KESHAV MEMORIAL INSTITUTE OF TECHNOLOGY

AN AUTONOMOUS INSTITUTION - ACCREDITED BY NAAC WITH 'A' GRADE

Narayanaguda, Hyderabad.

AUTOMATA THEORY AND COMPILER DESIGN(KR23) (23CC401PC)

BY:
Dr. Patil Yogita Dattatraya
Associate Professor CSE Dept, KMIT

Unit-5 Syllabus

Runtime Environment: Storages organization, Storage allocation strategies: Static, Stack, Heap allocations, Activation Record.

Code Optimization: Introduction, Principal sources of optimization, basic block, partition algorithm of basic block, flow graph, techniques of loop and global optimizations.

Code generation: Issues in code generation, DAG, Simple code generator.

Topics:

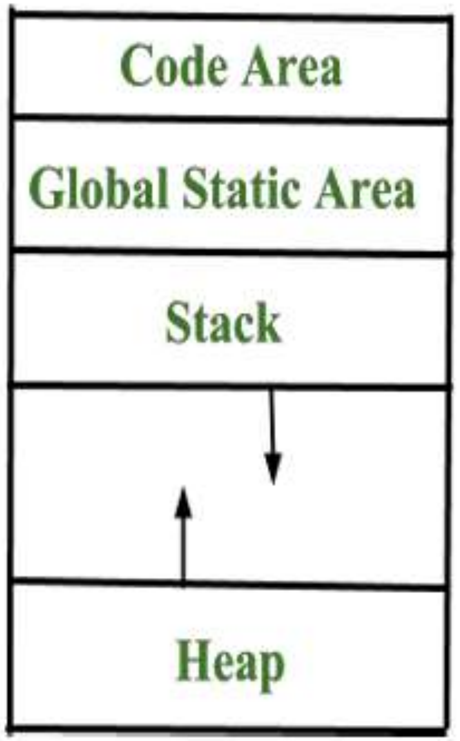
Runtime Environment:

- **Storages organization, Storage allocation strategies: Static, Stack, Heap**
- **Activation Record**

Run Time Environment

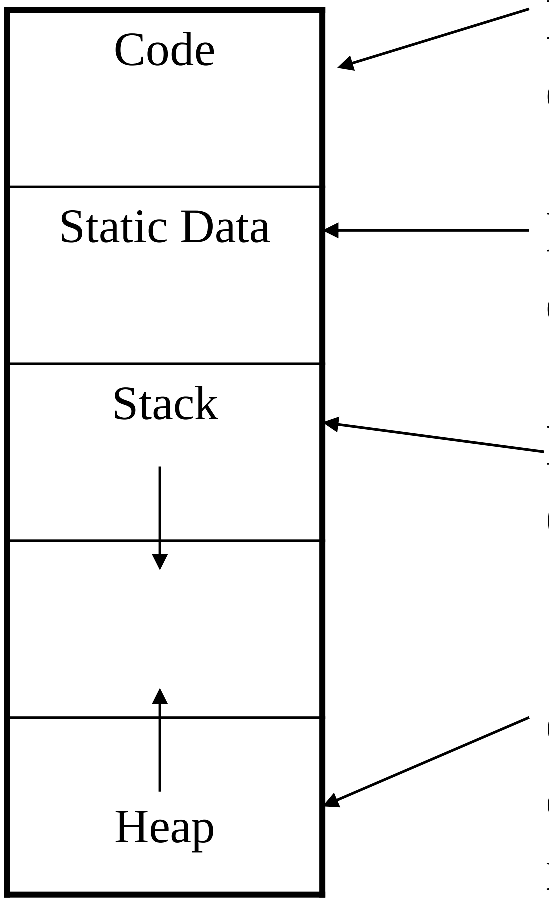
Runtime storage can be subdivided to hold:

- Code area- the program code, it is static as its size can be determined at compile time
- Static data objects
- Dynamic data objects- heap
- Automatic data objects- stack



Typical subdivision of ru
time memory

Run-Time Storage Organization



Memory locations for code are determined at compile time.

Locations of static data can also be determined at compile time.

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

Other dynamically allocated data objects at run-time. (For example, malloc area in C).

STORAGE ORGANISATION

The executing target program runs in its own logical address space in which each program value has allocation.

The management and organization of this logical address space is shared between the complier, operating system and target machine.

The operating system maps the logical address into physical addresses, which are usually spread through out memory.

Storage Allocation

The three (3) different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time

2. **Stack allocation** – manages the run-time storage as a stack.

3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap

1.Static Storage Allocation

• For any program if we create memory at compile time, memory will be created in the static area.

• It doesn’t support dynamic data structure i.e memory is created at compile time and deallocated after program completion.

• The drawback with static storage allocation is recursion is not supported.

• Another drawback is size of data should be known at compile time

Example:

i) int arr [200]; size can’t be reduced during compile time

ii) Int arr [10]; size can’t be increased during the compile time

2. Stack Storage Allocation

• Storage is organised as a stack and activation records are pushed and popped as activation begins and ends respectively.

Locals are contained in activation records so they are bound to fresh storage in each activation.

• **Recursion is supported in stack allocation.**

Example of Function call:

3. Heap Storage Allocation

• Memory allocation and deallocation can be done at any time and at any place depending on the requirement of the user.

• Heap allocation is used to dynamically allocate memory to the variables and deallocates back when the variables are no more required.

• Recursion is supported.

Activation Record

An activation record is another name for Stack Frame.

Information needed by a single execution of a procedure is managed using a contiguous block of storage called activation record

It is used by calling procedure to return a value to calling procedure. It is used by calling procedures to supply parameters to the called procedures.

It points to activation record of the caller.

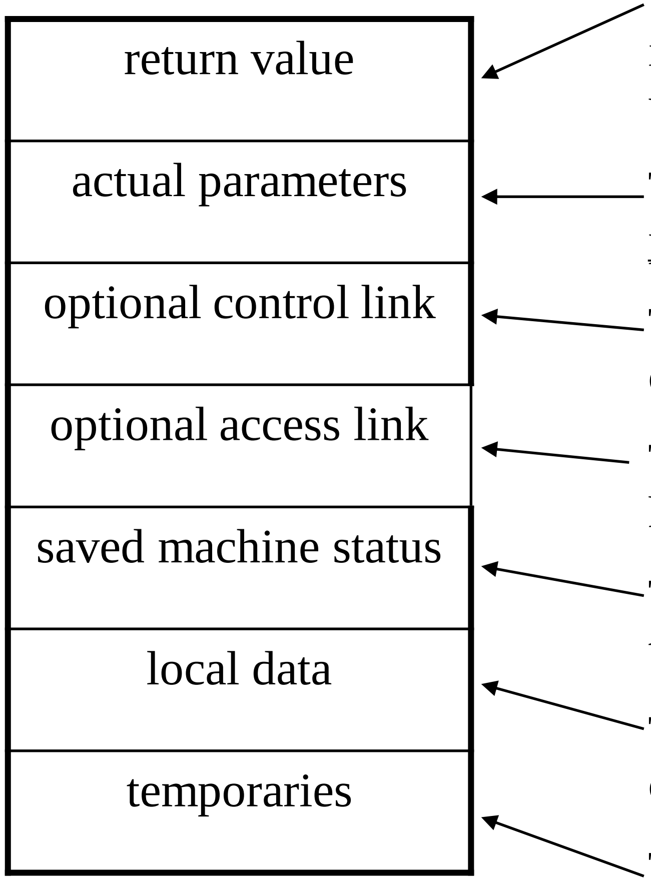
It is used to refer to non-local data held in other activation records.

It holds the information about status of machine before the procedure is called.

It holds the data that is local to the execution of the procedure.

It stores the value that arises in the evaluation of an expression.

Activation Records (cont.)



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. [called in]

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

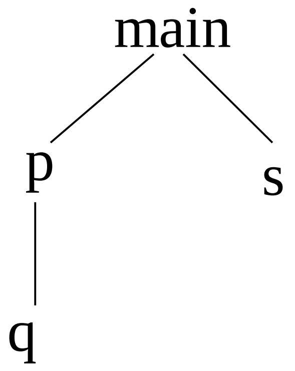
The field for saved machine status holds information about the state of the machine before the procedure is called.

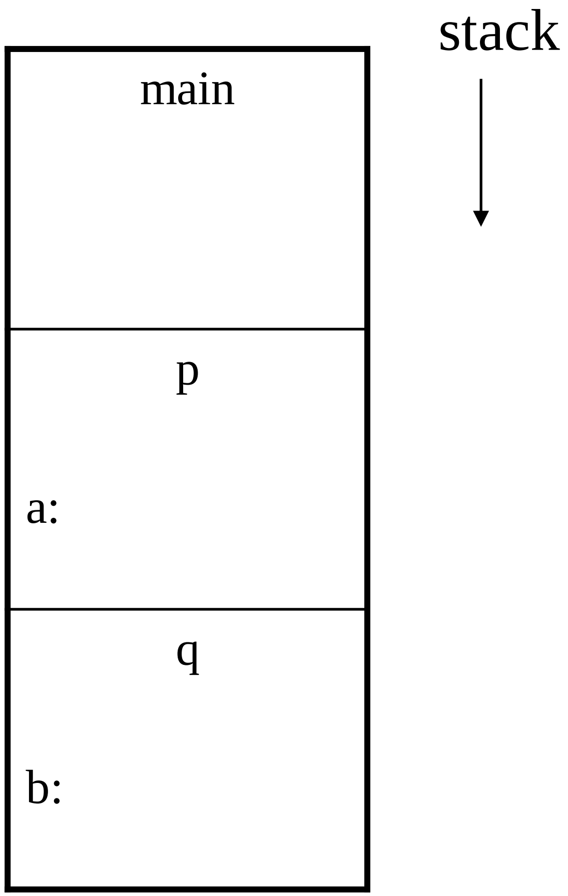
The field of local data holds data that local to an execution of a procedure..

Temporary variables is stored in the field of temporaries.

Activation Records (Ex1)

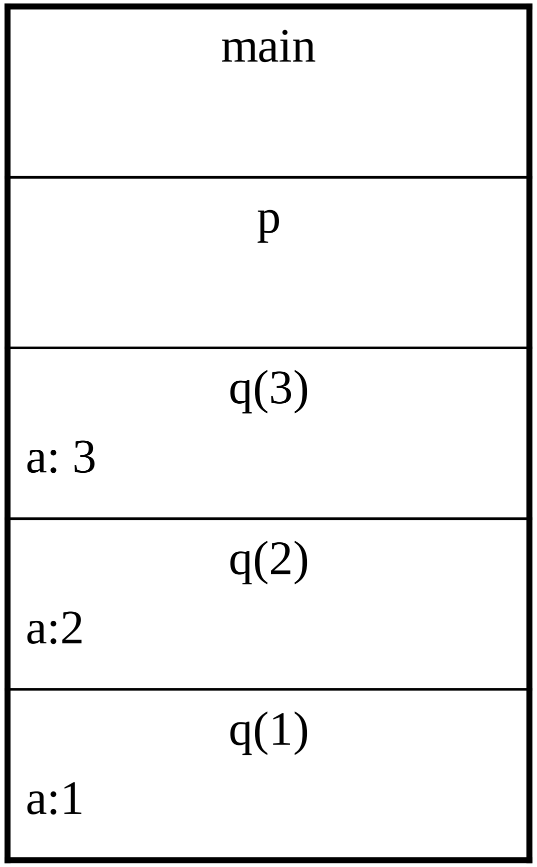
```
program main;
  procedure p;
    var a:real;
    procedure q;
      var b:integer;
      begin ... end;
    begin q; end;
  procedure s;
    var c:integer;
    begin ... end;
  begin p; s; end;
```





Activation Records for Recursive Procedures

```
program main;
  procedure p;
    function q(a:integer):integer;
      begin
        if (a=1) then q:=1;
        else q:=a+q(a-1);
      end;
    begin q(3); end;
  begin p; end;
```



Code Optimization:

Introduction, Principal sources of optimization, basic block, partition algorithm of basic block, flow graph, techniques of loop optimizations

Code optimization

Code Optimization is an approach to enhance the performance of the code.

The process of code optimization involves-

- Eliminating the unwanted code lines
- Rearranging the statements of the code

Advantages-

The optimized code has the following advantages-

- Optimized code has faster execution speed.
- Optimized code utilizes the memory efficiently.
- Optimized code gives better performance.

Types of Code Optimization:
The optimization process can be broadly classified into two types

1. **Machine Independent Optimization:** This code optimization phase attempts to **improve the intermediate code to get a better target code** as the output. The part of the intermediate **code which is transformed here does not involve any CPU registers or absolute memory locations.**

```
do
{
   item = 10;
   value = value + item;
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
   value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

Types of Code Optimization

2. **Machine Dependent Optimization:**

Machine-dependent optimization is **done after the target code has been generated** and when the code is transformed according to the target machine architecture.

It **involves CPU registers and may have absolute memory references** rather than relative references.

Machine-dependent optimizers put efforts to **take maximum advantage of the memory hierarchy.**

Code optimization

Steps before optimization:

1) Source program should be converted to Intermediate code
2) Basic blocks construction
3) Generating flow graph
4) Apply optimization

Basic Blocks

A basic block is defined as a sequence of consecutive statements with only one entry(at the beginning) and one exit (at the end).

Characteristics of Basic Blocks:

1. They do not contain any kind of jump statements in middle of the code.
2. There is no possibility of branching or getting halt in the middle.
3. All the statements execute in the same order they appear.
4. They do not lose the flow control of the program.

Partitioning three-address code into basic blocks.

Algorithm: Partition into basic blocks

Input: A sequence of three-address statements

Output: A list of basic blocks with each three-address statement in exactly one block.

Method:

1. We first determine the **set of leaders**, the first statements of basic blocks. The rules we use are of the following:
   a. The first statement is a leader.
   b. Any statement that is the 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 the leader and all statements up to but not including the next leader or the end of the program.

Example:
Consider the following source code for dot product of two vectors:

```
begin
        prod :=0;
        i:=1;
        do begin
        prod :=prod+ a[i] * b[i];
        i :=i+1;
        end
        while i <= 20
end
```

The three address code sequence for the above source is given as follows:

```
(1) PROD = 0
(2) I = 1
(3) T1 = 4 x I
(4) T2 = addr(A) – 4
(5) T3 = T2[T1]
(6) T4 = addr(B) – 4
(7) T5 = T4[T1]
(8) T6 = T3 * T5
(9) PROD = PROD + T6
(10) I = I + 1
(11) IF I <=20 GOTO (3)
```

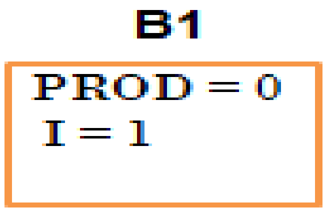
Example: Consider the following source code for dot product of two vectors:

Solution: For partitioning the three address code to basic blocks.

Rule1 Determine leaders

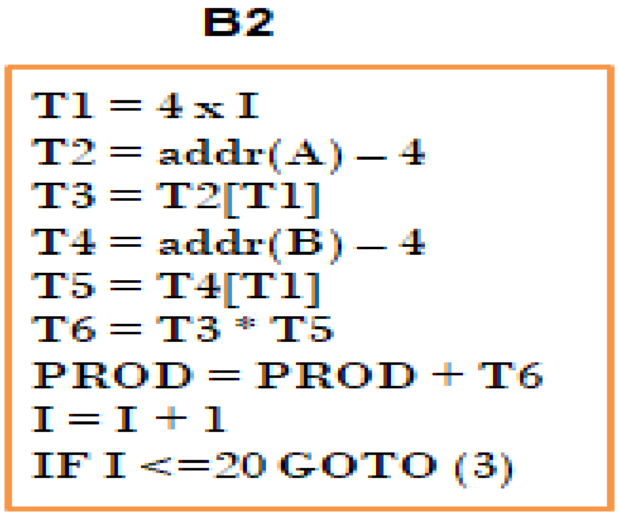
- PROD = 0 i.e. instruction 1 is a leader since first statement of the code is a leader.
- T1 = 4 * I i.e. instruction 3 is a leader since target of the conditional goto statement is a leader.
- There is no statement that immediately follows a goto or conditional goto statement . So no further leader.

Rule 2 Determine basic block : The basic block begins with leader and ends before the next leader. Two blocks are B1 and B2.



B1

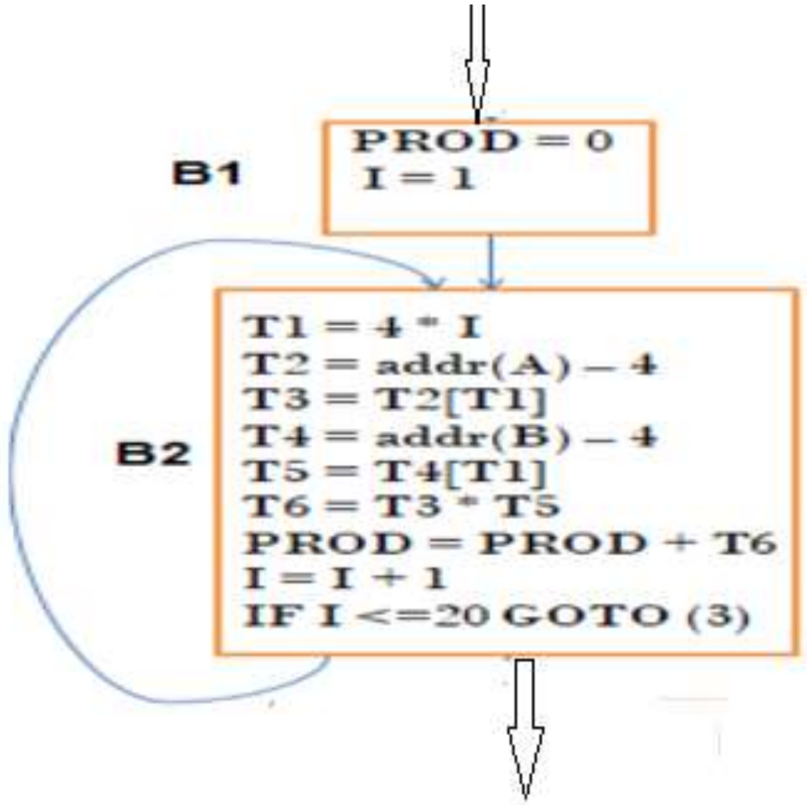
```
PROD = 0
 I = 1
```



B2

```
T1 = 4 x I
T2 = addr(A) – 4
T3 = T2[T1]
T4 = addr(B) – 4
T5 = T4[T1]
T6 = T3 * T5
PROD = PROD + T6
I = I + 1
IF I <=20 GOTO (3)
```

Note: Go through the next slides to understand the flow graph



Flow Graph

A flow graph is a directed graph with flow control information added to the basic blocks.

The nodes of the flow graph are basic blocks.

Properties of Flow Graphs

1. The control flow graph is process-oriented.
2. A control flow graph shows how program control is passed among the blocks.
3. The control flow graph depicts all of the paths that can be traversed during the execution of a program.
4. It can be used in software optimization to find unwanted loops.

Representation of Flow Graphs

Flow graphs are directed graphs.

The nodes/bocks of the control flow graph are the basic blocks of the program.

There are two designated blocks in Control Flow Graph:

1. **Entry Block:** The entry block allows the control to enter in the control flow graph.
2. **Exit Block:** Control flow leaves through the exit block.

Important [Construction of Flow Graph]

- The first step is to divide a group of three-address codes into the **Basic blocks**.
- To find **Basic Blocks** find **leader statements** from three address code.
- Where each leader statement is the first statement in each basic block.

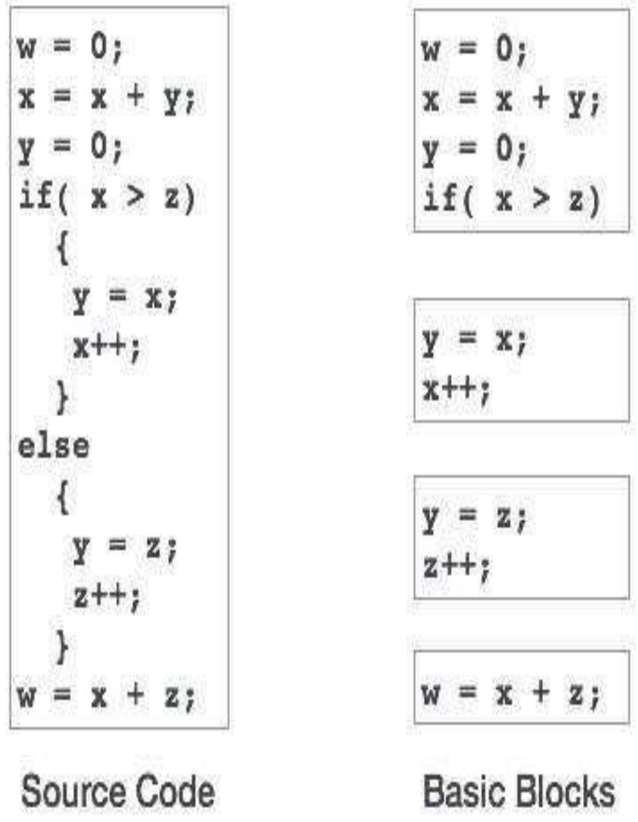
The steps to find leader statements are:

1. The **first instruction** in the intermediate code is a leader.
2. The instructions that is target OF conditional or unconditional jump statement are considered as a leader.
3. Any instructions that are just after a conditional or unconditional jump statement are considered as a leader.

The steps to determine Basic Blocks are:

- The basic block begins with leader and ends before the next leader.

Flow Graph



```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Code

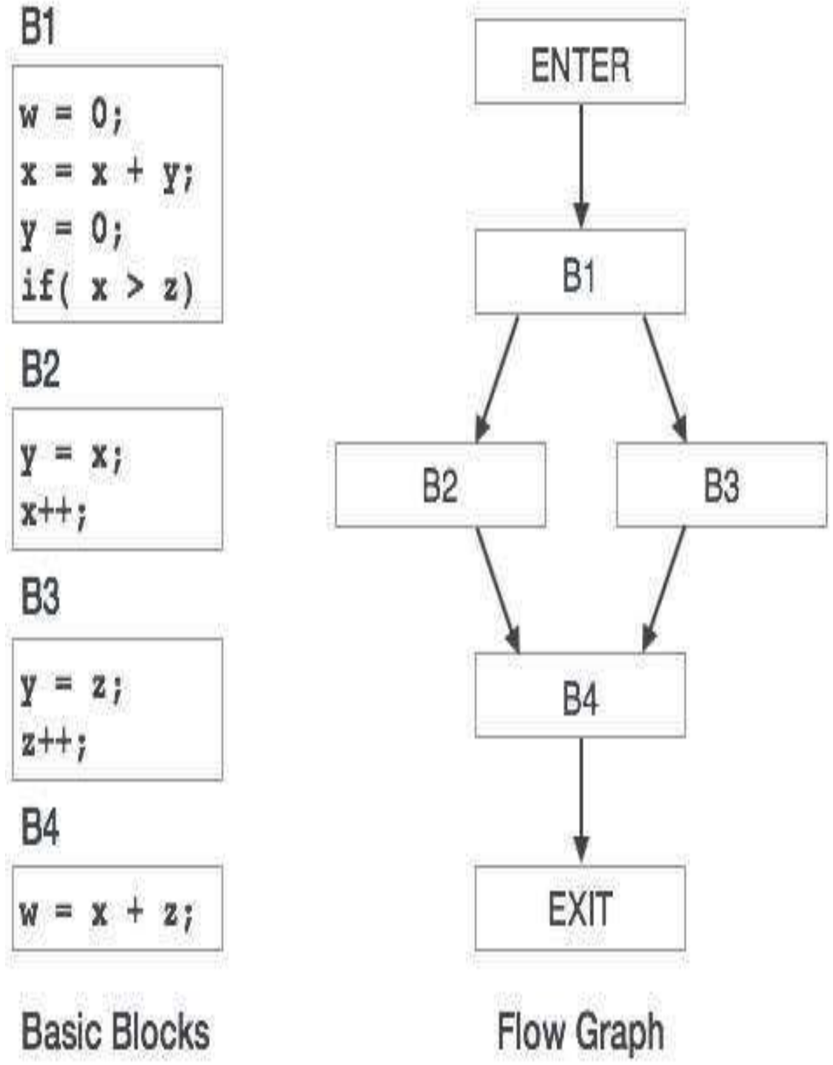
```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

```
w = x + z;
```

Basic Blocks



B1

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

B2

```
y = x;
x++;
```

B3

```
y = z;
z++;
```

B4

```
w = x + z;
```

Basic Blocks

$$\begin{tikzcd}
 & \text{ENTER} \arrow[d] & \\
 & \text{B1} \arrow[dl] \arrow[dr] & \\
\text{B2} \arrow[dr] & & \text{B3} \arrow[dl] \\
 & \text{B4} \arrow[d] & \\
 & \text{EXIT} &
\end{tikzcd}$$

Flow Graph

Flow Graph

A control flow graph **depicts how the program control is being passed among the blocks.**

It is a useful tool that helps in optimization by locating any unwanted loops in the program.

Principal sources of optimization

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.

Function-Preserving Transformations:

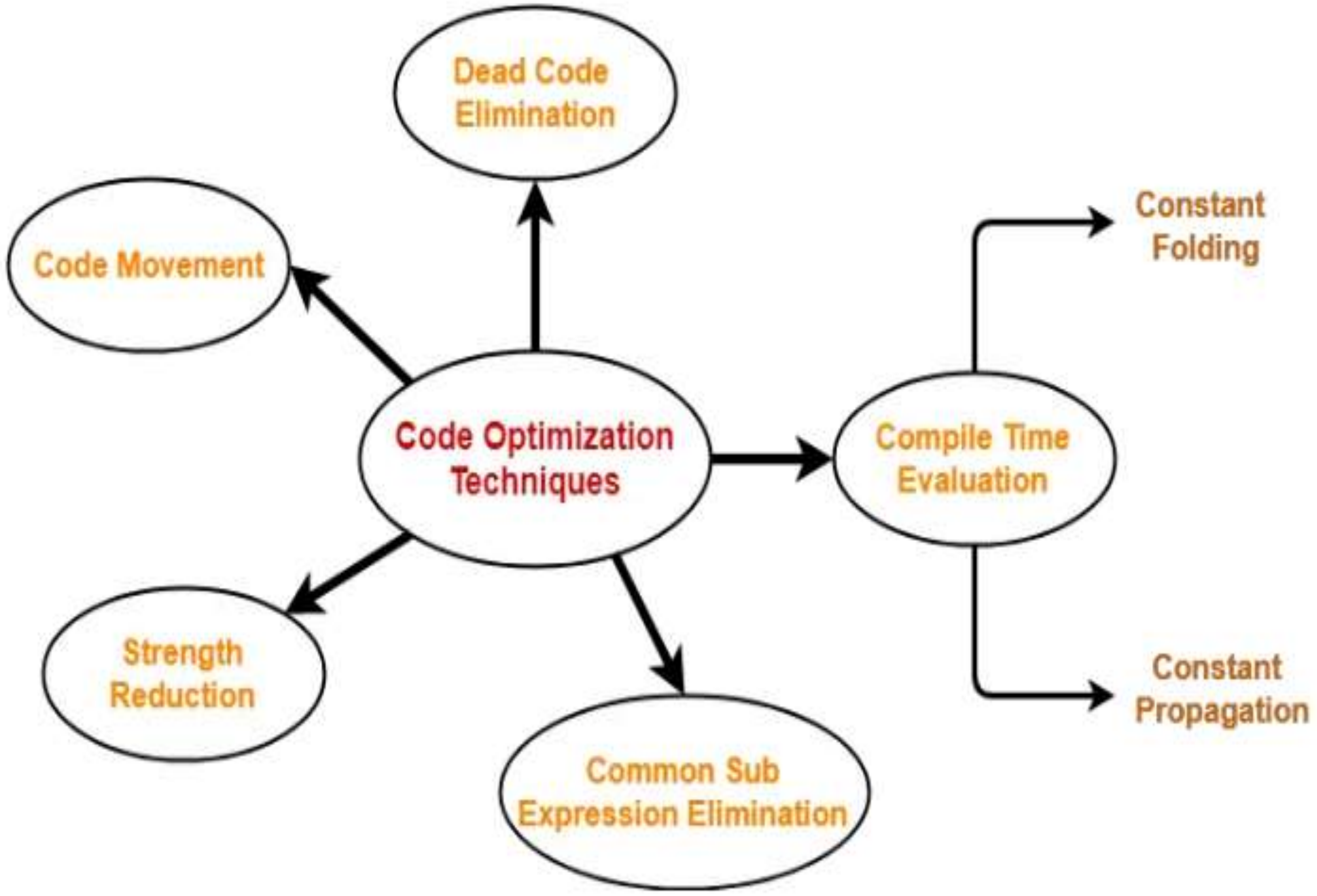
There are a number of ways in which a compiler can improve a program without changing the function.

Function preserving transformations techniques are:

or Principle sources of optimization are

- **Compile Time Evaluation**
- **Common sub-expression elimination**
- **Dead Code Elimination**
- **Code Movement**
- **Strength Reduction**

Code Optimization Techniques



1. Compile Time Evaluation

Two techniques that falls under compile time evaluation are-

A. **Constant Folding-**

B. **Constant Propagation-**

Compile Time Evaluation

Constant Folding-In this technique,

- As the name suggests, it involves folding the constants.
- The expressions that contain the operands having constant values at compile time are evaluated.
- Those expressions are then replaced with their respective results.

Ex: Circumference of Circle = (22/7) x Diameter

Here,

This technique evaluates the expression 22/7 at compile time.

The expression is then replaced with its result 3.14. as below

Circumference of Circle = (3.14) x Diameter

This saves the time at run time.

Compile Time Evaluation

Constant Propagation-_In this technique,

- If some variable has been assigned some constant value, then it replaces that variable with its constant value in the further program during compilation.
- The condition is that the value of variable must not get alter in between.

Ex:

pi = 3.14

radius = 10

Area of circle = pi x radius x radius

Here,

- This technique substitutes the value of variables ‘pi’ and ‘radius’ at compile time as below

Area of circle = 3.14 x 10 x 10

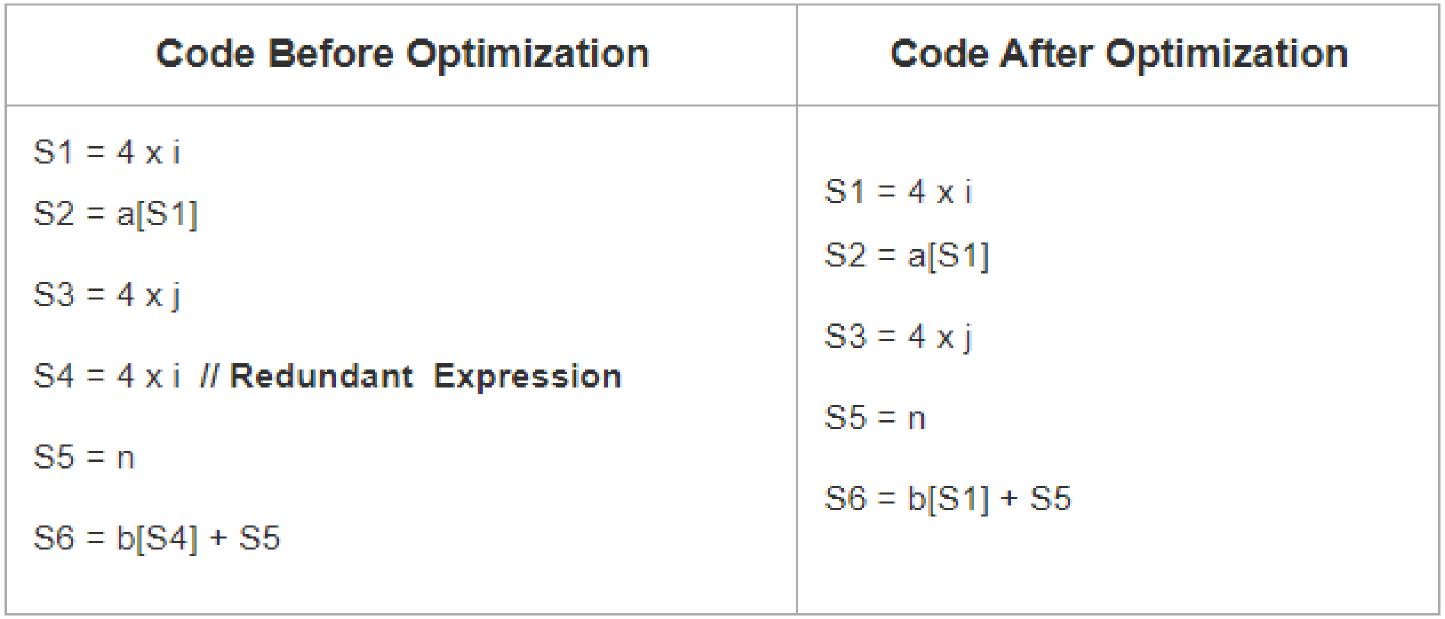
2. Common Sub expressions elimination

In this technique,

- As the name suggests, it involves eliminating the common sub expressions.
- The redundant expressions are eliminated to avoid their re-computation.
- The already computed result is used in the further program when required.

Common Sub expressions elimination

Example-



Code Before Optimization	Code After Optimization

S1 = 4 x i
S2 = a[S1]
S3 = 4 x j
S4 = 4 x i // **Redundant Expression**
S5 = n
S6 = b[S4] + S5
	S1 = 4 x i
	S2 = a[S1]
	S3 = 4 x j
	S5 = n
	S6 = b[S1] + S5

Here sub expression 4 x i is redundant expression so its result in S1 is used in expression S6= b[S1] + S5 as b[S1] instead of b[S4]

3. Dead-code elimination

In this technique,

- As the name suggests, it involves eliminating the dead code.
- The statements of the code which either never executes or are unreachable or their output is never used are eliminated.

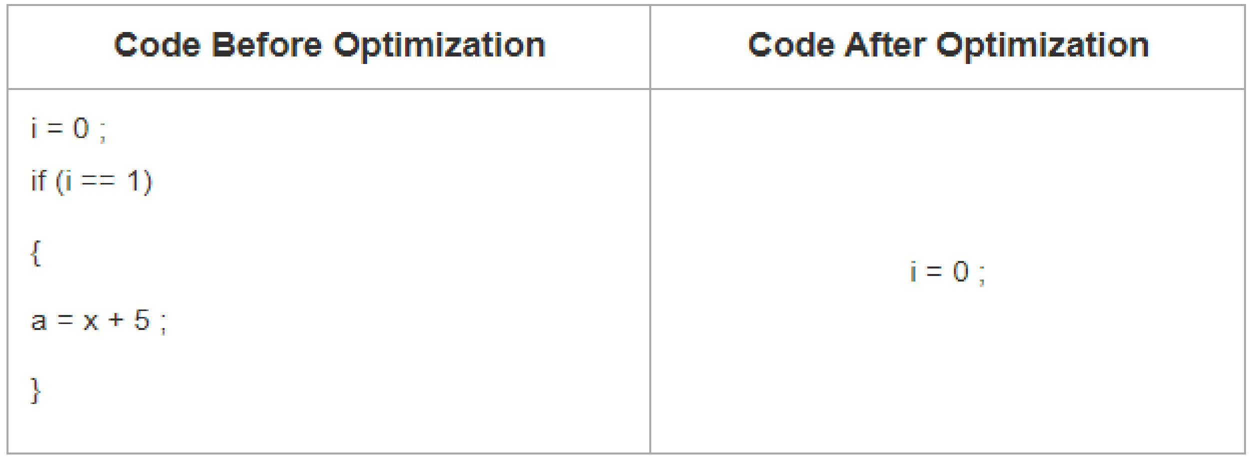
A variable is live at a point in a program if its value can be used subsequently;

otherwise, it is dead at that point.

.

Dead-code elimination

Example-



Code Before Optimization	Code After Optimization

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

Here, I value is o. The condition in if(i== 1) becomes false as I vaule is o. Therefore, if statement block is not executed. So it is dead code and is eliminated as shown in code after optimization

4. Code Movement

In this technique,

- As the name suggests, it involves movement of the code.
- The code present inside the loop is moved out if it does not matter whether it is present inside or outside.
- Such a code unnecessarily gets execute again and again with each iteration of the loop.
- This leads to the wastage of time at run time.

Code Movement

Example-

Code Before Optimization	Code After Optimization
for (int j = 0 ; j < n ; j ++)	x = y + z ;
{	for (int j = 0 ; j < n ; j ++)
x = y + z ;	{
a[j] = 6 x j;	a[j] = 6 x j;
}	}

5. Strength Reduction

In this technique,

- As the name suggests, it involves reducing the strength of expressions.
- This technique replaces the expensive and costly operators with the simple and cheaper ones.

Strength Reduction

Example-

Code Before Optimization	Code After Optimization
B = A x 2	B = A + A

Here,

- The expression "A x 2" is replaced with the expression "A + A".
- This is because the cost of multiplication operator is higher than that of addition operator.

Loop Optimization Techniques

- Loop Optimization is a machine independent optimization.
- It is the process of increasing execution speed and reducing the overheads associated with loops.
- The code optimization can be done in handling the loop optimization. Execution time is more when code is placed in inner loops. So, avoiding the code without placing in the inner loop, the running time is significantly reduced.
- So loop optimization is a technique in which code optimization performed on inner loops.

1. Code Motion (Frequency Reduction)
2. Loop unrolling
3. Loop Jamming
4. Strength Reduction

Code Motion

Code motion is a technique that moves code outside the loop. If there is some expression in the loop, if its result remains without any change even after executing the loop several times.

Before optimization:	After optimization:

while(i<100)
{
a = Sin(x)/Cos(x) + i;
i++;
}
	t = Sin(x)/Cos(x);
	
	while(i<100)
	{
	a = t + i;
	i++;
	}

Loop Unrolling

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program.

We basically **remove or reduce iterations**. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions

Before optimization:	**After optimization:**

while (i < 10000)
{
 	
arr[i]=0;
i++;
 	
}
	while (i < 5000)
	{
	
	arr[i]=0;
	i++;
	arr[i]=0;
	i++;
	
	}

Loop Jamming

we combine the bodies of two loops, or decreasing the number of loops.

Before optimization:	**After optimization:**

for (int i=0; i < 5 ; i++)
a = i + 5;
for (inti=0; i<5; i++)
b = i + 10;
	for(int i=0; i < 5; i++)
	{
	a = i + 5;
	b = i + 10;
	}

Strength Reduction

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

For example,

x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine.

Global Optimization

- Global optimization is a process within the compiler that identifies and eliminates inefficiencies in the code.
- Unlike local optimization that focuses on small, specific portions of code, global optimization looks at the entire program.
- It checks for redundant instructions, unreachable code, and other inefficiencies.
- Global optimization is often referred to as **machine-independent optimization**.

This is because it focuses on optimizing the logical structure of the program which is independent of the hardware it will eventually run on.

Techniques in of Global Optimization

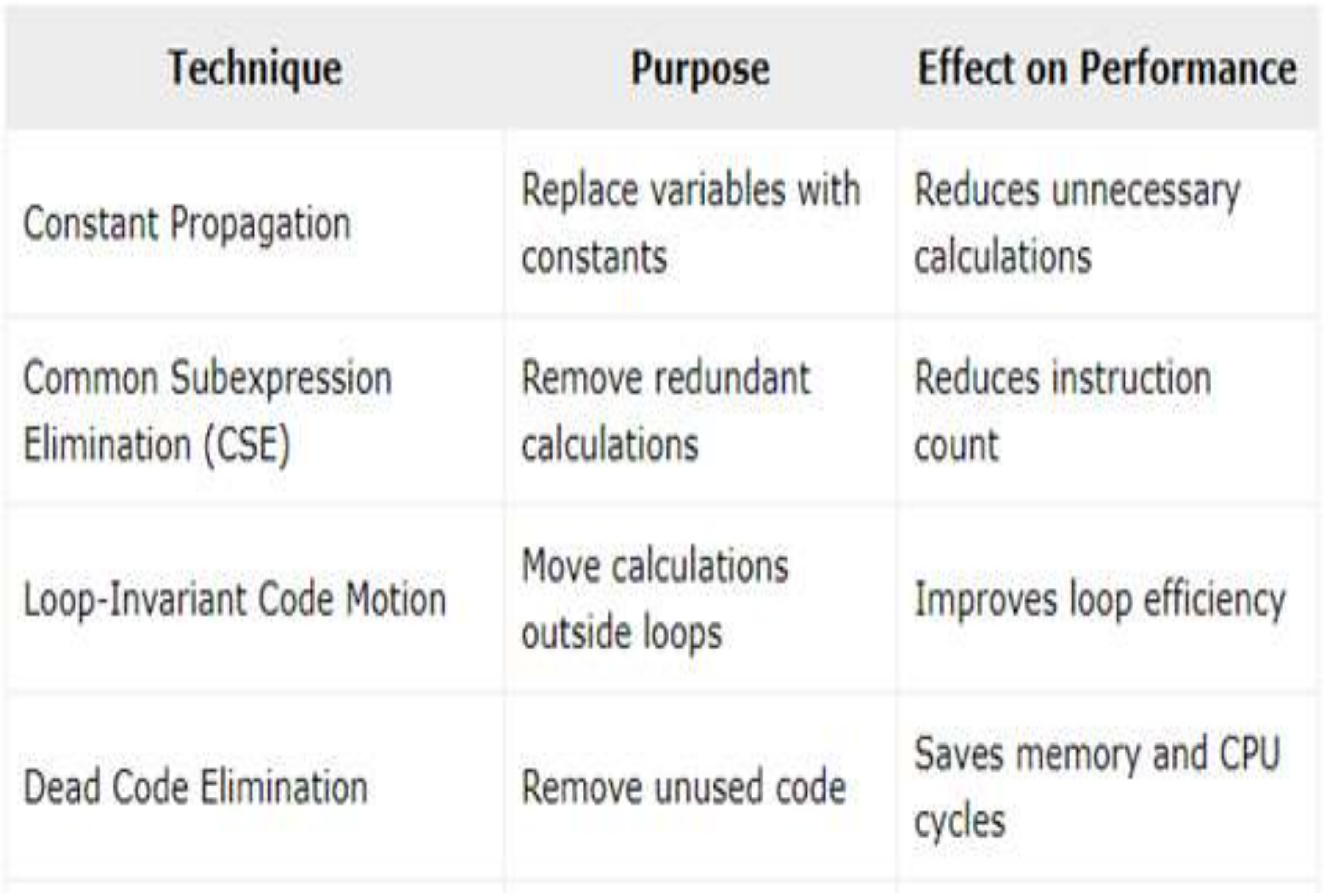
Dead Code Elimination − Removing code that has no impact on the program's output.

Common Sub expression Elimination − Identifying and reusing previously computed expressions to avoid redundant calculations.

Loop-Invariant Code Motion − Moving calculations or assignments outside loops when their values do not change during iterations.

Constant Folding − Replacing expressions with constant values wherever possible.

Comparison of Global Optimization Techniques

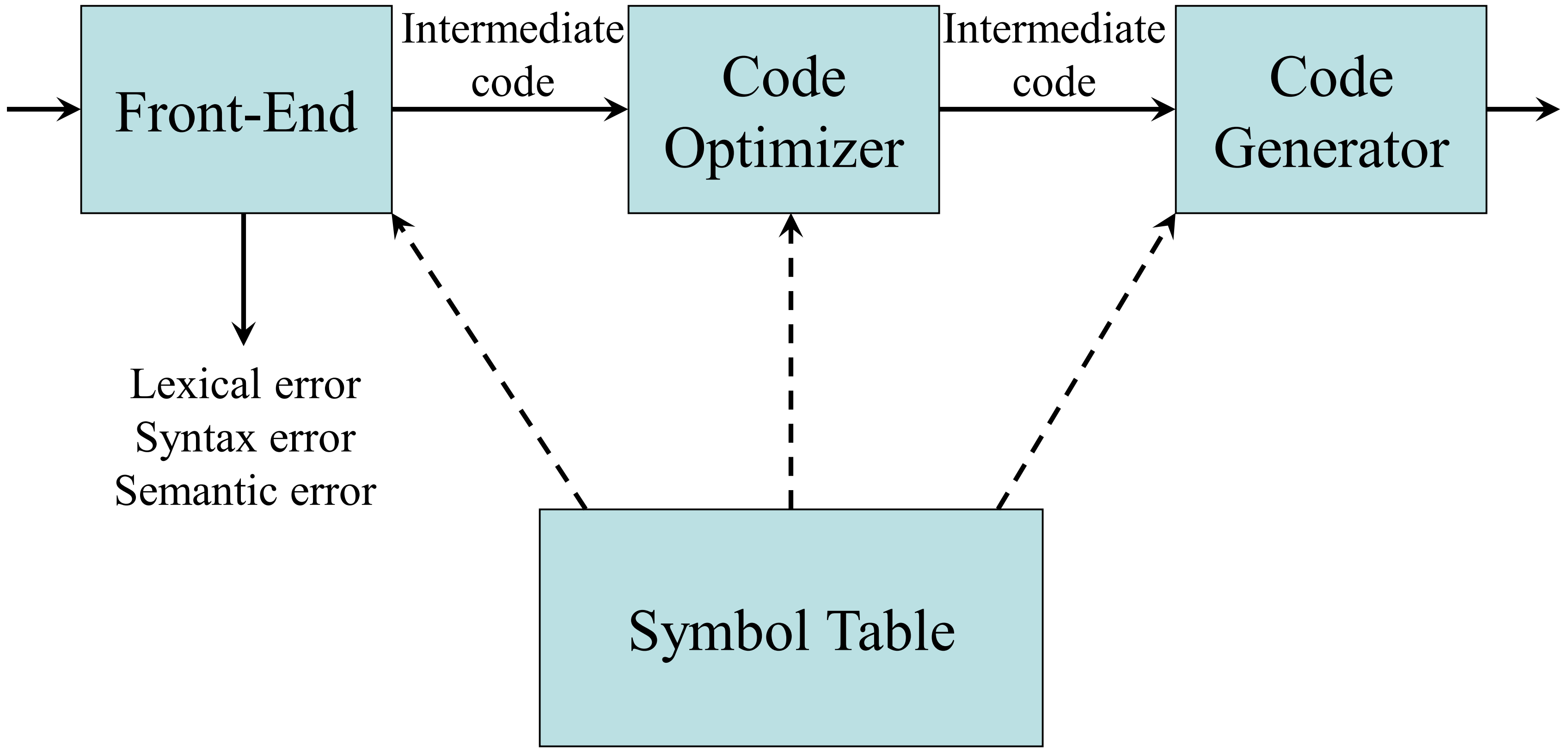


Technique	Purpose	Effect on Performance
Constant Propagation	Replace variables with constants	Reduces unnecessary calculations
Common Subexpression Elimination (CSE)	Remove redundant calculations	Reduces instruction count
Loop-Invariant Code Motion	Move calculations outside loops	Improves loop efficiency
Dead Code Elimination	Remove unused code	Saves memory and CPU cycles

Code Generation:

- Issues in Code generation phase
- DAG (Directed Acyclic Graph) Construction
- Code generation algorithm
- Simple code generator

Position of a Code Generator in the Compiler Model



Requirements

- Output code must be correct
- Must be of high quality; must make effective use of resources
- Code generator must run efficiently
- Good Vs Optimal

Issues in the design of a Code Generator

1. Input to the Code Generator
2. Target Programs
3. Memory Management
4. Instruction Selection
5. Register Allocation
6. Choice of evaluation order

1. Input to the Code Generator

- Input to code generator is intermediate code
  - Intermediate representation produced by the front end in 3 forms as

    1.Postfix form

    2. Three address code

      - Quadruple, Triple, Indirect triple

    3. Syntax trees, DAG
  - code generator also uses information from the symbol table, which holds the addresses of variables and other data objects
  - input must be free from syntactic and semantic errors

2. Target Programs

The target program is the final output of the code generator, which can be in the form

- Absolute machine language is easy to execute but lacks flexibility because it is bound to specific memory locations.
- Relocatable machine language allows parts of the program to be moved around in memory
  - Expensive
  - Flexible
- Assembly language is symbolic and needs an assembler to convert it into machine code
  - Easier

3. Memory management

- Memory management in the code generation phase involves mapping variable names to their corresponding memory locations.
- It access the symbol table, where memory addresses for variables are stored.
- A major challenge is ensuring that Code generator uses memory efficiently, avoids memory conflicts, and correctly handles dynamic memory allocation.

4. Instruction Selection

- Process of choosing the most suitable machine instructions to translate intermediate code into executable code.
- If the right instructions are not selected, the resulting code can be inefficient and slow.
- Consider **Three Address Code:**

```
 P:= Q + R
S:= P + T
```

Inefficient Instruction Selection

- **Assembly Code (Inefficient):**

  1. MOV Q, R0 (Load the value of Q into register R0)
  2. ADD R, R0 (Add the value of R to the value in R0)
  3. MOV R0, P (Store the value of R0 into the variable P)
  4. MOV P, R0 (Load the value of P back into R0)
  5. ADD T, R0 (Add the value of T to R0)
  6. MOV R0, S (Store the value of R0 into the variable S)

Here the fourth statement is redundant as the value of the P is loaded again in R0. In 3rd statement R0 already had P value . It leads to an inefficient code sequence.

Efficient Instruction Selection

- **Assembly Code (Efficient):**

  1. MOV Q, R0 (Load Q into R0)
  2. ADD R, R0 (Add R to R0)
  3. ADD T, R0 (Add T to R0)
  4. MOV R0, S (Store the final result in S)

Register Allocation

- Efficient utilization of registers is needed
- Register allocation involves two stages

➢ Register Allocation : It is selecting <u>which variables will reside in the registers</u>

➢ Register Assignment: <u>Assigning specific registers to those variables selected in Register Allocation</u>

  – Optimal assignment is an NP-complete problem

- Register pairs may be required for some instructions

Choice of Computation order

- Fewer registers may be required
- Choosing order is also NP complete

6. Evaluation Order

- The evaluation order refers to the sequence in which expressions are evaluated in the generated code.

Directed Acyclic Graph

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.

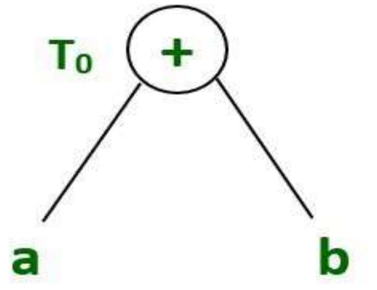
Example 1

$T_0 = a + b$ —Expression 1

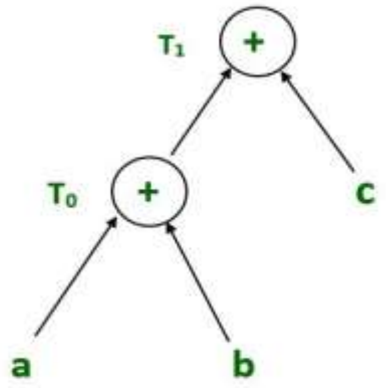
$T_1 = T_0 + c$ —-Expression 2

$d = T_0 + T_1$ ——Expression 3

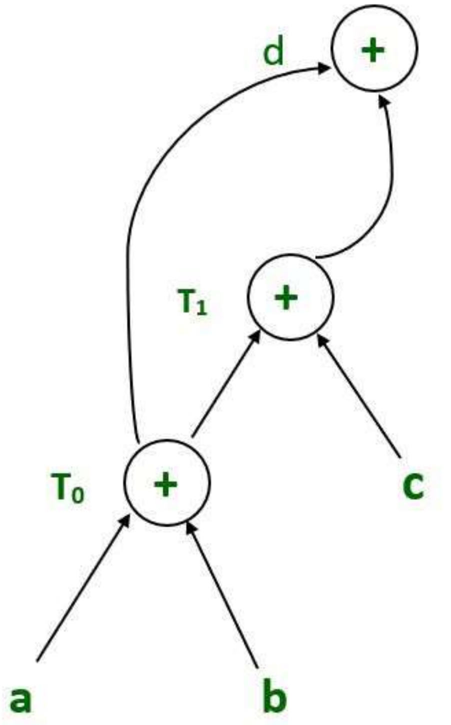
Expression 1 : $T_0 = a + b$



Expression 2: $T_1 = T_0 + c$

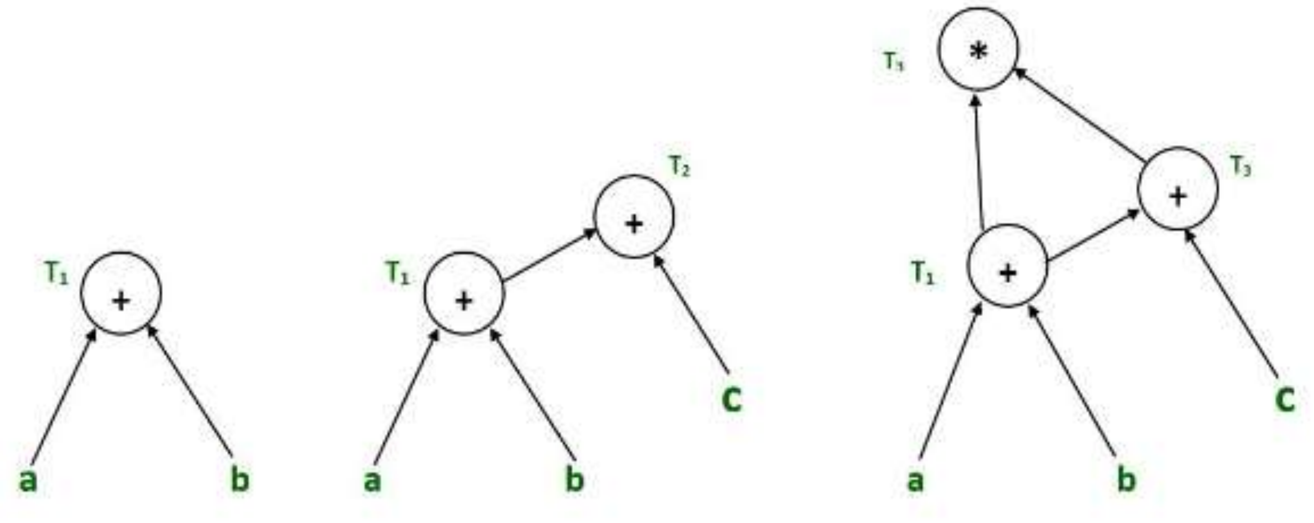


Expression 3 : $d = T_0 + T_1$



Example 2

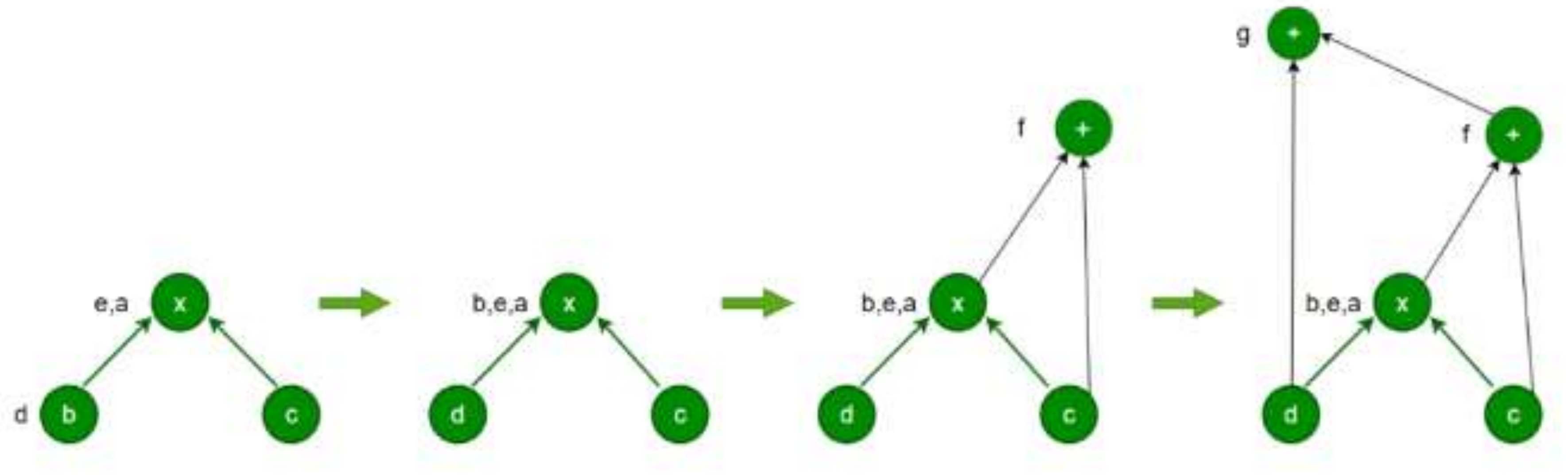
T_1 = a + b
T_2 = T1 + c
T_3 = T1 x T2



$T_1 = a + b$ $T_2 = T_1 + c$ $T_3 = T_1 * T_2$

Example 3

a = b x c
d = b
e = d x c
b = e
f = b + c
g = f + d



Here d is assigned b value and e = d x c means same as expression e = b x c [because d = b]. Thus we can take e value as a value.

Application of DAGs:

1. **Expression Optimization**: DAGs are used to optimize expressions by **identifying common subexpressions** and representing them efficiently, and reduces redundant computations.
2. **Code Generation**: DAGs assist in generating efficient code by representing intermediate code and optimizing it before translating it into machine code.
3. **Register Allocation**: DAGs aid in register allocation by identifying variables that can share the same register, minimizing the number of memory accesses.
4. **Control Flow Analysis**: DAGs help in analyzing control flow structures and optimizing the flow of control within a program.

Code Generation

The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code.

For both of them, the following two descriptors are used:

•**Register descriptor :** Register descriptor is used to inform the code generator about the availability of registers. **Register descriptor keeps track of values(variables) stored in each register.** Whenever a new register is required during code generation, this descriptor is consulted for register availability.

•**Address descriptor** : Values of the names (identifiers) used in the program might be stored at different locations while in execution. **Address descriptors are used to keep track of memory locations where the values(identifiers) are stored**. These locations may include CPU registers, heaps, stacks, memory address or a combination of the mentioned locations.

A code-generation algorithm

Code generator uses getReg function to determine the status of available registers and the location of name values(variables).

getReg works as follows:

- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions

The algorithm takes as input a sequence of three-address statements constituting a basic block. (**This explanation, no need to write in exam**)

For each three-address statement of the form x : = y op z, perform the following actions:

1. Call the function getreg to determine the location L where the result of the computation y op z should be stored.
2. Determine the location of y by Consulting the address descriptor of y

   If y is not present in the L location, generate the instruction **MOV y' , L** to place a copy of y in L.

   (Note y' means it refers to address descriptor of y)

3. The present location of Z is determined using the step-2 and the instruction is generated as **OP z' , L**

4. Now L contains the value of y op z, i.e. is assigned to x

   Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

5. If the current values of y or z have no next uses, then update the descriptor to remove y and z , after execution of x : = y op z , those registers will no longer contain y or z.

The Code Generation Algorithm

Note: write this in exam

For each statement *x* := *y* op *z*

1. Set location *L* = *getreg*(*y*, *z*)
2. If $y \notin L$ then generate

   `MOV` y’ , *L*

   where *y’* denotes one of the locations where the value of *y* is available (choose register if possible)
3. Generate

   `OP` z’ , *L*

   -where *z’* is one of the locations of *z*;
   -Update address descriptor of *x* to indicate it is in *L*

   -If L is a register update its Register descriptor that it contains value of x
4. If *y* and/or *z* has no next use and is stored in register, update register descriptors to remove *y* and/or *z*

The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three address code sequence:

1. t := a-b
2. u:= a-c
3. v:= t +u
4. d:= v+u

Statement	Code Generated	Register descriptor Register empty	Address descriptor
t:= a - b	MOV a, R0 SUB b, R0	R0 contains a R0 contains t	t in R0
u:= a - c	MOV a, R1 SUB c, R1	R0 contains t R1 contains a R1 contains u	t in R0 u in R1
v:= t + u	ADD R1, R0	R0 contains v R1 contains u	v in R0 u in R1
d:= v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory