FR. CONCEICAO RODRIGUES COLLEGE OF ENGINEERING

System Programming and Compiler Construction

VI Semester (Computer) Academic Year: 23-24

Experiment No 4

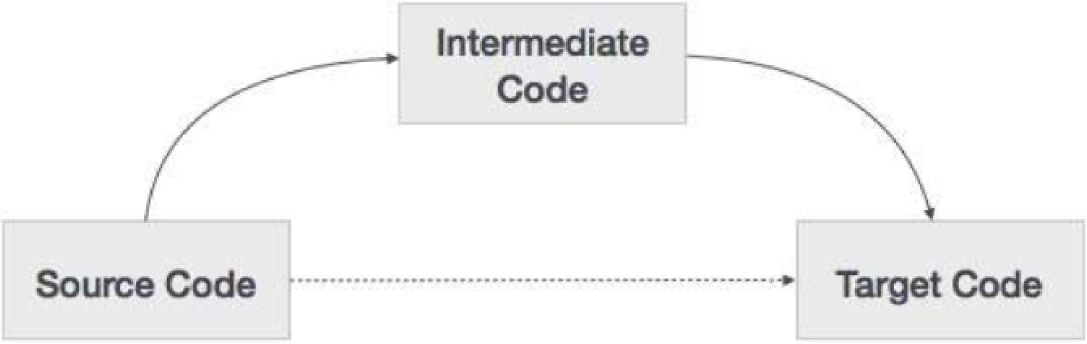
- Ivan Dsouza

- 9601

- T.E. Comps A (Batch C)

Aim : To generate an Intermediate code.

Description:



- 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.

- ## Three-Address Code

  Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code.
- For example:

```
a = b + c * d;
```

- The intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

- r being used as registers in the target program.
- A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms : quadruples and triples.

Quadruples

Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. The above example is represented below in quadruples format:

Op	arg$_1$	arg$_2$	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Triples

Each instruction in triples presentation has three fields : op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg$_1$	arg$_2$
*	c	d
+	b	(0)
+	(1)	(0)
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

Indirect Triples

This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Postlab Question

1. **Write the intermediate code generated for ---- while (a<b) do**

   If (c< d) then

   X= y+z

   Else

   X= y-z

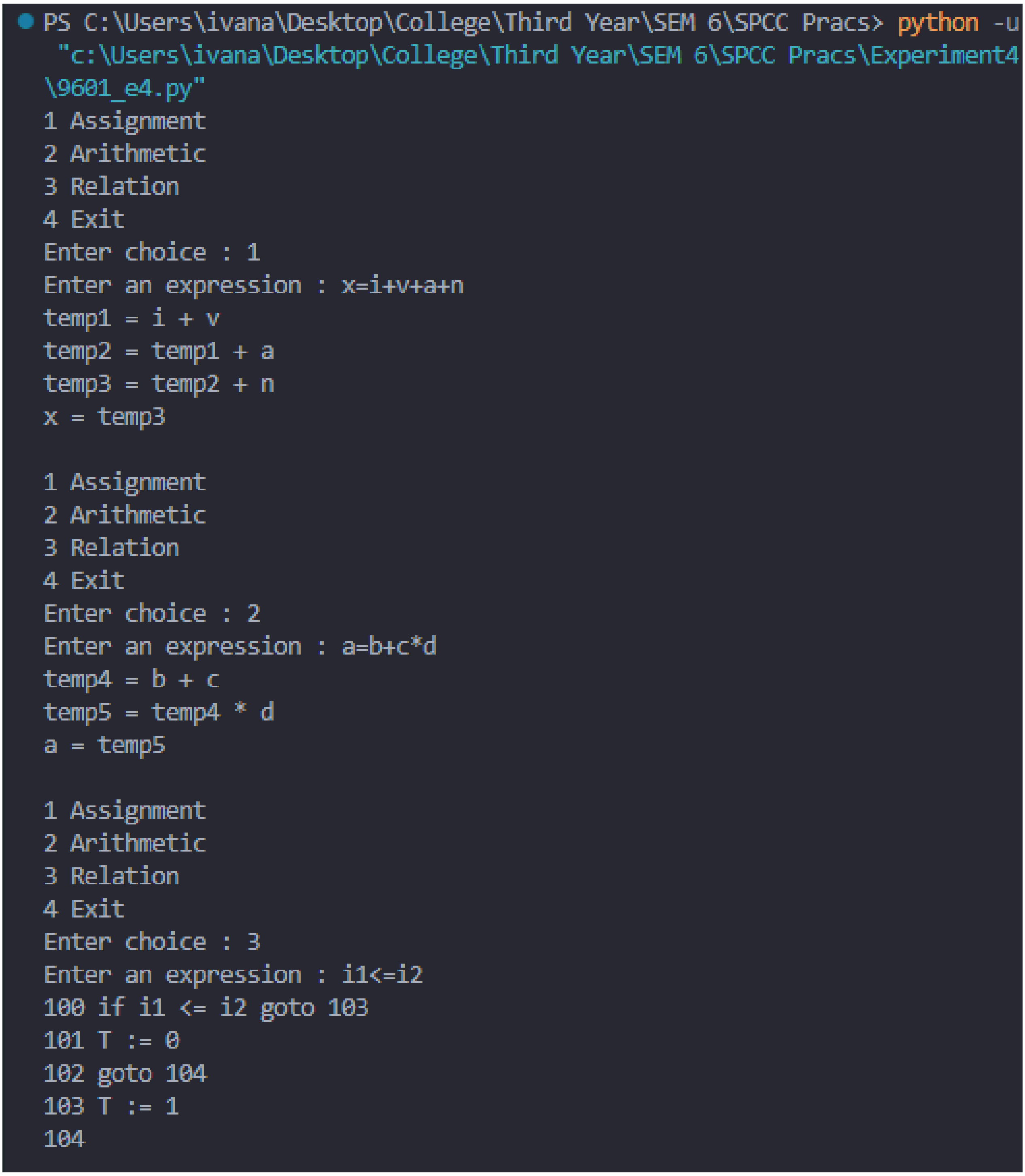
2. **Write the intermediate code generated for ---- switch E**
   Begin
   case V_1 : S_1
   case V_2 : S_2
   ....
   default: S_n
   end

Code:

```
import re


op = set("+-/*")
address = 100
count = 0


def arithematic(exp):
    global count
    symbols = []
```

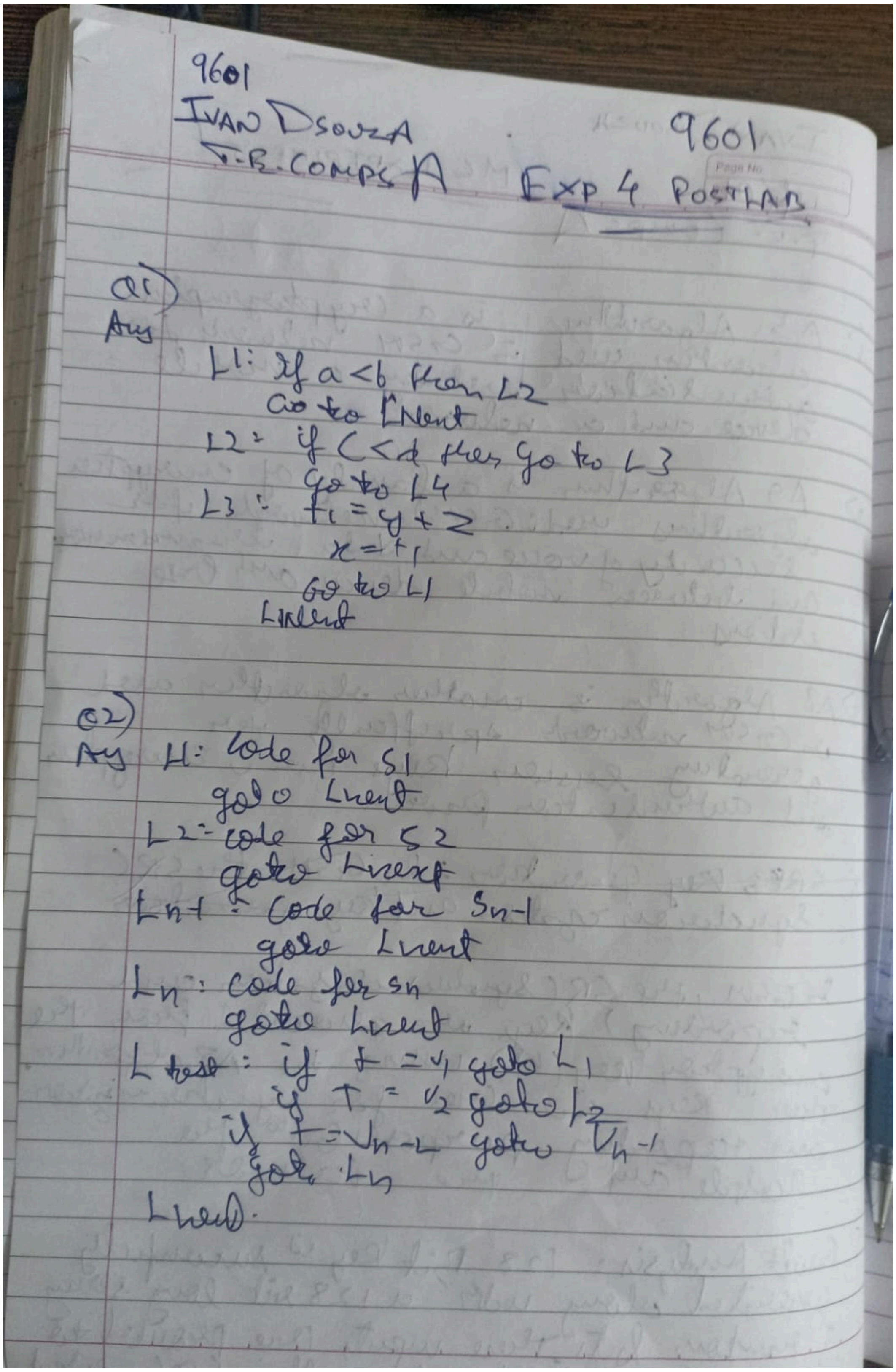
```
operators = []

for i in exp:
    if i in op:
        operators.append(i)
    else:
        symbols.insert(0, i)

if "=" in symbols:
    while True:
        s = symbols.pop()
        if s == "=":
            break
        symbols.insert(0, s)

for i in operators:
    count += 1
    e = "temp{0} = {1} {2} {3}".format(
        count, symbols.pop(), i, symbols.pop())
    symbols.append("temp{}".format(count))
    print(e)

if len(symbols) != 2:
    return

temp = symbols.pop()
print("{} = {}".format(symbols.pop(), temp))
```

```
def relation(exp):
    global address
    tokens = re.split(r">=|<=|==|>|<", exp)
    operators = re.findall(r">=|<=|==|>|<", exp)

    print("{0} if {2} {3} {1} goto {4}".format(
          address, tokens.pop(), tokens.pop(), operators.pop(), address +
3))
    print("{} T := 0 ".format(address + 1))
    print("{} goto {}".format(address+2, address+4))
    print("{} T := 1".format(address + 3))
    address += 4
    print(address)


if __name__ == "__main__":
    while True:
        option = input(
             "1 Assignment\n2 Arithmetic\n3 Relation\n4 Exit\nEnter choice
: ")
        if option == "1":
            exp = input("Enter an expression : ")
            arithematic(exp)
        if option == "2":
            exp = input("Enter an expression : ")
            arithematic(exp)
        if option == "3":
            exp = input("Enter an expression : ")
            relation(exp)
        if option == "4":
            break
        print()
```



```
PS C:\Users\ivana\Desktop\College\Third Year\SEM 6\SPCC Pracs> python -u
 "c:\Users\ivana\Desktop\College\Third Year\SEM 6\SPCC Pracs\Experiment4
\9601_e4.py"
1 Assignment
2 Arithmetic
3 Relation
4 Exit
Enter choice : 1
Enter an expression : x=i+v+a+n
temp1 = i + v
temp2 = temp1 + a
temp3 = temp2 + n
x = temp3

1 Assignment
2 Arithmetic
3 Relation
4 Exit
Enter choice : 2
Enter an expression : a=b+c*d
temp4 = b + c
temp5 = temp4 * d
a = temp5

1 Assignment
2 Arithmetic
3 Relation
4 Exit
Enter choice : 3
Enter an expression : i1<=i2
100 if i1 <= i2 goto 103
101 T := 0
102 goto 104
103 T := 1
104
```

Postlab



9601
IVAN DSOUZA
T.E. COMPS A

9601

EXP 4 POSTLAB

Q1)
Ans

```
L1: If a<b then L2
    Go to LNext
L2: if c<d then go to L3
    go to L4
L3: t1 = y+z
      x = t1
    Go to L1
  Lnext
```

Q2)
Ans

```
L1: code for S1
    goto Lnext
L2: code for S2
    goto Lnext
Ln-1: Code for Sn-1
    goto Lnext
Ln: code for Sn
    goto Lnext
L test: if t = v1 goto L1
        if t = v2 goto L2
     if t = vn-1 goto Ln-1
     goto Ln
Lnext.
```