MyCode

Intermediate Code Generation using 3 Address Code

Main.py

```python
class Conversion:
    def __init__(self, capacity):
        self.top = -1
        self.capacity = capacity
        self.array = []
        self.output = []
        self.precedence = {'+':1, '-':1, '*':2, '/':2, '^':3}

    def isEmpty(self):
        return True if self.top == -1 else False

    def peek(self):
        return self.array[-1]
    def pop(self):
        if not self.isEmpty():
            self.top -= 1
            return self.array.pop()
        else:
            return "$"

    def push(self, op):
        self.top += 1
        self.array.append(op)

    def isOperand(self, ch):
        return ch.isalpha()

    def notGreater(self, i):
        try:
            a = self.precedence[i]
            b = self.precedence[self.peek()]
            return True if a  <= b else False
        except KeyError:
            return False

    def infixToPostfix(self, exp):
        for i in exp:
            if self.isOperand(i):
                self.output.append(i)
            elif i  == '(':
                self.push(i)
            elif i == ')':
                while( (not self.isEmpty()) and self.peek() != '('):
                    a = self.pop()
                    self.output.append(a)
                if (not self.isEmpty() and self.peek() != '('):
                    return -1
                else:
                    self.pop()
            else:
                while(not self.isEmpty() and self.notGreater(i)):
                    self.output.append(self.pop())
                self.push(i)
        while not self.isEmpty():
            self.output.append(self.pop())
        print("Postfix notation: ", end="")
```

```python
        print ("".join(self.output))
        return "".join(self.output)

def T_A_C(exp):
        stack = []
        x = 1
        obj = Conversion(len(exp))
        postfix = obj.infixToPostfix(exp)
        print()
        for i in postfix:
            if i in "abcdefghijklmnopqrstuvwxyz" or i in "0123456789":
                stack.append(i)
            elif i == '-':
                op1 = stack.pop()
                print("t(",x,")","=",i,op1)
                stack.append("t(%s)" %x)
                x = x+1
                if stack != []:
                    op2 = stack.pop()
                    op1 = stack.pop()
                    print("t(",x,")","=",op1,"+",op2)
                    stack.append("t(%s)" %x)
                    x = x+1
            elif i == '=':
                    op2 = stack.pop()
                    op1 = stack.pop()
                    print(op1,i,op2)

            else:
                op1 = stack.pop()
                if stack !=[]:
                        op2 = stack.pop()
                        print("t(",x,")","=",op2,i,op1)
                        stack.append("t(%s)" %x)
                        x = x+1
        return


def Quadruple(exp):
        stack = []
        op = []
        x = 1
        obj = Conversion(len(exp))
        postfix = obj.infixToPostfix(exp)
        print("\n{0:^4s} | {1:^4s} | {2:^4s} |
{3:4s}".format('op','arg1','arg2','result'))
        for i in postfix:
            if i in "abcdefghijklmnopqrstuvwxyz" or i in "0123456789":
                stack.append(i)
            elif i == '-':
                op1 = stack.pop()
                stack.append("t(%s)" %x)
                print("{0:^4s} | {1:^4s} | {2:^4s} |
{3:4s}".format(i,op1,"(-)"," t(%s)" %x))
                x = x+1
                if stack != []:
                    op2 = stack.pop()
                    op1 = stack.pop()
                    print("{0:^4s} | {1:^4s} | {2:^4s} |
{3:4s}".format("+",op1,op2," t(%s)" %x))
                    stack.append("t(%s)" %x)
                    x = x+1
            elif i == '=':
                    op2 = stack.pop()
                    op1 = stack.pop()
                    print("{0:^4s} | {1:^4s} | {2:^4s} |
{3:4s}".format(i,op2,"(-)",op1))
            else:
                op1 = stack.pop()
                op2 = stack.pop()
                print("{0:^4s} | {1:^4s} | {2:^4s} |
{3:4s}".format(i,op2,op1," t(%s)" %x))
                stack.append("t(%s)" %x)
                x = x+1
        return


def Triple(exp):
        stack = []
        op = []
        x = 0
        h = 0
        obj = Conversion(len(exp))
        postfix = obj.infixToPostfix(exp)
        print("\n{0:^4s} | {1:^4s} | {2:^4s} |
{3:^4s}".format('#','op','arg1','arg2'))
```

```python
        for i in postfix:
            if i in "abcdefghijklmnopqrstuvwxyz" or i in "0123456789":
                stack.append(i)
            elif i == '-':
                op1 = stack.pop()
                stack.append("(%s)" %x)
                print("({0:^4s}) | {1:^4s} | {2:^4s} |
{3:^4s}".format(str(h),i,op1,"(-)"))
                x = x+1
                h = h+1
                if stack != []:
                    op2 = stack.pop()
                    op1 = stack.pop()
                    print("{0:^4s} | {1:^4s} | {2:^4s} |
{3:^4s}".format(str(h),"+",op1,op2))
                    stack.append("(%s)" %x)
                    x = x+1
                    h = h+1
            elif i == '=':
                    op2 = stack.pop()
                    op1 = stack.pop()
                    print("{0:^4s} | {1:^4s} | {2:^4s} |
{3:^4s}".format(str(h),i,op1,op2))
                    h = h+1
            else:
                op1 = stack.pop()
                if stack != []:
                        op2 = stack.pop()
                        print("{0:^4s} | {1:^4s} | {2:^4s} |
{3:^4s}".format(str(h),i,op2,op1))
                        stack.append("(%s)" %x)
                        x = x+1
                        h = h+1
        return


if __name__ == '__main__':
    print("Intermediate Code Generation using 3 Address Code")
    exp = input("Enter a valid infix expression: ")
    print("\nThree Address Code")
    T_A_C(exp)
    print("\nQuardruple")
    Quadruple(exp)
    print("\nTriple")
    Triple(exp)
```

OUTPUT:

```
Intermediate Code Generation using 3 Address Code
Enter a valid infix expression: b*c+b*c

Three Address Code
Postfix notation: bc*bc*+

t( 1 ) = b * c
t( 2 ) = b * c
t( 3 ) = t(1) + t(2)

Quardruple
Postfix notation: bc*bc*+

 op   | arg1 | arg2 | result
 *    |  b   |  c   |  t(1)
 *    |  b   |  c   |  t(2)
 +    | t(1) | t(2) |  t(3)

Triple
Postfix notation: bc*bc*+

 #    |  op  | arg1 | arg2
 0    |  *   |  b   |  c
 1    |  *   |  b   |  c
 2    |  +   | (0)  | (1)
```

Lexical Analyzer

MyCode.txt

```
#include<stdio.h>
void main()
```

```
{
int a;
}
```

Main.py

```
with open("MyCode.txt", "r") as f:
  data = f.readlines()
  data = list(map(lambda x: x.replace("\n", "" ), data))
  #print(data)
  keywords =
['include','stdio.h','auto','break','case','char','const','continue','defau
lt','do','double','else','enum','extern','float','for','goto','if','int','l
ong',

'register','return','short','signed','sizeof','static','struct','switch','t
ypedef','union','unsigned','void','volatile','while']
  paranthesis = ['{','}','[',']','(',')','<','>']
  delimiters = [';',' ',',']
  print("Lexical Analyzer")
  for j in data:
    j = j.split()
    for i in j:
      if i in keywords:
        print("Keyword: \t\t",i)
      elif '#' in i:
        print("Header File: \t",i)
      elif '(' and ')' in i:
        print("Function: \t\t",i)
      elif i in paranthesis:
        print("Paranthesis: \t",i)
      elif i in delimiters:
        print("Delimiters: \t\t",i)
      else:
        print("Variable: \t\t",i)
```

Code Optimization - Dead Code Elimination

MyCode.txt

```
c = a * b
x = a
c = d * e
d = a * b + 4
```

Main.py

```python
import os
os.system("clear")


with open("MyCode.txt") as f:
    lines = []
    lines = [i.replace("\n", "") for i in f.readlines()]


def used(variable, code):
    counts = 0
    for i in code:
        if variable in i:
            counts += 1
    return False if counts > 0 else True

def redefined_variables():
    pass

def dead_code_elimination(lines):
    variable = []
    expressions = []

    lines = [i.replace(" ", "") for i in lines]

    # print(lines)
    print("------------------------------")
    print("Original code...")
    for i in lines:
        print(i)


    for i in lines:
        left, right = i.split("=")
        variable.append(left)
        expressions.append(right)

    # FIRST: REMOVING REDEFINED VARIABLES
    redefined_variables_line = []

    for i in range(len(variable)):
        if variable[i] in variable[i+1:]:
            redefined_variables_line.append(i)

    # print(redefined_variables_line)

    variable = [j for i, j in enumerate(variable) if i not in
redefined_variables_line]
    expressions = [j for i, j in enumerate(expressions) if i not in
redefined_variables_line]
    lines = [j for i, j in enumerate(lines) if i not in
redefined_variables_line]

    # SECOND: REMOVING SIMPLE ASSIGNMENT STATEMENTS NOT USED like x = 3
    dead_code = []
    dead_code_line = []

    for i in range(len(variable)):
        if len(expressions[i]) == 1:
            # print(variable[i], expressions[i+1:])
            if used(variable[i], expressions[i+1:]):
                # print(variable[i]+"="+expressions[i])
                dead_code.append((i, variable[i]+"="+expressions[i]))
                dead_code_line.append(i)


    print("------------------------------")
    print("Removing dead code...")
    for i, j in enumerate(lines):
        if i in dead_code_line:
            continue
        else:
            print(j)


    lines = [j for i, j in enumerate(lines) if i not in dead_code_line]
    return lines


dead_code_elimination(lines)
```

Code Optimization - Strength Reduction

MyCode.txt

```
i = 1
while (i<10)
{
    y = i * 5
}
```

Main.py

```
import os
os.system("clear")


with open("MyCode.txt") as f:
    lines = []
    lines = [i.replace("\n", "") for i in f.readlines()]

lines = [i.replace(" ", "") for i in lines]

print("-------------------------------")
print("Original code...")
for i in lines:
    print(i)


# find loop variable and update count
variable = []
expressions = []

for i, j in enumerate(lines):
```

```python
    try:
        left, right = j.split("=")
        variable.append((i, left))
        expressions.append((i, right))
    except:
        pass

# print(expressions)

loop_variable = variable[0][1]
update_variable = variable[1][1]

if expressions[1][1][-1].isdigit():
    update_count = int(expressions[1][1][-1])


# get count multiplied with And loop count

for i in lines:
    if "<" in i:
        start = i.find("<") + 1
        end = i.find(")")
        loop_count = int(i[start:end])

# define dummy variable
dummy_variable = f"t = {update_count}"

# use dummy variable for loop
dummy_in_loop = f"t = t + {update_count}"

loop_count_update = f"t<{loop_count * update_count}"

# replace where loop variable used
replacing_  = list(expressions.pop())
replacing_[1] = "t"

expressions.append(tuple(replacing_))


lines.insert(1, dummy_variable)
lines.insert(len(lines)-1, dummy_in_loop)

# update loop variable
```

```
line = lines[:]


for i, j in enumerate(lines):
    if "<" in j:
        j = list(j)
        start = j.index("<") - 1
        end = j.index(")")
        j[start: end] = list(loop_count_update)
        j = "".join(j)

        line[i] = j

    if "y" in j:
        j = list(j)
        start = j.index("=") + 1
        j[start:] = "t"
        j = "".join(j)
        line[i] = j

print("-------------------------------")
print("Updated code...")
for i in line:
    print(i)
```

Two Pass Assembler

MyCode.txt

```
PG1 START 0
    USING *,BASE
    L 1,FOUR
    A 1,FIVE
    A 1,=F'7'
    A 1,=D'8'
    ST 1,TEMP
FOUR DC F'4'
FIVE DC F'5'
BASE EQV 8
TEMP DC '1'D
```

```
    END
```

Main.py

```python
# import string
def readData(f):
    with open(f) as t:
        data = []
        da = t.read().split("\n")
        for line in da:
            if len(line) != 0:
                data.append(line.split())
    return data


def create_MDT_MNT(data):
    mdt = []
    mnt = []
    mdtc,mntc = 0,0
    flag = 0
    temp = None
    for i, j in enumerate(data):
        if flag == 1:
            if len(j) == 2 and temp != 1:
                j[1] = j[1].replace('&arg','#')
            mdt.append([mdtc,j])
            mdtc += 1

        if temp:
            mnt.append([mntc,j[0],mdtc-1])
            mntc += 1
            temp=0

        if j[0].lower() == 'macro':
            flag = temp = 1

        if j[0].lower() == 'mend':
            flag = 0
    return mdt, mnt
```

```python
def expand_Macro(data, mnt, mdt):
    output = []
    lines = []

    for i, j in enumerate(data):
        for k in mnt:
            if j[0] == k[1] and data[i-1][0] != 'MACRO':
                arg = j[1].split(',')
                ind = k[2]
                temp = mdt[ind:]
                for l in temp:
                    if l[1][0] == 'MEND':
                        break
                    if j[0] == k[1]:
                        if l[1][0] != j[0]:
                            aIndex = l[1][1].find('#')
                            aIndex = int(l[1][1][aIndex+1])-1
                            old = l[1][1].split(',')[1]
                            l[1] = [l[1][0],
l[1][1].replace(old,arg[aIndex])]
                            lines.append(l[1])
                            output.append(l[1])
        if len(lines):
            pass
        else:
            output.append(j)
        lines.clear()

    return output


def printData(data):
    for i, j in enumerate(data):
        print(i, end = "\t")
        for k in j:
            print(k, end = " ")
        print()
    return


if __name__ == '__main__':
    data = readData('MyCode.txt')
    mdt, mnt = create_MDT_MNT(data)
```

```python
print("\nMacro Program")
printData(data)
print("\nMDT")
for i in mdt:
    print(i)
print("\nMNT")
for i in mnt:
    print(i)
print()

output = expand_Macro(data, mnt, mdt)
print("\nExpanded Macro Program")
printData(output)
```