ГУАП

КАФЕДРА № 44

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

канд. техн. наук, доцент		Н.В. Кучин
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ ПО ЛАБОРАТОРНЫМ РАБОТАМ №8-9

ОПРЕДЕЛЕНИЕ ОБЪЕМА ПАМЯТИ ДЛЯ СТРУКТУР ДАННЫХ

по курсу: СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 4941		Н. С. Горбунов
	подпись, дата	инициалы, фамилия

Санкт-Петербург 2022

1 Задание по лабораторной работе

Вариант 7

- № варианта грамматики: 1;
- Скалярные типа: byte (1 байт), real (6 байт);
- Кратность распределения памяти: 4;
- Кратность элементов структур: Нет.

Исходная грамматика 2:

S → type L var R**;**

L → T; | T; L

T → t=c | t=D

R → V; | V; R

V → K:t | K:c | K:D

K → a | K, a

D → **record** F **end**

F → E; | E; F

E → K:c | K:t

2 Цель работы

- Построение распознавателя исходного текста программы, содержащего описания типов данных, структур данных и переменных;
- Изучение основных принципов распределения памяти, ознакомление с алгоритмами расчета объема памяти, занимаемой простыми и составными структурами данных, получение практических навыков создания простейшего анализатора для расчета объема памяти, занимаемого заданной структурой данных.

3 Описание использованных лабораторной

Для выполнения лабораторной работы требуется написать программу, которая анализирует текст входной программы, содержащий описания типов данных и переменных, и рассчитывает объем памяти, требуемый для размещения всех переменных, описанных во входной программе.

Все переменные в исходной программе считаются статическими глобальными переменными. Результатом работы программы является значение требуемого объема памяти в байтах с учетом фрагментации памяти и без учета ее (объем памяти для скалярных типов данных и коэффициент фрагментации даются в задании).

Текст на входном языке задается в виде символьного (текстового) файла. Программа должна выдавать сообщения о наличие во входном тексте ошибок, если структура входной программы не соответствует заданию. Если в исходном тексте встречаются типы данных или структуры данных, не предусмотренные заданием, программа должна сигнализировать об ошибке/

4 Программная реализация

Было принято решение заменить C#, так как потребовалась полная перестройка логики работы программы, а это заняло бы продолжительное время. Поэтому перешел на использование Python из-за PLY (**P**ython **L**ex-**Y**acc) для анализа грамматики заполнения памяти.

Заданы токены лексем и правила поиска их при помощи LEX модуля.

Через модуль YACC была установлена грамматика и проверка синтаксиса полученного кода.

После синтаксического и лексического анализа проводится проверка на соответствие объявленных переменных и заданных скалярных типов.

При успешном проходе всех проверок программа начинает подсчёт занимаемой с выводом результатов подсчёта на экран.

В конечном итоге программа выводит итоговое значение, занимаемой памяти:

5 Полученные результаты

```
type
        Temperature=real;
        Address=byte;
        Sensor=record SensType:byte;Sens:Address;Current:Temperature; end;
    var
        Last, First: real;
        TMP:Sensor;
        Needed:Temperature;
```

Рисунок 1 – Исходный пример для анализа

Вывод программы:

```
All types in section are correct
All vars in section are correct

Calculation...

['Last', 'First'] 6 byte per element
 - with multiplicity:  48 byte
 - without multiplicity:  12 byte

................................................
['SensType'] 1 byte per element
 - with multiplicity:  4 byte
 - without multiplicity:  1 byte

['Sens'] 1 byte per element
 - with multiplicity:  4 byte
 - without multiplicity:  1 byte

['Current'] 6 byte per element
 - with multiplicity:  24 byte
 - without multiplicity:  6 byte

................................................
['TMP'] n byte
 - with multiplicity:  24 byte
 - without multiplicity:  6 byte

['Needed'] 6 byte per element
 - with multiplicity:  24 byte
 - without multiplicity:  6 byte

................................................
Usage memory with multiplicity 96
Usage memory without multiplicity 24
```

Рисунок 2. Результат работы программы

6 Вывод

Построен распознаватель исходного текста программы, содержащего описания типов данных, структур данных и переменных.

Изучены основных принципов распределения памяти. Ознакомился с алгоритмами расчета объема памяти, занимаемой простыми и составными структурами данных.

Получены практические навыки создания простейшего анализатора для расчета объема памяти, занимаемого заданной структурой данных.

ПРИЛОЖЕНИЕ 1

Код лексического анализатора

```
# Модуль лексического разбора кода
import ply.lex as lex


class CustomLexer(object):
    # Токены лексем
    tokens = ('TYPE', 'VAR', 'RECORD', 'TERM', 'COLON', 'SEMICOLON', 'COMMA',
'SCALAR', 'TIPE', 'SIGN', 'END')

    # Добавление символов игнорирования
    t_ignore = ' \r\t\f'

    # Добавление регулярных выражений для поиска лексем
    t_TYPE = r'((?:type)\w*)'
    t_VAR = r'((?:var)\w*)'
    t_END = r'((?:end)\w*)'
    t_SCALAR = r'((?:byte|real)\w*)'
    t_RECORD= r'((?:record)\w*)'
    t_TERM = r'[a-zA-Z]\w*'
    t_COMMA = r','
    t_COLON = r':'
    t_SEMICOLON = r';'
    t_SIGN = r'='
    t_TIPE = r'((?!byte|real))(((?<=[:])([a-zA-Z]\w*)(?=[;]))|([a-zA-Z]\w*)(?=[=]))'

    # Функция игнорирования комментария
    def t_comment(self, t):
        r'[#].*\n'
        t.lexer.skip(1)

    # Функция вывода лексемы в новой строке
    def t_newline(self, t):
        r'\n+'
        t.lexer.lineno += len(t.value)

    # Функция выведения ошибки лексического анализа
    def t_error(self, t):
        print("Illegal character '%s'" % t.value[0])
        t.lexer.skip(1)

    # Build the lexer
    def build(self, **kwargs):
        self.lexer = lex.lex(module=self, **kwargs)

    # Test it output
    def test(self, data):
        self.lexer.input(data)
        while True:
            tok = self.lexer.token()
            if not tok:
                break
            print(tok)


if __name__ == "__main__":
    data = '''
```

```
type
    Temperature=real;
    Address=byte;
    Sensor=record SensType:byte;Sens:Address;Current:Temperature; end;
var
    Last, First: real;
    TMP:Sensor;
    Needed:Temperature;
'''

lexer = CustomLexer()
lexer.build()
lexer.test(data)
```

ПРИЛОЖЕНИЕ 2

Код для построения дерева вывода

```
# Модуль синтаксического разбора кода
# Добавлен метод получения дерева синтаксического вывода
from lab89 import CustomLexer
import ply.yacc as yacc

# Список лексем
tokens = CustomLexer.tokens


# По функциям раскиданы правила грамматики
# S -> TYPE L VAR R;
# L -> T SEMICOLON | T SEMICOLON L
# T -> TIPE SIGN SCALAR | TIPE SIGN D
# R -> V SEMICOLON | V SEMICOLON R
# V -> K COLON TIPE | K COLON SCALAR | K COLON D
# K -> TERM | K COMMA TERM
# D -> RECORD F END
# F -> E | F SEMICOLON E
# E -> K COLON SCALAR | K COLON TIPE

# S -> TYPE L VAR R;
def p_Sstr(p):
    '''Sstr : TYPE Lstr VAR Rstr'''

    p[0] = [p[1]]

    type = {}
    var = {}
#разбираем type секцию
    for i in range(0, len(p[2]) - 1, 3):
        type[p[2][i]] = p[2][i + 1]

    p[0] += [type, p[3]]

    i = 0
#проверка на наличие record
    while i < (len(p[4]) - 1):
        key = p[4][i + 2]
        str = key[0]
        if str != 'record':
            var[key] = p[4][i]
            i += 4
        else:
            strok = ''
            for kol in p[4][i]:
                strok = strok + '' + kol + ''
            record_var = '(record) ' + strok
            var[record_var] = key[1]
            i += 5

    p[0] += [var]


# L -> T SEMICOLON | T SEMICOLON L
def p_Lstr(p):
    '''Lstr :  Tstr SEMICOLON
```

```
            |  Tstr SEMICOLON Lstr'''
    if len(p) == 3:
        p[0] = p[1]
    else:
        p[0] = [p[1][0], p[1][1], p[2]] + p[3]


# T -> TIPE SIGN SCALAR | TIPE SIGN D
def p_Tstr(p):
    '''Tstr : TIPE SIGN SCALAR
            | TIPE SIGN Dstr'''
    p[0] = [p[1], p[3]]


#R -> V SEMICOLON | V SEMICOLON R
def p_Rstr(p):
    '''Rstr : Vstr SEMICOLON
            | Vstr SEMICOLON Rstr'''
    if len(p) == 3:
        p[0] = p[1] + [p[2]]
    else:
        p[0] = p[1] + [p[2]] + p[3]


#V -> K COLON TIPE | K COLON SCALAR | K COLON D
def p_Vstr(p):
    '''Vstr   : Kstr COLON TIPE
              | Kstr COLON SCALAR
              | Kstr COLON Dstr'''
    if p[3][0] != 'record':
        p[0] = [p[1], p[2]] + [p[3]]
    else:
        p[0] = [p[1]] + [p[2]] + [p[3]]


# K -> TERM | K COMMA TERM
def p_Kstr(p):
    '''Kstr  : TERM
             | Kstr COMMA TERM'''
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1] + [p[3]]


# D -> RECORD F END
def p_Dstr(p):
    '''Dstr  : RECORD Fstr END'''
    p[0] = [p[1]]
    var = {}
#для каждого объявления внутри record
    for i in range(0, len(p[2]) - 1, 2):
        var[p[2][i + 1]] = p[2][i]

    p[0] += [var]

# F -> E | F SEMICOLON E
def p_Fstr(p):
    '''Fstr  : Estr SEMICOLON
             | Fstr Estr SEMICOLON'''
    if len(p) == 3:
```

```
        p[0] = p[1]
    else:
        p[0] = p[1] + p[2]


# E -> K COLON SCALAR | K COLON TIPE
def p_Estr(p):
    '''Estr  : Kstr COLON SCALAR
             | Kstr COLON TIPE'''
    p[0] = [p[1]] + [p[3]]


# Функция вывода ошибок
def p_error(p):
    print('Unexpected token:', p)


# Запуск синтаксического анализа и формирования файлов с правилами
parser = yacc.yacc()

# Запуск тестирования модуля
if __name__ == "__main__":
    data_var = '''
    type
        Temperature=real;
        Address=byte;
        Sensor=record SensType:byte;Sens:Address;Current:Temperature; end;
    var
        Last, First: real;
        TMP:Sensor;
        Needed:Temperature;
    '''

    new_lexer = CustomLexer()
    new_lexer.build()

    tree_list = parser.parse(data_var)
    print(tree_list)
```

ПРИЛОЖЕНИЕ 3

Программа вычисления занятой памяти

```
from lab89 import CustomLexer
from tree import parser

class MemoryCalculate:
#конструктор. В нем обнуляем все поля
    def __init__(self, type_dict, var_dict):
        self.type_dict = type_dict
        self.var_dict = var_dict

        self.c_types = ['byte', 'real']
        self.t_types = []

        self.status_of_type_section = None
        self.status_of_var_section = None

        self.usage_memory = [0, 0]

#вычисление размеров памяти
    def calculation(self):
        #для type секции проверка на правильность
        self.status_of_type_section = self.__type_check()
        #для var секции проверка на правильность
        self.status_of_var_section = self.__var_check()

        if self.status_of_type_section[0] == 'Error':
            print(self.status_of_type_section)
        elif self.status_of_var_section[0] == 'Error':
            print(self.status_of_var_section)
        else:
            print(self.status_of_type_section[0])
            print(self.status_of_var_section[0], '\n')

            self.__memory_count()

            print('.........................................')
            print('Usage memory with multiplicity', self.usage_memory[0])
            print('Usage memory without multiplicity', self.usage_memory[1])
            print()
#подсчитываем память
    def __memory_count(self):
        print('Calculation...')
        print()

        sample = [0, 0]

        for var in self.var_dict.keys():
            if var in self.c_types:
                sample = self.__memory_for_scalar(var, self.var_dict[var])
            elif var in self.t_types:
                if self.type_dict[var][0] != 'record':
                    sample = self.__memory_for_scalar(self.type_dict[var],
self.var_dict[var])
                else:
                    sample = self.__record_count(self.var_dict[var],
self.type_dict[var][1])
            elif 'record' in var:
```

```
                sample = self.__record_count(var, self.var_dict[var])

            self.usage_memory[0] += sample[0]
            self.usage_memory[1] += sample[1]

#определяем объем памяти для скалярных типов
    def __memory_for_scalar(self, scalar, lst):
        if scalar == 'byte':
            return self.__str_memory(lst, 1)

        elif scalar == 'real':
            return self.__str_memory(lst, 6)

    @staticmethod
    def __str_memory(lst, size):
        scalar_with_multiply, scalar_without_multiply = len(lst) * size * 4, len(lst)
* size

        print(f"{lst} {size} byte per element")
        print(' - with multiplicity: ', f"{scalar_with_multiply} byte")
        print(' - without multiplicity: ', f"{scalar_without_multiply} byte")
        print()

        return [scalar_with_multiply, scalar_without_multiply]

# подсчет и вывод памяти всех records
    def __record_count(self, name, record_dict):
        max_counters = [0, 0]

        print('.........................................')

        for elem in record_dict.keys():
            if elem in self.c_types:
                sample = self.__memory_for_scalar(elem, record_dict[elem])
            elif elem in self.t_types:
                sample = self.__memory_for_scalar(self.type_dict[elem],
record_dict[elem])
            else:
                sample = [0, 0]

            if sample[0] > max_counters[0]:
                max_counters[0] = sample[0]
                max_counters[1] = sample[1]

        print('.........................................')
        print(f"{name} n byte")
        print(' - with multiplicity: ', f"{max_counters[0]} byte")
        print(' - without multiplicity: ', f"{max_counters[1]} byte")
        print()

        return max_counters
#проверка на правильность объявления переменных в секции type
    def __type_check(self):
        for key in self.type_dict.keys():
            if self.type_dict[key] in self.c_types:
                self.t_types += [key]

            elif type_dict[key][0] == 'record':
                record_dict = type_dict[key][1]
                for jey in record_dict.keys():
                    if jey in self.c_types or jey in self.t_types:
```

```
                        continue
                    else:
                        return ['Error', 'type', self.type_dict[key][0]]
                self.t_types += [key]

            else:
                return ['Error', 'type', key]

        return ['All types in section are correct']
#проверка на правильность объявления переменных в секции var (описан ли тип в type)
    def __var_check(self):
        for var in self.var_dict.keys():
            if var in self.c_types or var in self.t_types:
                continue
            elif 'record' in var:
                for var_record in self.var_dict[var]:
                    if var_record in self.c_types or var_record in self.t_types:
                        continue
                    else:
                        return ['Error', 'var', var_record]
            else:
                return ['Error', 'var', var]

        return ['All vars in section are correct']


if __name__ == '__main__':
    # Открытие и чтение примера из файла
    input_file_path = 'Input.txt'
    file = open(input_file_path, 'r')
    example = file.read()
    file.close()

    # Синтаксический анализ примера и вывод двух словарей: var и type
    new_lexer = CustomLexer()
    new_lexer.build()
    program_list = parser.parse(example)

    # Определение словарей из дерева вывода
    type_dict = program_list[1]
    var_dict = program_list[3]

    # Вычисление занятой памяти
    calculator = MemoryCalculate(type_dict=type_dict, var_dict=var_dict)
    calculator.calculation()
```