**Трансляцiя у ПОЛIЗ арифметичних виразiв та iнтерпретацiя постфiксного коду**

Варіант № 7

Арифметика: цілі та дійсні числа, основні чотири арифметичні операції (додавання, віднімання, ділення та множення), піднесення до степеня (правоасоціативна операція), дужки

Особливості: унарний мінус

Інструкція повторення: for = by while do Інструкція розгалуження: if < логічний вираз > then fi

**Повна граматика мови FuriousPeach**

Program = program ProgName DeclSection DoSection

ProgName = Ident

Ident = Letter {Letter | Digit }

DeclSection = DeclarList

DeclarList = Declaration { Declaration }

Declaration = Type ’ ’ IdenttList

IdenttList = Ident {’,’ Ident}

Type = int | real | bool

DoSection = StatementList ’bye’

StatementList = Statement {’;’ Statement }

Statement = Assign | Input | Print | ForStatement

Assign = Ident ’=’ (Expression | BoolExpr)

Expression = ArithmExpression | BoolExpr

BoolExpr = ArithmExpression Rel ArithmExpression

ArithmExpression = [Sign] Term | ArithmExpression ’+’ Term | ArithmExpression ’-’ Term

Term = Factor | Term ’\*’ Factor | Term ’/’ Factor | Term ’^’ Factor

Factor = Ident | Const | ’(’ ArithmExpression ’)’

Input = input ’(’ IdenttList ’)’

Print = print ’(’ IdenttList ’)’

ForStatement = for ’(’ IndExpr ’)’ ’{’DoBlock ’}’

IndExpr = Ident ’=’ ArithmExpression1 ’;’

BoolExpr ’;’ ArithmExpression2

ArithmExpression1 = ArithmExpression

ArithmExpression2 = Ident Add Step

DoBlock = Statement | StatementList

IfStatement = if BoolExpr ’then ’ ’goto’ Ident1 | ’else’ ’goto’ Ident2

Ident1 = Letter {Letter | Digit }

Ident2 = Letter {Letter | Digit }

Const = IntNumb | RealNumb | BoolConst

IntNumb = [Sign] UnsignedInt

RealNumb = [Sign] UnsignedReal

Sign = ’+’ | ’-’

UnsignedInt = Digit {Digit}

UnsignedReal = ’.’ UnsignedInt | UnsignedInt ’.’ | UnsignedInt ’.’ UnsignedInt

Letter = ’a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ | ’g’ | ’h’ | ’i’ | ’j’ | ’k’ | ’l’ | ’m’ | ’n’ | ’o’ | ’p’ | ’q’ | ’r’ | ’s’ | ’t’ | ’u’ | ’v’ | ’w’ | ’x’ | ’y’ | ’z’

Digit = ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’ | ’8’ | ’9’

BoolConst = true | false

Rel = ’==’ | ’<= ’ | ’<’ | ’>’ | ’>=’ | ’<>’

**Специфiкацiя арифметики**

**Типи**

Мова FuriousPeach пiдтримує значення трьох арифметичних типiв: int, real та bool.

1. Цiлий тип int може бути представлений константою Int, або змiнною.

2. Дiйсний тип real може бути представлений константою Real, або змiнною.

3. Бінарний тип Bool може бути представлений константою Bool, або змiнною.

**Константи**

Семантика

1. Кожна константа має тип, визначений її формою.

Приклади

1. 12, 234, 1.54, 34.567, 23.5

**Вирази**

Опис

1. Вираз - це послiдовнiсть операторiв i операндiв, що визначає порядок об числення значення.

2. Розрiзняються арифметичнi та логiчнi вирази.

3. Значення, обчислене за арифметичним виразом, має тип float або int.

4. Значення, обчислене за логiчним виразом, має тип boolean.

5. Всi бiнарнi оператори у виразах цiєї мови лiвоасоцiативнi.

Обмеження

1. Використання неоголошеної змiнної, викликає помилку на етапi трансляцiї або часу виконання.

2. Використання змiнної, що не набула значення, викликає помилку.

Семантика

1. Кожна константа має тип, визначений її формою та значенням.

2. Змiнна набуває значення в iнструкцiї присвоювання Assign або в iнструкцiї введення Inp.

Приклади

1. Factor:   x, 12, (a + 234)

2. Term:            m\*z, 32/(b + 786)

3. Expression:   b, f1 + g, (c - 24), 0.145\*8.7 + 32/(b + 786)

**Арифметичнi оператори**

В мовi FuriousPeachвсi арифметичнi оператори iнфiкснi: адитивнi – ’+’ та ’-’, мультиплiкаттивнi – ’\*’ та ’/’.

Семантика

1. Типи обох операндiв мають бути однаковим, iнакше генерується виклю чення (помилка).

2. Тип результату збiгається з типом операндiв.

3. Дiлення на нуль викликає помилку.

4. Дiлення цiлих дає цiлий результат, отриманий вiдкиданням дробової ча стини вiд дiлення тих же чисел дiйсного типу.

Приклад

1. 1.234\*x1/45.67, 53-34+6, 7/8

**Iнструкцiя присвоювання**

Опис

1. Значення, якi можуть використовуватись у лiвiй та правiй частинах iн струкцiї присвоювання називають l-значенням та r-значенням (або lvalue та rvalue, або left-value та right-value тощо).

Семантика

1. l-значення має тип вказiвника на мiсце зберiгання значення змiнної з iдентифiкатором Ident.

2. r-значення має тип значення, обчисленого за виразом Expression.

3. При присвоюваннi за адресою lvalue зберiгається значення типу rvalue.

Приклад

1. f5 = 3/4+1.23, b = 2 \* a + 3 - 7 div 5

**Лексичний аналiзатор вхiдної мови**

Використовується приклад лексичного аналiзатора до лабораторної роботи №1, реалiзований мовою python. Лексичний аналiзатор (сканер, лексер) та таблиця символiв програми iмпортуються так:

from lex\_my\_lang\_03 import lex, tableToPrint

from lex\_my\_lang\_03 import tableOfSymb, tableOfId, tableOfConst, \ tableOfLabel, sourceCode, FSuccess

Виклик сканера: lex()

**Синтаксичний аналiзатор вхiдної мови**

Використовується приклад синтаксичного аналiзатора до лабораторної роботи №2, реалiзований мовою python. Пiсля його доповнення семантичними процедурами, вiн буде здiйснювати одночасно синтаксичний аналiз та трансляцiю.

**Формат таблиць**

Формат усiх таблиць визначений при розробцi лексичного аналiзатора. Таблицi реалiзованi як словники (dictionary) мови python.

Таблиця мiток тут не використовується, проте формат наводиться.

**Таблиця символiв**

Таблиця символiв tableOfSymb :

{ n\_rec : (num\_line, lexeme, token, idxIdConst) }

де:

n\_rec – номер запису в таблицi символiв програми;

num\_line – номер рядка програми;

lexeme – лексема;

token – токен лексеми;

idxIdConst – iндекс iдентифiкатора або константи у таблицi iдентифiкаторiв та констант вiдповiдно.

**Таблиця iдентифiкаторiв**

Таблиця iдентифiкаторiв tableOfId:

{ Id: (idxId, type, val) }

де:

Id – iдентифiкатор (лексема);

idxId – iндекс iдентифiкатора у таблицi iдентифiкаторiв;

type – тип значення змiнної з iдентифiкатором Id, при (першому) занесеннi до таблицi тип вважається невизначеним type\_undef;

val – значення змiнної з iдентифiкатором Id, при (першому) занесеннi до та блицi значення вважається невизначеним val\_undef.

**Таблиця констант**

Таблиця констант tableOfConst:

{ Const: (idxConst, type, val) }

де:

Const –константа (лексема);

idxConst – iндекс константи у таблицi констант;

type – тип константи;

val – значення константи.

**Таблиця мiток**

Таблиця мiток tableOfLabel:

{ Label : val) }

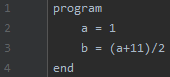
де:

Label –мiтка (лексема);

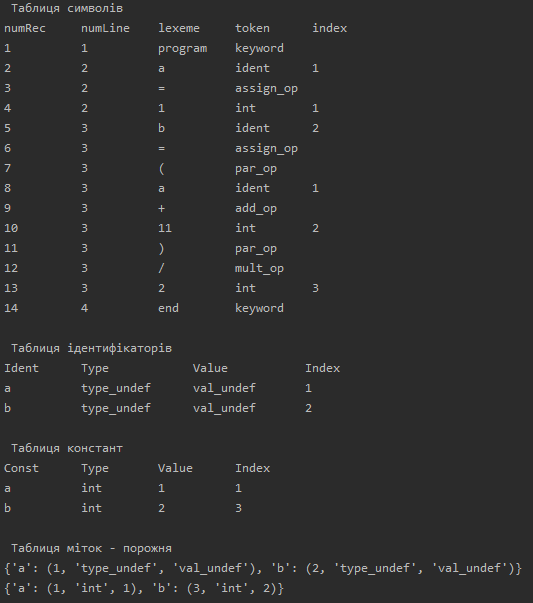
val – значення мiтки.

Приклад

Для програми з п’яти рядкiв:



     будуються таблицi, якi можна переглянути у консолi за допомогою команди tableToPrint(’All’):



Зауважимо, що iндекси iдентифiкаторiв та констант виводяться у консолi в останнiй позицiї. Таблицi та функцiя tableToPrint(’All’) визначенi у сканерi та iмпортуються до iнших модулiв.

**Приклади програм для тестування**

Серед прикладiв програм для тестування довiльного модуля має бути один, називатимемо його базовим, який має мiстити всi синтаксичнi конструкцiї, на явнi у розробленiй мовi. Код у базовому прикладi має бути синтаксично та семантично коректним. Всi iншi приклади мiстять код, який або демонструє бiльш складнi конструкцiї (синтаксично чи семантично), або мiстять помилки рiзних типiв.

**Лексичний аналiзатор мови**

Лексичний аналiзатор (сканер, лексер) та таблиця символiв програми iмпортуються так:

from lex\_my\_lang import lex

from lex\_my\_lang import tableOfSymb

Виклик сканера: lex()

**Формат таблицi символiв**

Таблиця символiв програми tableOfSymb – це словник (dictionary) мови

python. Формат був визначений при розробцi лексичного аналiзатора:

{ n\_rec : (num\_line, lexeme, token, idxIdConst) }

де:

n\_rec – номер запису в таблицi символiв програми;

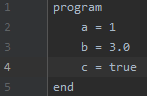
num\_line – номер рядка програми;

lexeme – лексема;

token – токен лексеми;

idxIdConst – iндекс iдентифiкатора або константи у таблицi iдентифiкаторiв та констант вiдповiдно.

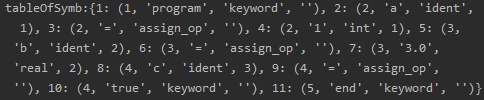
Наприклад, для програми з п’яти рядкiв:



Лексичний аналiзатор будує таблицю символiв, яка синтаксичним аналiзатором

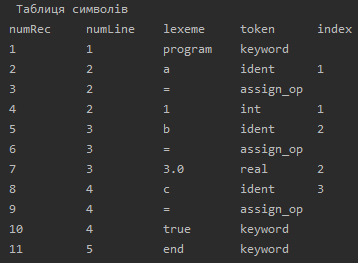
виводиться у консоль командою

print(’tableOfSymb:0’.format(tableOfSymb)):



Бiльш зручний для читання варiант виводиться лексичним аналiзатором у

форматi num\_line lexeme token idxIdConst (без номера запису у таблицi розбору n\_rec):



**Приклади програм для тестування**

Серед прикладiв програм для тестування лексичного аналiзатора є один базовий , який мiстить синтаксичнi конструкцiї, наявнi у розробленiй мовi. Всi iншi приклади мiстять код, який або демонструє бiльш складнi синтаксичнi конструкцiї, наприклад вкладенi, або мiстять синтаксичнi помилки рiзних типiв.

Тестування парсера передбачає синтаксичний розбiр прикладу коду та оцiнку правильностi його роботи (висновок про синтаксичну коректнiсть чи некоректнiсть коду, повiдомлення про знайдену помилку/помилки, її/їх локалiзацiю, дiагностичнi повiдомлення тощо).

**Програмна реалiзацiя транслятора**

**Загальнi положення**

На входi транслятора – результати лексичного аналiзу, включно з ознакою успiшностi розбору FSuccess зi значеннями (True,’Lexer’) або (False,’Lexer’), див. файл. lex\_my\_lang\_03.py.

Постфiксний код програми вхiдною мовою формується у списку postfixCode, до початку трансляцiї postfixCode = [].

Для трансляцiї використовується функцiя postfixTranslator(), яка викли кає функцiю-парсер parseProgram() тiльки у випадку коректностi лексичного аналiзу:

def postfixTranslator():

# чи був успiшним лексичний розбiр

if (True,’Lexer’) == FSuccess:

return parseProgram()

Граматика вхiдної мови, вiдрiзняється вiд розглянутої у лабораторнiй роботi № 2, тим, що в нiй нетермiнал Statement не має альтернативи: Statement = Assign. Тому функцiї синтаксичного аналiзатора parseProgram() та parseStatementList(), залишаться практично незмiнними, а в parseStatement()

залишиться тiльки виклик parseAssign() i не буде виклику parseIf(), див. файл.postfixExpr\_translator.py:

def parseStatement():

# print(’\t\t parseStatement():’)

# прочитаємо поточну лексему в таблицi розбору

numLine, lex, tok = getSymb()

# якщо токен - iдентифiкатор

# обробити iнструкцiю присвоювання

if tok == ’ident’:

parseAssign()

return True

# тут - ознака того, що всi iнструкцiї були коректно

# розiбранi i була знайдена остання лексема програми.

# тому parseStatement() має завершити роботу

elif (lex, tok) == (’end’,’keyword’):

return False

else:

# жодна з iнструкцiй не вiдповiдає

# поточнiй лексемi у таблицi розбору,

failParse(’невiдповiднiсть iнструкцiй’,

(numLine,lex,tok,’ident або if’))

return False

А от вже реалiзацiя parseAssign() у складi транслятора вiдрiзнятиметься вiд її реалiзацiї у синтаксичному аналiзаторi наявнiстю семантичних процедур.

**Семантичнi процедури**

ПОЛIЗ виразу можна знайти так:

*•* додати до ПОЛIЗУ вираз, якщо вiн – iдентифiкатор, константа або опе ратор (пiсля рекурсивної обробки аргументiв (аргументу) опрератора);

*•* обробляючи вираз у дужках – iгнорувати дужки.

Доповнимо синтаксичний аналiзатор, зважаючи на те, що вiн якраз обробляє всi елементи виразу рекурсивно, семантичними процедурами (у формi iнструкцiй).

Цi семантичнi процедури, як ми тiльки-но з’ясували, додаватимуть до ПОЛIЗУ вираз, якщо вiн – iдентифiкатор, константа або оператор (пiсля рекурсивної обробки аргументiв опрератора), а при обробцi виразу у дужках iгнору ватимуть дужки.

Як це видно з граматики вхiдної мови, всi цi випадки тра пляються при обробцi нетермiналiв Assign, Expression, Term i Factor. Отож треба доповнити функцiї parseAssign(), parseExpression(), parseTerm() i parseFactor() семантичними процедурами.

Далi розглянемо приклад реалiзацiї семантичних процедур у названих функцiях.

**У кодi parseAssign()**

    Отже при обробцi кожної iнструкцiї присвоювання функцiя parseAssign() безпосередньо доповнює ПОЛIЗ двома символами у рядках 11 та 27. Все, що потрапить у ПОЛIЗ при розборi виразу Expression, додасть функцiя parseExpression() та викликанi нею функцiї.

def parseAssign():

# номер запису таблицi розбору

global numRow, postfixCode

# print(’\t’\*4+’parseAssign():’)

# взяти поточну лексему

# вже вiдомо, що це - iдентифiкатор

 \_numLine, lex, \_tok = getSymb()

  # починаємо трансляцiю iнструкцiї присвоювання за означенням:

postfixCode.append(lex) # Трансляцiя

  # ПОЛIЗ iдентифiкатора - iдентифiкатор

  if toView: configToPrint(lex,numRow)

# встановити номер нової поточної лексеми

 numRow += 1

  # print(’\t’\*5+’в рядку {0} - {1}’.format(numLine,(lex, tok)))

# якщо була прочитана лексема - ’:=’

  if parseToken(’=’,’assign\_op’,’\t\t\t\t\t’):

  # розiбрати арифметичний вираз

  parseExpression()

# Трансляцiя (тут нiчого не робити)

# ця функцiя сама згенерує

# та додасть ПОЛIЗ виразу

  postfixCode.append(’=’)# Трансляцiя

  # Бiнарний оператор ’=’

# додається пiсля своїх операндiв

if toView: configToPrint(’=’,numRow)

  return True

  else: return False

**У кодi parseExpression()**

Власне додавання коду до ПОЛIЗ здiйснюється тiльки у рядку 21, а саме додається оператор (лексема) ’+’ або ’-’ пiсля того, як будуть обробленi його операнди у рядках 5 та 16:

def parseExpression():

global numRow, postfixCode

#print(’\t’\*5+’parseExpression():’)

 \_numLine, lex, tok = getSymb()

parseTerm() # Трансляцiя (тут нiчого не робити)

# ця функцiя сама згенерує

# та додасть ПОЛIЗ доданка

F = True

 # продовжувати розбирати Доданки (Term)

 # роздiленi лексемами ’+’ або ’-’

 while F:

\_numLine, lex, tok = getSymb()

if tok in (’add\_op’):

numRow += 1

# print(’\t’\*6+’в рядку {0} - {1}’.format(numLine,(lex, tok))) parseTerm()

# Трансляцiя (тут нiчого не робити)

# ця функцiя сама згенерує

# та додасть ПОЛIЗ доданка

# Трансляцiя

postfixCode.append(lex)

# lex - бiнарний оператор ’+’ чи ’-’

# додається пiсля своїх операндiв

if toView: configToPrint(lex,numRow)

else:

  F = False

return True

**У parseTerm() та parseFactor()**

Таким же чином додаються семантичнi процедури у функцiях розбору не термiналiв Term та Factor – parseTerm() та parseFactor() вiдповiдно. Код цих функцiй мiстить iнструкцiю postfixCode.append(lex), який допов нює ПОЛIЗ лексемою lex, що має симсл оператора (лексеми) ’\*’ або ’/’ у фун кцiї parseTerm(), або константи чи iдентифiкатора у функцiї parseFactor(), див. код функцiй у файлi postfixExpr\_translator.py.

**Покроковий перегляд процесу трансляцiї**

Для перегляду конфiгурацiї транслятора була визначена функцiя configToPrint(lex,numRow), виклик якої здiйснюється командою if toView: configToPrint(lex,numRow) пiсля кожної семантичної процедури, див. напр. рядки 14 та 30 коду функцiї parseAssign(). Функцiя configToPrint(lex,numRow) виводить у консоль: оброблену лексему, запис таблицi символiв та отриманий пiсля виконання семантичної процедури ПОЛIЗ:

def configToPrint(lex,numRow):

stage = ’\nКрок трансляцiї\n’

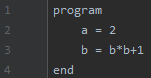
stage += ’лексема: \’{0}\’\n’

stage += ’tableOfSymb[{1}] = {2}\n’

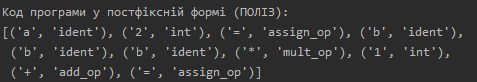
stage += ’postfixCode = {3}\n’

# tpl = (lex,numRow,str(tableOfSymb[numRow]),str(postfixCode)) print(stage.format(lex,numRow,str(tableOfSymb[numRow]),str(postfixCode)))

Переглянемо процес трансляцiї програми



Запуск транслятора та його робота виглядатимуть у консолi приблизно так: >python postfixExpr\_translator.py



**Програмна реалiзацiя iнтерпретатора**

**Загальнi положення**

На входi iнтерпретатора – побудований транслятором ПОЛIЗ вхiдної про грами у списку postfixCode, таблицi iдентифiкаторiв та констант, а також ознака успiшностi синтаксичного розбору та трансляцiї FSuccess зi значеннями (True, ’Translator’) або (False, ’Translator’).

Постфiксний код, крiм лексеми, мiстить i її токен, так для попереднього прикладу будується ПОЛIЗ у формi:

postfixCode=[('6', 'int'), ('=', 'assign\_op'), ('b', 'ident'), ('5', 'int'), ('5', 'int'), ('\*', 'mult\_op'), ('a', 'ident'), ('/', 'mult\_op'), ('=', 'assign\_op')]

Для цього замiсть семантичних процедур типу postfixCode.append(lex) використовуюються iнструкцiї postfixCode.append((lex,tok)) див. файл postfixExpr\_translator\_02.py у тецi postfixExpr\postfixExpr\_interpreter. Клас Stack iмпортується з модуля stack01.

**Iнтерпретатор**

**Базовi кроки алгоритму**

Функцiя postfixInterpreter(), за умови успiшностi етапу трансляцiї, ви кликає postfixProcessing():

def postfixInterpreter():

FSuccess = postfixTranslator()

# чи була успiшною трансляцiя

if (True,’Translator’) == FSuccess:

print(’\nПостфiксний код: \n{0}’.format(postfixCode))

return postfixProcessing()

else:

# Повiдомити про факт виявлення помилки

print(’Interpreter: Translator завершив роботу аварiйно’)

return False

Оскiльки ПОЛIЗ арифметичних виразiв, як i iнструкцiй присвоювання, ви конується послiдовно у тому порядку, як вiн записаний у списку postfixCode, то функцiя postfixProcessing() просто за iндексами елементiв коду, див. ря док 5, вилучає найлiвiший елемент, рядок 6, i якщо це – iдентифiкатор, або константа, кладе його на стек, рядок 8, або передає на обробку, рядок 9, у doIt(lex,tok):

def postfixProcessing():

global stack, postfixCode

maxNumb=len(postfixCode)

try:

for i in range(0,maxNumb):

lex,tok = postfixCode.pop(0)

if tok in (’int’,’float’,’ident’):

stack.push((lex,tok))

else: doIt(lex,tok)

if toView: configToPrint(i+1,lex,tok,maxNumb)

  return True

  except SystemExit as e:

  \# Повiдомити про факт виявлення помилки

  print(’RunTime: Аварiйне завершення програми з кодом {0}’.format(e))

  return True

*Зауважимо*, що *вилучення запису* з postfix-коду запроваджено для зручностi покрокового перегляду, i *можливе тiльки* при вiдсутностi у кодi iнструкцiй умовного чи безумовного переходу, як у цiй лабораторнiй роботi.

Оператори обробляються функцiєю doIt(lex,tok) таким чином. Якщо це – оператор присвоювання, то знiмаються два операнди, рядки 5 i 7, i правий записують у таблицю iдентифiкаторiв як значення змiнної з iдентифiкатором-лiвим операндом, рядки 15-16. Якщо це – це арифметичний оператор, то операнди та оператор передають на обробку функцiї

processing\_add\_mult\_op((lexL,tokL),lex,(lexR,tokR)), рядок 26.

def doIt(lex,tok):

   global stack, postfixCode, tableOfId, tableOfConst, tableOfLabel

   if (lex,tok) == ('=', 'assign\_op'):

       # зняти з вершини стека запис (правий операнд = число)

       (lexL,tokL) = stack.pop()

       # зняти з вершини стека ідентифікатор (лівий операнд)

       (lexR,tokR) = stack.pop()

       # виконати операцію:

       # оновлюємо запис у таблиці ідентифікаторів

       # ідентифікатор/змінна

       # (index не змінюється,

       # тип - як у константи,

       # значення - як у константи)

       tableOfId[lexR] = (tableOfId[lexR][0],  tableOfConst[lexL][1], tableOfConst[lexL][2])

   elif tok in ('add\_op','mult\_op'):

       # зняти з вершини стека запис (правий операнд)

       (lexR,tokR) = stack.pop()

       # зняти з вершини стека запис (лівий операнд)

       (lexL,tokL) = stack.pop()

       if (tokL,tokR) in (('int', 'real'),('real','int')):

           failRunTime('невідповідність типів',((lexL,tokL),lex,(lexR,tokR)))

       else:

           processing\_add\_mult\_op((lexL,tokL),lex,(lexR,tokR))

           # stack.push()

           pass

   return True

**Семантика арифметики вхiдної мови**

Семантика арифметичних операторiв мови, забезпечується функцiями processing\_add\_mult\_op(operandL,operator,operandR) та getValue((valL,lexL,tokL),lex,(valR,lexR,tokR)).

Функцiя processing\_add\_mult\_op((lexL,tokL),lex,(lexR,tokR)) визначає значення та типи значень лiвого i правого операндiв-iдентифiкаторiв, рядки 10 та 18 вiдповiдно, чи значення констант, рядки 12 та 20. Потiм передає всi цi ле ксеми, їх типи та значення функцiї getValue((valL,lexL,tokL),lex,(valR,lexR,tokR)), рядок 21, для обчислення результату. Помилка генерується, якщо один з опе рандiв – iдентифiкатор з невизначеним типом (’type\_undef’) , р. 7 та 15.

def processing\_add\_mult\_op(ltL,lex,ltR):

   global stack, postfixCode, tableOfId, tableOfConst, tableOfLabel

   lexL,tokL = ltL

   lexR,tokR = ltR

   if tokL == 'ident':

       # print(('===========',tokL , tableOfId[lexL][1]))

       # tokL = tableOfId[lexL][1]

       if tableOfId[lexL][1] == 'type\_undef':

           failRunTime('неініціалізована змінна',(lexL,tableOfId[lexL],(lexL,tokL),lex,(lexR,tokR)))

       else:

           valL,tokL = (tableOfId[lexL][2],tableOfId[lexL][1])

   else:

       valL = tableOfConst[lexL][2]

   if tokR == 'ident':

       # print(('===========',tokL , tableOfId[lexL][1]))

       # tokL = tableOfId[lexL][1]

       if tableOfId[lexR][1] == 'type\_undef':

           failRunTime('неініціалізована змінна',(lexR,tableOfId[lexR],(lexL,tokL),lex,(lexR,tokR)))

       else:

           valR,tokR = (tableOfId[lexR][2],tableOfId[lexR][1])

   else:

       valR = tableOfConst[lexR][2]

   # if :

       # print(('lexL',lexL,tableOfConst))

       # valL = tableOfConst[lexL][2]

       # valR = tableOfConst[lexR][2]

   getValue((valL,lexL,tokL),lex,(valR,lexR,tokR))

Функцiя getValue((valL,lexL,tokL),lex,(valR,lexR,tokR)) реалiзує семантику обчислення: генерує винятки у випадку невiдповiдностi типiв аргументiв, р. 5-6, або у випадку, якщо правий аргумент має значення нуль, р. 13-14. В рештi випадкiв, в залежностi вiд лексеми оператора та типу операндiв, засобами мови python обчислюється результат, р. 7-12 та 15-18, який потiм кладеться на стек, р. 21, та заноситься до таблицi констант, р. 22.

def getValue(vtL,lex,vtR):

   global stack, postfixCode, tableOfId, tableOfConst, tableOfLabel

   valL,lexL,tokL = vtL

   valR,lexR,tokR = vtR

   if (tokL,tokR) in (('int', 'real'),('real' ,'int')):

           failRunTime('невідповідність типів',((lexL,tokL),lex,(lexR,tokR)))

   elif lex == '+':

       value = valL + valR

   elif lex == '-':

       value = valL - valR

   elif lex == '\*':

       value = valL \* valR

   elif lex == '/' and valR ==0:

       failRunTime('ділення на нуль',((lexL,tokL),lex,(lexR,tokR)))

   elif lex == '/' and tokL=='real':

       value = valL / valR

   elif lex == '/' and tokL=='int':

       value = int(valL / valR)

   else:

       pass

   stack.push((str(value),tokL))

   toTableOfConst(value,tokL)

Функцiя failRunTime() обробляє помилки.

def failRunTime(str,tuple):

if str == ’невiдповiднiсть типiв’:

((lexL,tokL),lex,(lexR,tokR))=tuple

print(’RunTime ERROR: \n\t Типи операндiв вiдрiзняються

у {0} {1} {2}’.format((lexL,tokL),lex,(lexR,tokR)))

exit(1)

elif str == ’неiнiцiалiзована змiнна’:

(lx,rec,(lexL,tokL),lex,(lexR,tokR))=tuple

print(’RunTime ERROR: \n\t Значення змiнної {0}:{1} не визначене. Зустрiлось у {2} {3} {4}’.format(lx,rec,(lexL,tokL),lex,(lexR,tokR))) exit(2)

elif str == ’дiлення на нуль’:

((lexL,tokL),lex,(lexR,tokR))=tuple

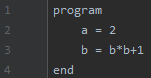
print(’RunTime ERROR: \n\t Дiлення на нуль

у {0} {1} {2}. ’.format((lexL,tokL),lex,(lexR,tokR)))

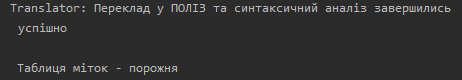
exit(3)

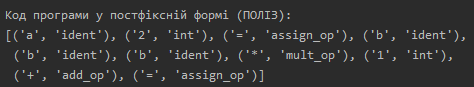
**Iнтерпретацiя**

Перевiримо iнтерпретатор на прикладi програми

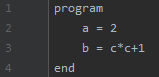


Результат:

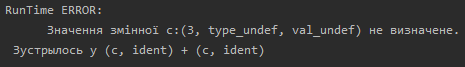




Невизначена на момент використання змiнна у арифметичному виразi, як-от с у програмi.



спричинює повiдомлення:



**Висновки:** Був розроблений інтерпретатор для мови програмування FuriousPeach на мові Python. Було розглянуто основні принципи роботи інтерпретатора та проведені тестування коректності його роботи. Інтерпретатор працює як потрібно.