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**

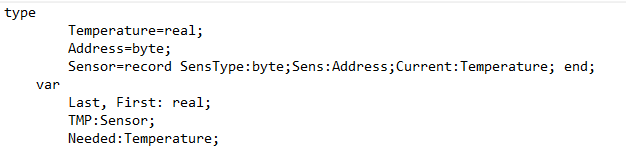
****

Рисунок 1 – Исходный пример для анализа

Вывод программы:

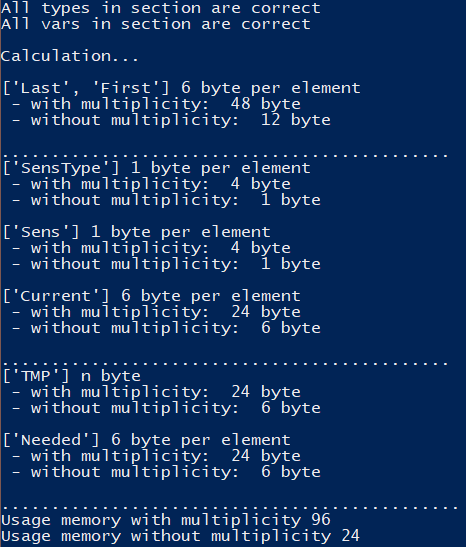


Рисунок 2. Результат работы программы

**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()