Лабораторная работа № 2.4 «Рекурсивный спуск»

24 апреля 2024 г.

Илья Афанасьев, ИУ9-61Б

Цель работы

Целью данной работы является изучение алгоритмов построения парсеров методом рекурсивного спуска.

Индивидуальный вариант

Статически типизированный функциональный язык программирования с сопоставлением с образцом:

```
@ Объединение двух списков
zip (*int, *int) :: *(int, int) is
  (x : xs, y : ys) = (x, y) : zip (xs, ys);
  (xs, ys) = {}
end

@ Декартово произведение
cart_prod (*int, *int) :: *(int, int) is
  (x : xs, ys) = append (bind (x, ys), cart_prod(xs, ys));
  ({}, ys) = {}
end

bind (int, *int) :: *(int, int) is
  (x, {}) = {};
  (x, y : ys) = (x, y) : bind (x, ys)
end

@ Конкатенация списков пар
append (*(int, int), *(int, int)) :: *(int, int) is
  (x : xs, ys) = x : append (xs, ys);
  ({}, ys) = ys
end
```

```
@ Расплющивание вложенного списка
flat **int :: *int is
  [x : xs] : xss = x : flat [xs : xss];
  {} : xss = flat xss;
  {} = {}
end

@ Сумма элементов списка
sum *int :: int is
  x : xs = x + sum xs;
  {} = 0
end

@ Вычисление полинома по схеме Горнера
polynom (int, *int) :: int is
  (x, {}) = 0;
  (x, coef : coefs) = polynom (x, coefs) * x + coef
end

@ Вычисление полинома x³+x²+x+1
polynom1111 int :: int is x = polynom (x, {1, 1, 1, 1}) end
```

Комментарии начинаются на знак @.

Все функции в рассматриваемом языке являются функциями одного аргумента. Когда нужно вызвать функцию с несколькими аргументами, они передаются в виде кортежа.

Круглые скобки служат для создания кортежа, фигурные — для создания списка, квадратные — для указания приоритета.

Наивысший приоритет имеет операция вызова функции. Вызов функции правоассоциативен, т.е. выражение x y z трактуется как x [y z] (аргументом функции y является z, аргументом функции x — выражение y z.

За вызовом функции следуют арифметические операции *, /, +, - с обычным приоритетом (у * и / он выше, чем у + и -) и ассоциативностью (левая).

Наинизшим приоритетом обладает операция создания cons-ячейки :, ассоциативность — правая (т.е. x : y : z трактуется как x : [y : z]).

Функция состоит из заголовка, в котором указывается её тип, и тела, содержащего несколько предложений. Предложения разделяются знаком ;.

Предложение состоит из образца и выражения, разделяемых знаком =. В образце, в отличие от выражения, недопустимы арифметические операции и вызовы функций.

Тип списка описывается с помощью одноместной операции *, предваряющей тип, тип кортежа — как перечисление типов элементов через запятую в круглых скобках.

Реализация

Лексическая структура

```
WHITESPACE ::= [ \t\r\n]
COMMENT ::= @.*
PLUS ::= +
MINUS ::= -
STAR ::= \*
SLASH ::= /
EQUAL ::= =
COMMA ::= ,
COLON ::= :
COLON_COLON ::= ::
SEMICOLON ::= ;
PARENTHESIS_LEFT ::= (
PARENTHESIS_RIGHT ::= )
CURLY_BRACKET_LEFT ::= {
CURLY_BRACKET_RIGHT ::= }
SQUARE_BRACKET_LEFT ::= [
SQUARE_BRACKET_RIGHT ::= ]
INT ::= int
IS ::= is
END ::= end
IDENT ::= [A-Za-z_][A-Za-z_0-9]*
INT_CONST ::= [0-9]+
```

Грамматика языка

```
Program ::= Func*.
Func ::= Ident FuncType 'is' FuncBody 'end'.

FuncType ::= Type '::' Type.
Type ::= ElementaryType | ListType | TupleType.
ElementaryType ::= 'int'.
ListType ::= '*' Type.
TupleType ::= '(' (Type (',' Type)*)? ')'.

FuncBody ::= Sentence (';' Sentence)*.
Sentence ::= Pattern '=' Result.

Pattern ::= PatternUnit (':' Pattern)?.
```

```
PatternUnit ::= Ident | Const | PatternList | PatternTuple | '[' Pattern ']'.
Const ::= IntConst.
PatternList ::= '{' (Pattern (',' Pattern)*)? '}'.
PatternTuple ::= '(' (Pattern (',' Pattern)*)? ')'.

Result ::= ResultUnit (':' Result)?.
ResultUnit ::= Expr | ResultList | ResultTuple.
Expr ::= Term ('+' Term | '-' Term)*.
Term ::= Factor ('*' Factor | '/' Factor)*.
Factor ::= Atom | '[' Expr ']'.
Atom ::= Const | Ident FuncArg?.
FuncArg ::= Atom | ResultList | ResultTuple | '[' Result ']'.
ResultList ::= '{' (Result (',' Result)*)? '}'.
ResultTuple ::= '(' (Result (',' Result)*)? ')'.
```

Программная реализация

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: lab2-4 <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);
  auto parser = parser::Parser(std::move(scanner));

  try {
    const auto root = parser.RecursiveDescentParse();
    std::cout << boost::json::serialize(root->ToJson()) << "\n";
  } catch (const std::exception& e) {
    std::cerr << e.what() << std::endl;
    return 1;
  }
}
```

Лексический анализ

Файл `position.h`:

```
#pragma once

#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`:

```
#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`:

```
#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 {
  kPlus,
  kMinus,
  kStar,
  kSlash,
  kEqual,
  kComma,
  kColon,
  kColonColon,
  kSemicolon,
  kParanthesisLeft,
  kParanthesisRight,
  kCurlyBracketLeft,
  kCurlyBracketRight,
  kSquareBracketLeft,
  kSquareBracketRight,
  kInt,
  kIs,
  kEnd,
  kIdent,
  kIntConst,
  kEndOfProgram,
};

std::string ToString(const DomainTag tag);

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

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

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

  DomainTag tag_;
  Fragment coords_;
};

class IdentToken final : public Token {
  std::size_t code_;

 public:
  IdentToken(const std::size_t code, const Fragment& coords) noexcept
      : Token(DomainTag::kIdent, coords), code_(code) {}

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

class IntConstToken final : public Token {
  std::int64_t value_;

 public:
  IntConstToken(const std::int64_t value, const Fragment& coords) noexcept
      : Token(DomainTag::kIntConst, coords), value_(value) {}

  std::int64_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 <ostream>

namespace lexer {

std::string ToString(const DomainTag tag) {
  switch (tag) {
    case DomainTag::kPlus: {
      return "PLUS";
    }
    case DomainTag::kMinus: {
      return "MINUS";
    }
    case DomainTag::kStar: {
      return "STAR";
    }
    case DomainTag::kSlash: {
```

```
  return "SLASH";
}
case DomainTag::kEqual: {
  return "EQUAL";
}
case DomainTag::kComma: {
  return "COMMA";
}
case DomainTag::kColon: {
  return "COLON";
}
case DomainTag::kColonColon: {
  return "COLON_COLON";
}
case DomainTag::kSemicolon: {
  return "SEMICOLON";
}
case DomainTag::kParanthesisLeft: {
  return "PARENTHESIS_LEFT";
}
case DomainTag::kParanthesisRight: {
  return "PARENTHESIS_RIGHT";
}
case DomainTag::kCurlyBracketLeft: {
  return "CURLY_BRACKET_LEFT";
}
case DomainTag::kCurlyBracketRight: {
  return "CURLY_BRACKET_RIGHT";
}
case DomainTag::kSquareBracketLeft: {
  return "SQUARE_BRACKET_LEFT";
}
case DomainTag::kSquareBracketRight: {
  return "SQUARE_BRACKET_RIGHT";
}
case DomainTag::kInt: {
  return "INT";
}
case DomainTag::kIs: {
  return "IS";
}
case DomainTag::kEnd: {
  return "END";
}
case DomainTag::kIdent: {
  return "IDENT";
```

```
    }
    case DomainTag::kIntConst: {
      return "INT_CONST";
    }
    case DomainTag::kEndOfProgram: {
      return "END_OF_PROGRAM";
    }
  }
}

}  // namespace lexer
```

Файл compiler.h:

```
#pragma once

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

#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(); }

  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_;
};

}  // namespace lexer
```

Файл compiler.cc:

```
#include "compiler.h"
```

```
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_[code];
}

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 <variant>
#include <vector>

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

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

namespace lexer {
```

```
using Attribute = std::variant<std::size_t, std::int64_t>;

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)
      : yyFlexLexer(is, os), compiler_(std::move(compiler)) {}

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

  void SetDebug(const bool is_active) { set_debug(is_active); }

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

    const auto tag = Lex(attr, coords);

    switch (tag) {
      case DomainTag::kIdent: {
        return std::make_unique<IdentToken>(std::get<std::size_t>(attr),
                                            coords);
      }

      case DomainTag::kIntConst: {
        return std::make_unique<IntConstToken>(std::get<std::int64_t>(attr),
                                               coords);
      }

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

 private:
  DomainTag Lex(Attribute& attr, Fragment& coords);
  void 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 HandleIdent(Attribute& attr) const {
    attr = compiler_->AddName(yytext);
    return DomainTag::kIdent;
  }

  DomainTag HandleIntConst(Attribute& attr) const {
    attr = std::stoll(yytext);
    return DomainTag::kIntConst;
  }

  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

IDENT      [A-Za-z_][A-Za-z_0-9]*
INT_CONST  [0-9]+

%%

[ \t\r\n]+   /* pass */
\+           { return DomainTag::kPlus; }
\-           { return DomainTag::kMinus; }
\*           { return DomainTag::kStar; }
\/           { return DomainTag::kSlash; }
=            { return DomainTag::kEqual; }
,            { return DomainTag::kComma; }
:            { return DomainTag::kColon; }
::           { return DomainTag::kColonColon; }
;            { return DomainTag::kSemicolon; }
\(           { return DomainTag::kParanthesisLeft; }
\)           { return DomainTag::kParanthesisRight; }
\{           { return DomainTag::kCurlyBracketLeft; }
\}           { return DomainTag::kCurlyBracketRight; }
\[           { return DomainTag::kSquareBracketLeft; }
\]           { return DomainTag::kSquareBracketRight; }
int          { return DomainTag::kInt; }
is           { return DomainTag::kIs; }
end          { return DomainTag::kEnd; }
{IDENT}      { return HandleIdent(attr); }
{INT_CONST}  { return HandleIntConst(attr); }
@.*          { comments_.emplace_back(coords.starting, coords.following); }
.            { compiler_->AddMessage(MessageType::kError, coords.starting,
                                     "unexpected character"); }

%%

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

Синтаксический анализ

Файл node.h:

```
#pragma once

#include <iterator>
#include <memory>
#include <vector>

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

#include "token.h"

namespace parser {

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

  virtual boost::json::value ToJson() const = 0;
};

namespace ast {

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

class PatternBinary final : public Pattern {
  std::unique_ptr<Pattern> lhs_, rhs_;
  lexer::DomainTag op_;

 public:
  PatternBinary(std::unique_ptr<Pattern>&& lhs, std::unique_ptr<Pattern>&& rhs,
                const lexer::DomainTag op)
      : lhs_(std::move(lhs)), rhs_(std::move(rhs)), op_(op) {}

  boost::json::value ToJson() const override;
};

class PatternTuple final : public Pattern {
  std::vector<std::unique_ptr<Pattern>> patterns_;

 public:
  using PatternsIterator = decltype(patterns_)::iterator;
  PatternTuple(const std::move_iterator<PatternsIterator> begin,
               const std::move_iterator<PatternsIterator> end)
      : patterns_(begin, end) {}

  boost::json::value ToJson() const override;
};

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

class ResultBinary final : public Result {
  std::unique_ptr<Result> lhs_, rhs_;
  lexer::DomainTag op_;

 public:
  ResultBinary(std::unique_ptr<Result>&& lhs, std::unique_ptr<Result>&& rhs,
               const lexer::DomainTag op)
      : lhs_(std::move(lhs)), rhs_(std::move(rhs)), op_(op) {}

  boost::json::value ToJson() const override;
};

class ResultTuple final : public Result {
  std::vector<std::unique_ptr<Result>> results_;

 public:
  using ResultsIterator = decltype(results_)::iterator;

  ResultTuple(const std::move_iterator<ResultsIterator> begin,
              const std::move_iterator<ResultsIterator> end)
      : results_(begin, end) {}

  boost::json::value ToJson() const override;
};

class EmptyList final : public Pattern, public Result {
 public:
  EmptyList() = default;

  boost::json::value ToJson() const override;
};

class Var final : public Pattern, public Result {
  std::size_t ident_code_;
 public:
  Var(const std::size_t ident_code) : ident_code_(ident_code) {}

  boost::json::value ToJson() const override;
};

template <typename Value>
class Const final : public Pattern, public Result {
  Value value_;
  lexer::DomainTag tag_;

 public:
  Const(Value&& value, const lexer::DomainTag tag)
      : value_(std::forward<Value>(value)), tag_(tag) {}

  boost::json::value ToJson() const override;
};

class FuncCall final : public Result {
  std::unique_ptr<Result> arg_;
  std::size_t ident_code_;

 public:
  FuncCall(std::unique_ptr<Result>&& arg, const std::size_t ident_code)
      : arg_(std::move(arg)), ident_code_(ident_code) {}

  boost::json::value ToJson() const override;
};

class Sentence final : public JsonSerializible {
  std::unique_ptr<Pattern> pattern_;
  std::unique_ptr<Result> result_;

 public:
  Sentence(std::unique_ptr<Pattern>&& pattern, std::unique_ptr<Result>&& result)
      : pattern_(std::move(pattern)), result_(std::move(result)) {}

  boost::json::value ToJson() const override;
};

class FuncBody final : public JsonSerializible {
  std::vector<std::unique_ptr<Sentence>> sents_;

 public:
  using SentsIterator = decltype(sents_)::iterator;
  FuncBody(const std::move_iterator<SentsIterator> begin,
           const std::move_iterator<SentsIterator> end)
      : sents_(begin, end) {}

  boost::json::value ToJson() const override;
};

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

class ListType final : public Type {
  std::unique_ptr<Type> type_;

 public:
  ListType(std::unique_ptr<Type>&& type) : type_(std::move(type)) {}

  boost::json::value ToJson() const override;
};

class TupleType final : public Type {
  std::vector<std::unique_ptr<Type>> types_;

 public:
  using TypesIterator = decltype(types_)::iterator;

  TupleType(const std::move_iterator<TypesIterator> begin,
            const std::move_iterator<TypesIterator> end)
      : types_(begin, end) {}

  boost::json::value ToJson() const override;
};

class ElementaryType final : public Type {
  lexer::DomainTag tag_;

 public:
  ElementaryType(const lexer::DomainTag type) : tag_(type) {}

  boost::json::value ToJson() const override;
};

class FuncType final : public JsonSerializible {
  std::unique_ptr<Type> input_;
  std::unique_ptr<Type> output_;

 public:
  FuncType(std::unique_ptr<Type>&& input, std::unique_ptr<Type>&& output)
      : input_(std::move(input)), output_(std::move(output)) {}

  boost::json::value ToJson() const override;
};

class Func final : public JsonSerializible {
  std::unique_ptr<FuncType> type_;
  std::unique_ptr<FuncBody> body_;
  std::size_t ident_code_;

 public:
  Func(std::unique_ptr<FuncType>&& type, std::unique_ptr<FuncBody>&& body,
       const std::size_t ident_code)
      : type_(std::move(type)),
        body_(std::move(body)),
        ident_code_(ident_code) {}

  boost::json::value ToJson() const override;
};

class Program final : public JsonSerializible {
  std::vector<std::unique_ptr<Func>> funcs_;

 public:
  using FuncsIterator = decltype(funcs_)::iterator;

  Program(const std::move_iterator<FuncsIterator> begin,
          const std::move_iterator<FuncsIterator> end)
      : funcs_(begin, end) {}

  boost::json::value ToJson() const override;
};

}  // namespace ast

}  // namespace parser
```

Файл node.cc:

```
#include "node.h"

#include "token.h"
```

```
namespace parser {

namespace ast {

static constexpr std::string_view kDiscriminatorType = "discriminator_type";

boost::json::value Program::ToJson() const {
  auto program = boost::json::object{};

  auto& funcs = (program["funcs"] = boost::json::array{}).as_array();
  funcs.reserve(funcs_.size());

  for (auto&& func : funcs_) {
    funcs.push_back(func->ToJson());
  }

  return program;
}

boost::json::value Func::ToJson() const {
  return {
      {"ident_code", ident_code_},
      {"type", type_->ToJson()},
      {"body", body_->ToJson()},
  };
}

boost::json::value FuncType::ToJson() const {
  return {
      {"input", input_->ToJson()},
      {"output", output_->ToJson()},
  };
}

boost::json::value ElementaryType::ToJson() const {
  return {
      {kDiscriminatorType, "elementary_type"},
      {"tag", lexer::ToString(tag_)},
  };
}

boost::json::value ListType::ToJson() const {
  return {
      {kDiscriminatorType, "list_type"},
      {"type", type_->ToJson()},
  };
```

```
}

boost::json::value TupleType::ToJson() const {
  auto tuple = boost::json::object{};
  tuple[kDiscriminatorType] = "tuple_type";

  auto& types = (tuple["types"] = boost::json::array{}).as_array();
  types.reserve(types_.size());

  for (auto&& type : types_) {
    types.push_back(type->ToJson());
  }

  return tuple;
}

boost::json::value FuncBody::ToJson() const {
  auto func_body = boost::json::object{};

  auto& sents = (func_body["sents"] = boost::json::array{}).as_array();
  sents.reserve(sents_.size());

  for (auto&& sent : sents_) {
    sents.push_back(sent->ToJson());
  }

  return func_body;
}

boost::json::value Sentence::ToJson() const {
  return {
      {"pattern", pattern_->ToJson()},
      {"result", result_->ToJson()},
  };
}

boost::json::value PatternBinary::ToJson() const {
  return {
      {kDiscriminatorType, "pattern_binary"},
      {"op", lexer::ToString(op_)},
      {"lhs", lhs_->ToJson()},
      {"rhs", rhs_->ToJson()},
  };
}

boost::json::value PatternTuple::ToJson() const {
  auto pattern_tuple = boost::json::object{};
  pattern_tuple[kDiscriminatorType] = "pattern_tuple";

  auto& patterns =
      (pattern_tuple["patterns"] = boost::json::array{}).as_array();
  patterns.reserve(patterns_.size());

  for (auto&& pattern : patterns_) {
    patterns.push_back(pattern->ToJson());
  }

  return pattern_tuple;
}

boost::json::value EmptyList::ToJson() const {
  return {
      {kDiscriminatorType, "empty_list"},
  };
}

boost::json::value Var::ToJson() const {
  return {
      {kDiscriminatorType, "var"},
      {"ident_code", ident_code_},
  };
}

template <>
boost::json::value Const<std::int64_t>::ToJson() const {
  return {
      {kDiscriminatorType, "int_const"},
      {"value", value_},
  };
}

boost::json::value ResultBinary::ToJson() const {
  return {
      {kDiscriminatorType, "result_binary"},
      {"op", lexer::ToString(op_)},
      {"lhs", lhs_->ToJson()},
      {"rhs", rhs_->ToJson()},
  };
}

boost::json::value ResultTuple::ToJson() const {
  auto result_tuple = boost::json::object{};
  result_tuple[kDiscriminatorType] = "result_tuple";

  auto& results = (result_tuple["results"] = boost::json::array{}).as_array();
  results.reserve(results_.size());

  for (auto&& result : results_) {
    results.push_back(result->ToJson());
  }

  return result_tuple;
}

boost::json::value FuncCall::ToJson() const {
  return {
      {kDiscriminatorType, "func_call"},
      {"ident_code", ident_code_},
      {"arg", arg_->ToJson()},
  };
}

}  // namespace ast

}  // namespace parser
```

Файл parser.h:

```
#pragma once

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

namespace parser {

class Parser final {
 public:
  Parser(std::unique_ptr<lexer::IScanner>&& scanner)
      : scanner_(std::move(scanner)) {}

  Parser(const Parser& other) = delete;
  Parser& operator=(const Parser& other) = delete;

  std::unique_ptr<ast::Program> RecursiveDescentParse();

 private:
  std::unique_ptr<ast::Program> Program();
  std::unique_ptr<ast::Func> Func();
  std::unique_ptr<ast::FuncType> FuncType();
  std::unique_ptr<ast::Type> Type();
  std::unique_ptr<ast::ElementaryType> ElementaryType();
  std::unique_ptr<ast::ListType> ListType();
  std::unique_ptr<ast::TupleType> TupleType();
  std::unique_ptr<ast::FuncBody> FuncBody();
  std::unique_ptr<ast::Sentence> Sentence();
  std::unique_ptr<ast::Pattern> Pattern();
  std::unique_ptr<ast::Pattern> PatternUnit();
  template <typename Value>
  std::unique_ptr<ast::Const<Value>> Const();
  std::unique_ptr<ast::Pattern> PatternList();
  std::unique_ptr<ast::PatternBinary> PatternListItems();
  std::unique_ptr<ast::PatternTuple> PatternTuple();
  std::unique_ptr<ast::Result> Result();
  std::unique_ptr<ast::Result> ResultUnit();
  std::unique_ptr<ast::Result> Expr();
  std::unique_ptr<ast::Result> Term();
  std::unique_ptr<ast::Result> Factor();
  std::unique_ptr<ast::Result> Atom();
  std::unique_ptr<ast::Result> FuncArg();
  std::unique_ptr<ast::Result> ResultList();
  std::unique_ptr<ast::ResultBinary> ResultListItems();
  std::unique_ptr<ast::ResultTuple> ResultTuple();

  template <typename T>
  std::unique_ptr<T> ExpectAndCast(const lexer::DomainTag tag);
  void Expect(const lexer::DomainTag tag);
  template <typename T>
  std::unique_ptr<T> SymTo();
  [[noreturn]] void ThrowParseError(std::vector<lexer::DomainTag>&& expected);

  std::unique_ptr<lexer::IScanner> scanner_;
  std::unique_ptr<lexer::Token> sym_;
};

}  // namespace parser
```

Файл parser.cc:

```
#include "parser.h"

#include <iterator>
#include <sstream>
#include <stdexcept>

#include "node.h"
```

```
#include "token.h"

namespace parser {

using lexer::DomainTag;

std::unique_ptr<ast::Program> Parser::RecursiveDescentParse() {
  sym_ = scanner_->NextToken();
  auto program = Program();
  Expect(DomainTag::kEndOfProgram);
  return program;
}

// Program ::= Func*.
std::unique_ptr<ast::Program> Parser::Program() {
  auto funcs = std::vector<std::unique_ptr<ast::Func>>{};
  while (sym_->get_tag() == DomainTag::kIdent) {
    funcs.push_back(Func());
  }

  return std::make_unique<ast::Program>(std::make_move_iterator(funcs.begin()),
                                        std::make_move_iterator(funcs.end()));
}

// Func ::= Ident FuncType 'is' FuncBody 'end'.
std::unique_ptr<ast::Func> Parser::Func() {
  const auto ident = ExpectAndCast<lexer::IdentToken>(DomainTag::kIdent);
  auto type = FuncType();
  Expect(DomainTag::kIs);
  auto body = FuncBody();
  Expect(DomainTag::kEnd);

  return std::make_unique<ast::Func>(std::move(type), std::move(body),
                                     ident->get_code());
}

// FuncType ::= Type '::' Type.
std::unique_ptr<ast::FuncType> Parser::FuncType() {
  auto input = Type();
  Expect(DomainTag::kColonColon);
  auto output = Type();

  return std::make_unique<ast::FuncType>(std::move(input), std::move(output));
}

// Type ::= ElementaryType | ListType | TupleType.
```

```
std::unique_ptr<ast::Type> Parser::Type() {
  switch (sym_->get_tag()) {
    case DomainTag::kInt: {
      return ElementaryType();
    }
    case DomainTag::kStar: {
      return ListType();
    }
    case DomainTag::kParanthesisLeft: {
      return TupleType();
    }
    default: {
      ThrowParseError(
          {DomainTag::kInt, DomainTag::kStar, DomainTag::kParanthesisLeft});
    }
  }
}

// ElementaryType ::= 'int'.
std::unique_ptr<ast::ElementaryType> Parser::ElementaryType() {
  Expect(DomainTag::kInt);
  return std::make_unique<ast::ElementaryType>(DomainTag::kInt);
}

// ListType ::= '*' Type.
std::unique_ptr<ast::ListType> Parser::ListType() {
  Expect(DomainTag::kStar);
  return std::make_unique<ast::ListType>(Type());
}

// TupleType ::= '(' (Type (',' Type)*)? ')'.
std::unique_ptr<ast::TupleType> Parser::TupleType() {
  auto types = std::vector<std::unique_ptr<ast::Type>>{};

  Expect(DomainTag::kParanthesisLeft);
  if (const auto tag = sym_->get_tag(); tag == DomainTag::kInt ||
                                        tag == DomainTag::kStar ||
                                        tag == DomainTag::kParanthesisLeft) {
    types.push_back(Type());
    while (sym_->get_tag() == DomainTag::kComma) {
      sym_ = scanner_->NextToken();
      types.push_back(Type());
    }
  }
  Expect(DomainTag::kParanthesisRight);
  return std::make_unique<ast::TupleType>(
      std::make_move_iterator(types.begin()),
      std::make_move_iterator(types.end()));
}

// FuncBody ::= Sentence (';' Sentence)*.
std::unique_ptr<ast::FuncBody> Parser::FuncBody() {
  auto sents = std::vector<std::unique_ptr<ast::Sentence>>{};

  sents.push_back(Sentence());
  while (sym_->get_tag() == DomainTag::kSemicolon) {
    sym_ = scanner_->NextToken();
    sents.push_back(Sentence());
  }

  return std::make_unique<ast::FuncBody>(std::make_move_iterator(sents.begin()),
                                         std::make_move_iterator(sents.end()));
}

// Sentence ::= Pattern '=' Result.
std::unique_ptr<ast::Sentence> Parser::Sentence() {
  auto pattern = Pattern();
  Expect(DomainTag::kEqual);
  auto result = Result();

  return std::make_unique<ast::Sentence>(std::move(pattern), std::move(result));
}

// Pattern ::= PatternUnit (':' Pattern)?.
std::unique_ptr<ast::Pattern> Parser::Pattern() {
  auto pattern = PatternUnit();
  if (sym_->get_tag() == DomainTag::kColon) {
    sym_ = scanner_->NextToken();
    return std::make_unique<ast::PatternBinary>(std::move(pattern), Pattern(),
                                                DomainTag::kColon);
  }

  return pattern;
}

// PatternUnit ::= Ident | Const | PatternList | PatternTuple |
//                 '[' Pattern ']'.
std::unique_ptr<ast::Pattern> Parser::PatternUnit() {
  switch (sym_->get_tag()) {
    case DomainTag::kIdent: {
      const auto ident = SymTo<lexer::IdentToken>();
      sym_ = scanner_->NextToken();
      return std::make_unique<ast::Var>(ident->get_code());
    }
    case DomainTag::kIntConst: {
      return Const<std::int64_t>();
    }
    case DomainTag::kCurlyBracketLeft: {
      return PatternList();
    }
    case DomainTag::kParanthesisLeft: {
      return PatternTuple();
    }
    case DomainTag::kSquareBracketLeft: {
      sym_ = scanner_->NextToken();
      auto pattern = Pattern();
      Expect(DomainTag::kSquareBracketRight);
      return pattern;
    }
    default: {
      ThrowParseError({DomainTag::kIdent, DomainTag::kIntConst,
                       DomainTag::kCurlyBracketLeft,
                       DomainTag::kParanthesisLeft,
                       DomainTag::kSquareBracketLeft});
    }
  }
}

// Const ::= IntConst.
template <typename Value>
std::unique_ptr<ast::Const<Value>> Parser::Const() {
  const auto int_const =
      ExpectAndCast<lexer::IntConstToken>(DomainTag::kIntConst);
  return std::make_unique<ast::Const<std::int64_t>>(int_const->get_value(),
                                                    DomainTag::kIntConst);
}

// PatternList ::= '{' PatternListItems? '}' .
std::unique_ptr<ast::Pattern> Parser::PatternList() {
  Expect(DomainTag::kCurlyBracketLeft);

  std::unique_ptr<ast::Pattern> pattern;
  if (const auto tag = sym_->get_tag(); tag == DomainTag::kIdent ||
                                        tag == DomainTag::kIntConst ||
                                        tag == DomainTag::kCurlyBracketLeft ||
                                        tag == DomainTag::kParanthesisLeft ||
                                        tag == DomainTag::kSquareBracketLeft) {
```

```
    pattern = PatternListItems();
  } else {
    pattern = std::make_unique<ast::EmptyList>();
  }

  Expect(DomainTag::kCurlyBracketRight);

  return pattern;
}

// PatternListItems ::= Pattern (',' PatternListItems)? .
std::unique_ptr<ast::PatternBinary> Parser::PatternListItems() {
  auto lhs = Pattern();

  std::unique_ptr<ast::Pattern> rhs;
  if (sym_->get_tag() == DomainTag::kComma) {
    sym_ = scanner_->NextToken();
    rhs = PatternListItems();
  } else {
    rhs = std::make_unique<ast::EmptyList>();
  }

  return std::make_unique<ast::PatternBinary>(std::move(lhs), std::move(rhs),
                                              DomainTag::kColon);
}

// PatternTuple ::= '(' (Pattern (',' Pattern)*)? ')'.
std::unique_ptr<ast::PatternTuple> Parser::PatternTuple() {
  auto patterns = std::vector<std::unique_ptr<ast::Pattern>>{};

  Expect(DomainTag::kParanthesisLeft);
  if (const auto tag = sym_->get_tag(); tag == DomainTag::kIdent ||
                                        tag == DomainTag::kIntConst ||
                                        tag == DomainTag::kCurlyBracketLeft ||
                                        tag == DomainTag::kParanthesisLeft ||
                                        tag == DomainTag::kSquareBracketLeft) {
    patterns.push_back(Pattern());
    while (sym_->get_tag() == DomainTag::kComma) {
      sym_ = scanner_->NextToken();
      patterns.push_back(Pattern());
    }
  }
  Expect(DomainTag::kParanthesisRight);

  return std::make_unique<ast::PatternTuple>(
      std::make_move_iterator(patterns.begin()),
      std::make_move_iterator(patterns.end()));
}

// Result ::= ResultUnit (':' Result)?.
std::unique_ptr<ast::Result> Parser::Result() {
  auto result = ResultUnit();
  if (sym_->get_tag() == DomainTag::kColon) {
    sym_ = scanner_->NextToken();
    return std::make_unique<ast::ResultBinary>(std::move(result), Result(),
                                               DomainTag::kColon);
  }

  return result;
}

// ResultUnit ::= Expr | ResultList | ResultTuple.
std::unique_ptr<ast::Result> Parser::ResultUnit() {
  const auto tag = sym_->get_tag();
  if (tag == DomainTag::kIntConst || tag == DomainTag::kIdent ||
      tag == DomainTag::kSquareBracketLeft) {
    return Expr();
  } else if (tag == DomainTag::kCurlyBracketLeft) {
    return ResultList();
  } else if (tag == DomainTag::kParanthesisLeft) {
    return ResultTuple();
  } else {
    ThrowParseError(
        {DomainTag::kIntConst, DomainTag::kIdent, DomainTag::kSquareBracketLeft,
         DomainTag::kCurlyBracketLeft, DomainTag::kParanthesisLeft});
  }
}

// Expr ::= Term ('+' Term | '-' Term)*.
std::unique_ptr<ast::Result> Parser::Expr() {
  auto result = Term();
  for (auto tag = sym_->get_tag();
       tag == DomainTag::kPlus || tag == DomainTag::kMinus;
       tag = sym_->get_tag()) {
    sym_ = scanner_->NextToken();
    result =
        std::make_unique<ast::ResultBinary>(std::move(result), Term(), tag);
  }

  return result;
}
```

```
// Term ::= Factor ('*' Factor | '/' Factor)*.
std::unique_ptr<ast::Result> Parser::Term() {
  auto result = Factor();
  for (auto tag = sym_->get_tag();
       tag == DomainTag::kStar || tag == DomainTag::kSlash;
       tag = sym_->get_tag()) {
    sym_ = scanner_->NextToken();
    result =
        std::make_unique<ast::ResultBinary>(std::move(result), Factor(), tag);
  }

  return result;
}

// Factor ::= Atom | '[' Expr ']'.
std::unique_ptr<ast::Result> Parser::Factor() {
  const auto tag = sym_->get_tag();
  if (tag == DomainTag::kIntConst || tag == DomainTag::kIdent) {
    return Atom();
  } else if (tag == DomainTag::kSquareBracketLeft) {
    sym_ = scanner_->NextToken();
    auto expr = Expr();
    Expect(DomainTag::kSquareBracketRight);
    return expr;
  } else {
    ThrowParseError({DomainTag::kIntConst, DomainTag::kIdent,
                     DomainTag::kSquareBracketLeft});
  }
}

// Atom ::= Const | Ident FuncArg?.
std::unique_ptr<ast::Result> Parser::Atom() {
  const auto tag = sym_->get_tag();
  if (tag == DomainTag::kIntConst) {
    return Const<std::int64_t>();
  } else if (tag == DomainTag::kIdent) {
    const auto ident = SymTo<lexer::IdentToken>();
    sym_ = scanner_->NextToken();
    if (const auto tag = sym_->get_tag();
        tag == DomainTag::kIntConst || tag == DomainTag::kIdent ||
        tag == DomainTag::kCurlyBracketLeft ||
        tag == DomainTag::kParanthesisLeft ||
        tag == DomainTag::kSquareBracketLeft) {
      return std::make_unique<ast::FuncCall>(FuncArg(), ident->get_code());
    }
    return std::make_unique<ast::Var>(ident->get_code());
  } else {
    ThrowParseError({DomainTag::kIntConst, DomainTag::kIdent});
  }
}

// FuncArg ::= Atom | ResultList | ResultTuple | '[' Result ']'.
std::unique_ptr<ast::Result> Parser::FuncArg() {
  const auto tag = sym_->get_tag();
  if (tag == DomainTag::kIntConst || tag == DomainTag::kIdent) {
    return Atom();
  } else if (tag == DomainTag::kCurlyBracketLeft) {
    return ResultList();
  } else if (tag == DomainTag::kParanthesisLeft) {
    return ResultTuple();
  } else if (tag == DomainTag::kSquareBracketLeft) {
    sym_ = scanner_->NextToken();
    auto result = Result();
    Expect(DomainTag::kSquareBracketRight);
    return result;
  } else {
    ThrowParseError({DomainTag::kIntConst, DomainTag::kIdent,
                     DomainTag::kCurlyBracketLeft, DomainTag::kParanthesisLeft,
                     DomainTag::kSquareBracketLeft});
  }
}

// ResultList ::= '{' ResultListItems? '}' .
std::unique_ptr<ast::Result> Parser::ResultList() {
  Expect(DomainTag::kCurlyBracketLeft);

  std::unique_ptr<ast::Result> result;
  if (const auto tag = sym_->get_tag(); tag == DomainTag::kIdent ||
                                        tag == DomainTag::kIntConst ||
                                        tag == DomainTag::kCurlyBracketLeft ||
                                        tag == DomainTag::kParanthesisLeft ||
                                        tag == DomainTag::kSquareBracketLeft) {
    result = ResultListItems();
  } else {
    result = std::make_unique<ast::EmptyList>();
  }

  Expect(DomainTag::kCurlyBracketRight);

  return result;
}
```

```
// ResultListItems ::= Result (',' ResultListItems)? .
std::unique_ptr<ast::ResultBinary> Parser::ResultListItems() {
  auto lhs = Result();

  std::unique_ptr<ast::Result> rhs;
  if (sym_->get_tag() == DomainTag::kComma) {
    sym_ = scanner_->NextToken();
    rhs = ResultListItems();
  } else {
    rhs = std::make_unique<ast::EmptyList>();
  }

  return std::make_unique<ast::ResultBinary>(std::move(lhs), std::move(rhs),
                                             DomainTag::kColon);
}

// ResultTuple ::= '(' (Result (',' Result)*)? ')'.
std::unique_ptr<ast::ResultTuple> Parser::ResultTuple() {
  auto results = std::vector<std::unique_ptr<ast::Result>>{};

  Expect(DomainTag::kParanthesisLeft);
  if (const auto tag = sym_->get_tag(); tag == DomainTag::kIntConst ||
                                        tag == DomainTag::kIdent ||
                                        tag == DomainTag::kSquareBracketLeft ||
                                        tag == DomainTag::kCurlyBracketLeft ||
                                        tag == DomainTag::kParanthesisLeft) {
    results.push_back(Result());
    while (sym_->get_tag() == DomainTag::kComma) {
      sym_ = scanner_->NextToken();
      results.push_back(Result());
    }
  }
  Expect(DomainTag::kParanthesisRight);

  return std::make_unique<ast::ResultTuple>(
      std::make_move_iterator(results.begin()),
      std::make_move_iterator(results.end()));
}

template <typename T>
std::unique_ptr<T> Parser::ExpectAndCast(const DomainTag tag) {
  if (sym_->get_tag() != tag) {
    ThrowParseError({tag});
  }

  auto casted_sym = SymTo<T>();
  sym_ = scanner_->NextToken();
  return casted_sym;
}

template <typename T>
std::unique_ptr<T> Parser::SymTo() {
  return std::unique_ptr<T>{static_cast<T*>(sym_.release())};
}

void Parser::Expect(const DomainTag tag) {
  if (sym_->get_tag() != tag) {
    ThrowParseError({tag});
  }

  sym_ = scanner_->NextToken();
}

[[noreturn]] void Parser::ThrowParseError(std::vector<DomainTag>&& expected) {
  std::ostringstream oss;
  oss << sym_->get_coords() << ": expected ";

  for (const auto tag : expected) {
    oss << lexer::ToString(tag) << ", ";
  }

  oss << "got " << lexer::ToString(sym_->get_tag());
  throw std::runtime_error(oss.str());
}

}  // namespace parser
```

Тестирование

Вывод АСТ для программы из индивидуального варианта:

```
{
  "funcs": [
    {
      "ident_code": 0,
      "type": {
        "input": {
          "discriminator_type": "tuple_type",
          "types": [
            {
              "discriminator_type": "list_type",
            "type": {
              "discriminator_type": "elementary_type",
              "tag": "INT"
            }
          },
          {
            "discriminator_type": "list_type",
            "type": {
              "discriminator_type": "elementary_type",
              "tag": "INT"
            }
          }
        ]
      },
      "output": {
        "discriminator_type": "list_type",
        "type": {
          "discriminator_type": "tuple_type",
          "types": [
            {
              "discriminator_type": "elementary_type",
              "tag": "INT"
            },
            {
              "discriminator_type": "elementary_type",
              "tag": "INT"
            }
          ]
        }
      }
    },
    "body": {
      "sents": [
        {
          "pattern": {
            "discriminator_type": "pattern_tuple",
            "patterns": [
              {
                "discriminator_type": "pattern_binary",
                "op": "COLON",
                "lhs": {
                  "discriminator_type": "var",
                  "ident_code": 1
                },
                "rhs": {
                  "discriminator_type": "var",
        "ident_code": 2
      }
    },
    {
      "discriminator_type": "pattern_binary",
      "op": "COLON",
      "lhs": {
        "discriminator_type": "var",
        "ident_code": 3
      },
      "rhs": {
        "discriminator_type": "var",
        "ident_code": 4
      }
    }
  ]
},
"result": {
  "discriminator_type": "result_binary",
  "op": "COLON",
  "lhs": {
    "discriminator_type": "result_tuple",
    "results": [
      {
        "discriminator_type": "var",
        "ident_code": 1
      },
      {
        "discriminator_type": "var",
        "ident_code": 3
      }
    ]
  },
  "rhs": {
    "discriminator_type": "func_call",
    "ident_code": 0,
    "arg": {
      "discriminator_type": "result_tuple",
      "results": [
        {
          "discriminator_type": "var",
          "ident_code": 2
        },
        {
          "discriminator_type": "var",
          "ident_code": 4
```

```
                }
              ]
            }
          }
        }
      },
      {
        "pattern": {
          "discriminator_type": "pattern_tuple",
          "patterns": [
            {
              "discriminator_type": "var",
              "ident_code": 2
            },
            {
              "discriminator_type": "var",
              "ident_code": 4
            }
          ]
        },
        "result": {
          "discriminator_type": "empty_list"
        }
      }
    ]
  }
},
{
  "ident_code": 5,
  "type": {
    "input": {
      "discriminator_type": "tuple_type",
      "types": [
        {
          "discriminator_type": "list_type",
          "type": {
            "discriminator_type": "elementary_type",
            "tag": "INT"
          }
        },
        {
          "discriminator_type": "list_type",
          "type": {
            "discriminator_type": "elementary_type",
            "tag": "INT"
          }
      }
    ]
  },
  "output": {
    "discriminator_type": "list_type",
    "type": {
      "discriminator_type": "tuple_type",
      "types": [
        {
          "discriminator_type": "elementary_type",
          "tag": "INT"
        },
        {
          "discriminator_type": "elementary_type",
          "tag": "INT"
        }
      ]
    }
  }
},
"body": {
  "sents": [
    {
      "pattern": {
        "discriminator_type": "pattern_tuple",
        "patterns": [
          {
            "discriminator_type": "pattern_binary",
            "op": "COLON",
            "lhs": {
              "discriminator_type": "var",
              "ident_code": 1
            },
            "rhs": {
              "discriminator_type": "var",
              "ident_code": 2
            }
          },
          {
            "discriminator_type": "var",
            "ident_code": 4
          }
        ]
      },
      "result": {
        "discriminator_type": "func_call",
    "ident_code": 6,
    "arg": {
      "discriminator_type": "result_tuple",
      "results": [
        {
          "discriminator_type": "func_call",
          "ident_code": 7,
          "arg": {
            "discriminator_type": "result_tuple",
            "results": [
              {
                "discriminator_type": "var",
                "ident_code": 1
              },
              {
                "discriminator_type": "var",
                "ident_code": 4
              }
            ]
          }
        },
        {
          "discriminator_type": "func_call",
          "ident_code": 5,
          "arg": {
            "discriminator_type": "result_tuple",
            "results": [
              {
                "discriminator_type": "var",
                "ident_code": 2
              },
              {
                "discriminator_type": "var",
                "ident_code": 4
              }
            ]
          }
        }
      ]
    }
  }
},
{
  "pattern": {
    "discriminator_type": "pattern_tuple",
    "patterns": [
            {
              "discriminator_type": "empty_list"
            },
            {
              "discriminator_type": "var",
              "ident_code": 4
            }
          ]
        },
        "result": {
          "discriminator_type": "empty_list"
        }
      }
    ]
  }
},
{
  "ident_code": 7,
  "type": {
    "input": {
      "discriminator_type": "tuple_type",
      "types": [
        {
          "discriminator_type": "elementary_type",
          "tag": "INT"
        },
        {
          "discriminator_type": "list_type",
          "type": {
            "discriminator_type": "elementary_type",
            "tag": "INT"
          }
        }
      ]
    },
    "output": {
      "discriminator_type": "list_type",
      "type": {
        "discriminator_type": "tuple_type",
        "types": [
          {
            "discriminator_type": "elementary_type",
            "tag": "INT"
          },
          {
            "discriminator_type": "elementary_type",
          "tag": "INT"
        }
      ]
    }
  }
},
"body": {
  "sents": [
    {
      "pattern": {
        "discriminator_type": "pattern_tuple",
        "patterns": [
          {
            "discriminator_type": "var",
            "ident_code": 1
          },
          {
            "discriminator_type": "empty_list"
          }
        ]
      },
      "result": {
        "discriminator_type": "empty_list"
      }
    },
    {
      "pattern": {
        "discriminator_type": "pattern_tuple",
        "patterns": [
          {
            "discriminator_type": "var",
            "ident_code": 1
          },
          {
            "discriminator_type": "pattern_binary",
            "op": "COLON",
            "lhs": {
              "discriminator_type": "var",
              "ident_code": 3
            },
            "rhs": {
              "discriminator_type": "var",
              "ident_code": 4
            }
          }
        ]
        },
        "result": {
          "discriminator_type": "result_binary",
          "op": "COLON",
          "lhs": {
            "discriminator_type": "result_tuple",
            "results": [
              {
                "discriminator_type": "var",
                "ident_code": 1
              },
              {
                "discriminator_type": "var",
                "ident_code": 3
              }
            ]
          },
          "rhs": {
            "discriminator_type": "func_call",
            "ident_code": 7,
            "arg": {
              "discriminator_type": "result_tuple",
              "results": [
                {
                  "discriminator_type": "var",
                  "ident_code": 1
                },
                {
                  "discriminator_type": "var",
                  "ident_code": 4
                }
              ]
            }
          }
        }
      }
    ]
  }
},
{
  "ident_code": 6,
  "type": {
    "input": {
      "discriminator_type": "tuple_type",
      "types": [
        {
```

```
      "discriminator_type": "list_type",
      "type": {
        "discriminator_type": "tuple_type",
        "types": [
          {
            "discriminator_type": "elementary_type",
            "tag": "INT"
          },
          {
            "discriminator_type": "elementary_type",
            "tag": "INT"
          }
        ]
      }
    },
    {
      "discriminator_type": "list_type",
      "type": {
        "discriminator_type": "tuple_type",
        "types": [
          {
            "discriminator_type": "elementary_type",
            "tag": "INT"
          },
          {
            "discriminator_type": "elementary_type",
            "tag": "INT"
          }
        ]
      }
    }
  ]
},
"output": {
  "discriminator_type": "list_type",
  "type": {
    "discriminator_type": "tuple_type",
    "types": [
      {
        "discriminator_type": "elementary_type",
        "tag": "INT"
      },
      {
        "discriminator_type": "elementary_type",
        "tag": "INT"
      }
      ]
    }
  }
},
"body": {
  "sents": [
    {
      "pattern": {
        "discriminator_type": "pattern_tuple",
        "patterns": [
          {
            "discriminator_type": "pattern_binary",
            "op": "COLON",
            "lhs": {
              "discriminator_type": "var",
              "ident_code": 1
            },
            "rhs": {
              "discriminator_type": "var",
              "ident_code": 2
            }
          },
          {
            "discriminator_type": "var",
            "ident_code": 4
          }
        ]
      },
      "result": {
        "discriminator_type": "result_binary",
        "op": "COLON",
        "lhs": {
          "discriminator_type": "var",
          "ident_code": 1
        },
        "rhs": {
          "discriminator_type": "func_call",
          "ident_code": 6,
          "arg": {
            "discriminator_type": "result_tuple",
            "results": [
              {
                "discriminator_type": "var",
                "ident_code": 2
              },
              {
                  "discriminator_type": "var",
                  "ident_code": 4
                }
              ]
            }
          }
        }
      },
      {
        "pattern": {
          "discriminator_type": "pattern_tuple",
          "patterns": [
            {
              "discriminator_type": "empty_list"
            },
            {
              "discriminator_type": "var",
              "ident_code": 4
            }
          ]
        },
        "result": {
          "discriminator_type": "var",
          "ident_code": 4
        }
      }
    ]
  }
},
{
  "ident_code": 8,
  "type": {
    "input": {
      "discriminator_type": "list_type",
      "type": {
        "discriminator_type": "list_type",
        "type": {
          "discriminator_type": "elementary_type",
          "tag": "INT"
        }
      }
    },
    "output": {
      "discriminator_type": "list_type",
      "type": {
        "discriminator_type": "elementary_type",
      "tag": "INT"
    }
  }
},
"body": {
  "sents": [
    {
      "pattern": {
        "discriminator_type": "pattern_binary",
        "op": "COLON",
        "lhs": {
          "discriminator_type": "pattern_binary",
          "op": "COLON",
          "lhs": {
            "discriminator_type": "var",
            "ident_code": 1
          },
          "rhs": {
            "discriminator_type": "var",
            "ident_code": 2
          }
        },
        "rhs": {
          "discriminator_type": "var",
          "ident_code": 9
        }
      },
      "result": {
        "discriminator_type": "result_binary",
        "op": "COLON",
        "lhs": {
          "discriminator_type": "var",
          "ident_code": 1
        },
        "rhs": {
          "discriminator_type": "func_call",
          "ident_code": 8,
          "arg": {
            "discriminator_type": "result_binary",
            "op": "COLON",
            "lhs": {
              "discriminator_type": "var",
              "ident_code": 2
            },
            "rhs": {
              "discriminator_type": "var",
                "ident_code": 9
              }
            }
          }
        }
      },
      {
        "pattern": {
          "discriminator_type": "pattern_binary",
          "op": "COLON",
          "lhs": {
            "discriminator_type": "empty_list"
          },
          "rhs": {
            "discriminator_type": "var",
            "ident_code": 9
          }
        },
        "result": {
          "discriminator_type": "func_call",
          "ident_code": 8,
          "arg": {
            "discriminator_type": "var",
            "ident_code": 9
          }
        }
      },
      {
        "pattern": {
          "discriminator_type": "empty_list"
        },
        "result": {
          "discriminator_type": "empty_list"
        }
      }
    ]
  }
},
{
  "ident_code": 10,
  "type": {
    "input": {
      "discriminator_type": "list_type",
      "type": {
        "discriminator_type": "elementary_type",
        "tag": "INT"
    }
  },
  "output": {
    "discriminator_type": "elementary_type",
    "tag": "INT"
  }
},
"body": {
  "sents": [
    {
      "pattern": {
        "discriminator_type": "pattern_binary",
        "op": "COLON",
        "lhs": {
          "discriminator_type": "var",
          "ident_code": 1
        },
        "rhs": {
          "discriminator_type": "var",
          "ident_code": 2
        }
      },
      "result": {
        "discriminator_type": "result_binary",
        "op": "PLUS",
        "lhs": {
          "discriminator_type": "var",
          "ident_code": 1
        },
        "rhs": {
          "discriminator_type": "func_call",
          "ident_code": 10,
          "arg": {
            "discriminator_type": "var",
            "ident_code": 2
          }
        }
      }
    },
    {
      "pattern": {
        "discriminator_type": "empty_list"
      },
      "result": {
        "discriminator_type": "int_const",
        "value": 0
        }
      }
    ]
  }
},
{
  "ident_code": 11,
  "type": {
    "input": {
      "discriminator_type": "tuple_type",
      "types": [
        {
          "discriminator_type": "elementary_type",
          "tag": "INT"
        },
        {
          "discriminator_type": "list_type",
          "type": {
            "discriminator_type": "elementary_type",
            "tag": "INT"
          }
        }
      ]
    },
    "output": {
      "discriminator_type": "elementary_type",
      "tag": "INT"
    }
  },
  "body": {
    "sents": [
      {
        "pattern": {
          "discriminator_type": "pattern_tuple",
          "patterns": [
            {
              "discriminator_type": "var",
              "ident_code": 1
            },
            {
              "discriminator_type": "empty_list"
            }
          ]
        },
        "result": {
          "discriminator_type": "int_const",
    "value": 0
  }
},
{
  "pattern": {
    "discriminator_type": "pattern_tuple",
    "patterns": [
      {
        "discriminator_type": "var",
        "ident_code": 1
      },
      {
        "discriminator_type": "pattern_binary",
        "op": "COLON",
        "lhs": {
          "discriminator_type": "var",
          "ident_code": 12
        },
        "rhs": {
          "discriminator_type": "var",
          "ident_code": 13
        }
      }
    ]
  },
  "result": {
    "discriminator_type": "result_binary",
    "op": "PLUS",
    "lhs": {
      "discriminator_type": "result_binary",
      "op": "STAR",
      "lhs": {
        "discriminator_type": "func_call",
        "ident_code": 11,
        "arg": {
          "discriminator_type": "result_tuple",
          "results": [
            {
              "discriminator_type": "var",
              "ident_code": 1
            },
            {
              "discriminator_type": "var",
              "ident_code": 13
            }
          ]
```

```
              }
            },
            "rhs": {
              "discriminator_type": "var",
              "ident_code": 1
            }
          },
          "rhs": {
            "discriminator_type": "var",
            "ident_code": 12
          }
        }
      }
    ]
  }
},
{
  "ident_code": 14,
  "type": {
    "input": {
      "discriminator_type": "elementary_type",
      "tag": "INT"
    },
    "output": {
      "discriminator_type": "elementary_type",
      "tag": "INT"
    }
  },
  "body": {
    "sents": [
      {
        "pattern": {
          "discriminator_type": "var",
          "ident_code": 1
        },
        "result": {
          "discriminator_type": "func_call",
          "ident_code": 11,
          "arg": {
            "discriminator_type": "result_tuple",
            "results": [
              {
                "discriminator_type": "var",
                "ident_code": 1
              },
              {
                    "discriminator_type": "result_binary",
                    "op": "COLON",
                    "lhs": {
                      "discriminator_type": "int_const",
                      "value": 1
                    },
                    "rhs": {
                      "discriminator_type": "result_binary",
                      "op": "COLON",
                      "lhs": {
                        "discriminator_type": "int_const",
                        "value": 1
                      },
                      "rhs": {
                        "discriminator_type": "result_binary",
                        "op": "COLON",
                        "lhs": {
                          "discriminator_type": "int_const",
                          "value": 1
                        },
                        "rhs": {
                          "discriminator_type": "result_binary",
                          "op": "COLON",
                          "lhs": {
                            "discriminator_type": "int_const",
                            "value": 1
                          },
                          "rhs": {
                            "discriminator_type": "empty_list"
                          }
                        }
                      }
                    }
                  }
                ]
              }
            }
          }
        ]
      }
    }
  ]
}
```

Вывод

В результате выполнения лабораторной работы я закрепил навыки написания парсера методом рекурсивного спуска. Рекурсивный спуск — самая простая и интуитивно понятная техника для решения такого рода задач, предлагающая, к тому же, хорошие возможности “кастомизации” разбора. Поэтому реализовывать формальный шаг алгоритма было нетрудно и даже в каком-то смысле приятно.