**ASSIGNMENT 1**

**Aim:** Generate Symbol table, Literal table, Pool table & Intermediate code of a two-pass Assembler for the given source code.

1d: Generate intermediate code from given assembly code

**Objective:**

1. Generating intermediate code from assembly program .

2, Understanding the working of two-pass Assembler.

**Theory:**

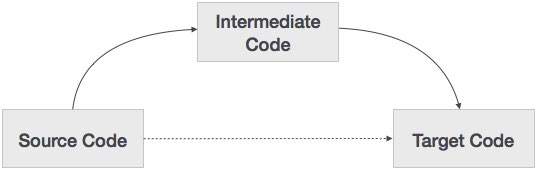
Intermediate code consists of a set of IC units, each unit consisting of the following three fields

- Address

- Representation of mnemonics opcode

- Representation of operands

A source code can directly be translated into its target machine code, then why at all we need to translate the source code into an intermediate code which is then translated to its target code? Let us see the reasons why we need an intermediate code.



* If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
* Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
* The second part of compiler, synthesis, is changed according to the target machine.
* It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.

Intermediate Representation

Intermediate codes can be represented in a variety of ways and they have their own benefits.

* **High Level IR** - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.
* **Low Level IR** - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations.

Intermediate code can be either language specific (e.g., Byte Code for Java) or language independent (three-address code).

**Code :**

import re      # regular expression library to detect re

import pandas as pd  # pandas to convert dict to

file = open('input.txt','r')

symbol = dict()

var = list()

literal = dict()

var1 = list()

LocCount = 0  # locationn counter to keep track of address

label = dict()

regx\_literal = re.compile(r'=[0-9]')   # re for literal

regx\_symbol = re.compile(r'=[A-Z]')    # re for symbol

for line in file:

    line.strip()     # remove white space from left and right

    words = line.split()    # split string into word

    if len(words)>3 :

        symbol[str(words[0])] = LocCount

    if line.startswith('ORIGIN'):

            no = words[1].split('+')

            if no[0] in symbol.keys():

                LocCount = symbol[no[0]]+int(no[1])

            continue

    if line.startswith('START'):            # if line starts with start initialize loc counter to corresponding value

        LocCount = int(words[-1])

        continue

    if regx\_symbol.search(line):            # if line contains symbol store it in symbol table

        var.append(words[-1])

        symbol[str(words[-1])] = 0

    if regx\_literal.search(line):           # if line contains literal store it in symbol table

        var1.append(words[-1])

        literal[str(words[-1])] = 0

    if 'DC' in line:        # if line contains DS or DC assign address to the symbols stored previously

        symbol[str(words[0])] = LocCount

    if 'DS' in line:

        symbol[str(words[0])] = LocCount

        LocCount += int(words[-1])

        continue

    if line.startswith('LTORG'):              # if line contains LTORG word assign address to all the literals stored previously

        for w in var1:

            literal[w] = LocCount

            LocCount += 1

        LocCount -= 1

    if line.startswith('END'):              # if line contains end word assign address to all the literals stored previously

        for w in var1:

            if literal.get(w)==0:

                literal[w] = LocCount

                LocCount += 1

    LocCount+=1

file.close()

file = open('input.txt','r')

op = open('output.txt','w')

optable = {'START':"('AD',1)",

           'END':"('AD',2)",

          'LTORG':"('AD',5)",

          'ORIGIN':"('AD',3)",

          'EQU':"('AD',4)",

          'DC':"('DL', 01)",

          'DS':"('DL', 02)",

          'ADD':"('IS', 01)",

          'SUB':"('IS', 02)",

          'MOVER':"('IS', 04)",

          'MOVEM':"('IS', 05)",

          'READ':"('IS', 09)",

          'PRINT':"('IS', 10)"

          }

register = ['AREG','BREG']

regx\_constant = re.compile('[0-9]+')

for line in file:

    line.strip()

    words = line.split()

    ic = ""

    if words[0] in optable.keys():

        if words[0]=='ORIGIN':

            ic+=optable[words[0]]+" "+words[1]

            op.write(ic+'\n')

            continue

        temp = optable[words[0]]

        ic+=temp+" "

    elif words[0] in symbol.keys():

        if words[1] in optable.keys():

            temp = optable[words[1]]

            ic+=temp+" "

    if "AREG" in words:

        ic+="AREG"+" "

    if "BREG" in words:

        ic+="BREG"+" "

    if words[-1] in symbol.keys():

        ic+=words[-1]+" "

    if words[-1] in literal.keys():

        ic+="(L,"+words[-1]+")"

    if words[-1][0] != "=":

        if regx\_constant.search(words[-1][0]):

            temp = "(C,"+words[-1]+")"

            ic+=temp+" "

    op.write(ic+'\n')

op.close()

file.close()

**Input :**

START 200

MOVERAREG =7

MOVER BREG X

L1 MOVER BREG =1

ORIGIN L1+3

LTORG

NEXT ADD AREG =2

X DS 1

END

**Output :**

('AD',1) (C,200)

('IS', 04) AREG (L,01)

('IS', 04) BREG X

('IS', 04) BREG (L,02)

('AD',3) L1+3

('AD',5)

('IS', 01) AREG (L,03)

('DL', 02) (C,1)

('AD',2)