SRM Institute of Science and Technology
College of Engineering and Technology
SCHOOL OF COMPUTING

SET-B

SRM Nagar, Kattankulathur – 603203, Chengalpattu District, Tamilnadu

Academic Year: 2022-23 (EVEN)

Test: CLAT-3 **Date: 04.5.2023**
Course Code & Title: 18CSC304J COMPILER DESIGN **Duration: 2 periods**
Year & Sem: III & V **Max. Marks: 50**

Course Articulation Matrix:

S.No.	Course Outcome	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8	PO 9	PO10	PO11	PO12
1	CO4	H	H	H	H	M	L	L	L	M	M	L	H
2	CO5	H	H	H	H	M	L	L	L	M	M	L	H
3	CO6	H	H	H	H	M	L	L	L	M	M	L	H

Part – A (10 x 1 = 10 Marks) Instructions: Answer all						
Q. No	**Question**	**Marks**	**BL**	**CO**	**PO**	**PI Code**
1	Three address statement has (i) Maximum of 3 references among that 2 for operands and one for result (ii) Exactly 3 references and all the 3 for operands only (ii) Exactly 3 references among that 2 for operands and one for result (iv) Minimum of 3 references among that 2 for operands and one for result	**1**	**1**	**4**	**1**	**1.1.3**
2	Syntax Directed Translations are (i) The other representation of context-free grammars for specifying translations for programming language constructs. (ii) Context-free grammar symbols are associated with set of Attributes (iii) Context-free grammar productions are associated with Semantic Rules (iv) All of the above	**1**	**1**	**4**	**1**	**1.1.3**
3	Intermediate code tends to be (i) Machine-independent code (ii) Machine-dependent code (iii) Both machine-independent and machine-dependent code (iv) Machine code	**1**	**1**	**4**	**1**	**1.1.1**
4	It enables the optimizers to liberally re-position the ub-expression to produce an optimized code.	**1**	**1**	**5**	**1**	**1.1.1**

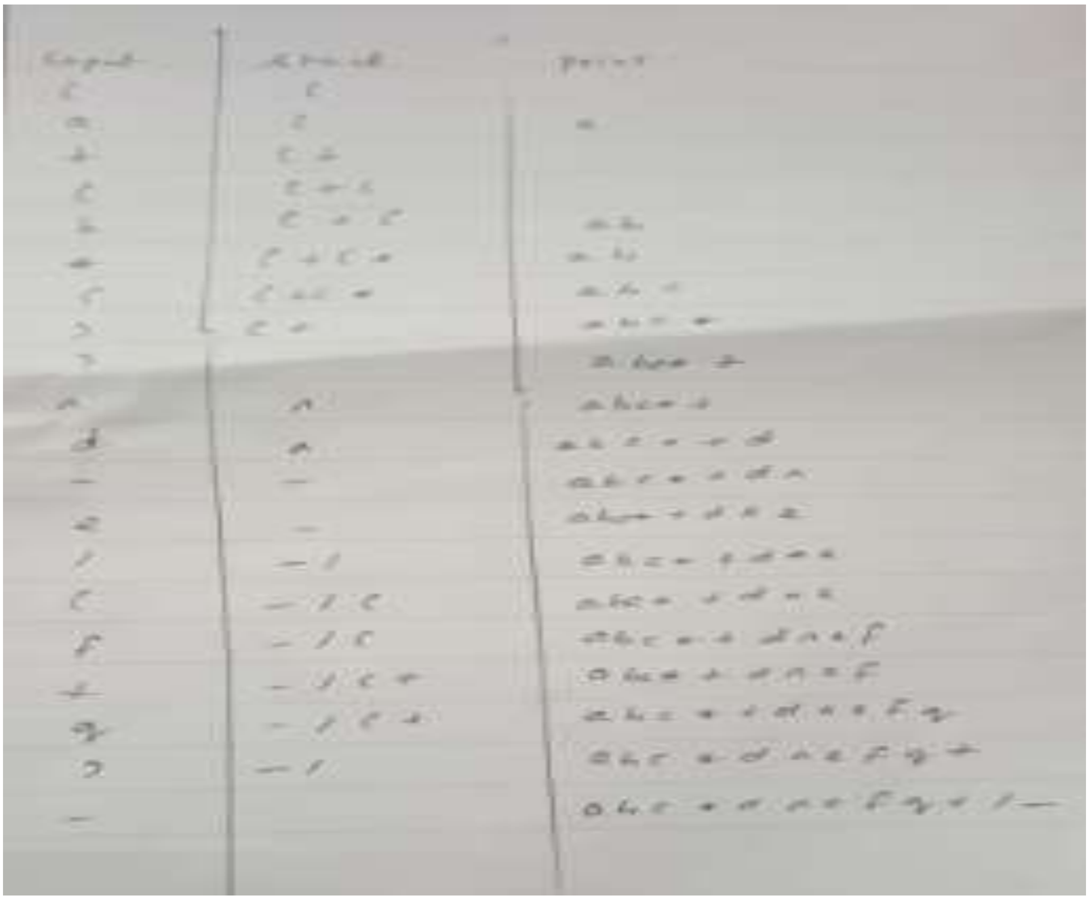
	(i) Quadruples (ii) Triples (iii) Indirect Triples (iv) Quadriples					
5	Semantic rules in a S-Attributed Definition can be evaluated by a (i) Bottom-up order (ii) PostOrder traversal (iii) InOrder traversal (iv) Either (i) or (ii)	**1**	**1**	**5**	**1**	**1.1.1**
6	The evaluation order of Synthesized attributes and inherited attribute are ----------- and ----------- (i) In-Order and Pre-Order (ii) Pre-Order and Post-Order (iii) Bottom-up order and Post-Order (iv) Post-Order and Pre-Order	**1**	**2**	**4**	**1**	**1.1.3**
7	---------------- Keeps track of location where current value of the name can be found and ------------- informs the availability of registers to the code generator. (i) Register descriptor and address descriptor (ii) Address descriptor and register descriptor (iii) Register tracker and address descriptor (iv) Address descriptor and register tracker	**1**	**1**	**6**	**1**	**1.1.1**
8	Peep-hole optimization is a form of a) loop optimization b) local optimization c) constant folding d) data flow analysis	**1**	**1**	**6**	**1**	**1.1.1**
9	Substitution of values for names whose values are constant, is done in a) local optimization b) loop optimization c) constant folding d) none of these	**1**	**1**	**5**	**1**	**1.1.1**
10	Local and loop optimization in turn provide motivation for a) data flow analysis b) constant folding	**1**	**1**	**5**	**1**	**1.1.1**

	c) peep hole optimization d) DFA and constant folding					

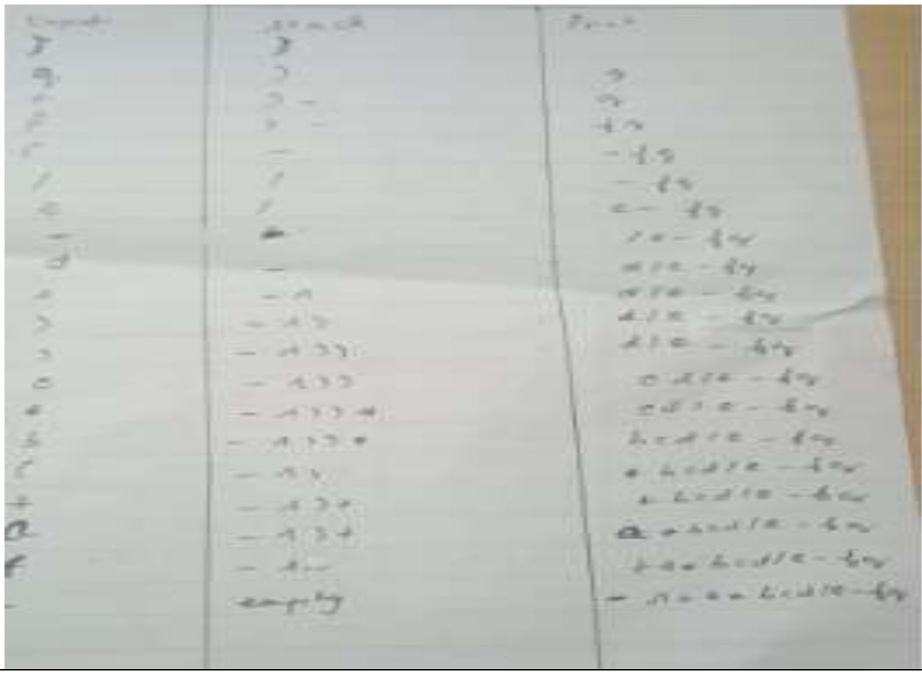
Part – B (4 x 4 = 16 Marks) Instructions: Answer FOUR

11 Find the polish and reverse polish notation using stack method for the following expression (a+(b*c))^d-e/(f+q)

Reverse Polish Notation or POSTFIX



Polish Notation or PREFIX



12	Consider the following pseudo code if (a>b) then x=a+b else x =a-b , write the quadruple, triple and indirect triple.					

```
100 : if a > b then  goto 102
101 : goto 105
102 : t1:=a+b
103 : x:= t1
104 : goto 108
105 : t1:=a–b
106 : x:= t1
107 : goto 108
108 : ...
```

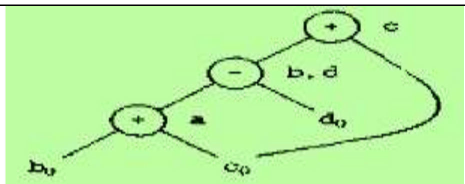
	Operator	Operand 1	Operand 2	Result
100	>	a	b	goto 102
101		goto 105		
102	+	a	b	t1
103	:=	t1		x
104		goto 108		
105	–	a	b	t1
106	:=	t1		x
107		goto 108		

13	Draw the DAG by converting the following expression into three address code. a=b+e b=c[i]+d[j]; c= a+b a= a+b*c-(a+b)	**4**	**4**	**5**	**3**	**2.2.3**
14	Write the comparison among Static allocation, Stack allocation and Heap Allocation with their merits and limitations.					

Sr.No.	Static allocation	Stack allocation	Heap allocation
1.	Static allocation is done for all data objects at compile time.	In stack allocation, stack is used to manage runtime storage.	In heap allocation, heap is used to manage dynamic memory allocation.
2.	Data structures can not be created dynamically because in static allocation compiler can determine the amount of storage required by each data object.	Data structures and data objects can be created dynamically.	Data structures and data objects can be created dynamically.
3.	**Memory allocation :** The names of data objects are bound to storage at compile time.	**Memory allocation :** Using Last In First Out (LIFO) activation records and data objects are pushed onto the stack. The memory addressing can be done using index and registers.	**Memory allocation :** A contiguous block of memory from heap is allocated for activation record or data object. A linked list is maintained for free blocks.
4.	**Merits and limitations :** This allocation strategy is simple to implement but supports static allocation only. Similarly recursive procedures are not supported by static allocation strategy.	**Merits and limitations :** It supports dynamic memory allocation but it is slower than static allocation strategy. Supports recursive procedures but references to non local variables after activation record can not be retained.	**Merits and limitations :** Efficient memory management is done using linked list. The deallocated space can be reused. But since memory block is allocated using best fit, holes may get introduced in the memory.

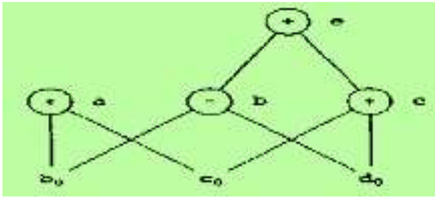
15 Discuss in detail about optimization of basic blocks

- A number of code-improving transformations such as structure-preserving transformations, dead-code elimination and algebraic transformations can be applied for basic blocks
- Many of the structure-preserving transformations can be implemented by constructing a DAG for a basic block
- Consider the block.   The DAG for the block is
  a := b + c
  b := a – d
  c := b + c
  d := a – d



- Consider the block. The DAG for the block is

  a := b + c
  b := b – d
  c := c + d
  e := b + c



The use of Algebraic Identities

Algebraic identities represent an important class of optimizations on basic blocks

1. x+0 = 0+x = x
   x-0 = x
   x*1 = 1*x = x
   x/1 = x

2. Reduction in strength
   x**2 = x*x
   2.0*x = x+x
   x/2 = x*0.5

3. Constant folding: Constant expressions are evaluated at compile time and the constant expressions are replaced by their values

4. Commutativity
   x*y = y*x

5. Associativity
   (x+y)+z = x+(y+z)

Sometimes, associative laws may also be applied to expose common sub expressions

Example:

Source Code

a := b + c
e := c + d + b

Intermediate Code

a := b + c
t := c + d
e := t + b

OR

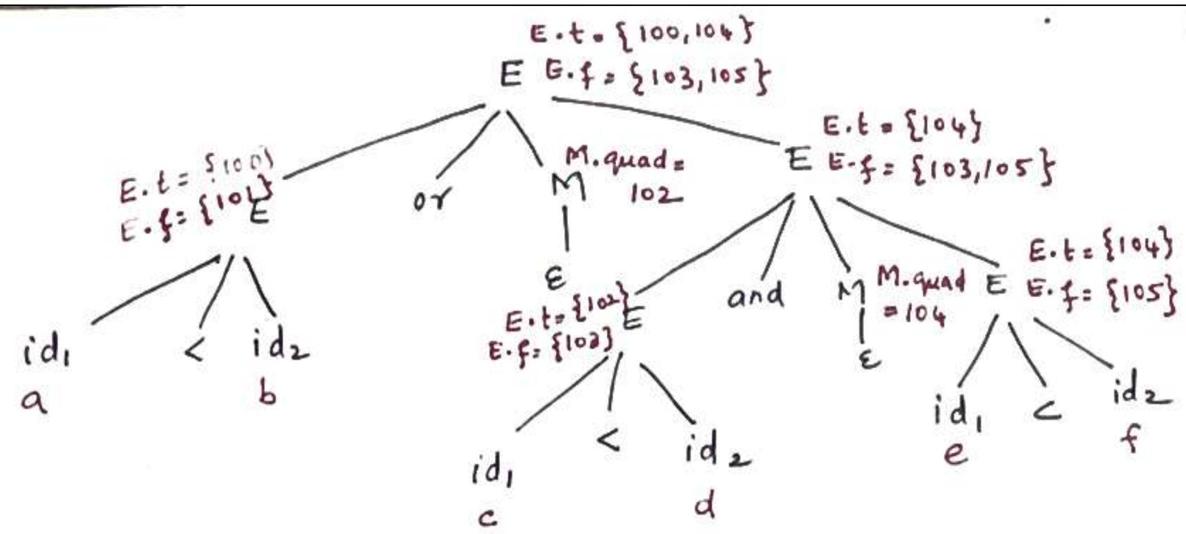
a := b + c
e := a + d

Part – C (2 x12 = 24 Marks)

16 Consider the following expression

a<b or c<d and e<f

How will you generate the three address (4 marks) code by forming annotated parse tree (4 marks) using the translation scheme with backtracking.



Three Address Code

100 : if a<b goto 103

101 : t1=0

102 : goto 104

103 : t1=1

104 : if c<d goto 107

105 : t2=0

106 : goto 108

107 : t1=1

108 : if e<f goto 111

109 : t3=0

110 : goto 112

111 : t3=1

112 : t4=t2 and t3

113 : t5=t1 or t4

ii) Describe the sematic rules for translating Boolean Expressions (4 marks)

(1)	$E \rightarrow E_1$ **or** $M\ E_2$	{ *backpatch* (E_1.*falselist*, *M.quad*); *E.truelist* := *merge*(E_1.*truelist*, E_2.*truelist*); *E.falselist* := E_2.*falselist* }
(2)	$E \rightarrow E_1$ **and** $M\ E_2$	{ *backpatch* (E_1.*truelist*, *M.quad*); *E.truelist* := E_2.*truelist*; *E.falselist* := *merge*(E_1.*falselist*, E_2.*falselist*) }
(3)	$E \rightarrow$ **not** E_1	{ *E.truelist* := E_1.*falselist*; *E.falselist* := E_1.*truelist* }
(4)	$E \rightarrow$ (E_1)	{ *E.truelist* := E_1.*truelist*; *E.falselist* := E_1.*falselist* }
(5)	$E \rightarrow$ **id**$_1$ **relop id**$_2$	{ *E.truelist* := *makelist*(*nextquad*); *E.falselist* := *makelist*(*nextquad* + 1); *emit*('if' **id**$_1$.*place* **relop**.*op* **id**$_2$.*place* 'goto _') *emit*('goto _') }
(6)	$E \rightarrow$ **true**	{ *E.truelist* := *makelist*(*nextquad*); *emit*('goto _') }
(7)	$E \rightarrow$ **false**	{ *E.falselist* := *makelist*(*nextquad*); *emit*('goto _') }
(8)	$M \rightarrow \epsilon$	{ *M.quad* := *nextquad* }

OR

17 Write the three address code (4 marks) for the following and find the quadruple (3 marks) ,triple(3 marks) and indirect triple (2 marks)

i)

Switch (a+b)
{
case 1:x-x+1:
case 2: y-y+2
case 3: +3 default-1;
}

t=a+b

if t = 1 goto L1

if t = 2 goto L2

if t = 3 goto L3

L1:

T1 = x-x

T2 = T1+1

L2:

T3 = y-y

T4 = T3+2

L3:

T4 = z+3

QUADRUPLE

Location	OP	Arg1	Arg2	Result
(1)	+	a	b	t
(2)	=	t	1	(3) GOTO L1
(3)	-	x	X	T1
(4)	+	T1	1	T2
(5)	=	t	2	(6) GO TO L2
(6)	-	y	y	T3
(7)	+	T3	2	T4
(8)	=	t	3	(9) GOTO L3
(9)	+	z	3	T3

Triples

Location	OP	Arg1	Arg2
(1)	+	a	b
(2)	=	t	1
(3)	-	x	x
(4)	+	(3)	1
(5)	=	t	2
(6)	-	y	y
(7)	+	(6)	2
(8)	=	t	3
(9)	+	z	3

Indirect triples

Statement	
(31)	(1)
(32)	(2)

(33)	(3)
(34)	(4)
(35)	(5)
(36)	(6)
(37)	(7)
(38)	(8)
(39)	(9)

ii) int sum = 0;
for (int i = 1; i <= n; i++) { sum += i*i; }

1. sum = 0;
2. i = 1;
3. if (i >n) goto 9
4. t1 = i * i;
5. sum = sum+t1;
6. t2 = i + 1;
7. i = t2;
8. goto(3)
9. goto calling program

Quadruple

Location	OP	Arg1	Arg2	Result
(1)	=	Sum	0	
(2)	=	i	1	
(3)	>	i	n	(4)
(4)	*	i	1	T1
(5)	+	Sum	T1	Sum
(6)	+	i	1	T2
(7)	=	i	T2	
(8)	JMP	(3)		

Triple

Location	OP	Arg1	Arg2
(1)	=	Sum	0
(2)	=	i	1
(3)	>	i	n
(4)	*	i	1
(5)	+	sum	(4)
(6)	+	i	1
(7)	=	i	(6)
(8)	JMP	(3)	

Indirect triples

Statement	
(31)	(1)
(32)	(2)
(33)	(3)
(34)	(4)
(35)	(5)
(36)	(6)
(37)	(7)
(38)	(8)
(39)	(9)
(40)	(10)

Location	OP	Arg1	Arg2
(1)	=	Sum	0
(2)	=	i	1
(3)	>	i	n

(4)	*	i	1
(5)	+	sum	(4)
(6)	+	i	1
(7)	=	i	(6)
(8)	JMP	(3)	

Perform the following optimization techniques for the quick sort
a. Dead code elimination
b. Variable elimination
c. Code motion
d. Reduction in strength

Sol:

Three address code for quick sort (2 marks)

```
(1)   i = m-1                 (16)  t7 = 4*i
(2)   j = n                   (17)  t8 = 4*j
(3)   t1 = 4*n                (18)  t9 = a[t8]
(4)   v = a[t1]               (19)  a[t7] = t9
(5)   i = i+1                 (20)  t10 = 4*j
(6)   t2 = 4*i                (21)  a[t10] = x
(7)   t3 = a[t2]              (22)  goto (5)
(8)   if t3<v goto (5)        (23)  t11 = 4*i
(9)   j = j-1                 (24)  x  = a[t11]
(10)  t4 = 4*j                (25)  t12 = 4*i
(11)  t5 = a[t4]              (26)  t13 = 4*n
(12)  if t5>v goto (9)        (27)  t14 = a[t13]
(13)  if i>=j goto (23)       (28)  a[t12] = t14
(14)  t6 = 4*i                (29)  t15 = 4*n
(15)  x = a[t6]               (30)  a[t15] = x
```

common sub expression elimination:

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```
B_5

(a) Before.

```
t6 = 4*i
x = a[t6]
t8 = 4*j
t9 = a[t8]
a[t6] = t9
a[t8] = x
goto B2
```
B_5

(b) After.

Dead code elimination: (2 marks)

A variable is *live* at a point in a program if its value can be used subsequently; otherwise, it is *dead* at that point. A related idea is *dead* (or *useless*) *code* — statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.

```
if (debug) print ...
```

It may be possible for the compiler to deduce that each time the program reaches this statement, the value of debug is FALSE. Usually, it is because there is one particular statement

```
debug = FALSE
```

that must be the last assignment to debug prior to any tests of the value of debug, no matter what sequence of branches the program actually takes. If copy propagation replaces debug by FALSE, then the print statement is dead because it cannot be reached. We can eliminate both the test and the print operation from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as *constant folding*. □

Induction-variable elimination (2marks)

•Any two variables are said to be induction variables ,if there is a change in any one of the variable, then there is a corresponding change in the other variable.

Code motion: (2 marks)

It moves code outside the loop • Thus transformation takes an expression that yields the same result independent of the number of times a loop is executed and places the expression before the loop.

Example Consider the stmt:

while(i<=limit-2)

Code motion : t :=limit-2; while(i<=t)

Reduction in strength (2 marks)

The replacement of an expensive operation by a cheaper one. • Example : • step t2 :=4*i; in B2 • Replaced with t2 :=t2+4; • This replacement will speed up the object code ,if addition takes less time than multiplication

OR

19 Consider the following program code:

```
prod=0;
i=1;
do{
prod=prod+a[i]*b[i];
i=i+1;
}while (i<=10);
```

i) Partition a sequence of three-address statements into basic blocks by finding the leader and write the rules (6 marks)
ii). Perform the Transformation on Basic Blocks (6 marks)

I) Three address code for the given code is- (6 marks)

prod = 0

i = 1

T1 = 4 x i

T2 = a[T1]

T3 = 4 x i

T4 = b[T3]

T5 = T2 x T4

T6 = T5 + prod

prod = T6

T7 = i + 1

i = T7

if (i <= 10) goto (3)

Step-01:

We identify the leader statements as-

II) prod = 0 is a leader because first statement is a leader.

III) T1 = 4 x i is a leader because target of conditional or unconditional goto is a leader.

Step-02:

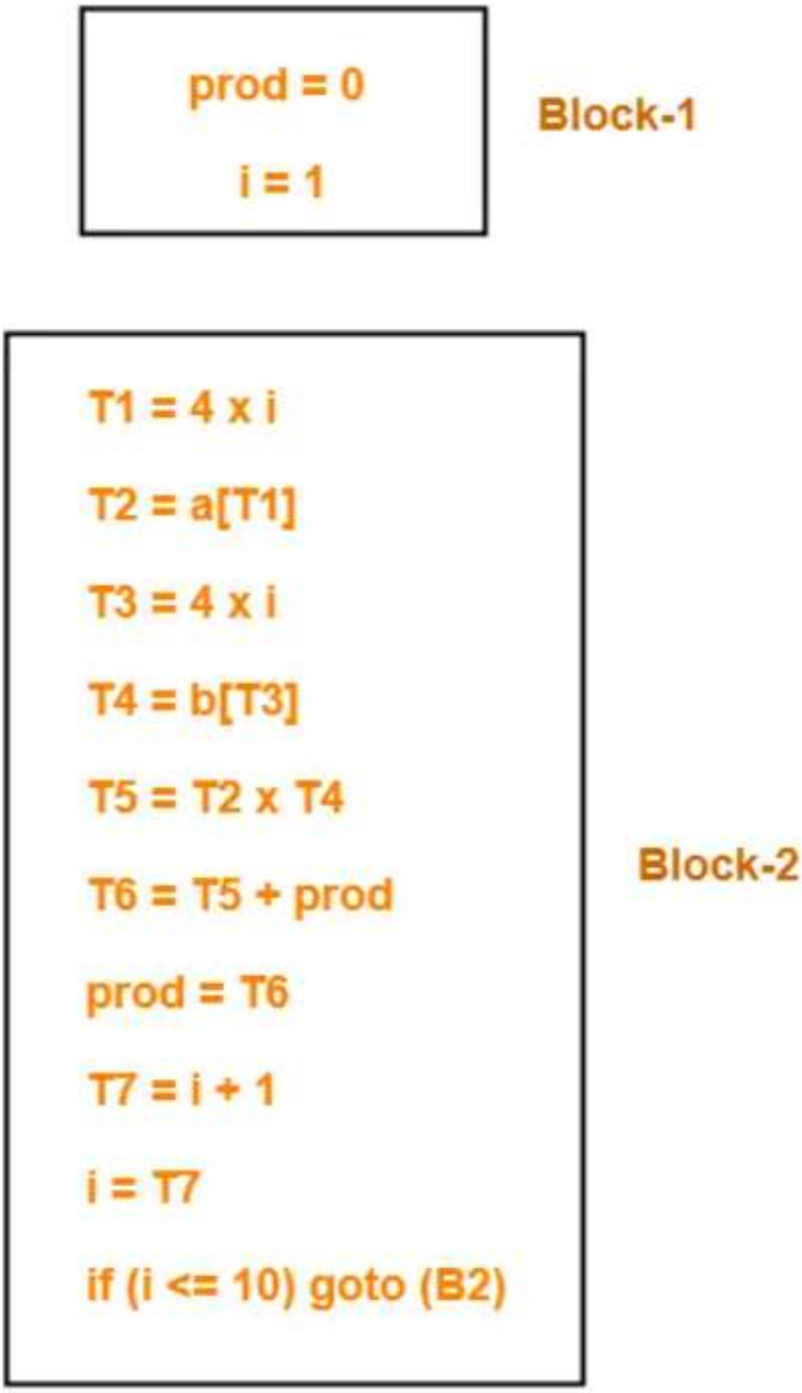
The above generated three address code can be partitioned into 2 basic blocks as-

IV)Transformation on Basic Blocks

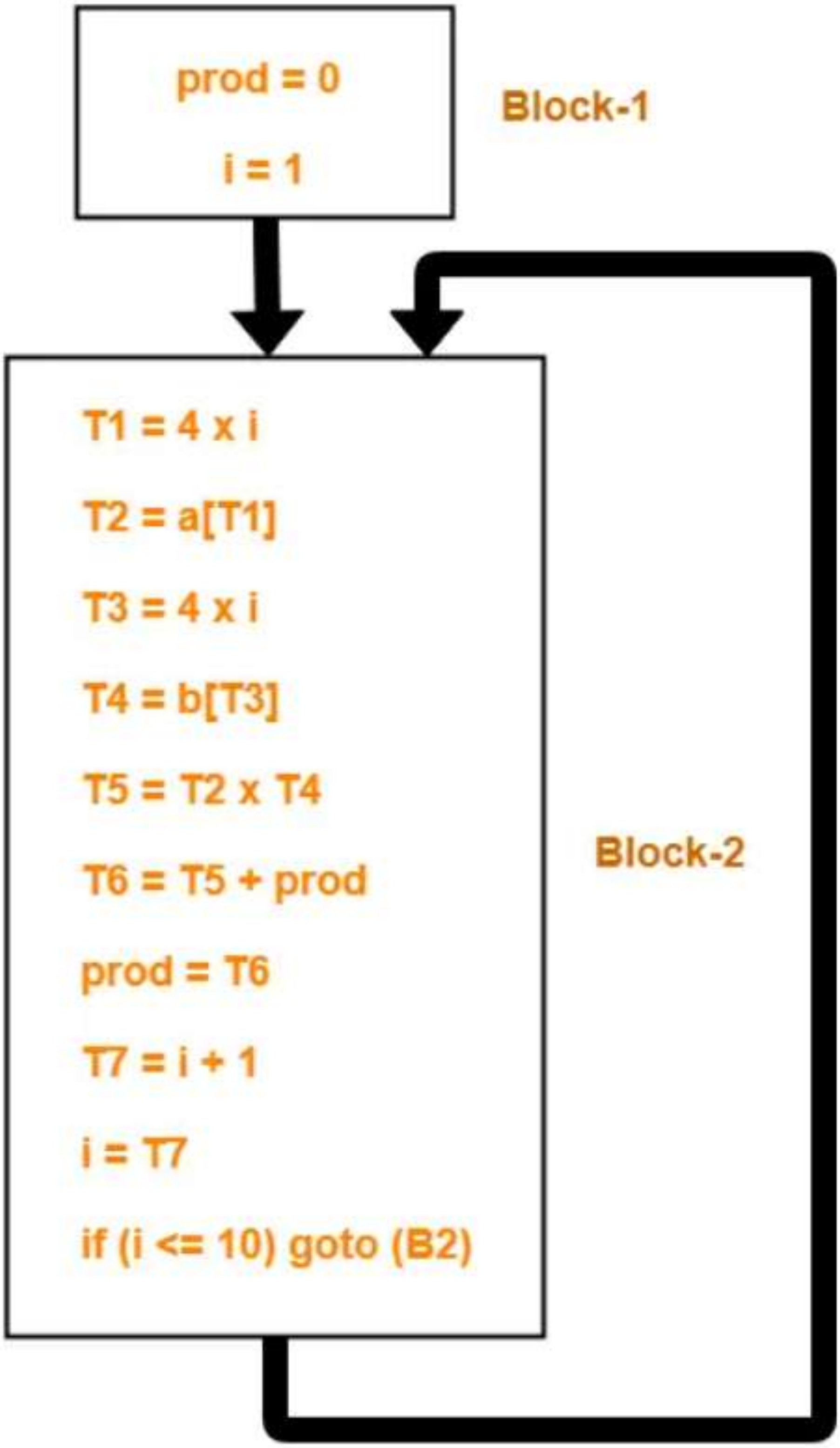
There are two types of transformations:

Structure-preserving transformations

Algebraic transformation



Basic Blocks


prod = 0
i = 1
Block-1
T1 = 4 x i
T2 = a[T1]
T3 = 4 x i
T4 = b[T3]
T5 = T2 x T4
T6 = T5 + prod
prod = T6
T7 = i + 1
i = T7
if (i <= 10) goto (B2)
Block-2

Flow Graph