НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

Лабораторна робота №6

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

Виконав:

студент групи ІО-42, ФІОТ

Коваленко В’ячеслав Сергійович

ЗК : 4209

м. Київ 2016 р.

Варіант



Лістинг програм

expression = res.expression

memory\_bank = []

def gen\_exc(exc\_text):

print('Exception : ' + exc\_text)

def save\_empty\_val(data\_type, key):

global memory\_bank

for el in memory\_bank:

if el.key == key:

print('already in memory')

return False

if len(key) == 0:

gen\_exc('empty name')

return False

if not key[0] in res.liters[0]:

print('key[0] trouble, key = ', key)

return False

for i in range(len(key)):

if not key[i] in res.liters[0] + res.numbers[0]:

print('key in array')

return False

memory\_bank.append(DataVar(data\_type=data\_type, key=key))

return True

def save\_empty\_array(data\_type, key, size):

global memory\_bank

# check key

for el in memory\_bank:

if el.key == key:

print('already in memory')

return False

if len(key) == 0 or len(size) == 0:

gen\_exc('size or name is empty')

return False

if not key[0] in res.liters[0]:

print('key[0] not in array, key = ', key)

return False

for i in range(len(key)):

if not key[i] in res.liters[0] + res.numbers[0]:

print('key in array')

return False

for i in range(len(size)):

if not size[i] in res.numbers[0]:

print('size array error')

return False

memory\_bank.append(DataArray(data\_type=data\_type, key=key, size=size))

return True

def save\_var(data\_type, key, value):

global memory\_bank

# check key

for el in memory\_bank:

if el.key == key:

print('already in memory')

return False

if len(key) == 0 or len(str(value)) == 0:

gen\_exc('value or name is empty')

return False

if not key[0] in res.liters[0]:

print('key[0] not in var, key = ', key)

return False

for i in range(len(key)):

if not key[i] in res.liters[0] + res.numbers[0]:

print('key in var')

return False

# depends on type

if data\_type == 'int' or data\_type == 'short' or data\_type == 'long':

start = 1 if value[0] == '-' else 0

for i in range(start, len(value)):

if not value[i] in res.numbers[0]:

print('value error')

return False

elif data\_type == 'double' or data\_type == 'float':

start = 1 if value[0] == '-' else 0

dot\_ind = value.find('.')

if dot\_ind == -1 or dot\_ind == start or dot\_ind == len(value) - 1:

print('error with dot')

return False

for i in range(start, dot\_ind):

if not value[i] in res.numbers[0]:

print('value error')

return False

for i in range(dot\_ind+1, len(value)):

if not value[i] in res.numbers[0]:

print('value error')

return False

elif data\_type == 'char':

if not (len(value) == 3 and value[0] == "'" and value[-1] == "'"):

print('value error')

return False

memory\_bank.append(DataVar(data\_type=data\_type, key=key, value=value))

return True

def save\_elem\_array(el\_ind, index, value):

data\_type = memory\_bank[el\_ind].data\_type

# depends on type

if data\_type == 'int' or data\_type == 'short' or data\_type == 'long':

start = 1 if value[0] == '-' else 0

for i in range(start, len(value)):

if not value[i] in res.numbers[0]:

print('value error')

return False

elif data\_type == 'double' or data\_type == 'float':

start = 1 if value[0] == '-' else 0

dot\_ind = value.find('.')

if dot\_ind == -1 or dot\_ind == start or dot\_ind == len(value) - 1:

print('error with dot')

return False

for i in range(start, dot\_ind):

if not value[i] in res.numbers[0]:

print('value error')

return False

for i in range(dot\_ind + 1, len(value)):

if not value[i] in res.numbers[0]:

print('value error')

return False

elif data\_type == 'char':

if not (len(value) == 3 and value[0] == "'" and value[-1] == "'"):

print('value error')

return False

if index > memory\_bank[el\_ind].size:

print('value error')

return False

memory\_bank[el\_ind].set\_element(ind=index, value=value)

return True

def parse\_declaration(sequence):

print('parse declaration - ', sequence)

for ind in reversed(range(len(sequence))):

sequence = sequence[:]

if sequence[ind][-1] == ',':

sequence[ind] = sequence[ind][:-1]

sequence.insert(ind + 1, ',')

sequence = [x for x in sequence if x != '']

type = sequence.pop(0)

print('sequence ', sequence)

cur\_ind = 0

is\_next = True if len(sequence) > 0 else False

if not is\_next:

return False

is\_wrong = False

while (is\_next):

if not sequence[cur\_ind] in res.common\_list:

for el in res.undefined:

if el in sequence[cur\_ind]:

is\_wrong = True

gen\_exc('undefined symbol')

return False

if is\_wrong: is\_next = False

return False

else:

# check is array

if '[' in sequence[cur\_ind]:

bkt\_st\_ind = sequence[cur\_ind].find('[')

if bkt\_st\_ind != 0 and bkt\_st\_ind < len(sequence[cur\_ind]) - 2:

bkt\_end\_ind = sequence[cur\_ind].find(']')

if bkt\_end\_ind == -1 or bkt\_end\_ind < bkt\_st\_ind or bkt\_end\_ind != len(sequence[cur\_ind]) - 1:

gen\_exc('error with brackets in array declaration')

is\_wrong = True

return False

else:

if save\_empty\_array(type, sequence[cur\_ind][:bkt\_st\_ind],

sequence[cur\_ind][bkt\_st\_ind + 1:bkt\_end\_ind]):

# check next element

if len(sequence)-1 == cur\_ind:

return True

if len(sequence) - 1 - cur\_ind == 1:

gen\_exc("cannot exist next element")

is\_wrong = True

return False

if len(sequence) - 1 - cur\_ind >= 2:

if sequence[cur\_ind + 1] == ',' and sequence[cur\_ind + 2] != ',':

cur\_ind += 2

continue

else:

gen\_exc('next or next+1 is wrong')

is\_wrong = True

return False

else:

gen\_exc('error occurred in storing array')

is\_wrong = True

return False

else:

gen\_exc('error with "["')

is\_wrong = True

return False

# check if is assignment

elif '=' in sequence[cur\_ind]:

eq\_ind = sequence[cur\_ind].find('=')

if eq\_ind == 0 or eq\_ind == len(sequence[cur\_ind]) - 1:

gen\_exc('error with "="')

is\_wrong = True

return False

for i in sequence[cur\_ind]:

if i == '[' or i == ']':

gen\_exc("don't support square brackets in assignment")

is\_wrong = True

return False

if not is\_wrong:

if save\_var(type, sequence[cur\_ind][:eq\_ind], sequence[cur\_ind][eq\_ind + 1:]):

# check next element

if len(sequence) - 1 == cur\_ind:

return True

if len(sequence) - 1 - cur\_ind == 1:

gen\_exc('cannot exist next element')

is\_wrong = True

return False

if len(sequence) - 1 - cur\_ind >= 2:

if sequence[cur\_ind + 1] == ',' and sequence[cur\_ind + 2] != ',':

cur\_ind += 2

continue

else:

gen\_exc('next or next+1 is wrong')

is\_wrong = True

return False

else:

gen\_exc('error occurred in storing var')

is\_wrong = True

return False

else:

if save\_empty\_val(type, sequence[cur\_ind]):

# check next element

if len(sequence) - 1 == cur\_ind:

return True

if len(sequence) - 1 - cur\_ind == 1:

gen\_exc('cannot exist next element')

is\_wrong = True

return False

if len(sequence) - 1 - cur\_ind >= 2:

if sequence[cur\_ind + 1] == ',' and sequence[cur\_ind + 2] != ',':

cur\_ind += 2

continue

else:

gen\_exc('next or next+1 is wrong')

is\_wrong = True

return False

else:

gen\_exc('error occurred in storing var without value')

is\_wrong = True

return False

else:

gen\_exc('wrong symbol in variable')

return False

if is\_wrong:

gen\_exc('wrong symbol in variable')

return False

print('Okay')

def calc\_operation(text, data\_type):

operands = []

for i in range(len(text)):

if text[i] in res.operation\_list:

operands.append(text[:i])

operands.append(text[i])

operands.append(text[i+1:])

if (operands[0] is None) or (operands[2] is None):

gen\_exc('None object')

return False

name = ''

index = ''

if '[' in operands[0]:

bkt\_st\_ind = operands[0].find('[')

if bkt\_st\_ind != 0 and bkt\_st\_ind < len(operands[0]) - 2:

bkt\_end\_ind = operands[0].find(']')

if bkt\_end\_ind == -1 or bkt\_end\_ind < bkt\_st\_ind or bkt\_end\_ind != len(operands[0]) - 1:

gen\_exc('error with brackets in array declaration')

return False

else:

name = operands[0][:bkt\_st\_ind]

index = operands[0][bkt\_st\_ind+1:bkt\_end\_ind]

else:

gen\_exc('error in statement')

return False

else:

name = operands[0]

is\_in\_memory = False

ind = -1

for i in range(len(memory\_bank)):

if memory\_bank[i].key == name:

ind = i

is\_in\_memory = True

if is\_in\_memory:

# PASTE HERE

if index != '':

is\_number = True

for i in index:

if i not in res.numbers:

is\_number = False

break

# array index not number

if not is\_number:

try:

for el in memory\_bank:

if el.key == index:

index = el.value

except Exception:

gen\_exc('wrong array\_index')

return False

is\_number = True

for i in index:

if i not in res.numbers[0]:

gen\_exc('wrong array\_index')

return False

prost = True

try:

operands[0] = (memory\_bank[ind].get\_value(int(index)))

prost = False

except Exception:

if prost:

operands[0] = (memory\_bank[ind].get\_value())

print('Operands, after - ', operands)

if operands[0] is None:

gen\_exc('not initialized')

return False

ind = ''

name = ''

if '[' in operands[2]:

bkt\_st\_ind = operands[2].find('[')

if bkt\_st\_ind != 0 and bkt\_st\_ind < len(operands[2]) - 2:

bkt\_end\_ind = operands[2].find(']')

if bkt\_end\_ind == -1 or bkt\_end\_ind < bkt\_st\_ind or bkt\_end\_ind != len(operands[2]) - 1:

gen\_exc('error with brackets in array declaration')

return False

else:

name = operands[2][:bkt\_st\_ind]

index = operands[2][bkt\_st\_ind+1:bkt\_end\_ind]

else:

gen\_exc('error in statement')

return False

else:

name = operands[2]

is\_in\_memory = False

ind = -1

for i in range(len(memory\_bank)):

if memory\_bank[i].key == name:

ind = i

is\_in\_memory = True

if is\_in\_memory:

def parseOperation(token):

text = ''

for el in token:

text += el

print('PARSE OPERATION - ', text)

eq\_ind = text.find('=')

if eq\_ind == -1 or eq\_ind == 0 or eq\_ind == len(text)-1:

gen\_exc('error with "=" in operation')

return False

if '?' in text[eq\_ind+1:] and ':' in text[eq\_ind+1:]:

if '[' in text[:eq\_ind]:

end = -1

if ']' in text[:eq\_ind]:

for i in range(len(text[:eq\_ind])):

if text[i] == ']':

end = i

break

else:

gen\_exc('no "]"')

return False

real\_key = text[:end]

real\_index = text[text[0:eq\_ind].find('['):text[0:eq\_ind].find(']')]

else:

real\_key = text[:eq\_ind]

index\_in\_memory = -1

for i in range(len(memory\_bank)):

if memory\_bank[i].key == real\_key:

index\_in\_memory = i

if index\_in\_memory == -1:

gen\_exc('mot in memory')

return False

print('Real key - ', real\_key, ' Real ind - ', real\_index)

# parse ternary operator

quest\_ind = text.find('?')

colon\_ind = text.find(':')

if (quest\_ind < eq\_ind) or (quest\_ind-eq\_ind == 1) or (quest\_ind>colon\_ind) or (colon\_ind-quest\_ind==1) or (colon\_ind == len(text)-1):

gen\_exc('error in ternary statement')

return False

condition = text[eq\_ind+1:quest\_ind]

if\_true = text[quest\_ind+1:colon\_ind]

if\_false = text[colon\_ind+1:]

if len(condition) == 0 or len(if\_true) == 0 or len(if\_false) == 0:

gen\_exc('empty condition for ternary operator')

return False

# check condition

print('Condition - ', condition)

el\_ind = -1

for i in range(len(memory\_bank)):

if memory\_bank[i].key == condition:

el\_ind = i

result = True

if(el\_ind != -1):

# is in memory

if memory\_bank[el\_ind].data\_type == 'int' or memory\_bank[el\_ind].data\_type == 'short' or memory\_bank[el\_ind].data\_type == 'long':

if memory\_bank[el\_ind].value is None:

gen\_exc('Null Value')

return False

elif memory\_bank[el\_ind].value == '0':

result = False

else:

result = True

elif memory\_bank[el\_ind].data\_type == 'float' or memory\_bank[el\_ind].data\_type == 'double':

if memory\_bank[el\_ind].value is None:

gen\_exc('Null Value')

return False

elif memory\_bank[el\_ind].value != '0.0':

result = True

else:

result = False

else:

if condition == '0' or condition == '0.0' or condition == 'false':

result = False

else:

if condition == 'true':

result = True

else:

for i in condition:

if not i in res.numbers[0]:

gen\_exc('undefined condition')

return False

tex = if\_true if result else if\_false

has\_operation = False

for oper in res.operation\_list:

if oper in tex:

has\_operation = True

break

if has\_operation:

calc\_result = calc\_operation(tex, memory\_bank[index\_in\_memory].data\_type)

temp\_obj = memory\_bank.pop(index\_in\_memory)

save\_var(data\_type = temp\_obj.data\_type, key=temp\_obj.key, value=str(calc\_result))

else:

temp\_ind = -1

for i in range(len(memory\_bank)):

if memory\_bank[i].key == tex: if index\_in\_memory != -1:

typ = memory\_bank[index\_in\_memory].data\_type

name = memory\_bank[index\_in\_memory].key

if temp\_ind == -1:

# is not in list

memory\_bank.pop(index\_in\_memory)

save\_var(data\_type=typ, key=name, value=tex)

pass

else:

memory\_bank.pop(index\_in\_memory)

save\_var(data\_type=typ, key=name, value=str(memory\_bank[temp\_ind-1].value))

else:

# parse simple operation

#very dangerous

key = text[0]

print('Key = ', key)

data\_type = ''

el\_ind = -1

for i in range(len(memory\_bank)):

if memory\_bank[i].key == key:

el\_ind = i

if el\_ind == -1:

gen\_exc('not in memory')

return False

else:

data\_type = memory\_bank[el\_ind].data\_type

is\_operation = False

operation = text[eq\_ind + 1:]

for op in res.operation\_list:

if op in operation:

is\_operation = True

break

if is\_operation:

gen\_exc('unsupported') return False

else:

if '[' in text[0:eq\_ind]:

bkt\_st\_ind = text[0:eq\_ind].find('[')

if bkt\_st\_ind != 0 and bkt\_st\_ind < len(text[0:eq\_ind]) - 2:

bkt\_end\_ind = text[0:eq\_ind].find(']')

if bkt\_end\_ind == -1 or bkt\_end\_ind < bkt\_st\_ind or bkt\_end\_ind != len(text[0:eq\_ind])-1:

gen\_exc('error with brackets in array declaration') return False

else:

if save\_elem\_array(el\_ind=el\_ind, index=text[bkt\_st\_ind+1:bkt\_end\_ind], value=text[eq\_ind+1:]):

print('okay')

else:

gen\_exc('error occurred in saving')

return False

else:pass

else:

memory\_bank.pop(el\_ind)

save\_var(data\_type=data\_type, key=key, value=text[eq\_ind+1:]) return True

def recognize\_operation(token):

elements\_list = token.split(' ')

elements\_list = [el for el in elements\_list if el != '' and el != ' ']

if len(elements\_list) > 0:

# if first element is data type, else otherwise

if elements\_list[0] in res.data\_type:

return parse\_declaration(elements\_list)

else:

return parseOperation(elements\_list)

def parse(expression):

token\_list = expression.split(';')

token\_list = [token for token in token\_list if token != '']

for token in token\_list:

if not recognize\_operation(token):

break

for el in memory\_bank:

print(el)

def main():

parse(expression)

if \_\_name\_\_ == '\_\_main\_\_':

main()