . . . . .  .  .  NONTERMINAL: E
. . . . . . .  .  }
. . . . . . .  .  Term1 {
. . . . . . .  .  .  Symbol {
. . . . . . .  .  .  .  TERMINAL: )
. . . . . . .  .  .  }
. . . . . . .  .  .  Term1 {
. . . . . . .  .  .  }
. . . . . . .  .  }
. . . . . . .  }
. . . . . . .  Expr1 {
. . . . . . .  }
. . . . . .  }
. . . . .  }
. . . .  KW_END:
. . .  }
. .  }
. .  Rules {
. . .  Rule {
. . . .  RuleLHS {
. . . . .  NONTERMINAL: T
. . . .  }
. . . .  OP_ARROW:
. . . .  RuleRHS {
. . . . .  Expr {
. . . . . .  Term {
. . . . . .  .  Symbol {
. . . . . .  .  .  NONTERMINAL: F
. . . . . .  .  }
. . . . . .  .  Term1 {
. . . . . .  .  .  Symbol {
. . . . . .  .  .  .  NONTERMINAL: T1
. . . . . .  .  .  }
. . . . . .  .  .  Term1 {
. . . . . .  .  .  }
. . . . . .  .  }
. . . . . .  }
. . . . . .  Expr1 {
```

```
. . . . . .  }
. . . . . .  }
. . . . . .  KW_END:
. . . . .  }
. . . .  }
. . . .  Rules {
. . . . .  Rule {
. . . . . .  RuleLHS {
. . . . . . .  NONTERMINAL: T1
. . . . . .  }
. . . . . .  OP_ARROW:
. . . . . .  RuleRHS {
. . . . . . .  Expr {
. . . . . . . .  Term {
. . . . . . . . .  Symbol {
. . . . . . . . . .  TERMINAL: *
. . . . . . . . .  }
. . . . . . . . .  Term1 {
. . . . . . . . . .  Symbol {
. . . . . . . . . . .  NONTERMINAL: F
. . . . . . . . . .  }
. . . . . . . . . .  Term1 {
. . . . . . . . . . .  Symbol {
. . . . . . . . . . . .  NONTERMINAL: T1
. . . . . . . . . . .  }
. . . . . . . . . . .  Term1 {
. . . . . . . . . . .  }
. . . . . . . . . .  }
. . . . . . . . .  }
. . . . . . . .  }
. . . . . . . .  Expr1 {
. . . . . . . . .  KW_OR:
. . . . . . . . .  Term {
. . . . . . . . . .  KW_EPSILON:
. . . . . . . . .  }
. . . . . . . . .  Expr1 {
. . . . . . . . .  }
. . . . . . . .  }
. . . . . . .  }
. . . . . . .  KW_END:
. . . . . .  }
. . . . .  }
. . . . .  Rules {
. . . . . .  Rule {
. . . . . . .  RuleLHS {
. . . . . . . .  KW_AXIOM:
```

```
. . . . . .  NONTERMINAL: E
. . . . . .  }
. . . . . .  OP_ARROW:
. . . . . .  RuleRHS {
. . . . . .  Expr {
. . . . . .  Term {
. . . . . .  Symbol {
. . . . . .  NONTERMINAL: T
. . . . . .  }
. . . . . .  Term1 {
. . . . . .  Symbol {
. . . . . .  NONTERMINAL: E1
. . . . . .  }
. . . . . .  Term1 {
. . . . . .  }
. . . . . .  }
. . . . . .  }
. . . . . .  Expr1 {
. . . . . .  }
. . . . . .  }
. . . . . .  KW_END:
. . . . . .  }
. . . . . .  }
. . . . . .  Rules {
. . . . . .  Rule {
. . . . . .  RuleLHS {
. . . . . .  NONTERMINAL: E1
. . . . . .  }
. . . . . .  OP_ARROW:
. . . . . .  RuleRHS {
. . . . . .  Expr {
. . . . . .  Term {
. . . . . .  Symbol {
. . . . . .  TERMINAL: +
. . . . . .  }
. . . . . .  Term1 {
. . . . . .  Symbol {
. . . . . .  NONTERMINAL: T
. . . . . .  }
. . . . . .  Term1 {
. . . . . .  Symbol {
. . . . . .  NONTERMINAL: E1
. . . . . .  }
. . . . . .  Term1 {
. . . . . .  }
. . . . . .  }
```

```
. . . . . . . . . . }
. . . . . . . . . . }
. . . . . . . . . . Expr1 {
. . . . . . . . . .   KW_OR:
. . . . . . . . . .   Term {
. . . . . . . . . .     KW_EPSILON:
. . . . . . . . . .   }
. . . . . . . . . .   Expr1 {
. . . . . . . . . .   }
. . . . . . . . . . }
. . . . . . . . . }
. . . . . . . . .   KW_END:
. . . . . . . .   }
. . . . . . .   }
. . . . . .   Rules {
. . . . . .   }
. . . . .   }
. . . .   }
. . .   }
. .   }
.   }
}
```

Вывод

В результате выполнения лабораторной работы я закрепил приведение грамматики к виду LL(1), построение для неё FIRST и FOLLOW-множеств, а также освоил построение таблицы предсказывающего анализатора и соответствующий алгоритм предсказывающего разбора.