

Run-time Environments

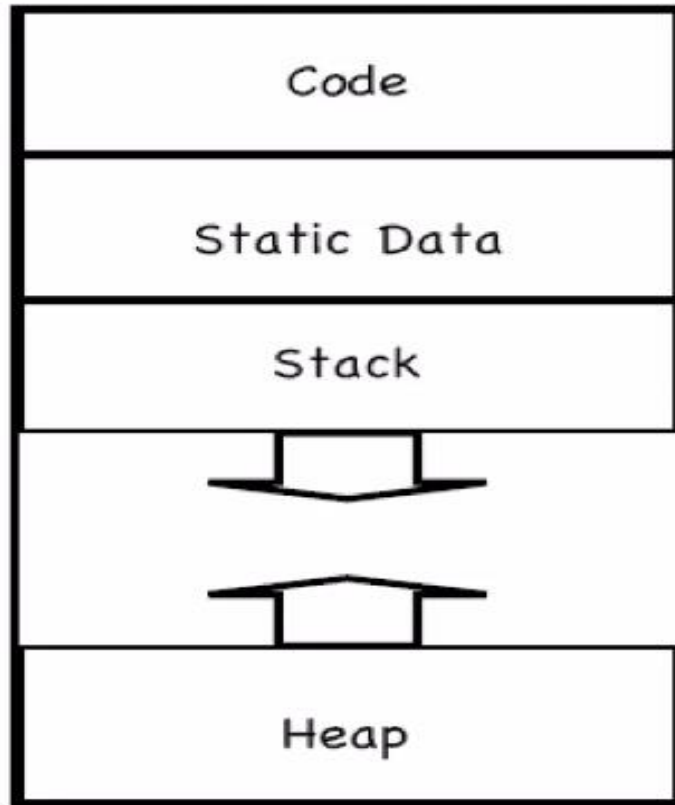
Deekshika Tomar

106119026

Run-time Environment

- Compiler must cooperate with OS and other system software to support implementation of different abstractions (names, scopes, bindings, data types, operators, procedures, parameters, flow-of-control) on the target machine
- Compiler does this by **Run-Time Environment** in which it assumes its target programs are being executed
- Run-Time Environment deals with
 - Layout and allocation of storage
 - Access to variable and data
 - Linkage between procedures
 - Parameter passing
 - Interface to OS, I/O devices etc

Organization of storage



- Fixed-size objects can be placed in predefined location
- The heap and the stack need room to grow, however.

Run-Time Environments

- How do we allocate the space for the generated target code and the data object of our source programs?
- The places of the data objects that can be determined at **compile time** will be **allocated statically**.
- But the places for the some of data objects will be *allocated at run-time*.
- The allocation and de-allocation of the data objects is managed by the **run-time support package**.
 - run-time support package is loaded together with the generated target code.
 - the structure of the run-time support package depends on the semantics of the programming language (especially the semantics of procedures in that language).

Activation Records

- Information needed by a single execution of a procedure is managed using a contiguous block of storage called **activation record**.
- An activation record is allocated when a procedure is entered, and it is de-allocated when that procedure exited.
- **Size of each field** can be determined at compile time (Although actual location of the activation record is determined at run-time).
 - Except that if the procedure has a local variable and its size depends on a parameter, its size is determined at the run time.

Activation Records (cont.)

return value	← 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.
actual parameters	← The field for actual parameters is used by the calling procedure to supply parameters to the called procedure.
optional control link	← The optional control link points to the activation record of the caller.
optional access link	← The optional access link is used to refer to nonlocal data held in other activation records.
saved machine status	← The field for saved machine status holds information about the state of the machine before the procedure is called.
local data	← The field of local data holds data that local to an execution of a procedure..
temporaries	← Temporary variables is stored in the field of temporaries.

Storage allocation strategies

- Static allocation
- Stack allocation
- Heap Allocation

Static allocation

- Lays out storage for all data objects at compile time.
- Constraints:
 - size of object must be known and alignment requirements must be known at compile time.
- No recursion.
- No dynamic data structure

Stack allocation

- Stack allocation manages the run time storage as a stack
- The activation record is pushed on as a function is entered.
- The activation record is popped off as a function exits.
- Constraints:
- values of locals cannot be retained when an activation ends.
- A called activation cannot outlive a caller.

Heap allocation

- Heap allocation -- allocates and deallocates storage as needed at runtime from a data area called as heap.
- Does not require the activation of procedures to be LIFO.
- Requires true dynamic memory management.

How is stack memory managed?

- Everything must be done by the compiler.
- What makes this happen is known as calling sequence (how to implement a procedure call).
- A calling sequence allocates an activation record and enters information into its fields (push the activation record).
- On the opposite of the calling sequence is the return sequence.
- Return sequence restores the state of the machine so that the calling procedure can continue execution.

Calling sequence

- The caller evaluates actuals and push the actuals on the stack
- The caller saves return address(pc) the old value of sp into the stack
- The caller increments the sp
- The callee saves registers and other status information
- The callee initializes local variables & begin execution.

Return sequence

- The callee places a return value next to the activation record of the caller.
- The callee restores other registers and sp and return (jump to pc).
- The caller copies the return value to its activation record.

Parameter Passing

- Communication between procedure
 - Through –non- local names
 - Parameters
- Methods
 - Call-by-value
 - Call-by-reference
 - Copy-restore
 - Call-by-name

Questions

Consider the following statements.

S1: The sequence of procedure calls corresponds to a preorder traversal of the activation tree.

S2: The sequence of procedure returns corresponds to a postorder traversal of the activation tree.

Which one of the following options is correct?

A S1 is true and S2 is false

B S1 is false and S2 is true

C S1 is true and S2 is true

D S1 is false and S2 is false



Which languages necessarily need heap allocation in the runtime environment?

A

Those that support recursion

B

Those that use dynamic scoping

C

Those that allow dynamic data structures

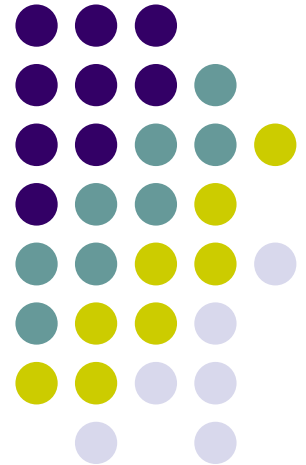


D

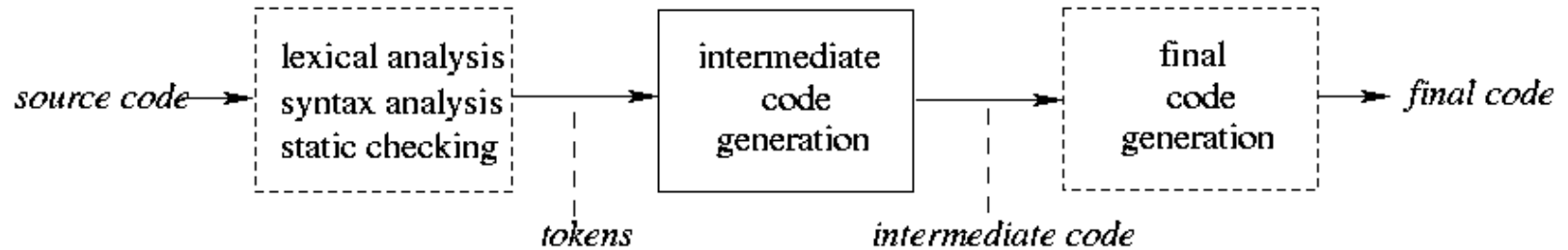
Those that use global variables

Intermediate Code Generation

Thota Pavan Kumar
106119136



Overview

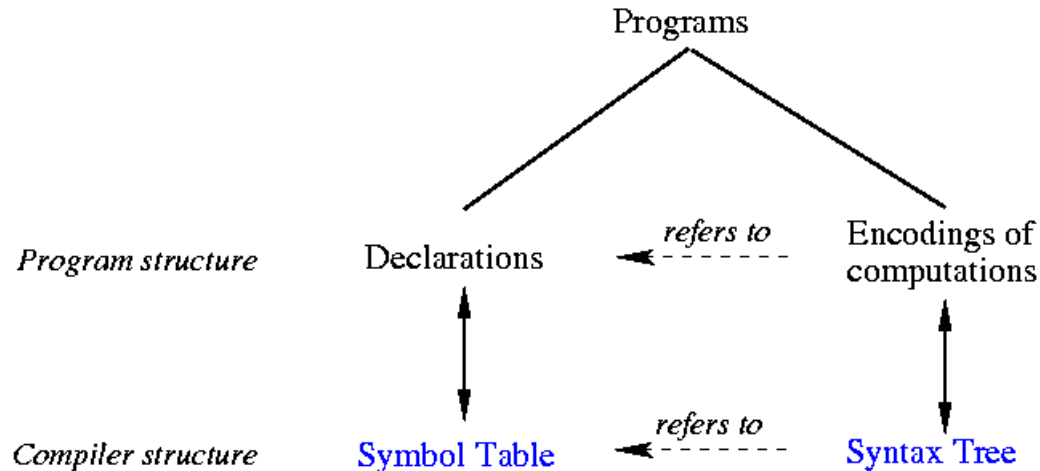


- Intermediate representations span the gap between the source and target languages:
 - closer to target language;
 - (more or less) machine independent;
 - allows many optimizations to be done in a machine-independent way.

Intermediate Representations

- High Level Representations:
 - closer to the source language
 - easy to generate from an input program.
 - code optimizations may not be straightforward.
 - Abstract Syntax Trees fall under these category.
- Low Level Representations :
 - closer to the target machine.
 - easier for optimizations, final code generation.
 - Three Address Code falls under these category.

Abstract Syntax Trees



An Abstract syntax tree shows the structure of a program by abstracting away irrelevant details from a parse tree.

- Each node represents a computation to be performed;
- The children of the node represents what that computation is performed on.

Abstract Syntax Trees

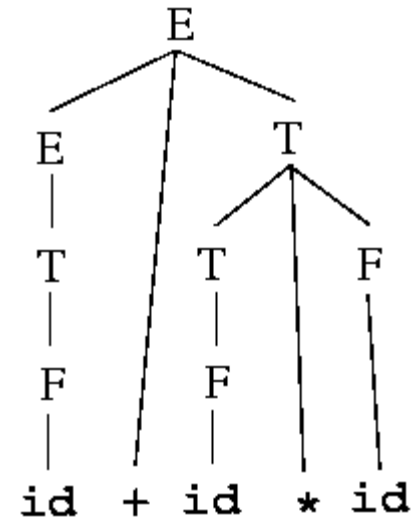
Grammar :

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

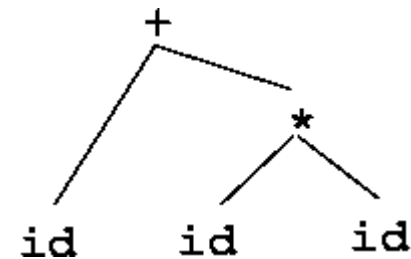
$F \rightarrow (E) \mid \text{id}$

Parse tree:



Input: `id + id * id`

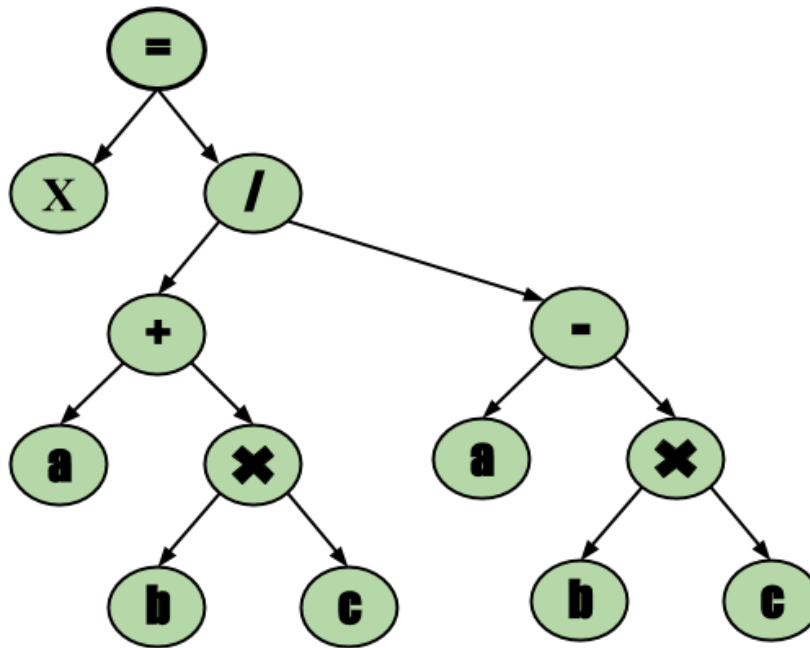
Syntax tree:



Abstract Syntax Trees

Example: $X = (a + (b * c)) / (a - (b * c))$

Operator Root



Postfix Notation

- *In postfix notation, the operator appears after the operands, i.e., the operator between operands is taken out & is attached after operands.*

Example:

Translate $a * d - (b + c)$ into Postfix form

Sol: $ad * bc + -$

Three-Address Code

- A statement involving no more than three references(two for operands and one for result) is known as a three address statement.
- A sequence of three address statements is known as a three address code.
- Sometimes a statement might contain less than three references but it is still called a three address statement.
- In a three address code there is at most one operator at the right side of an instruction

Three-Address Code

Example: The three address code for the expression

$a + b * c + d$ is as follows:

$T_1 = b * c ;$

$T_2 = a + T_1 ;$

$T_3 = T_2 + d ;$

Where T_1 , T_2 , T_3 are temporary variables.

Three-Address Code

Example:

```
for(i = 1; i<=10; i++)  
{ a[i] = x * 5; }
```

Solution:

i=1

L : T1 = x * 5;

 T2 = &a;

 T3 = sizeof(int);

 T4 = T3 * i;

 T5 = T2 + T4;

 *T5 = T1;

 i = i+1;

If i <= 10 goto L

Types of Three-Address Code

There are 3 representations of three address code namely

1. Quadruple
2. Triples
3. Indirect Triples

Quadruple

- It is a structure which consists of 4 fields namely op, arg1, arg2 and result. op denotes the operator and arg1 and arg2 denotes the two operands and result is used to store the result of the expression.

Advantages:

Easy to rearrange code for global optimization.

One can quickly access value of temporary variables using symbol table.

Disadvantages:

Contain lot of temporaries.

Temporary variable creation increases time and space complexity.

Quadruple

Example – Consider expression $a = b * -c + b * -c$.

The three address code is:

$t1 = \text{uminus } c$

$t2 = b * t1$

$t3 = \text{uminus } c$

$t4 = b * t3$

$t5 = t2 + t4$

$a = t5$

Quadruple

#	Op	Arg1	Arg2	Result
(0)	uminus	c		t1
(1)	*	t1	b	t2
(2)	uminus	c		t3
(3)	*	t3	b	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Quadruple representation

Triples

- This representation doesn't make use of extra temporary variable to represent a single operation instead when a reference to another triple's value is needed, a pointer to that triple is used. So, it consist of only three fields namely op, arg1 and arg2.

Disadvantage

- Temporaries are implicit and difficult to rearrange code.
- It is difficult to optimize because optimization involves moving intermediate code. When a triple is moved, any other triple referring to it must be updated also. With help of pointer one can directly access symbol table entry.

Triples

Example: Consider expression $a = b * - c + b * - c$

#	Op	Arg1	Arg2
(0)	uminus	c	
(1)	*	(0)	b
(2)	uminus	c	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	a	(4)

Triples representation

Indirect Triples

- This representation makes use of pointer to the listing of all references to computations which is made separately and stored.
- Its similar in utility as compared to quadruple representation but requires less space than it.
- Temporaries are implicit and easier to rearrange code.

Indirect Triples

Example: Consider expression $a = b * -c + b * -c$

List of pointers to table

#	Op	Arg1	Arg2
(14)	uminus	c	
(15)	*	(14)	b
(16)	uminus	c	
(17)	*	(16)	b
(18)	+	(15)	(17)
(19)	=	a	(18)

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

Indirect Triples representation

Questions

In the context of compilers, which of the following is/are NOT an intermediate representation of the source program?

- ☐ A Three address code
- ☐ B Abstract Syntax Tree (AST)
- ☐ C Control Flow Graph (CFG)
- ☒ Symbol table

GATE CSE 2021 SET-2 Compiler Design

CODE OPTIMIZATION

Tejavath Durga Nayak
106119132

- The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives :
 - The optimization must be correct, it must not, in any way, change the meaning of the program.
 - Optimization should increase the speed and performance of the program.
 - The compilation time must be kept reasonable.
 - The optimization process should not delay the overall compiling process.

- **WHEN TO OPTIMIZE?**

- Optimization of the code is often performed at the end of the development stage since it reduces readability and adds code that is used to increase the performance.

- **WHY OPTIMIZE?**

- Optimizing an algorithm is beyond the scope of the code optimization phase. So the program is optimized. And it may involve reducing the size of the code. So optimization helps to:
 - Reduce the space consumed and increases the speed of compilation.
 - Manually analyzing datasets involves a lot of time. Hence we make use of software like Tableau for data analysis. Similarly manually performing the optimization is also tedious and is better done using a code optimizer.
 - An optimized code often promotes re-usability.

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

- 
- 1. Compile Time Evaluation:

(i) $A = 2 * (22.0 / 7.0) * r$
time. Perform $2 * (22.0 / 7.0) * r$ at compile
(ii) $x = 12.4$
 $y = x / 2.3$
time. Evaluate $x / 2.3$ as $12.4 / 2.3$ at compile



2. Variable Propagation:

```
//Before Optimization
```

```
c = a * b
```

```
x = a
```

```
till
```

```
d = x * b + 4
```

```
//After Optimization
```

```
c = a * b
```

```
x = a
```

```
till
```

```
d = a * b + 4
```

3. Constant Propagation:

- If the value of a variable is a constant, then replace the variable with the constant. The variable may not always be a constant.
- **Example:**

(i) $A = 2 * (22.0 / 7.0) * r$

Performs $2 * (22.0 / 7.0) * r$ at compile time.

(ii) $x = 12.4$

$y = x / 2.3$

Evaluates $x / 2.3$ as $12.4 / 2.3$ at compile time.

(iii) `int k=2;`

`if(k) go to L3;`

It is evaluated as :

`go to L3 (Because k = 2 which implies condition is always true)`

4. Constant Folding:

- Consider an expression : $a = b \text{ op } c$ and the values b and c are constants, then the value of a can be computed at compile time.

- **Example:**

```
#define k 5  
x = 2 * k  
y = k + 5
```

This can be computed at compile **time** and the values of x and y are :

```
x = 10  
y = 10
```

- ***Note: Difference between Constant Propagation and Constant Folding:***
- In Constant Propagation, the variable is substituted with its assigned constant where as in Constant Folding, the variables whose values can be computed at compile time are considered and computed.

5. Copy Propagation:

- It is extension of constant propagation.
- After a is assigned to x, use a to replace x till a is assigned again to another variable or value or expression.
- It helps in reducing the compile time as it reduces copying.
- **Example :**

```
//Before Optimization
```

```
c = a * b
```

```
x = a
```

```
till
```

```
d = x * b + 4
```

```
//After Optimization
```

```
c = a * b
```

```
x = a
```

```
till
```

```
d = a * b + 4
```

6. COMMON SUB EXPRESSION ELIMINATION:


- In the above example, $a*b$ and $x*b$ is a common sub expression.

7. DEAD CODE ELIMINATION:

- Copy propagation often leads to making assignment statements into dead code.
- A variable is said to be dead if it is never used after its last definition.
- In order to find the dead variables, a data flow analysis should be done.

- **8. UNREACHABLE CODE ELIMINATION:**

- First, Control Flow Graph should be constructed.
- The block which does not have an incoming edge is an Unreachable code block.
- After constant propagation and constant folding, the unreachable branches can be eliminated.



```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int num;
```

```
    num=10;
```

```
    cout << "GFG!";
```

```
    return 0;
```

```
    cout << num; //unreachable code
```

```
}
```

```
//after elimination of unreachable code
```

```
int main() {
```

```
    int num;
```

```
    num=10;
```

```
    cout << "GFG!";
```

```
    return 0;
```

9. FUNCTION INLINING:

- Here, a function call is replaced by the body of the function itself.
- This saves a lot of time in copying all the parameters, storing the return address, etc.

10. FUNCTION CLONING:

- Here, specialized codes for a function are created for different calling parameters.
- **Example:** Function Overloading

11. INDUCTION VARIABLE AND STRENGTH REDUCTION:

- An induction variable is used in the loop for the following kind of assignment $i = i + \text{constant}$. It is a kind of Loop Optimization Technique.
- Strength reduction means replacing the high strength operator with a low strength.

LOOP OPTIMIZATION TECHNIQUES:

1. Code Motion or Frequency Reduction:

- The evaluation frequency of expression is reduced.
- The loop invariant statements are brought out of the loop.
- **Example:**

```
a = 200;
while(a>0)
{
    b = x + y;
    if (a % b == 0)
        printf("%d", a);
}
```

//This code can be further optimized as

```
a = 200;
b = x + y;
while(a>0)
{
    if (a % b == 0)
        printf("%d", a);
}
```

2. Loop Jamming:

- Two or more loops are combined in a single loop. It helps in reducing the compile time.

- ***Example:***

```
// Before loop jamming
for(int k=0;k<10;k++)
{
    x = k*2;
}

for(int k=0;k<10;k++)
{
    y = k+3;
}

//After loop jamming
for(int k=0;k<10;k++)
{
    x = k*2;
    y = k+3;
}
```

3. Loop Unrolling:

- It helps in optimizing the execution time of the program by reducing the iterations.
- It increases the program's speed by eliminating the loop control and test instructions.
- **Example:**

//Before Loop Unrolling

```
for(int i=0;i<2;i++)  
{  
    printf("Hello");  
}
```

//After Loop Unrolling

```
printf("Hello");  
printf("Hello");
```




COMMON SUBEXPRESSION ELIMINATION



COMMON SUBEXPRESSION ELIMINATION

- Common Subexpression Elimination (CSE) is a compiler optimization technique that seeks to improve program performance by identifying and eliminating redundant computations of expressions that have already been computed before in the program.
- The basic idea of CSE is to search for pairs of expressions that are the same, meaning they have the same operands and operators. When such pairs of expressions are found, one of them is retained, and the other is replaced by a reference to the first expression.
- This avoids recomputing the same expression and reduces the total number of operations that must be performed to execute the program, leading to faster execution times which improves the efficiency of the program.

COMMON SUBEXPRESSION ELIMINATION

There are two types of common subexpression elimination(CSE)

- Local CSE
 - Within a basic block
- Global CSE
 - Across multiple basic blocks

LOCAL CSE

- Local Common Subexpression Elimination (CSE) is a compiler optimization technique that eliminates redundant computations of expressions within a single basic block.
- A basic block is a sequence of instructions that always execute in order and do not include any jumps or branches.
- The basic idea of local CSE is to search for identical expressions within a basic block, and replace redundant expressions with a temporary variable that holds the result of the expression.
- When the expression is encountered again in the same basic block, the temporary variable is used instead of recomputing the expression. This eliminates the redundant computation and improves the efficiency of the program.

LOCAL CSE

```
t6 := 4 * i  
x := a[t6]  
t7 := 4*i  
t8 := 4*j  
t9 := a[t8]  
a[t7]:= t9  
t10 := 4*j  
a[t10]:= x
```



```
t6 := 4 * i  
x := a[t6]  
t8 := 4*j  
t9 := a[t8]  
a[t6]:= t9  
a[t8]:= x
```

```
t11 := 4*i  
x := a[t11]  
t12 := 4*i  
t13 := 4*n  
t14 := a[t13]  
a[t12]:= t14  
t15 := 4*n  
a[t15]:= x
```

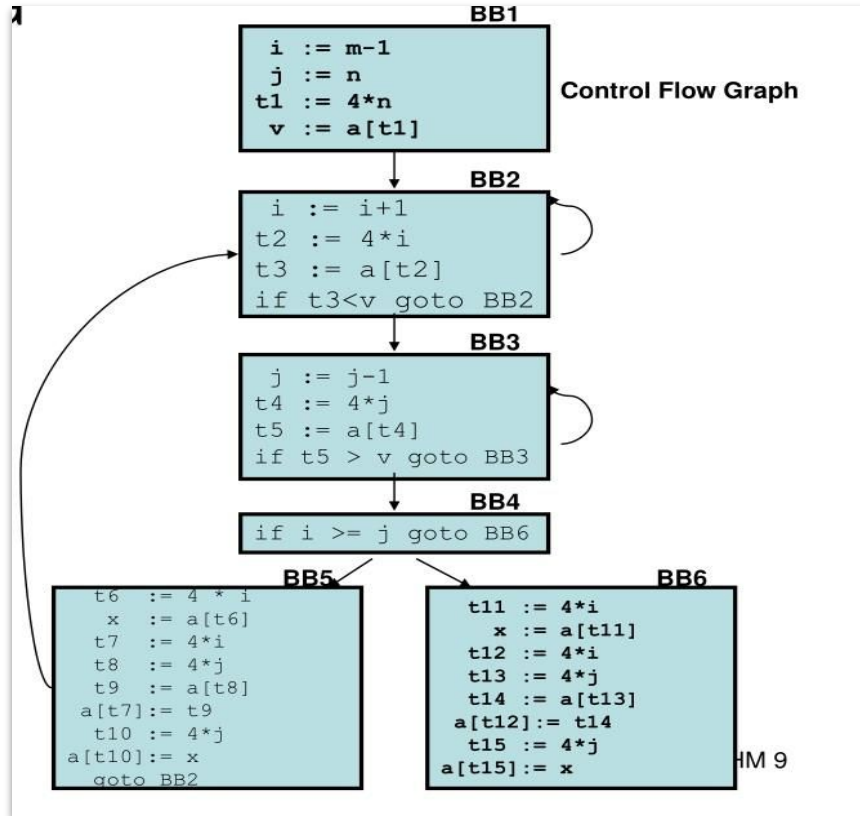


```
t11 := 4*i  
x := a[t11]  
t13 := 4*n  
t14 := a[t13]  
a[t11]:= t14  
a[t13]:= x
```

GLOBAL CSE

- Global Common Subexpression Elimination (CSE) is a compiler optimization technique that eliminates redundant computations of expressions across multiple basic blocks.
- It works by identifying identical expressions that are computed in different parts of the program and replacing them with temporary variables that hold the result of the expression.
- The basic idea of global CSE is to search for identical expressions across different basic blocks, and replace redundant expressions with a temporary variable that holds the result of the expression.
- When the expression is encountered again in a different basic block, the temporary variable is used instead of recomputing the expression.

GLOBAL CSE



```
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 BB2
```



```
x := a[t2]
t9 := a[t4]
a[t2]:= t9
a[t4]:= x
goto BB2
```



```
x := t3
t9 := t5
a[t2]:= t9
a[t4]:= x
goto BB2
```

WHICH IS BETTER?

- Depends on the program and the characteristics of the expressions being optimized.
- Local CSE is typically faster and simpler than global CSE because it only considers expressions within a single basic block.
- Local CSE is also less likely to introduce new dependencies between basic blocks.
- Global CSE, on the other hand, can eliminate redundant computations across different basic blocks, which can lead to greater improvements in program efficiency than local CSE.
- Global CSE can also reduce the overall number of instructions executed in the program, which can lead to faster execution times.

WHICH IS BETTER?

- Global CSE is also more complex than local CSE and can introduce new dependencies between basic blocks leading to longer instruction sequences and decreased performance.
- Global CSE also requires more analysis of the program, which can increase the time and resources required to perform the optimization.
- So, it is best to use a combination of local and global CSE depending on the specific program and the expressions being optimized so that it can take advantage of the benefits of both techniques while minimizing their drawbacks.

QUESTIONS

Consider the following ANSI C code segment .Common sub-expression elimination (CSE) optimization is applied on the code. The number of addition and the dereference operations (of the form $y \rightarrow f1$ or $y \rightarrow f2$) in the optimized code, respectively, are:

```
z=x + 3 + y->f1 + y->f2;
```

```
for (i = 0; i < 200; i = i + 2)
```

```
{
```

```
if (z > i)
```

```
{
```

```
p = p + x + 3;
```

```
q = q + y->f1;
```

```
} else
```

```
{
```

```
p = p + y->f2;
```

```
q = q + x + 3;
```

```
}
```

```
}
```

a) 403 & 102

b) 203 & 2

c) 303 & 102

d) 303 & 2

Ans :- D

QUESTIONS

Which one of the following is FALSE?

- a) A basic block is a sequence of instructions where control enters the sequence at the beginning and exits at the end.
- b) Available expression analysis can be used for common subexpression elimination.
- c) Live variable analysis can be used for dead code elimination.
- d) $x = 4 * 5 \Rightarrow x = 20$ is an example of common subexpression elimination.

Ans :- D

QUESTIONS

Which of the following statements is true about CSE?

- a) CSE can only be performed on arithmetic expressions.
- b) CSE can only be performed on variables that have been initialized.
- c) CSE can introduce new dependencies between basic blocks.
- d) CSE is only useful for reducing code size, not improving performance.

Ans :- C

QUESTIONS

In the context of CSE, what is meant by the term "availability"?

- a) The ability to access external resources
- b) The ability to reuse a previously computed value
- c) The ability to allocate memory for a value
- d) The ability to define a new variable

Ans :- B

QUESTIONS

Which of the following statements is true about Global CSE?

- a) Global CSE cannot introduce new dependencies between basic blocks.
- b) Global CSE is always better than Local CSE.
- c) Global CSE can only eliminate expressions that have already been computed in a previous basic block.
- d) Global CSE can introduce new dependencies between basic blocks.

Ans :- D