Лабораторная работа № 3.3 «Семантический анализ»

5 июня 2024 г.

Илья Афанасьев, ИУ9-61Б

Цель работы

Целью данной работы является получение навыков выполнения семантического анализа.

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
```

Семантический анализ:

- В программе не может быть двух функций с одинаковыми именами.
- В образцах не может быть одноимённых переменных.
- Образец и правая часть должны соответствовать заявленному типу.
- Тип формального и фактического параметров в вызовах функций совпадает.
- Если тип выражения x — T, а тип xs — *T, то тип x : xs — *T.
- Если выражения x1, x2, …, xN имеют тип T, то тип {x1, x2, …, xN} — *T.
- Выражение {} имеет тип «список элементов произвольного типа».

Реализация

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


class SemanticError(pe.Error):
    pass
```

```
class FunctionRedefinition(SemanticError):
    def __init__(self, pos, funcname):
        self.pos = pos
        self.funcname = funcname

    @property
    def message(self):
        return f'Переопределение функции {self.funcname}'


class UnknownFunction(SemanticError):
    def __init__(self, pos, funcname):
        self.pos = pos
        self.funcname = funcname

    @property
    def message(self):
        return f'Неизвестная функция {self.funcname}'


class RepeatedVariable(SemanticError):
    def __init__(self, pos, varname):
        self.pos = pos
        self.varname = varname

    @property
    def message(self):
        return f'Повторная переменная {self.varname} в образце'


class UnknownVariable(SemanticError):
    def __init__(self, pos, varname):
        self.pos = pos
        self.varname = varname

    @property
    def message(self):
        return f'Неизвестная переменная {self.varname} в правой части'


class TypeMismatch(SemanticError):
    def __init__(self, pos: pe.Position, expected: str):
        self.pos = pos
        self.expected = expected
    @property
    def message(self):
        return f'Ожидался объект типа {self.expected}'


class Op(enum.Enum):
    Cons = ':'
    Add = '+'
    Sub = '-'
    Mul = '*'
    Div = '/'


class Type(abc.ABC):
    @abc.abstractmethod
    def pretty(self):
        pass


class IntType(Type):
    def pretty(self):
        return 'int'


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

    def pretty(self):
        return f'({", ".join(map(lambda type_: type_.pretty(), self.types))})'


@dataclass
class ListType(Type):
    type_: Type

    def pretty(self):
        return f'*{self.type_.pretty()}'


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

```
class Pattern(abc.ABC):
    @abc.abstractmethod
    def check(self, expected_type, var_types):
        pass


@dataclass
class PatternEmptyList(Pattern):
    coord: pe.Position

    @pe.ExAction
    def create(attrs, coords, res_coord):
        lcb_coord, rcb_coord = coords
        return PatternEmptyList(lcb_coord)

    def check(self, expected_type, var_types):
        actual_type = ListType
        if actual_type != type(expected_type):
            raise TypeMismatch(self.coord, expected_type.pretty())


@dataclass
class PatternConst(Pattern):
    value: typing.Any
    value_coord: pe.Position
    type_: Type

    @staticmethod
    def create(type_):
        @pe.ExAction
        def action(attrs, coords, res_coord):
            value, = attrs
            value_coord, = coords
            return PatternConst(value, value_coord, type_)

        return action

    def check(self, expected_type, var_types):
        if self.type_ != expected_type:
            raise TypeMismatch(self.value_coord, expected_type.pretty())


@dataclass
class PatternVar(Pattern):
    name: str
    name_coord: pe.Position
    @pe.ExAction
    def create(attrs, coords, res_coord):
        name, = attrs
        name_coord, = coords
        return PatternVar(name, name_coord)

    def check(self, expected_type, var_types):
        if self.name in var_types:
            raise RepeatedVariable(self.name_coord, self.name)

        var_types[self.name] = expected_type


@dataclass
class PatternBinary(Pattern):
    lhs: Pattern
    op: Op
    op_coord: pe.Position
    rhs: Pattern

    @pe.ExAction
    def create_cons_with_empty_list(attrs, coords, res_coord):
        lhs, = attrs
        lhs_coord, = coords
        # NOTE: fictive op, empty list coord
        return PatternBinary(lhs, Op.Cons, lhs_coord, PatternEmptyList(lhs_coord))

    @pe.ExAction
    def create_cons(attrs, coords, res_coord):
        lhs, rhs = attrs
        lhs_coord, comma_coord, rhs_coord = coords
        # NOTE: fictive op coord
        return PatternBinary(lhs, Op.Cons, comma_coord, rhs)

    @pe.ExAction
    def create(attrs, coords, res_coord):
        lhs, op, rhs = attrs
        lhs_coord, op_coord, rhs_coord = coords
        return PatternBinary(lhs, op, op_coord, rhs)

    def check(self, expected_type, var_types):
        actual_type = ListType
        if actual_type != type(expected_type):
            raise TypeMismatch(self.op_coord, expected_type.pretty())
        self.lhs.check(expected_type.type_, var_types)
        self.rhs.check(expected_type, var_types)


@dataclass
class PatternTuple(Pattern):
    patterns: list[Pattern]
    patterns_coord: pe.Position

    @pe.ExAction
    def create(attrs, coords, res_coord):
        patterns, = attrs
        lp_coord, patterns_coord, rp_coord = coords
        return PatternTuple(patterns, patterns_coord)

    def check(self, expected_type, var_types):
        actual_type = TupleType
        if actual_type != type(expected_type) or \
           len(self.patterns) != len(expected_type.types):
            raise TypeMismatch(self.patterns_coord, expected_type.pretty())

        for pattern, expected_type in zip(self.patterns, expected_type.types):
            pattern.check(expected_type, var_types)


class Result(abc.ABC):
    @abc.abstractmethod
    def check(self, expected_type, func_types, var_types):
        pass


@dataclass
class ResultEmtpyList(Result):
    coord: pe.Position

    @pe.ExAction
    def create(attrs, coords, res_coord):
        lcb_coord, rcb_coord = coords
        return ResultEmtpyList(lcb_coord)

    def check(self, expected_type, func_types, var_types):
        actual_type = ListType
        if actual_type != type(expected_type):
            raise TypeMismatch(self.coord, expected_type.pretty())
```

```python
@dataclass
class ResultConst(Result):
    value: typing.Any
    value_coord: pe.Position
    type_: Type

    @staticmethod
    def create(type_):
        @pe.ExAction
        def action(attrs, coords, res_coord):
            value, = attrs
            value_coord, = coords
            return ResultConst(value, value_coord, type_)

        return action

    def check(self, expected_type, func_types, var_types):
        if self.type_ != expected_type:
            raise TypeMismatch(self.value_coord, expected_type.pretty())


@dataclass
class ResultVar(Result):
    name: str
    name_coord: pe.Position

    @pe.ExAction
    def create(attrs, coords, res_coord):
        name, = attrs
        name_coord, = coords
        return ResultVar(name, name_coord)

    def check(self, expected_type, func_types, var_types):
        if self.name not in var_types:
            raise UnknownVariable(self.name_coord, self.name)

        actual_type = var_types[self.name]
        if actual_type != expected_type:
            raise TypeMismatch(self.name_coord, expected_type.pretty())


@dataclass
class FuncCallExpr(Result):
    funcname: str
    funcname_coord: pe.Position
    argument: Result
    @pe.ExAction
    def create(attrs, coords, res_coord):
        funcname, argument = attrs
        funcname_coord, argument_coord = coords
        return FuncCallExpr(funcname, funcname_coord, argument)

    def check(self, expected_type, func_types, var_types):
        if self.funcname not in func_types:
            raise UnknownFunction(self.funcname_coord, self.funcname)

        func_type = func_types[self.funcname]
        if func_type.output != expected_type:
            raise TypeMismatch(self.funcname_coord, expected_type.pretty())

        self.argument.check(func_type.input_, func_types, var_types)


@dataclass
class ResultBinary(Result):
    lhs: Result
    op: Op
    op_coord: pe.Position
    rhs: Result

    @pe.ExAction
    def create_cons_with_empty_list(attrs, coords, res_coord):
        lhs, = attrs
        lhs_coord, = coords
        # NOTE: fictive op, empty list coords
        return ResultBinary(lhs, Op.Cons, lhs_coord, ResultEmtpyList(lhs_coord))

    @pe.ExAction
    def create_cons(attrs, coords, res_coord):
        lhs, rhs = attrs
        lhs_coord, comma_coord, rhs_coord = coords
        # NOTE: fictive op_coord
        return ResultBinary(lhs, Op.Cons, comma_coord, rhs)

    @pe.ExAction
    def create(attrs, coords, res_coord):
        lhs, op, rhs = attrs
        lhs_coord, op_coord, rhs_coord = coords
        return ResultBinary(lhs, op, op_coord, rhs)

    def check(self, expected_type, func_types, var_types):
        if self.op != Op.Cons:
            self.lhs.check(expected_type, func_types, var_types)
            self.rhs.check(expected_type, func_types, var_types)
            return

        actual_type = ListType
        if actual_type != type(expected_type):
            raise TypeMismatch(self.op_coord, expected_type.pretty())

        self.lhs.check(expected_type.type_, func_types, var_types)
        self.rhs.check(expected_type, func_types, var_types)


@dataclass
class ResultTuple(Result):
    results: list[Result]
    results_coord: pe.Position

    @pe.ExAction
    def create(attrs, coords, res_coord):
        results, = attrs
        lp_coord, results_coord, rp_coord = coords
        return ResultTuple(results, results_coord)

    def check(self, expected_type, func_types, var_types):
        actual_type = TupleType
        if actual_type != type(expected_type) or \
           len(self.results) != len(expected_type.types):
            raise TypeMismatch(self.results_coord, expected_type.pretty())

        for result, expected_type in zip(self.results, expected_type.types):
            result.check(expected_type, func_types, var_types)


@dataclass
class Sentence:
    pattern: Pattern
    result: Result

    def check(self, func_types: dict[str, FuncType], funcname: str):
        functype = func_types[funcname]

        var_types = {}
        self.pattern.check(functype.input_, var_types)

        self.result.check(functype.output, func_types, var_types)
```

```
@dataclass
class Func:
    name: str
    name_coord: pe.Position
    type_: FuncType
    body: list[Sentence]

    @pe.ExAction
    def create(attrs, coords, res_coord):
        name, func_type, func_body = attrs
        name_coord, type_coord, is_coord, body_coord, end_coord = coords
        return Func(name, name_coord, func_type, func_body)

    def check(self, func_types):
        for sentence in self.body:
            sentence.check(func_types, self.name)


@dataclass
class Program:
    funcs: list[Func]

    def check(self):
        funcs = {}
        for func in self.funcs:
            if func.name in funcs:
                raise FunctionRedefinition(func.name_coord, func.name)
            funcs[func.name] = func.type_

        for func in self.funcs:
            func.check(funcs)


INT_TYPE = IntType()

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')
NIntType = pe.NonTerminal('IntType')
NListType = pe.NonTerminal('ListType')
NTupleType = pe.NonTerminal('TupleType')
NTupleTypeContent = pe.NonTerminal('TupleTypeContent')
NTupleTypeItems = pe.NonTerminal('TupleTypeItems')
NTupleTypeItem = pe.NonTerminal('TupleTypeItem')

NFuncBody = pe.NonTerminal('FuncBody')
NSentences = pe.NonTerminal('Sentences')
NSentence = pe.NonTerminal('Sentence')

NPattern = pe.NonTerminal('Pattern')
NConsOp = pe.NonTerminal('ConsOp')
NPatternTerm = pe.NonTerminal('PatternTerm')

NPatternList = pe.NonTerminal('PatternList')
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')
NResultTerm = pe.NonTerminal('ResultTerm')

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')
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 xs, x: xs + [x]

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

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

NType |= NIntType
NType |= NListType
NType |= NTupleType

NIntType |= KW_INT, lambda: INT_TYPE

NListType |= '*', NType, ListType

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

NTupleTypeContent |= lambda: []
NTupleTypeContent |= NTupleTypeItems

NTupleTypeItems |= NTupleTypeItem, lambda x: [x]
NTupleTypeItems |= NTupleTypeItems, ',', NTupleTypeItem, lambda xs, x: xs + [x]

NTupleTypeItem |= NType

NFuncBody |= NSentences

NSentences |= NSentence, lambda x: [x]
NSentences |= NSentences, ';', NSentence, lambda xs, x: xs + [x]

NSentence |= NPattern, '=', NResult, Sentence

NPattern |= NPatternTerm
NPattern |= NPatternTerm, NConsOp, NPattern, PatternBinary.create

NConsOp |= ':', lambda: Op.Cons

NPatternTerm |= IDENT, PatternVar.create
NPatternTerm |= INT_CONST, PatternConst.create(INT_TYPE)
NPatternTerm |= NPatternList,
NPatternTerm |= NPatternTuple,
NPatternTerm |= '[', NPattern, ']',

NPatternList |= '{', '}', PatternEmptyList.create
NPatternList |= '{', NPatternListItems, '}'

NPatternListItems |= NPatternListItem, PatternBinary.create_cons_with_empty_list
NPatternListItems |= NPatternListItem, ',', NPatternListItems, PatternBinary.create_cons

NPatternListItem |= NPattern

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

NPatternTupleContent |= lambda: []
NPatternTupleContent |= NPatternTupleItems

NPatternTupleItems |= NPatternTupleItem, lambda x: [x]
NPatternTupleItems |= NPatternTupleItems, ',', NPatternTupleItem, lambda xs, x: xs + \
    [x]

NPatternTupleItem |= NPattern

NResult |= NResultTerm
NResult |= NResultTerm, NConsOp, NResult, ResultBinary.create

NResultTerm |= NExpr
NResultTerm |= NResultList,
NResultTerm |= NResultTuple,

NExpr |= NTerm
NExpr |= NExpr, NAddOp, NTerm, ResultBinary.create

NAddOp |= '+', lambda: Op.Add
NAddOp |= '-', lambda: Op.Sub

NTerm |= NFactor
NTerm |= NTerm, NMulOp, NFactor, ResultBinary.create

NMulOp |= '*', lambda: Op.Mul
NMulOp |= '/', lambda: Op.Div

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

```
NAtom |= IDENT, ResultVar.create
NAtom |= INT_CONST, ResultConst.create(INT_TYPE)
NAtom |= NFuncCall

NFuncCall |= IDENT, NFuncArg, FuncCallExpr.create

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

NResultList |= '{', '}', ResultEmtpyList.create
NResultList |= '{', NResultListItems, '}'

NResultListItems |= NResultListItem, ResultBinary.create_cons_with_empty_list
NResultListItems |= NResultListItem, ',', NResultListItems, ResultBinary.create_cons

NResultListItem |= NResult

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

NResultTupleContent |= lambda: []
NResultTupleContent |= NResultTupleItems

NResultTupleItems |= NResultTupleItem, lambda x: [x]
NResultTupleItems |= NResultTupleItems, ',', NResultTupleItem, lambda xs, x: xs + \
    [x]

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:
                tree = p.parse(f.read())
                tree.check()
                print('Семантических ошибок не найдено')
        except pe.Error as e:
            print(f'Ошибка {e.pos}: {e.message}')
```

Тестирование

На программе из индивидуального варианта анализатор, разумеется, выдаёт “Семантических ошибок не найдено”. Некоторые случаи с ошибкой:

- **Входные данные**:

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

  Вывод на `stdout`:

  ```
  Ошибка (8, 1)-(8, 8): Переопределение функции polynom
  ```

- **Входные данные**:

  ```
  @ Объединение двух списков
  zip (*int, *int) :: *(int, int) is
    @ В образце вместо ys указано использованное имя xs
    (x : xs, y : xs) = (x, y) : zip (xs, ys);
    (xs, ys) = {}
  end
  ```

  Вывод на `stdout`:

  ```
  Ошибка (4, 16)-(4, 18): Повторная переменная xs в образце
  ```

- **Входные данные**:

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

  Вывод на `stdout`:

  ```
  Ошибка (4, 30)-(4, 31): Ожидался объект типа *(int, int)
  ```

Вывод

В результате выполнения лабораторной работы я закрепил навыки проведения семантического анализа. Средствами библиотеки parser_edsl задача решается просто и красиво.