Лабораторная работа № 3.3 «Семантический анализ»

27 мая 2024 г.

Сергей Виленский, ИУ9-62Б

Цель работы

Целью данной работы является получение навыков выполнения семантического анализа.

Индивидуальный вариант

Объявления типов и констант в Паскале:

В `record`’е точка с запятой *разделяет* поля и после `case` дополнительный `end` не ставится. См. https://bernd-oppolzer.de/PascalReport.pdf, третья с конца страница.

```
Type
  Coords = Record x, y: INTEGER end;
Const
  MaxPoints = 100;
type
  CoordsVector = array 1..MaxPoints of Coords;

(* графический и текстовый дисплеи *)
const
  Heigh = 480;
  Width = 640;
  Lines = 24;
  Columns = 80;
type
  BaseColor = (red, green, blue, highlited);
  Color = set of BaseColor;
  GraphicScreen = array 1..Heigh of array 1..Width of Color;
  TextScreen = array 1..Lines of array 1..Columns of
    record
      Symbol : CHAR;
      SymColor : Color;
      BackColor : Color
    end;

{ определения токенов }
TYPE
  Domain = (Ident, IntNumber, RealNumber);
  Token = record
    fragment : record
      start, following : record
        row, col : INTEGER
      end
    end;
    case tokType : Domain of
      Ident : (
        name : array 1..32 of CHAR
      );
      IntNumber : (
        intval : INTEGER
      );
      RealNumber : (
        realval : REAL
      )
  end;

  Year = 1900..2050;

  List = record
    value : Token;
    next : ^List
  end;
```

Семантический анализ

Проверки: * Используемые идентификаторы должны быть определены выше по тексту. * Имена констант и типов находятся в общей области видимости и не должны повторяться. Перечислимые типы тоже определяют константы. * В записях не могут встречаться одноимённые поля. Результат: * Программа должна выводить на экран значения всех констант (значения констант перечислений нумеруются с нуля). * Для каждого типа должен вычисляться его объём. Считаем, что размеры целых чисел и перечислимых типов — 2 байта, вещественных чисел — 4 байта, размер указателя — 4 байта, размер множества определяется как количество байт, требуемых для его представления (каждый элемент множества — бит), размер case-части записи определяется как размер поля тега + размер наибольшего варианта.

Реализация

```python
import abc
import enum
import parser_edsl as pe
import sys
import re
import typing
from dataclasses import dataclass
from pprint import pprint
from json import dumps

@dataclass
class Identifier:
    name : str

class SemanticError(pe.Error): pass

@dataclass
class UnknownType(SemanticError):
    pos : typing.Any
    typename : Identifier

    @property
    def message(self):
        return f'Неопределенный тип {self.typename}'

@dataclass
class RepeatedType(SemanticError):
    pos : typing.Any
    typename : Identifier

    @property
    def message(self):
        return f'Повторное определение типа {self.typename}'

@dataclass
class UnknownConstant(SemanticError):
    pos : typing.Any
    constname : Identifier

    @property
    def message(self):
        return f'Неопределенная константа {self.constname}'

@dataclass
```

```
class RepeatedConstant(SemanticError):
    pos : typing.Any
    constname : Identifier

    @property
    def message(self):
        return f'Повторное определение константы {self.constname}'

@dataclass
class RepeatedField(SemanticError):
    pos : typing.Any
    fieldname : Identifier

    @property
    def message(self):
        return f'Повторное использование в записи поля {self.fieldname}'


# constant
class UnarSign(enum.Enum):
    Plus = 'PLUS'
    Minus = 'MINUS'

class ConstantIdentifier(Identifier): pass

@dataclass
class Constant(abc.ABC):
    @abc.abstractmethod
    def check(self, types, consts): pass
    @abc.abstractmethod
    def getValue(self, consts): pass

@dataclass
class SignedIdentifierConstant(Constant):
    unar_sign : UnarSign
    constant_identifier : ConstantIdentifier
    constant_identifier_coord : pe.Position
    @pe.ExAction
    def create(attrs, coords, res_coord):
        unar_sign, constant_identifier = attrs
        cunar_sign, cconstant_identifier = coords
        return SignedIdentifierConstant(
            unar_sign, constant_identifier, cconstant_identifier.start)

    def check(self, types, consts):
        if self.constant_identifier not in consts:
            raise UnknownConstant(self.constant_identifier_coord, self.constant_identifier)

    def getValue(self, consts):
        if self.unar_sign == UnarSign.Minus:
            signing = lambda x: -x
        else:
            signing = lambda x: x

        self.value = singing(consts[self.constant_identifier])
        return self.value

@dataclass
class UnsignedIdentifierConstant(Constant):
    constant_identifier : ConstantIdentifier
    constant_identifier_coord : pe.Position
    @pe.ExAction
    def create(attrs, coords, res_coord):
        constant_identifier, = attrs
        cconstant_identifier, = coords
        return UnsignedIdentifierConstant(
            constant_identifier, cconstant_identifier.start)

    def check(self, types, consts):
        if self.constant_identifier not in consts:
            raise UnknownConstant(self.constant_identifier_coord, self.constant_identifier)

    def getValue(self, consts):
        self.value = consts[self.constant_identifier]
        return self.value

@dataclass
class SignedNumberConstant(Constant):
    unar_sign : UnarSign
    unsingned_number : float

    def check(self, types, consts): pass

    def getValue(self, consts):
        if self.unar_sign == UnarSign.Minus:
            signing = lambda x: -x
        else:
            signing = lambda x: x

        self.value = signing(self.unsingned_number)
        return self.value
```

```
@dataclass
class UnsignedNumberConstant(Constant):
    unsingned_number : float

    def check(self, types, consts): pass

    def getValue(self, consts):
        self.value = self.unsingned_number
        return self.value

@dataclass
class CharacterConstant(Constant):
    char_sequence : str

    def check(self, types, consts): pass

    def getValue(self, consts):
        self.value = self.char_sequence
        return self.value

# simple type
class TypeIdentifier(Identifier): pass

@dataclass
class SimpleType(abc.ABC):
    @abc.abstractmethod
    def check(self, types, consts): pass
    @abc.abstractmethod
    def calcConsts(self, consts): pass
    @abc.abstractmethod
    def getTypeSize(self, types): pass
    @abc.abstractmethod
    def getValuesCount(self): pass

@dataclass
class DefaultSimpleType(SimpleType):
    type_identifier : TypeIdentifier
    type_identifier_coord : pe.Position
    @pe.ExAction
    def create(attrs, coords, res_coord):
        type_identifier, = attrs
        ctype_identifier, = coords
        return DefaultSimpleType(
            type_identifier, ctype_identifier.start)

    def check(self, types, consts):
        if self.type_identifier not in types:
            raise UnknownType(self.type_identifier_coord, self.type_identifier)

        self.actual_type = types[self.type_identifier]

    def calcConsts(self, consts): pass

    def getTypeSize(self, types):
        return types[self.type_identifier]

    def getValuesCount(self):
        return self.actual_type.getValuesCount()

@dataclass
class ListSimpleType(SimpleType):
    identifier_list : tuple[ConstantIdentifier]
    identifier_list_coord : pe.Position
    @pe.ExAction
    def create(attrs, coords, res_coord):
        identifier_list, = attrs
        copbr, cidentifier_list, cclbr = coords
        return ListSimpleType(
            identifier_list, cidentifier_list.start)

    def check(self, types, consts):
        for identifier in self.identifier_list:
            if identifier in consts:
                raise RepeatedConstant(self.identifier_list_coord, identifier)
            consts.append(identifier)

    def calcConsts(self, consts):
        for i, identifier in enumerate(self.identifier_list):
            consts[identifier] = i

    def getTypeSize(self, types):
        return 2

    def getValuesCount(self):
        return len(self.identifier_list)

@dataclass
class BoundedSimpleType(SimpleType):
    left_constant : Constant
    right_constant : Constant

    def check(self, types, consts):
```

```python
        self.left_constant.check(types, consts)
        self.right_constant.check(types, consts)

    def calcConsts(self, consts):
        self.left_constant.getValue(consts)
        self.right_constant.getValue(consts)

    def getTypeSize(self, types):
        const_val = self.left_constant.value

        if isinstance(const_val, str):
            return None
        if const_val % 1 == 0:
            return 2
        else:
            return 4

    def getValuesCount(self):
        return int(self.right_constant.value - self.left_constant.value) + 1

# type
@dataclass
class Type(abc.ABC):
    @abc.abstractmethod
    def check(self, types, consts): pass
    @abc.abstractmethod
    def calcConsts(self, consts): pass
    @abc.abstractmethod
    def getTypeSize(self, types): pass
    @abc.abstractmethod
    def getValuesCount(self): pass

@dataclass
class DefaultType(Type):
    simple_type : SimpleType

    def check(self, types, consts):
        self.simple_type.check(types, consts)

    def calcConsts(self, consts):
        self.simple_type.calcConsts(consts)

    def getTypeSize(self, types):
        return self.simple_type.getTypeSize(types)

    def getValuesCount(self):
        return self.simple_type.getValuesCount()

@dataclass
class RefType(Type):
    type_identifier : TypeIdentifier
    type_identifier_coord : pe.Position
    @pe.ExAction
    def create(attrs, coords, res_coord):
        type_identifier, = attrs
        cref_sym, ctype_identifier, = coords
        return RefType(
            type_identifier, ctype_identifier.start)

    def check(self, types, consts):
        if self.type_identifier not in types:
            raise UnknownType(self.type_identifier_coord, self.type_identifier)

    def calcConsts(self, consts): pass

    def getTypeSize(self, types):
        return 4

    def getValuesCount(self):
        return None

@dataclass
class PackedType(Type):
    simple_type : SimpleType

    def check(self, types, consts):
        self.simple_type.check(types, consts)

    def calcConsts(self, consts):
        self.simple_type.calcConsts(consts)

    def getTypeSize(self, types):
        return self.simple_type.getTypeSize(types)

    def getValuesCount(self):
        return None

@dataclass
class ArrayType(Type):
    simple_types : tuple[SimpleType]
    type : Type
    def check(self, types, consts):
        for simple_type in self.simple_types:
            simple_type.check(types, consts)
        self.type.check(types, consts)

    def calcConsts(self, consts):
        for simple_type in self.simple_types:
            simple_type.calcConsts(consts)
        self.type.calcConsts(consts)

    def getTypeSize(self, types):
        return sum(
            simple_type.getValuesCount()
                for simple_type in self.simple_types
            ) * self.type.getTypeSize(types)

    def getValuesCount(self):
        return None

@dataclass
class FileType(Type):
    type : Type

    def check(self, types, consts):
        self.type.check(types, consts)

    def calcConsts(self, consts):
        self.type.calcConsts(consts)

    def getTypeSize(self, types):
        return None

    def getValuesCount(self):
        return None

@dataclass
class SetType(Type):
    simple_type : SimpleType

    def check(self, types, consts):
        self.simple_type.check(types, consts)

    def calcConsts(self, consts):
        self.simple_type.calcConsts(consts)

    def getTypeSize(self, types):
        return (self.simple_type.getValuesCount() + 7) // 8

    def getValuesCount(self):
        return None

@dataclass
class RecordType(Type):
    class FieldList: pass

    field_list : FieldList

    def check(self, types, consts):
        self.field_list.check(types, consts, set())

    def calcConsts(self, consts):
        self.field_list.calcConsts(consts)

    def getTypeSize(self, types):
        return self.field_list.getTypeSize(types)

    def getValuesCount(self):
        return None

# field list
@dataclass
class IdentifierWithType:
    identifier_list : tuple[Identifier]
    identifier_list_coord : pe.Position
    type : Type
    @pe.ExAction
    def create(attrs, coords, res_coord):
        identifier_list, type_ = attrs
        cidentifier_list, csemicol, ctype = coords
        return IdentifierWithType(
            identifier_list, cidentifier_list.start, type_)

    def check(self, types, consts, case_vars):
        for field in self.identifier_list:
            if field in case_vars:
                raise RepeatedField(self.identifier_list_coord, field)
            case_vars.add(field)

        self.type.check(types, consts)

    def calcConsts(self, consts):
        self.type.calcConsts(consts)
    def getTypeSize(self, types):
        return len(self.identifier_list) * self.type.getTypeSize(types)

@dataclass
class CaseVariant:
    class FieldList: pass

    constant_list : tuple[Constant]
    constant_list_coord : pe.Position
    field_list : FieldList
    @pe.ExAction
    def create(attrs, coords, res_coord):
        constant_list, field_list = attrs
        cconstant_list, csemicol, copbr, cfield_list, cclbr = coords
        return CaseVariant(
            constant_list, cconstant_list.start, field_list)

    def check(self, types, consts, case_vars):
        for constant in self.constant_list:
            constant.check(types, consts)

    def calcConsts(self, consts):
        self.field_list.calcConsts(consts)

    def getTypeSize(self, types):
        return self.field_list.getTypeSize(types)

@dataclass
class CaseBlock:
    identifier : TypeIdentifier
    identifier_coord : pe.Position
    type_identifier : TypeIdentifier
    type_identifier_coord : pe.Position
    case_variant_sequence : tuple[CaseVariant]
    @pe.ExAction
    def create(attrs, coords, res_coord):
        identifier, type_identifier, case_variant_sequence = attrs
        (ccase, cidentifier, csemicol, ctype_identifier, cof,
            ccase_variant_sequence) = coords
        return CaseBlock(
            identifier, cidentifier.start, type_identifier, ctype_identifier.start,
            case_variant_sequence)

    def check(self, types, consts, case_vars):
        if self.identifier in case_vars:
            raise RepeatedField(self.identifier_coord, self.identifier)

        if self.type_identifier not in types:
            raise UnknownType(self.type_identifier_coord, self.type_identifier)

        for case_variant in self.case_variant_sequence:
            case_variant.check(types, consts, case_vars)

    def calcConsts(self, consts):
        for case_variant in self.case_variant_sequence:
            case_variant.calcConsts(consts)

    def getTypeSize(self, types):
        type_size = types[self.type_identifier]
        type_size += max(
            case_variant.getTypeSize(types)
                for case_variant in self.case_variant_sequence)

        return type_size


@dataclass
class FieldList:
    identifier_with_types_list : tuple[IdentifierWithType]
    case_block : typing.Optional[CaseBlock] = None

    def check(self, types, consts, case_vars):
        for identifier_with_types in self.identifier_with_types_list:
            identifier_with_types.check(types, consts, case_vars)

        if self.case_block:
            self.case_block.check(types, consts, case_vars)

    def calcConsts(self, consts):
        for identifier_with_types in self.identifier_with_types_list:
            identifier_with_types.calcConsts(consts)
        if self.case_block:
            self.case_block.calcConsts(consts)

    def getTypeSize(self, types):
        type_size = 0
        for identifier_with_types in self.identifier_with_types_list:
            type_size += identifier_with_types.getTypeSize(types)
        if self.case_block:
            type_size += self.case_block.getTypeSize(types)
        return type_size

# block
class Block(abc.ABC):
    @abc.abstractmethod
    def check(self, types, consts): pass
    @abc.abstractmethod
    def calcConsts(self, consts): pass
    @abc.abstractmethod
    def calcTypeSizes(self, types): pass

@dataclass
class BlockConst(Block):
    identifier : Identifier
    identifier_coord : pe.Position
    constant : Constant
    @pe.ExAction
    def create(attrs, coords, res_coord):
        identifier, constant = attrs
        cidentifier, ceq, cconstant, csemicol = coords
        return BlockConst(
            identifier, cidentifier.start, constant)

    def check(self, types, consts):
        if self.identifier in consts:
            raise RepeatedConstant(self.identifier_coord, self.identifier)

        self.constant.check(types, consts)
        consts.append(self.identifier)

    def calcConsts(self, consts):
        consts[self.identifier] = self.constant.getValue(consts)

    def calcTypeSizes(self, types): pass

@dataclass
class BlockType(Block):
    identifier : Identifier
    identifier_coord : pe.Position
    type : Type
    @pe.ExAction
    def create(attrs, coords, res_coord):
        identifier, type_ = attrs
        cidentifier, ceq, ctype, csemicol = coords
        return BlockType(
            identifier, cidentifier.start, type_)
    def check(self, types, consts):
        if self.identifier in consts:
            raise RepeatedType(self.identifier_coord, self.identifier)

        types[self.identifier] = self.type
        self.type.check(types, consts)

    def calcConsts(self, consts):
        self.type.calcConsts(consts)

    def calcTypeSizes(self, types):
        types[self.identifier] = self.type.getTypeSize(types)

# program
@dataclass
class Program:
    block : Block

    def check(self):
        types = {
            'INTEGER': None,
            'BOOLEAN': None,
            'REAL': None,
            'CHAR': None,
            'TEXT': None,
        }
        consts = []

        for blocks_seq in self.block:
            for block in blocks_seq:
                block.check(types, consts)

    def getConsts(self):
        consts = {}

        for blocks_seq in self.block:
            for block in blocks_seq:
                block.calcConsts(consts)

        return consts

    def getTypeSizes(self):
        types = {
            'INTEGER': 2,
            'BOOLEAN': None,
            'REAL': 4,
            'CHAR': 0,
            'TEXT': None,
        }

        for blocks_seq in self.block:
            for block in blocks_seq:
                block.calcTypeSizes(types)

        return types

UNAR_SIGN = pe.Terminal(
    'UNAR_SIGN',
    r'[+-]?',
    str
)
IDENTIFIER = pe.Terminal(
    'IDENTIFIER',
    r'[a-zA-Z][a-zA-Z0-9]*',
    str.upper
)
UNSINGNED_NUMBER = pe.Terminal(
    'UNSINGNED_NUMBER',
    r'[0-9]+(\.[0-9]+)?(E[+-]?[0-9]+)?',
    float
)
CHAR_SEQUENCE = pe.Terminal(
    'CHAR_SEQUENCE',
    r'(?<=\')[^\']+(?=\')',
    str
)

def make_keyword(image):
    return pe.Terminal(
        image, image, lambda name: None,
        re_flags=re.IGNORECASE, priority=10
    )

KW_PACKED   = make_keyword('PACKED')
KW_ARRAY    = make_keyword('ARRAY')
KW_OF       = make_keyword('OF')
KW_FILE     = make_keyword('FILE')
KW_SET      = make_keyword('SET')
KW_RECORD   = make_keyword('RECORD')
KW_END      = make_keyword('END')
KW_CASE     = make_keyword('CASE')
```

```
KW_CONST    = make_keyword('CONST')
KW_TYPE     = make_keyword('TYPE')

# constant
NConstant               = pe.NonTerminal('constant')
NUnarSign               = pe.NonTerminal('unar sign')
NConstantIdentifier     = pe.NonTerminal('constant identifier')
# simple type
NSimpleType             = pe.NonTerminal('simple type')
NIdentifierList         = pe.NonTerminal('identifier list')
NTypeIdentifier         = pe.NonTerminal('type identifier')
NCommonTypeIdentifier   = pe.NonTerminal('common type identifier')
# type
NType                   = pe.NonTerminal('type')
NTypeAfterPacked        = pe.NonTerminal('type after packed')
NSimpleTypeList         = pe.NonTerminal('simple type list')
# field list
NFieldList              = pe.NonTerminal('field list')
NIdentifierWithTypeList = pe.NonTerminal('identifier with type list')
NIdentifierWithTypeSeq  = pe.NonTerminal('identifier with type seq')
NIdentifierWithType     = pe.NonTerminal('identifier with type')
NCaseBlock              = pe.NonTerminal('case block')
NCaseVariantSequence    = pe.NonTerminal('case block sequence')
NCaseVariant            = pe.NonTerminal('case block')
NConstantList           = pe.NonTerminal('constant list')
# block
NBlock                  = pe.NonTerminal('block')
NBlockConstSequence     = pe.NonTerminal('block const sequence')
NBlockConst             = pe.NonTerminal('block const')
NBlockTypeSequence      = pe.NonTerminal('block type sequence')
NBlockType              = pe.NonTerminal('block type')
# program
NProgram                = pe.NonTerminal('program')

# constant
NConstant |= NUnarSign, NConstantIdentifier, SignedIdentifierConstant.create
NConstant |= NConstantIdentifier, UnsignedIdentifierConstant.create
NConstant |= NUnarSign, UNSINGNED_NUMBER, SignedNumberConstant
NConstant |= UNSINGNED_NUMBER, UnsignedNumberConstant
NConstant |= '\'', CHAR_SEQUENCE, '\'', CharacterConstant

NUnarSign |= '+', lambda: UnarSign.Plus
NUnarSign |= '-', lambda: UnarSign.Minus

NConstantIdentifier |= IDENTIFIER
```

```
# simple type
NSimpleType |= IDENTIFIER, DefaultSimpleType.create
NSimpleType |= '(', NIdentifierList, ')', ListSimpleType.create
NSimpleType |= NConstant, '..', NConstant, BoundedSimpleType

NIdentifierList |= IDENTIFIER, lambda id: (id,)
NIdentifierList |= (
    IDENTIFIER, ',', NIdentifierList,
    lambda id, idlist: (id, *idlist)
)

# type
NType |= NSimpleType, DefaultType
NType |= '^', NTypeIdentifier, RefType.create
NType |= KW_PACKED, NTypeAfterPacked, PackedType
NType |= NTypeAfterPacked

NTypeAfterPacked |= (
    KW_ARRAY, NSimpleTypeList, KW_OF, NType,
    ArrayType
)
NTypeAfterPacked |= KW_FILE, KW_OF, NType, FileType
NTypeAfterPacked |= KW_SET, KW_OF, NSimpleType, SetType
NTypeAfterPacked |= KW_RECORD, NFieldList, KW_END, RecordType

NSimpleTypeList |= NSimpleType, lambda st: (st,)
NSimpleTypeList |= (
    NSimpleType, ',', NSimpleTypeList,
    lambda st, stlist: (st, *stlist)
)

NTypeIdentifier |= IDENTIFIER

# field list
NFieldList |= NIdentifierWithTypeList, FieldList
NFieldList |= NIdentifierWithTypeSeq, NCaseBlock, FieldList
NFieldList |= NCaseBlock, lambda c: FieldList((), c)

NIdentifierWithTypeList |= NIdentifierWithType, lambda iwt: (iwt,)
NIdentifierWithTypeList |= (
    NIdentifierWithType, ';', NIdentifierWithTypeList,
    lambda iwt, iwtlist: (iwt, *iwtlist)
)

NIdentifierWithTypeSeq |= NIdentifierWithType, ';', lambda iwt: (iwt,)
NIdentifierWithTypeSeq |= (
    NIdentifierWithType, ';', NIdentifierWithTypeSeq,
    lambda iwt, iwtseq: (iwt, *iwtseq)
)

NIdentifierWithType |= NIdentifierList, ':', NType, IdentifierWithType.create

NCaseBlock |= (
    KW_CASE, IDENTIFIER, ':', NTypeIdentifier, KW_OF,
    NCaseVariantSequence,
    CaseBlock.create
)

NCaseVariantSequence |= NCaseVariant, lambda cblock: (cblock,)
NCaseVariantSequence |= (
    NCaseVariant, ';', NCaseVariantSequence,
    lambda cb, cbseq: (cb, *cbseq)
)

NCaseVariant |= (
    NConstantList, ':', '(', NFieldList, ')', CaseVariant.create
)

NConstantList |= NConstant, lambda c: (c,)
NConstantList |= (
    NConstant, ',', NConstantList,
    lambda c, clist: (c, *clist)
)

# block
NBlock |= (
    KW_CONST, NBlockConstSequence, NBlock,
    lambda bcseq, block: (bcseq, *block)
)
NBlock |= (
    KW_TYPE, NBlockTypeSequence, NBlock,
    lambda btseq, block: (btseq, *block)
)
NBlock |= lambda: ()

NBlockConstSequence |= NBlockConst, lambda bc: (bc,)
NBlockConstSequence |= (
    NBlockConst, NBlockConstSequence,
    lambda bc, bcseq: (bc, *bcseq)
)

NBlockConst |= NConstantIdentifier, '=', NConstant, ';', BlockConst.create
```

```
NBlockTypeSequence |= NBlockType, lambda bt: (bt,)
NBlockTypeSequence |= (
    NBlockType, NBlockTypeSequence,
    lambda bt, btseq: (bt, *btseq)
)

NBlockType |= NTypeIdentifier, '=', NType, ';', BlockType.create

# program
NProgram |= NBlock, Program


p = pe.Parser(NProgram)
assert p.is_lalr_one()

p.add_skipped_domain(r'\s')
p.add_skipped_domain(r'{[^}]*}')
p.add_skipped_domain(r'\(\*([^*]|\*[^\)])*\*\)')

for filename in sys.argv[1:]:
    try:
        with open(filename) as f:
            tree = p.parse(f.read())
            tree.check()
            print('Программа корректна')

            print(dumps(tree.getConsts(), indent=2))
            print(dumps(tree.getTypeSizes(), indent=2))
    except pe.Error as e:
        print(f'Ошибка {e.pos}: {e.message}')
```

Тестирование

Входные данные

```
Type
  Coords = Record x, y: INTEGER end;
  Boolean = (False, True);
Const
  MaxPoints = 100;
type
  CoordsVector = array 1..MaxPoints of Coords;

const
```

```
  Heigh = 480;
  Width = 640;
  Lines = 24;
  Columns = 80;
type
  BaseColor = (red, green, blue, highlited);
  Color = set of BaseColor;
  GraphicScreen = array 1..Heigh of array 1..Width of Color;
  TextScreen = array 1..Lines of array 1..Columns of
    record
      Symbol : CHAR;
      SymColor : Color;
      BackColor : Color
    end;
  Screen = record
    case isText : Boolean of
      True : (text : TextScreen);
      False : (graphic : GraphicScreen)
  end;

(* определения токенов }
{ определения токенов *)
(* определения токенов *)
{ определения токенов }
{ определения токенов *)
(* определения токенов }
TYPE
  Domain = (Ident, IntNumber, RealNumber);
  Token = record
    fragment : record
      start, following : record
        row, col : INTEGER
      end
    end;
    case tokType : Domain of
      Ident : (
        name : array 1..32 of CHAR
      );
      IntNumber : (
        intval : INTEGER
      );
      RealNumber : (
        realval : REAL
      )
  end;
  Year = 1900..2050;

  List = record
    value : Token;
    next : ^List
  end;
```

Вывод на `stdout`

```
Программа корректна
{
  "FALSE": 0,
  "TRUE": 1,
  "MAXPOINTS": 100.0,
  "HEIGH": 480.0,
  "WIDTH": 640.0,
  "LINES": 24.0,
  "COLUMNS": 80.0,
  "RED": 0,
  "GREEN": 1,
  "BLUE": 2,
  "HIGHLITED": 3,
  "IDENT": 0,
  "INTNUMBER": 1,
  "REALNUMBER": 2
}
{
  "INTEGER": 2,
  "BOOLEAN": 2,
  "REAL": 4,
  "CHAR": 0,
  "TEXT": null,
  "COORDS": 4,
  "COORDSVECTOR": 400,
  "BASECOLOR": 2,
  "COLOR": 1,
  "GRAPHICSCREEN": 307200,
  "TEXTSCREEN": 3840,
  "SCREEN": 307202,
  "DOMAIN": 2,
  "TOKEN": 14,
  "YEAR": 2,
  "LIST": 18
}
```

Вывод

В результатате выполенния данной работы были получены навыки выполнения семантического анализа.