

Лабораторная работа № 2.2 «Абстрактные синтаксические деревья»

27 марта 2024 г.

Илья Афанасьев, ИУ9-61Б

Цель работы

Целью данной работы является получение навыков составления грамматик и проектирования синтаксических деревьев.

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));
  (xs, 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);
  (xs, 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  $x^3+x^2+x+1$ 
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 ($y\ * \text{ и } /$ он выше, чем $y\ + \text{ и } -$) 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

Абстрактный синтаксис

```
Program → Funcs
Funcs → ε | Funcs Func
Func → IDENT FuncType IS FuncBody END

FuncType → Type :: Type
Type → ElementaryType | ListType | TupleType
ElementaryType → INT
ListType → * Type
TupleType → ( TupleTypeContent )
TupleTypeContent → ε | TupleTypeItems
TupleTypeItems → Type | TupleTypeItems , Type

FuncBody → Statements
Statements → Statement | Statements ; Statement
Statement → Pattern = Result

Pattern → IDENT
        | Const
        | PatternList
        | PatternTuple
        | [ Pattern ]
        | Pattern PatternBinaryOp Pattern
PatternBinaryOp → :
Const → INT_CONST

PatternList → { PatternListContent }
PatternListContent → ε | PatternListItems
PatternListItems = PatternListItem | PatternListItems , PatternListItem
PatternListItem = Pattern

PatternTuple → ( PatternTupleContent )
PatternTupleContent → ε | PatternTupleItems
PatternTupleItems → PatternTupleItem | PatternTupleItems , PatternTupleItem
PatternTupleItem → Pattern

Result → IDENT
        | Const
        | ResultList
```

```

    | ResultTuple
    | [ Result ]
    | FuncCall
    | Result ResultBinaryOp Result
ResultBinaryOp → : | + | - | * | /
FuncCall → IDENT FuncArg
FuncArg → Result

ResultList → { ResultListContent }
ResultListContent → ε | ResultListItems
ResultListItems → ResultListItem | ResultListItems , ResultListItem
ResultListItem → Result

ResultTuple → ( ResultTupleContent )
ResultTupleContent → ε | ResultTupleItems
ResultTupleItems → ResultTupleItem | ResultTupleItems , ResultTupleItem
ResultTupeItem → Result

```

Лексическая структура и конкретный синтаксис

```

Program → Funcs
Funcs → ε | Funcs Func
Func → IDENT FuncType IS FuncBody END

FuncType → Type :: Type
Type → ElementaryType | ListType | TupleType
ElementaryType → INT
ListType → * Type
TupleType → ( TupleTypeContent )
TupleTypeContent → ε | TupleTypeItems
TupleTypeItems → Type | TupleTypeItems , Type

FuncBody → Statements
Statements → Statement | Statements ; Statement
Statement → Pattern = Result

Pattern → PatternUnit | PatternUnit ConsOp Pattern
ConsOp → :
PatternUnit → IDENT | Const | PatternList | PatternTuple | [ Pattern ]
Const → INT_CONST

PatternList → { PatternListContent }
PatternListContent → ε | PatternListItems
PatternListItems = PatternListItem | PatternListItems , PatternListItem
PatternListItem = Pattern

```

```

PatternTuple → ( PatternTupleContent )
PatternTupleContent → ε | PatternTupleItems
PatternTupleItems → PatternTupleItem | PatternTupleItems , PatternTupleItem
PatternTupleItem → Pattern

Result → ResultUnit | ResultUnit ConsOp Result
ResultUnit → Expr | ResultList | ResultTuple

Expr → Term | Expr AddOp Term
AddOp → + | -
Term → Factor | Term MulOp Factor
MulOp → * | /
Factor → Atom | [ Expr ]
Atom → IDENT | Const | FuncCall
FuncCall → IDENT FuncArg
FuncArg → Atom | ResultList | ResultTuple | [ Result ]

ResultList → { ResultListContent }
ResultListContent → ε | ResultListItems
ResultListItems → ResultListItem | ResultListItems , ResultListItem
ResultListItem → Result

ResultTuple → ( ResultTupleContent )
ResultTupleContent → ε | ResultTupleItems
ResultTupleItems → ResultTupleItem | ResultTupleItems , ResultTupleItem
ResultTupeItem → Result

```

Программная реализация

```

import abc
import enum
import parser_edsl as pe
import sys
import typing
from dataclasses import dataclass
from pprint import pprint

```

```

class Type(abc.ABC):
    pass

```

```

class TypeEnum(enum.Enum):
    Int = 'int'

```

```

@dataclass
class ElementaryType(Type):
    type_: TypeEnum

@dataclass
class TupleType(Type):
    types: list[Type]

@dataclass
class ListType(Type):
    types: list[Type]

class Pattern(abc.ABC):
    pass

@dataclass
class PatternBinary(Pattern):
    lhs: Pattern
    op: str
    rhs: Pattern

@dataclass
class PatternList(Pattern):
    patterns: list[Pattern]

@dataclass
class PatternTuple(Pattern):
    patterns: list[Pattern]

class Result(abc.ABC):
    pass

@dataclass
class ResultBinary(Result):
    lhs: Result
    op: str
    rhs: Result

```

```
@dataclass
class ResultList(Result):
    results: list[Result]
```

```
@dataclass
class ResultTuple(Result):
    results: list[Result]
```

```
@dataclass
class VarExpr(Pattern, Result):
    varname: str
```

```
@dataclass
class ConstExpr(Pattern, Result):
    value: typing.Any
    type_: TypeEnum
```

```
@dataclass
class FuncCallExpr(Result):
    funcname: str
    argument: Result
```

```
@dataclass
class Statement:
    pattern: Pattern
    result: Result
```

```
@dataclass
class FuncType:
    input_: Type
    output: Type
```

```
@dataclass
class Func:
    name: str
    type_: FuncType
    body: list[Statement]
```

```

@dataclass
class Program:
    funcs: list[Func]

IDENT = pe.Terminal('IDENT', '[A-Za-z_][A-Za-z_0-9]*', str)
INT_CONST = pe.Terminal('INT_CONST', '[0-9]+', int)

def make_keyword(image):
    return pe.Terminal(image, image, lambda _: None, priority=10)

KW_IS, KW_END, KW_INT = map(make_keyword, ['is', 'end', 'int'])

NProgram = pe.NonTerminal('Program')
NFuncs = pe.NonTerminal('Funcs')
NFunc = pe.NonTerminal('Func')

NFuncType = pe.NonTerminal('FuncType')
NType = pe.NonTerminal('Type')
NElementaryType = pe.NonTerminal('ElementaryType')
NListType = pe.NonTerminal('ListType')
NTupleType = pe.NonTerminal('TupleType')
NTupleTypeContent = pe.NonTerminal('TupleTypeContent')
NTupleTypeItems = pe.NonTerminal('TupleTypeContent')

NFuncBody = pe.NonTerminal('FuncBody')
NStatements = pe.NonTerminal('Statements')
NStatement = pe.NonTerminal('Statement')

NPattern = pe.NonTerminal('Pattern')
NConsOp = pe.NonTerminal('ConsOp')
NPatternUnit = pe.NonTerminal('PatternUnit')
NConst = pe.NonTerminal('Const')

NPatternList = pe.NonTerminal('PatternList')
NPatternListContent = pe.NonTerminal('PatternListContent')
NPatternListItems = pe.NonTerminal('PatternListItems')
NPatternListItem = pe.NonTerminal('PatternListItem')

NPatternTuple = pe.NonTerminal('PatternTuple')
NPatternTupleContent = pe.NonTerminal('PatternTupleContent')
NPatternTupleItems = pe.NonTerminal('PatternTupleItems')
NPatternTupleItem = pe.NonTerminal('PatternTupleItem')

```



```

NResult = pe.NonTerminal('Result')
NResultUnit = pe.NonTerminal('ResultUnit')

NExpr = pe.NonTerminal('Expr')
NAddOp = pe.NonTerminal('AddOp')
NTerm = pe.NonTerminal('Term')
NMulOp = pe.NonTerminal('MulOp')
NFactor = pe.NonTerminal('Factor')
NAtom = pe.NonTerminal('Atom')
NFuncCall = pe.NonTerminal('FuncCall')
NFuncArg = pe.NonTerminal('FuncArg')

NResultList = pe.NonTerminal('ResultList')
NResultListContent = pe.NonTerminal('ResultListContent')
NResultListItems = pe.NonTerminal('ResultListItems')
NResultListItem = pe.NonTerminal('ResultListItem')

NResultTuple = pe.NonTerminal('ResultTuple')
NResultTupleContent = pe.NonTerminal('ResultTupleContent')
NResultTupleItems = pe.NonTerminal('ResultTupleItems')
NResultTupleItem = pe.NonTerminal('ResultTupleItem')

NProgram |= NFuncs, Program

NFuncs |= lambda: []
NFuncs |= NFuncs, NFunc, lambda fs, f: fs + [f]

NFunc |= IDENT, NFuncType, KW_IS, NFuncBody, KW_END, Func

NFuncType |= NType, '::', NType, FuncType

NType |= NElementaryType
NType |= NListType
NType |= NTupleType

NElementaryType |= KW_INT, lambda: ElementaryType(TypeEnum.Int)

NListType |= '*', NType, ListType

NTupleType |= '(', NTupleTypeContent, ')', TupleType

NTupleTypeContent |= lambda: []
NTupleTypeContent |= NTupleTypeItems

NTupleTypeItems |= NType, lambda t: [t]

```

```

NTupleTypeItems |= NTupleTypeItems, ',', NType, lambda ts, t: ts + [t]

NFuncBody |= NStatements

NStatements |= NStatement, lambda s: [s]
NStatements |= NStatements, ';', NStatement, lambda ss, s: ss + [s]

NStatement |= NPattern, '=', NResult, Statement

NPattern |= NPatternUnit
NPattern |= NPatternUnit, NConsOp, NPattern, PatternBinary

NConsOp |= ':', lambda: ':'

NPatternUnit |= IDENT, VarExpr
NPatternUnit |= NConst
NPatternUnit |= NPatternList,
NPatternUnit |= NPatternTuple,
NPatternUnit |= '[', NPattern, ']',

NConst |= INT_CONST, lambda value: ConstExpr(value, TypeEnum.Int)

NPatternList |= '{', NPatternListContent, '}', PatternList

NPatternListContent |= lambda: []
NPatternListContent |= NPatternListItems

NPatternListItems |= NPatternListItem, lambda pli: [pli]
NPatternListItems |= NPatternListItems, ',', NPatternListItem, \
    lambda plis, pli: plis + [pli]

NPatternListItem |= NPattern

NPatternTuple |= '(', NPatternTupleContent, ')', PatternTuple

NPatternTupleContent |= lambda: []
NPatternTupleContent |= NPatternTupleItems

NPatternTupleItems |= NPatternTupleItem, lambda pti: [pti]
NPatternTupleItems |= NPatternTupleItems, ',', NPatternTupleItem, \
    lambda ptis, pti: ptis + [pti]

NPatternTupleItem |= NPattern

NResult |= NResultUnit
NResult |= NResultUnit, NConsOp, NResult, ResultBinary

```

```

NResultUnit |= NExpr
NResultUnit |= NResultList,
NResultUnit |= NResultTuple,

NExpr |= NTerm
NExpr |= NExpr, NAddOp, NTerm, ResultBinary

NAddOp |= '+', lambda: '+'
NAddOp |= '-', lambda: '-'

NTerm |= NFactor
NTerm |= NTerm, NMulOp, NFactor, ResultBinary

NMulOp |= '*', lambda: '*'
NMulOp |= '/', lambda: '/'

NFactor |= NAtom
NFactor |= '[', NExpr, ']'

NAtom |= IDENT, VarExpr
NAtom |= NConst
NAtom |= NFuncCall

NFuncCall |= IDENT, NFuncArg, FuncCallExpr

NFuncArg |= NAtom
NFuncArg |= NResultList
NFuncArg |= NResultTuple
NFuncArg |= '[', NResult, ']'

NResultList |= '{', NResultListContent, '}', ResultList

NResultListContent |= lambda: []
NResultListContent |= NResultListItems

NResultListItems |= NResultListItem, lambda rli: [rli]
NResultListItems |= NResultListItems, ',', NResultListItem, \
    lambda rlis, rli: rlis + [rli]

NResultListItem |= NResult

NResultTuple |= '(', NResultTupleContent, ')', ResultTuple

NResultTupleContent |= lambda: []
NResultTupleContent |= NResultTupleItems

```

```

NResultTupleItems |= NResultTupleItem, lambda rti: [rti]
NResultTupleItems |= NResultTupleItems, ',', NResultTupleItem, \
    lambda rtis, rti: rtis + [rti]

NResultTupleItem |= NResult

if __name__ == "__main__":
    p = pe.Parser(NProgram)
    assert p.is_lalr_one()

    p.add_skipped_domain('\\\\s')
    p.add_skipped_domain('@[^\n]*')

    for filename in sys.argv[1:]:
        try:
            with open(filename) as f:
                pass
                tree = p.parse(f.read())
                pprint(tree)
        except pe.Error as e:
            print(f'Ошибка {e.pos}: {e.message}')
        except Exception as e:
            print(e)

```

Тестирование

Входные данные

```

@ 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  $x^3+x^2+x+1$ 
polynom1111 int :: int is x = polynom (x, {1, 1, 1, 1}) end

```

Вывод на stdout

```

Program(funcs=[Func(name='zip',
  type_=FuncType(input_=TupleType(types=[ListType(types=ElementaryType(type_=<TypeEnum.Int: 'int'>),
    ListType(types=ElementaryType(type_=<TypeEnum.Int: 'int'>))
    output=ListType(types=TupleType(types=[ElementaryType(type_=<TypeEnum.Int: 'int'>),
      ElementaryType(type_=<TypeEnum.Int: 'int'>))
    body=[Statement(pattern=PatternTuple(patterns=[PatternBinary(lhs=VarExpr(varname='x'),
      op=':',
      rhs=VarExpr(varname='xs')),
      PatternBinary(lhs=VarExpr(varname='y'),
      op=':',
      rhs=VarExpr(varname='ys'))]),
    result=ResultBinary(lhs=ResultTuple(results=[VarExpr(varname='x'),
      VarExpr(varname='y')]),
      op=':',
      rhs=FuncCallExpr(funcname='zip',

```

```

        argument=ResultTuple(results=[VarExpr(varname='xs'),
        VarExpr(varname='ys')]),
    Statement(pattern=PatternTuple(patterns=[VarExpr(varname='xs'),
        VarExpr(varname='ys')]),
        result=ResultList(results=[]))),
    Func(name='cart_prod',
type_=FuncType(input_=TupleType(types=[ListType(types=ElementaryType(type_=<TypeEnum.Int: 'int')>),
        ListType(types=ElementaryType(type_=<TypeEnum.Int: 'int')>),
        output=ListType(types=TupleType(types=[ElementaryType(type_=<TypeEnum.Int: 'int')>),
        ElementaryType(type_=<TypeEnum.Int: 'int')>)),
body=[Statement(pattern=PatternTuple(patterns=[PatternBinary(lhs=VarExpr(varname='xs'),
        op=':',
        rhs=VarExpr(varname='ys'))),
        VarExpr(varname='ys')]),
        result=FuncCallExpr(funcname='append',
        argument=ResultTuple(results=[FuncCallExpr(funcname='bind',
        argument=ResultTuple(results=[VarExpr(varname='xs'),
        VarExpr(varname='ys')]),
        FuncCallExpr(funcname='cart_prod',
        argument=ResultTuple(results=[VarExpr(varname='xs'),
        VarExpr(varname='ys')]),
        Statement(pattern=PatternTuple(patterns=[PatternList(patterns=[]),
        VarExpr(varname='ys')]),
        result=ResultList(results=[]))),
    Func(name='bind',
type_=FuncType(input_=TupleType(types=[ElementaryType(type_=<TypeEnum.Int: 'int')>),
        ListType(types=ElementaryType(type_=<TypeEnum.Int: 'int')>),
        output=ListType(types=TupleType(types=[ElementaryType(type_=<TypeEnum.Int: 'int')>),
        ElementaryType(type_=<TypeEnum.Int: 'int')>)),
body=[Statement(pattern=PatternTuple(patterns=[VarExpr(varname='x'),
        PatternList(patterns=[])]),
        result=ResultList(results=[])),
        Statement(pattern=PatternTuple(patterns=[VarExpr(varname='x'),
        PatternBinary(lhs=VarExpr(varname='y'),
        op=':',
        rhs=VarExpr(varname='ys'))]),
        result=ResultBinary(lhs=ResultTuple(results=[VarExpr(varname='x'),
        VarExpr(varname='y')]),
        op=':',
        rhs=FuncCallExpr(funcname='bind',
        argument=ResultTuple(results=[VarExpr(varname='xs'),
        VarExpr(varname='ys')]),
    Func(name='append',
type_=FuncType(input_=TupleType(types=[ListType(types=TupleType(types=[ElementaryType(type_=<TypeEnum.Int: 'int')>),
        ElementaryType(type_=<TypeEnum.Int: 'int')>),
        ListType(types=TupleType(types=[ElementaryType(type_=<TypeEnum.Int: 'int')>),
        ElementaryType(type_=<TypeEnum.Int: 'int')>)),

```

```

        ElementaryType(type_=<TypeEnum.Int: 'int'>)),
    output=ListType(types=TupleType(types=[ElementaryType(type_=<TypeEnum.Int: 'int'>)),
        ElementaryType(type_=<TypeEnum.Int: 'int'>)),
    body=[Statement(pattern=PatternTuple(patterns=[PatternBinary(lhs=VarExpr(varname='x'),
        op=':',
        rhs=VarExpr(varname='xs'))),
        VarExpr(varname='ys')]),
        result=ResultBinary(lhs=VarExpr(varname='x'),
        op=':',
        rhs=FuncCallExpr(funcname='append',
        argument=ResultTuple(results=[VarExpr(varname='xs'),
        VarExpr(varname='ys')])),
        Statement(pattern=PatternTuple(patterns=[PatternList(patterns=[]),
        VarExpr(varname='ys')]),
        result=VarExpr(varname='ys'))]),
    Func(name='flat',
    type_=FuncType(input_=ListType(types=ListType(types=ElementaryType(type_=<TypeEnum.Int: 'int'>)),
        output=ListType(types=ElementaryType(type_=<TypeEnum.Int: 'int'>))),
    body=[Statement(pattern=PatternBinary(lhs=PatternBinary(lhs=VarExpr(varname='x'),
        op=':',
        rhs=VarExpr(varname='xs'))),
        op=':',
        rhs=VarExpr(varname='xss')),
        result=ResultBinary(lhs=VarExpr(varname='x'),
        op=':',
        rhs=FuncCallExpr(funcname='flat',
        argument=ResultBinary(lhs=VarExpr(varname='x'),
        op=':',
        rhs=VarExpr(varname='xss')))),
        Statement(pattern=PatternBinary(lhs=PatternList(patterns=[]),
        op=':',
        rhs=VarExpr(varname='xss')),
        result=FuncCallExpr(funcname='flat',
        argument=VarExpr(varname='xss'))),
        Statement(pattern=PatternList(patterns=[]),
        result=ResultList(results=[]))]),
    Func(name='sum',
    type_=FuncType(input_=ListType(types=ElementaryType(type_=<TypeEnum.Int: 'int'>)),
        output=ElementaryType(type_=<TypeEnum.Int: 'int'>)),
    body=[Statement(pattern=PatternBinary(lhs=VarExpr(varname='x'),
        op=':',
        rhs=VarExpr(varname='xs')),
        result=ResultBinary(lhs=VarExpr(varname='x'),
        op='+',
        rhs=FuncCallExpr(funcname='sum',
        argument=VarExpr(varname='xs')))),

```

```

Statement(pattern=PatternList(patterns=[]),
          result=ConstExpr(value=0,
                             type_=<TypeEnum.Int: 'int'>)),
Func(name='polynom',
type_=FuncType(input_=TupleType(types=[ElementaryType(type_=<TypeEnum.Int: 'int'>,
ListType(types=ElementaryType(type_=<TypeEnum.Int: 'int'>),
output=ElementaryType(type_=<TypeEnum.Int: 'int'>)),
body=[Statement(pattern=PatternTuple(patterns=[VarExpr(varname='x'),
PatternList(patterns=[])]),
result=ConstExpr(value=0,
type_=<TypeEnum.Int: 'int'>)),
Statement(pattern=PatternTuple(patterns=[VarExpr(varname='x'),
PatternBinary(lhs=VarExpr(varname='coef'),
op=':',
rhs=VarExpr(varname='coefs'))]),
result=ResultBinary(lhs=ResultBinary(lhs=FuncCallExpr(funcname='poly
argument=ResultTuple(results=[VarExpr(varname='x'),
op='*',
rhs=VarExpr(varname='x')),
op='+',
rhs=VarExpr(varname='coef')))]),
Func(name='polynom1111',
type_=FuncType(input_=ElementaryType(type_=<TypeEnum.Int: 'int'>),
output=ElementaryType(type_=<TypeEnum.Int: 'int'>)),
body=[Statement(pattern=VarExpr(varname='x'),
result=FuncCallExpr(funcname='polynom',
argument=ResultTuple(results=[VarExpr(varname='x'),
ResultList(results=[ConstExpr(value=1,
type_=<TypeEnum.Int: 'int'>),
ConstExpr(value=1,
type_=<TypeEnum.Int: 'int'>),
ConstExpr(value=1,
type_=<TypeEnum.Int: 'int'>),
ConstExpr(value=1,
type_=<TypeEnum.Int: 'int'>)])))]),

```

Вывод

В результате выполнения лабораторной работы я получил навыки составления грамматик и проектирования синтаксических деревьев. При выполнении я осознал полезность построения и абстрактной, и конкретной грамматик, что до сих пор смешивалось в моём представлении: первое позволяет “свободно” описать язык и выделить основные узлы АСТ, например, без учёта приоритета операций, а второе используется при разборе с учётом всех деталей. Такое

разделение хорошо сказывается на проектировании: например, при исправлении ошибки в реализации конкретной грамматики я совсем не изменял часть, относящуюся к абстрактной грамматике. Также было интересно поработать с `parser_edsl`. Библиотека позволяет просто и красиво порождать АСТ.