НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

**РОЗРАХУНКОВО-ГРАФІЧНА РОБОТА**

з дисципліни «СИСТЕМНЕ ПРОГРАМУВАННЯ-2».

на тему: «Розробка компілятора підмножини команд мови Python, що містять арифметичні дії і дужкові форми»

Студента 3 курсу групи ІО-81

спеціальності

123 «Комп’ютерна  інженерія»

Рибніков Є. О.

Керівник      доцент Павлов В.Г.

Національна оцінка \_\_\_\_\_\_\_\_\_\_\_\_\_

Кількість балів \_\_\_\_\_\_\_\_\_\_

Оцінка:  ECTS   \_\_\_\_\_\_\_\_\_\_\_

Київ- 2020 рік

Національний технічний університет України

“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет інформатики та обчислювальної техніки

Кафедра   обчислювальної техніки

Освітньо-кваліфікаційний рівень   бакалавр

Спеціальність    123 «Комп’ютерна інженерія»

*З  А  В  Д  А  Н  Н  Я*

НА РОЗРАХУНКОВО-ГРАФІЧНУ РОБОТУ СТУДЕНТА

Рибнікова Євгенія Олександровича

1. Тема  роботи    «Розробка компілятора підмножини команд мови Python, що містять арифметичні дії і дужкові форми»

керівник роботи   Павлов В.Г.    к.т.н.**,** доцент

2. Строк подання студентом  роботи   28 грудня 2020 р.

3. Вхідні дані до роботи

- вхідна мова:       Python;

- цільова мова: Ассемблер x86;

- типи конструкцій: арифметичні дії, цикли, функції.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)

    - розробити лексичний аналізатор мови Python;

    - розробити синтаксичний аналізатор мови Python;

    - розробити генератор коду мови Ассемблер x86.

5. Перелік графічного матеріалу;

- алгоритм  лексичного аналізу;

- алгоритм  синтаксичного аналізу;

     - дерева розбору

6. Дата видачі завдання      05 листопада 2020 р

Зміст

[1.Вступ 4](#__RefHeading___Toc3439_2700349471)

[2. Етапи компіляції 4](#__RefHeading___Toc3441_2700349471)

[2.1 Лексичний аналіз 4](#__RefHeading___Toc3443_2700349471)

[2.2 Синтаксичний аналіз 4](#__RefHeading___Toc3445_2700349471)

[2.3 Генерація асемблерного коду 5](#__RefHeading___Toc3447_2700349471)

[3. Перевірка коректної роботи компілятора 5](#__RefHeading___Toc3449_2700349471)

[3.1 Лістинг розробленої програми 5](#__RefHeading___Toc3451_2700349471)

[3.2 Згенерований асемблерний код: 6](#__RefHeading___Toc11925_2700349471)

[3.3 Список токенів 7](#__RefHeading___Toc3455_2700349471)

[3.4 Контрольні приклади 7](#__RefHeading___Toc3457_2700349471)

[4. Висновок 20](#__RefHeading___Toc3459_2700349471)

[Додаток 1 20](#__RefHeading___Toc3461_2700349471)

# 1.Вступ

Під час виконання роботи був написаний компілятор на мові Python для перетворення коду з мови Python у мову асемблера MASM32. Його робота полягає в наступному: проведення лексичного та синтаксичного аналізу вхідної програми та генерування файлу з ассемблерним кодом. Якщо виникне помилка, файл не буде згенеровано, натомість компілятор видасть повідомлення про помилку з її розташуванням(рядок та номер символу).

# 2. Етапи компіляції

Компілятор отримує файл в якому програма написана на мові Python. Далі проводиться Лексичний аналіз, створення списку токенів, синтаксичний аналіз і генерація асемблерного коду.

## 2.1 Лексичний аналіз

Лексичний аналіз відбувається у файлі Lexer.py. В ньому розділені за групами всі можливі символи та операції вхідного коду, за якими виконується перевірка усієї вхідної програми. Якщо трапляється символ або слово вхідної програми, який належить певній групі, з цього символу генерується відповідний токен з відновідним тегом та значенням і заноситься до загального спику токенів.

## 2.2 Синтаксичний аналіз

Синтаксичний розбір відбувається у файлі Parser.py. Парсер отримує список токенів, які згенерував лексер, та перевіряє кожен токен. Якщо при перевірці виникне непередбачувана ситуація, парсер видасть повідомлення про помилку, де вкаже на рядок та індекс у вхідному коду, де було допущено помилку.

## 2.3 Генерація асемблерного коду

Генерація асемблерного коду частково відбувається у файлі Parcer.py, частково у файлі main.py. У main.py генерується частина коду, яка відповідає за включення необхідних бібліотек та необхідних параметрів для запуску програми. У файлі Parcer.py генеруються всі інші процедури, які вже безпосередньо відповідають за виконання основної процедури, а також за вивід результату у вікно відповідно до списку токенів та граматиці.

# 3. Перевірка коректної роботи компілятора

Для перевірки коректної роботи компілятора, написаного на мові Python, була створено програму мовою Python, що реалізує алгоритм за варіантом.



**1****. Числа Фібоначчі. Рекурсивно. Ітераційно.**

Реалізуйте функцію знаходження n-го числа послідовності Фібоначчі за допомогою рекурсивної функції та ітераційно. Вхідними даними є номер необхідного числа послідовності; вихідним є ціле невід’ємне число, що належить до послідовності.

## 3.1 Лістинг розробленої програми

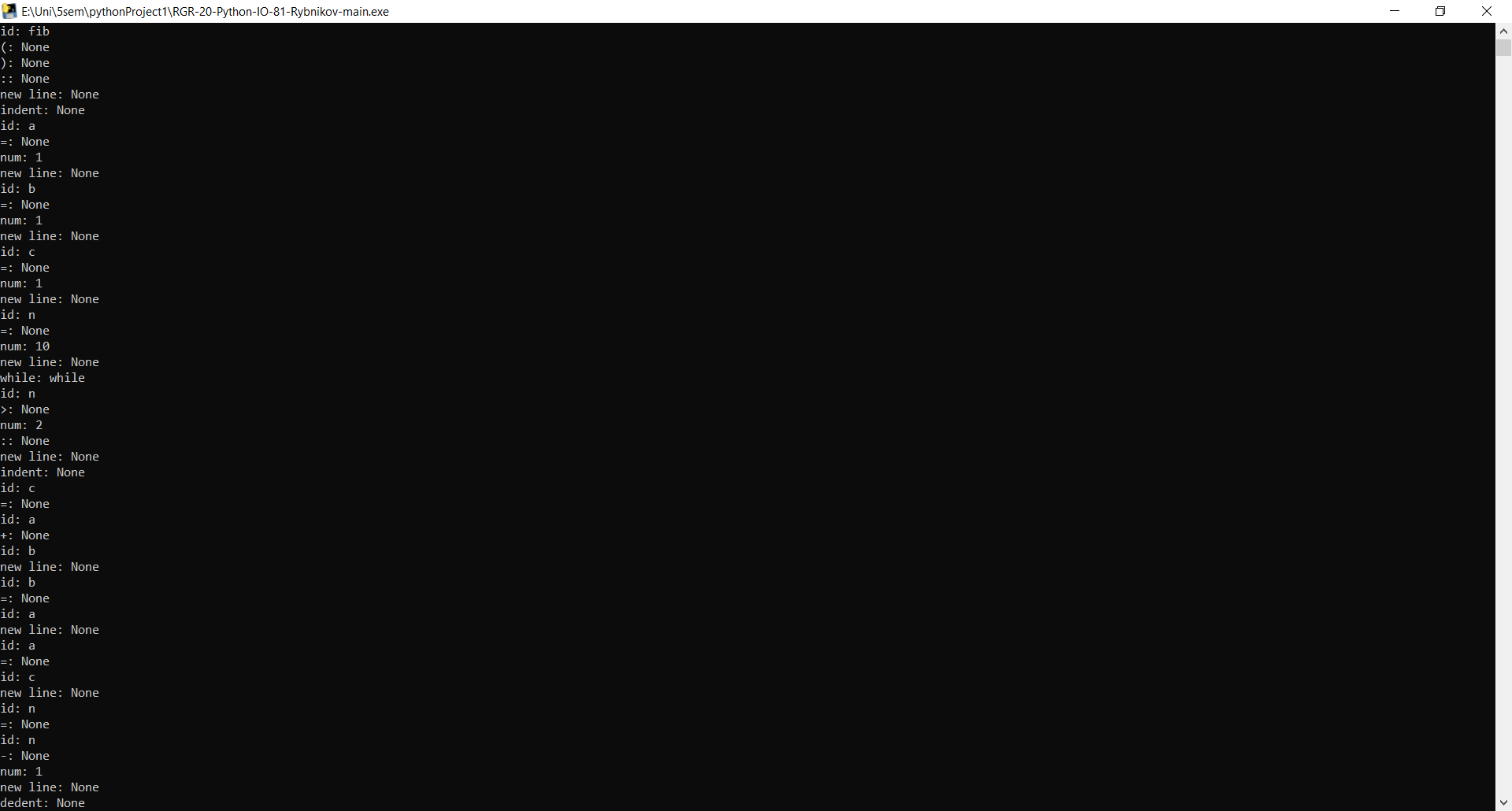
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | def fib():  a = 1  b = 1  c = 1  n = 10  while n > 2:  c = a + b  b = a  a = c  n = n - 1  return c  fib() |

## 

## 3.2 Згенерований асемблерний код:

|  |  |
| --- | --- |
| 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 | .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  fib proc  mov eax, 1  mov ebx, 1  mov ecx, 1  mov edx, 10  cycle:  sub edx, 1  add ecx, ebx  mov ebx, eax  mov eax, ecx  cmp edx, 2  jne cycle  fn MessageBox,0,str$(ecx),"Rybnikov",MB\_OK  ret  fib endp  start:  invoke fib  invoke ExitProcess, 0  end start |

## 3.3 Список токенів

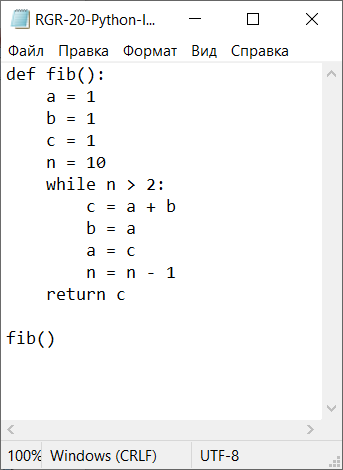


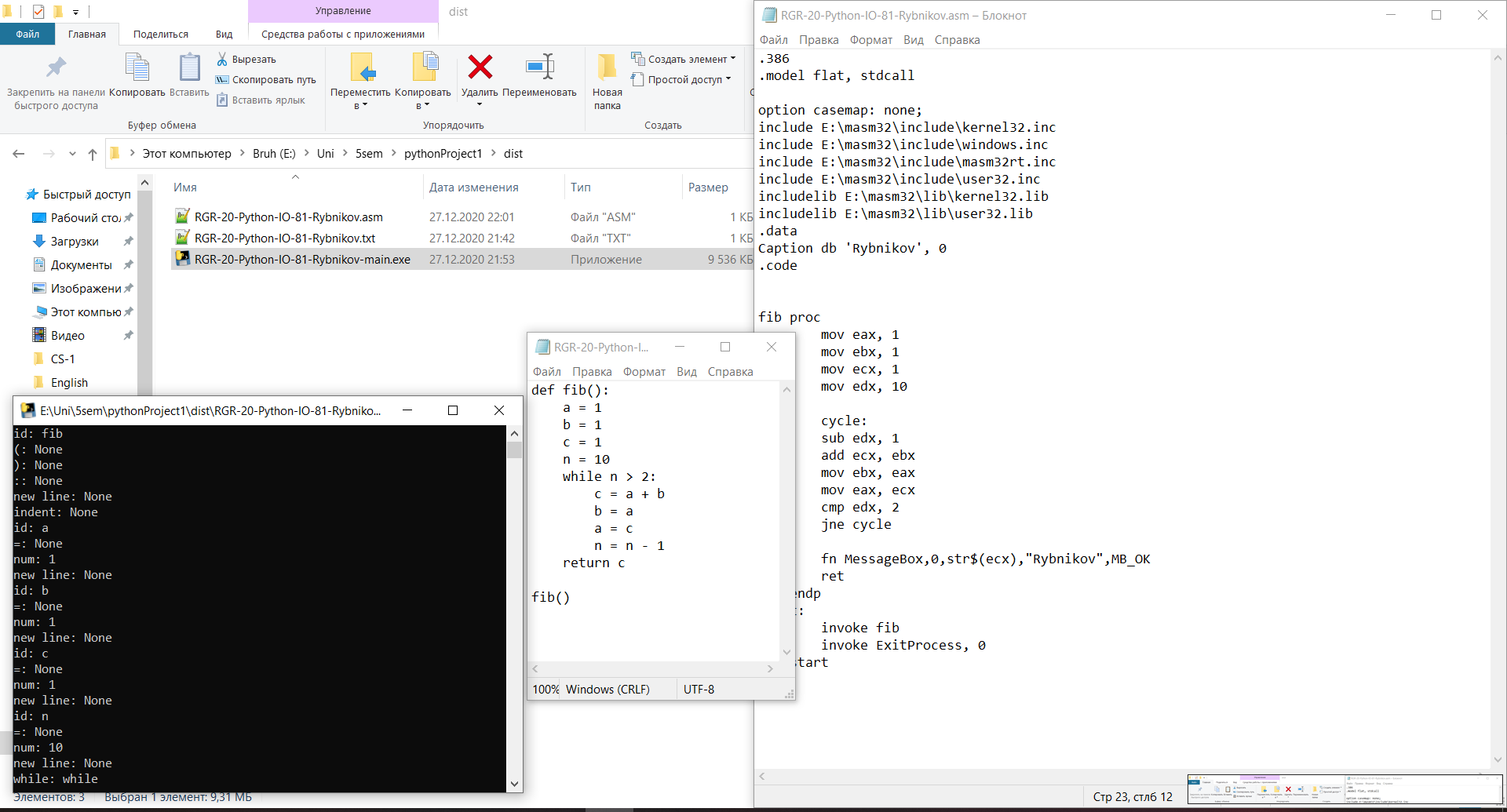
## 3.4 Контрольні приклади

Розроблена програма на мові Python має функцію fib()(назву можна змінити), яка використовується для обчислення числа Фібоначчі. На вході ми маємо номер числа Фібоначчі яке необхідно знайти, записане у змінну n, яка буде використовуватися у якості лічильник, а також змінні a, b, c назви яких можна змінити, у яких буде відбуватися операції над числами. Далі за допомогою циклу while поки значення лічильника не досягне 2, відбуваються операції над числами поки лічильник не досягне позначки 2. Потім виводиться значення шуканого числа Фібоначчі.

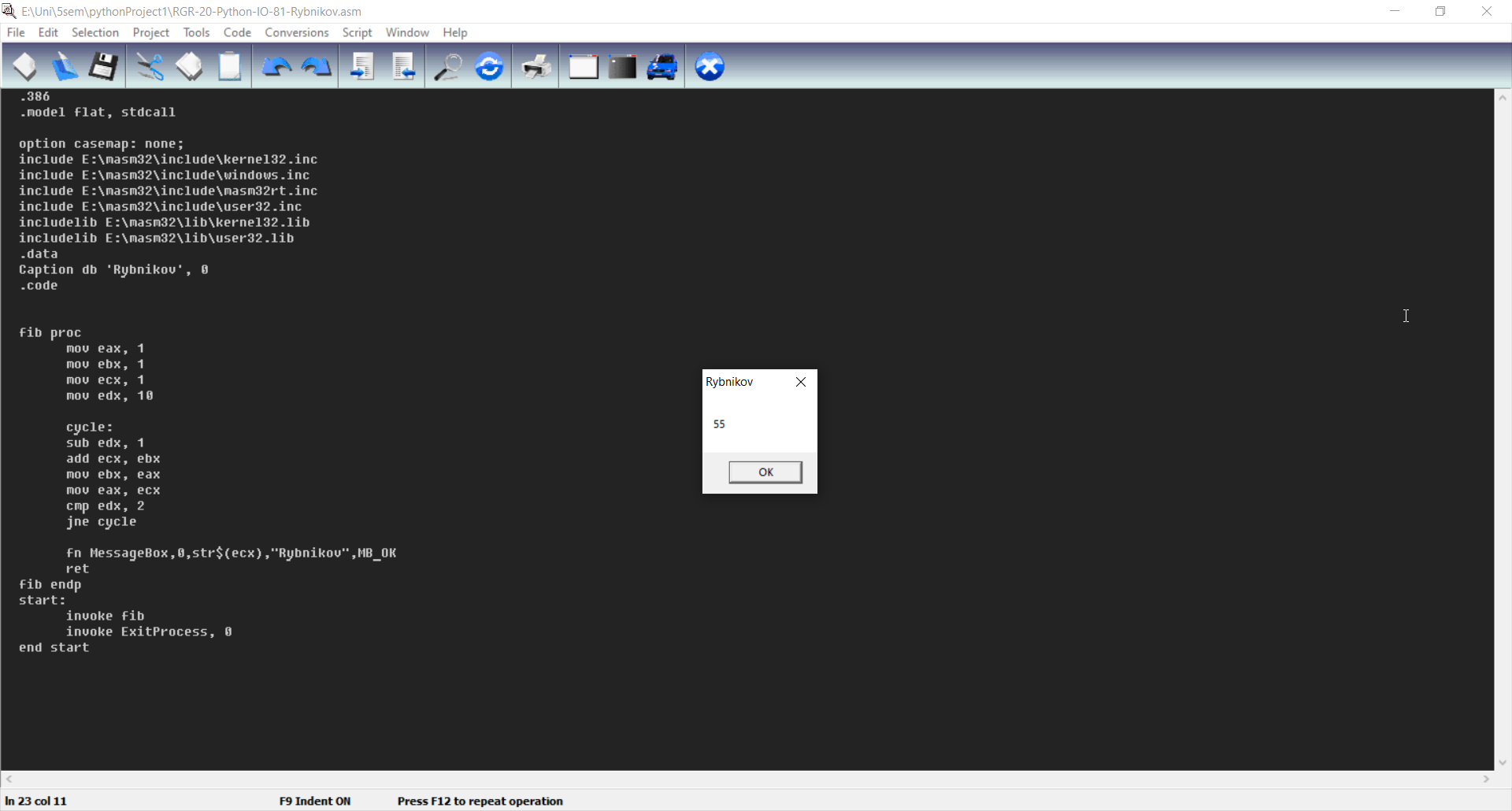
**Контрольний приклад 1**

Вхідний код:



Генерація коду:

Перевірка роботи вихідного коду:



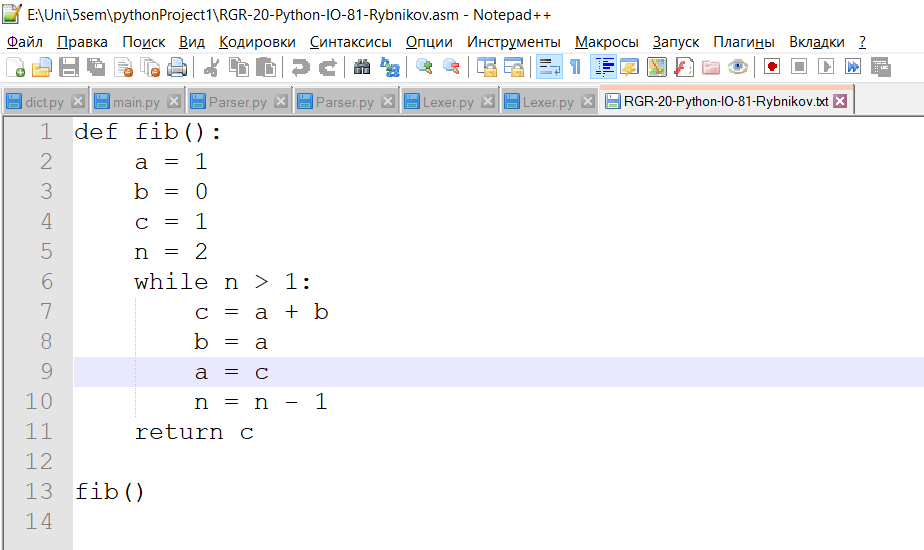
Перевірка:

n = 10

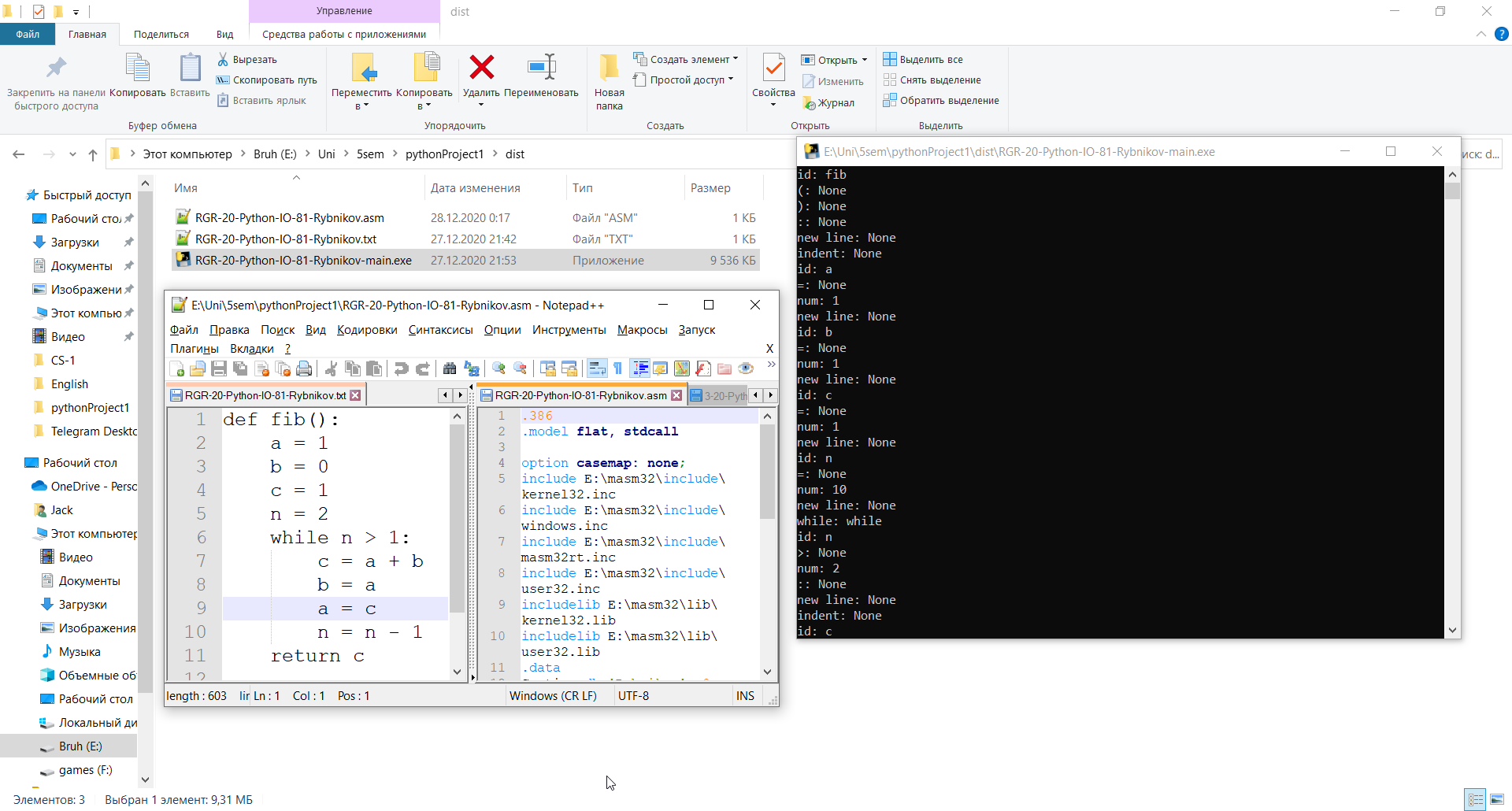
1. 1
2. 1 + 0 = 1
3. 1 + 1 = 2
4. 2 + 1 = 3
5. 3 + 2 = 5
6. 5 + 3 = 8
7. 8 + 5 = 13
8. 13 + 8 = 21
9. 21 + 13 = 34
10. 34 + 21 = 55

Контрольний приклад 2

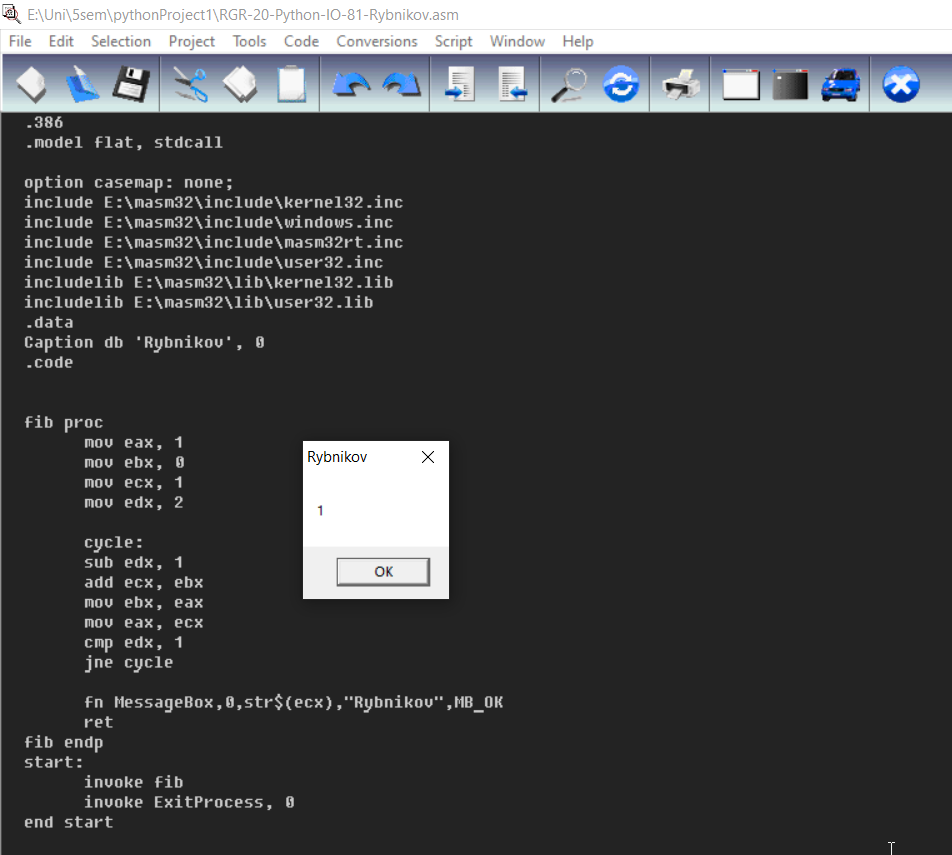
Вхідний код:



Генерація коду:



Перевірка роботи вихідного коду:



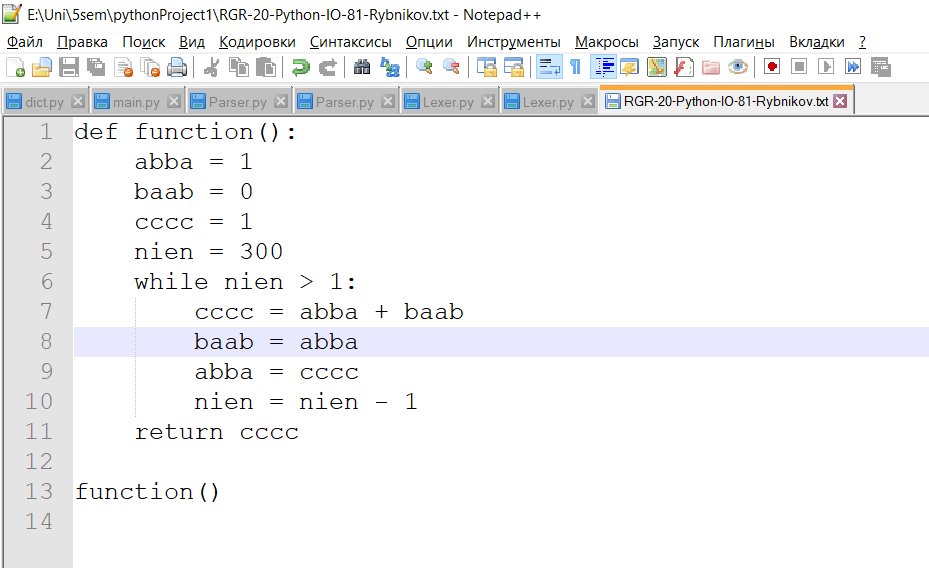
Перевірка:

n = 2

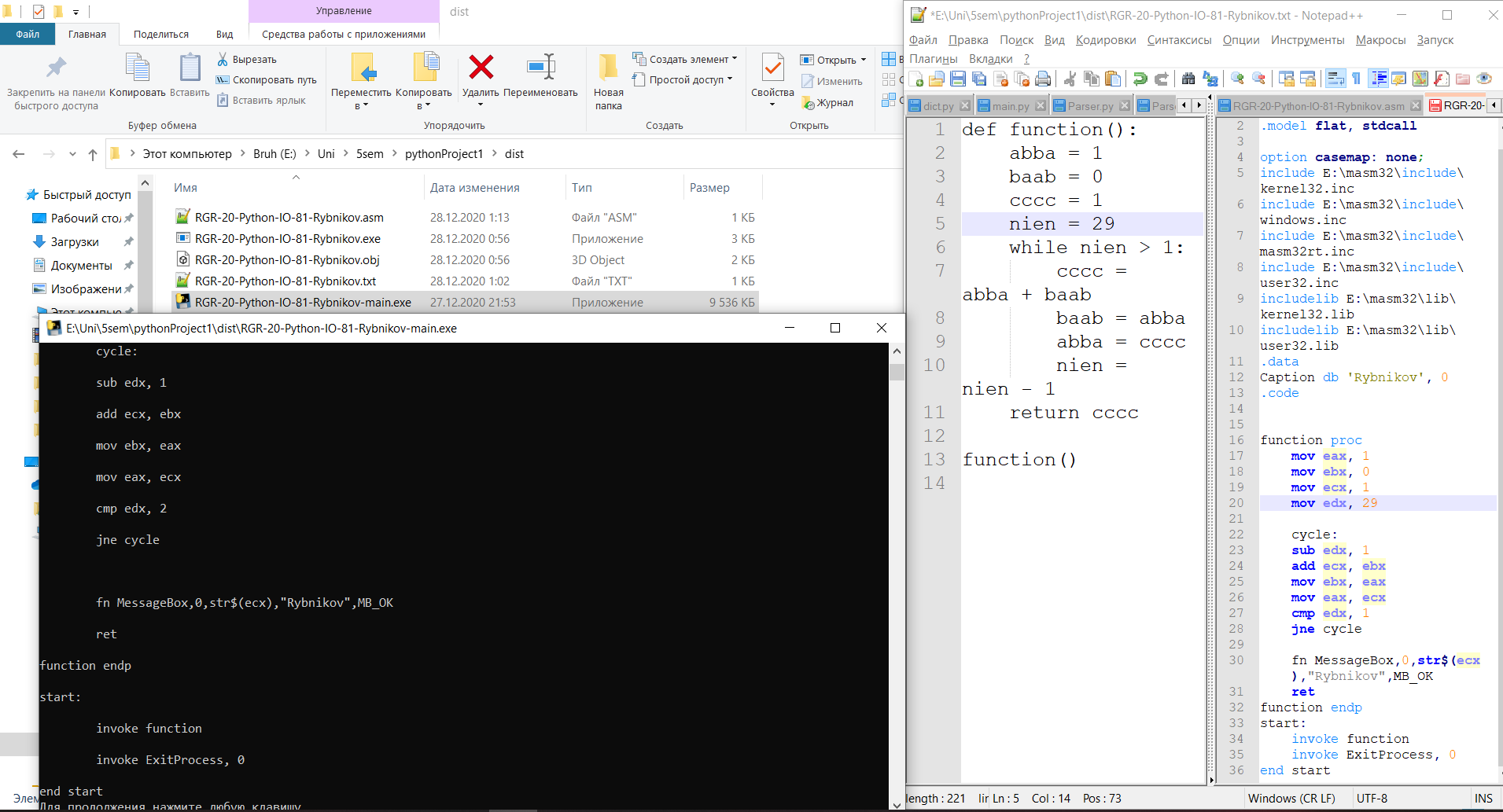
1. 1
2. 1 + 0 = 1

Контрольний приклад 3

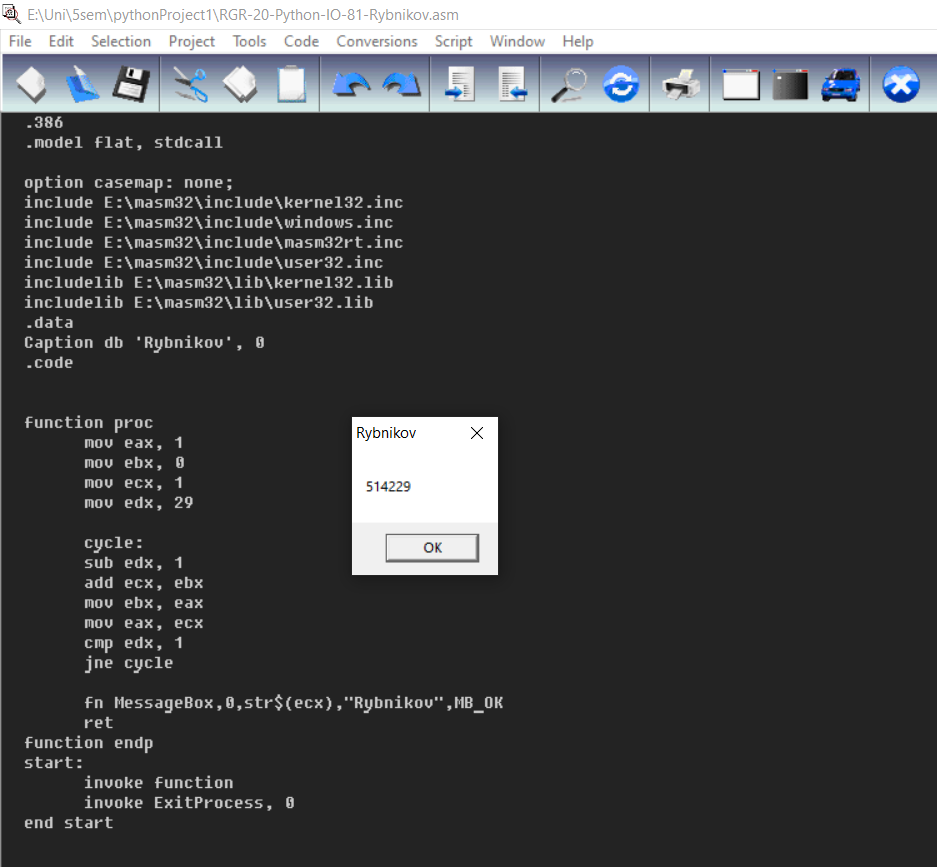
Вхідний код:



Генерація коду:



Перевірка роботи вихідного коду:

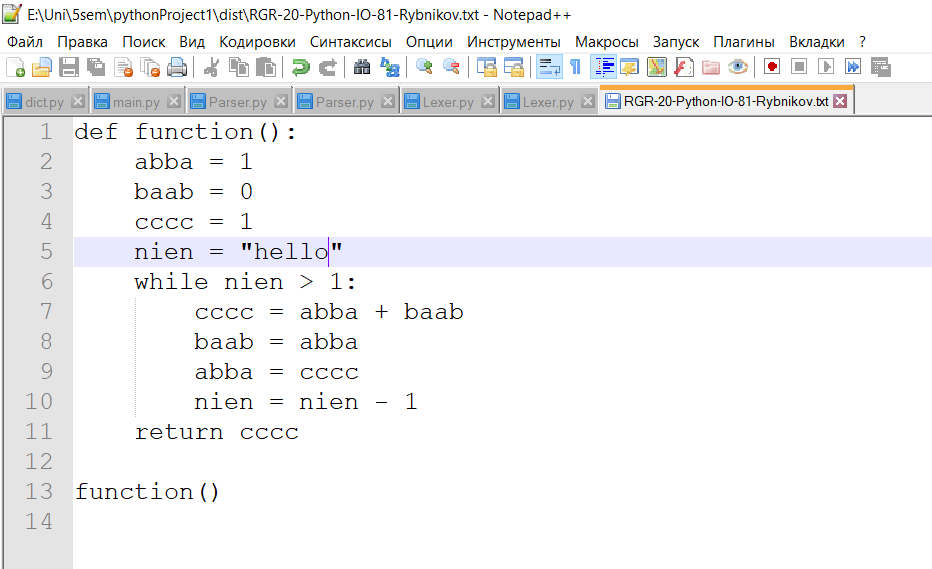
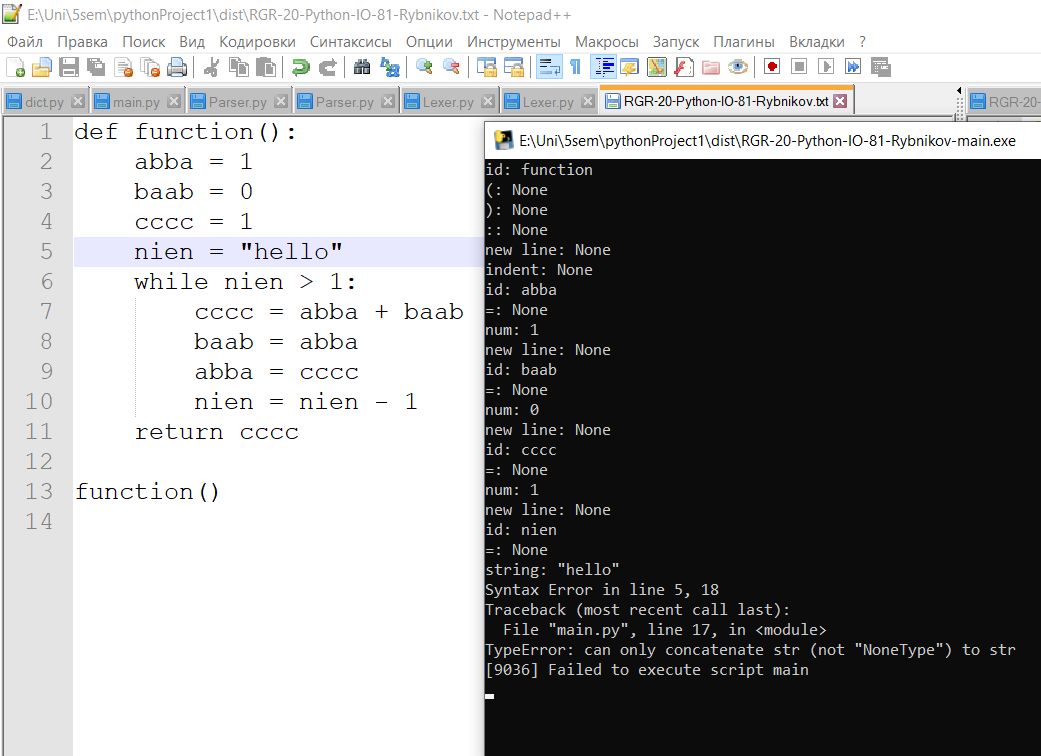


Перевірка:

n = 29

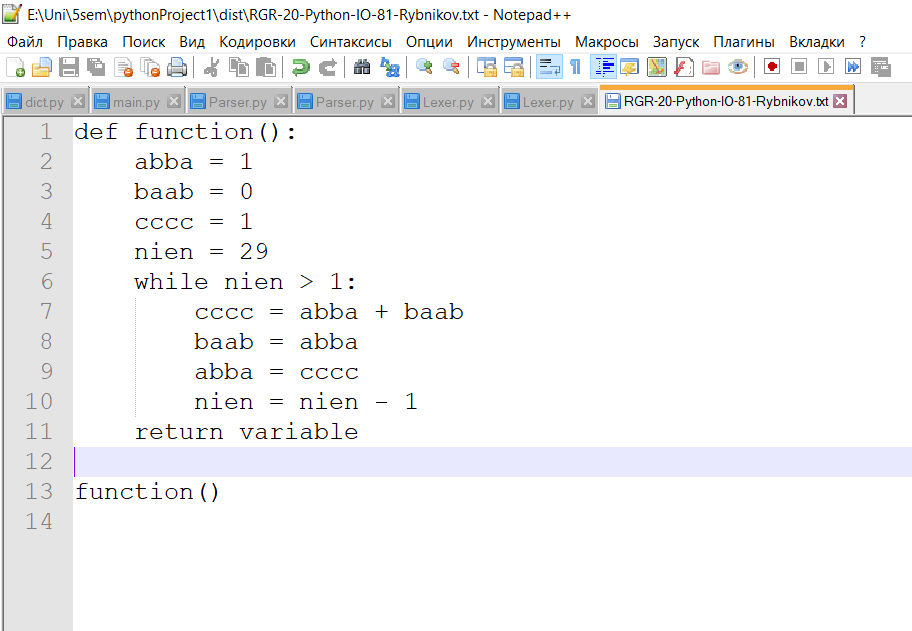
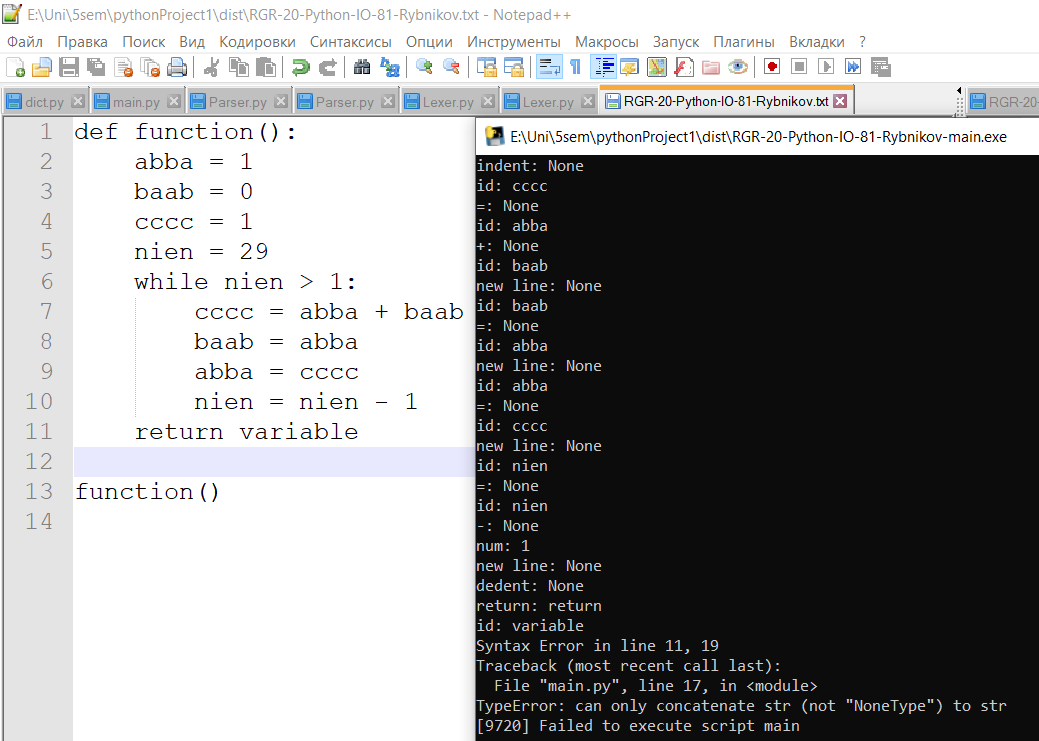
1. 1
2. 1 + 0 = 1
3. 1 + 1 = 2
4. 2 + 1 = 3
5. 3 + 2 = 5
6. 5 + 3 = 8
7. 8 + 5 = 13
8. 13 + 8 = 21
9. 21 + 13 = 34
10. 34 + 21 = 55
11. 55 + 34 = 89
12. 89 + 55 = 144
13. 144 + 89 = 233
14. 233 + 144 = 377
15. 377 + 233 = 610
16. 610 + 377 = 987
17. 987 + 610 = 1597
18. 1597 + 987 = 2584
19. 2584 + 1597 = 4181
20. 4181 + 2584 = 6765
21. 6765 + 4181 = 10946
22. 10946 + 6765 = 17711
23. 17711 + 10946 = 28657
24. 28657 + 17711 = 46368
25. 46368 + 28657 = 75025
26. 75025 + 46368 = 121393
27. 121393 + 75025 = 196418
28. 196418 + 121393 = 317811
29. 317811 + 196418 = 514229

Контрольний приклад 4

Вхідний код:Вихідний код:

Контрольний приклад 5

Вхідний код:

ВВихідний код:

# 4. Висновок

В ході виконання розрахункової роботи було розроблено програму для реалізації рекурсивного алгоритму розрахунку числа Фібаначчі, а також було розроблено компілятор, який перетворює вхідний код мовою Python у код мовою MASM32. Опис роботи компілятору було наведено в пункті «Етапи компіляції». Для написання програми було використано цикл while, оператор декременту, а також умовний оператор “>”. Роботу компілятора було протестовано на 5 контрольних прикладах: 3 правильних та 2 неправильних. Результати виявилися корентними: для правильних прикладів виводиться правильний результат обрахування виразу, а для неправильного виводиться повідомлення про помилку. Всі правильні приклади було обраховано вручну – їх результати ідентичні.

# Додаток 1

Лістинг коду

**Terminals.py**

|  |  |
| --- | --- |
| 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 | 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 = ';' |

**Token.py**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 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**

|  |  |
| --- | --- |
| 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  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145 | 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   d = Token(self.peek)  if self.index < len(content) - 1:  self.index += 1  self.error += 1  self.peek = ' '  return d |

**Parser.py**

|  |  |
| --- | --- |
| 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  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322  323  324  325  326  327  328  329  330  331  332  333 | import sys import os from Lexer import Lexer from Token import Token from dict import \*  def\_list = [] var\_list = [] a = open("RGR-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")) flag\_unarop = False   def match(a):  global lookahead  global input  if lookahead.tag == a:  lookahead = lexer.scan(input)  print(lookahead)  else:  raise SyntaxError   def term(t):  global lookahead  global flag\_unarop  if lookahead.tag == NUM:  if flag\_unarop:  t = "\tmov ebx, " + str(lookahead.value) + "\n" + t  t += "\tmov eax, ebx\n"  match(NUM)  flag\_unarop = 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:  flag\_unarop = 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)  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)  match(ID)  if lookahead.tag == LESS:  match(LESS)  if lookahead.tag == MORE:  match(MORE)  t += "\tmov ecx, "  t = expression(t)  t += "\tcycle:\n"  match(COLON)  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("Unknown variable in " + str(lexer.error) + " 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():  global lookahead  lookahead = lexer.scan(input)  try:  match(DEF)  invoker = ''  invoker += f"\n{lookahead.value} proc\n"  def\_list.append(lookahead.value)  match(ID)  match(LPAR)  match(RPAR)  match(COLON)  while lookahead.tag == NL:  match(NL)  match(INDENT)  var\_list.append(lookahead.value)  match(ID)  match(EQUAL)  invoker += f"\tmov eax, {lookahead.value}\n"  match(NUM)  while lookahead.tag == NL:  match(NL)  var\_list.append(lookahead.value)  match(ID)  match(EQUAL)  invoker += f"\tmov ebx, {lookahead.value}\n"  match(NUM)  while lookahead.tag == NL:  match(NL)  var\_list.append(lookahead.value)  match(ID)  match(EQUAL)  invoker += f"\tmov ecx, {lookahead.value}\n"  match(NUM)  while lookahead.tag == NL:  match(NL)  var\_list.append(lookahead.value)  match(ID)  match(EQUAL)  invoker += f"\tmov edx, {lookahead.value}\n"  match(NUM)  while lookahead.tag == NL:  match(NL)  match(WHILE)  if lookahead.value == var\_list[3]:  match(ID)  else:  raise SyntaxError  match(MORE)  match(NUM)  match(COLON)  #match(INDENT)  while lookahead.tag == NL:  match(NL)  match(INDENT)  if lookahead.value == var\_list[2]:  match(ID)  else:  raise SyntaxError  match(EQUAL)  if lookahead.value == var\_list[0]:  match(ID)  else:  raise SyntaxError  match(PLUS)  if lookahead.value == var\_list[1]:  match(ID)  else:  raise SyntaxError  while lookahead.tag == NL:  match(NL)  if lookahead.value == var\_list[1]:  match(ID)  else:  raise SyntaxError  match(EQUAL)  if lookahead.value == var\_list[0]:  match(ID)  else:  raise SyntaxError  while lookahead.tag == NL:  match(NL)  if lookahead.value == var\_list[0]:  match(ID)  else:  raise SyntaxError  match(EQUAL)  if lookahead.value == var\_list[2]:  match(ID)  else:  raise SyntaxError  while lookahead.tag == NL:  match(NL)  if lookahead.value == var\_list[3]:  match(ID)  else:  raise SyntaxError  match(EQUAL)  if lookahead.value == var\_list[3]:  match(ID)  else:  raise SyntaxError  match(MINUS)  match(NUM)  while lookahead.tag == NL:  match(NL)  match(DEDENT)  invoker += f"\n\tcycle:" \  f"\n\tsub edx, 1" \  f"\n\tadd ecx, ebx" \  f"\n\tmov ebx, eax" \  f"\n\tmov eax, ecx" \  f"\n\tcmp edx, 1" \  f"\n\tjne cycle\n" \  f"\n\tfn MessageBox,0,str$(ecx),\"Rybnikov\",MB\_OK" \  f"\n\tret" \  f"\n{def\_list[0]} endp"  match(RETURN)  if lookahead.value == var\_list[2]:  match(ID)  else:  raise SyntaxError  while lookahead.tag == NL:  match(NL)  while lookahead.tag == DEDENT:  match(DEDENT)  while lookahead.tag == NL:  match(NL)  invoker += f"\nstart:" \  f"\n\tinvoke {def\_list[0]}" \  f"\n\tinvoke ExitProcess, 0" \  f"\nend start"  if lookahead.value == def\_list[0]:  match(ID)  else:  raise SyntaxError  match(LPAR)  match(RPAR)  return invoker  except SyntaxError:  print(f"Syntax Error in line {lexer.line}, {lexer.error}") |

**main.py**

|  |  |
| --- | --- |
| 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 | import os from Parser\_RGR 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() b = open("RGR-20-Python-IO-81-Rybnikov.asm", "w") b.write(a) b.close() c = open("RGR-20-Python-IO-81-Rybnikov.asm", "r") line = c.readline() while line:  print(line)  line = c.readline() c.close() os.system("PAUSE") |