Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина «Методы трансляции»

|  |  |
| --- | --- |
|  |  |

**ОТЧЕТ**

к лабораторной работе № 5

на тему

**ИНТЕРПРЕТАЦИ ИСХОДНОГО КОДА**

Выполнил             К. А. Тимофеев

Проверил                          Н. Ю. Гриценко

Минск 2024

**СОДЕРЖАНИЕ**

[1 Постановка задачи 3](#_Toc157722973)

[2 Краткие теоретические сведения 4](#_Toc157722974)

[3 Результаты выполнения лабораторной работы 5](#_Toc157722975)

[Выводы 6](#_Toc157722976)

[Список использованных источников 8](#_Toc157722977)

[Приложение А (обязательное) Листинг исходного кода 9](#_Toc157722978)

# **1 ПОСТАНОВКА ЗАДАЧИ**

Целью выполнения данной лабораторной работы является на основе результатов анализа лабораторных работы 1-4 выполнить трансляцию программы с языка программирования С на язык программирования Python, после чего выполнить интерпретацию программы.

# **2 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ**

К этапам трансляции относятся следующие этапы:

– лексический анализ;

– синтаксический анализ;

– семантический анализ;

– оптимизация;

– генерация кода.

На этапе генерации компилятор создает код, который представляет собой набор инструкций, понятных для целевой аппаратной платформы, итоговый файл компилируется в исполняемый файл, который может быть запущен на целевой платформе без необходимости наличия кода.

Фаза эмуляции интерпретатора происходит во время выполнения программы. В отличие от компилятора, интерпретатор работает с кодом напрямую, без предварительной генерации машинного кода.

Лексический анализатор – первый этап трансляции. Лексический анализатор читает поток символов, составляющих исходную программу, и группирует эти символы в лексемы или значащие последовательности. Лексема – это элементарная единица, которая может являться ключевым словом, идентификатором, константным значением. Для каждой лексемы анализатор строит токен, который по сути является кортежем, содержащим имя и значение.[1]

Синтаксический анализатор выясняет, удовлетворяют ли предложения, из которых состоит исходная программа, правилам грамматики языка программирования. Синтаксический анализатор получает на вход результат лексического анализатора и разбирает его в соответствии с грамматикой. Результат синтаксического анализа обычно представляется в виде синтаксического дерева разбора.[2]

Семантический анализ обычно заключается в проверке правильности типа и вида всех идентификаторов и данных, используемых в программе.

Семантический анализатор использует синтаксическое дерево и информацию из таблицы символов для проверки исходной программы на семантическую согласованность с определением языка. Он также собирает информацию о типах и сохраняет ее в синтаксическом дереве или в таблице идентификаторов для последующего использования в процессе генерации промежуточного кода.

В данной лабораторной работе были использованы результаты анализа лексического, синтаксического и семантического анализаторов. После чего была выполнена интерпретация программ. Программами называются тестовые исходные коды, представленные в лабораторной работе 1.

# **3 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ             РАБОТЫ**

В ходе лабораторной работы был реализован интерпретатор программ с языка программирования С на язык программирования Python с последующей интерпретацией кода.

Листинг первого тестового кода представлен на рисунке 3.1.

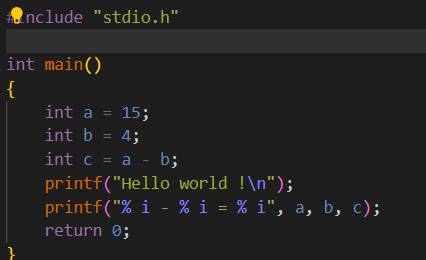


Рисунок 3.1 – Листинг первого тестового кода

Результат интерпретации первого исходного кода представлен   
на рисунке 3.2.



Рисунок 3.2 – Результат интерпретации первого исходного кода

Листинг второго исходного кода представлен на рисунке 3.3.

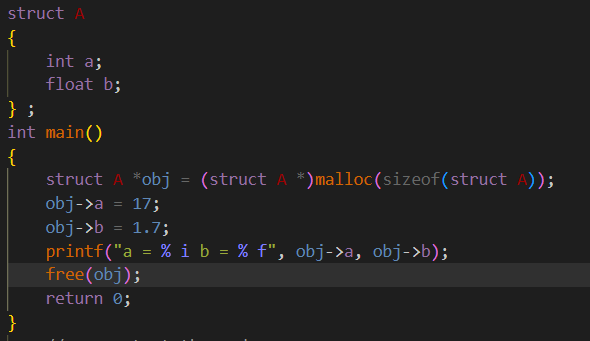


Рисунок 3.3 – Листинг второго тестового кода

Результат интерпретации второго тестового кода представлен на рисунке 3.4.



Рисунок 3.4 – Результат интерпретации второго тестового кода

Таким образом в ходе лабораторной работы был реализован интерпретатор для программ на языке С, который интерпретирует исходный язык при помощи языка программирования Python.

# **ВЫВОДЫ**

В ходе лабораторной работы был реализован был реализован интерпретатор для программ на языке С, который переводит их на язык программирования Python после чего проводит интерпретацию полученного при трансляции кода.

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Лексический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/. – Дата доступа: 27.02.2024.
2. Синтаксический анализатор [Электронный ресурс]. – Режим доступа: https://csc.sibsutis.ru/sites/csc.sibsutis.ru/files/courses/trans/. – Дата доступа: 27.02.2024.
3. Введение в С++ [Электронный ресурс]. – Режим доступа: <https://metanit.com/cpp/tutorial/2.5.php>. – Дата доступа: 28.02.2024.
4. Типы данных [Электронный ресурс]. – Режим доступа: <https://metanit.com/cpp/tutorial/2.3.php>. – Дата доступа: 28.02.2024.
5. Операторы в С++ [Электронный ресурс]. – Режим доступа: <https://learn.microsoft.com/ru-ru/cpp/c-operators>. – Дата доступа: 27.02.2024.
6. Функции С++ [Электронный ресурс]. – Режим доступа: https://metanit.com/cpp/tutorial/3.1.php. – Дата доступа: 27.02.2024.
7. Классы С++ [Электронный ресурс]. – Режим доступа: <https://ravesli.com/urok-113-klassy-obekty-i-metody-klassov/>. – Дата доступа: 27.02.2024.

## **ПРИЛОЖЕНИЕ А**

## **(обязательное)**

## **Листинг исходного кода**

Листинг 1 – Программный код parser.py

def execute(

self,

stack: bytearray,

heap: bytearray,

variables: list[tuple[VariableTableCell, int]],

literals: list[tuple[MyLiteral, int]],

):

lres = self.left.execute(stack, heap, variables, literals)

l = readValueAtAddr(stack, lres, self.left.ValidateTypes())

res = self.operator.handler(l)

return res

def execute(self, stack: bytearray, heap: bytearray, variables: list[tuple[VariableTableCell, int]], literals: list[tuple[MyLiteral, int]]):

lres = self.left.execute(stack, heap, variables, literals)

l = readValueAtAddr(stack, lres, self.left.ValidateTypes())

r = readValueAtAddr(

stack,

self.right.execute(stack, heap, variables, literals),

self.right.ValidateTypes(),)

res = self.operator.handler(l,r)

if self.operator.prior == 15:

writeValueAtAddr(stack, lres, res, self.ValidateTypes())

return res

def execute(self, stack: bytearray, heap: bytearray, variables: list[tuple[VariableTableCell, int]], literals: list[tuple[MyLiteral, int]]):

return readValueAtAddr(stack, find(literals, lambda l: l[0] == self.literal))

def execute(self, stack: bytearray, heap: bytearray, variables: list[tuple[VariableTableCell, int]], literals: list[tuple[MyLiteral, int]]):

res = self.condition.execute(stack, heap, variables, literals)

while res:

self.body.execute(stack, heap, variables, literals)

res = self.condition.execute(stack, heap, variables, literals)

def execute(self, stack: bytearray, heap: bytearray, variables: list[tuple[VariableTableCell, int]], literals: list[tuple[MyLiteral, int]]):

res = self.condition.execute(stack, heap, variables, literals)

while res:

self.body.execute(stack, heap, variables, literals)

res = self.condition.execute(stack, heap, variables, literals)

def execute(self, stack, heap, variables, literals):

res = self.condition.execute(stack, heap, variables)

if res:

return self.thenBody.execute(stack, heap, variables, literals)

else:

return self.elseBody.execute(stack, heap, variables, literals)

def execute(self, stack, heap, variables, literals):

res = find(variables, lambda v: v[0] == self.variable)[1]

res = readValueAtAddr(stack, res, self.variable.mytype.size)

writeValueAtAddr(stack, len(stack), res, self.variable.mytype)

def execute(self, stack, heap, variables, literals):

program: Program = self.getRoot()

sn = program.structs[self.structName]

shift = 0

for t, n in sn.fields:

if n != self.fieldName:

shift += t.size

else:

break

res = self.obj.execute()

return res + shift

def execute(self, stack: bytearray, heap: bytearray, variables: list[tuple[VariableTableCell, int]],

literals: list[tuple[VariableTableCell, int]]):

return self.value.execute()

def execute(self, stack:bytearray, heap:bytearray, variables: list[tuple[VariableTableCell, int]],

literals: list[tuple[VariableTableCell, int]]):

addr = len(stack)

stack.append([0 for i in range(self.varType.size)])

if self.initialValue is not None:

res = self.initialValue.execute()

resType = self.initialValue.ValidateTypes()

if resType != self.varType:

res = convert(res, resType, self.varType)

writeValueAtAddr(stack, addr, res, self.varType)

variables.append((self, addr))

def execute(

self,

stack: bytearray,

heap: bytearray,

variables: list[tuple[VariableTableCell, int]],

literals: list[tuple[VariableTableCell, int]]

):

start = len(stack)

varcount = len(variables)

for line in self.instructions:

if type(line) == ReturnNode:

res = line.execute(stack, heap, variables, literals)

stack = stack[:start]

return res

line.execute(stack, heap, variables, literals)

start = stack[:start]

variables = variables[:varcount]

return None

def execute(self, stack:bytearray, heap:bytearray, variables: list[tuple[VariableTableCell, int]],

literals: list[tuple[MyLiteral, int]]):

return self.body.execute(stack, heap, variables, literals)

def toBytes(val)->bytes:

if type(val) == int:

return bytes(struct.pack("i", val))

elif type(val) == float:

return bytes(struct.pack("f", val))

elif type(val) == str:

if len(val) == 1:

return bytes(val.encode())

return bytes((val + '\0').encode())

def fromBytes(bts:bytes, typ):

if typ == int:

return struct.unpack('i', bts)[0]

elif typ == float:

return struct.unpack("f", bts)[0]

elif typ == str:

if len(bts) == 1:

return bts.decode()

return bts.decode()[:-1]

def writeValueAtAddr(space:bytearray, addr:int, val, valsize):

res = bytearray(toBytes(val))

res = res[len(res) - valsize:]

i = addr

while i - addr < valsize:

space[i] = res[i - addr]

def readValueAtAddr(space, addr, valsize):

arr = bytearray()

for i in range(valsize):

arr.append(space[addr+i)

return fromBytes(bytes(arr), True)

def execute(self):

print(self.functions)

main = self.functions['main']

stack = bytearray()

heap = bytearray()

# [VariableTableCell, addr]

variables = list[tuple[VariableTableCell, int]]()

literals = list[tuple[MyLiteral, int]]()

for l in self.literals\_table:

addr = len(stack)

# stack.append([0 for i in range(l.mytype.size)])

res = l.value

# writeValueAtAddr(stack, addr, res, l.mytype.size)

literals.append((l, addr))

for gVar in self.variables.values():

gVar.execute(stack, heap, variables)

print()

print(f'\nProgram return code: {main.execute(stack, heap, variables, literals)}')