Experiment 05 : Intermediate Code Generator

Learning Objective: Student should be able to Apply Intermediate Code Generator using 3-Address code.

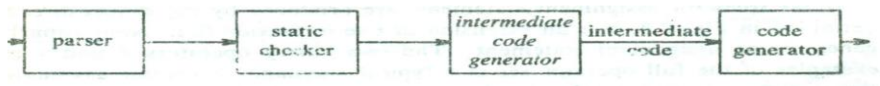
Tools: Jdk1.8,Turbo C/C++, Python, Notepad++

Theory:

Intermediate Code Generation:

In the analysis-synthesis model of a compiler, the front end translates a source program into an intermediate representation from which the back end generates target code. Details of the target language are confined to the backend, as far as possible. Although a source program can be translated directly into the target language, some benefits of using a machine-independent intermediate form are:

1. Retargeting is facilitated; a compiler for a different machine can be created by attaching a back end for the new machine to an existing front end.
2. A machine-independent code optimizer can be applied to the intermediate representation.



Position of intermediate code generator.

(Intermediate Languages) Intermediate Code Representation:

a) Syntax trees or DAG

b) postfix notation

c) Three address code

Three-Address Code:

Three-address code is a sequence of statements of the general form

x:= y op z

where x, y, and z are names, constants, or compiler generated temporaries; op stands for any operator, such as fixed-or floating-point arithmetic operator, or a logical operator on Boolean-valued data.

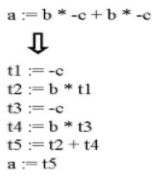
A source language expression like x+y+z might be translated into a sequence

t 1 := y*z

t 2:=x+t1

where t1 and t2 are compiler-generated temporary names.

Example: a: =b* -c +b * -c



```
a := b * -c + b * -c
   ⇩
t1 := -c
t2 := b * t1
t3 := -c
t4 := b * t3
t5 := t2 + t4
a := t5
```

Types of Three-Address Statements:

Three-address statements are akin to assembly code. Statements can have symbolic labels and there are statements for flow of control. A symbolic label represents the index of a three-address statement in the array holding intermediate code.

Actual indices can be substituted for the labels either by making a separate pass, or by using "back patching,"

Implementations of Three-Address Statements

A three-address statement' is an abstract form of intermediate code. In a compiler, these statements can be implemented as records with fields for the operator and the operands. Three such representations are quadruples, triples, and indirect triples.

Quadruples:

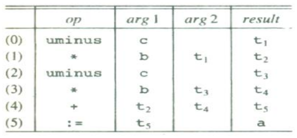
a) A quadruple is a record structure with four fields:
op, arg 1, arg 2, and result.

b) The op field contains an internal code for the operator.
c) The three-address statement x : = y op z is represented by placing y in arg1, Z in arg 2, and x in result.
d) Statements with unary operators like x : = -y or x : = y do not use arg 2. Operators like param use neither arg 2 nor result.
e) Conditional and unconditional jumps put the target label in result.
f) The quadruples are for the assignment a : = b * - c + b* - c.
They are obtained from the three-address code in Fig. (a).

g) The contents of fieldsarg I, arg 2, and result are normally pointers to thesymbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as they are created.

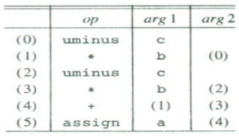
Triples:

a) To avoid entering temporary names into the symbol table, refer to a temporary value by the position of the statement that computes it.
b) Three-address statements can be represented by records with only three fields: op, arg1 and arg2, as in Fig.(b).
c) The fieldsarg1 and arg2, for the arguments of op, are pointer to the symbol table. Since three fields are used, this intermediate code format is known as triples.



	op	arg 1	arg 2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

(a) Quadruples

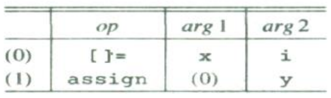


	op	arg 1	arg 2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

(b) Triples

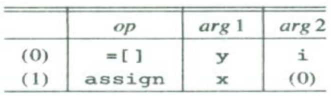
Quadruple and triple representations of three-address statements.

A ternary operation like x[i] := y requires two entries in the triple structure, as shown in Fig.(a), while x : = y[i] is naturally represented as two operations in Fig. (b).



	op	arg 1	arg 2
(0)	[ ]=	x	i
(1)	assign	(0)	y

(a) x[i] := y



	op	arg 1	arg 2
(0)	=[ ]	y	i
(1)	assign	x	(0)

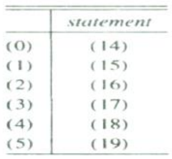
(b) x := y[i]

More triple representations.

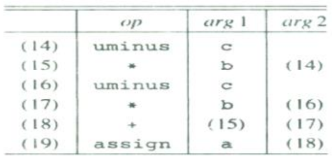
Indirect Triples:

Another implementation of three-address code that has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called indirect triples.

For example, let us use an array statement to list pointers to triples in the desired order.



	statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)



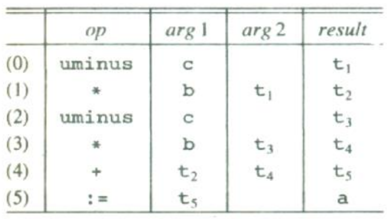
	op	arg 1	arg 2
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect triples representation of three-address statements

Input:

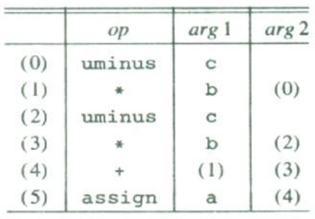
a : = b * - c + b* - c.

Output:



	op	arg 1	arg 2	result
(0)	uminus	c		t_1
(1)	*	b	t_1	t_2
(2)	uminus	c		t_3
(3)	*	b	t_3	t_4
(4)	+	t_2	t_4	t_5
(5)	:=	t_5		a

(a) Quadruples



	op	arg 1	arg 2
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

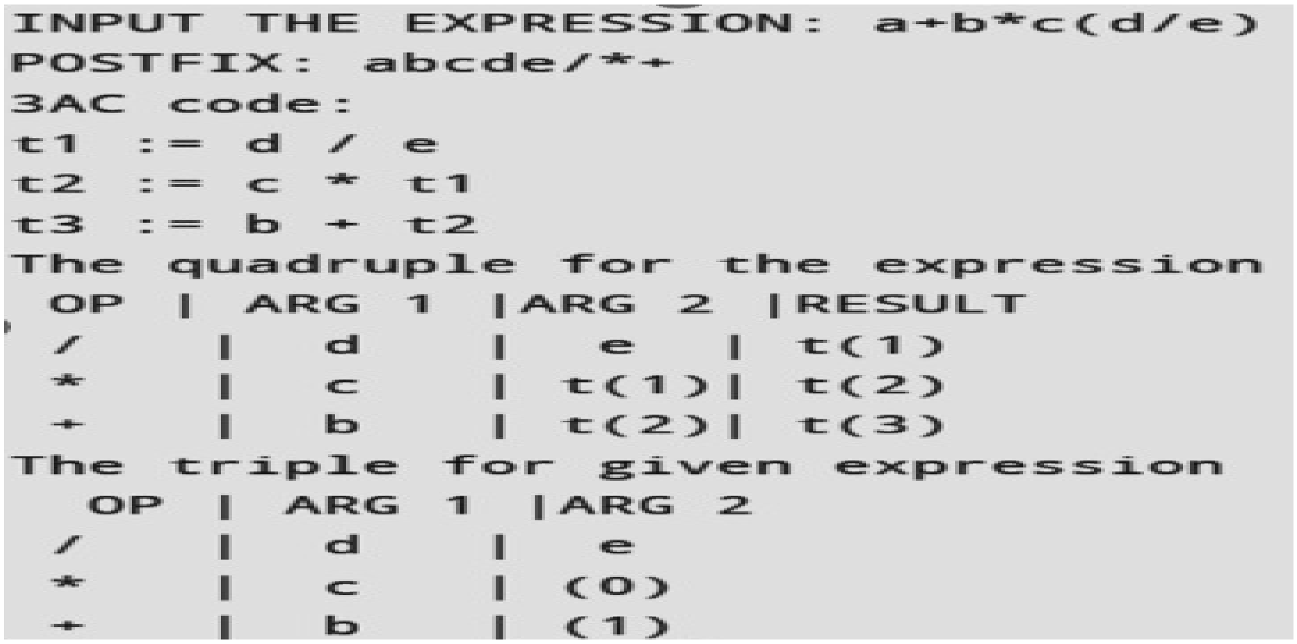
(b) Triples

Application: Intermediate code can be easily produced to the target code.

Design:

```
OPERATORS = set(['+', '-', '*', '/',
'(', ')'])
PRI = {'+':1, '-':1, '*':2, '/':2}

def infix_to_postfix(formula):
    stack = []
    output = ''
    for ch in formula:
        if ch not in OPERATORS:
            output += ch
        elif ch == '(':
            stack.append('(')
        elif ch == ')':
            while stack and stack[-
1] != '(':
                output              +=
stack.pop()
            stack.pop()
        else:
            while stack and stack[-
1] != '(' and PRI[ch] <= PRI[stack[-
1]]:
                output              +=
stack.pop()
            stack.append(ch)

    while stack:
        output += stack.pop()
    print(f'POSTFIX: {output}')
    return output

def generate3AC(pos):
    print("3AC code: ")
    exp_stack = []
    t = 1
    for i in pos:
        if i not in OPERATORS:
            exp_stack.append(i)
        else:
            print(f't{t}          :=
{exp_stack[-2]}   {i}   {exp_stack[-
1]}')
            exp_stack=exp_stack[:-
2]
exp_stack.append(f't{t}')
            t+=1

expres     =     input("INPUT     THE
EXPRESSION: ")
pos = infix_to_postfix(expres)
generate3AC(pos)

def Quadruple(pos):
    stack = []
    op = []
    x = 1
    for i in pos:
        if i not in OPERATORS:
            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()
            if stack != []:
                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
    print("The  quadruple  for  the
expression ")
    print(" OP | ARG 1 |ARG 2 |RESULT
")

def Triple(pos):
    stack = []
    op = []
    x = 0
    for i in pos:
        if i not in OPERATORS:
            stack.append(i)
        elif i == '-':
            op1 = stack.pop()
            stack.append("(%s)" %x)
            print("{0:^4s} | {1:^4s}
| {2:^4s}".format(i,op1,"(-)"))
            x = x+1
        if stack != []:
            op2 = stack.pop()
            op1 = stack.pop()
            print("{0:^4s} | {1:^4s}
| {2:^4s}".format("+",op1,op2))
            stack.append("(%s)" %x)
            x = x+1
        elif i == '=':
            op2 = stack.pop()
            op1 = stack.pop()
            print("{0:^4s} | {1:^4s}
| {2:^4s}".format(i,op1,op2))
        else:
            op1 = stack.pop()
            if stack != []:
                op2 = stack.pop()
                print("{0:^4s}     |
{1:^4s}                          |
{2:^4s}".format(i,op2,op1))
                stack.append("(%s)"
%x)
                x = x+1
    print("The  triple  for  given
expression")
    print(" OP | ARG 1 |ARG 2 ")

# Call the functions
Quadruple(pos)
Triple(pos)
```

OUTPUT:



```
INPUT THE EXPRESSION: a+b*c(d/e)
POSTFIX: abcde/*+
3AC code:
t1 := d / e
t2 := c * t1
t3 := b + t2
The quadruple for the expression
 OP | ARG 1 |ARG 2 |RESULT
 /    |  d    |  e   | t(1)
 *    |  c    | t(1)| t(2)
 +    |  b    | t(2)| t(3)
The triple for given expression
  OP | ARG 1 |ARG 2
 /    |  d    |  e
 *    |  c    | (0)
 +    |  b    | (1)
```

Result and Discussion:

Learning Outcomes: The student should have the ability to

LO1 **Define** the role of Intermediate Code Generator in Compiler design.
LO2: ***Describe*** the various ways to implement Intermediate Code Generator.
LO3: ***Specify*** the formats of 3 Address Code.
LO4: Illustrate the working of Intermediate Code Generator using 3-Address code
.

Course Outcomes: Upon completion of the course students will be able to Evaluate the synthesis phase to produce object code optimized in terms of high execution speed and less memory usage.

Conclusion:

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**
Marks Obtained			