НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

**Лабораторна робота №3**

з дисципліни **«**Системне програмування-2**»**

Виконав: студент 3 курсу

гр. ІО-81

Рибніков Євгеній

Варіант 20

Перевірив:

Павлов В. Г.

Київ 2020 р.

**Тема:** локальні змінні.

**Мета:** побудова компілятору з використанням локальних змінних.

**Завдання:**



**Лістинг коду:**

**#dict.py**

LPAR = '('

RPAR = ')'

COLON = ':'

RETURN = 'return'

DEF = 'def'

NUM = 'num'

STRING = 'string'

ID = 'id'

INDENT = 'indent'

DQM = '"'

SQM = "'"

DEDENT = 'dedent'

NL = 'new line'

ENDMARK = 'endmark'

PLUS = '+'

EQUAL = '='

MINUS = '-'

LOGAND = 'and'

IF = 'if'

ELSE = 'else'

WHILE = 'while'

MULT = '\*'

LESS = '<'

MORE = '>'

UNAR\_NOT = 'not'

SEMICOLON = ';'

INV = '~'

**#Token**

class Token():

def \_\_init\_\_(self, tag, value=None):

self.tag = tag

self.value = value

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

return f"{self.tag}: {self.value}"

**#Lexer.py**

from Token import Token

from dict import \*

def is\_hex(a):

set = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'}

if a in set:

return True

else:

return False

class Lexer():

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

self.line = 1

self.index = 0

self.error = 0

self.peek = ' '

self.words = {}

self.indent = 0

self.old\_indent = 0

self.spaces = 0

def hold(self, t):

self.words[t.value] = t

def scan(self, content):

if self.index == len(content):

if self.old\_indent != 0:

self.old\_indent -= 1

return Token(DEDENT)

return Token(ENDMARK)

self.peek = content[self.index]

while self.peek == '\t':

self.indent += 1

self.index += 1

self.error += 1

self.peek = content[self.index]

while self.index < len(content) - 1:

if self.peek == ' ':

self.index += 1

self.error += 1

self.spaces += 1

self.peek = content[self.index]

else:

self.indent += self.spaces // 4

self.spaces = 0

break

if self.old\_indent < self.indent:

self.old\_indent += 1

return Token(INDENT)

elif self.old\_indent > self.indent:

self.old\_indent -= 1

return Token(DEDENT)

if self.peek == '"' or self.peek == "'":

a = ''

condition = True

while condition:

a += self.peek

self.index += 1

self.error += 1

self.peek = content[self.index]

condition = (self.peek != '"' and self.peek != "'")

if self.index == len(content) - 1:

break

a += self.peek

self.index += 1

self.error += 1

self.peek = content[self.index]

return Token(STRING, a)

if self.peek == "\n":

self.index += 1

self.error = 0

self.line += 1

self.indent = 0

return Token(NL)

if self.peek.isdigit():

k = 0

cond = True

if self.peek == '0':

self.index += 1

self.error += 1

self.peek = content[self.index]

if self.peek == 'x':

s = ''

self.index += 1

self.error += 1

self.peek = content[self.index]

while cond:

s += self.peek

if self.index == len(content) - 1:

break

self.index += 1

self.error += 1

self.peek = content[self.index]

cond = is\_hex(self.peek)

k = int(s, 16)

else:

k = 0

cond = False

while cond:

k = 10 \* k + int(self.peek)

if self.index == len(content) - 1:

break

self.index += 1

self.error += 1

self.peek = content[self.index]

cond = self.peek.isdigit()

return Token(NUM, k)

if self.peek.isalpha():

s = ''

condition = True

while condition:

s += self.peek

if self.index == len(content) - 1:

break

self.index += 1

self.error += 1

self.peek = content[self.index]

if self.peek.isalpha() or self.peek.isdigit():

condition = True

else:

condition = False

w = self.words.get(s)

if w is not None:

return w

if s == "not":

return Token(UNAR\_NOT)

elif s == "while":

return Token(WHILE)

w = Token(ID, s)

self.words[s] = w

return w

if self.peek == "~":

self.index += 1

self.error += 1

self.peek = content[self.index]

return Token(INV)

d = Token(self.peek)

if self.index < len(content) - 1:

self.index += 1

self.error += 1

self.peek = ' '

return d

**#Parser.py**

import sys

import os

from Lexer import Lexer

from Token import Token

from dict import \*

def\_list = []

var\_list = []

a = open("6-20-Python-IO-81-Rybnikov.txt", 'r')

input = a.read()

lexer = Lexer()

lexer.hold(Token(LOGAND, 'and'))

lexer.hold(Token(RETURN, 'return'))

lexer.hold(Token(DEF, "def"))

lexer.hold(Token(IF, 'if'))

lexer.hold(Token(ELSE, "else"))

lexer.hold(Token(WHILE, "while"))

unar\_op\_not = False

unar\_op\_inv = False

def match(a):

global lookahead

global input

if lookahead.tag == a:

lookahead = lexer.scan(input)

else:

raise SyntaxError

def term(t):

global lookahead

global unar\_op\_not, unar\_op\_inv

if lookahead.tag == NUM:

if unar\_op\_not:

t = "\tmov ebx, " + str(lookahead.value) + "\n" + t

t += "\tmov eax, ebx\n"

match(NUM)

unar\_op\_not = False

elif unar\_op\_inv:

t = "\tmov ebx, " + str(lookahead.value) + "\n" + t

t += "\tmov eax, ebx\n"

match(NUM)

unar\_op\_inv = False

else:

t += str(lookahead.value) + "\n"

match(NUM)

elif lookahead.tag == LPAR:

match(LPAR)

t = expression(t)

match(RPAR)

elif lookahead.tag == UNAR\_NOT:

unar\_op\_not = True

match(UNAR\_NOT)

if lookahead.value in var\_list:

match(ID)

t += "\tcmp ecx, 0\n\tpushf\n\txor ecx, ecx\n\tpopf\n\tsetz cl\n"

t = term(t)

else:

t += "\tcmp ecx, 0\n\tpushf\n\txor ecx, ecx\n\tpopf\n\tsetz cl\n"

t = term(t)

elif lookahead.tag == INV:

unar\_op\_inv = True

match(INV)

if lookahead.value in var\_list:

match(ID)

t += "\tnot eax\n"

t = term(t)

else:

t += "\tnot ebx\n"

t = term(t)

while lookahead.tag == STRING:

print("Type error: Expected int, got String in " + str(lexer.line) + " line, " + str(lexer.error) +

" index")

os.system("PAUSE")

sys.exit(1)

return t

def expression(t = ""):

global lookahead

counter = 0

t = term(t)

if lookahead.tag == PLUS:

while lookahead.tag == PLUS:

match(PLUS)

if lookahead.value in var\_list:

t += "\tadd eax, ebp\n"

t = expression(t)

else:

t += "\tadd eax, "

t = expression(t)

elif lookahead.tag == MINUS:

while lookahead.tag == MINUS:

match(MINUS)

t += "\tsub eax, "

t = expression(t)

elif lookahead.tag == MULT:

while lookahead.tag == MULT:

match(MULT)

t += "\tmov ebx, "

t = expression(t)

t += "\tmul ebx\n"

elif lookahead.tag == LOGAND:

while lookahead.tag == LOGAND:

match(LOGAND)

t += "\tcmp eax, 0\n" \

"\tjne \_case2\n" \

"\tjmp \_end\n"

t += "\_case2:\n" \

"\tmov eax, "

t = expression(t)

t += "\tcmp eax, 0\n" \

"\tmov eax, 0\n" \

"\tsetne al\n" \

"\_end:\n"

t = expression(t)

elif lookahead.tag == ID:

var\_list.append(lookahead.value)

match(ID)

if lookahead.tag == MINUS:

if lookahead.tag != EQUAL:

match(MINUS)

t += "\tsub eax, "

t = expression(t)

match(EQUAL)

t += ""

t = expression(t)

t += ""

t = expression(t)

if lookahead.tag == PLUS:

if lookahead.tag != EQUAL:

match(PLUS)

t += "\tadd eax, "

t = expression(t)

match(EQUAL)

t += ""

t = expression(t)

t += ""

t = expression(t)

if counter < 1:

if lookahead.tag == EQUAL:

counter += 1

match(EQUAL)

t += "\tpush ebp\n" \

"\tmov ebp, esp\n" \

"\tmov eax, "

t = expression(t)

t += "\tpush eax \n" \

"\tmov esp, ebp\n" \

"\tpop ebp\n"

t = expression(t)

elif lookahead.tag == IF:

match(IF)

t += "\tjmp \_post\_conditional\n"

t = expression(t)

if lookahead.tag == ELSE:

match(ELSE)

t += "\t\_e2:\n" \

"\tmov eax, "

t = expression(t)

t += "\t\_post\_conditional:\n"

elif lookahead.tag == WHILE:

while lookahead.tag == WHILE:

match(WHILE)

if lookahead.value in var\_list:

match(ID)

else:

print(f"Unknown variable in {str(lexer.line)} line, {str(lexer.error)} index")

os.system("PAUSE")

sys.exit(1)

if lookahead.tag == LESS:

match(LESS)

if lookahead.tag == MORE:

match(MORE)

t += "\tmov ecx, "

t = expression(t)

t += "\tcycle:\n"

try:

match(COLON)

except SyntaxError:

print(f"SyntaxError in {str(lexer.line - 1)} line, {str(lexer.error + 9)} index. Expect \":\".")

match(NL)

match(INDENT)

t = expression(t)

t += "\tcmp eax, ecx\n" \

"\tjne cycle\n"

elif lookahead.tag == RETURN:

match(RETURN)

if lookahead.tag != UNAR\_NOT and lookahead.tag != ID:

t += "\tmov eax, "

if lookahead.value in var\_list:

match(ID)

if lookahead.tag == ID and lookahead.value not in var\_list:

if lookahead.value not in def\_list:

print(f"Unknown variable in {str(lexer.line)} line, {str(lexer.error)} index")

os.system("PAUSE")

sys.exit(1)

if lookahead.value in def\_list:

match(ID)

match(LPAR)

match(RPAR)

if lookahead.tag == ID and lookahead.value not in def\_list:

print("Unknown function call in " + str(lexer.line) + " line, " + str(

lexer.error) + " index")

os.system("PAUSE")

sys.exit(1)

t = expression(t)

while lookahead.tag == NL:

match(NL)

while lookahead.tag == NL:

match(NL)

if lookahead.tag == DEDENT:

match(DEDENT)

return t

def start(c):

global lookahead

inside = []

outside = []

outside\_s = ''

inside\_s = ''

counter\_of\_names = 0

counter\_in = 0

counter\_out = 0

lookahead = lexer.scan(input)

while lookahead.tag != ENDMARK:

while lookahead.tag == DEF:

counter\_out += 1

match(DEF)

invoker = ''

invoker += f"\n{lookahead.value} proc\n"

def\_list.append(lookahead.value)

match(ID)

match(LPAR)

match(RPAR)

match(COLON)

match(NL)

match(INDENT)

invoker += expression()

if def\_list[counter\_of\_names] != "main":

invoker += f"\tinvoke MessageBox,0,str$(eax), ADDR Caption, MB\_OK\n"

if def\_list[counter\_of\_names] == "main":

invoker += f"\tcall {def\_list[counter\_of\_names-1]}\n"

invoker += f"\tret\n"

while lookahead.tag == DEDENT:

match(DEDENT)

invoker += f"{def\_list[counter\_of\_names]} endp\n"

counter\_of\_names += 1

while lookahead.tag == NL:

match(NL)

outside.append(invoker)

while lookahead.tag == ID:

invoker = ''

counter\_in += 1

if lookahead.value in def\_list:

invoker += f"\tinvoke {lookahead.value}\n"

match(ID)

match(LPAR)

match(RPAR)

while lookahead.tag == NL:

match(NL)

else:

invoker = expression(invoker)

inside.append(invoker)

match(ENDMARK)

for i in range(counter\_out):

outside\_s += outside[i]

for j in range(counter\_in):

inside\_s += inside[j]

c = c + outside\_s + "\nstart:\n" + inside\_s + "\tinvoke ExitProcess, 0\nend start"

return c

**#main.py**

import os

from Parser import start

a = f".386\n" \

f".model flat, stdcall\n\n" \

f"option casemap: none;\n" \

f"include E:\masm32\include\kernel32.inc\n" \

f"include E:\masm32\include\windows.inc\n" \

f"include E:\masm32\include\masm32rt.inc\n" \

f"include E:\masm32\include\\user32.inc\n" \

f"includelib E:\masm32\lib\kernel32.lib\n" \

f"includelib E:\masm32\lib\\user32.lib\n" \

f".data\n" \

f"Caption db 'Rybnikov', 0\n" \

f".code\n\n"

a = start(a)

b = open("6-20-Python-IO-81-Rybnikov.asm", "w")

b.write(a)

b.close()

c = open("6-20-Python-IO-81-Rybnikov.asm", "r")

line = c.readline()

while line:

print(line)

line = c.readline()

c.close()

os.system("PAUSE")

**Контрольні приклади**

**Приклад 1**

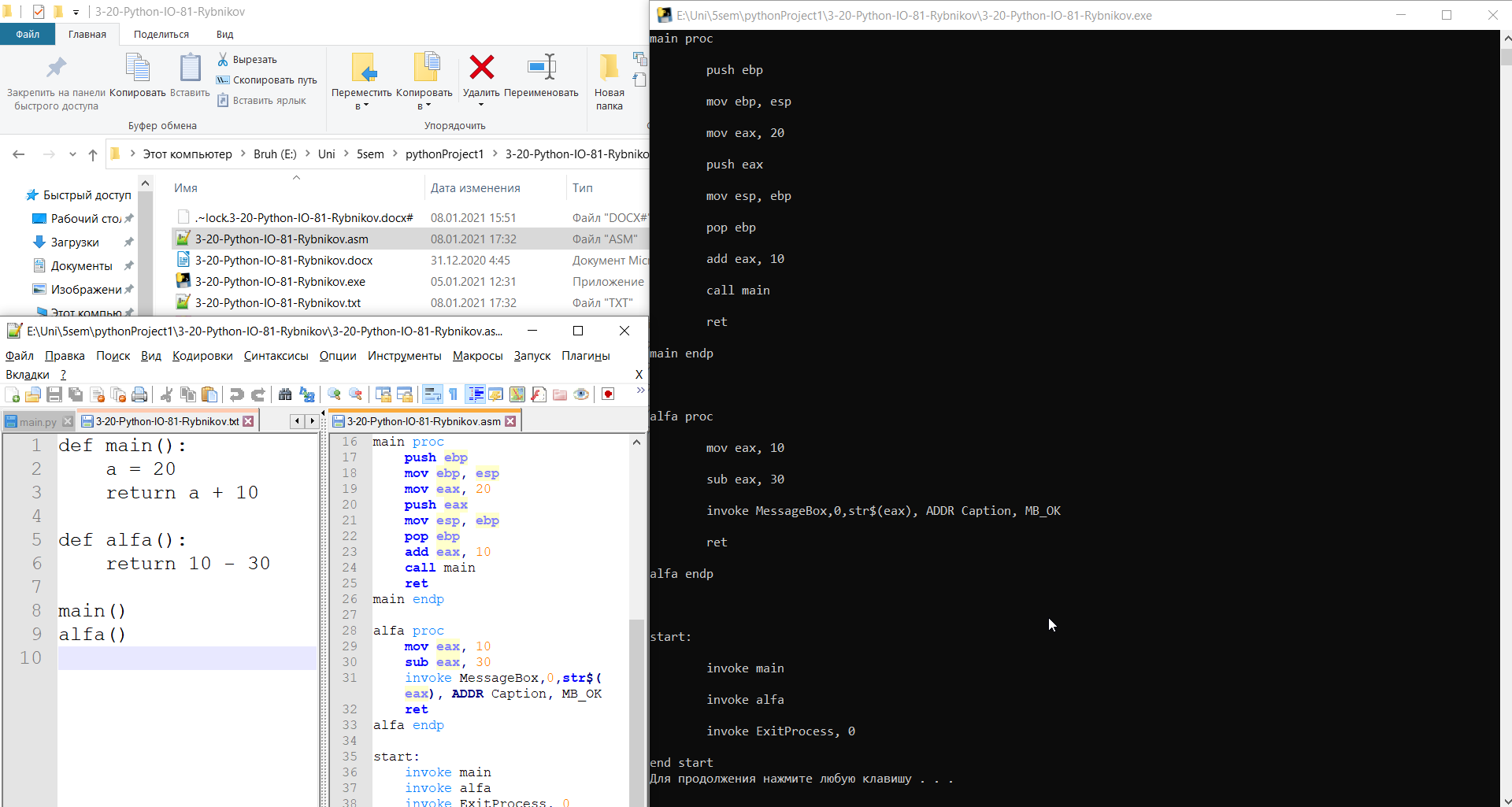
Вхідний код:

def main():  
 a = 20  
 return a + 10  
  
def alfa():  
 return 10 - 30  
  
main()  
alfa()

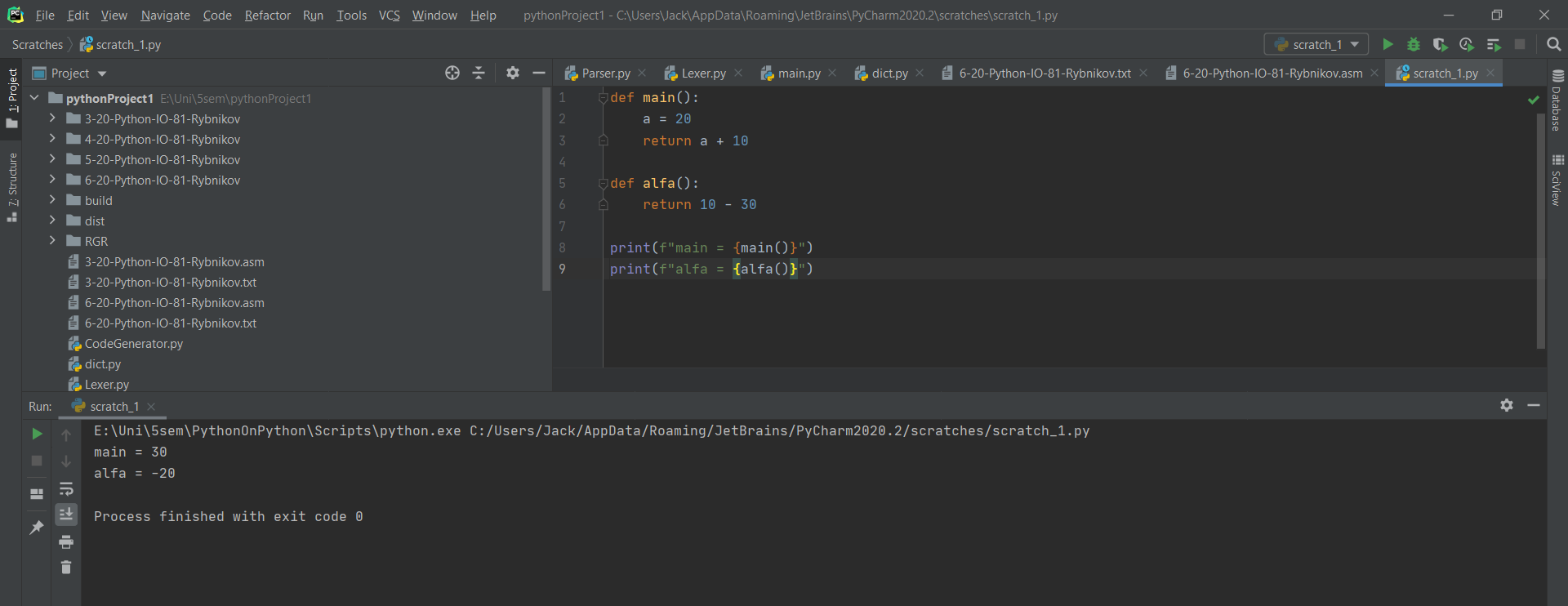
Вихідний код:

.386.model flat, stdcalloption casemap: none;include E:\masm32\include\kernel32.incinclude E:\masm32\include\windows.incinclude E:\masm32\include\masm32rt.incinclude E:\masm32\include\user32.incincludelib E:\masm32\lib\kernel32.libincludelib E:\masm32\lib\user32.lib.dataCaption db 'Rybnikov', 0.codemain proc push ebp mov ebp, esp mov eax, 20 push eax  mov esp, ebp pop ebp add eax, 10 retmain endpalfa proc mov eax, 10 sub eax, 30 invoke MessageBox,0,str$(eax), ADDR Caption, MB\_OK retalfa endpstart: invoke main invoke alfa invoke ExitProcess, 0end start

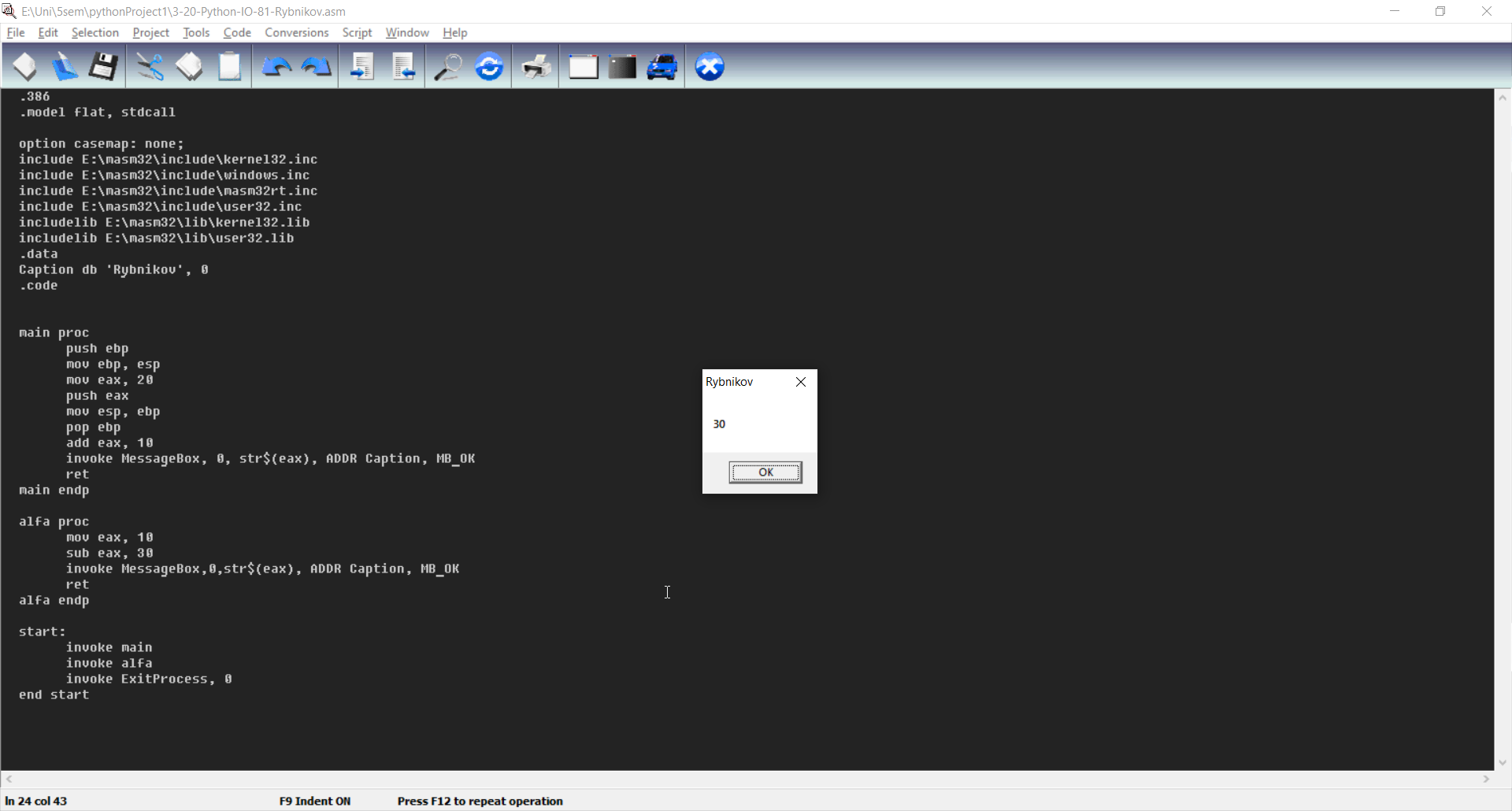
Отримання асемблерного коду:



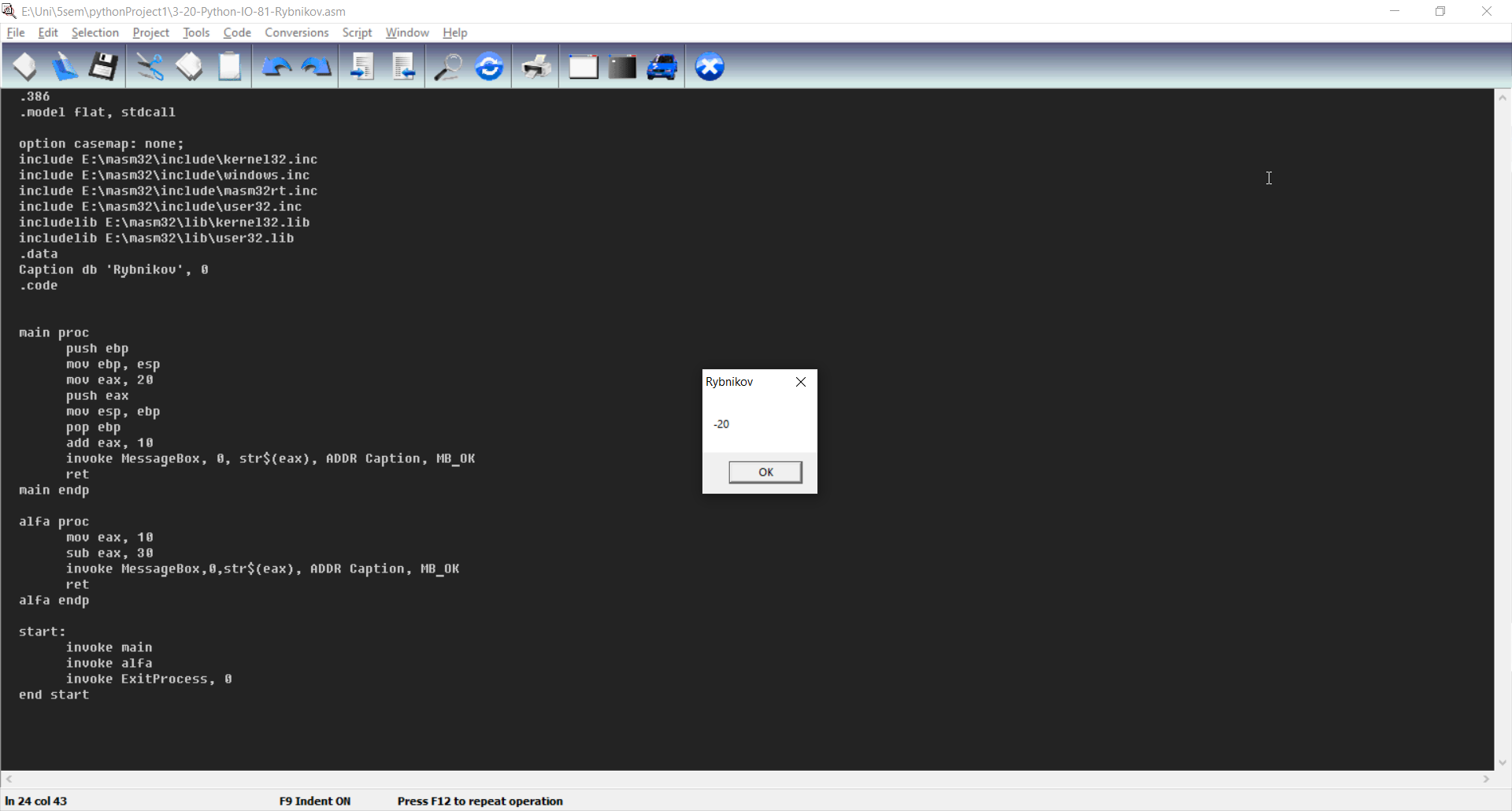
Результати виконання програми на мові Python:



Перевірка роботи вихідного коду:



20 + 10 = 30



10 – 30 = -20

**Приклад 2**

Вхідний код:

def alfa():  
 return 10 - not 30 + not 10 + 10  
  
alfa()

Вихідний код:

.386

.model flat, stdcall

option casemap: none;

include E:\masm32\include\kernel32.inc

include E:\masm32\include\windows.inc

include E:\masm32\include\masm32rt.inc

include E:\masm32\include\user32.inc

includelib E:\masm32\lib\kernel32.lib

includelib E:\masm32\lib\user32.lib

.data

Caption db 'Rybnikov', 0

.code

alfa proc

mov ebx, 10

mov ebx, 30

mov eax, 10

cmp ecx, 0

pushf

xor ecx, ecx

popf

setz cl

mov eax, ebx

cmp ecx, 0

pushf

xor ecx, ecx

popf

setz cl

mov eax, ebx

add eax, 10

invoke MessageBox,0,str$(eax), ADDR Caption, MB\_OK

ret

alfa endp

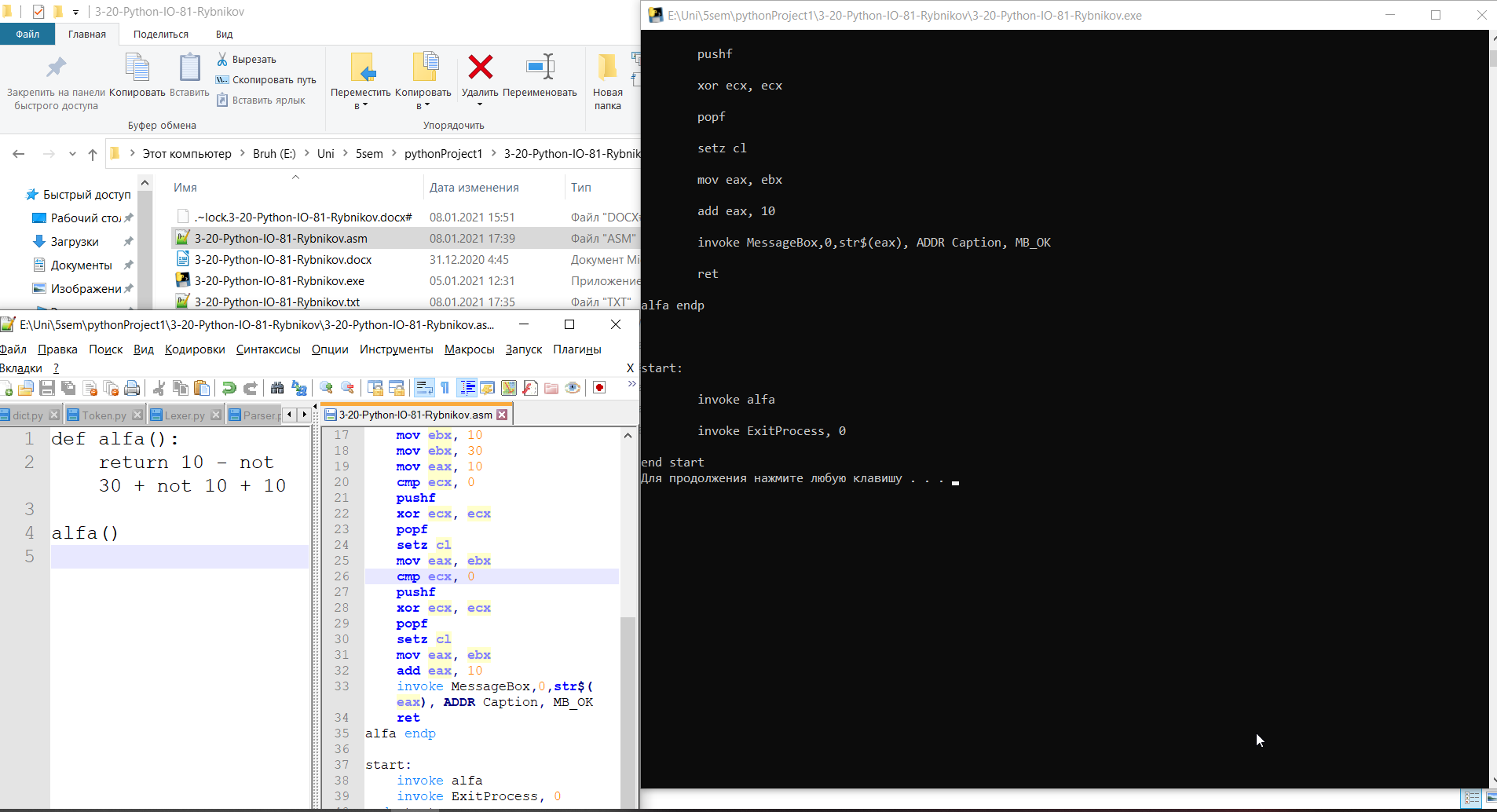
start:

invoke alfa

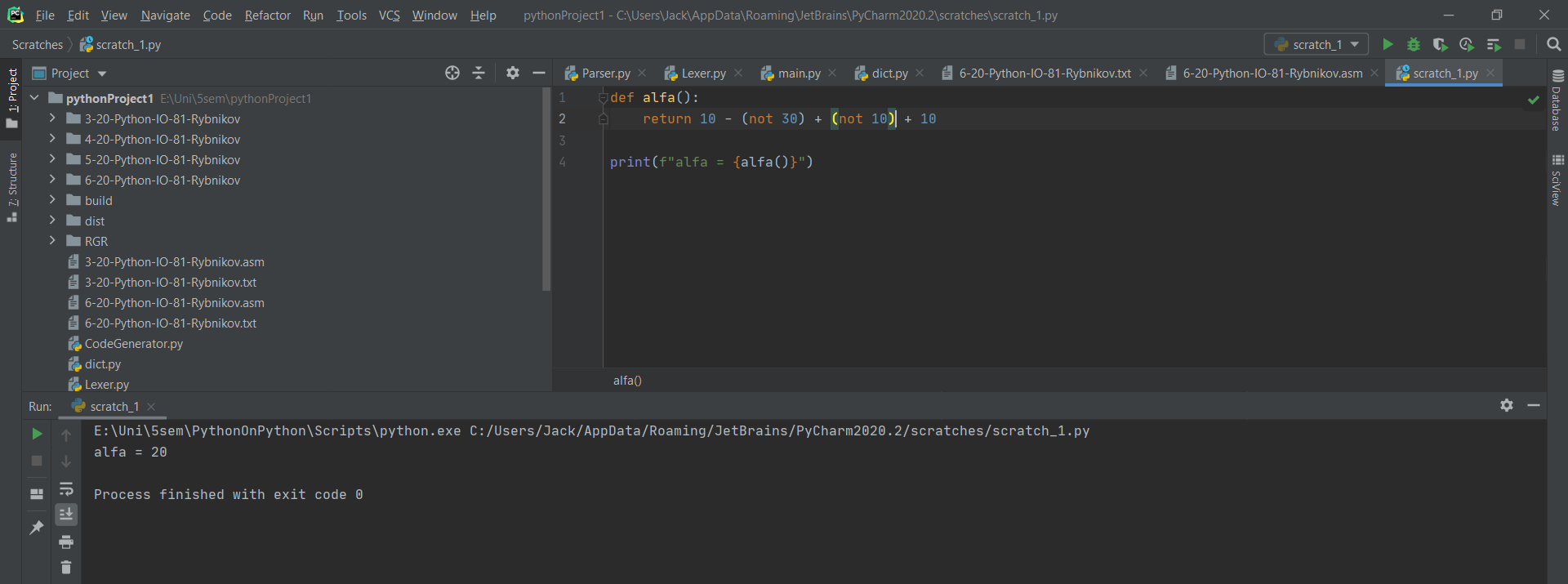
invoke ExitProcess, 0

end start

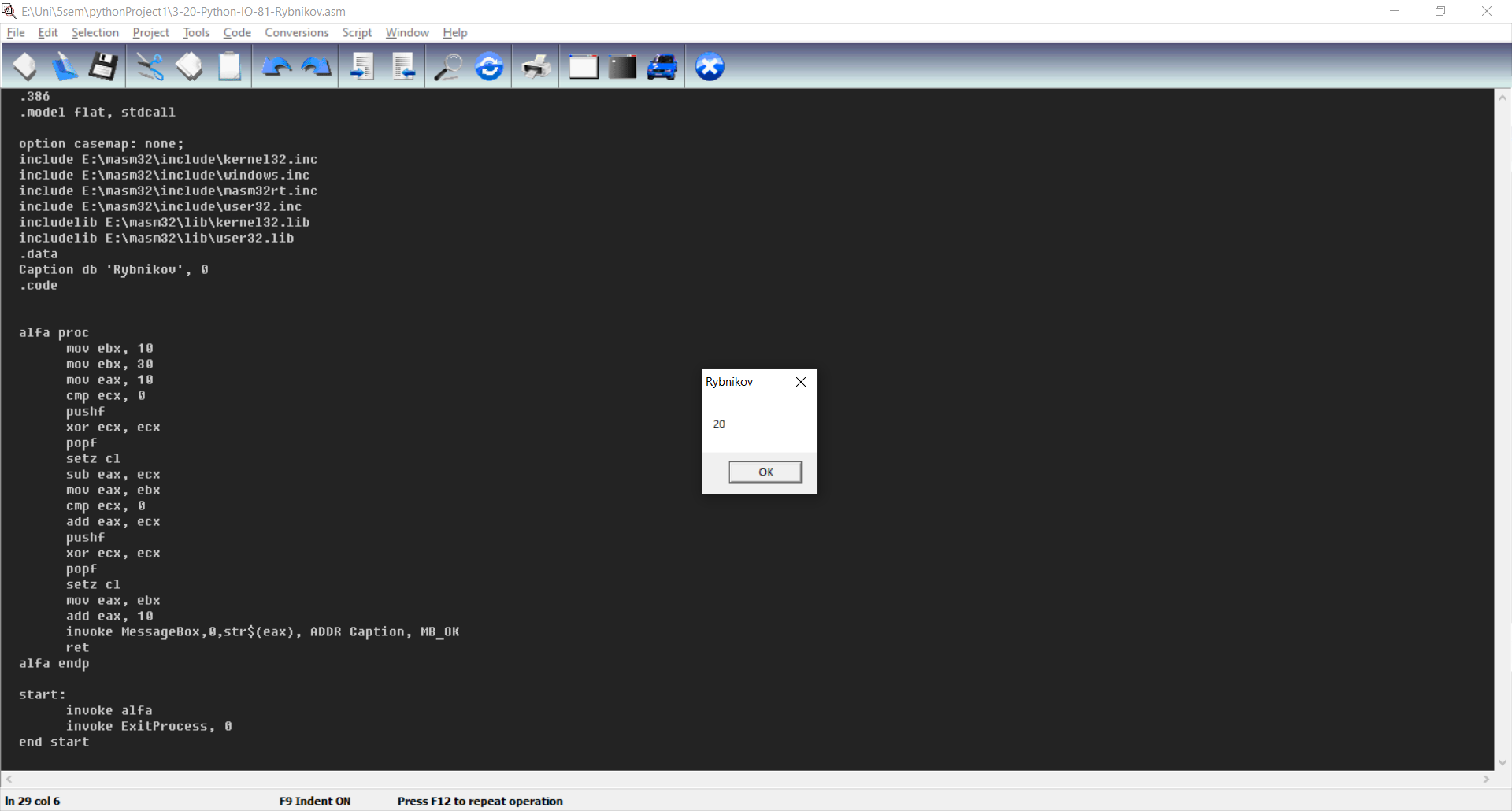
Отримання асемблерного коду:



Результати виконання програми на мові Python:



Перевірка роботи вихідного коду:



10 - not 30 + not 10 + 10

not 30 = 0

not 10 = 0

10 – 0 + 0 + 10 = 20

**Приклад 3**

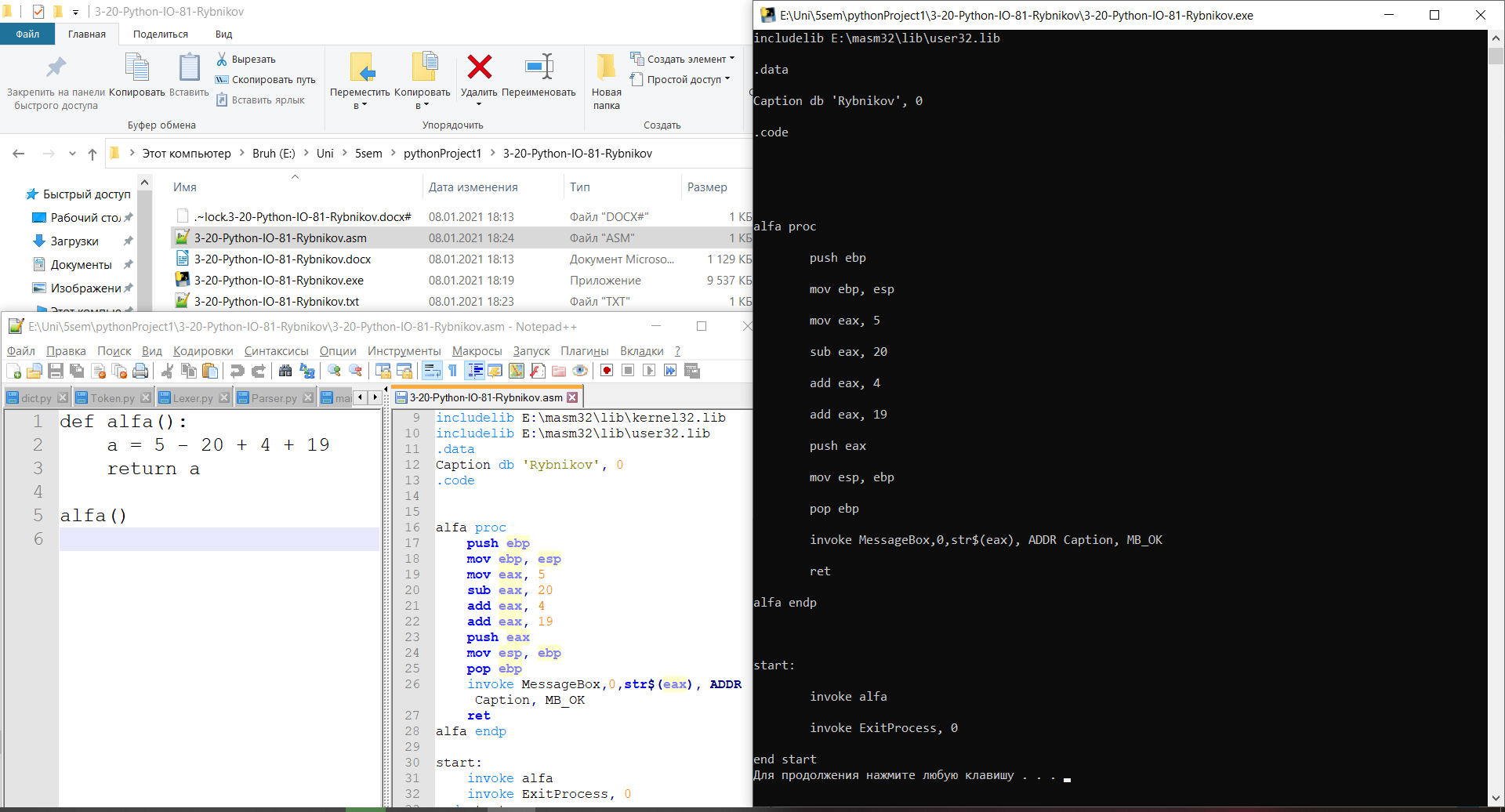
Вхідний код:

def alfa():  
 a = 5 - 20 + 4 + 19  
 return a  
  
alfa()

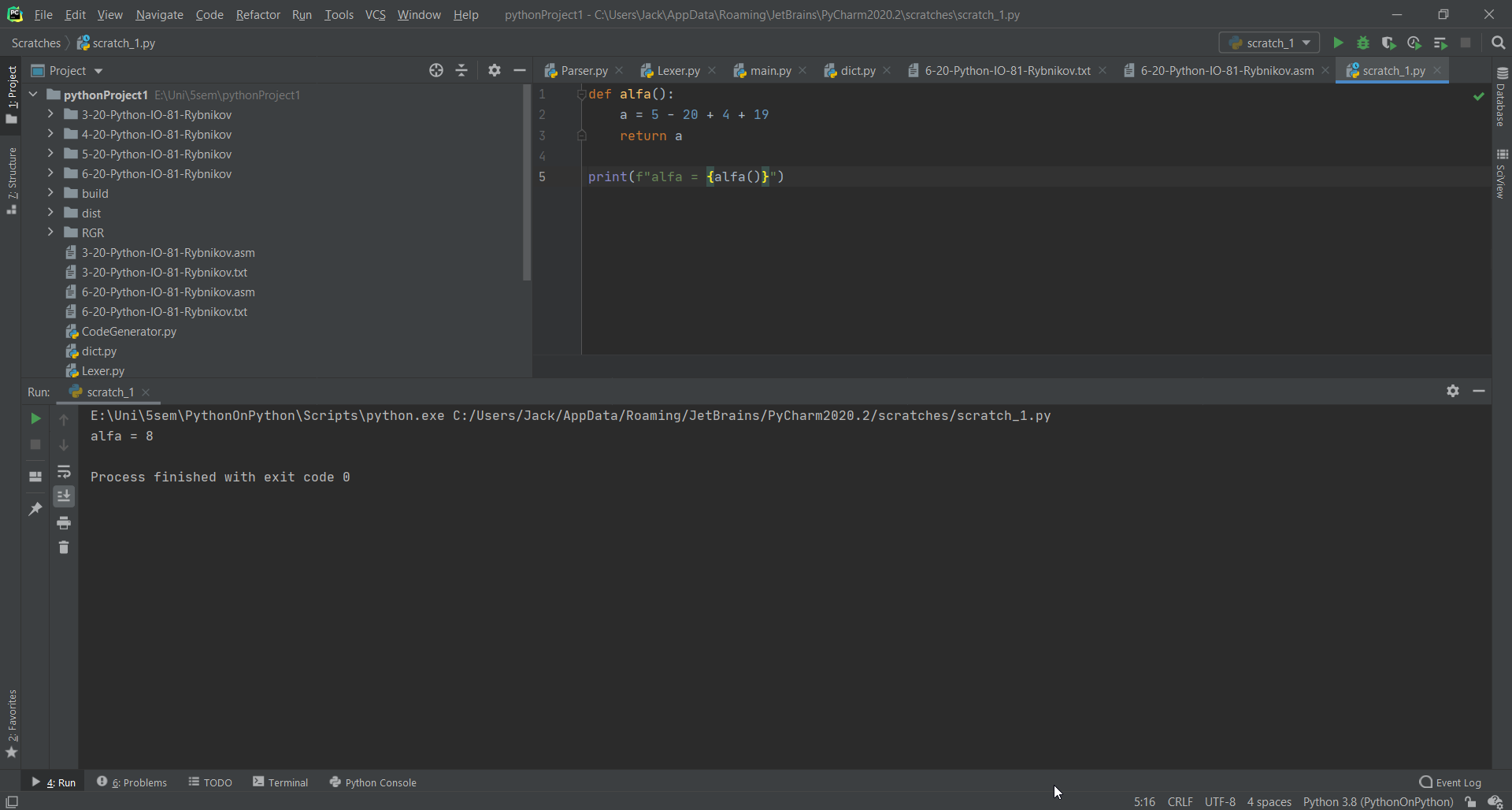
Вихідний код:

.386  
.model flat, stdcall  
  
option casemap: none;  
include E:\masm32\include\kernel32.inc  
include E:\masm32\include\windows.inc  
include E:\masm32\include\masm32rt.inc  
include E:\masm32\include\user32.inc  
includelib E:\masm32\lib\kernel32.lib  
includelib E:\masm32\lib\user32.lib  
.data  
Caption db 'Rybnikov', 0  
.code  
  
  
alfa proc  
 push ebp  
 mov ebp, esp  
 mov eax, 5  
 sub eax, 20  
 add eax, 4  
 add eax, 19  
 push eax   
 mov esp, ebp  
 pop ebp  
 invoke MessageBox,0,str$(eax), ADDR Caption, MB\_OK  
 ret  
alfa endp  
  
start:  
 invoke alfa  
 invoke ExitProcess, 0  
end start

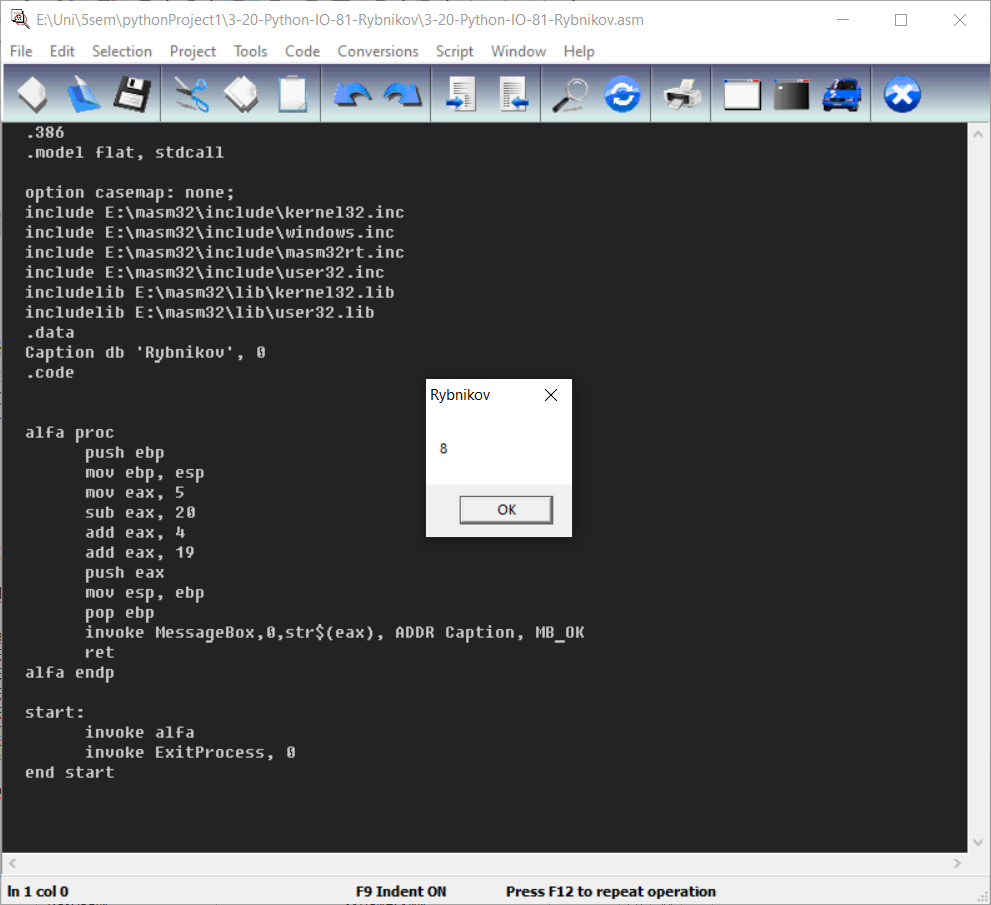
Отримання асемблерного коду:



Результати виконання програми на мові Python:



Перевірка роботи вихідного коду:

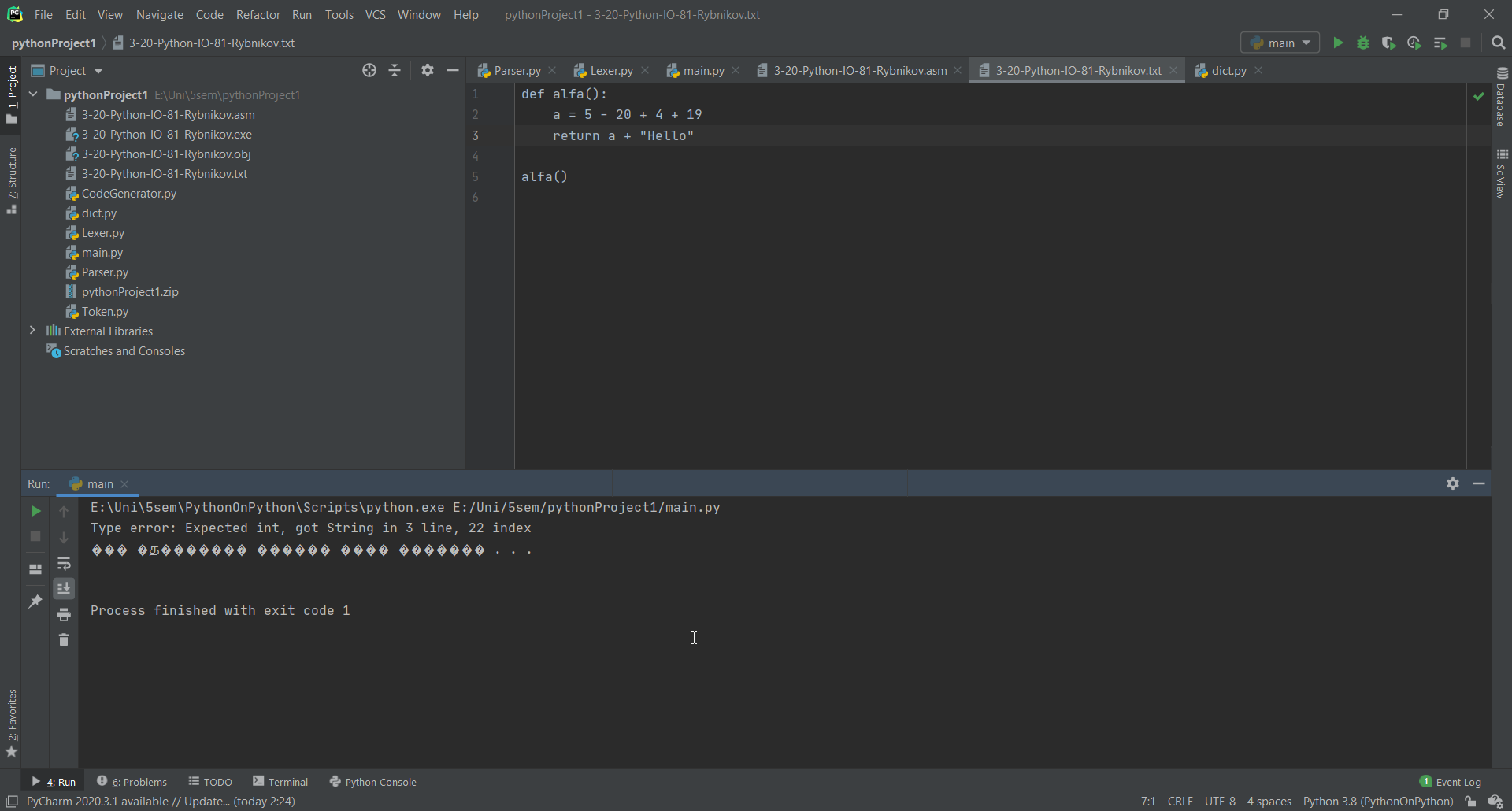
****

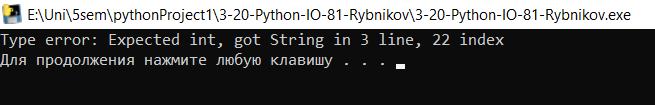
5 - 20 + 4 + 19

5 + 4 – 20 + 19 = 8

**Приклад 4**

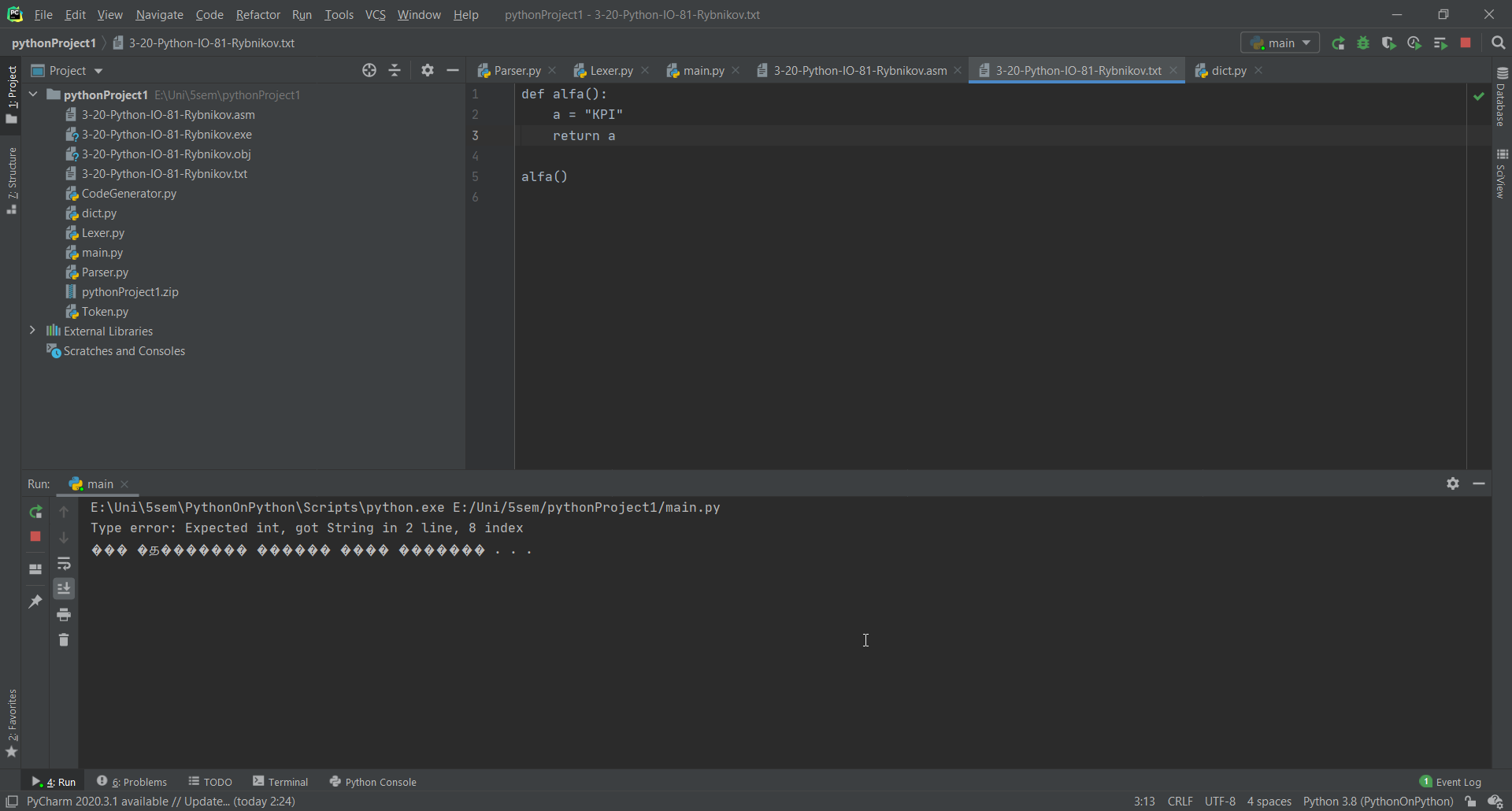
Вхідний код:

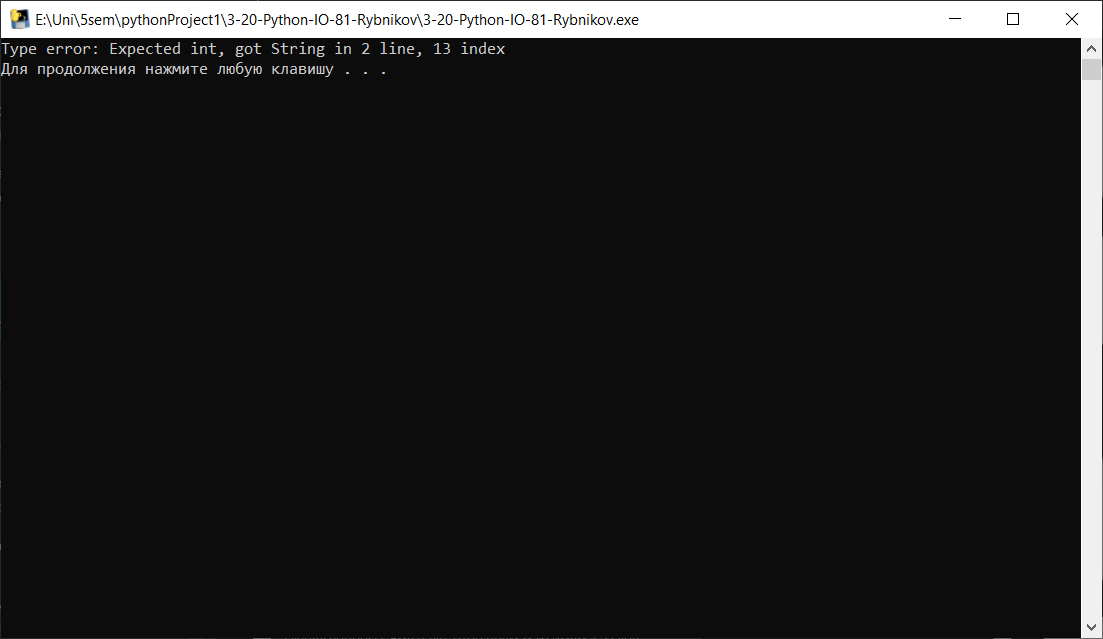
****

****

**Приклад 5**

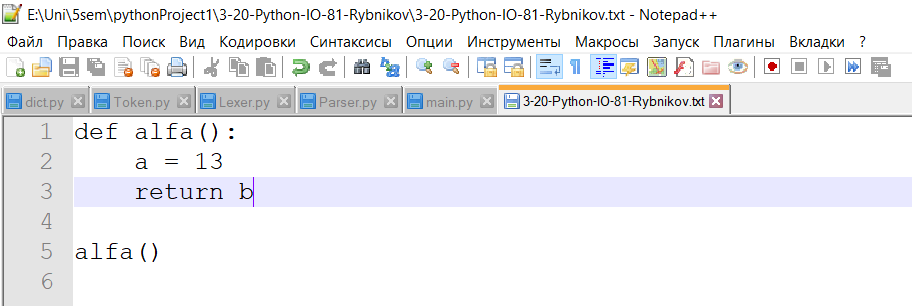
Вхідний код:

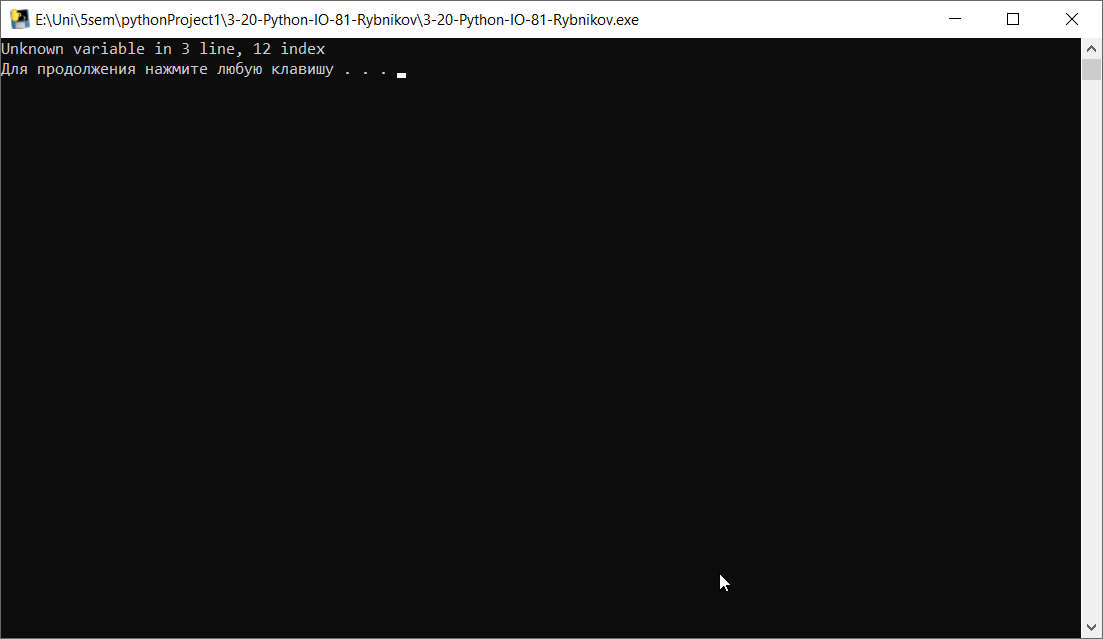




**Приклад 6**

Вхідний код





**Висновок**

До компілятора була додана можливість оброблювати локальні змінні. Також додана нова бінарна операція менше. Оскільки бінарні операції вже додавались, то проблем з ними не виникло.

**Контрольні питання**

**1. Яка основна відмінність обробки функцій в цій лабораторній роботі?**

Основна відмінність даної лабораторної роботи в тому, що ми додали можливість використовувати локальні змінні.

**2. Чи вірний цей вираз: a = 2 \* (b = 2)? Обґрунтуйте свою відповідь.**

Ні, бо не можна оголошувати змінні всередині інших змінних.

**3. Чому присвоювання не обробляється як звичайний бінарний оператор?**

В бінарних операціях ліва і права частина можуть бути якими завгодно, а ось під час присвоєння такого бути не може.

**4. Поясніть порядок розташування операторів в граматиці.**

Порядок розташування ми задавали пріоритетністю.

**5. Навіщо потрібен стек, стекові кадри та пари «ім’я – адреса»? Яка інша**

**назва цих пар?**

Стек – область оперативної пам’яті, яка виділяється для тимчасового зберігання даних програми. Працює за принципом «останнім прийшов — першим пішов» (LIFO – *last in, first out*).

Стекові кадри – механізм передачі аргументів і виділення тимчасової пам'яті (в процедурах мов програмування високого рівня) з використанням системного стека.

Пара «ім’я-адреса» потрібна для реалізації обробки змінних. Ми записуємо змінну та утворюємо пару з адресою значення у стеку.

**6. Яку роль виконує регістр EBP?**

Один із способів використання регістру ebp — зберігання базової адреси стекового кадру. На початку функції стан регістра esp зберігається в ebp і далі вже адреса аргументів функції та локальних змінних відраховується від нього. Далі в процесі роботи функції(та викликаємою нею функціях) esp може мінятись як завгодно, аале адреса локальної змінної в цій функції якою була, такою і залишиться при будь яких змінах esp.

**7. Що таке пролог та епілог функції?**

Пролог зберігає контекст виконання - ті дані (прапори, регістри процесора і т.п.), які можуть бути змінені спричиненої функцією. А епілог відновлює контекст, використовуючи збережені дані.

**8. Як змінюється індекс стеку?**

Ідекс стеку відповідає за наступне вільне місце в стеку, зсув відносно EBP. Наступна вільна позиція завжди після ESP, на ESP — sizeof(тип змінної). Одразу після прологу EBP та ESP однакові, отже початкове значення індексу буде -4. Як тільки змінна записується в стек, значення ESP зменшується на 4.

**9. Який алгоритм використання таблиці змінних?**

У таблиці ми зберігаємо пари змінна-адреса для подальшого їх використання.

**10. Як обробляється пріоритетність?**

Для кожного рівня створюється нове правило граматики.

**11. Чому AST залишилось незмінним?**

Бо ми імплементуємо бінарні операції.

**12. Що означає «встановити біт» (set a bit)?**

Коли ми встановлюємо біт, ми робимо його значення 1. Він виступає у ролі прапора.

**13. Чому використовується прапорець SF?**

SF — Sign Flag або прапор знаку.

**14.Що таке short-circuit evaluation і чому це важливо чи не важливо?**

Short-circuit evaluation використовується у булевих операціях для виразів, щоб коли умова виконалась не робити “зайві” розрахунки тим самим зменшуючи час обчислення і роблячи програми більш швидкодійними.

**15. В чому різниця між je та jmp ?**

jmp – безумовний перехід, виконується завжди.

je – умовний перехід, перевіряє прапор ZF. Якщо рівен 1, то виконується перехід до мітки.