“Лабораторная работа 3.2 «Форматтер исходных текстов»”

3 июня 2024 г.

Илья Афанасьев, ИУ9-61Б

Цель работы

Целью данной работы является приобретение навыков использования генератора синтаксических анализаторов bison.

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
  (xs, ys) =
    case xs of
      x : xs = append (bind (x, ys), cart_prod(xs, ys));
      {} = {}
    end
end

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

Реализация

Файл `main.cc`:

```
#include <exception>
#include <iostream>

#include "driver.h"
#include "formatter.h"

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

```
  auto driver = fmt::Driver{};
  driver.Parse(argv[1]);

  auto formatter = fmt::Formatter{driver.get_ident_table()};
  driver.get_program()->Accept(formatter);
  std::cout << formatter.ToString() << std::endl;
} catch (const std::exception& e) {
  std::cerr << e.what() << std::endl;
}
```

Драйвер

Файл `driver.h`:

```
#pragma once

#include <memory>

#include "ast.h"
#include "ident_table.h"
#include "scanner.h"

namespace fmt {

class Driver final {
  bool trace_scanning_, trace_parsing_;
  std::shared_ptr<Program> program_;
  std::shared_ptr<IdentTable> table_ = std::make_shared<IdentTable>();

 public:
  void Parse(const std::string& filename);

  void set_trace_scanning(const bool is_active) noexcept {
    trace_scanning_ = is_active;
  }
  void set_trace_parsing(const bool is_active) noexcept {
    trace_parsing_ = is_active;
  }

  void set_program(std::shared_ptr<Program>&& program) noexcept {
    program_ = std::move(program);
  }
  std::shared_ptr<const Program> get_program() const noexcept {
    return program_;
  }
```

```
  std::shared_ptr<IdentTable> get_ident_table() noexcept { return table_; }
  std::shared_ptr<const IdentTable> get_ident_table() const noexcept {
    return table_;
  }
};

}  // namespace fmt
```

Файл `driver.cc`:

```
#include "driver.h"

#include <fstream>

namespace fmt {

void Driver::Parse(const std::string& filename) {
  auto file = std::ifstream{filename};
  if (!file.is_open()) {
    throw std::runtime_error("Failed to open file " + filename);
  }

  auto scanner = Scanner{file, std::cout, &filename};
  scanner.set_debug(trace_scanning_);

  auto parser = Parser{scanner, *this};
  parser.set_debug_level(trace_parsing_);

  parser.parse();
}

}  // namespace fmt
```

Абстрактное синтаксическое дерево

Файл `ast.h`:

```
#pragma once

#include <memory>
#include <string>
#include <vector>

#include "visitor.h"

namespace fmt {
```

```
class Func;
class FuncType;
class FuncBody;
class IType;
class Statement;
class IPattern;
class IResult;

enum class Op {
  kCons,
  kAdd,
  kSub,
  kMul,
  kDiv,
};

std::ostream& operator<<(std::ostream& os, const Op op);

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

  virtual void Accept(IVisitor& visitor) const = 0;
};

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

 public:
  Program(std::vector<std::unique_ptr<Func>>&& funcs) noexcept
      : funcs_(std::move(funcs)) {}

  auto FuncsCbegin() const noexcept { return funcs_.cbegin(); }
  auto FuncsCend() const noexcept { return funcs_.cend(); }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

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

 public:
  Func(std::unique_ptr<FuncType>&& type, std::unique_ptr<FuncBody>&& body,
       const std::size_t ident_code) noexcept
```

```
      : ident_code_(ident_code),
        type_(std::move(type)),
        body_(std::move(body)) {}

  std::size_t get_ident_code() const noexcept { return ident_code_; }
  const FuncType& get_type() const noexcept { return *type_; }
  const FuncBody& get_body() const noexcept { return *body_; }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

class FuncType final : public INode {
  std::unique_ptr<IType> in_, out_;

 public:
  FuncType(std::unique_ptr<IType>&& in, std::unique_ptr<IType>&& out) noexcept
      : in_(std::move(in)), out_(std::move(out)) {}

  const IType& get_in() const noexcept { return *in_; }
  const IType& get_out() const noexcept { return *out_; }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

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

class ElementaryType final : public IType {
 public:
  enum class Kind {
    kInt,
  };

 public:
  ElementaryType(const Kind kind) noexcept : kind_(kind) {}

  Kind get_kind() const noexcept { return kind_; }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }

 private:
  Kind kind_;
};
```

```
std::ostream& operator<<(std::ostream& os, const ElementaryType::Kind kind);

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

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

  const IType& get_type() const noexcept { return *type_; }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

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

 public:
  TupleType(std::vector<std::unique_ptr<IType>>&& types) noexcept
      : types_(std::move(types)) {}

  auto TypesCbegin() const noexcept { return types_.cbegin(); }
  auto TypesCend() const noexcept { return types_.cend(); }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

class FuncBody final : public INode {
  std::vector<std::unique_ptr<Statement>> statements_;

 public:
  FuncBody(std::vector<std::unique_ptr<Statement>>&& statements) noexcept
      : statements_(std::move(statements)) {}

  auto StatementsCbegin() const noexcept { return statements_.cbegin(); }
  auto StatementsCend() const noexcept { return statements_.cend(); }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

class Statement final : public INode {
  std::unique_ptr<IPattern> pattern_;
  std::unique_ptr<IResult> result_;

 public:
  Statement(std::unique_ptr<IPattern>&& pattern,
            std::unique_ptr<IResult>&& result) noexcept
```

```
      : pattern_(std::move(pattern)), result_(std::move(result)) {}

  const IPattern& get_pattern() const noexcept { return *pattern_; }
  const IResult& get_result() const noexcept { return *result_; }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

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

class PatternList final : public IPattern {
  std::vector<std::unique_ptr<IPattern>> patterns_;

 public:
  PatternList(std::vector<std::unique_ptr<IPattern>>&& patterns) noexcept
      : patterns_(std::move(patterns)) {}

  auto PatternsCbegin() const noexcept { return patterns_.cbegin(); }
  auto PatternsCend() const noexcept { return patterns_.cend(); }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

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

 public:
  PatternTuple(std::vector<std::unique_ptr<IPattern>>&& patterns) noexcept
      : patterns_(std::move(patterns)) {}

  auto PatternsCbegin() const noexcept { return patterns_.cbegin(); }
  auto PatternsCend() const noexcept { return patterns_.cend(); }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

class PatternBinary final : public IPattern {
  std::unique_ptr<IPattern> lhs_, rhs_;
  Op op_;

 public:
  PatternBinary(std::unique_ptr<IPattern>&& lhs,
                std::unique_ptr<IPattern>&& rhs, const Op op) noexcept
```

```
      : lhs_(std::move(lhs)), rhs_(std::move(rhs)), op_(op) {}

  const IPattern& get_lhs() const noexcept { return *lhs_; }
  const IPattern& get_rhs() const noexcept { return *rhs_; }
  Op get_op() const noexcept { return op_; }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

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

class ResultList final : public IResult {
  std::vector<std::unique_ptr<IResult>> results_;

 public:
  ResultList(std::vector<std::unique_ptr<IResult>> results) noexcept
      : results_(std::move(results)) {}

  auto ResultsCbegin() const noexcept { return results_.cbegin(); }
  auto ResultsCend() const noexcept { return results_.cend(); }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

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

 public:
  ResultTuple(std::vector<std::unique_ptr<IResult>>&& results)
      : results_(std::move(results)) {}

  auto ResultsCbegin() const noexcept { return results_.cbegin(); }
  auto ResultsCend() const noexcept { return results_.cend(); }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

class ResultBinary final : public IResult {
  std::unique_ptr<IResult> lhs_, rhs_;
  Op op_;

 public:
  ResultBinary(std::unique_ptr<IResult>&& lhs, std::unique_ptr<IResult>&& rhs,
```

```
                const Op op) noexcept
      : lhs_(std::move(lhs)), rhs_(std::move(rhs)), op_(op) {}

  const IResult& get_lhs() const noexcept { return *lhs_; }
  const IResult& get_rhs() const noexcept { return *rhs_; }
  Op get_op() const noexcept { return op_; }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

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

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

  std::size_t get_ident_code() const noexcept { return ident_code_; }
  const IResult& get_arg() const noexcept { return *arg_; }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

class CaseExpr final : public IResult {
  std::size_t ident_code_;
  std::vector<std::unique_ptr<Statement>> statements_;

 public:
  CaseExpr(std::vector<std::unique_ptr<Statement>>&& statements,
           const std::size_t ident_code) noexcept
      : ident_code_(ident_code), statements_(std::move(statements)) {}

  std::size_t get_ident_code() const noexcept { return ident_code_; }
  auto StatementsCbegin() const noexcept { return statements_.cbegin(); }
  auto StatementsCend() const noexcept { return statements_.cend(); }

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

class Ident final : public IPattern, public IResult {
  std::size_t code_;

 public:
  Ident(const std::size_t code) : code_(code) {}
```

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

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

class IConst : public IPattern, public IResult {
 public:
  virtual ~IConst() = default;
};

class IntConst final : public IConst {
  std::size_t value_;

 public:
  IntConst(const std::size_t value) noexcept : value_(value) {}

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

  void Accept(IVisitor& visitor) const override { visitor.Visit(*this); }
};

}  // namespace fmt
```

Файл `ast.cc`:

```
#include "ast.h"

namespace fmt {

std::ostream& operator<<(std::ostream& os, const Op op) {
  switch (op) {
    case Op::kCons: {
      return os << ':';
    }
    case Op::kAdd: {
      return os << '+';
    }
    case Op::kSub: {
      return os << '-';
    }
    case Op::kMul: {
      return os << '*';
    }
    case Op::kDiv: {
      return os << '/';
    }
```

```
  }
}

std::ostream& operator<<(std::ostream& os, const ElementaryType::Kind kind) {
  switch (kind) {
    case ElementaryType::Kind::kInt: {
      return os << "int";
    }
  }
}

}  // namespace fmt
```

Лексический анализ

Файл scanner.h:

```
#pragma once

#include <iostream>
#include <ostream>

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

#undef YY_DECL
#define YY_DECL fmt::Parser::symbol_type fmt::Scanner::Get(fmt::Driver& driver)

#include "location.h"
#include "parser.h"

namespace fmt {

class Driver;

class Scanner final : public yyFlexLexer {
 public:
  Scanner(std::istream& is = std::cin, std::ostream& os = std::cout,
          const std::string* isname = nullptr)
      : yyFlexLexer(is, os), loc_(isname) {}

  Parser::symbol_type Get(Driver& driver);

 private:
  location loc_;
```

```
};

}  // namespace fmt
```

Файл `scanner.l`:

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

#define yyterminate() return Parser::make_YYEOF(loc_)

#define YY_USER_ACTION loc_.columns(yyleng);

using fmt::Parser;
%}

%option c++
%option yyclass="fmt::Scanner"
%option noyywrap nounput noinput
%option batch
%option debug

BLANK   [ \t\r]
IDENT   [A-Za-z_][A-Za-z_0-9]*
NUMBER  [0-9]+

%%

%{
  loc_.step();
%}

"@".*     { loc_.step(); }
{BLANK}+  { loc_.step(); }
\n+       { loc_.lines(yyleng); loc_.step(); }
"="       { return Parser::make_EQUALS(loc_); }
","       { return Parser::make_COMMA(loc_); }
";"       { return Parser::make_SEMICOLON(loc_); }
"::"      { return Parser::make_COLON_COLON(loc_); }
"("       { return Parser::make_LEFT_PARENTHESIS(loc_); }
")"       { return Parser::make_RIGHT_PARENTHESIS(loc_); }
"{"       { return Parser::make_LEFT_CURLY_BRACKET(loc_); }
"}"       { return Parser::make_RIGHT_CURLY_BRACKET(loc_); }
"["       { return Parser::make_LEFT_SQUARE_BRACKET(loc_); }
"]"       { return Parser::make_RIGHT_SQUARE_BRACKET(loc_); }
":"       { return Parser::make_COLON(loc_); }
"+"       { return Parser::make_PLUS(loc_); }
```

```
"-"       { return Parser::make_MINUS(loc_); }
"*"       { return Parser::make_STAR(loc_); }
"/"       { return Parser::make_SLASH(loc_); }
"case"    { return Parser::make_CASE(loc_); }
"end"     { return Parser::make_END(loc_); }
"int"     { return Parser::make_INT(loc_); }
"is"      { return Parser::make_IS(loc_); }
"of"      { return Parser::make_OF(loc_); }
{IDENT}   {
            auto ident_table = driver.get_ident_table();
            return Parser::make_IDENT(ident_table->GetCode(yytext), loc_);
          }
{NUMBER}  {
            try {
              return Parser::make_NUMBER(std::stoll(yytext), loc_);
            } catch (const std::logic_error& e) {
              throw Parser::syntax_error(loc_, e.what());
            }
          }
.         {
            const auto msg = "unexpected character: " + std::string{yytext};
            throw Parser::syntax_error(loc_, msg);
          }

%%
```

Синтаксический анализ

Файл `parser.y`:

```
%require "3.8.2"
%language "c++"
%skeleton "lalr1.cc"

%header
%locations

%define api.location.file "location.h"
%define api.namespace {fmt}
%define api.parser.class {Parser}
%define api.token.constructor
%define api.token.prefix {TOKEN_}
%define api.token.raw
%define api.value.automove
%define api.value.type variant
```

```
%define parse.assert
%define parse.error detailed
%define parse.trace
%define parse.lac full

%parse-param {Scanner& scanner}

%param {Driver& driver}

%code requires {

#include "ast.h"

namespace fmt {

class Driver;
class Scanner;

}  // namespace fmt

}

%code top {

#include <sstream>
#include <memory>

#include "driver.h"

#define yylex scanner.Get

}

%token
  <std::size_t>
    IDENT "identifier"
    NUMBER "number"

%token
  CASE "case"
  END  "end"
  INT  "int"
  IS   "is"
  OF   "of"

  EQUALS      "="
```

```
  COMMA       ","
  SEMICOLON   ";"
  COLON_COLON "::"

  LEFT_PARENTHESIS     "("
  RIGHT_PARENTHESIS    ")"
  LEFT_CURLY_BRACKET   "{"
  RIGHT_CURLY_BRACKET  "}"
  LEFT_SQUARE_BRACKET  "["
  RIGHT_SQUARE_BRACKET "]"

%right
  CONS_OP
  COLON ":"

%left
  ADD_OP
  PLUS  "+"
  MINUS "-"

%left
  MUL_OP
  STAR  "*"
  SLASH "/"

%precedence
  FUNC_CALL

%nterm
  <std::unique_ptr<Func>> func
  <std::vector<std::unique_ptr<Func>>> funcs
  <std::unique_ptr<FuncType>> func_type
  <std::unique_ptr<FuncBody>> func_body

  <std::unique_ptr<IType>> type
  <std::vector<std::unique_ptr<IType>>>
    tuple_type_content
    tuple_type_items
  <std::unique_ptr<ElementaryType>> elementary_type
  <std::unique_ptr<ListType>> list_type
  <std::unique_ptr<TupleType>> tuple_type

  <std::unique_ptr<Statement>> statement
  <std::vector<std::unique_ptr<Statement>>> statements

  <std::unique_ptr<IPattern>> pattern
```

```
  <std::unique_ptr<PatternList>> pattern_list
  <std::unique_ptr<PatternTuple>> pattern_tuple
  <std::vector<std::unique_ptr<IPattern>>>
    pattern_list_content
    pattern_list_items
    pattern_tuple_content
    pattern_tuple_items

  <std::unique_ptr<IResult>>
    result
    cons_expr
    cons_term
    arithm_expr
    arithm_term
    func_arg
  <std::unique_ptr<ResultList>> result_list
  <std::unique_ptr<ResultTuple>> result_tuple
  <std::unique_ptr<FuncCall>> func_call
  <std::unique_ptr<CaseExpr>> case_expr
  <std::vector<std::unique_ptr<IResult>>>
    result_list_content
    result_list_items
    result_tuple_content
    result_tuple_items

  <std::unique_ptr<Ident>> ident
  <std::unique_ptr<IConst>> const
  <Op>
    cons_op
    add_op
    mul_op

%%

program:
  funcs
  {
    driver.set_program(std::make_shared<Program>($1));
  }

funcs:
  funcs func
  {
    $$ = $1;
    $$.push_back($2);
  }
```

```
| %empty
  {
  }

func:
  IDENT func_type IS func_body END
  {
    $$ = std::make_unique<Func>($2, $4, $1);
  }

func_type:
  type "::" type
  {
    $$ = std::make_unique<FuncType>($1, $3);
  }

type:
  elementary_type
  {
    $$ = $1;
  }
| list_type
  {
    $$ = $1;
  }
| tuple_type
  {
    $$ = $1;
  }

elementary_type:
  INT
  {
    $$ = std::make_unique<ElementaryType>(ElementaryType::Kind::kInt);
  }

list_type:
  STAR type
  {
    $$ = std::make_unique<ListType>($2);
  }

tuple_type:
  "(" tuple_type_content ")"
  {
    $$ = std::make_unique<TupleType>($2);
```

```
  }

tuple_type_content:
  tuple_type_items
| %empty
  {
  }

tuple_type_items:
  type
  {
    $$.push_back($1);
  }
| tuple_type_items "," type
  {
    $$ = $1;
    $$.push_back($3);
  }

func_body:
  statements
  {
    $$ = std::make_unique<FuncBody>($1);
  }

statements:
  statement
  {
    $$.push_back($1);
  }
| statements ";" statement
  {
    $$ = $1;
    $$.push_back($3);
  }

statement:
  pattern "=" result
  {
    $$ = std::make_unique<Statement>($1, $3);
  }

pattern:
  ident
  {
    $$ = $1;
```

```
  }
| const
  {
    $$ = $1;
  }
| pattern_list
  {
    $$ = $1;
  }
| pattern_tuple
  {
    $$ = $1;
  }
| "[" pattern "]"
  {
    $$ = $2;
  }
| pattern cons_op pattern %prec CONS_OP
  {
    $$ = std::make_unique<PatternBinary>($1, $3, $2);
  }

pattern_list:
  "{" pattern_list_content "}"
  {
    $$ = std::make_unique<PatternList>($2);
  }

pattern_list_content:
  %empty
  {
  }
| pattern_list_items

pattern_list_items:
  pattern
  {
    $$.push_back($1);
  }
| pattern_list_items "," pattern
  {
    $$ = $1;
    $$.push_back($3);
  }

pattern_tuple:
```

```
  "(" pattern_tuple_content ")"
  {
    $$ = std::make_unique<PatternTuple>($2);
  }

pattern_tuple_content:
  %empty
  {
  }
| pattern_tuple_items

pattern_tuple_items:
  pattern
  {
    $$.push_back($1);
  }
| pattern_tuple_items "," pattern
  {
    $$ = $1;
    $$.push_back($3);
  }

result:
  case_expr
  {
    $$ = $1;
  }
| cons_expr

case_expr:
  CASE IDENT OF statements END
  {
    $$ = std::make_unique<CaseExpr>($4, $2);
  }

cons_expr:
  cons_term
| cons_expr cons_op cons_expr %prec CONS_OP
  {
    $$ = std::make_unique<ResultBinary>($1, $3, $2);
  }

cons_term:
  arithm_expr
| result_list
  {
```

```
    $$ = $1;
  }
| result_tuple
  {
    $$ = $1;
  }

arithm_expr:
  arithm_term
| "[" arithm_expr "]"
  {
    $$ = $2;
  }
| arithm_expr add_op arithm_expr %prec ADD_OP
  {
    $$ = std::make_unique<ResultBinary>($1, $3, $2);
  }
| arithm_expr mul_op arithm_expr %prec MUL_OP
  {
    $$ = std::make_unique<ResultBinary>($1, $3, $2);
  }

arithm_term:
  ident
  {
    $$ = $1;
  }
| const
  {
    $$ = $1;
  }
| func_call
  {
    $$ = $1;
  }

func_call:
  IDENT func_arg
  {
    $$ = std::make_unique<FuncCall>($2, $1);
  }

func_arg:
  arithm_term
| result_list
  {
```

```
    $$ = $1;
  }
| result_tuple
  {
    $$ = $1;
  }
| "[" cons_expr "]"
  {
    $$ = $2;
  }

result_list:
  "{" result_list_content "}"
  {
    $$ = std::make_unique<ResultList>($2);
  }

result_list_content:
  %empty
  {
  }
| result_list_items

result_list_items:
  cons_expr
  {
    $$.push_back($1);
  }
| result_list_items "," cons_expr
  {
    $$ = $1;
    $$.push_back($3);
  }

result_tuple:
  "(" result_tuple_content ")"
  {
    $$ = std::make_unique<ResultTuple>($2);
  }

result_tuple_content:
  %empty
  {
  }
| result_tuple_items
```

```
result_tuple_items:
  cons_expr
  {
    $$.push_back($1);
  }
| result_tuple_items "," cons_expr
  {
    $$ = $1;
    $$.push_back($3);
  }

ident:
  IDENT
  {
    $$ = std::make_unique<Ident>($1);
  }

const:
  NUMBER
  {
    $$ = std::make_unique<IntConst>($1);
  }

cons_op:
  COLON
  {
    $$ = Op::kCons;
  }

add_op:
  PLUS
  {
    $$ = Op::kAdd;
  }
| MINUS
  {
    $$ = Op::kSub;
  }

mul_op:
  STAR
  {
    $$ = Op::kMul;
  }
| SLASH
  {
```

```
    $$ = Op::kDiv;
  }

%%

void fmt::Parser::error(const location_type& loc, const std::string& msg) {
  throw syntax_error(loc, msg);
}
```

Семантический анализ

Файл `visitor.h`:

```
#pragma once

namespace fmt {

class Program;
class Func;
class FuncType;
class ElementaryType;
class ListType;
class TupleType;
class FuncBody;
class Statement;
class PatternList;
class PatternTuple;
class PatternBinary;
class ResultList;
class ResultTuple;
class ResultBinary;
class FuncCall;
class CaseExpr;
class Ident;
class IntConst;

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

  virtual void Visit(const Program& program) = 0;
  virtual void Visit(const Func& func) = 0;
  virtual void Visit(const FuncType& func_type) = 0;
  virtual void Visit(const ElementaryType& elementary_type) = 0;
  virtual void Visit(const ListType& list_type) = 0;
  virtual void Visit(const TupleType& tuple_type) = 0;
```

```
  virtual void Visit(const FuncBody& func_body) = 0;
  virtual void Visit(const Statement& statement) = 0;
  virtual void Visit(const PatternList& pattern_list) = 0;
  virtual void Visit(const PatternTuple& pattern_tuple) = 0;
  virtual void Visit(const PatternBinary& pattern_binary) = 0;
  virtual void Visit(const ResultList& result_list) = 0;
  virtual void Visit(const ResultTuple& result_tuple) = 0;
  virtual void Visit(const ResultBinary& result_binary) = 0;
  virtual void Visit(const FuncCall& func_call) = 0;
  virtual void Visit(const CaseExpr& case_expr) = 0;
  virtual void Visit(const Ident& ident) = 0;
  virtual void Visit(const IntConst& int_const) = 0;
};

}  // namespace fmt
```

Файл `formatter.h`:

```
#pragma once

#include <iterator>
#include <memory>
#include <ostream>
#include <sstream>

#include "ident_table.h"
#include "visitor.h"

namespace fmt {

// The object is supposed to be disposable.
class Formatter final : public IVisitor {
  std::shared_ptr<const IdentTable> ident_table_;
  std::ostringstream oss_;
  std::string current_indent_ = "";

  static constexpr std::string_view kIndent = "  ";

 public:
  Formatter(std::shared_ptr<const IdentTable>&& ident_table) noexcept
      : ident_table_(std::move(ident_table)) {}

  std::string ToString() const;

  void Visit(const Program& program) override;
  void Visit(const Func& func) override;
  void Visit(const FuncType& func_type) override;
```

```
  void Visit(const ElementaryType& elementary_type) override;
  void Visit(const ListType& list_type) override;
  void Visit(const TupleType& tuple_type) override;
  void Visit(const FuncBody& func_body) override;
  void Visit(const Statement& statement) override;
  void Visit(const PatternList& pattern_list) override;
  void Visit(const PatternTuple& pattern_tuple) override;
  void Visit(const PatternBinary& pattern_binary) override;
  void Visit(const ResultList& result_list) override;
  void Visit(const ResultTuple& result_tuple) override;
  void Visit(const ResultBinary& result_binary) override;
  void Visit(const FuncCall& func_call) override;
  void Visit(const CaseExpr& case_expr) override;
  void Visit(const Ident& ident) override;
  void Visit(const IntConst& int_const) override;

 private:
  using StatementIter = std::vector<std::unique_ptr<Statement>>::const_iterator;
  void FormatStatements(StatementIter b, const StatementIter e);

  template <typename Iter>
  requires std::input_iterator<Iter>
  void FormatContainer(Iter first, const Iter last, const std::string& start,
                       const std::string& end,
                       const std::string& delimiter = ", ");

  std::ostream& BeginOfLine();
  void IndentIncrease();
  void IndentDecrease();
  void IndentIncreaseLn();
  void IndentDecreaseLn();
};

}  // namespace fmt
```

Файл `formatter.cc`:

```
#include "formatter.h"

#include <cassert>
#include <iostream>

#include "ast.h"

namespace fmt {

std::string Formatter::ToString() const { return oss_.str(); }
```

```
void Formatter::Visit(const Program& program) {
  if (program.FuncsCbegin() == program.FuncsCend()) {
    return;
  }

  const auto e_prev = program.FuncsCend() - 1;
  for (auto b = program.FuncsCbegin(); b != e_prev; ++b) {
    b->get()->Accept(*this);
    BeginOfLine() << "\n\n";
  }

  e_prev->get()->Accept(*this);
}

void Formatter::Visit(const Func& func) {
  BeginOfLine() << ident_table_->At(func.get_ident_code()) << ' ';
  func.get_type().Accept(*this);
  oss_ << " is";

  IndentIncreaseLn();
  func.get_body().Accept(*this);
  IndentDecreaseLn();

  BeginOfLine() << "end";
}

void Formatter::Visit(const FuncType& func_type) {
  func_type.get_in().Accept(*this);
  oss_ << " :: ";
  func_type.get_out().Accept(*this);
}

void Formatter::Visit(const ElementaryType& elementary_type) {
  oss_ << elementary_type.get_kind();
}

void Formatter::Visit(const ListType& list_type) {
  oss_ << "*";
  list_type.get_type().Accept(*this);
}

void Formatter::Visit(const TupleType& tuple_type) {
  FormatContainer(tuple_type.TypesCbegin(), tuple_type.TypesCend(), "(", ")");
}
```

```
void Formatter::Visit(const FuncBody& func_body) {
  FormatStatements(func_body.StatementsCbegin(), func_body.StatementsCend());
}

void Formatter::Visit(const Statement& statement) {
  BeginOfLine();
  statement.get_pattern().Accept(*this);
  oss_ << " = ";
  statement.get_result().Accept(*this);
}

void Formatter::Visit(const PatternList& list) {
  FormatContainer(list.PatternsCbegin(), list.PatternsCend(), "{", "}");
}

void Formatter::Visit(const PatternTuple& tuple) {
  FormatContainer(tuple.PatternsCbegin(), tuple.PatternsCend(), "(", ")");
}

void Formatter::Visit(const PatternBinary& pattern_binary) {
  pattern_binary.get_lhs().Accept(*this);
  oss_ << ' ' << pattern_binary.get_op() << ' ';
  pattern_binary.get_rhs().Accept(*this);
}

void Formatter::Visit(const ResultList& list) {
  FormatContainer(list.ResultsCbegin(), list.ResultsCend(), "{", "}");
}

void Formatter::Visit(const ResultTuple& tuple) {
  FormatContainer(tuple.ResultsCbegin(), tuple.ResultsCend(), "(", ")");
}

void Formatter::Visit(const ResultBinary& result_binary) {
  result_binary.get_lhs().Accept(*this);
  oss_ << ' ' << result_binary.get_op() << ' ';
  result_binary.get_rhs().Accept(*this);
}

void Formatter::Visit(const FuncCall& func_call) {
  oss_ << ident_table_->At(func_call.get_ident_code()) << ' ';
  func_call.get_arg().Accept(*this);
}

void Formatter::Visit(const CaseExpr& case_expr) {
  IndentIncreaseLn();
```

```
  BeginOfLine() << "case " << ident_table_->At(case_expr.get_ident_code())
                << " of";

  IndentIncreaseLn();
  FormatStatements(case_expr.StatementsCbegin(), case_expr.StatementsCend());
  IndentDecreaseLn();

  BeginOfLine() << "end";
  IndentDecrease();
}

void Formatter::Visit(const Ident& ident) {
  oss_ << ident_table_->At(ident.get_code());
}

void Formatter::Visit(const IntConst& int_const) {
  oss_ << int_const.get_value();
}

template <typename Iter>
requires std::input_iterator<Iter>
void Formatter::FormatContainer(Iter first, const Iter last,
                                const std::string& start,
                                const std::string& end,
                                const std::string& delimiter) {
  oss_ << start;
  if (first == last) {
    oss_ << end;
    return;
  }

  const auto e_prev = last - 1;
  for (; first != e_prev; ++first) {
    first->get()->Accept(*this);
    oss_ << delimiter;
  }

  e_prev->get()->Accept(*this);
  oss_ << end;
}

void Formatter::FormatStatements(Formatter::StatementIter first,
                                 const Formatter::StatementIter last) {
  assert(first != last);

  const auto last_prev = last - 1;
```

```
  for (; first != last_prev; ++first) {
    first->get()->Accept(*this);
    oss_ << ";\n";
  }

  last_prev->get()->Accept(*this);
}

std::ostream& Formatter::BeginOfLine() { return oss_ << current_indent_; }

void Formatter::IndentIncrease() { current_indent_.append(kIndent); }

void Formatter::IndentDecrease() {
  static constexpr auto kIndentSize = kIndent.size();

  const auto cend = current_indent_.cend();
  current_indent_.erase(cend - kIndentSize, cend);
}

void Formatter::IndentIncreaseLn() {
  IndentIncrease();
  oss_ << '\n';
}

void Formatter::IndentDecreaseLn() {
  IndentDecrease();
  oss_ << '\n';
}

}  // namespace fmt
```

Тестирование

На входе программа из индивидуального варианта. Результат преобразования:

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

cart_prod (*int, *int) :: *(int, int) is
  (xs, ys) =
    case xs of
      x : xs = append (bind (x, ys), cart_prod (xs, ys));
      {} = {}
```

```
    end
end

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

append (*(int, int), *(int, int)) :: *(int, int) is
  (x : xs, ys) = x : append (xs, ys);
  ({}, ys) = ys
end

flat **int :: *int is
  x : xs : xss = x : flat xs : xss;
  {} : xss = flat xss;
  {} = {}
end

sum *int :: int is
  x : xs = x + sum xs;
  {} = 0
end

polynom (int, *int) :: int is
  (x, {}) = 0;
  (x, coef : coefs) = polynom (x, coefs) * x + coef
end

polynom1111 int :: int is
  x = polynom (x, {1, 1, 1, 1})
end
```

Вывод

В результате выполнения лабораторной работы я приобрёл навыки использования генератора синтаксических анализаторов bison.

Мне было интересно изучить и попробовать на практике возможности flex & bison, идиоматичные для языка C++. В соответствии с ними, например, и лексер, и парсер представляются экземплярами соответствующих классов, инкапсулирующих всю информацию о разборе, и потому анализаторы в C++

по умолчанию реентерантные. Также при работе с C++ bison предлагает множество полезных опций: `api.value.automove` для автоматического перемещения значений, `api.value.type` с возможностью использования `std::variant`-подобных объектов заместо “голых” `union`, `api.namespace` для указания `namespace`, содержащего парсер, и т.д. Работа с AST и генерация форматированного кода осуществляется посредством паттерна Visitor, что кажется удачным решением.