НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

**Лабораторна робота №5**

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

Виконав:

студент 3 курсу

гр. ІО-81

Сорожинський М. М.

Варіант 24

Перевірив:

Павлов В. Г.

Київ 2020 р.

**Тема**

Функції

**Завдання:**

|  |  |
| --- | --- |
| Варіант | Операція |
| 24 | -= |

**Приклади роботи копілятора:**

Вхідний файл:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def gello(op):  return 6 if op else 5  def main():  bin = 0  a = 7  a -= gello(bin)  return a |

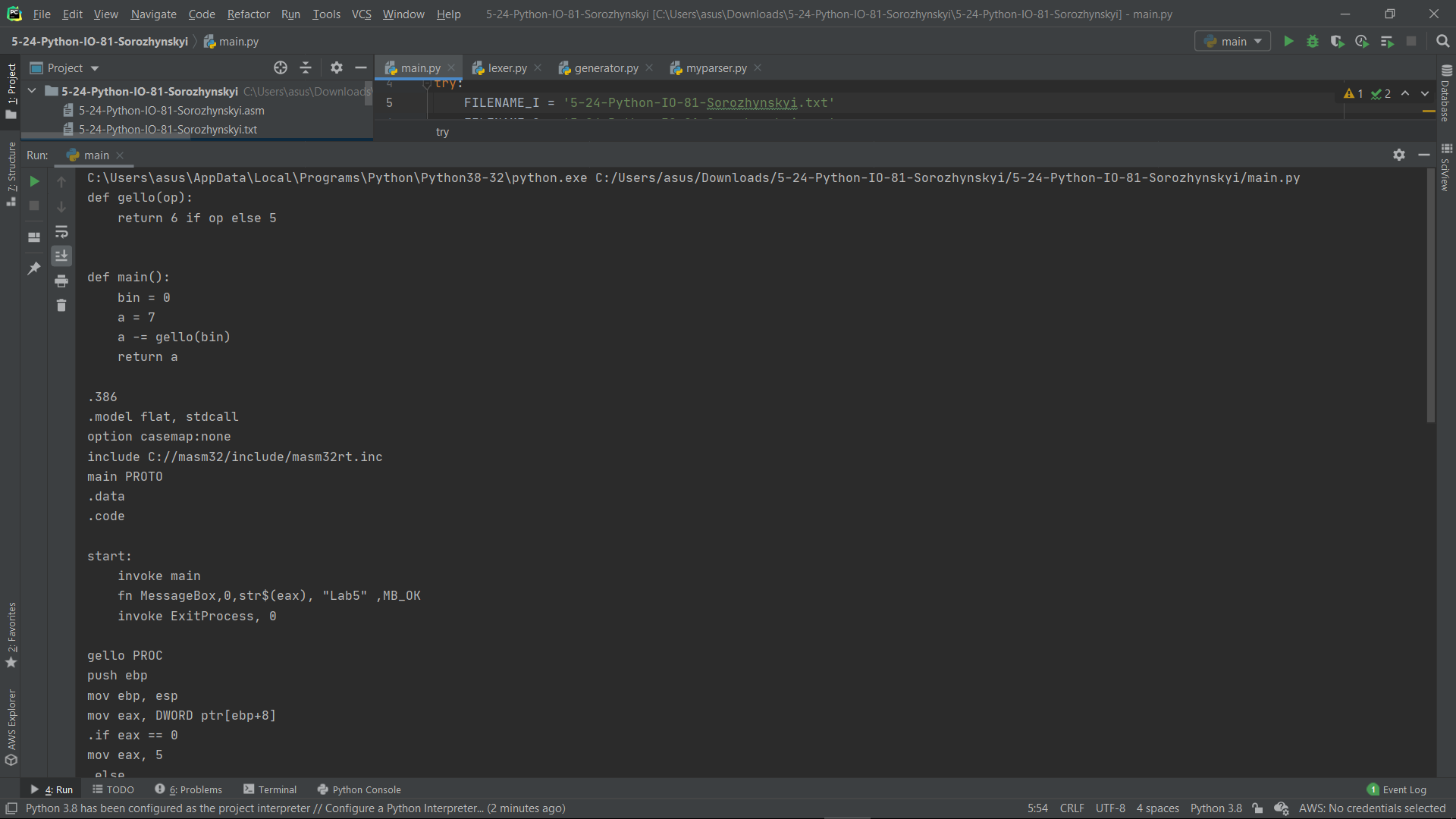
Вихідний файл:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | .386  .model flat, stdcall  option casemap:none  include C://masm32/include/masm32rt.inc  main PROTO  .data  .code  start:  invoke main  fn MessageBox,0,str$(eax), "Lab5" ,MB\_OK  invoke ExitProcess, 0    gello PROC  push ebp  mov ebp, esp  mov eax, DWORD ptr[ebp+8]  .if eax == 0  mov eax, 5  .else  mov eax, 6  .endif  mov esp, ebp  pop ebp  ret  mov esp, ebp  pop ebp  ret  gello ENDP  main PROC  push ebp  mov ebp, esp  mov eax, 0  push eax  mov eax, 7  push eax  mov eax, DWORD ptr[ebp+-8]  push eax  mov eax, DWORD ptr[ebp+-4]  push eax  call gello  add esp, 4  mov ebx, eax  pop eax  sub eax, ebx  mov DWORD ptr[ebp+-8], eax  mov eax, DWORD ptr[ebp+-8]  mov esp, ebp  pop ebp  ret  mov esp, ebp  pop ebp  ret  main ENDP  END start |

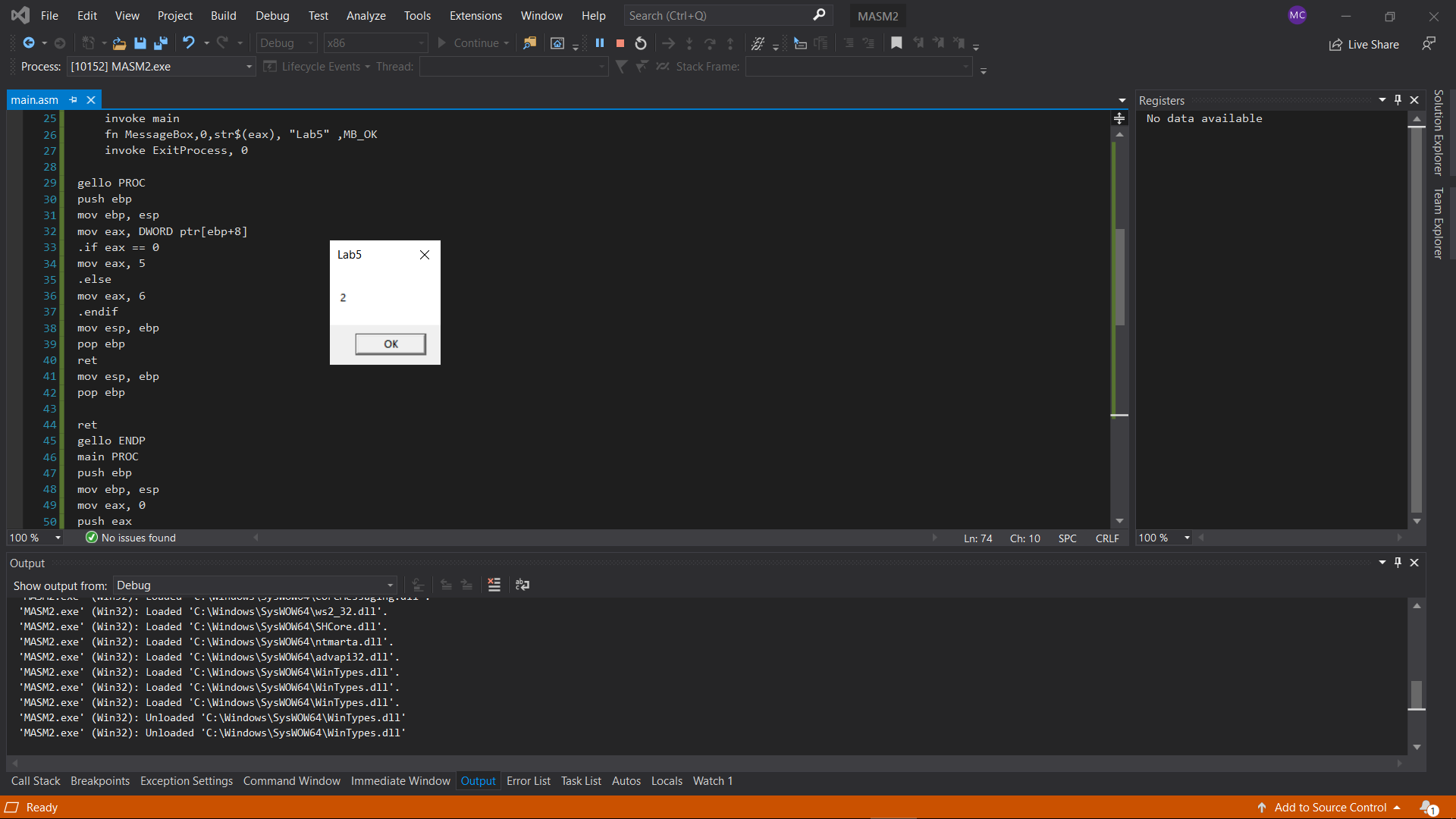
Попереднє обчислення:



Генерація коду:



Виконання коду:

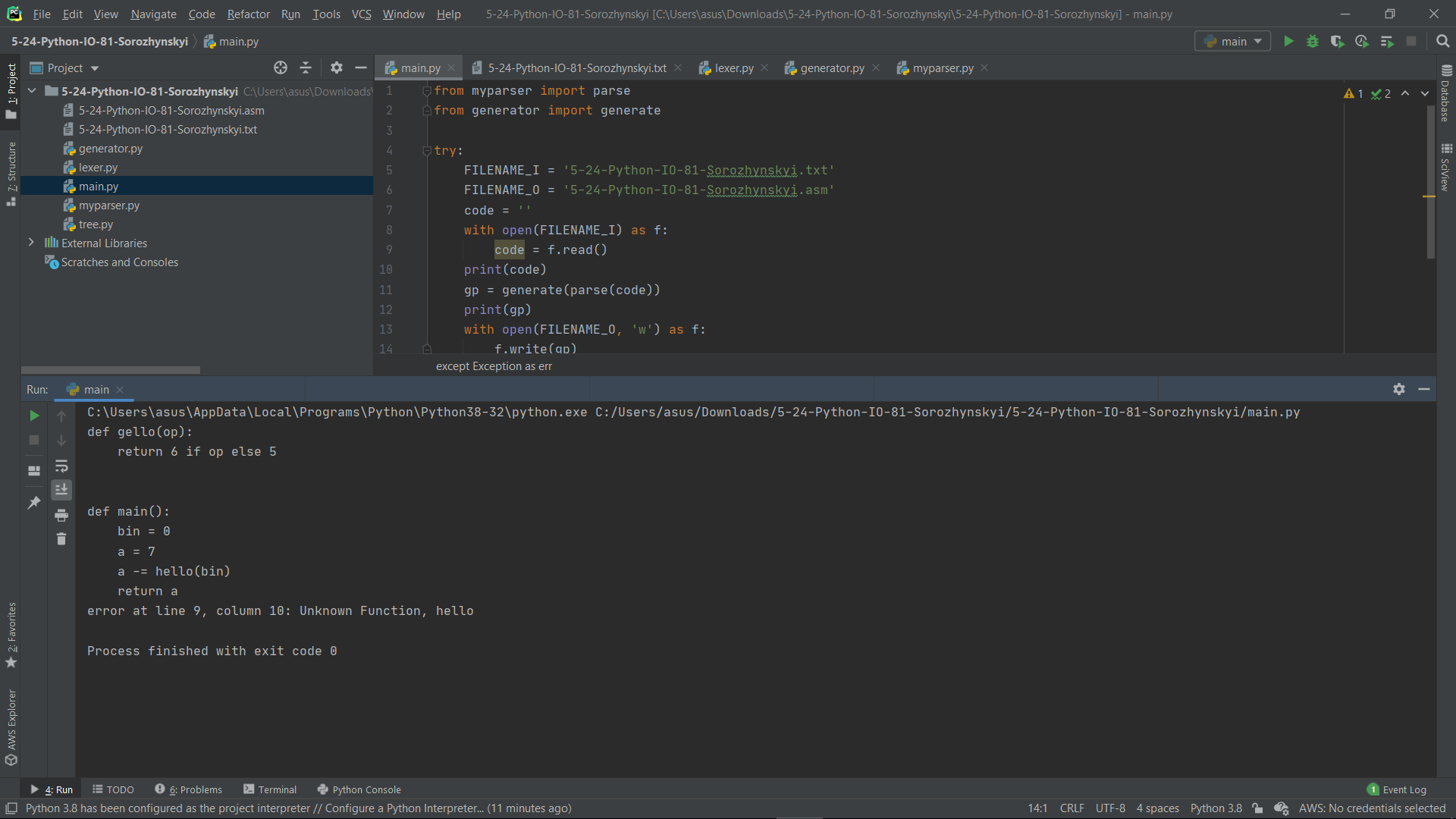


**Тестування**

Вхідні дані(1.py):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def gello(op):  return 6 if op else 5  def main():  bin = 0  a = 7  a -= hello(bin)  return a |

Результат(1.asm):



|  |  |
| --- | --- |
| 1 | error at line 9, column 10: Unknown Function, hello |

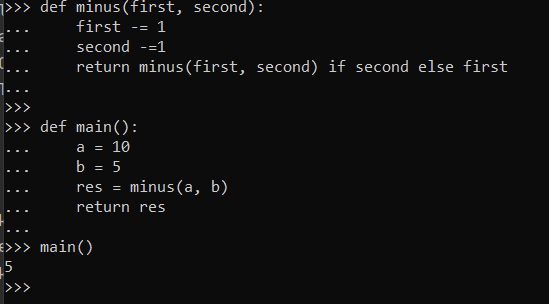
Вхідні дані(2.py):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def minus(first, second):  first -= 1  second -=1  return minus(first, second) if second else first  def main():  a = 10  b = 5  res = minus(a, b)  return res |

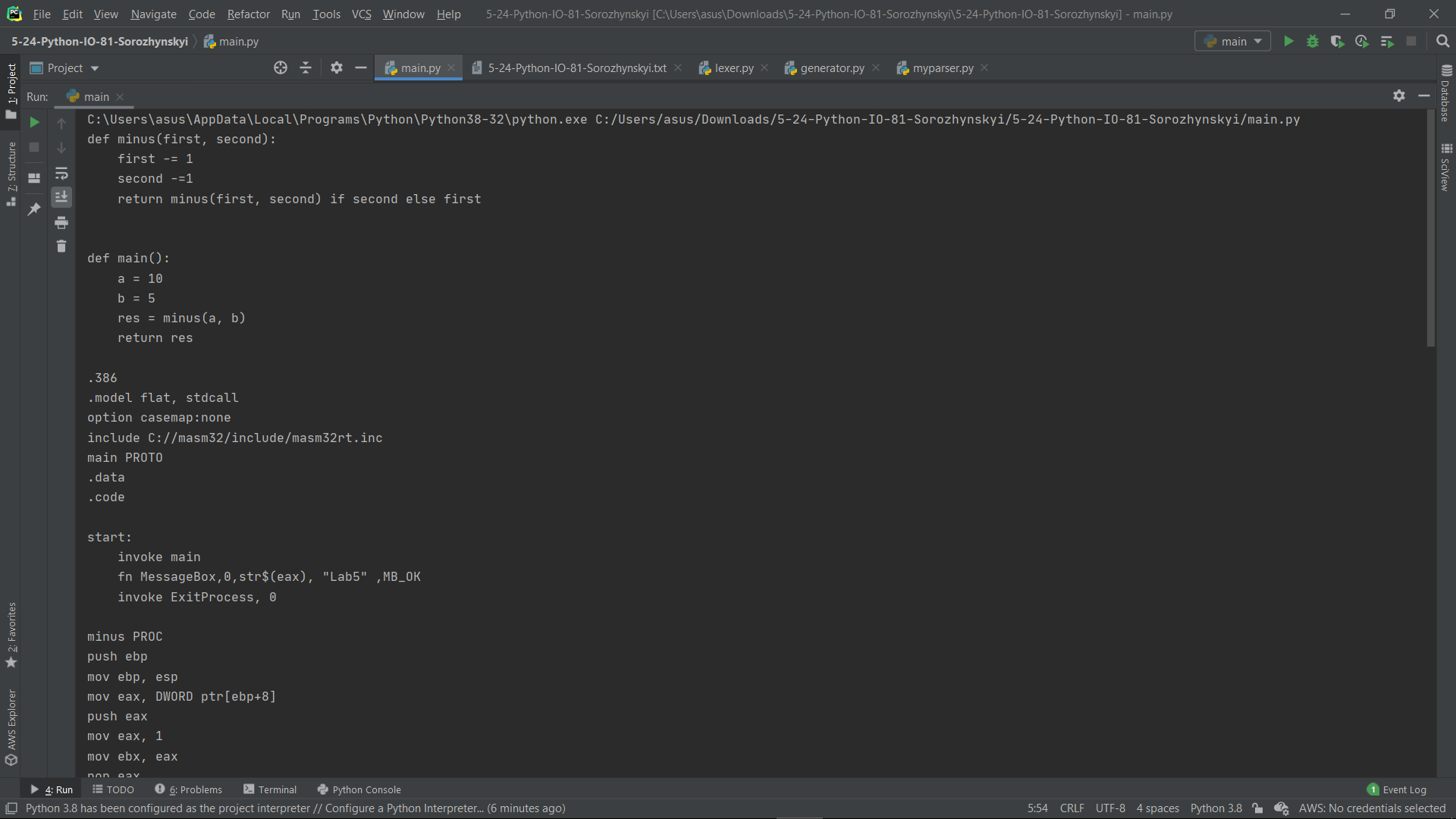
Результат (2.asm):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | minus PROC  push ebp  mov ebp, esp  mov eax, DWORD ptr[ebp+8]  push eax  mov eax, 1  mov ebx, eax  pop eax  sub eax, ebx  mov DWORD ptr[ebp+8], eax  mov eax, DWORD ptr[ebp+12]  push eax  mov eax, 1  mov ebx, eax  pop eax  sub eax, ebx  mov DWORD ptr[ebp+12], eax  mov eax, DWORD ptr[ebp+12]  .if eax == 0  mov eax, DWORD ptr[ebp+8]  .else  mov eax, DWORD ptr[ebp+12]  push eax  mov eax, DWORD ptr[ebp+8]  push eax  call minus  add esp, 8  .endif  mov esp, ebp  pop ebp  ret  mov esp, ebp  pop ebp  ret  minus ENDP  main PROC  push ebp  mov ebp, esp  mov eax, 10  push eax  mov eax, 5  push eax  mov eax, DWORD ptr[ebp+-8]  push eax  mov eax, DWORD ptr[ebp+-4]  push eax  call minus  add esp, 8  push eax  mov eax, DWORD ptr[ebp+-12]  mov esp, ebp  pop ebp  ret  mov esp, ebp  pop ebp  ret  main ENDP |

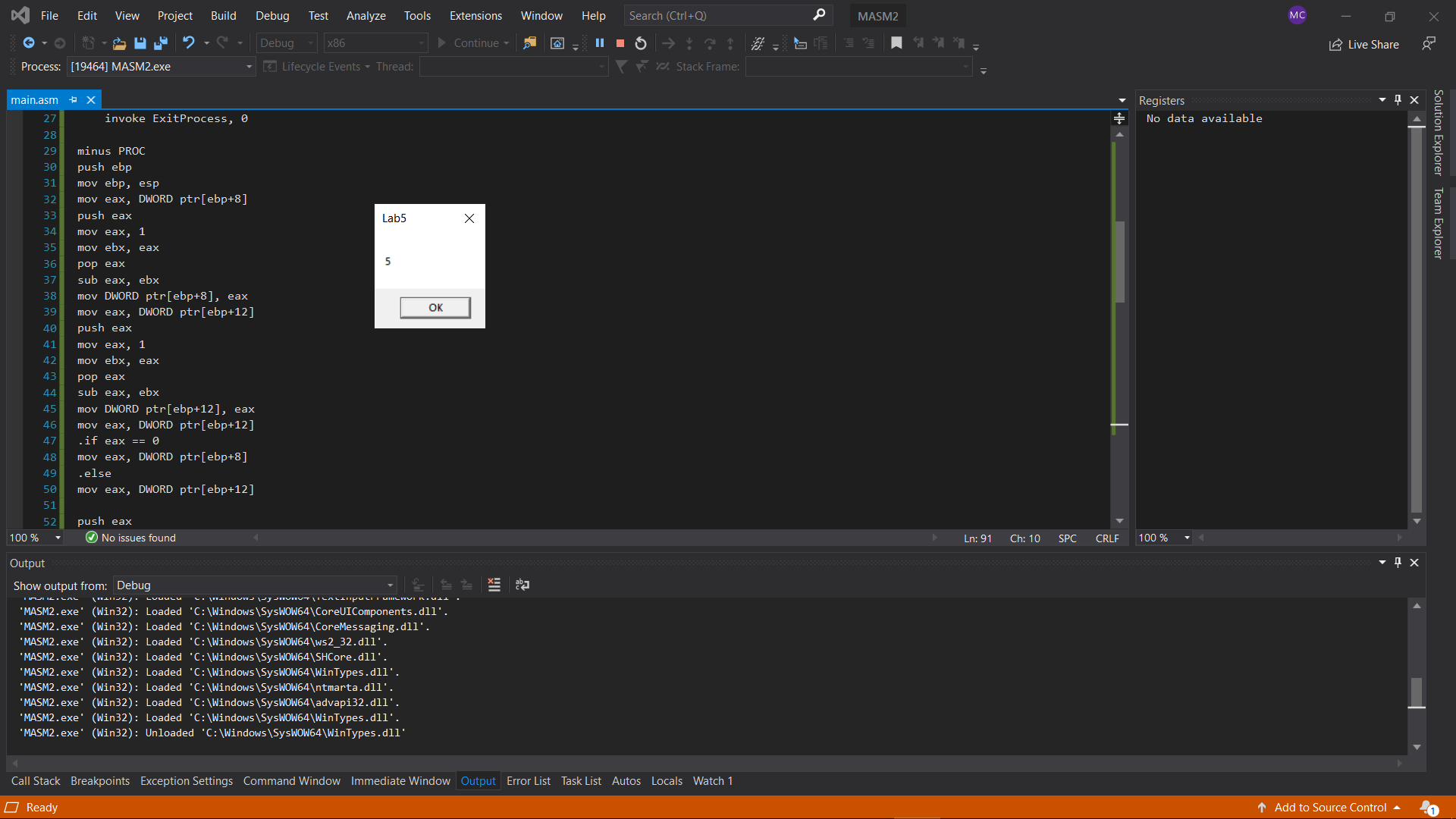
Попереднє обчислення:



Генерація коду:



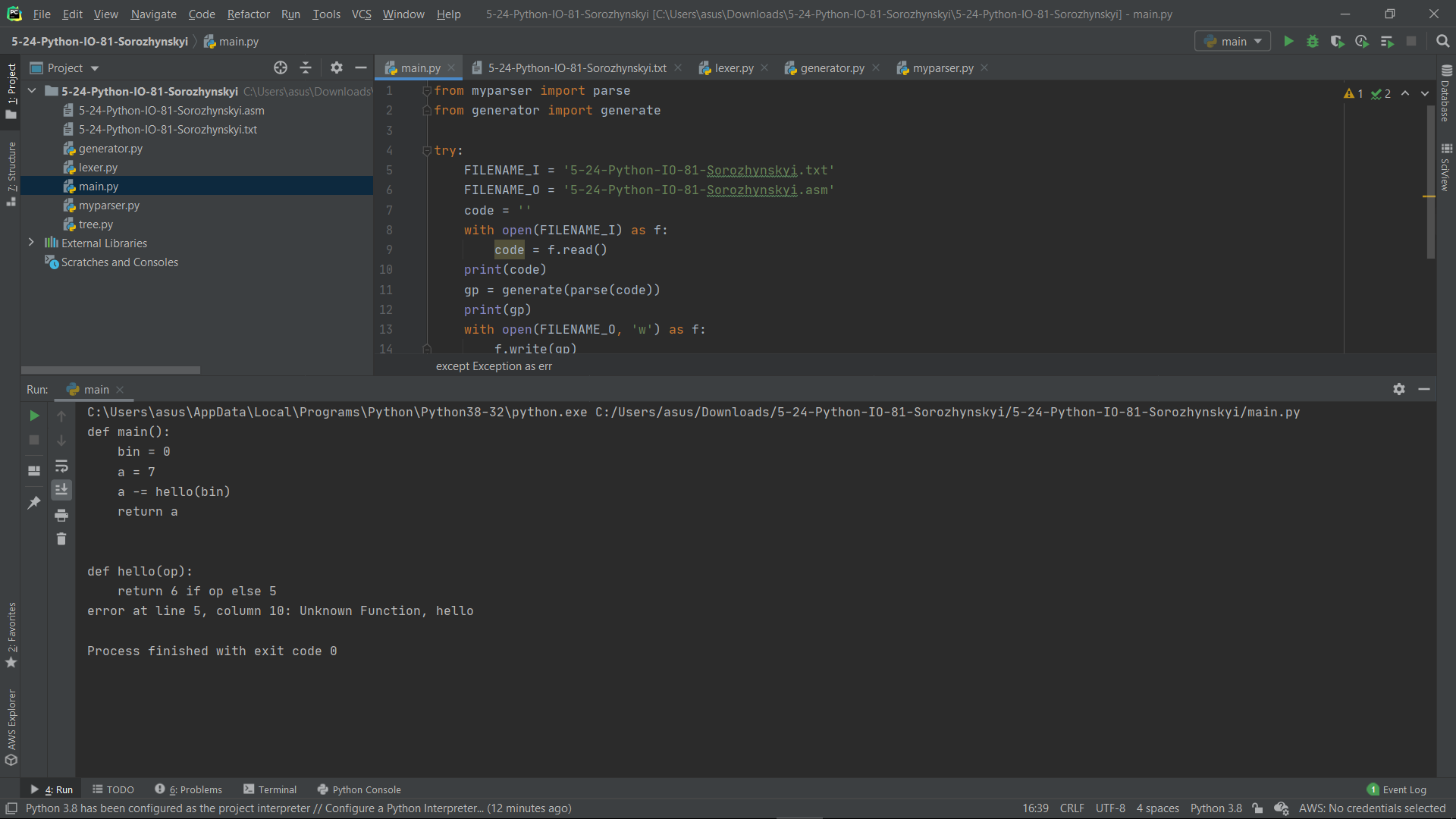
Виконання коду:



Вхідні дані(3.py):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def main():  bin = 0  a = 7  a -= gello(bin)  return a  def gello(op):  return 6 if op else 5 |

Результат(3.asm):



|  |  |
| --- | --- |
| 1 | error at line 5, column 10: Unknown Function, hello |

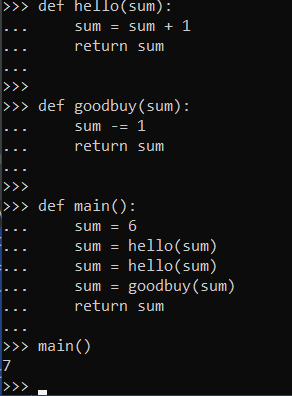
Вхідні дані(4.py):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | def hello(sum):  sum = sum + 1  return sum  def goodbuy(sum):  sum -= 1  return sum  def main():  sum = 6  sum = hello(sum)  sum = hello(sum)  sum = goodbuy(sum)  return sum |

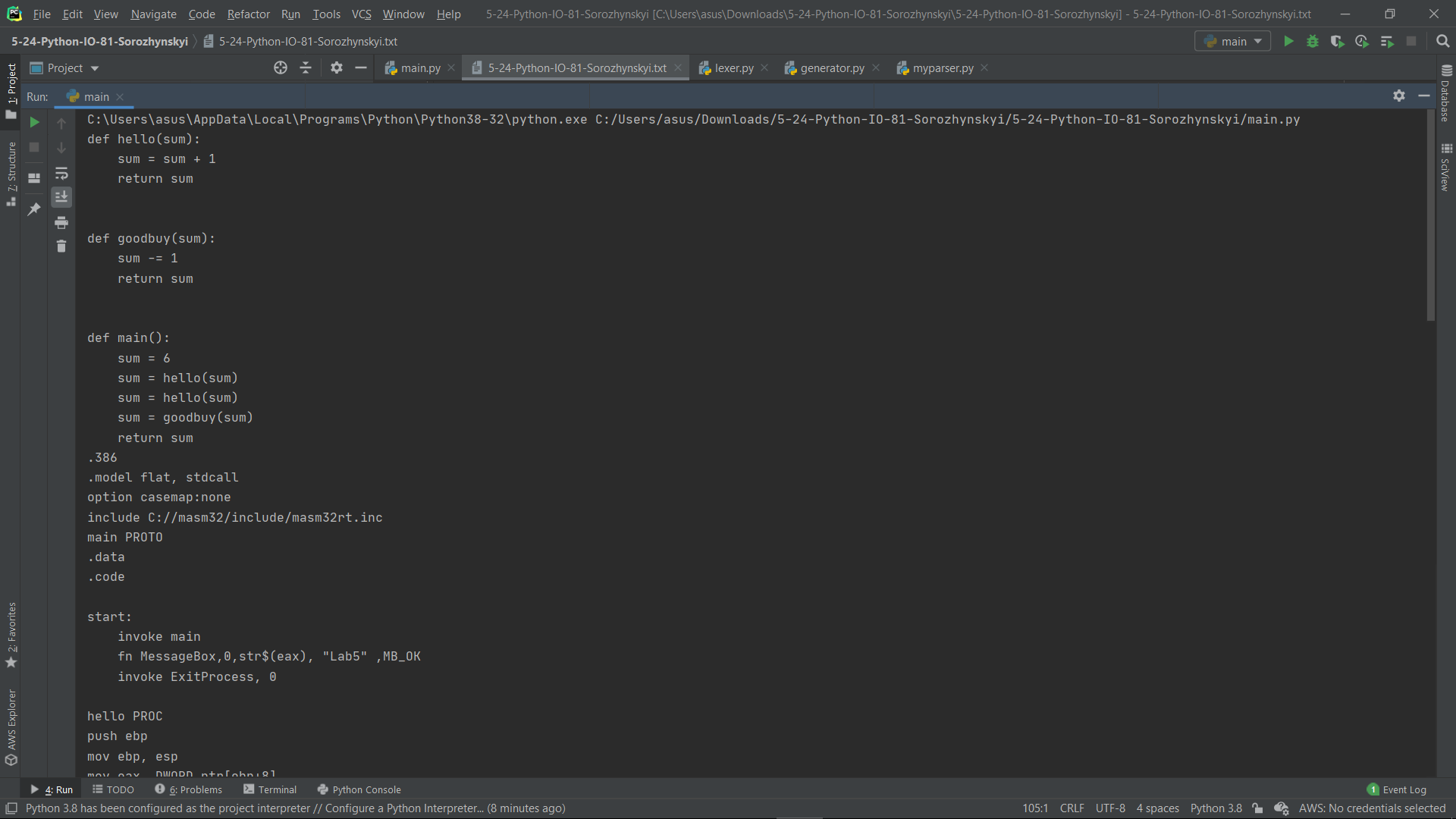
Результат (4.asm):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80 | hello PROC  push ebp  mov ebp, esp  mov eax, DWORD ptr[ebp+8]  push eax  mov eax, 1  mov ebx, eax  pop eax  add eax, ebx  mov DWORD ptr[ebp+8], eax  mov eax, DWORD ptr[ebp+8]  mov esp, ebp  pop ebp  ret  mov esp, ebp  pop ebp  ret  hello ENDP  goodbuy PROC  push ebp  mov ebp, esp  mov eax, DWORD ptr[ebp+8]  push eax  mov eax, 1  mov ebx, eax  pop eax  sub eax, ebx  mov DWORD ptr[ebp+8], eax  mov eax, DWORD ptr[ebp+8]  mov esp, ebp  pop ebp  ret  mov esp, ebp  pop ebp  ret  goodbuy ENDP  main PROC  push ebp  mov ebp, esp  mov eax, 6  push eax  mov eax, DWORD ptr[ebp+-4]  push eax  call hello  add esp, 4  mov DWORD ptr[ebp+-4], eax  mov eax, DWORD ptr[ebp+-4]  push eax  call hello  add esp, 4  mov DWORD ptr[ebp+-4], eax  mov eax, DWORD ptr[ebp+-4]  push eax  call goodbuy  add esp, 4  mov DWORD ptr[ebp+-4], eax  mov eax, DWORD ptr[ebp+-4]  mov esp, ebp  pop ebp  ret  mov esp, ebp  pop ebp  ret  main ENDP |

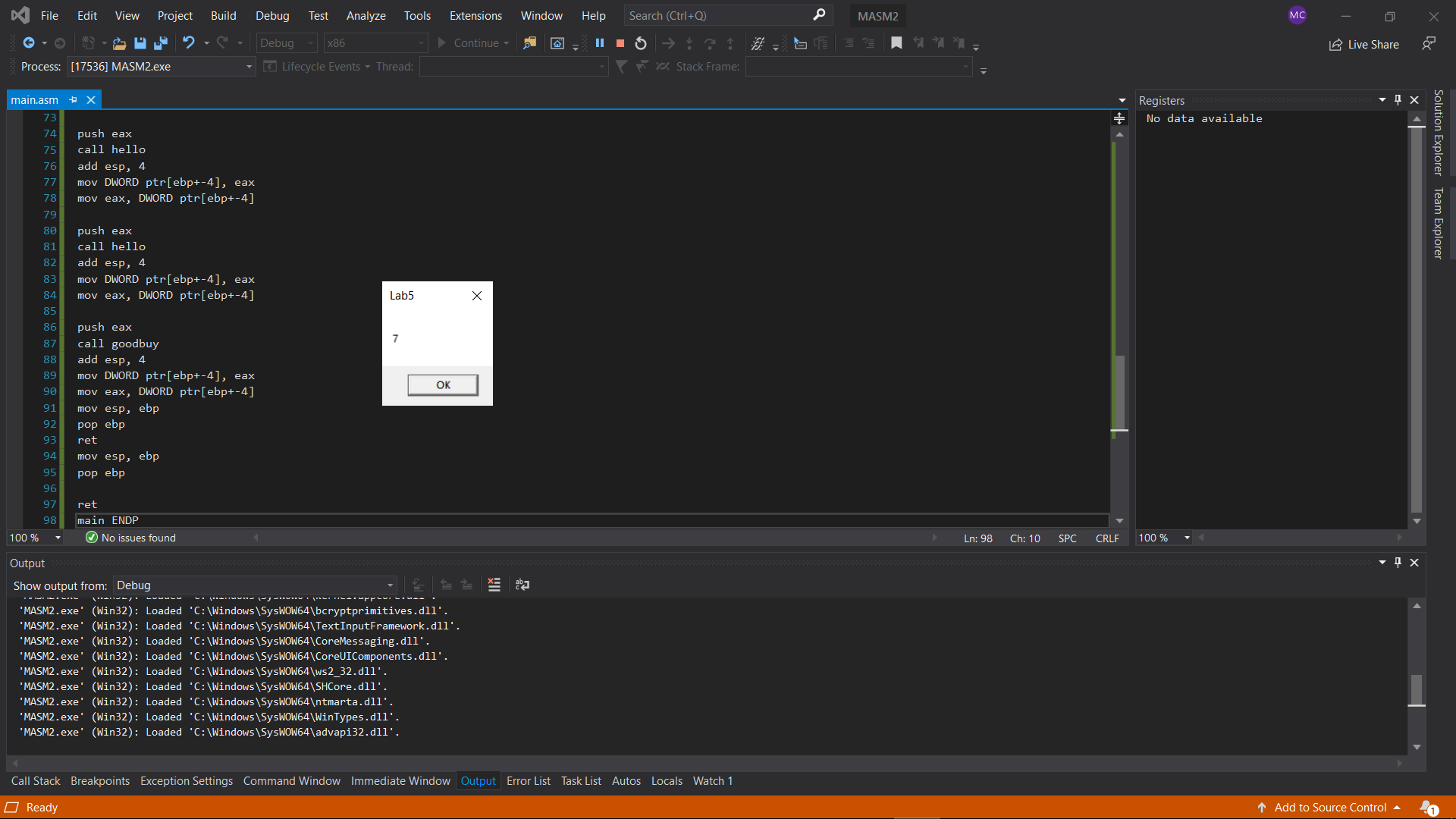
Попереднє обчислення:



Генерація коду:



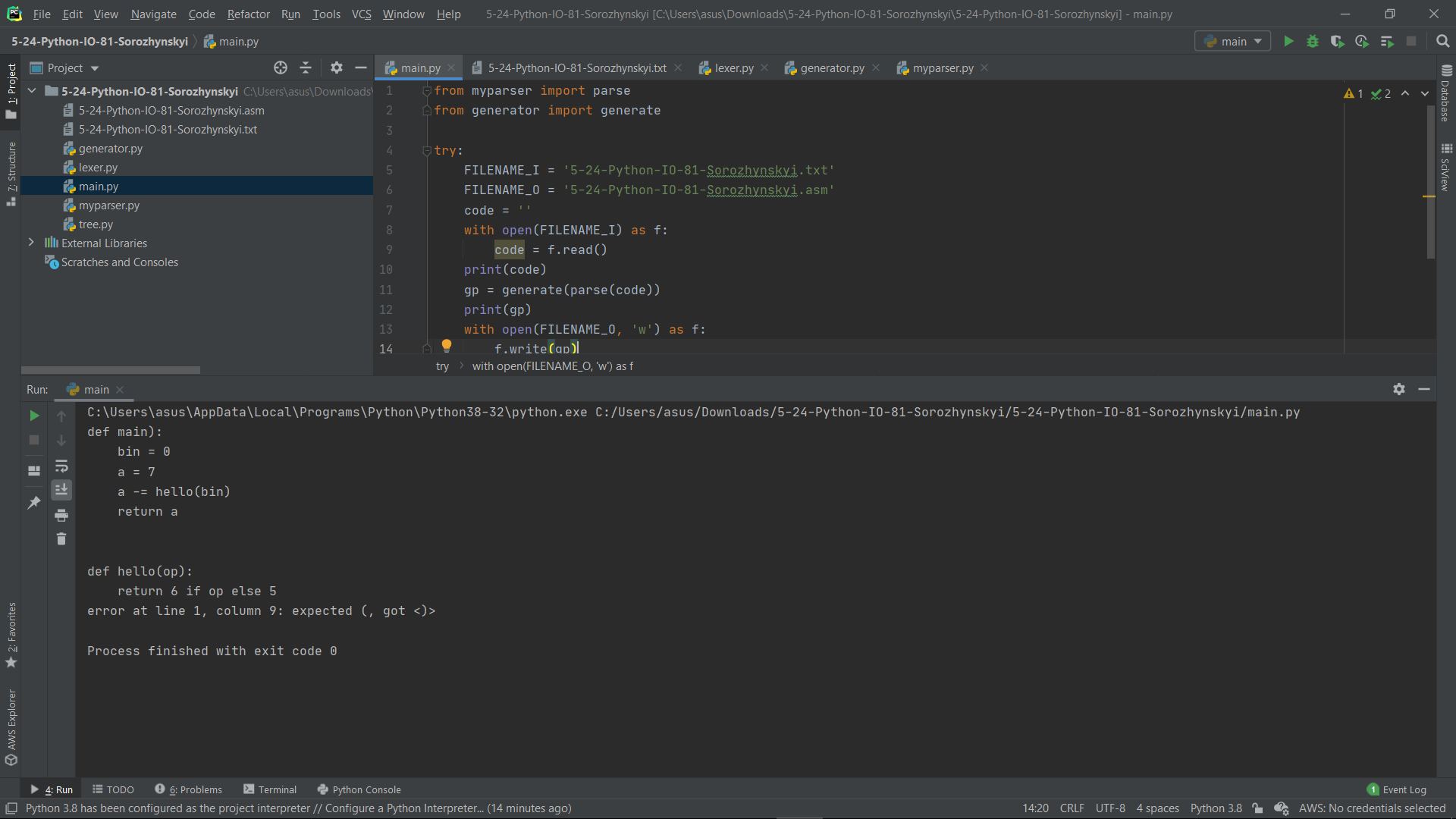
Виконання коду:



Вхідні дані(5.py):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def main):  bin = 0  a = 7  a -= hello(bin)  return a  def hello(op):  return 6 if op else 5 |

Результат(5.asm):



|  |  |
| --- | --- |
| 1 | error at line 1, column 9: expected (, got <)> |

**Лістинг програми компілятора**

main.py

from myparser import parse  
from generator import generate  
  
try:  
 FILENAME\_I = '5-24-Python-IO-81-Sorozhynskyi.txt'  
 FILENAME\_O = '5-24-Python-IO-81-Sorozhynskyi.asm'  
 code = ''  
 with open(FILENAME\_I) as f:  
 code = f.read()  
 print(code)  
 gp = generate(parse(code))  
 print(gp)  
 with open(FILENAME\_O, 'w') as f:  
 f.write(gp)  
except Exception as err:  
 print(f'error at line 1, column 9: expected (, got <)>')

lexer.py

*"""  
Simple lexer library.  
"""*from enum import Enum  
  
HELPERS = ['(', ')', ':', ',']  
OPERATORS = ['-', 'not', '=', '+', '%', '\*']  
  
  
class Tag(Enum):  
 NUM = 'num'  
 CHAR = 'char'  
 ID = 'id'  
 INDENT = 'indent'  
 DEDENT = 'dedent'  
 NEWLINE = 'newline'  
 RETURN = 'return'  
 DEF = 'def'  
 MIN = '-'  
 LOGZAP = 'not'  
 EQUAL = '='  
 PLUS = '+'  
 PROZENT = '%'  
 IF = 'if'  
 ELSE = 'else'  
 EOF = 'eof'  
 MNOZ = '\*'  
  
  
class Token:  
 def \_\_init\_\_(self, tag):  
 self.tag = tag  
  
 def \_\_eq\_\_(self, other):  
 return other is not None and self.tag == other.tag  
  
 def \_\_str\_\_(self):  
 return f'<{self.tag}>'  
  
 def \_\_repr\_\_(self):  
 return self.\_\_str\_\_()  
  
  
class Operator(Token):  
 def \_\_init\_\_(self, val):  
 tag = None  
 if val == '-':  
 tag = Tag.MIN  
 elif val == 'not':  
 tag = Tag.LOGZAP  
 elif val == '=':  
 tag = Tag.EQUAL  
 elif val == '+':  
 tag = Tag.PLUS  
 elif val == '%':  
 tag = Tag.PROZENT  
 elif val == '\*':  
 tag = Tag.MNOZ  
 super().\_\_init\_\_(tag)  
 self.value = val  
  
 def \_\_str\_\_(self):  
 return f'<{self.tag}, {self.value}>'  
  
 def \_\_repr\_\_(self):  
 return self.\_\_str\_\_()  
  
 def \_\_eq\_\_(self, other):  
 return other and self.value == other.value  
  
  
class Num(Token):  
 def \_\_init\_\_(self, val):  
 super().\_\_init\_\_(Tag.NUM)  
 self.value = val  
  
 def \_\_str\_\_(self):  
 return f'<{self.tag}, {self.value}>'  
  
 def \_\_repr\_\_(self):  
 return self.\_\_str\_\_()  
  
 def \_\_eq\_\_(self, other):  
 return other and self.value == other.value  
  
  
class Char(Token):  
 def \_\_init\_\_(self, val):  
 super().\_\_init\_\_(Tag.CHAR)  
 self.value = val  
  
 def \_\_str\_\_(self):  
 return f'<{self.tag}, {self.value}>'  
  
 def \_\_repr\_\_(self):  
 return self.\_\_str\_\_()  
  
  
class Word(Token):  
 def \_\_init\_\_(self, tag, lex):  
 super().\_\_init\_\_(tag)  
 self.lexeme = lex  
  
 def \_\_str\_\_(self):  
 return f'<{self.tag}, {self.lexeme}>'  
  
 def \_\_repr\_\_(self):  
 return self.\_\_str\_\_()  
  
  
class Lexer:  
 def \_\_init\_\_(self, inpstr): # inputstream(iterator)  
 self.inpstr = inpstr  
 self.indstk = [0] # indentstack

self.pended = 0  
 self.words = {} # dict keywords, IDs ->TokenTag  
 self.isnl = True #logicline detection

self.buff = ''  
 self.curr = next(self.inpstr, '')  
 self.line = 1  
 self.wasind = False  
  
 def \_\_iter\_\_(self):  
 return self  
  
 def get\_raw(self):  
 self.curr = next(self.inpstr, '')  
  
 def get(self):  
 if self.buff != '':  
 self.curr = self.buff[0]  
 self.buff = self.buff[1:]  
 else:  
 self.get\_raw()  
  
 def put(self):  
 self.buff += self.curr  
  
 def reserve(self, w):  
 *"""Adds new entry to words, associating the lexeme with the token."""* self.words.setdefault(w.lexeme, w)  
  
 def number(self):  
 *"""Returns NUM token."""* v = 0  
 p = 0  
 if self.curr == '0':  
 self.get()  
  
 if self.curr == 'x':  
 self.get()  
 hex\_digits = 'ABCDEFabcdef'  
 hex\_number = ''  
 while (self.curr.isdigit() or self.curr in hex\_digits) and self.curr != '':  
 hex\_number += self.curr  
 print(hex\_number)  
 self.get()  
 v = int(hex\_number, 16)  
 else:  
 part = False  
 decimal\_power = 0  
 while self.curr.isdigit() or self.curr == '.':  
 if self.curr == '.':  
 part = True  
 self.get()  
 if not part:  
 v = v \* 10 + int(self.curr)  
 else:  
 decimal\_power += 1  
 p = p + int(self.curr)/10\*\*decimal\_power  
 self.get()  
 return Num(v+p)  
  
 def word(self):  
 *"""Returns Word-type token (IDs, keywords)."""* s = ''  
 w = None  
 while self.curr.isalpha():  
 s += self.curr  
 self.get()  
 # self.put()  
 print(s, self.words)  
 w = self.words.get(s)  
 if w is not None:  
 return w  
  
 w = Word(Tag.ID, s)  
 self.words.setdefault(w.lexeme, w)  
 return w  
  
 def char(self):  
 *"""Returns Char token."""* self.get()  
 if (self.curr == '\''  
 or self.curr == '\"'):  
 raise SyntaxError(f'at line {self.line}: expected char, got {self.curr}')  
 c = list(self.curr)  
 self.get()  
 if self.curr != '\'':  
 raise SyntaxError(f'at line {self.line}: expected quote, got {self.curr}')  
  
 return Char(c)  
  
 def cut\_lines(self):  
 while self.isnl and (self.curr == '\n' or self.curr == ' '):  
 if self.curr == '\n':  
 self.line += 1  
 self.get()  
 if self.curr == ' ' and self.is\_indent():  
 self.get()  
 break  
 self.cut\_spaces()  
  
 def cut\_spaces(self):  
 spaces = 0  
 while self.curr == ' ':  
 self.get()  
 spaces += 1  
 return spaces  
  
 def is\_indent(self):  
 if self.curr != ' ':  
 return False  
 tmp = self.curr  
 while self.curr == ' ':  
 self.get\_raw()  
 self.put()  
 ch = self.curr  
 self.curr = tmp  
 return not (ch == '\n' or ch == '')  
  
 def newline(self):  
 self.line += 1  
 self.isnl = True  
 self.wasind = False  
 self.get()  
 return Token(Tag.NEWLINE)  
  
 def end(self):  
 if self.pended > 0:  
 self.pended -= 1  
 return Token(Tag.DEDENT)  
 return Token(Tag.EOF)  
  
 def other(self):  
 if self.curr.isdigit():  
 return self.number()  
 if self.curr in OPERATORS:  
 return self.operator()  
 if self.curr.isalpha():  
 return self.word()  
 if self.curr == '\'':  
 return self.char()  
 if self.curr in HELPERS:  
 return self.helper()  
 raise SyntaxError(f'illegal character, at line {self.line}: {self.curr}')  
  
 def operator(self):  
 op = Operator(self.curr)  
 self.get()  
 return op  
  
 def helper(self):  
 hl = Token(self.curr)  
 self.get()  
 return hl  
  
 def indent(self):  
 self.wasind = False  
 self.isnl = False  
 self.pended += 1  
 return Token(Tag.INDENT)  
  
 def dedent(self):  
 self.wasind = False  
 self.indstk.pop()  
 self.pended -= 1  
 return Token(Tag.DEDENT)  
  
 def dent(self):  
 width = self.cut\_spaces()  
 if width > self.indstk[-1]:  
 self.indstk.append(width)  
 return self.indent()  
 if width < self.indstk[-1] and width == self.indstk[-2]:  
 return self.dedent()  
 if width == self.indstk[-1]:  
 self.wasind = True  
 return self.\_\_next\_\_()  
  
 def \_\_next\_\_(self):  
 *"""Returns the next token."""* self.cut\_lines()  
  
 if self.curr == '':  
 return self.end()  
  
 if self.isnl and self.curr == ' ':  
 return self.dent()  
  
 if self.isnl and not self.curr == '\n' and len(self.indstk) > 1 and not self.wasind:  
 return self.dedent()  
 # elif self.isnl and len(self.indstk) > 1 :  
 # self.indstk.pop()  
 # self.isnl = False  
 # return self.dedent()  
 # print(self.curr, self.isnl, self.is\_indent())  
  
 self.isnl = False  
  
 self.cut\_spaces()  
  
 if not self.isnl and self.curr == '\n':  
 return self.newline()  
 return self.other()

myparser.py:

*"""  
Parser library.  
"""*from lexer import \*  
from tree import \*  
  
functions = []  
  
  
class Parser:  
 def \_\_init\_\_(self, lexer):  
 self.lexer = lexer  
 self.curr = next(self.lexer, Token(Tag.EOF))  
  
 def match(self, tag):  
 *"""Matches current token tag with the given and advances it."""* if self.curr is None or self.curr.tag != tag:  
 raise SyntaxError(f'at line {self.lexer.line}: expected {tag}, got {self.curr}')  
 self.curr = next(self.lexer, Token(Tag.EOF))  
  
 def module(self):  
 *"""Generates Module root node."""* funcs = []  
 while self.curr.tag != Tag.EOF:  
 funcs.append(self.func())  
 return Module(funcs)  
  
 def func(self):  
 *"""Generates FunctionDef node."""* global functions  
 params = []  
 self.match(Tag.DEF)  
 name = self.curr.lexeme  
 self.match(Tag.ID)  
 self.match('(')  
 if self.curr.tag == Tag.ID:  
 params.append(self.curr.lexeme)  
 self.match(Tag.ID)  
 while self.curr.tag != ')':  
 self.match(',')  
 params.append(self.curr.lexeme)  
 self.match(Tag.ID)  
 functions.append((name, len(params)))  
 self.match(')')  
 self.match(':')  
 body = []  
 if self.curr.tag == Tag.NEWLINE:  
 self.match(Tag.NEWLINE)  
 body = self.func\_body\_suite()  
 else:  
 body = list(self.stmt())  
 return FunctionDef(name, params, body)  
  
 def func\_body\_suite(self):  
 *"""Matches function body and generates related nodes."""* body = []  
 self.match(Tag.INDENT)  
 while not self.curr.tag == Tag.DEDENT:  
 body.append(self.stmt())  
 self.match(Tag.DEDENT)  
  
 #if self.curr.tag == Tag.NEWLINE:  
 # self.match(Tag.NEWLINE)  
 # self.match(Tag.INDENT)  
 # body.append(self.stmt())  
 # while not self.curr.tag == Tag.DEDENT:  
 # self.match(Tag.NEWLINE)  
 # body.append(self.stmt())  
 # self.match(Tag.DEDENT)  
 #else:  
 # body.append(self.stmt())  
 #  
 return body  
  
 def stmt(self):  
 # print(self.curr.tag)  
 *""" Generate statement"""* st = 0  
 if self.curr.tag == Tag.RETURN:  
 st = self.return\_stmt()  
 elif self.curr.tag == Tag.ID:  
 st = self.assign()  
 else:  
 raise SyntaxError(f'at line {self.lexer.line}: expected variable or return, got {self.curr}')  
 self.match(Tag.NEWLINE)  
 return st  
  
 def assign(self):  
  
 id = self.curr.lexeme  
 # print(id)  
 self.match(Tag.ID)  
 if self.curr.tag == Tag.EQUAL:  
 self.match(Tag.EQUAL)  
 return Assign(id, self.expression())  
 elif self.curr.tag == Tag.MIN and self.lexer.curr == '=':  
 self.match(Tag.MIN)  
 self.match(Tag.EQUAL)  
 return Assign(id, Bin\_Op(Id(id), self.expression(), '-'))  
 elif self.curr.tag == Tag.PLUS and self.lexer.curr == '=':  
 self.match(Tag.PLUS)  
 self.match(Tag.EQUAL)  
 return Assign(id, Bin\_Op(Id(id), self.expression(), '+'))  
  
 def div\_mul(self):  
 left = self.term()  
 while self.curr.tag == Tag.PROZENT or self.curr.tag == Tag.MNOZ:  
 op = '%' if self.curr == Tag.PROZENT else '\*'  
 self.match(self.curr.tag)  
 right = self.term()  
 left = Bin\_Op(left, right, op)  
 return left  
  
 def plus\_minus(self):  
 left = self.div\_mul()  
 while self.curr.tag == Tag.MIN or self.curr.tag == Tag.PLUS:  
 op = '-' if self.curr.tag == Tag.MIN else '+'  
 self.match(self.curr.tag)  
 right = self.div\_mul()  
 # print(op)  
 left = Bin\_Op(left, right, op)  
 return left  
  
 def operator\_not(self):  
 if self.curr.tag == Tag.LOGZAP:  
 self.match(Tag.LOGZAP)  
 return Unary\_Op(self.expression(), 'not')  
 return self.plus\_minus()  
  
 def expression(self):  
 true\_con = self.operator\_not()  
 if self.curr.tag == Tag.IF:  
 self.match(Tag.IF)  
 condition = self.operator\_not()  
 self.match(Tag.ELSE)  
 false\_con = self.expression()  
 return Ternary(true\_con, false\_con, condition)  
 return true\_con  
  
 def term(self):  
 # print(self.curr)  
 name = self.curr  
 if self.curr.tag == Tag.NUM:  
 self.match(Tag.NUM)  
 return Constant(name.value)  
 elif self.curr.tag == Tag.CHAR:  
 self.match(Tag.CHAR)  
 return Constant(name.value)  
 elif self.curr.tag == Tag.ID:  
 id = self.curr.lexeme  
 self.match(Tag.ID)  
 if self.curr.tag == '(':  
 self.match('(')  
 arguments = []  
 if self.curr.tag != ')':  
 arguments.append(self.expression())  
 while self.curr.tag != ')':  
 self.match(',')  
 arguments.append(self.expression())  
 self.match(')')  
 if (id, len(arguments)) not in functions:  
 raise SyntaxError(f'at line {self.lexer.line}: Unknown Function, {id}')  
 return CallFunc(id, arguments)  
 #if self.curr.tag == '-' and self.lexer.buff ==  
 return Id(id)  
  
  
 # elif self.curr.tag == Tag.LOGZAP:  
 # self.match(Tag.LOGZAP)  
 # return Unary\_Op(self.expression(), 'not')  
 else:  
 return 'hz'  
  
 def return\_stmt(self):  
 *"""Generates Return node."""* self.match(Tag.RETURN)  
 return Return(self.expression())  
  
  
def parse(name):  
 *"""Parses Python source file."""* code = ''  
  
 with open(name) as f:  
 code = f.read()  
  
 lx = Lexer(iter(code+'\n'))  
 lx.reserve(Word(Tag.DEF, 'def'))  
 lx.reserve(Word(Tag.LOGZAP, 'not'))  
 lx.reserve(Word(Tag.RETURN, 'return'))  
 lx.reserve(Word(Tag.IF, 'if'))  
 lx.reserve(Word(Tag.ELSE, 'else'))  
 ps = Parser(lx)  
  
 return ps.module()

generator.py

*"""  
Code generator.  
"""*from tree import FunctionDef  
  
  
class NoEntryError(Exception):  
 def \_\_init\_\_(self, message):  
 self.message = message  
  
  
def has\_main(tree):  
 *"""Determines whether a tree has main function."""* for node in tree.body:  
 if (isinstance(node, FunctionDef)  
 and node.name == 'main'):  
 return True  
  
 return False  
  
  
def generate(tree):  
 *"""Generates MASM32 code from AST."""* if not has\_main(tree):  
 raise NoEntryError('No entry point has been found.')  
  
 HEADER = (  
 """.386  
  
.model flat, stdcall  
  
option casemap:none  
  
include C://masm32/include/windows.inc  
include C://masm32/include/kernel32.inc  
include C://masm32/include/user32.inc  
include C://masm32/include/masm32.inc  
include C://masm32/include/masm32rt.inc   
  
includelib C://masm32/lib/kernel32.lib  
includelib C://masm32/lib/masm32.lib  
includelib C://masm32/lib/user32.lib  
  
main PROTO  
  
.data  
  
.code  
  
""")  
  
 generated = tree.visit()  
  
 ENTRY = ("""  
start:  
 invoke main  
 fn MessageBox,0,str$(eax), "Lab5" ,MB\_OK  
 invoke ExitProcess, 0  
 """)  
 END = ("""  
  
END start""")  
  
 return f'{HEADER}{ENTRY}{generated}{END}'

tree.py

from collections import ChainMap  
  
var\_map = ChainMap()  
shift = -4  
  
"""  
AST Classes.  
"""  
  
  
class Module():  
 def \_\_init\_\_(self, body):  
 self.body = body  
  
 def \_\_eq\_\_(self, other):  
 return (other is not None  
 and self.body == other.body)  
  
 def visit(self):  
 mod = ''  
 for node in self.body:  
 mod += node.visit()  
  
 return mod  
  
  
class FunctionDef():  
 def \_\_init\_\_(self, name, args, body):  
 self.name = name  
 self.args = args  
 self.body = body  
  
 def \_\_eq\_\_(self, other):  
 return (other is not None  
 and self.name == other.name  
 and self.args == other.args  
 and self.body == other.body)  
  
 def visit(self):  
 global var\_map, shift  
 func = f'\n{self.name} PROC\n'  
 func += 'push ebp\nmov ebp, esp\n'  
 scope = {}  
 var\_map = var\_map.new\_child(scope)  
 for num, param in enumerate(self.args):  
 var\_map[param] = num\*4 + 8  
 for node in self.body:  
 func += node.visit()  
 del var\_map.maps[0]  
 func += 'mov esp, ebp\npop ebp\n'  
 func += f'\nret\n{self.name} ENDP'  
 shift = -4  
 return func  
  
  
class CallFunc:  
 def \_\_init\_\_(self, id, args):  
 self.name = id  
 self.args = args  
  
 def visit(self):  
 return '\n'.join([f'{val.visit()}\npush eax' for val in reversed(self.args)])+f'\ncall {self.name}\nadd esp, ' \  
 f'{4\*len(self.args)}\n'  
  
  
class Return():  
 def \_\_init\_\_(self, value):  
 self.value = value  
  
 def \_\_eq\_\_(self, other):  
 return (other is not None  
 and self.value == other.value)  
  
 def visit(self):  
 ret = self.value.visit()  
  
 ret += 'mov esp, ebp\npop ebp\nret\n'  
 return ret  
  
  
class Bin\_Op():  
 def \_\_init\_\_(self, left, right, operation):  
 self.left = left  
 self.right = right  
 self.operation = operation  
  
 def \_\_eq\_\_(self, other):  
 return (other is not None  
 and self.left == other.left  
 and self.right == other.right  
 and self.operation == other.operation)  
  
 def visit(self):  
 if self.operation == '-':  
 return self.left.visit() + 'push eax\n' + self.right.visit() + 'mov ebx, eax\npop eax\nsub eax, ebx\n'  
 if self.operation == '+':  
 return self.left.visit() + 'push eax\n' + self.right.visit() + 'mov ebx, eax\npop eax\nadd eax, ebx\n'  
 if self.operation == '%':  
 return self.left.visit() + 'push eax\n' + self.right.visit() + 'mov ebx, eax\npop eax\nxor edx, edx\n' \  
 'div ebx\nmov eax, edx\n'  
 if self.operation == '\*':  
 return self.left.visit() + 'push eax\n' + self.right.visit() + 'mov ebx, eax\npop eax\nxor edx, edx\n' \  
 'mul ebx\n'  
  
  
class Unary\_Op:  
 def \_\_init\_\_(self, target, operation):  
 self.target = target  
 self.operation = operation  
  
 def visit(self):  
 if self.operation == 'not':  
 return self.target.visit() + '.if eax == 0\n mov eax, 1\n.else\nmov eax, 0\n.endif\n'  
  
  
class Ternary:  
 def \_\_init\_\_(self, true\_con, false\_con, condition):  
 self.true\_con = true\_con  
 self.false\_con = false\_con  
 self.condition = condition  
  
 def \_\_eq\_\_(self, other):  
 return (other is not None  
 and self.true\_con == other.true\_con  
 and self.false\_con == other.false\_con  
 and self.condition == other.condition)  
  
 def visit(self):  
 return self.condition.visit()+'.if eax == 0\n'+self.false\_con.visit()+'.else\n'+self.true\_con.visit()+'.endif\n'  
  
  
class Assign:  
 def \_\_init\_\_(self, id: str, expresion):  
 self.id = id  
 self.expression = expresion  
  
 def \_\_eq\_\_(self, other):  
 return (other is not None  
 and self.id == other.id  
 and self.expression == other.expression)  
  
 def visit(self):  
 global shift  
 if var\_map.get(self.id):  
 shift\_off = var\_map.get(self.id)  
 return self.expression.visit() + 'mov DWORD ptr[ebp+' + str(shift\_off) + '], eax\n'  
 var\_map[self.id] = shift  
 shift -= 4  
 return self.expression.visit() + 'push eax\n'  
  
  
class Id:  
 def \_\_init\_\_(self, id):  
 self.id = id  
  
 def \_\_eq\_\_(self, other):  
 return (other is not None  
 and self.id == other.id)  
  
 def visit(self):  
 if var\_map.get(self.id):  
 shift\_off = var\_map.get(self.id)  
 return 'mov eax, DWORD ptr[ebp+' + str(shift\_off) + ']\n'  
  
  
class Constant:  
 def \_\_init\_\_(self, value):  
 self.value = value  
  
 def \_\_eq\_\_(self, other):  
 return (other is not None  
 and self.value == other.value)  
  
 def visit(self):  
 if type(self.value) == float:  
 return 'mov eax, ' + str(int(self.value)) + '\n'  
 return 'mov eax, ' + str(self.value) + '\n'

**Контрольні питання**

1. Чим відрізняються функції у С та Python.

У С при оголошенні функції необхідно вкзати тип, ім’я функції, аргументи(оголошуючи їх типи та фігурну дужку, у Пайтоні оголошення функції відбуваєтьзя за топомогою def, ім’я функції та аргументи(без типу або з типом за бажанням)

1. В чому різниця між оголошенням та визначенням функції?

Оголошення функції дає лиш основні відомості про неї(повертаємий тип, її назву і т.д), а визначення функції має вкзати ще й тіло функції.

1. У якому порядку виконуються оголошення та визначення функції?

У пайтон не можна окремо оголосити функцію, тож визначення відбувається так: def, ім’я функції, аргументи(без типу або з типом за бажанням), дві крапки потім тіло функції з необхідними відступами

1. Чим відрізняються аргументи функції від її параметрів?

Параметри функції вказують при оголошенні функції.

Аргументи функції передають при виклику.

1. Коли на етапі парсингу повинна виникати помилка?

Після розпізнавання невідомого або непідтримуючого типу

1. В чому сутність конвенції виклику функцій, для чого вони потрібні, чим

відрізняються?

Конвенція дає поняття про те як передавати або повертати аргумент та про те чи можна переписати регістр

1. Чим відрізняються caller-saved та callee-saved регістри?

callee-saved мають бути скинуті після закінчення роботи з цими регістрами, а caller-saved можна переписувати

1. Який порядок занесення аргументів у стек та чим це пояснюється?

Під час виклику функції до стеку заноситься return address, тому необхідно записувати аргументи у зворотному порядку. Після цього створюємо кадр стеку, тоді в нас створюється зайнята комірка. Звернення до доданих аргументів до стеку буде відбуватися наступним чином: -8\* номер адреси аргументу, через зворотнє заповнення стеку, найближий аргумент має адресу -8, другий -16 і т.д.

**Висновок**

У данній лабораторній роботі було колосально переписано лексер та додано розпізнавання та виконання функцій крім main(). Для цього було переписано лексер для правильних початків та кінців функцій, крім того було додадано розпізнавання параметрів функцій та переписано у парсері функцію func(). Значні проблеми виникли лише при переписуванні лексеру, що знову зайняло багато часу через те що пайтон використовує пробіли для їх визначення. Та наразі мій компілятор тепер ще вміє розпізнавати та виконувати функції навіть рекурсивно.