Лабораторная работа № 3.1
«Самоприменимый генератор компиляторов на основе предсказывающего анализа»

30 мая 2024 г.

Илья Афанасьев, ИУ9-61Б

Цель работы

Целью данной работы является изучение алгоритма построения таблиц предсказывающего анализатора.

Индивидуальный вариант

Входной язык генератора — язык представления правил грамматики, варианты лексики и синтаксиса которого можно восстановить по примеру:

```
# 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
```

Грамматика на входном языке

```
'axiom Program -> Rules 'end
Rules -> Rule Rules 'or 'epsilon 'end
Rule -> RuleLHS "ARROW" RuleRHS 'end
RuleLHS -> "KW_AXIOM" "NONTERMINAL" 'or "NONTERMINAL" 'end
RuleRHS -> Expr "KW_END" 'end
Expr -> Term Expr1 'end
Expr1 -> "KW_OR" Term Expr1 'or 'epsilon 'end
Term -> Symbol Term1 'or "KW_EPSILON" 'end
```

```
Term1 -> Symbol Term1 'or 'epsilon 'end
Symbol -> "NONTERMINAL" 'or "TERMINAL" 'end
```

Реализация

Генератор компиляторов

Файл `main.cpp`:

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

#include <boost/program_options.hpp>

#include "analyzer_table_generator.h"
#include "dt_to_ast.h"
#include "first_follow.h"
#include "parser.h"
#include "scanner.h"

namespace po = boost::program_options;

namespace {

constexpr auto kHelpOption = "help";
constexpr auto kGrammarOption = "grammar";
constexpr auto kTemplateOption = "template";
constexpr auto kTableOption = "table";

}  // namespace

int main(int ac, char* av[]) try {
  po::options_description desc("Allowed options");
  desc.add_options()
      (kHelpOption, "produce help message")
      (kGrammarOption, po::value<std::string>(), "set grammar filename")
      (kTemplateOption, po::value<std::string>(), "set template filename")
      (kTableOption, po::value<std::string>(), "set table filename")
  ;

  po::variables_map vm;
  po::store(po::parse_command_line(ac, av, desc), vm);
  po::notify(vm);
```

```
  if (vm.contains(kHelpOption)) {
    std::cout << desc << "\n";
    return 1;
  }

  std::string grammar_filename;
  if (const auto it = vm.find(kGrammarOption); it != vm.cend()) {
    grammar_filename = it->second.as<std::string>();
  } else {
    std::cerr << "Grammar filename must be set\n";
    return 1;
  }

  std::string template_filename;
  if (const auto it = vm.find(kTemplateOption); it != vm.cend()) {
    template_filename = it->second.as<std::string>();
  } else {
    template_filename = "templates/analyzer_table.cc";
  }

  std::string table_filename;
  if (const auto it = vm.find(kTableOption); it != vm.cend()) {
    table_filename = it->second.as<std::string>();
  } else {
    table_filename = "src/build/analyzer_table.cc";
  }

  std::ifstream file(grammar_filename);
  if (!file.is_open()) {
    std::cerr << "Failed to open file " << grammar_filename << "\n";
    return 1;
  }

  auto scanner = lexer::Scanner(file);
  auto parser = parser::Parser();

  const auto dt = parser.TopDownParse(scanner);
  const auto& program_node = static_cast<const parser::InnerNode&>(*dt);

  auto dt_to_ast = semantics::DtToAst{};
  const auto program = dt_to_ast.Convert(program_node);

  const auto first_follow = semantics::FirstFollow(program);
  const auto generator = semantics::AnalyzerTableGenerator(first_follow);
  generator.GenerateTable(template_filename, table_filename);
} catch (const std::exception& e) {
```

```
  std::cerr << e.what() << std::endl;
  return 1;
}
```

Лексический анализ

Файл `position.h`:

```
#pragma once

#include <string>

namespace lexer {

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

  std::string ToString() const;
};

}  // 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`:

```
#include "position.h"

namespace lexer {

std::string Position::ToString() const {
  return '(' + std::to_string(line) + ", " + std::to_string(pos) + ')';
}

}  // namespace lexer
```

Файл fragment.h:

```
#pragma once

#include "position.h"

namespace lexer {

struct Fragment final {
  Position starting;
  Position following;

  std::string ToString() const;
};

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

}  // namespace lexer
```

Файл fragment.cc:

```
#include "fragment.h"

namespace lexer {

std::string Fragment::ToString() const {
  return starting.ToString() + '-' + following.ToString();
}

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

}  // namespace lexer
```

Файл token.h:

```
#pragma once

#include "fragment.h"

namespace lexer {

enum class DomainTag {
  kNonterminal,
  kTerminal,
  kArrow,
  kKwAxiom,
```

```
  kKwEpsilon,
  kKwOr,
  kKwEnd,
  kEndOfProgram,
};

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

class Token {
 public:
  virtual ~Token() = default;

  DomainTag get_tag() const noexcept { return tag_; }
  const Fragment& get_coords() const noexcept { return coords_; }

  [[noreturn]] void ThrowError(const std::string& msg) const;

 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_; }
};

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_; }
};

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

}  // namespace lexer
```

Файл token.cc:

```
#include "token.h"

#include <stdexcept>

namespace lexer {

void Token::ThrowError(const std::string& msg) const {
  throw std::runtime_error(coords_.ToString() + ": " + msg);
}

std::string ToString(const DomainTag tag) {
  switch (tag) {
    case DomainTag::kNonterminal: {
      return "NONTERMINAL";
    }
    case DomainTag::kTerminal: {
      return "TERMINAL";
    }
    case DomainTag::kArrow: {
      return "ARROW";
    }
    case DomainTag::kKwAxiom: {
      return "KW_AXIOM";
    }
    case DomainTag::kKwEpsilon: {
      return "KW_EPSILON";
    }
    case DomainTag::kKwOr: {
      return "KW_OR";
    }
    case DomainTag::kKwEnd: {
      return "KW_END";
    }
    case DomainTag::kEndOfProgram: {
```

```
      return "END_OF_PROGRAM";
    }
  }
}

std::ostream& operator<<(std::ostream& os, const DomainTag tag) {
  return os << ToString(tag);
}

}  // 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 "fragment.h"
#include "token.h"

namespace lexer {

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

class IScanner {
 public:
  virtual ~IScanner() = default;

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

class Scanner final : private yyFlexLexer, public IScanner {
 public:
  Scanner(std::istream& is = std::cin, std::ostream& os = std::cout)
      : yyFlexLexer(is, os) {}
```

```
  auto CommentsCbegin() const noexcept { return comments_.cbegin(); }
  auto CommentsCend() const noexcept { return comments_.cend(); }

  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;

 private:
  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;
%}

%option c++ noyywrap

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

%%

{WHITESPACE}+  /* pass */
```

```
{NONTERMINAL}  { return HandleNonterminal(attr); }
{TERMINAL}     { return HandleTerminal(attr); }
{ARROW}        { return DomainTag::kArrow; }
{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); }
.              { throw std::runtime_error(
                     "unexpected symbol " + std::string{yytext}); }

%%

namespace lexer {

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;
}
```

Синтаксический анализ

Данный модуль используется и генератором компиляторов, и калькулятором.

Файл `symbol.h`:

```
#pragma once

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

namespace parser {

class Symbol final {
 public:
  enum class Type {
    kTerminal,
    kNonterminal,
    kSpecial,
  };

 public:
  Symbol(std::string name, const Type type) noexcept
      : name_(std::move(name)), type_(type) {}
```

```
  bool operator==(const Symbol&) const = default;

  const std::string& get_name() const noexcept { return name_; }
  Type get_type() const noexcept { return type_; }

 private:
  std::string name_;
  Type type_;
};

const auto kEpsilon = Symbol{"ε", Symbol::Type::kSpecial};
const auto kEndOfProgram = Symbol{"END_OF_PROGRAM", Symbol::Type::kTerminal};

std::size_t hash_value(const Symbol& symbol);

using SymbolVecIter = std::vector<Symbol>::const_iterator;
using SymbolSetIter = boost::unordered_set<Symbol>::const_iterator;

}  // namespace parser
```

Файл `symbol.cc`:

```
#include "symbol.h"

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

namespace parser {

std::size_t hash_value(const Symbol& symbol) {
  std::size_t seed = 0;
  boost::hash_combine(seed, symbol.get_type());
  boost::hash_combine(seed, symbol.get_name());
  return seed;
}

}  // namespace parser
```

Файл `analyzer_table.h`:

```
#pragma once

#include <optional>

// clang-format off
#include <boost/unordered_map.hpp>
// clang-format on
```

```cpp
#include "symbol.h"

namespace parser {

class AnalyzerTable final {
  Symbol axiom_;
  boost::unordered_map<std::pair<Symbol, Symbol>, std::vector<Symbol>> table_;

 public:
  AnalyzerTable();

  const Symbol& get_axiom() const noexcept { return axiom_; }
  std::optional<std::pair<SymbolVecIter, SymbolVecIter>> Find(
      const Symbol& nonterminal, const Symbol& terminal) const;
};

}  // namespace parser
```

Файл `parser.h`:

```cpp
#pragma once

#include "scanner.h"

namespace parser {

class INode {
 public:
  virtual ~INode() = default;
};

class InnerNode final : public INode {
  std::vector<std::unique_ptr<INode>> children_;

 public:
  INode& AddChild(std::unique_ptr<INode>&& node);

  std::vector<std::unique_ptr<INode>>& Children() noexcept { return children_; }

  auto ChildrenCbegin() const noexcept { return children_.cbegin(); }
  auto ChildrenCend() const noexcept { return children_.cend(); }
};

class LeafNode final : public INode {
  std::unique_ptr<lexer::Token> token_;
```

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

  const lexer::Token* get_token() const noexcept { return token_.get(); }
};

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

}  // namespace parser
```

Файл `parser.cc`:

```
#include "parser.h"

#include <stack>

#include "analyzer_table.h"
#include "symbol.h"
#include "token.h"

namespace parser {

namespace {

void ThrowParseError(const lexer::Token& token, const std::string& name) {
  std::ostringstream err;
  err << token.get_coords() << ": expected " << name << ", got "
      << token.get_tag();
  throw std::runtime_error(err.str());
}

}  // namespace

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

std::unique_ptr<INode> Parser::TopDownParse(lexer::IScanner& scanner) {
  const auto table = AnalyzerTable();

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

```
  auto stack = std::stack<std::pair<Symbol, InnerNode*>>{};
  stack.push({kEndOfProgram, dummy.get()});
  stack.push({{table.get_axiom()}, dummy.get()});

  auto token = scanner.NextToken();

  do {
    const auto [symbol, parent] = stack.top();
    switch (symbol.get_type()) {
      case Symbol::Type::kTerminal: {
        if (symbol.get_name() != lexer::ToString(token->get_tag())) {
          ThrowParseError(*token, symbol.get_name());
        }

        stack.pop();
        parent->AddChild(std::make_unique<LeafNode>(std::move(token)));
        token = scanner.NextToken();
        break;
      }
      case Symbol::Type::kNonterminal: {
        const auto terminal =
            Symbol{lexer::ToString(token->get_tag()), Symbol::Type::kTerminal};
        const auto opt = table.Find(symbol, terminal);
        if (!opt.has_value()) {
          ThrowParseError(*token, symbol.get_name());
        }
        const auto [b, e] = opt.value();

        stack.pop();
        auto& child = static_cast<InnerNode&>(
            parent->AddChild(std::make_unique<InnerNode>()));
        for (auto rb = std::make_reverse_iterator(e),
                  re = std::make_reverse_iterator(b);
             rb != re; ++rb) {
          stack.push({*rb, &child});
        }
        break;
      }
    }
  } while (!stack.empty());

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

}  // namespace parser
```

Семантический анализ

Файл `ast.h`:

```
#pragma once

#include "symbol.h"

namespace semantics {

class Term final {
  std::vector<parser::Symbol> symbols_;

 public:
  Term(std::vector<parser::Symbol>&& symbols) noexcept
      : symbols_(std::move(symbols)) {}

  auto SymbolsCbegin() const noexcept { return symbols_.cbegin(); }
  auto SymbolsCend() const noexcept { return symbols_.cend(); }
};

class Rule final {
  parser::Symbol lhs_;
  std::vector<std::unique_ptr<Term>> rhs_;

 public:
  Rule(parser::Symbol&& lhs, std::vector<std::unique_ptr<Term>>&& rhs) noexcept
      : lhs_(std::move(lhs)), rhs_(std::move(rhs)) {
    assert(rhs_.size() > 0);
  }

  const parser::Symbol& get_lhs() const noexcept { return lhs_; }
  auto TermsCbegin() const noexcept { return rhs_.cbegin(); }
  auto TermsCend() const noexcept { return rhs_.cend(); }
};

class Program final {
  parser::Symbol axiom_;
  std::vector<std::unique_ptr<Rule>> rules_;

 public:
  Program(parser::Symbol&& axiom,
          std::vector<std::unique_ptr<Rule>>&& rules) noexcept
      : axiom_(std::move(axiom)), rules_(std::move(rules)) {
    Validate();
  }
```

```
  const parser::Symbol& get_axiom() const noexcept { return axiom_; }
  auto RulesCbegin() const noexcept { return rules_.cbegin(); }
  auto RulesCend() const noexcept { return rules_.cend(); }

 private:
  void Validate() const;
};

}  // namespace semantics
```

Файл ast.cc:

```
#include "ast.h"

#include <algorithm>
#include <vector>

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

namespace semantics {

void Program::Validate() const {
  boost::unordered_set<parser::Symbol> defined_nonterminals,
      involved_nonterminals;
  involved_nonterminals.insert(axiom_);

  for (auto&& rule : rules_) {
    const auto [_, is_inserted] = defined_nonterminals.insert(rule->get_lhs());
    if (!is_inserted) {
      throw std::runtime_error("nonterminal " + rule->get_lhs().get_name() +
                               " redefinition");
    }
  }

  for (auto&& rule : rules_) {
    for (auto b = rule->TermsCbegin(), e = rule->TermsCend(); b != e; ++b) {
      const auto& term = **b;

      for (auto b = term.SymbolsCbegin(), e = term.SymbolsCend(); b != e; ++b) {
        if (b->get_type() != parser::Symbol::Type::kNonterminal) {
          continue;
        }

        if (!defined_nonterminals.contains(*b)) {
```

```
          throw std::runtime_error("undefined nonterminal " + b->get_name());
        }
        involved_nonterminals.insert(*b);
      }
    }
  }

  const auto is_involved =
      [&involved_nonterminals](const std::unique_ptr<Rule>& rule) {
        return involved_nonterminals.contains(rule->get_lhs());
      };

  if (const auto it =
          std::find_if_not(rules_.cbegin(), rules_.cend(), is_involved);
      it != rules_.cend()) {
    throw std::runtime_error("unused nonterminal " +
                             it->get()->get_lhs().get_name());
  }
}

}  // namespace semantics
```

Файл dt_to_ast.h:

```
#pragma once

#include "ast.h"
#include "parser.h"

namespace semantics {

class DtToAst final {
 public:
  std::shared_ptr<Program> Convert(const parser::InnerNode& program);

 private:
  std::unique_ptr<Program> ParseProgram(const parser::InnerNode& program);
  std::vector<std::unique_ptr<Rule>> ParseRules(const parser::InnerNode& rules);
  std::unique_ptr<Rule> ParseRule(const parser::InnerNode& rule);
  parser::Symbol ParseRuleLHS(const parser::InnerNode& rule_lhs);
  std::vector<std::unique_ptr<Term>> ParseRuleRHS(
      const parser::InnerNode& rule_rhs);
  std::vector<std::unique_ptr<Term>> ParseExpr(const parser::InnerNode& expr);
  std::vector<std::unique_ptr<Term>> ParseExpr1(const parser::InnerNode& expr1);
  std::unique_ptr<Term> ParseTerm(const parser::InnerNode& term);
  std::vector<parser::Symbol> ParseTerm1(const parser::InnerNode& term1);
  parser::Symbol ParseSymbol(const parser::InnerNode& symbol);
```

```
 private:
  std::unique_ptr<parser::Symbol> axiom_;
};

}  // namespace semantics
```

Файл dt_to_ast.cc:

```
#include "dt_to_ast.h"

#include <algorithm>
#include <iterator>

namespace semantics {

std::shared_ptr<Program> DtToAst::Convert(const parser::InnerNode& program) {
  axiom_ = nullptr;
  return ParseProgram(program);
}

// Program ::= Rules
std::unique_ptr<Program> DtToAst::ParseProgram(
    const parser::InnerNode& program) {
  const auto& rules =
      static_cast<const parser::InnerNode&>(**program.ChildrenCbegin());
  auto ast_rules = ParseRules(rules);

  if (!axiom_) {
    throw std::runtime_error("axiom is not defined");
  }
  return std::make_unique<Program>(std::move(*axiom_), std::move(ast_rules));
}

// Rules ::= Rule Rules | ε
std::vector<std::unique_ptr<Rule>> DtToAst::ParseRules(
    const parser::InnerNode& rules) {
  const auto b = rules.ChildrenCbegin();
  if (b == rules.ChildrenCend()) {
    return {};
  }

  const auto& rule = static_cast<const parser::InnerNode&>(**b);
  auto ast_rule = ParseRule(rule);

  const auto& rules_rhs = static_cast<const parser::InnerNode&>(**(b + 1));
  auto ast_rules = ParseRules(rules_rhs);
```

```
  ast_rules.push_back(std::move(ast_rule));
  std::rotate(ast_rules.rbegin(), ast_rules.rbegin() + 1, ast_rules.rend());
  return ast_rules;
}

// Rule ::= RuleLHS ARROW RuleRHS
std::unique_ptr<Rule> DtToAst::ParseRule(const parser::InnerNode& rule) {
  const auto b = rule.ChildrenCbegin();

  const auto& rule_lhs = static_cast<const parser::InnerNode&>(**b);
  auto lhs = ParseRuleLHS(rule_lhs);

  const auto& rule_rhs = static_cast<const parser::InnerNode&>(**(b + 2));
  auto rhs = ParseRuleRHS(rule_rhs);

  return std::make_unique<Rule>(std::move(lhs), std::move(rhs));
}

// RuleLHS ::= KW_AXIOM NONTERMINAL | NONTERMINAL
parser::Symbol DtToAst::ParseRuleLHS(const parser::InnerNode& rule_lhs) {
  const auto b = rule_lhs.ChildrenCbegin();
  if (rule_lhs.ChildrenCend() - b == 2) {
    const auto& leaf = static_cast<const parser::LeafNode&>(**(b + 1));
    const auto* const nonterminal =
        static_cast<const lexer::NonterminalToken*>(leaf.get_token());

    if (axiom_) {
      nonterminal->ThrowError("axiom redefinition");
    }
    axiom_ = std::make_unique<parser::Symbol>(
        nonterminal->get_str(), parser::Symbol::Type::kNonterminal);
    return *axiom_;
  }

  const auto& leaf = static_cast<const parser::LeafNode&>(**b);
  const auto* const nonterminal =
      static_cast<const lexer::NonterminalToken*>(leaf.get_token());
  return {nonterminal->get_str(), parser::Symbol::Type::kNonterminal};
}

// RuleRHS ::= Expr KW_END
std::vector<std::unique_ptr<Term>> DtToAst::ParseRuleRHS(
    const parser::InnerNode& rule_rhs) {
  const auto& expr =
      static_cast<const parser::InnerNode&>(**rule_rhs.ChildrenCbegin());
```

```
  return ParseExpr(expr);
}

// Expr ::= Term Expr1
std::vector<std::unique_ptr<Term>> DtToAst::ParseExpr(
    const parser::InnerNode& expr) {
  const auto b = expr.ChildrenCbegin();

  const auto& term = static_cast<const parser::InnerNode&>(**b);
  auto ast_term = ParseTerm(term);

  const auto& expr1 = static_cast<const parser::InnerNode&>(**(b + 1));
  auto ast_expr1 = ParseExpr1(expr1);

  ast_expr1.push_back(std::move(ast_term));
  std::rotate(ast_expr1.rbegin(), ast_expr1.rbegin() + 1, ast_expr1.rend());
  return ast_expr1;
}

// Expr1 ::= KW_OR Term Expr1 | ε
std::vector<std::unique_ptr<Term>> DtToAst::ParseExpr1(
    const parser::InnerNode& expr1) {
  const auto b = expr1.ChildrenCbegin();
  if (b == expr1.ChildrenCend()) {
    return {};
  }

  const auto& term = static_cast<const parser::InnerNode&>(**(b + 1));
  auto ast_term = ParseTerm(term);

  const auto& expr1_rhs = static_cast<const parser::InnerNode&>(**(b + 2));
  auto ast_expr1 = ParseExpr1(expr1_rhs);

  ast_expr1.push_back(std::move(ast_term));
  std::rotate(ast_expr1.rbegin(), ast_expr1.rbegin() + 1, ast_expr1.rend());
  return ast_expr1;
}

// Term ::= Symbol Term1 | KW_EPSILON
std::unique_ptr<Term> DtToAst::ParseTerm(const parser::InnerNode& term) {
  const auto b = term.ChildrenCbegin();
  if (term.ChildrenCend() - b == 1) {
    return std::make_unique<Term>(std::vector<parser::Symbol>{});
  }

  const auto& symbol = static_cast<const parser::InnerNode&>(**b);
```

```
  auto ast_symbol = ParseSymbol(symbol);

  const auto& term1 = static_cast<const parser::InnerNode&>(**(b + 1));
  auto ast_term1 = ParseTerm1(term1);

  ast_term1.push_back(std::move(ast_symbol));
  std::rotate(ast_term1.rbegin(), ast_term1.rbegin() + 1, ast_term1.rend());
  return std::make_unique<Term>(std::move(ast_term1));
}

// Term1 ::= Symbol Term1 | ε
std::vector<parser::Symbol> DtToAst::ParseTerm1(
    const parser::InnerNode& term1) {
  const auto b = term1.ChildrenCbegin();
  if (b == term1.ChildrenCend()) {
    return {};
  }

  const auto& symbol = static_cast<const parser::InnerNode&>(**b);
  auto ast_symbol = ParseSymbol(symbol);

  const auto& term1_rhs = static_cast<const parser::InnerNode&>(**(b + 1));
  auto ast_term1 = ParseTerm1(term1_rhs);

  ast_term1.push_back(std::move(ast_symbol));
  std::rotate(ast_term1.rbegin(), ast_term1.rbegin() + 1, ast_term1.rend());
  return ast_term1;
}

// Symbol ::= TERMINAL | NONTERMINAL
parser::Symbol DtToAst::ParseSymbol(const parser::InnerNode& symbol) {
  const auto& leaf =
      static_cast<const parser::LeafNode&>(**symbol.ChildrenCbegin());

  if (const auto* const terminal =
          dynamic_cast<const lexer::TerminalToken*>(leaf.get_token())) {
    return {terminal->get_str(), parser::Symbol::Type::kTerminal};
  }

  const auto& nonterminal =
      static_cast<const lexer::NonterminalToken&>(*leaf.get_token());
  return {nonterminal.get_str(), parser::Symbol::Type::kNonterminal};
}

}  // namespace semantics
```

Файл `first_follow.h`:

```
#pragma once

// clang-format off
#include <boost/unordered_set.hpp>
#include <boost/unordered_map.hpp>
// clang-format on

#include "ast.h"

namespace semantics {

class FirstFollow final {
 public:
  FirstFollow(std::shared_ptr<const Program> program);

  std::shared_ptr<const Program> get_program() const noexcept {
    return program_;
  }

  boost::unordered_set<parser::Symbol> GetFirstSet(
      parser::SymbolVecIter b, const parser::SymbolVecIter e) const;
  std::pair<parser::SymbolSetIter, parser::SymbolSetIter> GetFollowSet(
      const parser::Symbol& nonterminal) const;

 private:
  void BuildFirstSets();
  void BuildFollowSets();

  void PrintSets(auto&& sets) const;

 private:
  std::shared_ptr<const Program> program_;
  boost::unordered_map<parser::Symbol, boost::unordered_set<parser::Symbol>>
      first_sets_, follow_sets_;
};

}  // namespace semantics
```

Файл `first_follow.cc`:

```
#include "first_follow.h"

#include <iostream>

#include "ast.h"
```

```
namespace semantics {

FirstFollow::FirstFollow(std::shared_ptr<const Program> program)
    : program_(std::move(program)) {
  BuildFirstSets();
  BuildFollowSets();
}

void FirstFollow::BuildFirstSets() {
  bool sets_are_filling;
  do {
    sets_are_filling = false;

    for (auto b = program_->RulesCbegin(), e = program_->RulesCend(); b != e;
         ++b) {
      const auto& rule = **b;

      auto new_first_set = boost::unordered_set<parser::Symbol>{};
      for (auto b = rule.TermsCbegin(), e = rule.TermsCend(); b != e; ++b) {
        const auto& term = **b;

        new_first_set.merge(
            GetFirstSet(term.SymbolsCbegin(), term.SymbolsCend()));
      }

      auto& first_set = first_sets_[rule.get_lhs()];
      if (first_set.size() != new_first_set.size()) {
        sets_are_filling = true;
        first_set = std::move(new_first_set);
      }
    }
  } while (sets_are_filling);
}

boost::unordered_set<parser::Symbol> FirstFollow::GetFirstSet(
    parser::SymbolVecIter b, const parser::SymbolVecIter e) const {
  if (b == e) {
    return {parser::kEpsilon};
  }
  auto new_first_set = boost::unordered_set<parser::Symbol>{};

  for (const auto e_prev = e - 1; b != e; ++b) {
    if (b->get_type() == parser::Symbol::Type::kTerminal) {
      new_first_set.insert(*b);
      break;
```

```
    }

    auto first_set = boost::unordered_set<parser::Symbol>{};
    if (const auto it = first_sets_.find(*b); it != first_sets_.cend()) {
      first_set = it->second;
    }

    if (!first_set.contains(parser::kEpsilon)) {
      new_first_set.merge(std::move(first_set));
      break;
    }

    if (b != e_prev) {
      first_set.erase(parser::kEpsilon);
    }
    new_first_set.merge(std::move(first_set));
  }

  return new_first_set;
}

void FirstFollow::BuildFollowSets() {
  follow_sets_[program_->get_axiom()].insert(parser::kEndOfProgram);
  auto followed_sets =
      boost::unordered_map<parser::Symbol,
                           boost::unordered_set<parser::Symbol>>{};
  for (auto b = program_->RulesCbegin(), e = program_->RulesCend(); b != e;
       ++b) {
    const auto& rule = **b;

    for (auto b = rule.TermsCbegin(), e = rule.TermsCend(); b != e; ++b) {
      const auto& term = **b;
      if (term.SymbolsCbegin() == term.SymbolsCend()) {
        continue;
      }

      const auto e_prev = term.SymbolsCend() - 1;
      for (auto b = term.SymbolsCbegin(), e = e_prev + 1; b != e_prev; ++b) {
        if (b->get_type() != parser::Symbol::Type::kNonterminal) {
          continue;
        }

        auto first_set = GetFirstSet(b + 1, e);
        if (first_set.erase(parser::kEpsilon) && *b != rule.get_lhs()) {
          followed_sets[*b].insert(rule.get_lhs());
        }
```

```
        follow_sets_[*b].merge(std::move(first_set));
      }

      if (e_prev->get_type() == parser::Symbol::Type::kNonterminal &&
          *e_prev != rule.get_lhs()) {
        followed_sets[*e_prev].insert(rule.get_lhs());
      }
    }
  }

  bool sets_are_filling;
  do {
    sets_are_filling = false;

    for (auto&& [follower, followed_set] : followed_sets) {
      auto& follow_set = follow_sets_[follower];
      const auto initial_size = follow_set.size();

      for (auto&& nonterminal : followed_set) {
        follow_set.merge(boost::unordered_set{follow_sets_[nonterminal]});
      }

      if (follow_set.size() != initial_size) {
        sets_are_filling = true;
      }
    }
  } while (sets_are_filling);
}

std::pair<parser::SymbolSetIter, parser::SymbolSetIter>
FirstFollow::GetFollowSet(const parser::Symbol& nonterminal) const {
  const auto& follow_set = follow_sets_.at(nonterminal);
  return {follow_set.cbegin(), follow_set.cend()};
}

void FirstFollow::PrintSets(auto&& sets) const {
  for (auto&& [nonterminal, set] : sets) {
    std::cout << nonterminal.get_name() << ": ";
    for (auto&& symbol : set) {
      std::cout << symbol.get_name() << ' ';
    }
    std::cout << '\n';
  }
}

}  // namespace semantics
```

Файл analyzer_table_generator.h:

```
#pragma once

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

#include "ast.h"
#include "first_follow.h"

namespace semantics {

class AnalyzerTableGenerator final {
 public:
  using Key = std::pair<parser::Symbol, parser::Symbol>;
  using Value = std::pair<parser::SymbolVecIter, parser::SymbolVecIter>;

  AnalyzerTableGenerator(const FirstFollow& first_follow);

  void GenerateTable(const std::string& template_filename,
                     const std::string& table_filename) const;

 private:
  boost::unordered_map<Key, Value> table_;
  std::shared_ptr<const Program> program_;
};

}  // namespace semantics
```

Файл analyzer_table_generator.cc:

```
#include "analyzer_table_generator.h"

#include <fstream>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>

// clang-format off
#include <boost/format.hpp>
#include <boost/algorithm/string/join.hpp>
// clang-format on

#include "ast.h"
```

```cpp
#include "first_follow.h"

namespace semantics {

namespace {

std::string Slurp(std::ifstream& in) {
  std::ostringstream oss;
  oss << in.rdbuf();
  return oss.str();
}

std::string GetSymbolTypeDefinition(const parser::Symbol::Type type) {
  switch (type) {
    case parser::Symbol::Type::kNonterminal: {
      return "Symbol::Type::kNonterminal";
    }
    case parser::Symbol::Type::kTerminal: {
      return "Symbol::Type::kTerminal";
    }
    case parser::Symbol::Type::kSpecial: {
      return "Symbol::Type::kSpecial";
    }
  }
}

std::string GetSymbolDefinition(const parser::Symbol& symbol) {
  return boost::str(boost::format("{%1%, %2%}") %
                    std::quoted(symbol.get_name()) %
                    GetSymbolTypeDefinition(symbol.get_type()));
}

}  // namespace

AnalyzerTableGenerator::AnalyzerTableGenerator(const FirstFollow& first_follow)
    : program_(first_follow.get_program()) {
  for (auto b = program_->RulesCbegin(), e = program_->RulesCend(); b != e;
       ++b) {
    const auto& rule = **b;

    for (auto b = rule.TermsCbegin(), e = rule.TermsCend(); b != e; ++b) {
      const auto& term = **b;

      auto first_set =
          first_follow.GetFirstSet(term.SymbolsCbegin(), term.SymbolsCend());
      const auto is_epsilon_erased = first_set.erase(parser::kEpsilon);
```

```
      for (auto&& symbol : first_set) {
        const auto [_, is_inserted] =
            table_.insert({{rule.get_lhs(), symbol},
                           {term.SymbolsCbegin(), term.SymbolsCend()}});
        if (!is_inserted) {
          throw std::runtime_error("Not LL(1) grammar");
        }
      }

      if (!is_epsilon_erased) {
        continue;
      }

      for (auto [b, e] = first_follow.GetFollowSet(rule.get_lhs()); b != e;
           ++b) {
        const auto [_, is_inserted] = table_.insert(
            {{rule.get_lhs(), *b}, {term.SymbolsCbegin(), term.SymbolsCend()}});
        if (!is_inserted) {
          throw std::runtime_error("Not LL(1) grammar");
        }
      }
    }
  }
}

void AnalyzerTableGenerator::GenerateTable(
    const std::string& template_filename,
    const std::string& table_filename) const {
  auto template_file = std::ifstream(template_filename);
  if (!template_file.is_open()) {
    throw std::runtime_error("Failed to open file " + template_filename);
  }

  auto table_file = std::ofstream(table_filename);
  if (!table_file.is_open()) {
    throw std::runtime_error("Failed to create file " + table_filename);
  }

  auto records = std::vector<std::string>{};
  records.reserve(table_.size());
  for (auto&& [key, value] : table_) {
    const auto [nonterminal, symbol] = key;
    auto [b, e] = value;

    auto symbols = std::vector<std::string>{};
```

```
    symbols.reserve(e - b);
    for (; b != e; ++b) {
      symbols.push_back(GetSymbolDefinition(*b));
    }

    auto record = boost::str(boost::format("{{%1%, %2%}, {%3%}}") %
                             GetSymbolDefinition(nonterminal) %
                             GetSymbolDefinition(symbol) %
                             boost::algorithm::join(symbols, ", "));
    records.push_back(std::move(record));
  }
  const auto table_definition = boost::str(
      boost::format("{%1%}") % boost::algorithm::join(records, ", "));

  auto fmter = boost::format(Slurp(template_file));
  fmter % GetSymbolDefinition(program_->get_axiom()) % table_definition;
  table_file << fmter.str();
}

}  // namespace semantics
```

Шаблон реализации таблицы `analyzer_table.cc`:

```
#include "analyzer_table.h"

#include "ast.h"

namespace parser {

namespace ast {

AnalyzerTable::AnalyzerTable() : axiom_(%1%), table_(%2%) {}

std::optional<std::pair<SymbolVecIter, SymbolVecIter>> AnalyzerTable::Find(
    const Symbol& nonterminal, const Symbol& symbol) const {
  if (const auto it = table_.find({nonterminal, symbol}); it != table_.cend()) {
    const auto& symbols = it->second;
    return std::make_pair(symbols.cbegin(), symbols.cend());
  }
  return std::nullopt;
}

}  // namespace ast

}  // namespace parser
```

Калькулятор

Файл `main.cc`:

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

#include "node.h"
#include "parser.h"
#include "scanner.h"
#include "semantics/semantics.h"

int main(int argc, char* argv[]) try {
  if (argc != 2) {
    std::cerr << "Usage: " << argv[0] << " <filename>\n";
    return 1;
  }

  std::ifstream file(argv[1]);
  if (!file.is_open()) {
    std::cerr << "Failed to open file " << argv[1] << "\n";
    return 1;
  }

  auto scanner = lexer::Scanner(file);
  auto parser = parser::Parser();

  const auto dt = parser.TopDownParse(scanner);
  const auto& e = static_cast<const parser::InnerNode&>(*dt);
  std::cout << semantics::Interpret(e) << std::endl;
} catch (const std::exception& e) {
  std::cerr << e.what() << std::endl;
  return 1;
}
```

Лексический анализ

Файл `token.h`:

```
#pragma once

#include <cstdint>

#include "fragment.h"
```

```cpp
namespace lexer {

enum class DomainTag {
  kNumber,
  kPlus,
  kStar,
  kLeftParenthesis,
  kRightParenthesis,
  kEndOfProgram,
};

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

class Token {
 public:
  virtual ~Token() = default;

  DomainTag get_tag() const noexcept { return tag_; }
  const Fragment& get_coords() const noexcept { return coords_; }

  [[noreturn]] void ThrowError(const std::string& msg) const;

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

  DomainTag tag_;
  Fragment coords_;
};

class NumberToken final : public Token {
  std::uint64_t value_;

 public:
  NumberToken(const std::uint64_t value, const Fragment& coords) noexcept
      : Token(DomainTag::kNumber, coords), value_(value) {}

  std::uint64_t get_value() const noexcept { return value_; }
};

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

```
}  // namespace lexer
```

Файл token.cc:

```
#include "token.h"

#include <stdexcept>

namespace lexer {

void Token::ThrowError(const std::string& msg) const {
  throw std::runtime_error(coords_.ToString() + ": " + msg);
}

std::string ToString(const DomainTag tag) {
  switch (tag) {
    case DomainTag::kNumber: {
      return "n";
    }
    case DomainTag::kPlus: {
      return "+";
    }
    case DomainTag::kStar: {
      return "*";
    }
    case DomainTag::kLeftParenthesis: {
      return "(";
    }
    case DomainTag::kRightParenthesis: {
      return ")";
    }
    case DomainTag::kEndOfProgram: {
      return "END_OF_PROGRAM";
    }
  }
}

std::ostream& operator<<(std::ostream& os, const DomainTag tag) {
  return os << ToString(tag);
}

}  // namespace lexer
```

Файл scanner.h:

```
#pragma once
```

```cpp
#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 "fragment.h"
#include "token.h"

namespace lexer {

using Attribute = std::uint64_t;

class IScanner {
 public:
  virtual ~IScanner() = default;

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

class Scanner final : private yyFlexLexer, public IScanner {
 public:
  Scanner(std::istream& is = std::cin, std::ostream& os = std::cout)
      : yyFlexLexer(is, os) {}

  auto CommentsCbegin() const noexcept { return comments_.cbegin(); }
  auto CommentsCend() const noexcept { return comments_.cend(); }

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

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

  void AdjustCoords(Fragment& coords) noexcept;

  DomainTag HandleNumber(Attribute& attr) const;

 private:
  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;
%}

%option c++ noyywrap

WHITESPACE [ \t\r\n]
NUMBER     [0-9][0-9]*

%%

{WHITESPACE}+ /* pass */
{NUMBER}      { return HandleNumber(attr); }
"+"           { return DomainTag::kPlus; }
"*"           { return DomainTag::kStar; }
"("           { return DomainTag::kLeftParenthesis; }
")"           { return DomainTag::kRightParenthesis; }
.             { throw std::runtime_error("unexpected character"); }

%%

namespace lexer {

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

  const auto tag = Lex(attr, coords);
  switch (tag) {
    case DomainTag::kNumber: {
      return std::make_unique<NumberToken>(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::HandleNumber(Attribute& attr) const {
  attr = std::stoull(yytext);
  return DomainTag::kNumber;
}

}  // namespace lexer

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

Семантический анализ

Файл `semantics.h`:

```
#pragma once

#include "node.h"

namespace semantics {

std::uint64_t Interpret(const parser::InnerNode& e);

}  // namespace semantics
```

Файл semantics.cc:

```
#include "semantics.h"

#include "node.h"

namespace semantics {

namespace {

constexpr std::uint64_t kAdditionNeutral = 0;
constexpr std::uint64_t kMultiplicationNeutral = 1;

std::uint64_t ParseE1(const parser::InnerNode& e1);
std::uint64_t ParseT(const parser::InnerNode& t);
std::uint64_t ParseT1(const parser::InnerNode& t1);
std::uint64_t ParseF(const parser::InnerNode& f);

// 'axiom E -> T E1 'end
std::uint64_t ParseE(const parser::InnerNode& e) {
  const auto b = e.ChildrenCbegin();
  const auto t = ParseT(static_cast<const parser::InnerNode&>(**b));
  const auto e1 = ParseE1(static_cast<const parser::InnerNode&>(**(b + 1)));
  return t + e1;
}

// E1 -> "+" T E1 'or 'epsilon 'end
std::uint64_t ParseE1(const parser::InnerNode& e1) {
  const auto b = e1.ChildrenCbegin();
  if (b == e1.ChildrenCend()) {
    return kAdditionNeutral;
  }

  const auto t = ParseT(static_cast<const parser::InnerNode&>(**(b + 1)));
  const auto e1_rhs = ParseE1(static_cast<const parser::InnerNode&>(**(b + 2)));
  return t + e1_rhs;
}

// T -> F T1 'end
std::uint64_t ParseT(const parser::InnerNode& t) {
  const auto b = t.ChildrenCbegin();
  const auto f = ParseF(static_cast<const parser::InnerNode&>(**b));
  const auto t1 = ParseT1(static_cast<const parser::InnerNode&>(**(b + 1)));
  return f * t1;
}
```

```
// T1 -> "*" F T1 'or 'epsilon 'end
std::uint64_t ParseT1(const parser::InnerNode& t1) {
  const auto b = t1.ChildrenCbegin();
  if (b == t1.ChildrenCend()) {
    return kMultiplicationNeutral;
  }

  const auto f = ParseF(static_cast<const parser::InnerNode&>(**(b + 1)));
  const auto t1_rhs = ParseT1(static_cast<const parser::InnerNode&>(**(b + 2)));
  return f * t1_rhs;
}

// F -> "n" 'or "(" E ")" 'end
std::uint64_t ParseF(const parser::InnerNode& f) {
  const auto b = f.ChildrenCbegin();
  if (f.ChildrenCend() - b == 3) {
    return ParseE(static_cast<const parser::InnerNode&>(**(b + 1)));
  }

  const auto& leaf = static_cast<const parser::LeafNode&>(**b);
  const auto& number =
      static_cast<const lexer::NumberToken&>(*leaf.get_token());
  return number.get_value();
}

}  // namespace

std::uint64_t Interpret(const parser::InnerNode& e) { return ParseE(e); }

}  // namespace semantics
```

Тестирование

Генератор компиляторов

Сгенерированный implementation-файл для калькулятора:

```
#include "analyzer_table.h"

#include "symbol.h"

namespace parser {

AnalyzerTable::AnalyzerTable()
    : axiom_({"E", Symbol::Type::kNonterminal}),
      table_(
```

```
          {{{{"T1", Symbol::Type::kNonterminal},
             {"END_OF_PROGRAM", Symbol::Type::kTerminal}},
            {}},
           {{{"F", Symbol::Type::kNonterminal}, {"(", Symbol::Type::kTerminal}},
            {{"(", Symbol::Type::kTerminal},
             {"E", Symbol::Type::kNonterminal},
             {")", Symbol::Type::kTerminal}}},
           {{{"E1", Symbol::Type::kNonterminal},
             {")", Symbol::Type::kTerminal}},
            {}},
           {{{"E1", Symbol::Type::kNonterminal},
             {"END_OF_PROGRAM", Symbol::Type::kTerminal}},
            {}},
           {{{"E", Symbol::Type::kNonterminal}, {"(", Symbol::Type::kTerminal}},
            {{"T", Symbol::Type::kNonterminal},
             {"E1", Symbol::Type::kNonterminal}}},
           {{{"E", Symbol::Type::kNonterminal}, {"n", Symbol::Type::kTerminal}},
            {{"T", Symbol::Type::kNonterminal},
             {"E1", Symbol::Type::kNonterminal}}},
           {{{"T1", Symbol::Type::kNonterminal},
             {")", Symbol::Type::kTerminal}},
            {}},
           {{{"T1", Symbol::Type::kNonterminal},
             {"+", Symbol::Type::kTerminal}},
            {}},
           {{{"T1", Symbol::Type::kNonterminal},
             {"*", Symbol::Type::kTerminal}},
            {{"*", Symbol::Type::kTerminal},
             {"F", Symbol::Type::kNonterminal},
             {"T1", Symbol::Type::kNonterminal}}},
           {{{"T", Symbol::Type::kNonterminal}, {"n", Symbol::Type::kTerminal}},
            {{"F", Symbol::Type::kNonterminal},
             {"T1", Symbol::Type::kNonterminal}}},
           {{{"F", Symbol::Type::kNonterminal}, {"n", Symbol::Type::kTerminal}},
            {{"n", Symbol::Type::kTerminal}}},
           {{{"E1", Symbol::Type::kNonterminal},
             {"+", Symbol::Type::kTerminal}},
            {{"+", Symbol::Type::kTerminal},
             {"T", Symbol::Type::kNonterminal},
             {"E1", Symbol::Type::kNonterminal}}},
           {{{"T", Symbol::Type::kNonterminal}, {"(", Symbol::Type::kTerminal}},
            {{"F", Symbol::Type::kNonterminal},
             {"T1", Symbol::Type::kNonterminal}}}}}) {}

std::optional<std::pair<SymbolVecIter, SymbolVecIter>> AnalyzerTable::Find(
    const Symbol& nonterminal, const Symbol& terminal) const {
```

```
  if (const auto it = table_.find({nonterminal, terminal});
      it != table_.cend()) {
    const auto& symbols = it->second;
    return std::make_pair(symbols.cbegin(), symbols.cend());
  }
  return std::nullopt;
}

}  // namespace parser
```

Сгенерированный implementation-файл для собственной грамматики:

```
#include "analyzer_table.h"

#include "symbol.h"

namespace parser {

AnalyzerTable::AnalyzerTable()
    : axiom_({"Program", Symbol::Type::kNonterminal}),
      table_({{{{"Expr", Symbol::Type::kNonterminal},
                {"KW_EPSILON", Symbol::Type::kTerminal}},
               {{"Term", Symbol::Type::kNonterminal},
                {"Expr1", Symbol::Type::kNonterminal}}},
              {{{"Rule", Symbol::Type::kNonterminal},
                {"KW_AXIOM", Symbol::Type::kTerminal}},
               {{"RuleLHS", Symbol::Type::kNonterminal},
                {"ARROW", Symbol::Type::kTerminal},
                {"RuleRHS", Symbol::Type::kNonterminal}}},
              {{{"Rules", Symbol::Type::kNonterminal},
                {"KW_AXIOM", Symbol::Type::kTerminal}},
               {{"Rule", Symbol::Type::kNonterminal},
                {"Rules", Symbol::Type::kNonterminal}}},
              {{{"Term", Symbol::Type::kNonterminal},
                {"TERMINAL", Symbol::Type::kTerminal}},
               {{"Symbol", Symbol::Type::kNonterminal},
                {"Term1", Symbol::Type::kNonterminal}}},
              {{{"Expr", Symbol::Type::kNonterminal},
                {"NONTERMINAL", Symbol::Type::kTerminal}},
               {{"Term", Symbol::Type::kNonterminal},
                {"Expr1", Symbol::Type::kNonterminal}}},
              {{{"Expr1", Symbol::Type::kNonterminal},
                {"KW_OR", Symbol::Type::kTerminal}},
               {{"KW_OR", Symbol::Type::kTerminal},
                {"Term", Symbol::Type::kNonterminal},
                {"Expr1", Symbol::Type::kNonterminal}}},
              {{{"Rules", Symbol::Type::kNonterminal},
```

```
  {"NONTERMINAL", Symbol::Type::kTerminal}},
 {{"Rule", Symbol::Type::kNonterminal},
  {"Rules", Symbol::Type::kNonterminal}}},
{{{"Rule", Symbol::Type::kNonterminal},
  {"NONTERMINAL", Symbol::Type::kTerminal}},
 {{"RuleLHS", Symbol::Type::kNonterminal},
  {"ARROW", Symbol::Type::kTerminal},
  {"RuleRHS", Symbol::Type::kNonterminal}}},
{{{"Term1", Symbol::Type::kNonterminal},
  {"KW_OR", Symbol::Type::kTerminal}},
 {}},
{{{"Term1", Symbol::Type::kNonterminal},
  {"TERMINAL", Symbol::Type::kTerminal}},
 {{"Symbol", Symbol::Type::kNonterminal},
  {"Term1", Symbol::Type::kNonterminal}}},
{{{"Expr", Symbol::Type::kNonterminal},
  {"TERMINAL", Symbol::Type::kTerminal}},
 {{"Term", Symbol::Type::kNonterminal},
  {"Expr1", Symbol::Type::kNonterminal}}},
{{{"Term1", Symbol::Type::kNonterminal},
  {"KW_END", Symbol::Type::kTerminal}},
 {}},
{{{"RuleLHS", Symbol::Type::kNonterminal},
  {"KW_AXIOM", Symbol::Type::kTerminal}},
 {{"KW_AXIOM", Symbol::Type::kTerminal},
  {"NONTERMINAL", Symbol::Type::kTerminal}}},
{{{"Expr1", Symbol::Type::kNonterminal},
  {"KW_END", Symbol::Type::kTerminal}},
 {}},
{{{"Program", Symbol::Type::kNonterminal},
  {"KW_AXIOM", Symbol::Type::kTerminal}},
 {{"Rules", Symbol::Type::kNonterminal}}},
{{{"Symbol", Symbol::Type::kNonterminal},
  {"TERMINAL", Symbol::Type::kTerminal}},
 {{"TERMINAL", Symbol::Type::kTerminal}}},
{{{"Program", Symbol::Type::kNonterminal},
  {"NONTERMINAL", Symbol::Type::kTerminal}},
 {{"Rules", Symbol::Type::kNonterminal}}},
{{{"RuleRHS", Symbol::Type::kNonterminal},
  {"KW_EPSILON", Symbol::Type::kTerminal}},
 {{"Expr", Symbol::Type::kNonterminal},
  {"KW_END", Symbol::Type::kTerminal}}},
{{{"RuleRHS", Symbol::Type::kNonterminal},
  {"NONTERMINAL", Symbol::Type::kTerminal}},
 {{"Expr", Symbol::Type::kNonterminal},
  {"KW_END", Symbol::Type::kTerminal}}},
```

```
                {{{"RuleRHS", Symbol::Type::kNonterminal},
                  {"TERMINAL", Symbol::Type::kTerminal}},
                 {{"Expr", Symbol::Type::kNonterminal},
                  {"KW_END", Symbol::Type::kTerminal}}},
                {{{"Term1", Symbol::Type::kNonterminal},
                  {"NONTERMINAL", Symbol::Type::kTerminal}},
                 {{"Symbol", Symbol::Type::kNonterminal},
                  {"Term1", Symbol::Type::kNonterminal}}},
                {{{"Symbol", Symbol::Type::kNonterminal},
                  {"NONTERMINAL", Symbol::Type::kTerminal}},
                 {{"NONTERMINAL", Symbol::Type::kTerminal}}},
                {{{"Term", Symbol::Type::kNonterminal},
                  {"NONTERMINAL", Symbol::Type::kTerminal}},
                 {{"Symbol", Symbol::Type::kNonterminal},
                  {"Term1", Symbol::Type::kNonterminal}}},
                {{{"RuleLHS", Symbol::Type::kNonterminal},
                  {"NONTERMINAL", Symbol::Type::kTerminal}},
                 {{"NONTERMINAL", Symbol::Type::kTerminal}}},
                {{{"Term", Symbol::Type::kNonterminal},
                  {"KW_EPSILON", Symbol::Type::kTerminal}},
                 {{"KW_EPSILON", Symbol::Type::kTerminal}}},
                {{{"Rules", Symbol::Type::kNonterminal},
                  {"END_OF_PROGRAM", Symbol::Type::kTerminal}},
                 {}},
                {{{"Program", Symbol::Type::kNonterminal},
                  {"END_OF_PROGRAM", Symbol::Type::kTerminal}},
                 {{"Rules", Symbol::Type::kNonterminal}}}}}) {}

std::optional<std::pair<SymbolVecIter, SymbolVecIter>> AnalyzerTable::Find(
    const Symbol& nonterminal, const Symbol& terminal) const {
  if (const auto it = table_.find({nonterminal, terminal});
      it != table_.cend()) {
    const auto& symbols = it->second;
    return std::make_pair(symbols.cbegin(), symbols.cend());
  }
  return std::nullopt;
}

}  // namespace parser
```

Калькулятор

Входные данные:

```
1 + 2 * (3 + 4)
```

Вывод:

Вывод

В результате выполнения лабораторной работы я изучил алгоритм построения таблиц предсказывающего анализатора. Реализация самоприменимого генератора оказалась нетривиальной и довольно интересной задачей, поскольку в работе было особенно важно хорошо определить представление данных и грамотно связать модули.

Резюмируя итоговое решение: дерево вывода, строящееся предсказывающим анализатором, конвертируется в абстрактное синтаксические дерево по упрощённому алгоритму рекурсивного спуска ("упрощённому", поскольку дерево вывода гарантированно корректно, и многие проверки опущены). Над построенным AST проводится семантический анализ (проверки на отсутствие неизвестных, дублирующихся, неиспользованных нетерминалов, единственность аксиомы грамматики и т.п.). По AST генерируются FIRST- и FOLLOW-множества по известному алгоритму; наполнение множеств также происходит "до насыщения". Наконец, по множествам строится таблица генератора как программная сущность, которая затем конвертируется в литерал языка. Если в ячейке таблицы оказывается более одной правой части правил грамматики, рабочий язык не является LL(1). Генератор фактически порождает implementation-файл `analyzer_table.cc` для класса `AnalyzerTable`, определённого в `analyzer_table.h`, который теперь используется исходным предсказывающим анализатором.

В текущем решении мне не нравится представление терминалов и нетерминалов генерируемой табилицы строковыми литералами. Кажется, было бы оптимальнее породить соответствующий `enum class`, и его экземпляры уже хранить в таблице. Также в работе оказалось удобным использование некоторых компонентов библиотеки Boost: опции программы, инструменты форматирования, алгоритмы работы со строками, а также Boost хеш-таблицы, предлагающие удобный механизм хеширования сущностей.