“Лабораторная работа 3.2 «Форматтер исходных текстов»”

18 июня 2025 г.

Александр Старовойтов, ИУ9-61Б

Цель работы

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

Индивидуальный вариант

Статически типизированный функциональный язык программирования:

```
-- Объединение двух списков
zip(xs : [int], ys : [int]) : [(int, int)] =
  if null(xs) or null(ys) then
    []
  else
    cons((car(xs), car(ys)), zip(cdr(xs), cdr(ys)));

-- Декартово произведение
cart_prod(xs : [int], ys : [int]) : [(int, int)] =
  if null(xs) then
    []
  else
    append(bind(car(xs), ys), cart_prod(cdr(xs), ys));

bind(x : int, ys : [int]) : [(int, int)] =
  if null(ys) then
    []
  else
    cons((x, car(ys)), bind(x, cdr(xs)));

-- Конкатенация списков пар
append(xs : [(int, int)], ys : [(int, int)]) : [(int, int)] =
  if null(xs) then
```

```
    ys
  else
    cons(car(xs), append(cdr(xs), ys);

-- Расплющивание вложенного списка
flat(xss : [[int]]) : [int] =
  if null(xss) then
    []
  else
    append(car(xss), flat(cdr(xss)));

-- Сумма элементов списка
sum(xs : [int]) : int =
  if null(xs) then
    0
  else
   car(xs) + sum(cdr(xs));

-- Вычисление полинома по схеме Горнера
polynom(x : int, coefs : [int]) : int =
  if null(coefs) then
    0
  else
    polynom(x, cdr(coefs)) * x + car(coefs);

-- Вычисление полинома x³+x²+x+1
polynom1111(x : int) : int = polynom(x, [1, 1, 1, 1]);
```

Реализация

```
#include <cstdint>
#include <iostream>

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

std::int32_t main(std::int32_t _, char* argv[]) {
  bool trace_parsing = true;
  bool trace_scanning = true;
  stewkk::lab11::Driver driver{trace_scanning, trace_parsing};
  driver.Parse(argv[1]);

  std::cout << stewkk::lab11::Formatter(driver.get_ident_table())
                   .Format(driver.get_program());
}
```

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

%header
%locations

%define api.location.file "location.h"
%define api.namespace {stewkk::lab11}
%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 stewkk::lab11 {

class Driver;
class Scanner;

}  // namespace stewkk::lab11

}

%code top {

#include <sstream>
#include <memory>

#include "driver.h"

#define yylex scanner.Get
```

```
}

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

%token
  IF   "if"
  ELSE "else"
  INT  "int"
  THEN "then"
  NULL "null"
  CONS "cons"
  CAR  "car"
  CDR  "cdr"
  OR   "or"

  EQUALS      "="
  COMMA       ","
  SEMICOLON   ";"
  COLON ":"

  LEFT_PARENTHESIS     "("
  RIGHT_PARENTHESIS    ")"
  LEFT_SQUARE_BRACKET  "["
  RIGHT_SQUARE_BRACKET "]"

%left
  ADD_OP
  PLUS  "+"
  MINUS "-"

%left
  MUL_OP
  STAR  "*"
  SLASH "/"

%precedence
  FUNC_CALL

%nterm
  <Func> func
  <std::vector<Func>> funcs
  <FuncType> func_type
```

```
  <FuncBody> func_body
  <std::vector<Arg>> args
  <Arg> arg

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

  <Statement> statement
  <IfStatement> if
  <BoolExpr> bool_expr
  <Expr> expr
  <Call> call
  <std::vector<Expr>> call_args
  <Callee> callee
  <std::vector<std::unique_ptr<Expr>>> list_elements
  <std::vector<std::unique_ptr<Expr>>> list_elements_tail

  <Ident> ident
  <Const> const
  <Op>
    add_op
    mul_op
  <Builtin>
    car_op
    cdr_op
    cons_op

%%

program:
  funcs
  {
    driver.set_program(std::move($1));
  }

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

```
  {
  }

func:
  IDENT func_type EQUALS func_body SEMICOLON
  {
    $$ = Func($1, $2, $4);
  }

func_type:
  "(" args ")" COLON type
  {
    $$ = FuncType($2, $5);
  }

args:
  args "," arg
  {
    $$ = $1;
    $$.push_back($3);
  }
| arg
  {
    $$.push_back($1);
  }

arg:
  IDENT ":" type
  {
    $$ = Arg($1, $3);
  }

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

```
elementary_type:
  INT
  {
    $$ = ElementaryType(ElementaryType::Kind::kInt);
  }

list_type:
  "[" type "]"
  {
    $$ = ListType(std::make_unique<Type>($2));
  }

tuple_type:
  "(" tuple_type_items ")"
  {
    $$ = TupleType($2);
  }

tuple_type_items:
  type
  {
    $$.push_back(std::make_unique<Type>($1));
  }
| tuple_type_items "," type
  {
    $$ = $1;
    $$.push_back(std::make_unique<Type>($3));
  }

func_body:
  statement
  {
    $$ = FuncBody($1);
  }

statement:
  if
  {
    $$ = Statement($1);
  }
| expr
  {
    $$ = Statement($1);
  }

if:
```

```
  "if" bool_expr "then" expr "else" expr
  {
    $$ = IfStatement($2, $4, $6);
  }

bool_expr:
  bool_expr "or" bool_expr
  {
    $$ = OrExpr(std::make_unique<BoolExpr>($1), std::make_unique<BoolExpr>($3));
  }
| expr
  {
    $$ = $1;
  }
| "null" "(" expr ")"
  {
    $$ = NullExpr($3);
  }

expr:
  call
  {
    $$ = $1;
  }
| ident
  {
    $$ = $1;
  }
| const
  {
    $$ = $1;
  }
| expr add_op expr
  {
    $$ = BinaryExpr(std::make_unique<Expr>($1), std::make_unique<Expr>($3), $2);
  }
| expr mul_op expr
  {
    $$ = BinaryExpr(std::make_unique<Expr>($1), std::make_unique<Expr>($3), $2);
  }
| "[" list_elements "]"
  {
    $$ = ListExpr($2);
  }
| "(" list_elements ")"
  {
```

```
    $$ = TupleExpr($2);
  }

list_elements:
  list_elements_tail expr
  {
    $$ = $1;
    $$.push_back(std::make_unique<Expr>($2));
  }
| %empty
  {
  }

list_elements_tail:
  list_elements_tail expr ","
  {
    $$ = $1;
    $$.push_back(std::make_unique<Expr>($2));
  }
| %empty
  {
  }


call:
  callee "(" call_args ")"
  {
    $$ = Call($1, $3);
  }

call_args:
  call_args "," expr
  {
    $$ = $1;
    $$.push_back($3);
  }
| expr
  {
    $$.push_back($1);
  }

callee:
  ident
  {
    $$ = $1;
  }
```

```
| car_op
  {
    $$ = $1;
  }
| cdr_op
  {
    $$ = $1;
  }
| cons_op
  {
    $$ = $1;
  }

ident:
  IDENT
  {
    $$ = Ident($1);
  }

const:
  NUMBER
  {
    $$ = IntConst($1);
  }

car_op:
  CAR
  {
    $$ = Builtin::kCar;
  }

cdr_op:
  CDR
  {
    $$ = Builtin::kCdr;
  }

cons_op:
  CONS
  {
    $$ = Builtin::kCons;
  }

add_op:
  PLUS
  {
```

```
    $$ = Op::kAdd;
  }
| MINUS
  {
    $$ = Op::kSub;
  }

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

%%

namespace stewkk::lab11 {

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

}  // namespace stewkk::lab11
```

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

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

#define YY_USER_ACTION loc_.columns(yyleng); \
                                        printf("%s", yytext);

using stewkk::lab11::Parser;
%}

%option c++
%option yyclass="stewkk::lab11::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_LEFT_PARENTHESIS(loc_); }
")"        { return Parser::make_RIGHT_PARENTHESIS(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_); }
"if"       { return Parser::make_IF(loc_); }
"else"     { return Parser::make_ELSE(loc_); }
"int"      { return Parser::make_INT(loc_); }
"then"     { return Parser::make_THEN(loc_); }
"null"     { return Parser::make_NULL(loc_); }
"cons"     { return Parser::make_CONS(loc_); }
"car"      { return Parser::make_CAR(loc_); }
"cdr"      { return Parser::make_CDR(loc_); }
"or"       { return Parser::make_OR(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);
```

```
          }

%%
#pragma once

#include <iostream>
#include <ostream>

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

#undef YY_DECL
#define YY_DECL stewkk::lab11::Parser::symbol_type stewkk::lab11::Scanner::Get(stewkk::lab11

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

namespace stewkk::lab11 {

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 stewkk::lab11
#pragma once

#include <string>
#include <unordered_map>
#include <vector>

namespace stewkk::lab11 {

class IdentTable {
  std::unordered_map<std::string, std::size_t> codes_;
```

```
  std::vector<std::string> names_;

 public:
  std::size_t GetCode(const std::string& name);
  std::string At(const std::size_t code) const;
};

}  // namespace stewkk::lab11
```

```
#include "ident_table.h"

#include <cassert>
#include <iostream>

namespace stewkk::lab11 {

std::size_t IdentTable::GetCode(const std::string& name) {
  if (const auto it = codes_.find(name); it != codes_.cend()) {
    return it->second;
  }

  const auto code = names_.size();
  codes_[name] = code;
  names_.push_back(name);
  return code;
}

std::string IdentTable::At(const std::size_t code) const {
  assert(code < names_.size());
  return names_.at(code);
}

}  // namespace stewkk::lab11
```

```
#pragma once

#include <memory>
#include <vector>
#include <variant>
#include <cstdint>

namespace stewkk::lab11 {

enum class Op {
  kAdd,
  kSub,
  kMul,
```

```
  kDiv,
};

enum class Builtin {
  kCons,
  kCar,
  kCdr,
};

struct ElementaryType {
  enum class Kind {
    kInt,
  };
  Kind kind;
};

std::string ToString(enum ElementaryType::Kind kind);
std::string ToString(enum Op op);
std::string ToString(enum Builtin builtin);

struct ListType;
struct TupleType;

using Type = std::variant<ElementaryType, ListType, TupleType>;

struct ListType {
  std::unique_ptr<Type> type;
};

struct TupleType {
  std::vector<std::unique_ptr<Type>> types;
};

struct Arg {
  std::size_t ident_code;
  Type type;
};

struct FuncType {
  std::vector<Arg> args;
  Type result;
};

struct IfStatement;

struct Call;
```

```
struct Ident {
  std::size_t code;
};

struct IntConst {
  std::int64_t value;
};

using Const = std::variant<IntConst>;

struct BinaryExpr;
struct ListExpr;
struct TupleExpr;

using Expr = std::variant<Call, Ident, Const, BinaryExpr, ListExpr, TupleExpr>;

struct ListExpr {
  std::vector<std::unique_ptr<Expr>> elements;
};

struct TupleExpr {
  std::vector<std::unique_ptr<Expr>> elements;
};

struct BinaryExpr {
  std::unique_ptr<Expr> lhs;
  std::unique_ptr<Expr> rhs;
  Op op;
};


using Callee = std::variant<Ident, Builtin>;

struct Call {
  Callee callee;
  std::vector<Expr> args;
};

using Statement = std::variant<IfStatement, Expr>;

struct OrExpr;

struct NullExpr {
  Expr inner;
};
```

```
using BoolExpr = std::variant<OrExpr, Expr, NullExpr>;

struct OrExpr {
  std::unique_ptr<BoolExpr> lhs;
  std::unique_ptr<BoolExpr> rhs;
};

struct IfStatement {
  BoolExpr condition;
  Expr then_expr;
  Expr else_expr;
};

struct FuncBody {
  Statement statement;
};

struct Func {
  std::size_t ident_code;
  FuncType type;
  FuncBody body;
};

using Program = std::vector<Func>;

}  // namespace stewkk::lab11
```

```
#include "ast.h"

namespace stewkk::lab11 {

std::string ToString(enum ElementaryType::Kind kind) {
  switch (kind) {
      case ElementaryType::Kind::kInt:
          return "int";
  }
  throw std::logic_error{"unreachable"};
}

std::string ToString(enum Op op) {
  switch (op) {
      case Op::kAdd:
          return "+";
      case Op::kSub:
          return "-";
```

```
      case Op::kMul:
          return "*";
      case Op::kDiv:
          return "/";
  }
  throw std::logic_error{"unreachable"};
}

std::string ToString(enum Builtin builtin) {
  switch (builtin) {
      case Builtin::kCar:
          return "car";
      case Builtin::kCdr:
          return "cdr";
      case Builtin::kCons:
          return "cons";
  }
  throw std::logic_error{"unreachable"};
}

}  // namespace stewkk::lab11
```

```
#pragma once

#include <optional>

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

namespace stewkk::lab11 {

class Driver final {
  bool trace_scanning_, trace_parsing_;
  std::optional<Program> program_;
  IdentTable table_{};

 public:
  Driver(bool trace_scanning, bool trace_parsing);

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

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

```
    assert(program_.has_value());
    return program_.value();
  }

  IdentTable& get_ident_table() noexcept { return table_; }
};

}  // namespace stewkk::lab11
```

```
#include "driver.h"

#include <fstream>

namespace stewkk::lab11 {

Driver::Driver(bool trace_scanning, bool trace_parsing)
    : trace_scanning_(trace_scanning), trace_parsing_(trace_parsing) {}

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

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

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

  parser.parse();
}

}  // namespace stewkk::lab11
```

```
#pragma once

#include <string>

#include "ast.h"
#include "ident_table.h"
#include "output.h"

namespace stewkk::lab11 {

class Formatter {
public:
```

```
    explicit Formatter(const IdentTable& ident_table);

    std::string Format(const Program& program);
private:
    void Format(const Func& func);
    void Format(const std::vector<Arg>& args);
    void Format(const FuncBody& body);
    void Format(const Expr& expr);
    void Format(const BoolExpr& expr);
    void Format(const Const& expr);
    void Format(const Callee& expr);

    void FormatSingleLine(const Type& type);

    Output output_;
    std::size_t limit_ = 80; // TODO: прокидывать через конструктор
    const IdentTable& ident_table_;
};

}  // namespace stewkk::lab11
```

```
#include "formatter.h"

namespace stewkk::lab11 {

Formatter::Formatter(const IdentTable& ident_table) : ident_table_(ident_table) {}

std::string Formatter::Format(const Program& program) {
    for (const auto& func : program) {
        Format(func);
        output_.ResetIdent();
        output_.NewLine();
        output_.NewLine();
    }
    return output_.GetStr();
}

void Formatter::Format(const Func& func) {
    const auto func_name = ident_table_.At(func.ident_code);
    output_.Put(func_name);
    if (output_.GetPrefixLength() + 1 > limit_) {
        output_.IncreaseIdent();
        output_.NewLine();
    }
    output_.Put("(");
    Format(func.type.args);
```

```
    output_.Put(")");

    output_.CheckpointLine();

    output_.Put(" : ");
    FormatSingleLine(func.type.result);
    output_.Put(" =");

    // TODO: если превысили длину префикса, восстанавливаемся и переносим на следующую строк

    Format(func.body);
    output_.Put(";");
}

void Formatter::Format(const FuncBody& body) {
    struct StatementFormatter {
        void operator()(const IfStatement& stmt) {
            output.IncreaseIdent();
            output.NewLine();
            output.Put("if ");
            auto if_ident = output.GetIdent();
            formatter.Format(stmt.condition);
            output.Put(" then");
            output.IncreaseIdent();
            output.NewLine();
            formatter.Format(stmt.then_expr);
            output.SetIdent(std::move(if_ident));
            output.NewLine();
            output.Put("else");
            output.IncreaseIdent();
            output.IncreaseIdent();
            output.NewLine();
            formatter.Format(stmt.else_expr);
        }
        void operator()(const Expr& stmt) {
            output.Put(" ");
            formatter.Format(stmt);
        }

        Formatter& formatter;
        Output& output;
    };

    std::visit(StatementFormatter{*this, output_}, body.statement);
}
```

```
void Formatter::Format(const Expr& expr) {
    struct ExprFormatter {
        void operator()(const Call& stmt) {
            formatter.Format(stmt.callee);
            output.Put("(");
            bool is_first = true;
            for (const auto& arg : stmt.args) {
                if (!is_first) {
                    output.Put(", ");
                }
                formatter.Format(arg);
                is_first = false;
            }
            output.Put(")");
        }
        void operator()(const Ident& stmt) {
            output.Put(formatter.ident_table_.At(stmt.code));
        }
        void operator()(const Const& stmt) {
            formatter.Format(stmt);
        }
        void operator()(const BinaryExpr& stmt) {
            formatter.Format(*stmt.lhs.get());
            output.Put(" ");
            output.Put(ToString(stmt.op));
            output.Put(" ");
            formatter.Format(*stmt.rhs.get());
        }
        void operator()(const ListExpr& stmt) {
            output.Put("[");
            bool is_first = true;
            for (const auto& elem : stmt.elements) {
                if (!is_first) {
                    output.Put(", ");
                }
                formatter.Format(*elem.get());
                is_first = false;
            }
            output.Put("]");
        }
        void operator()(const TupleExpr& stmt) {
            output.Put("(");
            bool is_first = true;
            for (const auto& elem : stmt.elements) {
                if (!is_first) {
                    output.Put(", ");
```

```
                }
                formatter.Format(*elem.get());
                is_first = false;
            }
            output.Put(")");
        }

        Formatter& formatter;
        Output& output;
    };

    std::visit(ExprFormatter{*this, output_}, expr);
}

void Formatter::Format(const BoolExpr& expr) {
    struct BoolExprFormatter {
        void operator()(const Expr& stmt) {
            formatter.Format(stmt);
        }
        void operator()(const OrExpr& stmt) {
            formatter.Format(*stmt.lhs.get());
            output.Put(" or ");
            formatter.Format(*stmt.rhs.get());
        }
        void operator()(const NullExpr& stmt) {
            output.Put("null(");
            formatter.Format(stmt.inner);
            output.Put(")");
        }

        Formatter& formatter;
        Output& output;
    };

    std::visit(BoolExprFormatter{*this, output_}, expr);
}

void Formatter::Format(const std::vector<Arg> &args) {
  bool is_first = true;
  for (const auto& arg : args) {
      if (!is_first) {
          output_.Put(", ");
      }
      const auto arg_name = ident_table_.At(arg.ident_code);
      output_.Put(arg_name);
      output_.Put(" : ");
```

```
      FormatSingleLine(arg.type);
      is_first = false;
  }
  // TODO: добавить форматирование каждого аргумента на отдельной строке
}

void Formatter::FormatSingleLine(const Type& type) {
    struct SingleLineFormatter {
        void operator()(const ElementaryType& type) {
            output.Put(ToString(type.kind));
        }
        void operator()(const ListType& type) {
            output.Put("[");
            formatter.FormatSingleLine(*type.type.get());
            output.Put("]");
        }
        void operator()(const TupleType& type) {
            output.Put("(");
            bool is_first = true;
            for (const auto& type : type.types) {
                if (!is_first) {
                    output.Put(", ");
                }
                formatter.FormatSingleLine(*type.get());
                is_first = false;
            }
            output.Put(")");
        }

        Formatter& formatter;
        Output& output;
    };

    std::visit(SingleLineFormatter{*this, output_}, type);
}

void Formatter::Format(const Const& expr) {
    struct ConstFormatter {
        void operator()(const IntConst& const_val) {
            output.Put(std::to_string(const_val.value));
        }

        Formatter& formatter;
        Output& output;
    };
```

```
    std::visit(ConstFormatter{*this, output_}, expr);
}

void Formatter::Format(const Callee& expr) {
    struct CalleeFormatter {
        void operator()(const Ident& ident) {
            output.Put(formatter.ident_table_.At(ident.code));
        }
        void operator()(const Builtin& builtin) {
            output.Put(ToString(builtin));
        }

        Formatter& formatter;
        Output& output;
    };

    std::visit(CalleeFormatter{*this, output_}, expr);
}

}  // namespace stewkk::lab11
```

```
#pragma once

#include <string>
#include <sstream>

namespace stewkk::lab11 {

class Output {
  public:
    std::string GetStr() const;

    void NewLine();
    void ClearLine();
    void IncreaseIdent();
    void ResetIdent();
    std::string GetIdent();
    void SetIdent(std::string ident);
    void Put(const std::string& str);
    std::size_t GetPrefixLength() const;
    void CheckpointLine();
    void RestoreCheckpoint();

  private:

    std::ostringstream ident_;
```

```
    std::ostringstream line_;
    std::ostringstream checkpoint_;
    std::ostringstream out_;
};

}  // namespace stewkk::lab11
```

```
#include "output.h"

namespace stewkk::lab11 {

std::string Output::GetStr() const {
    return out_.str();
}

void Output::NewLine() {
    out_ << line_.view() << '\n';
    ClearLine();
    line_ << ident_.view();
}

void Output::ClearLine() {
    line_ = std::ostringstream{};
}

void Output::IncreaseIdent() {
    ident_ << "  ";
}

void Output::ResetIdent() {
    ident_ = std::ostringstream{};
}

void Output::Put(const std::string& str) {
    line_ << str;
}

std::size_t Output::GetPrefixLength() const {
    return line_.view().size();
}

void Output::CheckpointLine() {
    checkpoint_ = std::ostringstream{line_.str()};
}

void Output::RestoreCheckpoint() {
```

```
    line_ = std::move(checkpoint_);
}

std::string Output::GetIdent() {
    return ident_.str();
}

void Output::SetIdent(std::string ident) {
    ident_ = std::ostringstream{std::move(ident)};
}

}  // namespace stewkk::lab11
```

Тестирование

Входные данные

```
-- 1
zipblablabla(xs : [int], ys : [int])
  : [(int, int)] = 0;

-- 2
zip(
  xs : [int],
  ys : [int]
) : [(int, int)] = 0;

-- 3
zip(
  xsblablablabla
    : [int],
  ys
    : [(
        int,
        int,
        int,
        int,
        int,
        int,
        int
      )]
) : [(int, int)] = 0;
```

Вывод на `stdout`

```
polynom1111(x : int) : int = polynom(x, [1, 1, 1, 1]);
```

```
zipblablabla(xs : [int], ys : [int]) : [(int, int)] = 0;

zip(xs : [int], ys : [int]) : [(int, int)] = 0;

zip(xsblablablabla : [int], ys : [(int, int, int, int, int, int, int)]) : [(int, int)] = 0;
```

Вывод

В рамках данной работы я приобрел навыки использования генератора синтаксических анализаторов bison.