Vallurupalli Nageswara Rao Vignana Jyothi Institute of Engineering &Technology

ESTD:1995

Department of Computer Science & Engineering

SUBJECT: *Automata and Compiler Design*

Subject Code:*22PC1CB303*

Topic Name: Phases of a compiler
III year-I sem, sec: B and D

Dr. M.Gangappa
Associate Professor
Email: gangappa_m@vnrvjiet

<Web link of your created resource if any>

Syllabus

- **Run Time Environments**: Storage organization, storage allocation strategies, access to non-local names, language facilities for dynamics storage allocation.
- **Code Optimization**: Principal sources of optimization, Optimization of basic blocks, peephole optimization, flow graphs, optimization techniques.

Agenda

ESTD:1995

Basic terminologies

- Basic blocks and flow graphs
- Partition algorithm for flow graphs

Unit – 5

- Code optimization - introduction
- Criteria for code optimization(properties of code optimization)
- Principle sources of code optimization(function preserving or structure preserving)
- Loop optimization
- DAG based optimization
- Control flow and data flow analysis

Run time environment

The run time environment is the structure of a target machine which is used to manage memory and maintain information to guide the programs execution process.

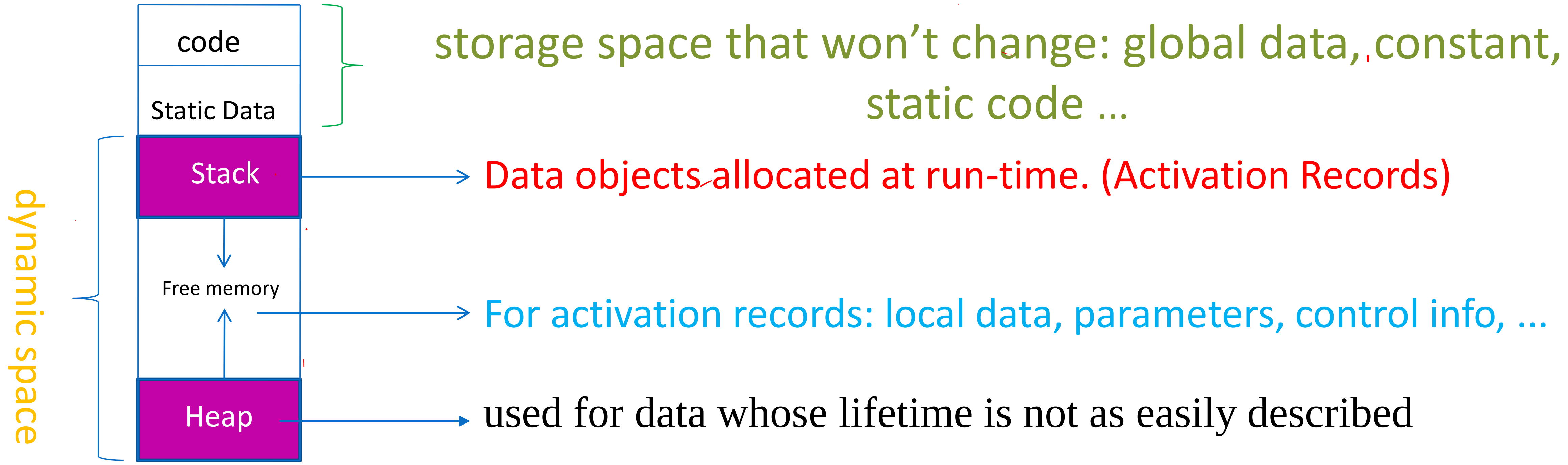
Run-Time Storage Organization

The run time memory organization is a layout in memory of an executable program .

Run-time storage organization

Run-time storage can be subdivide to hold the different components of an executing program:

- ❑Generated code
- ❑Dynamic data-object- heap
- ❑Static data objects
- ❑Automatic data objects- stack



storage space that won’t change: global data, constant, static code …

Data objects allocated at run-time. (Activation Records)

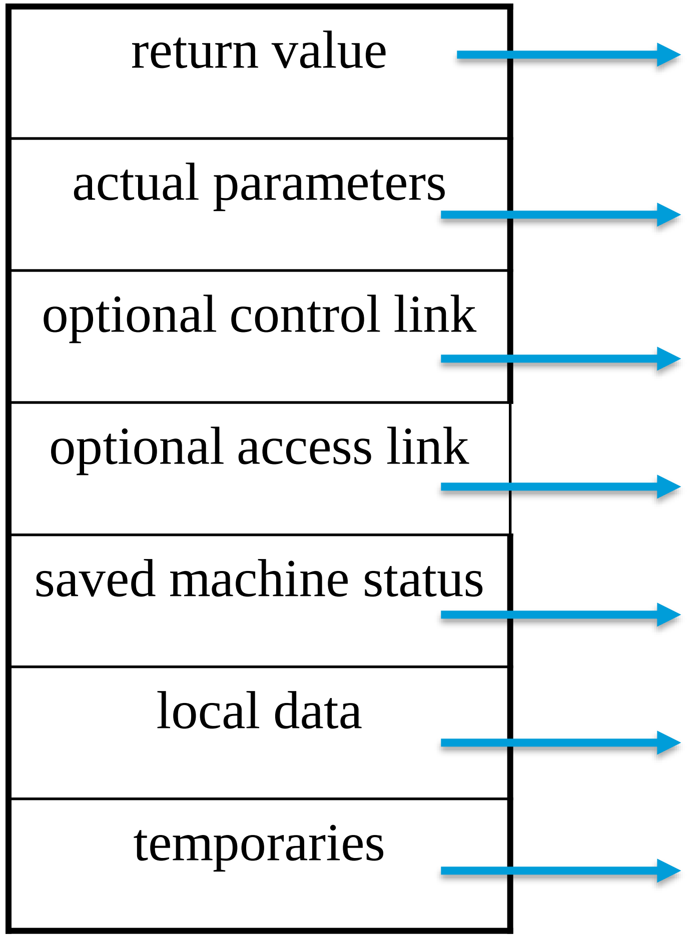
For activation records: local data, parameters, control info, ...

used for data whose lifetime is not as easily described

Run time environment

- ❑ The code area contains object code
  - ✓ For most languages, It is fixed in size and read only.
- ❑ The static area contains data (not code) with fixed addresses (e.g., global data)
  - ✓ It is in Fixed size for static data, and it may be readable or writable.
- ❑ The stack contains an AR (activation record) for each currently active procedure
  - ✓ Each AR usually fixed size, contains locals.
- ❑ The Heap contains all other data ie., whose lifetime is not described
  - ✓ Dynamic Data Structures
  - ✓ In C, the heap is managed by malloc and free functions.
  - ✓ In java , the new keyword is used for heap memory.

Activation Record



The returned value of the called procedure is returned in this field to the calling procedure. In practice, we may use a machine register for the return value.

The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.

The optional control link points to the activation record of the caller.

The optional access link is used to refer to nonlocal data held in other activation records.

The field for saved machine status holds information about the state of the machine before the procedure is called.

The field of local data holds data that local to an execution of a procedure..

Temporary variables are stored in the field of temporaries.

Storage allocation

The different ways to allocate the memory are :

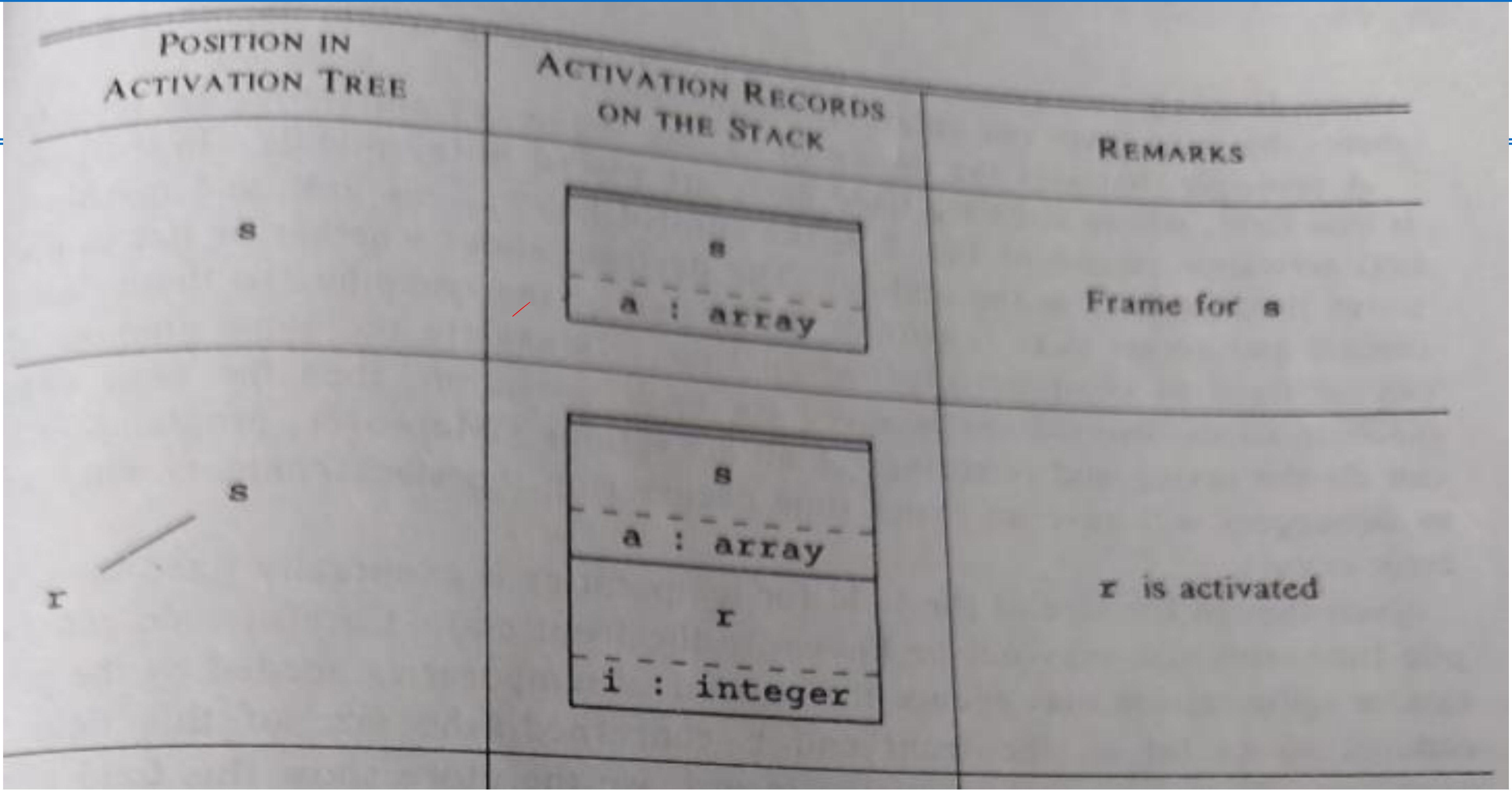
- ❑Static storage allocation
- ❑Stack allocation
- ❑Heap allocation

Static storage allocation

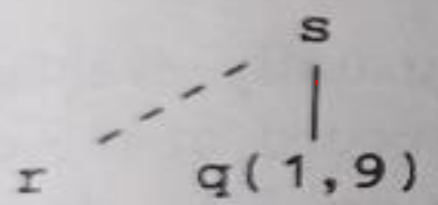
- ❑In static allocation, names are bound to storage locations.
- ❑If memory is created at compile time then the memory will be created in static area and only once.
- ❑Static allocation supports the dynamic data structure that means memory is created only at compile time and deallocated after program completion.
- ❑The drawback with static storage allocation is that the size and position of data objects should be known at compile time.
- ❑Another drawback is restriction of the recursion procedure.

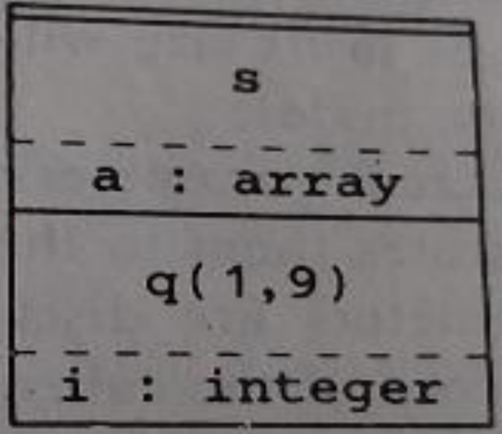
Stack Storage Allocation

- In static storage allocation, storage is organized as a stack.
- An activation record is pushed into the stack when activation begins and it is popped when the activation end.
- Activation record contains the locals so that they are bound to fresh storage in each activation record. The value of locals is deleted when the activation ends.
- It works on the basis of last-in-first-out (LIFO) and this allocation supports the recursion process.

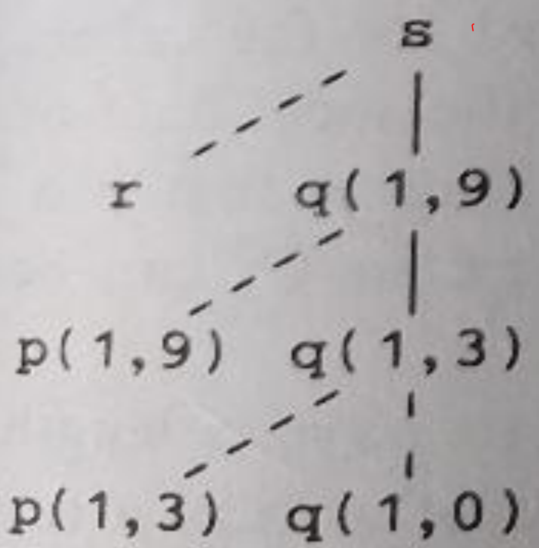


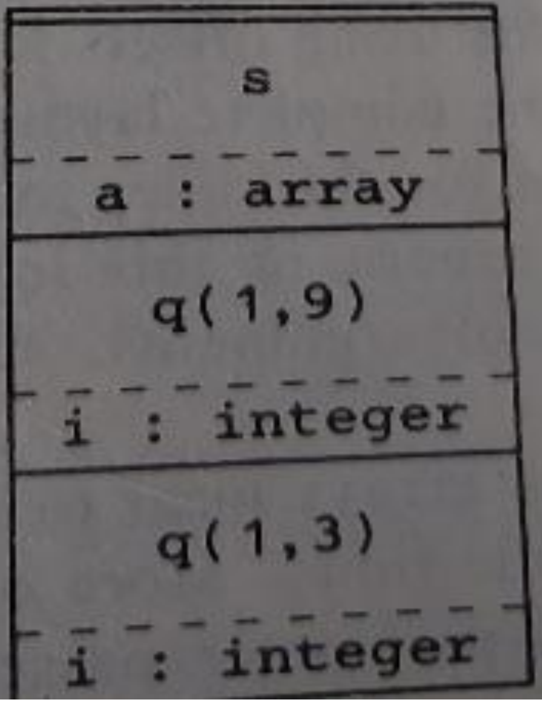
Position in Activation Tree	Activation Records on the Stack	Remarks
s	s a : array	Frame for s
s r	s a : array r i : integer	r is activated





Frame for r
has been popped
and q(1,9)
pushed



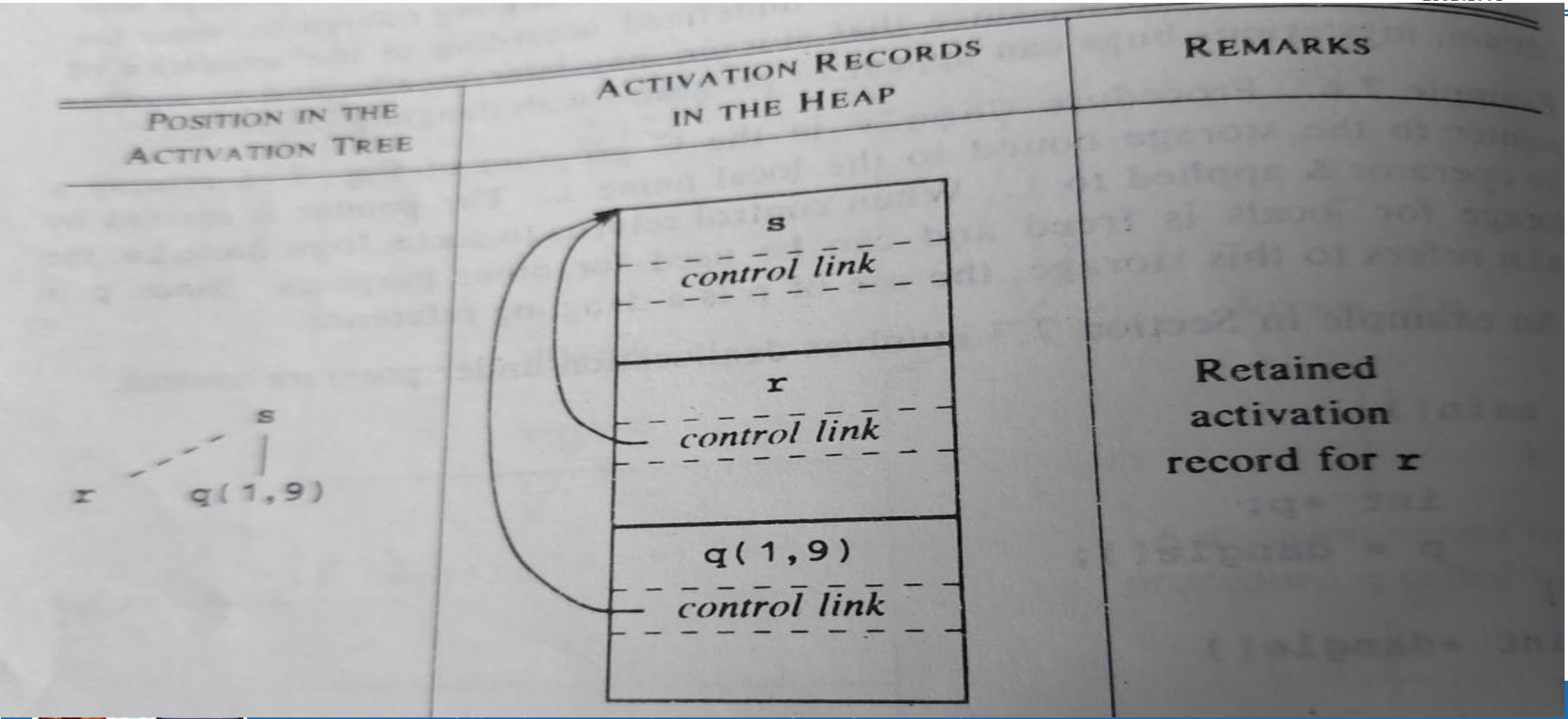


Control has just returned
to q(1,3)

Heap Storage Allocation

- Heap allocation is the most flexible allocation scheme.
- Allocation and deallocation of memory can be done at any time and at any place depending upon the user's requirement.
- Heap allocation is used to allocate memory to the variables dynamically.
- Heap storage allocation supports the recursion process.

Heap Storage Allocation



Position in the Activation Tree	Activation Records in the Heap	Remarks
s r q(1,9)	s *control link* r *control link* q(1,9) *control link*	Retained activation record for r

LANGUAGE FACILITCES FOR DYNAMIC STORAGE ALLOCATION

ESTD:1995

- Storage for dynamic data is usually taken from a heap. Allocated data is often retained until it is explicitly deallocated. The allocation itself can be either **explicit** or **implicit.**
- In Pascal, for example, **explicit allocation** is performed using the standard procedure new.
  - Execution of new(p) allocates storage for the type of object pointed to by p.
  - Deallocation is done by calling dispose in most implementations of Pascal.

- **Implicit** allocation occurs when evaluation of an expression results in storage being obtained to hold the value of the expression.
  - Snobol allows the length of a string to vary at run time, and manages the space needed to hold the string in a heap.

LANGUAGE FACILITCES FOR DYNAMIC STORAGE ALLOCATION

ESTD:1995

Language facilities for dynamic storage allocation : Garbage collection and dangling references

The **garbage collection** is a crucial component of memory management. It is the procedure of a program's memory being automatically identified and released.

C offers low-level memory management mechanisms through its malloc() and free() functions. The ***free()*** method is used to release memory when it is no longer required, while the ***malloc()*** function is used to allocate memory dynamically during runtime. These functions' fundamental syntax is as follows:

```
void* malloc(size_t size);
void free(void* ptr);
```

Garbage collection

The basic principles of garbage collection are finding data objects in a program that cannot be access.

Commonly implemented Garbage Collection techniques.

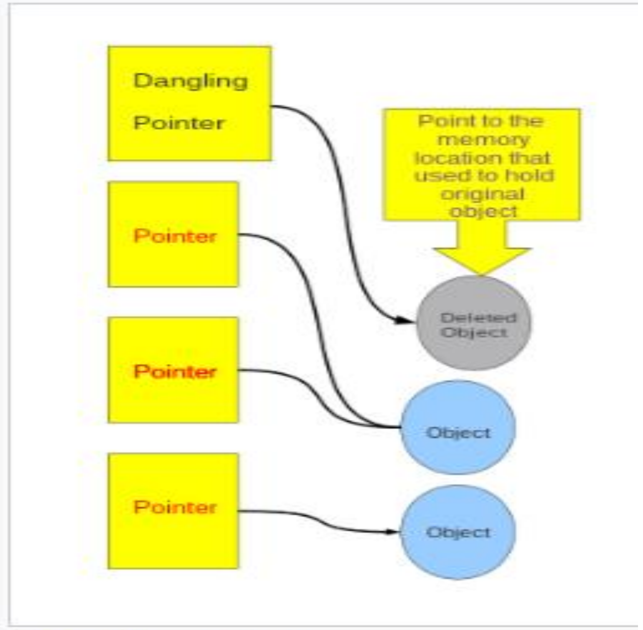
- ***Mark and Sweep***
- ***Reference Counting***

- ***Refer this link: https://www.javatpoint.com/garbage-collection-in-data-structure.***

Dangling references

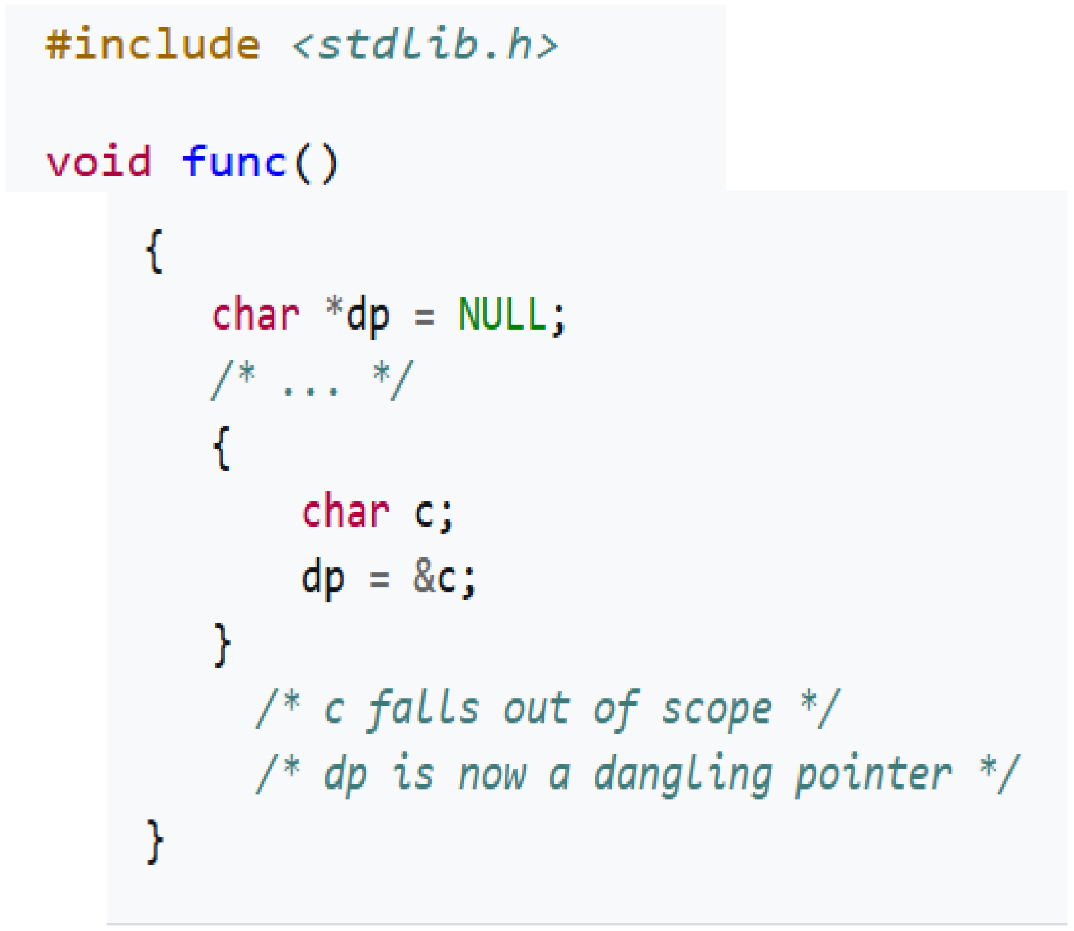
- A dangling reference is a situation in compiler design where a pointer or reference points to a memory location that has been deallocated or released. This can happen when an object is deleted or goes out of scope, but the reference to it still exists and is later accessed.

- Dangling pointers and wild pointers in computer programming are pointers that do not point to a valid object of the appropriate type. These are special cases of memory safety violations. More generally, dangling references and wild references are references that do not resolve to a valid destination.



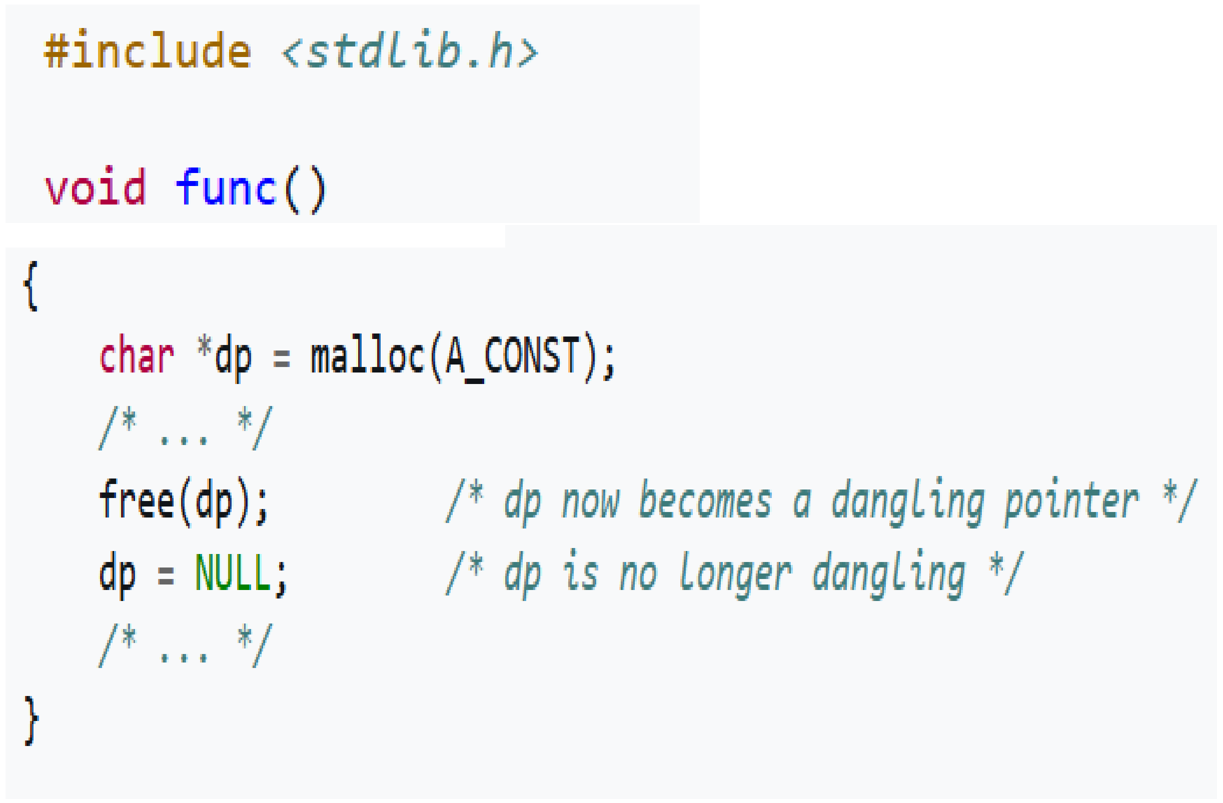
Dangling pointer

Dangling references



```c
#include <stdlib.h>

void func()
{
    char *dp = NULL;
    /* ... */
    {
        char c;
        dp = &c;
    }
      /* c falls out of scope */
      /* dp is now a dangling pointer */
}
```

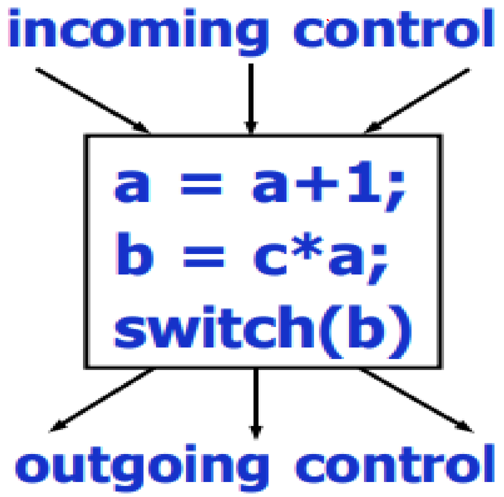


```c
#include <stdlib.h>

void func()
{
    char *dp = malloc(A_CONST);
    /* ... */
    free(dp);         /* dp now becomes a dangling pointer */
    dp = NULL;        /* dp is no longer dangling */
    /* ... */
}
```

Basics Block and flow graph

- **Basic block** = sequence of consecutive statements such that:
  - Control enters only at beginning of sequence
  - Control leaves only at end of sequence



- No branching in or out in the middle of basic blocks

Basics Block and flow graph

- Flow graph is a graphical representation of three address statements.

- Flow graph is useful for understanding code generation algorithms.

Partition algorithm

ESTD:1995

Input : A sequence of 3 address statements

Output: A sequence of basic blocks with each 3A Statement in exactly one block.

Method:

(1) First determine a set of leaders, the 1st statement of basic blocks:

- a) The first statement is a leader
- b) Any statement that is a target of a conditional or unconditional goto is a leader
- c) Any statement that immediately follows a goto, or conditional goto statement is a leader.

(2) For each leader its basic block consists of:

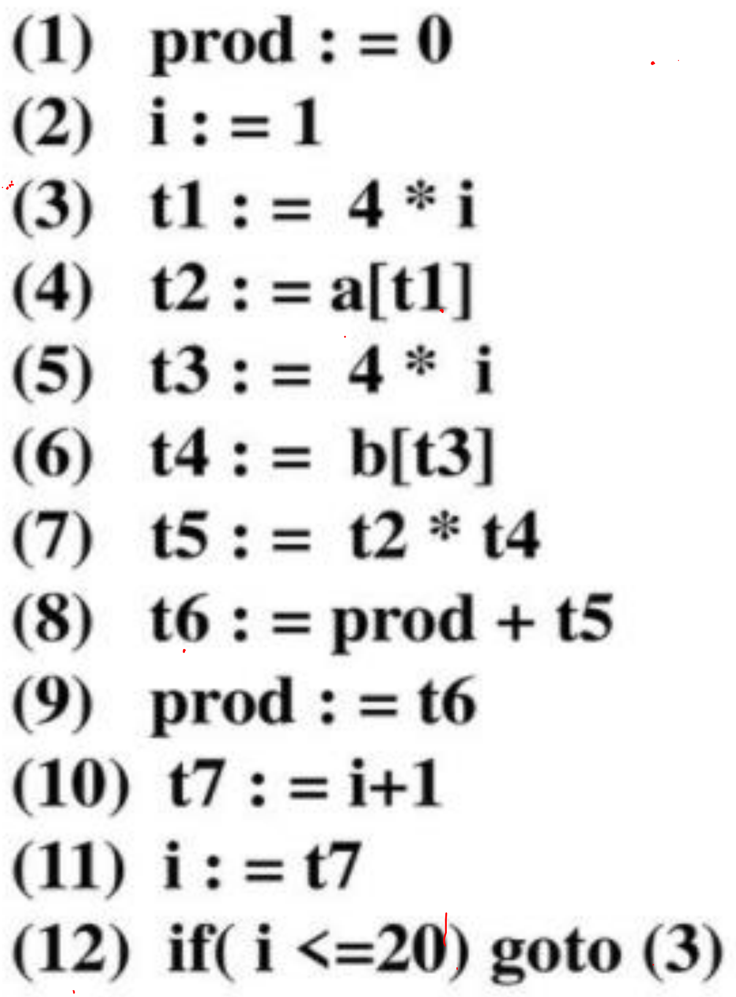
- a) The leader
- b) All statements upto but not including the next leader or the end of the program.

Example of Partition into Basic Blocks

Consider the program fragment:

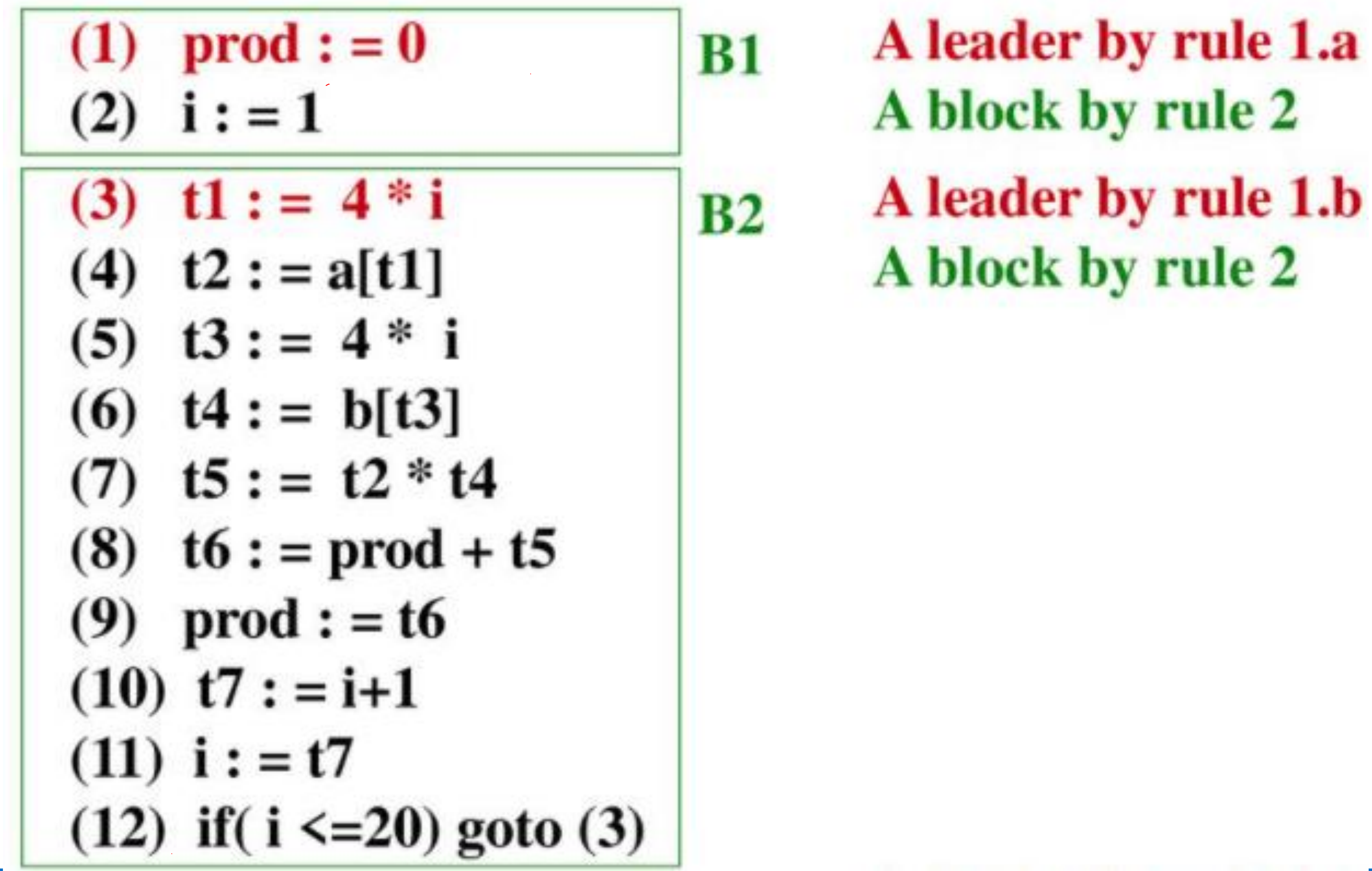
```
begin
    prod :=  o;
    i       :=  1
    do begin
          prod :=  prod + a[i] * b[i];
          i       :=  i   +  1 ;
         end
    while  i  <=  20
end
```

Three address code



```
(1)   prod : = 0
(2)   i : = 1
(3)   t1 : =  4 * i
(4)   t2 : = a[t1]
(5)   t3 : =  4 *  i
(6)   t4 : =  b[t3]
(7)   t5 : =  t2 * t4
(8)   t6 : = prod + t5
(9)   prod : = t6
(10)  t7 : = i+1
(11)  i : = t7
(12)  if( i <=20) goto (3)
```

Basics blocks by partition algorithm



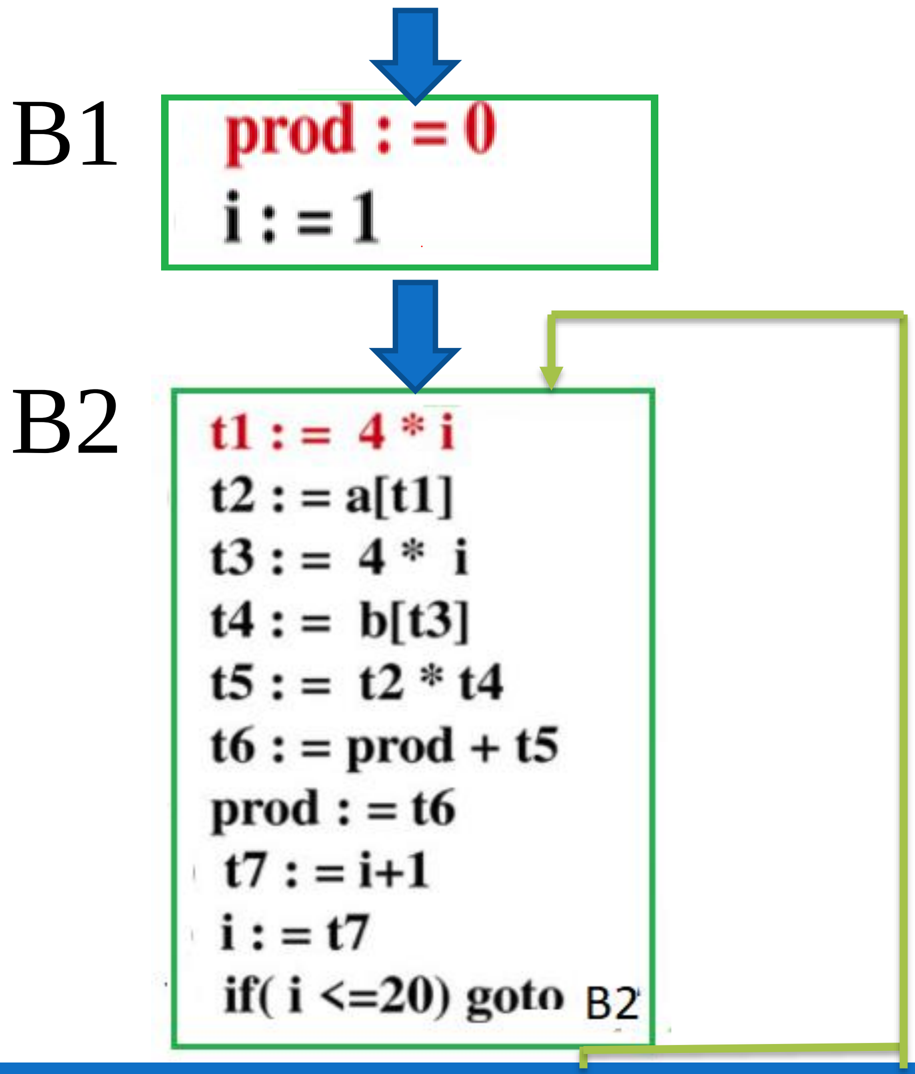
```
(1)  prod : = 0
(2)  i : = 1
```

B1 — A leader by rule 1.a
A block by rule 2

```
(3)  t1 : =  4 * i
(4)  t2 : = a[t1]
(5)  t3 : =  4 *  i
(6)  t4 : =  b[t3]
(7)  t5 : =  t2 * t4
(8)  t6 : = prod + t5
(9)  prod : = t6
(10) t7 : = i+1
(11) i : = t7
(12) if( i <=20) goto (3)
```

B2 — A leader by rule 1.b
A block by rule 2

Flow graph


B1
prod : = 0
i : = 1
B2
t1 : = 4 * i
t2 : = a[t1]
t3 : = 4 * i
t4 : = b[t3]
t5 : = t2 * t4
t6 : = prod + t5
prod : = t6
t7 : = i+1
i : = t7
if(i <=20) goto B2

Draw the flow graph for

```
1)  i  :=  1
2)  j  :=  1
3)  t1 := 10 * i
4)  t2 := t1  +  j
5)  t3 := 8 *t2
6)  t4 : = t3 -88
7)  a[t4] := 0.0
8)  j   :=  j  + 1
9)   if  j < =10 goto(3)
10)   i  := i  + 1
 11) if  i < =10 goto(2)
 12) i := 1
 13) t5 := i -1
 14) t6 := 88*t5
15)  a[t6] := 1.0
16) i := i  + 1
17) if  i < =10 goto(13)
```

Problem on flow graph

ESTD:1995

- Construct a flow graph for Quick sort partition algorithm (text book page numbers 588,590,591)

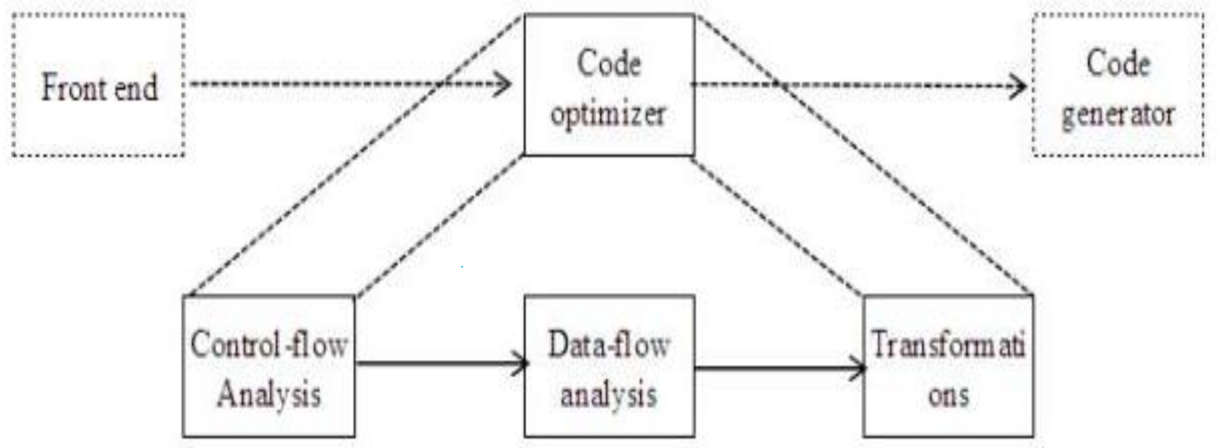
588 ↑ Alg

590 ↑ 3AC

591 ↑ flow graph

Refer : Text book : compilers Principles, Techniques & tools by AHO, Ravi Sethi Ullman

Code Optimization



A Code optimizer sits between the front end and the code generator.

- ❑Works with intermediate code.
- ❑Can do control flow analysis.
- ❑Can do data flow analysis.
- ❑Does transformations to improve the intermediate code.

Code Optimization

ESTD:1995

- ❑ The code produced by the straightforward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations.

- ❑ Compilers that apply code-improving transformations are called optimizing compilers.

Optimizations are classified into two categories. They are

- ❑ Machine independent optimizations
- ❑ Machine dependent optimizations

Machine independent optimizations:

- ❑ Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

Machine dependent optimizations:

- ❑ Machine dependent optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations
or
Properties of code optimization transformations

Simply stated, the best program transformations are those that yield the most benefit for the least effort. The transformations provided by an optimizing compiler should have several properties. They are:

1. The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program.

2. A transformation must, on the average, speedup programs by a measurable amount.

3. The transformation must be worth the effort.

PRINCIPAL SOURCES OF OPTIMISATION

- ❑ A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.
- ❑ Many transformations can be performed at both the local and global levels.
- ❑ Local transformations are usually performed first.

1. Function preserving
2. Loop optimization

Function-Preserving Transformations
or
Structure preserving transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

Function preserving transformations

Function preserving (or structure preserving)transformations examples:

- ❑ Common sub expression elimination
- ❑ Copy propagation
- ❑ Dead-code elimination
- ❑ Constant folding

Common Sub expressions elimination:

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation.

We can avoid recomputing the expression if we can use the previously computed value.

For example

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5
```

```
t1: = 4*i
 t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5
```

Copy Propagation:

- Assignments of the form f : = g called copy statements, or copies for short.
- Copy propagation means use of one variable instead of another.
- This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example:

```
x=Pi
A=x*r*r
```

The optimization using copy propagation can be done as follows: A=Pi*r*r

Here the variable x is eliminated

Dead-Code Eliminations:

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.

```
i=0;
if( i == 1)
{
        a=b+5;
}
```

Dead code

Here, ‘if’ statement is dead code because this condition will never get satisfied.

Constant folding:

ESTD:1995

Deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.

For example:

a=3.14 * 2 can be replaced by

a=6.28 there by eliminating a multiplication operation

Loop optimization

- ❑ Loop optimization is most valuable machine-independent optimization because program's inner loop takes bulk to time of a programmer.

- ❑ If we decrease the number of instructions in an inner loop then the running time of a program may be improved even if we increase the amount of code outside that loop.

For loop optimization the following three techniques are important:

1. Code motion
2. Induction-variable elimination
3. Strength reduction

1.Code motion or loop invariant competition

- Code motion is used to decrease the amount of code in loop.
- This transformation takes a statement or expression which can be moved outside the loop body without affecting the semantics of the program.

For example

```
while (i<=limit-2)
{
      ------
      ------
}
```

After code motion the result is as follows:

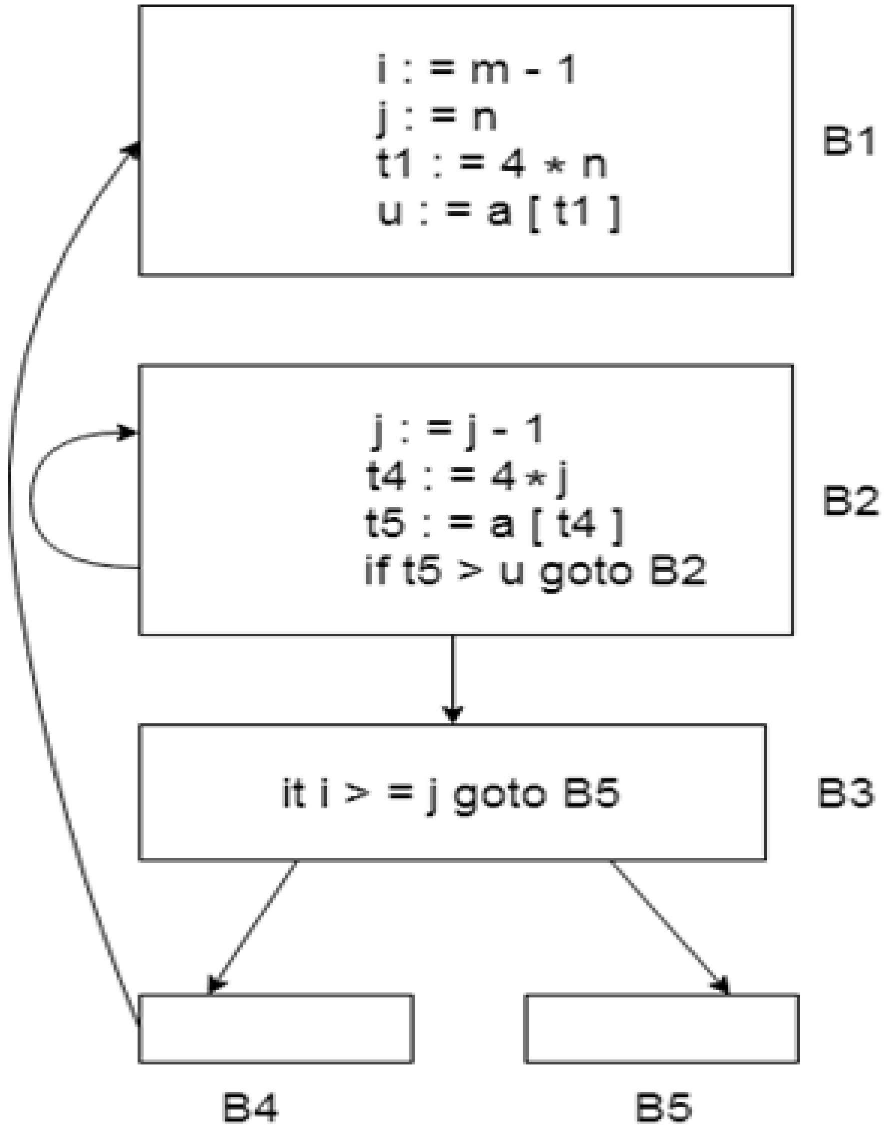
```
t= limit-2;
 while(i<=t)
{     ----
      ----
 }
```

2.Induction-Variable Elimination

Induction variable elimination can reduce the number of additions (or subtractions) in a loop, and improve both run-time performance and code space.

Definition: A variable x defined in a statement of the form x := x+ n or x := x - n is an induction variable and where n is a constant.
Induction variable x gets incremented or decremented by some constant in each iteration of the loop.

When there are two or more induction variables in a loop, it maybe possible to get rid of all but one, by the process of induction-variable elimination.



Example :

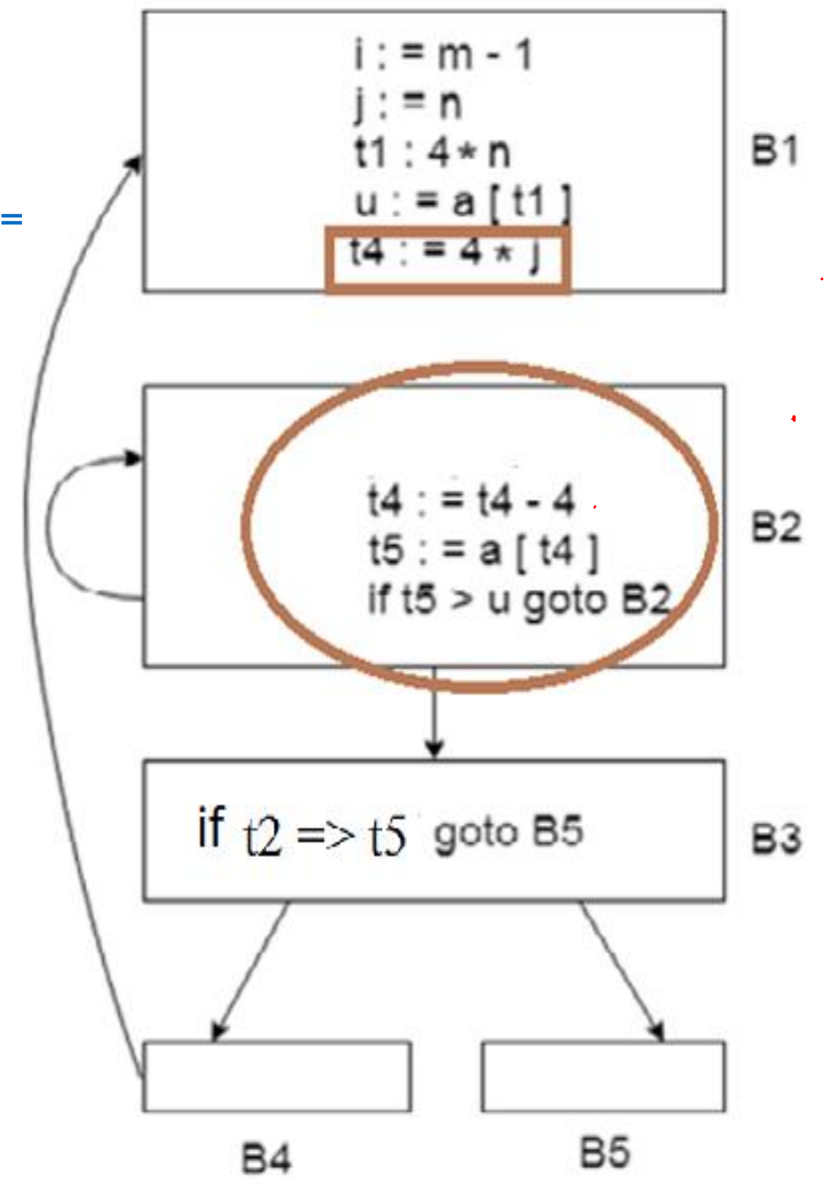
Assuming that j =10 :

Iteration	1	2	3	4	5	6
j	9	8	7	6	5	4
t4	36	32	28	24	20	16

The variable t4 gets the value decremented by 4 in each iteration
The variable j gets the value decremented by 1 in each iteration.

Therefore, the variables j and t4 are called induction variables. Thus, we can eliminate one of the two induction variable without affecting the other variables. That has been shown in the next slide.

Induction-Variable Elimination



3.Reduction In Strength:.

ESTD:1995

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine.

- ❑ Addition of a constant is cheaper than a multiplication. So we can replace multiplication with an addition within the loop.
- ❑ Multiplication is cheaper than exponentiation. So we can replace exponentiation with multiplication within the loop.

example 1: x^2 is invariably cheaper to implement as x*x

Strength of operators : bit operation < add/sub < multiply < divide < exponential

example 2:

a)	x := x+1	INC X
b)	x := y**2	x := y * y
c)	x := x*2	SHIFT X
d)	i := i+1; k := i*4	i := i+1; k=i; k := k+4

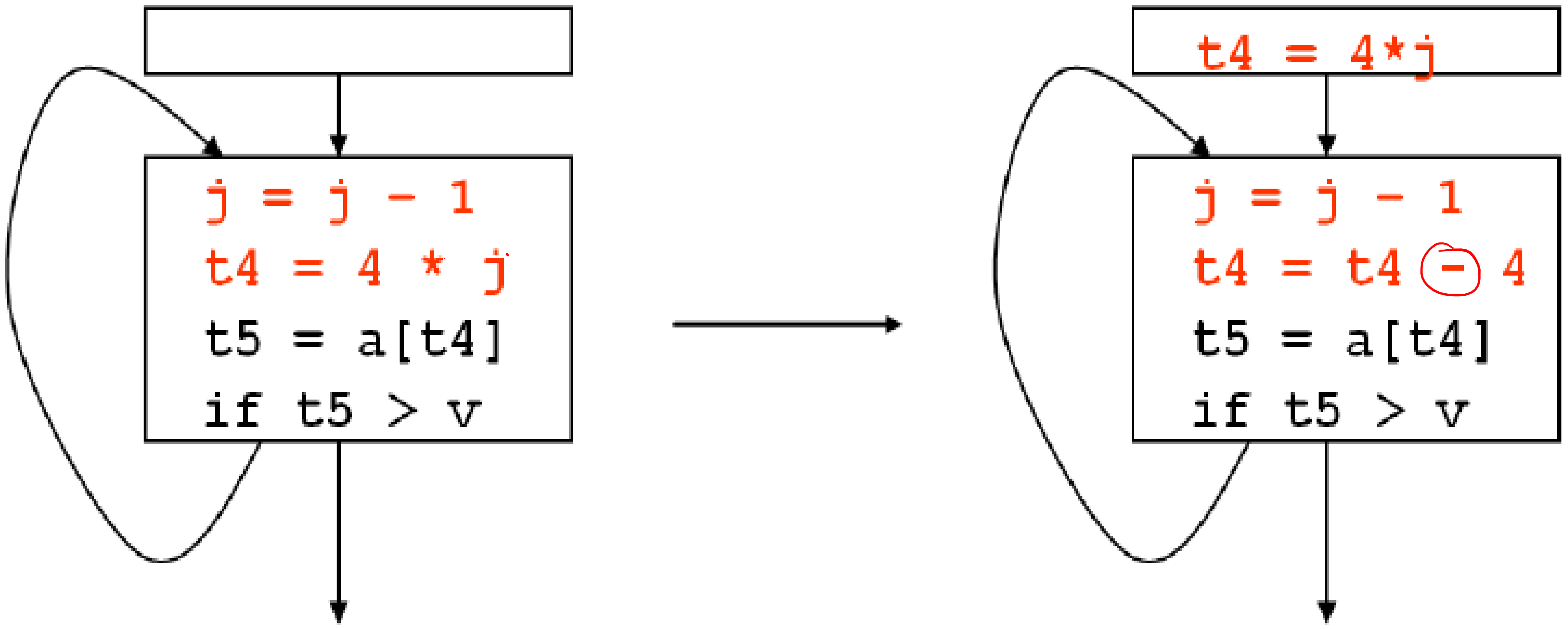
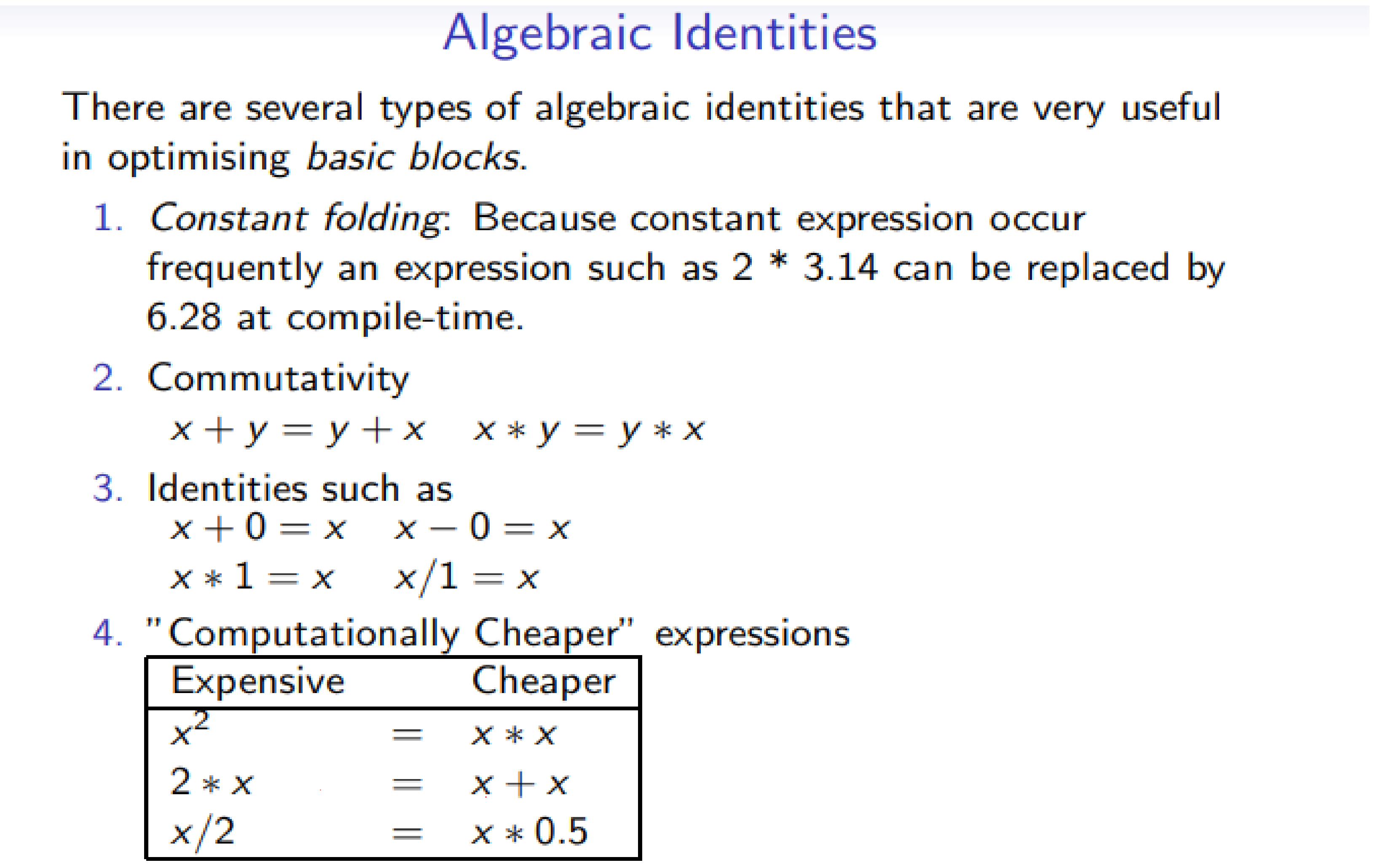


Figure : Reduction in strength.

Algebraic simplifications



Algebraic Identities

There are several types of algebraic identities that are very useful in optimising *basic blocks*.

1. *Constant folding*: Because constant expression occur frequently an expression such as 2 * 3.14 can be replaced by 6.28 at compile-time.
2. Commutativity

   $x + y = y + x \quad x * y = y * x$
3. Identities such as

   $x + 0 = x \quad x - 0 = x$

   $x * 1 = x \quad x/1 = x$
4. "Computationally Cheaper" expressions

Expensive		Cheaper
x^2	=	$x * x$
$2 * x$	=	$x + x$
$x/2$	=	$x * 0.5$

Loop Fusion

ESTD:1995

Loop fusion (or **loop** jamming) is a compiler optimization and **loop** transformation which replaces multiple **loops** with a single one. It is possible when two **loops** iterate over the same range and do not reference each other's data. **Loop fusion** does not always improve run-time speed.

The two adjacent loops on the code fragment below can be fused into on loop.

```
for (i = 0; i < 300; i++)
 a[i] = a[i] + 3;
 for (i = 0; i < 300; i++)
b[i] = b[i] + 4;
```

Below is the code fragment after loop fusion.

```
for (i = 0; i < 300; i++)
 {
      a[i] = a[i] + 3;
      b[i] = b[i] + 4;
 }
```

DAG Construction

DAG is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

- ❑ Leaf nodes represent identifiers, names or constants.
- ❑ Interior nodes represent operators.
- ❑ Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Application of DAGs:

1. Determining common sub expressions.
2. Identifying the names that are used in inside the block and out side the block.
3. Determining the statements computed values that could be used outside the block.
4. Finding the minimal number of register requiredfor the varibles in an application

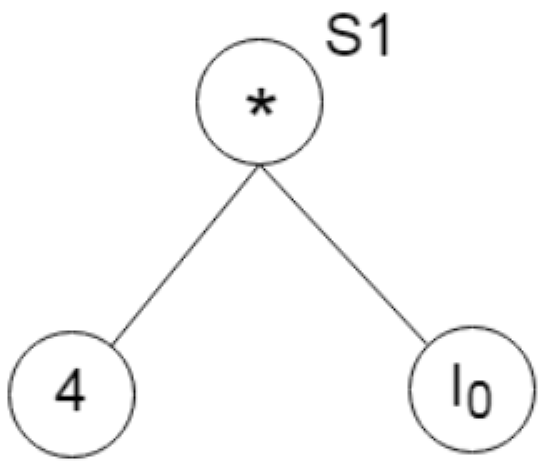
DAG Construction

Consider the following 3AC:

1.S1:= 4 * i
2.S2:= a[S1]
3.S3:= 4 * i
4.S4:= b[S3]
5.S5:= s2 * S4
6.S6:= prod + S5
7.Prod:= s6
8.S7:= i+1
9.i := S7
10.if i<= 20 **goto** (1)

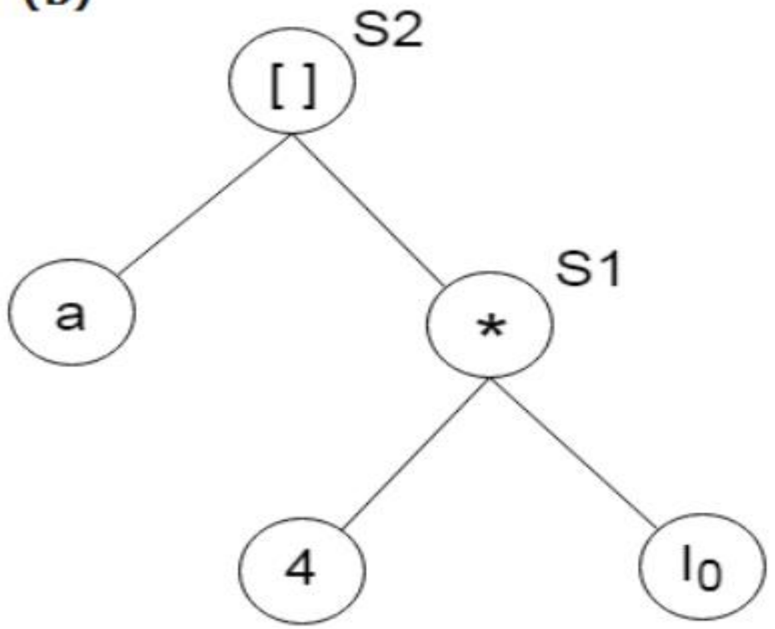
Stages in DAG Construction:

(a)



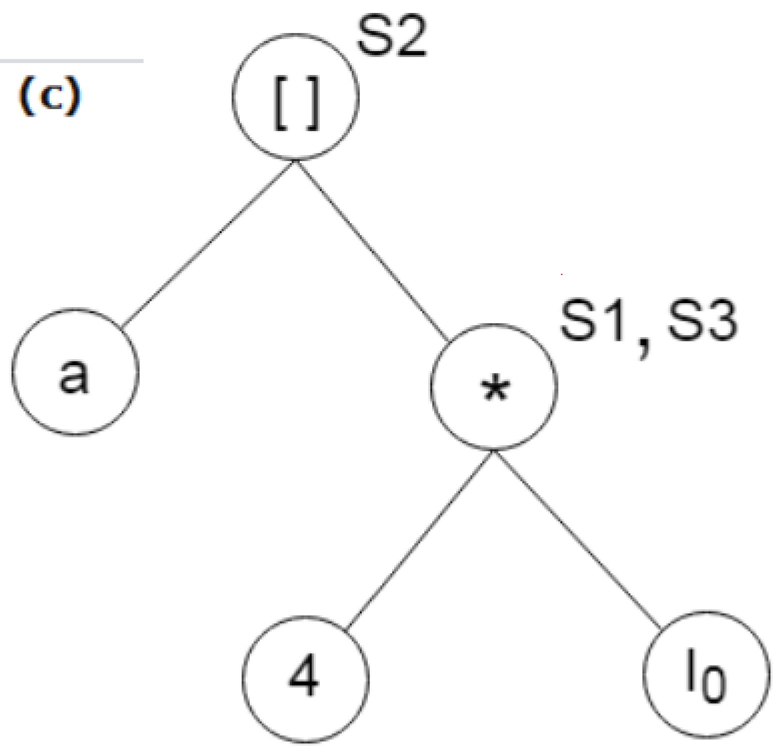
Statement (1)

(b)



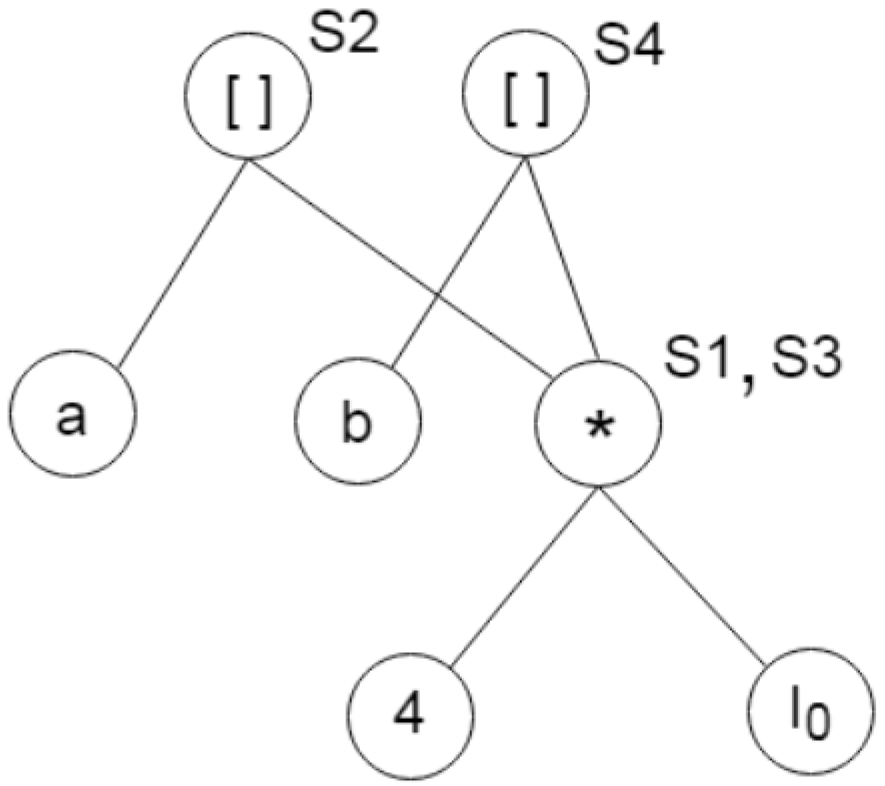
Statement (2)

(c)



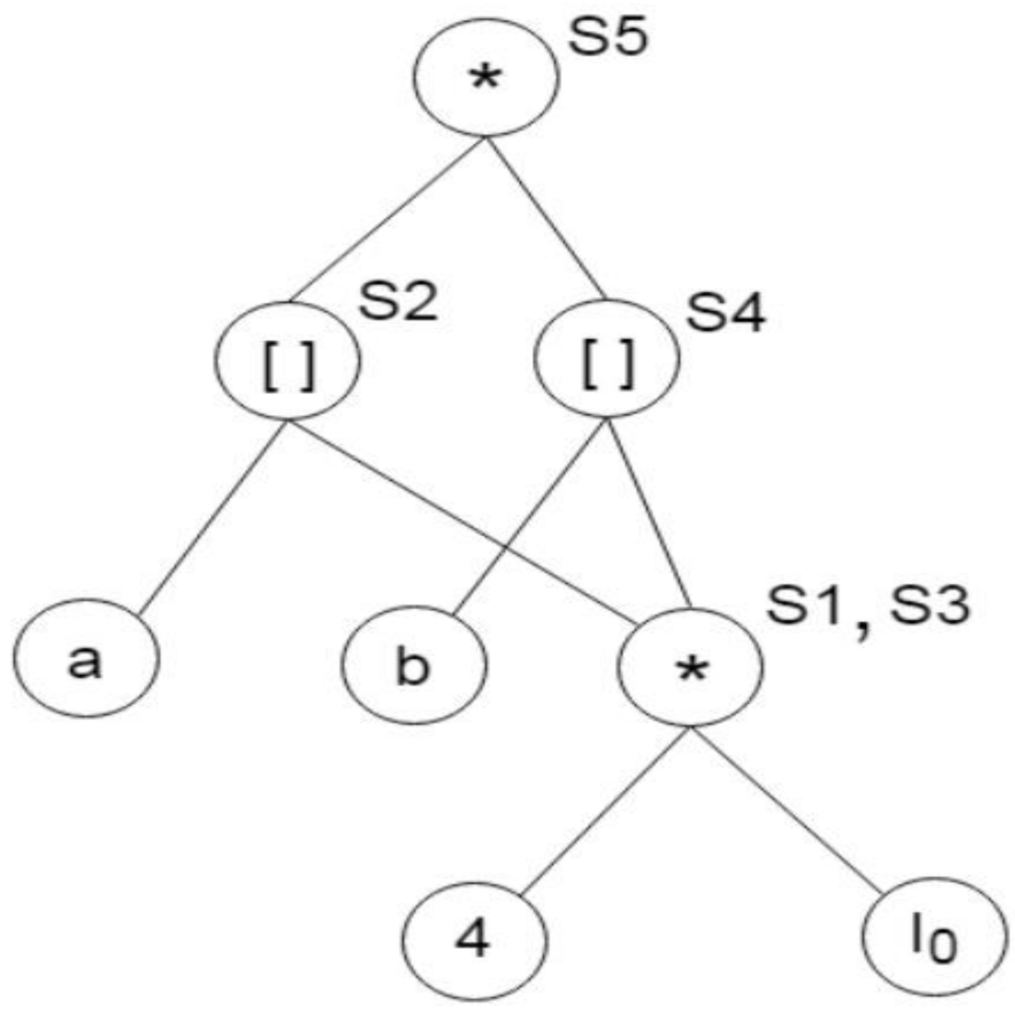
4 * I0 node exist already hence attach identifier S3 to the existing node for statement (3)

(d)



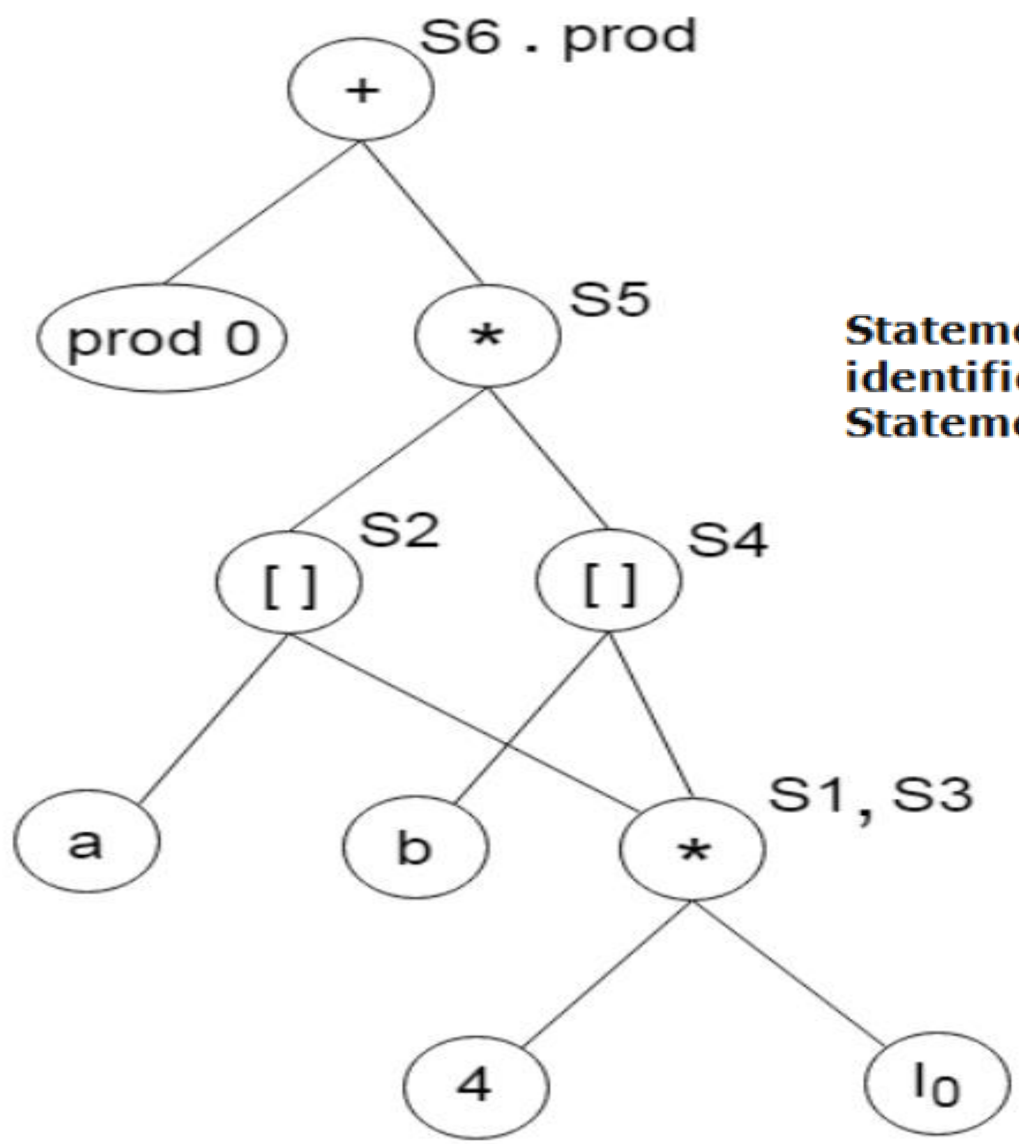
Statement (4)

(e)



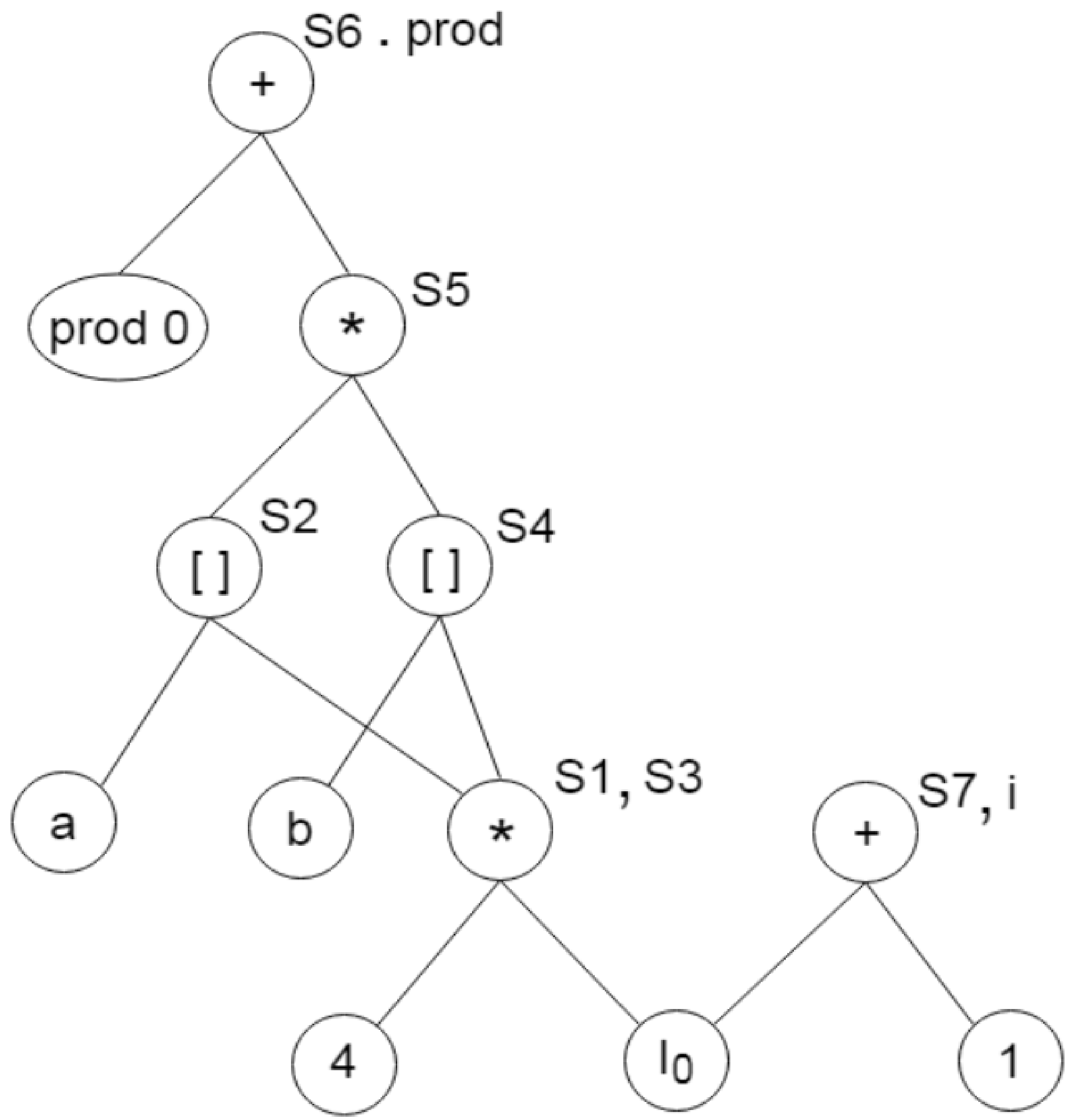
Statement (5)

(f)



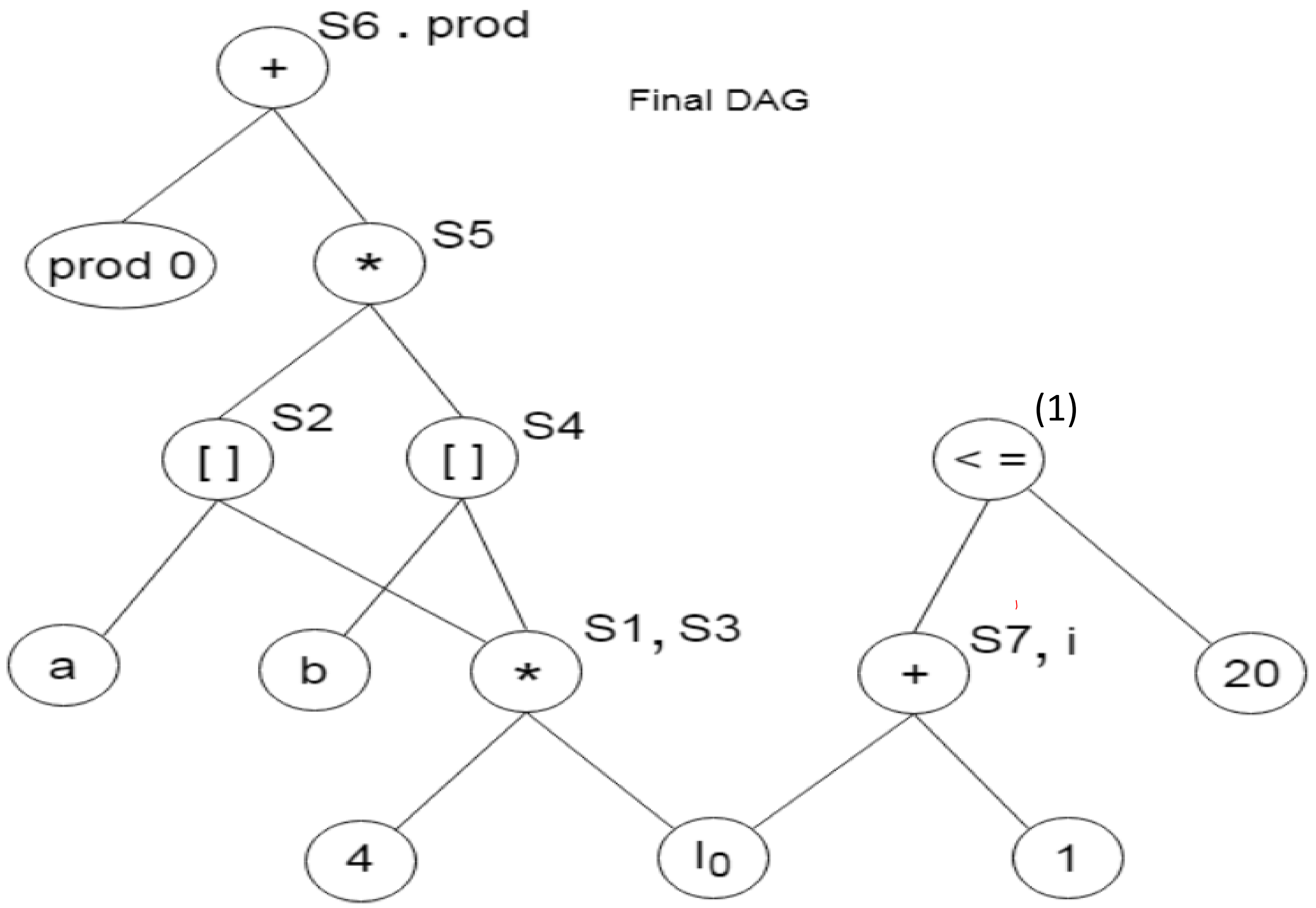
Statement (6), attach identifier prod for Statement (7)

(g)



Statement (8) , attach identifier i for Statement (9)

Final DAG



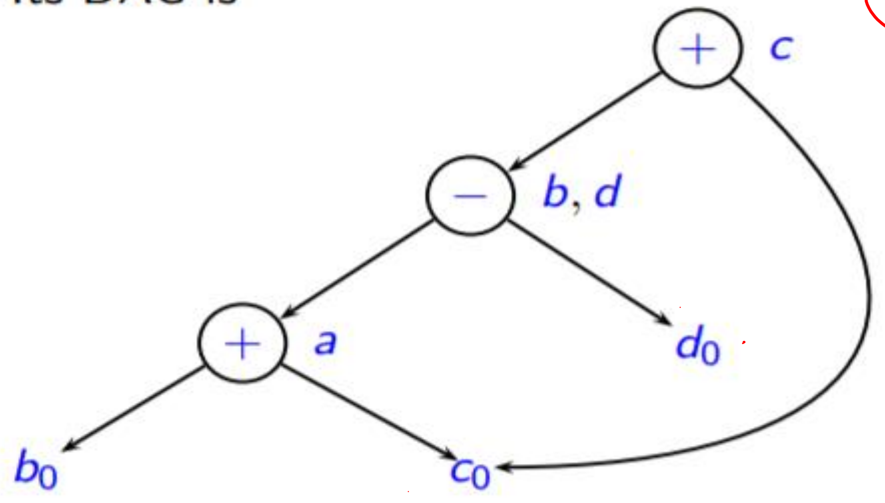
DAG based optimization

Local Common Subexpressions

Consider the block

```
a = b + c
b = a - d
c = b + c
d = a - d
```

Its DAG is



Imp

Explain about DAG based loop optimization.

From DAG the following is the code

```
a = b + c
d = a – d
c = d + c
```

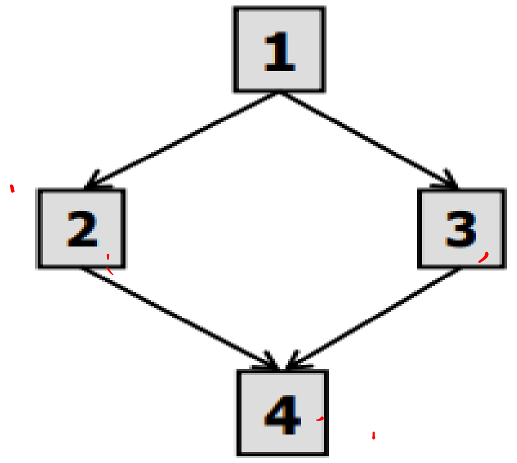
LOOPS IN FLOW GRAPH
(basic terminologies used in loops)

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms.

Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by d dom n



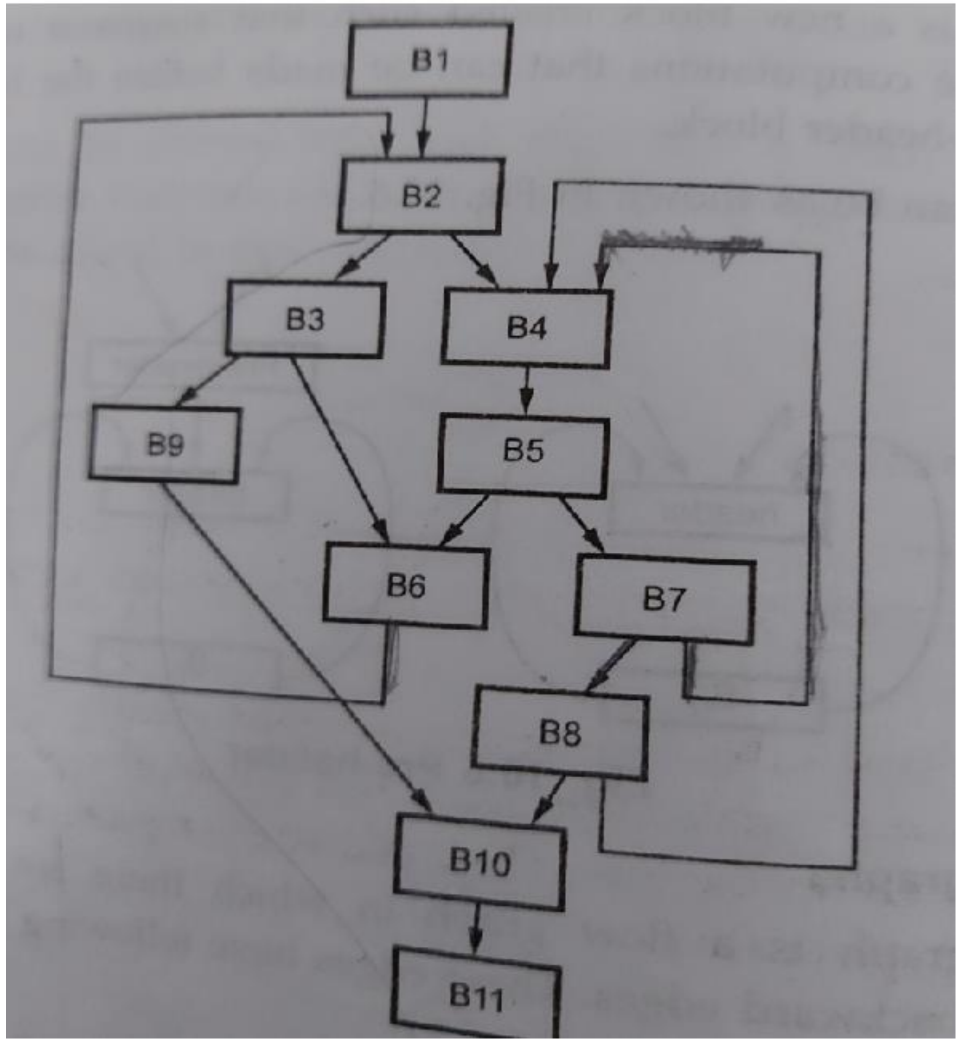
Every node dominates itself

1 dominates 1, 2, 3, 4

2 doesn't dominate 4

3 doesn't dominate 4

Home work : Determine the Dominators?



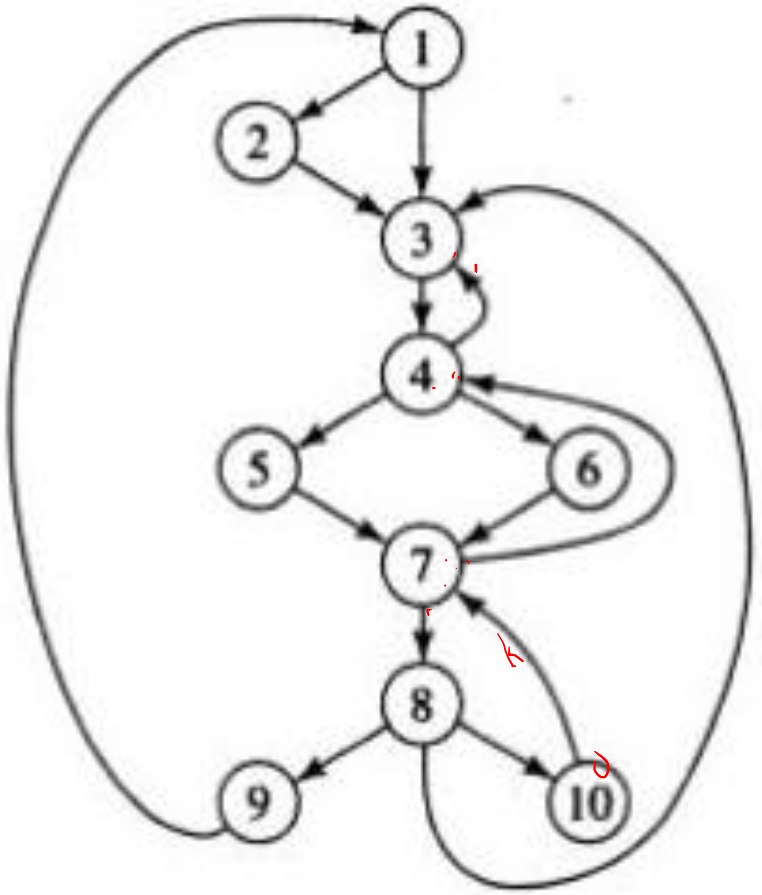
Natural Loops

One application of dominator information is in determining the loops of a flow graph. There are two essential properties of loops:

- A loop must have a single entry point, called the header. This entry point-dominates all nodes in the loop.
- There must be at least one path back to the header.

If $n \rightarrow d$ is an edge, d is the head and n is the tail. These types of edges are called as back edges.

Natural Loops example:



In the above graph,

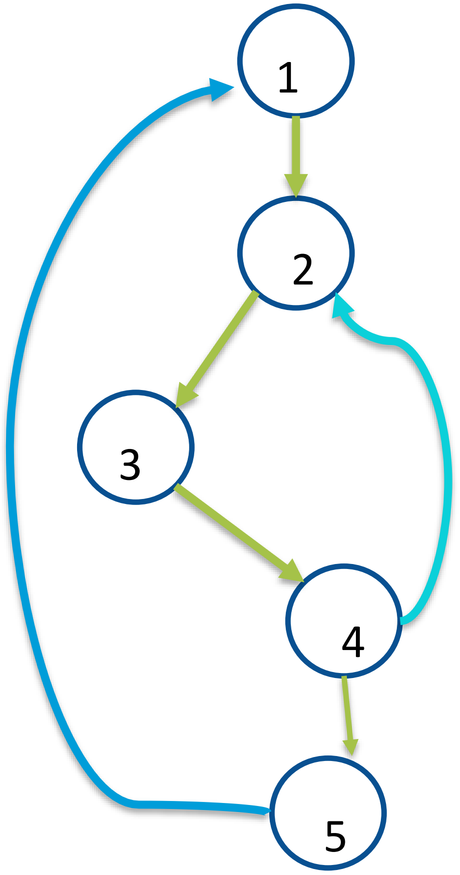
7→4   4 DOM 7

10 →7   7 DOM 10
4→3
8→3
9 →1

The above edges will form loop in flow graph.

Inner loops:

The inner loop is a loop that contains no other loops.



Here , the inner loop is 4 → 2.

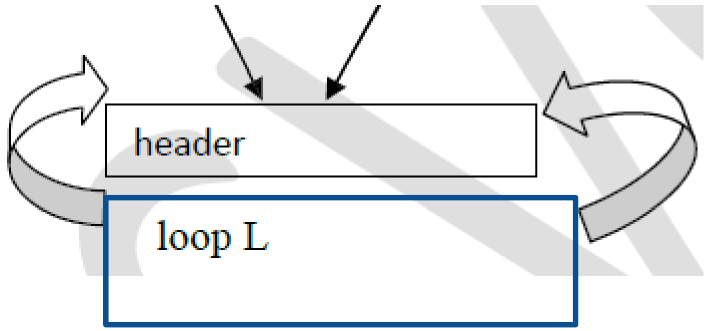
Pre-Headers

Several transformations require us to move statements “before the header”. Therefore begin treatment of a loop L by creating a new block, called the preheater.

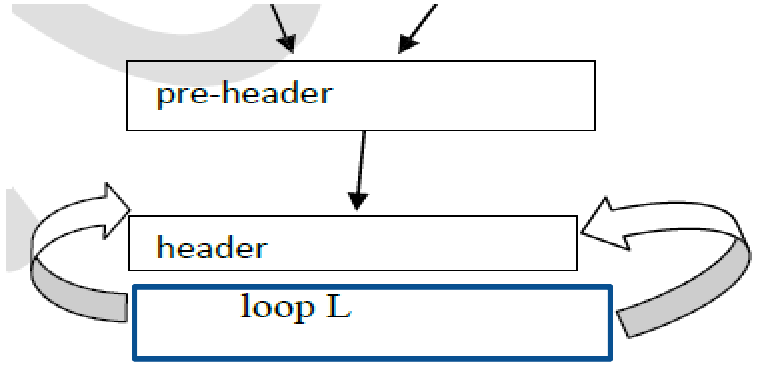
The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.

Edges from inside loop L to the header are not changed.

Initially the pre-header is empty, but transformations on L may place statements in it.



(a) Before



(b) After

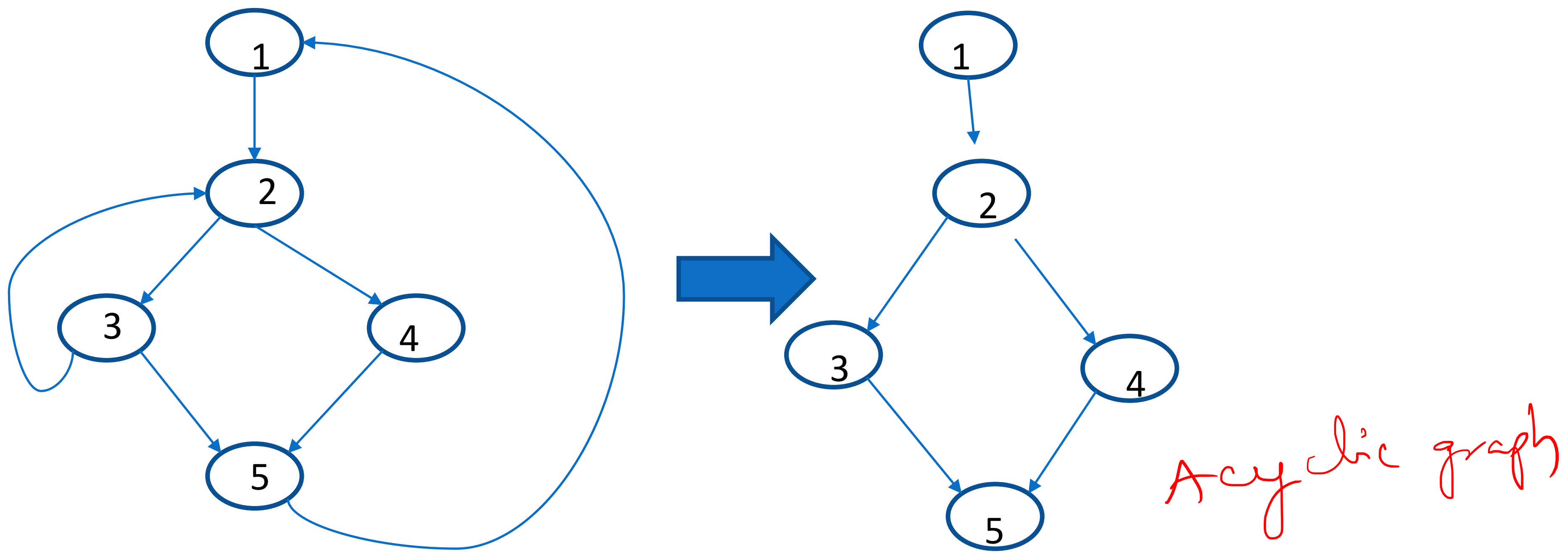
Reducible flow graphs

Definition: A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, *forward* edges and *back* edges, with the following properties.

- The forward edges from an acyclic graph in which every node can be reached from initial node of G.
- The back edges consist only of edges where heads dominate theirs tails.

By eliminating back edges , the resultant can be acyclic graph.

REDUCIABLE FLOW GRAPH



Global Optimization(control flow and data flow analysis)

Two types of global optimization:

- ❑ Control flow analysis
- ❑ Data flow analysis

The control flow analysis performs the analysis towards optimization by knowing the information about the arrangement of nodes(basic blocks),presence of loops, nesting of loops, nodes visited before the visiting the specific nodes.

Thus, in control flow analysis, the analysis is made on the flow of control by examining the program flow.

Data-flow analysis

ESTD:1995

- Data flow analysis techniques derive information about the flow of data along program execution paths.
- An execution path (or path) from point p1 to point pn is a sequence of points $p_1, p_2, ..., p_n$ such that for each i = 1, 2, ..., n − 1, either
  - p_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that same statement, or
  - pi is the end of some block and pi+1 is the beginning of a successor block
- Data flow information cab be collected by setting up and solving the system of equations that relate information at various points in a program. A typical equation has the form

$$OUT[B] = gen[B] \cup (IN[B] - KILL[B])$$

Data-flow properties

- The data flow analysis is a process in which the values are computed using data flow properties

The following are the data flow properties in the data flow analysis.

- **Live variables**
- **Available expressions**
- **Reaching definitions**

END of UNIT 5