Министерство науки и ВЫСШЕГО образования

Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Курский государственный университет»

Факультет физики, математики, информатики

Кафедра алгебры, геометрии и теории обучения математике

Курсовая РАБОТА

по дисциплине

Теория формальных языков и трансляции

на тему: РАЗРАБОТКА КОМПИЛЯТОРА МОДЕЛЬНОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ

ВАРИАНТ 1

Обучающегося 3 курса

очной формы обучения

направления подготовки

02.03.03 Математическое обеспечение и администрирование информационных систем

Направленность (профиль) Проектирование информационных систем и баз данных

Богданёнка Вячеслава Юрьевича

Руководитель:

доцент кафедры алгебры, геометрии и теории обучения математике

Селиванова Ирина Васильевна

Допустить к защите:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ / \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_\_\_\_г.

Курск, 2020

ВВЕДЕНИЕ

В данной курсовой работе разрабатывается компилятор модельного языка программирования

От примитивных вычислительных машин и до сегодняшнего дня человечество изобрело порядка 8 тысяч языков программирования, для каждого из которых так или иначе есть свой компилятор.

Компилятор – это программа, которая переводит набор команд, написанных на высоком уровне с учётом тех или иных особенностей языка программирования высокого уровня, в набор команд, понятных самой машине.

Разумеется перевод в байт код(чаще ассемблер) для понятия исходного текста программы машиной, происходит не в один этап, а в несколько. На одном из этапов из исходного текста программы удаляются все лишние символы отступов и пробелов(не влияющие на синтаксис команд), а также комментарии и другие не влияющие на выполнение самой программы символы. Затем уже происходит лексический, синтаксический и семантический анализ в соответствующем порядке.

В современном мире языки программирования вышли на принципиально новый, высокий уровень, что позволило создать наиболее благоприятные условия для разработки для программистов и снизить порог вхождения в профессию программиста. Однако, велика вероятность что в дальнейшем будут развиваться новые виды программирования и сферы применения вычислительных систем, для которых понадобятся новые языки программирования и компиляторы для них.

Таким образом, актуальность данной курсовой работы заключается в том, что существует необходимость получения навыков разработки компиляторов языков программирования, для решения задач, продиктованных временем.

Цель курсовой работы состоит в практическом применении теоретических основ проектирования трансляторов с языков программирования и основ проектирования трансляторов с языков программирования и освоения средств автоматизации их построения.

Для выполнения данной поставленной задачи(написание модели компилятора) понадобится также ознакомиться с такой системой поиска подстрок в строке как регулярные выражения.

В качестве инструмента для реализации задачи был выбран язык программирования python и его библиотека tkinter. Последняя нужна для написания пользовательского интерфейса визуализации этапов компиляции.

Исходный код представляет собой текст, имеющий синтаксис в соответствии с вариантом задания (Вариант 1):

<программа>::= program var <описание> begin <оператор> {; <оператор>} end.

<описание>::= {<идентификатор> {, <идентификатор> } : <тип> ;}

<идентификатор>::=<буква> <непустая последовательность цифр> <буква>

<тип>::= integer | real | boolean

<составной оператор>::= begin <оператор> { ; <оператор> } end

<присваивание>::= <идентификатор> ass <выражение>

<условный>::= if <выражение> then <оператор> [ else <оператор>]

<цикл со счётчиком>::= for ( [<выражение>] ; [<выражение>] ; [<выражение>] ) <оператор>

<условный цикл>::= while <выражение> do <оператор>

<ввод>::= input (<идентификатор> {пробел <идентификатор>})

<вывода>::= writeln <выражение> {, <выражение> }

1. Разработка грамматики модельного языка программирования

Грамматика языков программирования – это способ описания предложений, построенный на определенном алфавите.

Для описания грамматики используют метаязык. К наиболее распространенным относятся форма Бэкуса-Наура, диаграммы Вирта.

## 1.1 Форма Бэкуса-Наура

Форма Бэкуса-Наура – это формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории.

<программа>::= program var <описание> begin <оператор> {; <оператор>} end. <описание>::= <identifikator> {, <identifikator> } : <tip> ;

<описание>::= {<идентификатор> {, <идентификатор> } : <тип> ;} <тип>::= integer | real | boolean <tsifra>=0|1|2|3|4|5|6|7|8|9

<составной>::= «{» <оператор> { ; <оператор> } «}»

<ввод>::= read (<идентификатор> {, <идентификатор> })

<условного\_цикла>::= do while <выражение> <оператор> loop

<фиксированного\_цикла>::= for <присваивания> to <выражение> do <оператор>

<условный>::= if <выражение> then <оператор> [ else <оператор>]

<присваивание>::= <идентификатор> ass <выражение>

1.2 Регулярные выражения

Регулярные выражения используются на всех этапах анализа в том или ином виде. При разбиении на лексемы они могут понадобится для обозначения типа лексемы (идентификатор, ключевое слово или константа).

Регуля́рные выраже́ния ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) regular expressions) — формальный язык поиска и осуществления манипуляций с [подстроками](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D1%81%D1%82%D1%80%D0%BE%D0%BA%D0%B0) в тексте, основанный на использовании метасимволов ([символов-джокеров](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB%D1%8B-%D0%B4%D0%B6%D0%BE%D0%BA%D0%B5%D1%80%D1%8B), [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) wildcard characters). Для поиска используется строка-образец ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) pattern, по-русски её часто называют «шаблоном», «маской»), состоящая из символов и метасимволов и задающая правило поиска. Для манипуляций с текстом дополнительно задаётся строка замены, которая также может содержать в себе специальные символы.[[1]](#footnote-1)

Для обработки синтаксиса и лексического анализа будут использоваться некоторые примитивные конструкции регулярных выражений. Для понимания принципов поиска подстрок с помощью регулярных выражений необходимо ознакомиться с основными метасимволами для их применения.

. – Любой символ за исключением переноса строки  
\d - Цифры (0-9)  
\D – Любой символ кроме цифры (0-9)  
\w – Буква (a-z, A-Z, 0-9, \_)  
\W – Любой символ кроме буквы  
\s – Отступ (табуляция, пробел, перенос строки)  
\S – НЕ отступ  
  
\b - Слово  
\B – Любая последовательность за исключением слова  
^ - Начало строки  
$ - Окончание строки  
  
[] - Искать символы в скобках  
[^ ] – Искать все символы, за исключением тех, что в скобках  
| - Логическое или  
( ) - Группирование  
  
Кванторы:  
\* - 0 или более совпадение  
+ - 1 или более совпадение  
? - 0 или одно совпадение  
{n} – Ровно n совпадений  
{n,m} – от n до m совпадений

Пример:

Строка ^\d{4}\s\-.+$ - находит четыре цифры, затем отступ, затем тире и любую последовательность символов до конца строки. Таким образом, например, можно находить даты событий в тех или иных документах.

Регулярные выражения для заданного синтаксиса (Вариант 1):

Идентификатор :== [a-z]{1}[0-9]+[a-z]{1}$

Структура программы :== ^program\s+var\s+\(\{\w{1}[0-9]+[a-z]{1} : (integer|real|bool\) )+\s+(begin){n}\s+(<оператор>\s)+(end){n}\.{1}

Оператор :== <условие>|<цикл со счётчиком>|<чтение>|<запись>|< присвоение>

Условие :== (if){1}\{\w{1}[0-9]+[a-z]{1}(then){1}(else)?<оператор>+

Цикл со счётчиком :== (for){1}<присваивание>(to){1}<выражение>(do){1}<оператор>

Чтение :== (read){1}\{\w{1}[0-9]+[a-z]{1}(\,\{\w{1}[0-9]+[a-z]{1})\*

Запись :== (writeln){1}\{\w{1}[0-9]+[a-z]{1}(\,\s\{\w{1}[0-9]+[a-z]{1})\*

Присваивание :== \{\w{1}[0-9]+[a-z]{1}ass(\w{1}[0-9]+[a-z]{1}|[0-9]|false|true|[0-9]+\.[0-9]+){1}((\+|\-|\/|\*){1}(\w{1}[0-9]+[a-z]{1}|[0-9]|false|true|[0-9]+\.[0-9]+))\* пояснение:

встречаются: один идентификатор ass идентификатор или константа ( операция (один раз) константа или идентификатор (один раз) (ни одного либо много раз)

2. Разработка анализатора лексики, синтаксиса и семантики.

Для решения задачи необходимо было разработать анализаторы лексики, синтаксиса и семантики.

* 1. Лексический анализатор

Лексический анализ был реализован в несколько этапов.

Этап 1:

На этом этапе создаётся новый объект класса PreProcessor, который получает на вход исходный код программы в чистом виде. Исходный код программы обрабатывается для более удобного разбиения на лексемы. Сперва удаляются все комментарии, затем с помощью регулярного выражения \s+ находятся все символы пробела и отступы. Они заменяются на один символ пробела. С помощью регулярного выражения [\\\\*.+\\*\/](file:///\\\*.+\*\/) находятся все комментарии, которые затем удаляются из исходного кода с помощью метода *.replace(<substring>, <replace string>)*. Затем строка разбивается на массив строк по пробелу с помощью метода .split(<string>, <del>).

Этап 2:

На этом этапе массив строк превращается в массив лексем, путём сравнения каждого элемента на соответствие с регулярным выражением. Исходный код всех классов представлен ниже.

import string

import re

class Error(Exception):

pass

class LexError(Error):

pass

class SyntaxError(Error):

pass

class SymanticError(Error):

pass

class Lexem:

def \_\_init\_\_(self, name):

self.name = name

def \_\_repr\_\_(self):

return f'<Лексема "{self.name}">'

def \_\_str\_\_(self):

return f'<Лексема "{self.name}">'

class ProgramIterator:

def \_\_init\_\_(self, program):

self.program = program

self.\_index = 0

def \_\_next\_\_(self):

if self.\_index < len(self.program.lines):

r = self.program.lines[self.\_index]

self.\_index += 1

return r

raise StopIteration()

class Program:

name = ''

def \_\_init\_\_(self, text : str):

self.text = text

self.lines = text.split('\n')

def \_\_iter\_\_(self):

return ProgramIterator(self)

def \_\_len\_\_(self):

return len(self.lines)

def \_\_getitem\_\_(self, i: int):

return self.lines[i]

class Structure:

TYPE = 'keyword'

def \_\_init\_\_(self, line : int, line\_text: str, name: str = ''):

self.name = name

self.line = line

self.line\_text = line\_text

def check\_syntax(self):

raise NotImplementedError('Not implemented')

def \_\_str\_\_(self):

return self.\_\_repr\_\_()

class IdentifierKW(Structure):

REGEX = r'^[a-z]{1}[0-9]+[a-z]{1}$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(IdentifierKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'идентификатор "{self.name}"'

def \_\_eq\_\_(self, other):

return self.name == other.name

class Operator:

words = []

def \_\_init\_\_(self, words):

self.words = words

class ProgramKW(Structure):

REGEX = r'program'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ProgramKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово "{self.name}"'

class VarKW(Structure):

REGEX = r'var'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(VarKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово "{self.name}"'

class BeginKW(Structure):

REGEX = r'begin'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(BeginKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово "begin"'

class TypeKW(Structure):

REGEX = r'(^integer$|^real|^boolean$)'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(TypeKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'тип {self.name}'

class EndKW(Structure):

REGEX = r'(^end$)'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(EndKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово "end"'

class IntegerKW(Structure):

REGEX = r'^\-?\d+$'

TYPE = 'variable'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(IntegerKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'целое число {self.name}'

class RealKW(Structure):

REGEX = r'^\d+\.\d+$'

TYPE = 'variable'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(RealKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'дробное число число {self.name}'

class SemiColumnKW(Structure):

REGEX = r'^;$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(SemiColumnKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'точка с запятой'

class AssertionKW(Structure):

REGEX = r'^ass$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(AssertionKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'назначение'

class DotKW(Structure):

REGEX = r'^\.$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(DotKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'точка'

class ReadKW(Structure):

REGEX = r'^read$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ReadKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово read'

class ForKW(Structure):

REGEX = r'^for$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ForKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово for'

class ComaKW(Structure):

REGEX = r'^\,$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ComaKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'запятая'

class CurlBracketCloseKW(Structure):

REGEX = r'^\}$'

TYPE = 'bracket'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(CurlBracketCloseKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'закр. фигурная скобка'

class CurlBracketOpenKW(Structure):

REGEX = r'^\{$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(CurlBracketOpenKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'откр. фигурная скобка'

class BracketOpenKW(Structure):

REGEX = r'^\($'

TYPE = 'bracket'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(BracketOpenKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'откр. скобка'

class BracketCloseKW(Structure):

REGEX = r'^\)$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(BracketCloseKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'закр. скобка'

class WriteKW(Structure):

REGEX = r'^writeln$'

TYPE = 'keyword'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(WriteKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'вывод выражения'

class OperationKW(Structure):

REGEX = r'^(\-|\+|\\*|\/)$'

TYPE = 'operation'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(OperationKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'арифм. операция {self.name}'

class ColumnKW(Structure):

REGEX = r'^\:$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ColumnKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'двоеточие'

class IfKW(Structure):

REGEX = r'^if$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(IfKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово if'

class ThenKW(Structure):

REGEX = r'^then$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ThenKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово then'

class SqBracketOpenKW(Structure):

REGEX = r'^\[$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(SqBracketOpenKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'^[$'

class SqBracketCloseKW(Structure):

REGEX = r'^\]$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(SqBracketCloseKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f']'

class ElseKW(Structure):

REGEX = r'^else$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ElseKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово else'

class DoKW(Structure):

REGEX = r'^do$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(DoKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово do'

class LoopKW(Structure):

REGEX = r'^loop$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(LoopKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово loop'

class WhileKW(Structure):

REGEX = r'^while$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(WhileKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово while'

class BooleanKW(Structure):

REGEX = r'^(true|false)$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(BooleanKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово then'

class ComparisonKW(Structure):

REGEX = r'^(\<|\>|\>\=|\<\=|\=\=|\!\=)$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ComparisonKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'операция сравнения {self.name}'

class ToKW(Structure):

REGEX = r'^to$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ToKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово to'

class StepKW(Structure):

REGEX = r'^step$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(StepKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово step'

class NextKW(Structure):

REGEX = r'^next$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(NextKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово next'

Если же лексема не попадает ни под одно поле REGEX наследнников класса structure, то в этом случае выдаётся лексическая ошибка и анализ заканчивается, без возможности продолжить следующий вид анализа (синтаксический и семантический).

На этом лексический анализ заканчивается. Все лексемы, в случае успешной обработки хранятся в поле lexems класса Analyser.

Результат выполнения лексического анализа представлен на рисунках 1-2

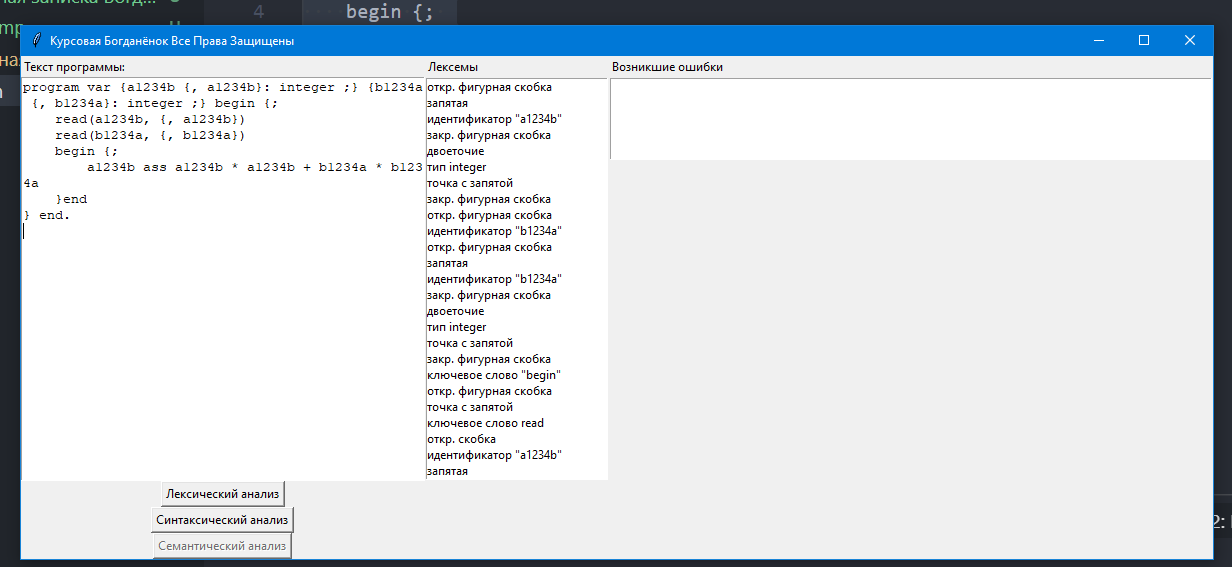


Рисунок 1 – успешный лексический анализ

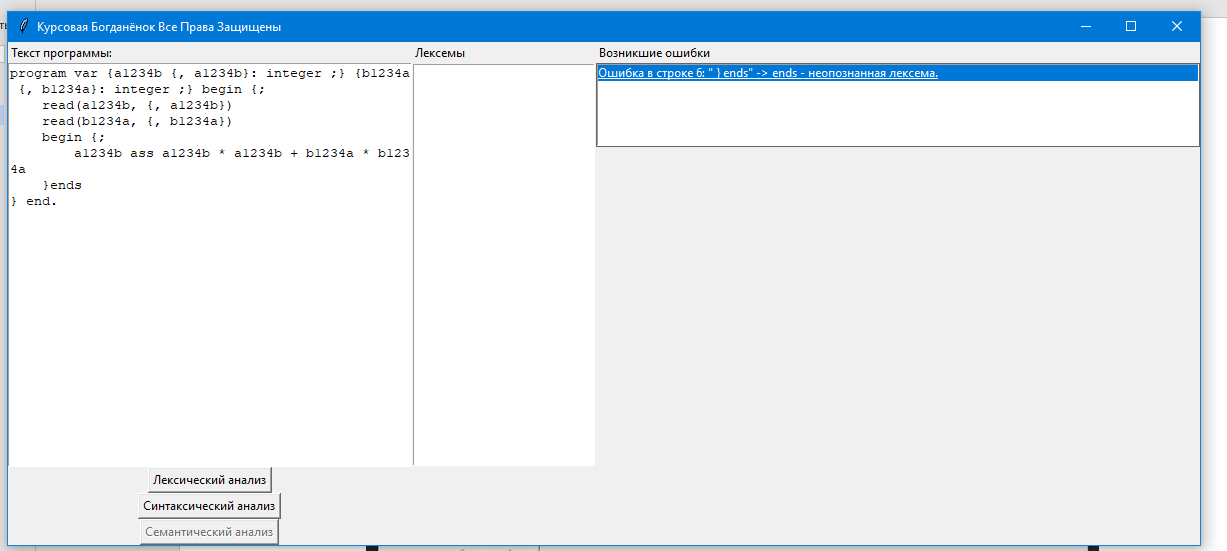


Рисунок 2 – ошибка в лексическом анализе

* 1. Синтаксический анализатор

Синтаксический анализ – это проверка на правильную последовательность ключевых слов в программе. Синтаксический анализатор – это средство, выполняющее синтаксический анализ исходного кода программы.

Синтаксический анализ выполняется методом *.syntax\_analyse()* класса Analyser, код которого представлен ниже:

def \_check\_syntax(self):

i = 0

checked = []

while i < len(self.lexems):

if i > 0 and (

self.lexems[i].TYPE == 'operation' and self.lexems[i-1].TYPE == 'keyword'

):

raise SyntaxError(f'Ошибка в строке {self.lexems[i].line}: {self.lexems[i].line\_text} -> {self.lexems[i].name} Ошибка синтаксиса')

last\_i = i

matched\_len = 0

for seq in FOLLOWING\_MATRIX:

matched = True

for j in range(len(seq)):

if seq[j].REGEX != self.lexems[i].REGEX:

matched = False

matched\_len = 0

i = last\_i

break

i += 1

matched\_len += 1

if matched\_len == len(seq):

break

if not matched:

raise SyntaxError(f'Ошибка в строке {self.lexems[i].line}: {self.lexems[i].line\_text} -> {self.lexems[i].name} Ошибка синтаксиса')

В данном методе реализована машина состояний. При состоянии 1 - на входе могут быть только ограниченное количество лексем, в поле TYPE которого указано значение ‘keyword’. При нахождении такого ключевого слова машина переключается в следующее состояние и проверяет следующую после найденной лексему, на соответствие с матрицей последовательности (FOLLOWING\_MATRIX). Как только мы находим последний элемент в матрице последовательности и он совпадает с таким же количеством последующих лексем машина переключается в изначальное состояние и снова принимает на вход только лексемы с типом keyword. Если найдена лексема с другим типом, например operation, то возникает ошибка синтаксиса и анализ прекращается.

Результат синтаксического анализа представлен на рисунках 3-4.

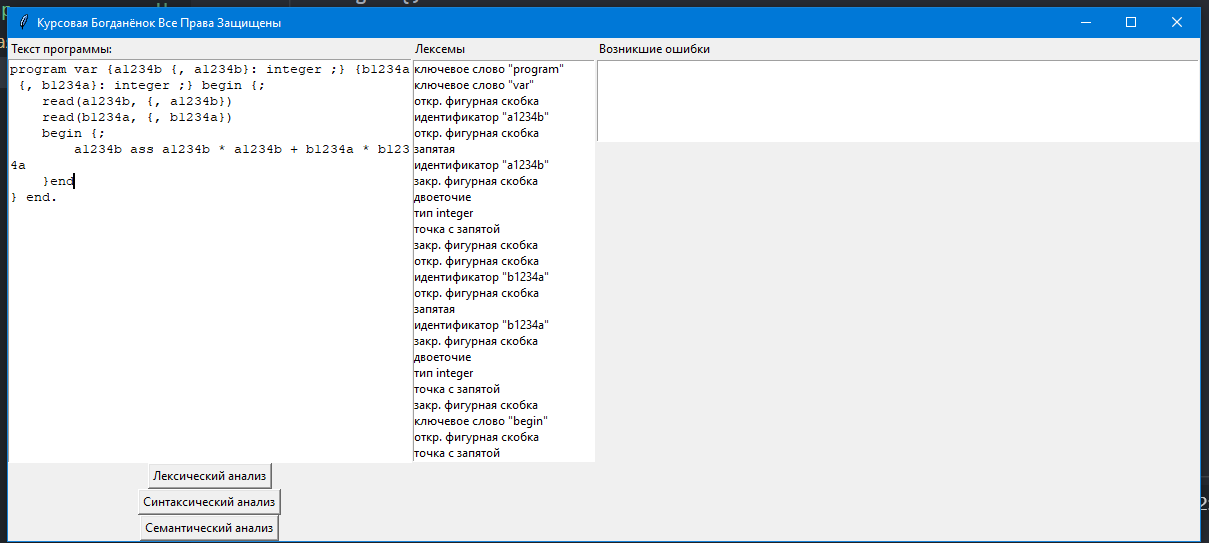


Рисунок 3 – успешный синтаксический анализ

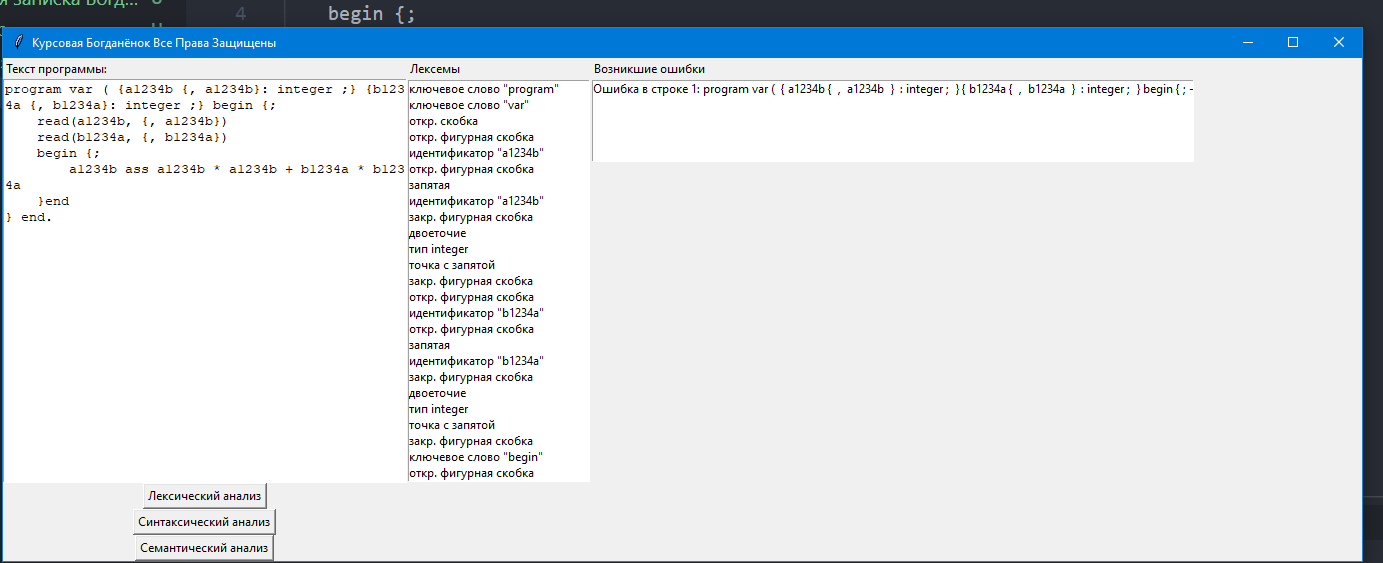


Рисунок 4 – ошибка в синтаксическом анализе

* 1. Семантический анализ

Семантический анализ – это процесс выявления идентификаторов на совпадение. Главная суть семантического анализа – это проверка идентификаторов на то, что они были объявлены, или же повторно объявлены.

Семантический анализ представлен в методе *symantix\_analyse()*, код которого представлен ниже.

def \_check\_defined\_idents(self, identifiers):

full\_text = self.preProcessor.program.text.replace('\n', ' ')

for ident in identifiers:

pattern = get\_define\_pattern(ident).replace('^', '').replace('$', '')

defined\_num = len(re.findall(pattern, full\_text))

if defined\_num == 0:

raise SymanticError(f'Ошибка в строке {ident.line}: "{ident.line\_text[:50]}" -> {ident.name} - Необъявленный идентификатор')

if defined\_num > 1:

raise SymanticError(f'Ошибка в строке {ident.line}: "{ident.line\_text[:50]}" -> {ident.name} - Многократное объявление')

Сперва в исходном тексте программы находятся все идентификаторы (учитывается тот факт, что синтаксический анализ прошёл успешно). Далее каждый из этих идентификаторов проверяется на объявление с помощью регулярного выражения \{<найденный идентификатор>: (integer|real|bool\). Если же объявление не было найдено для текущего идентификатора, то выдаётся ошибка идентификатора «необъявленный идентификатор».

Затем проверяются все объявленные идентификаторы с помощью регулярного выражения \w{1}.+\w{1}\:\s+(integer|bool|real). Если среди них встречаются повторные объявление (одна и та же строка найденная регулярным выражением найдена более одного раза), то выдаётся ошибка – повторное объявление.

Тестирование семантического анализа представлено на рисунках 5-6.

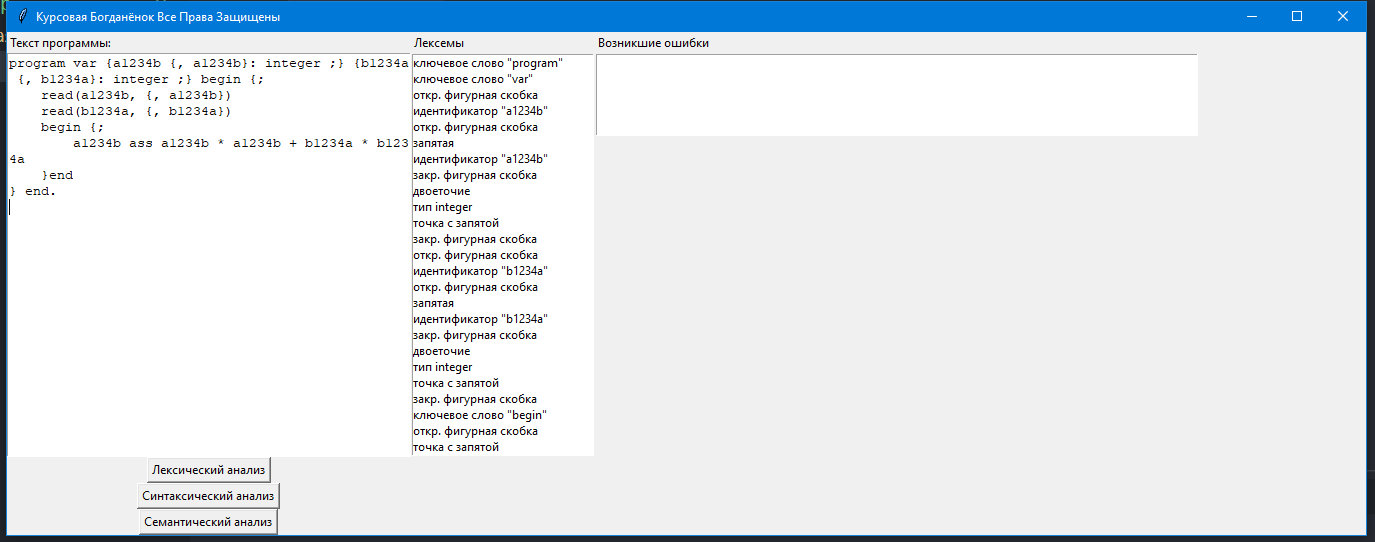


Рисунок 5 – семантический анализ прошёл успешно

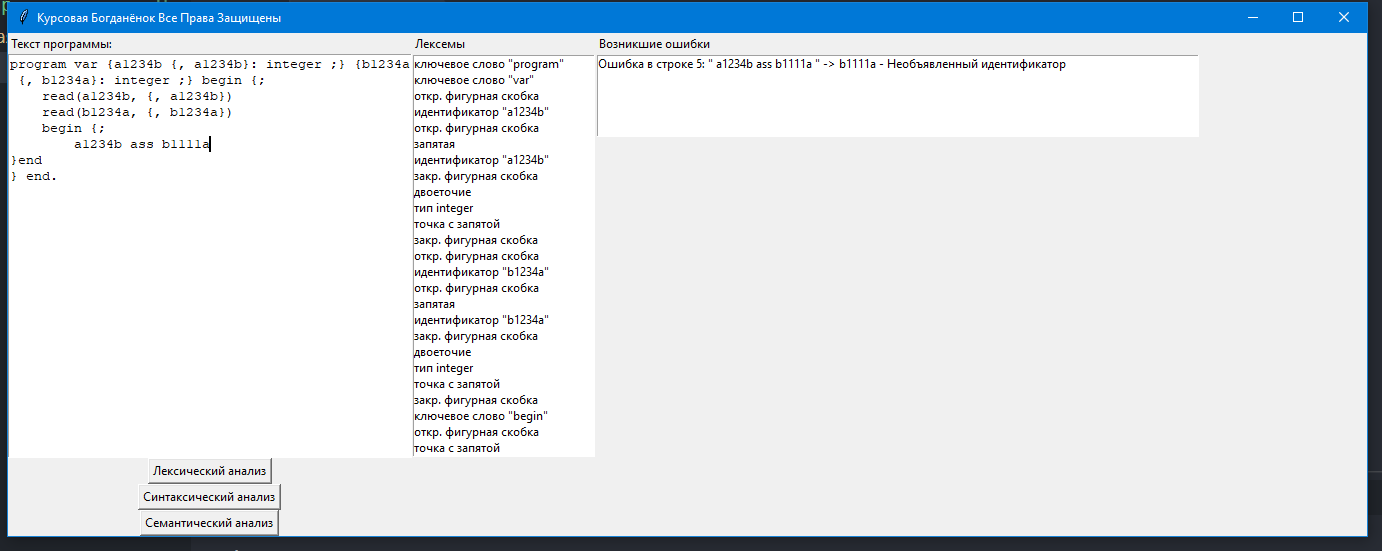


Рисунок 6 – семантический анализ завершён с ошибкой

1. Графический интерфейс

Так как сама программа выполнена на языке python, то для удобного построения визуального интерфейса пользователя была выбрана библиотека tkinter. Данная библиотека позволяет быстро и удобно составлять графический интерфейс.

Графический интерфейс приложения представлен на рисунке 7.

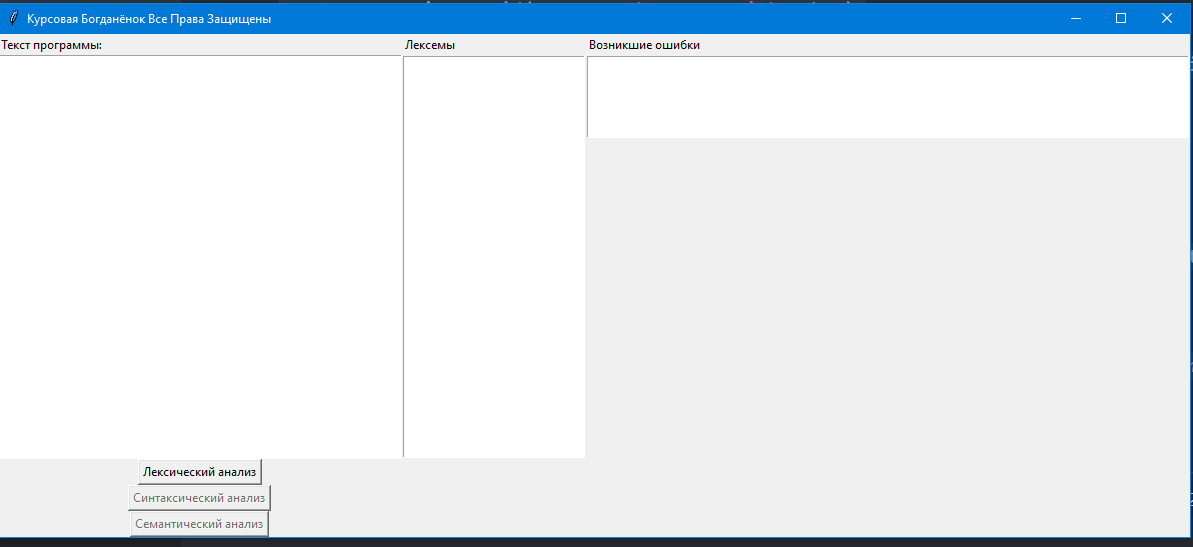


Рисунок 7 – графический интерфейс приложения

Слева в окне представлено поле для ввода исходного кода программы. Средняя колонка – список обнаруженных лексем (можно прокручивать с помощью колеса мыши). Самая правая колонка – поле для вывода ошибок, обнаруженных на той или иной стадии анализа. Снизу представлены три кнопки, нажав на которые пройдёт соответствующий анализ (чтобы начать синтаксический анализ нужно сначала провести лексический, чтобы начать семантический анализ, должен успешно быть пройден синтаксический). В случае прохождения более поздних стадий анализа, предыдущие этапы проходятся по новой автоматически (при вызове семантического синтаксический и лексический проходят автоматически и т.д.).

Заключение

В процессе выполнения курсовой работы была изучена тематическая литература, а также освоены базовые принципы разработки языка программирования.

Были реализованы следующие функции компилятора:

1. Лексический анализ. Были изучены и реализованы алгоритмы разбиения исходного кода программы на элементарные части (лексемы).
2. Синтаксический анализ. Были изучены и реализованы алгоритмы проверки синтаксиса исходного кода (правильной последовательности лексем).
3. Семантический анализ. Были изучены и реализованы алгоритмы проверки на использование необъявленных идентификаторов, а также их повторное объявление.

Таким образом были решены все поставленные задачи. Программа протестирована и готова к использованию.

В дальнейшем программа может быть усовершенствована, путём оптимизации алгоритмов поиска подстроки в строке, уменьшению перечня структур сгруппировав их по более базовым классам, а также есть возможность более подробного вывода конкретного места ошибки в исходном тексте в случае её возникновения.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Гросс М., Лантен А. Теория формальных грамматик/ Пер. с фр. — М.: Мир, 1971. — 296 с

2. Ишаков Е. Н. Разработка компиляторов – Оренбург: ОГУ. – 2005.

3. Фицджеральд М. Мастерство Регулярных выражений/ Оригинал – O’Reily, 2006. – 350 c

ПРИЛОЖЕНИЕ А

ТЕКСТ ПРОГРАММЫ

*app.py*

from analysers import PreProcessor, Analyser

from structures import Program, SyntaxError, LexError, SymanticError

import tkinter as tk

class MainApp(tk.Tk):

def initialize\_program(self):

program\_text = self.text\_area.get("1.0", tk.END)

self.program = Program(program\_text)

self.preProcessor = PreProcessor(self.program)

self.analyser = Analyser(self.preProcessor)

self.errors.delete(0, tk.END)

self.lexems.delete(0, tk.END)

try:

lexems = self.analyser.get\_lexems()

except Exception as e:

self.errors.insert(0, str(e))

return

for i in range(len(lexems)):

self.lexems.insert(i, str(lexems[i]))

self.syntaxButton.config(state=tk.ACTIVE)

def syntax\_analyse(self):

self.initialize\_program()

self.errors.delete(0, tk.END)

try:

self.analyser.syntax\_analyse()

except Exception as e:

self.errors.insert(0, str(e))

self.symanticButton.config(state=tk.ACTIVE)

def symantic\_analyse(self):

self.errors.delete(0, tk.END)

try:

self.analyser.symantic\_analyse()

except SymanticError as e:

self.errors.insert(0, str(e))

def \_\_init\_\_(self, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

container = tk.Frame(self)

self.initialized = False

self.title('Курсовая Богданёнок Все Права Защищены')

self.program\_text = tk.StringVar()

tk.Label(container, text='Текст программы:').grid(row=0, column=0, sticky=tk.W)

self.text\_area = tk.Text(container, width=50, height=25)

self.text\_area.grid(row=1, column=0)

container.grid(row=1, column=0, sticky=tk.W)

lex\_button = tk.Button(container, text="Лексический анализ", command=self.initialize\_program)

lex\_button.grid(row=2, column=0)

tk.Label(container, text='Лексемы').grid(row=0, column=1, sticky=tk.W)

self.lexems = tk.Listbox(container, height=25, width=30)

self.lexems.grid(row=1, column=1, sticky=tk.N)

tk.Label(container, text='Возникшие ошибки').grid(row=0, column=2, sticky=tk.W)

self.errors = tk.Listbox(container, height=5, width=100)

self.errors.grid(row=1, column=2, sticky=tk.N)

self.syntaxButton = tk.Button(container, text="Синтаксический анализ", command=self.syntax\_analyse)

self.syntaxButton.config(state=tk.DISABLED)

self.syntaxButton.grid(column=0, row=4)

self.symanticButton = tk.Button(container, text="Семантический анализ", command=self.symantic\_analyse)

self.symanticButton.config(state=tk.DISABLED)

self.symanticButton.grid(column=0, row=5)

main\_app = MainApp()

main\_app.mainloop()

*structures.py*

import string

import re

class Error(Exception):

pass

class LexError(Error):

pass

class SyntaxError(Error):

pass

class SymanticError(Error):

pass

class Lexem:

def \_\_init\_\_(self, name):

self.name = name

def \_\_repr\_\_(self):

return f'<Лексема "{self.name}">'

def \_\_str\_\_(self):

return f'<Лексема "{self.name}">'

class ProgramIterator:

def \_\_init\_\_(self, program):

self.program = program

self.\_index = 0

def \_\_next\_\_(self):

if self.\_index < len(self.program.lines):

r = self.program.lines[self.\_index]

self.\_index += 1

return r

raise StopIteration()

class Program:

name = ''

def \_\_init\_\_(self, text : str):

self.text = text

self.lines = text.split('\n')

def \_\_iter\_\_(self):

return ProgramIterator(self)

def \_\_len\_\_(self):

return len(self.lines)

def \_\_getitem\_\_(self, i: int):

return self.lines[i]

class Structure:

TYPE = 'keyword'

def \_\_init\_\_(self, line : int, line\_text: str, name: str = ''):

self.name = name

self.line = line

self.line\_text = line\_text

def check\_syntax(self):

raise NotImplementedError('Not implemented')

def \_\_str\_\_(self):

return self.\_\_repr\_\_()

class IdentifierKW(Structure):

REGEX = r'^[a-z]{1}[0-9]+[a-z]{1}$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(IdentifierKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'идентификатор "{self.name}"'

def \_\_eq\_\_(self, other):

return self.name == other.name

class Operator:

words = []

def \_\_init\_\_(self, words):

self.words = words

class ProgramKW(Structure):

REGEX = r'program'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ProgramKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово "{self.name}"'

class VarKW(Structure):

REGEX = r'var'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(VarKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово "{self.name}"'

class BeginKW(Structure):

REGEX = r'begin'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(BeginKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово "begin"'

class TypeKW(Structure):

REGEX = r'(^integer$|^real|^boolean$)'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(TypeKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'тип {self.name}'

class EndKW(Structure):

REGEX = r'(^end$)'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(EndKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово "end"'

class IntegerKW(Structure):

REGEX = r'^\-?\d+$'

TYPE = 'variable'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(IntegerKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'целое число {self.name}'

class RealKW(Structure):

REGEX = r'^\d+\.\d+$'

TYPE = 'variable'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(RealKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'дробное число число {self.name}'

class SemiColumnKW(Structure):

REGEX = r'^;$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(SemiColumnKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'точка с запятой'

class AssertionKW(Structure):

REGEX = r'^ass$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(AssertionKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'назначение'

class DotKW(Structure):

REGEX = r'^\.$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(DotKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'точка'

class ReadKW(Structure):

REGEX = r'^read$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ReadKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово read'

class ForKW(Structure):

REGEX = r'^for$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ForKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово for'

class ComaKW(Structure):

REGEX = r'^\,$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ComaKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'запятая'

class CurlBracketCloseKW(Structure):

REGEX = r'^\}$'

TYPE = 'bracket'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(CurlBracketCloseKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'закр. фигурная скобка'

class CurlBracketOpenKW(Structure):

REGEX = r'^\{$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(CurlBracketOpenKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'откр. фигурная скобка'

class BracketOpenKW(Structure):

REGEX = r'^\($'

TYPE = 'bracket'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(BracketOpenKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'откр. скобка'

class BracketCloseKW(Structure):

REGEX = r'^\)$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(BracketCloseKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'закр. скобка'

class WriteKW(Structure):

REGEX = r'^writeln$'

TYPE = 'keyword'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(WriteKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'вывод выражения'

class OperationKW(Structure):

REGEX = r'^(\-|\+|\\*|\/)$'

TYPE = 'operation'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(OperationKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'арифм. операция {self.name}'

class ColumnKW(Structure):

REGEX = r'^\:$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ColumnKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'двоеточие'

class IfKW(Structure):

REGEX = r'^if$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(IfKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово if'

class ThenKW(Structure):

REGEX = r'^then$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ThenKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово then'

class SqBracketOpenKW(Structure):

REGEX = r'^\[$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(SqBracketOpenKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'^[$'

class SqBracketCloseKW(Structure):

REGEX = r'^\]$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(SqBracketCloseKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f']'

class ElseKW(Structure):

REGEX = r'^else$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ElseKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово else'

class DoKW(Structure):

REGEX = r'^do$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(DoKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово do'

class LoopKW(Structure):

REGEX = r'^loop$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(LoopKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово loop'

class WhileKW(Structure):

REGEX = r'^while$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(WhileKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово while'

class BooleanKW(Structure):

REGEX = r'^(true|false)$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(BooleanKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово then'

class ComparisonKW(Structure):

REGEX = r'^(\<|\>|\>\=|\<\=|\=\=|\!\=)$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ComparisonKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'операция сравнения {self.name}'

class ToKW(Structure):

REGEX = r'^to$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(ToKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово to'

class StepKW(Structure):

REGEX = r'^step$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(StepKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово step'

class NextKW(Structure):

REGEX = r'^next$'

def \_\_init\_\_(self, line : int, line\_text: str, name : str):

super(NextKW, self).\_\_init\_\_(line, line\_text, name)

def \_\_repr\_\_(self):

return f'ключевое слово next'

def get\_define\_pattern(identifier):

pattern = CurlBracketOpenKW.REGEX + identifier.name + r'\s+' + CurlBracketOpenKW.REGEX + ComaKW.REGEX + r'\s+' + identifier.name + CurlBracketCloseKW.REGEX + ColumnKW.REGEX + r'\s+' + TypeKW.REGEX + r'\s+' + SemiColumnKW.REGEX + CurlBracketCloseKW.REGEX

return pattern

def unique\_identifiers(identifiers):

uniques = []

for identifier in identifiers:

not\_in = True

for unique in uniques:

if unique == identifier:

not\_in = False

if not\_in:

uniques.append(identifier)

return uniques

FOLLOWING\_MATRIX = [

[

CurlBracketOpenKW, IdentifierKW, CurlBracketOpenKW, ComaKW, IdentifierKW, CurlBracketCloseKW, ColumnKW, TypeKW, SemiColumnKW, CurlBracketCloseKW,

],

[

BeginKW, CurlBracketOpenKW, SemiColumnKW,

],

[

ProgramKW, VarKW

],

[

ReadKW, BracketOpenKW, IdentifierKW, ComaKW, CurlBracketOpenKW, ComaKW, IdentifierKW, CurlBracketCloseKW, BracketCloseKW,

],

[

CurlBracketCloseKW, EndKW, DotKW,

],

[

BeginKW, CurlBracketOpenKW, SemiColumnKW

],

[

IdentifierKW, AssertionKW, IntegerKW,

],

[

OperationKW, RealKW,

],

[

CurlBracketCloseKW, EndKW,

],

[

IfKW, BooleanKW,

],

[

ThenKW,

],

[

ElseKW,

],

[

DoKW, WhileKW,

],

[

IntegerKW, ComparisonKW, IntegerKW,

],

[

RealKW, ComparisonKW, RealKW,

],

[

LoopKW,

],

[

CurlBracketOpenKW, ComaKW,

],

[

CurlBracketCloseKW,

],

[

ForKW

],

[

ToKW,

],

[

SqBracketOpenKW, StepKW,

],

[

SqBracketCloseKW,

],

[

NextKW,

],

[

BracketOpenKW, IntegerKW, OperationKW,

],

[

IntegerKW, BracketCloseKW,

],

[

BracketOpenKW, RealKW, OperationKW,

],

[

RealKW, BracketCloseKW,

],

[

OperationKW, IntegerKW,

],

[

OperationKW, RealKW,

],

[

OperationKW, IdentifierKW,

],

[

IdentifierKW, AssertionKW, IdentifierKW,

],

[

WriteKW, IdentifierKW, CurlBracketOpenKW, ComaKW, IdentifierKW, CurlBracketCloseKW

]

]

*analysers.py*

from structures import Program, Lexem, IdentifierKW, LexError, FOLLOWING\_MATRIX, get\_define\_pattern, unique\_identifiers, SymanticError

import structures

import re

import inspect

import sys

def get\_structures():

structs = []

classes = inspect.getmembers(sys.modules['structures'], inspect.isclass)

for \_class in classes:

if \_class[0].endswith('KW'):

structs.append(\_class[1])

return structs

class PreProcessor:

def \_\_init\_\_(self, program: Program):

self.program = program

self.process()

def process(self):

lexems = []

bracets = r'(\{|\}|\(|\)|\)|\,|^end\.{1}$)'

for i in range(len(self.program.lines)):

self.program.lines[i] = self.program.lines[i].replace('end.', 'end .')

found\_comments = re.findall(

r'\/\\*.\*\\*\/', self.program.lines[i]

)

for comment in found\_comments:

self.program.lines[i] = self.program.lines[i].replace(

comment, ''

)

found\_bracets = re.findall(bracets, self.program.lines[i])

for bracet in found\_bracets:

self.program.lines[i] = self.program.lines[i].replace(bracet, f' {bracet} ')

found\_spaces = re.findall(r'\s+', self.program.lines[i])

for space in found\_spaces:

self.program.lines[i] = self.program.lines[i].replace(space, ' ')

for lexem in self.program.lines[i].split(' '):

if lexem != '':

lexems.append((Lexem(lexem), i))

self.lexems = lexems

class Analyser:

def \_\_init\_\_(

self,

preProcessor: PreProcessor,

):

self.preProcessor = preProcessor

def get\_lexems(self):

self.lexical\_analyse()

return self.lexems

def \_find\_last(self, lexem\_name):

last = None

prev = None

for lexem in self.lexems:

if lexem.name == lexem\_name:

last = lexem

break

return last

def \_check\_structure(self):

starting = r'^program var .+ begin'

struct = re.findall(starting, self.preProcessor.program.text)

init = len(struct)

if init != 1:

raise SyntaxError('Синтаксическая ошибка: нарушена структура программы')

def \_find\_errors(self):

self.\_check\_structure

bracets = r'(\{|\}|\(|\)|\)|\,|^end$|^begin$)'

errors = []

self.begins = 0

self.ends = 0

self.bracets\_op = 0

self.curl\_bracets\_op = 0

self.bracets\_cl = 0

self.curl\_bracets\_cl = 0

self.loop = 0

self.nxt = 0

self.sqr\_bracket\_close = 0

self.sqr\_bracket\_open = 0

self.whl = 0

self.prgm = 0

for lexem in self.lexems:

if lexem.name == 'begin':

self.begins += 1

elif lexem.name == 'end':

self.ends += 1

elif lexem.name == '{':

self.curl\_bracets\_op += 1

elif lexem.name == '}':

self.curl\_bracets\_cl += 1

elif lexem.name == '(':

self.bracets\_op += 1

elif lexem.name == ')':

self.bracets\_cl += 1

elif lexem.name == 'loop':

self.loop += 1

elif lexem.name == 'next':

self.nxt += 1

elif lexem.name == '[':

self.sqr\_bracket\_open += 1

elif lexem.name == ']':

self.sqr\_bracket\_close += 1

elif lexem.name == 'while':

self.whl += 1

elif lexem.name == 'program':

self.prgm += 1

error = None

if self.curl\_bracets\_cl > self.curl\_bracets\_op:

error = self.\_find\_last('}')

if self.curl\_bracets\_cl < self.curl\_bracets\_op:

error = self.\_find\_last('{')

if self.bracets\_cl > self.bracets\_op:

error = self.\_find\_last(')')

if self.bracets\_cl < self.bracets\_op:

error = self.\_find\_last('(')

if self.ends > self.begins:

error = self.\_find\_last('end')

if self.ends < self.begins:

error = self.\_find\_last('begin')

if self.whl < self.loop:

error = self.\_find\_last('loop')

if self.whl > self.loop:

error = self.\_find\_last('while')

if error:

raise SyntaxError(f'Ошибка в строке {error.line}: {error.line\_text} -> {error.name} - ошибка синтаксиса')

def \_check\_syntax(self):

i = 0

checked = []

while i < len(self.lexems):

if i > 0 and (

self.lexems[i].TYPE == 'operation' and self.lexems[i-1].TYPE == 'keyword'

):

raise SyntaxError(f'Ошибка в строке {self.lexems[i].line}: {self.lexems[i].line\_text} -> {self.lexems[i].name} Ошибка синтаксиса')

last\_i = i

matched\_len = 0

for seq in FOLLOWING\_MATRIX:

matched = True

for j in range(len(seq)):

if seq[j].REGEX != self.lexems[i].REGEX:

matched = False

matched\_len = 0

i = last\_i

break

i += 1

matched\_len += 1

if matched\_len == len(seq):

break

if not matched:

raise SyntaxError(f'Ошибка в строке {self.lexems[i].line}: {self.lexems[i].line\_text} -> {self.lexems[i].name} Ошибка синтаксиса')

def lexical\_analyse(self):

lexems = []

for lexem in self.preProcessor.lexems:

matched = False

for \_class in get\_structures():

new\_lexem = \_class(lexem[1]+1, self.preProcessor.program.lines[lexem[1]], lexem[0].name)

if re.match(\_class.REGEX, lexem[0].name):

lexems.append(new\_lexem)

matched = True

if not matched:

raise LexError(f'Ошибка в строке {new\_lexem.line}: "{new\_lexem.line\_text}" -> {new\_lexem.name} - неопознанная лексема.')

self.lexems = lexems

def syntax\_analyse(self):

self.\_check\_structure()

self.\_find\_errors()

self.\_check\_syntax()

def get\_identifiers(self):

identifiers = []

for lexem in self.lexems:

if lexem.REGEX == IdentifierKW.REGEX:

identifiers.append(lexem)

identifiers = unique\_identifiers(identifiers)

return identifiers

def \_check\_defined\_idents(self, identifiers):

full\_text = self.preProcessor.program.text.replace('\n', ' ')

for ident in identifiers:

pattern = get\_define\_pattern(ident).replace('^', '').replace('$', '')

defined\_num = len(re.findall(pattern, full\_text))

if defined\_num == 0:

raise SymanticError(f'Ошибка в строке {ident.line}: "{ident.line\_text[:50]}" -> {ident.name} - Необъявленный идентификатор')

if defined\_num > 1:

raise SymanticError(f'Ошибка в строке {ident.line}: "{ident.line\_text[:50]}" -> {ident.name} - Многократное объявление')

def symantic\_analyse(self):

identifiers = self.get\_identifiers()

self.\_check\_defined\_idents(identifiers)

1. Информация с википедии <https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%B3%D1%83%D0%BB%D1%8F%D1%80%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D1%8F> [↑](#footnote-ref-1)