Compiler Module 5 Important Questions

- Compiler Module 5 Important Questions
  - 1. List out the examples of function preserving transformations
  - 2. What are the actions performed by a simple code generator for a typical three address statement of the form x: = y op z.
  - 3. What is the role of peephole optimization in the compilation process
    - Peephole optimization
    - Purpose of peephole
    - Steps in peephole optimization
  - 4. What are the issues in the design of a code generator
    - Input to code generator
    - Target Program
    - Memory Management
    - Instruction Selection
    - Register Allocation
    - Evaluation order
  - 5. Describe the principal sources of optimization
    - Function Preserving transformations
      - a. Common subexpression elimination
      - b. Copy propagation
      - c. Dead code elimination
      - d. Constant folding
    - Loop optimization
      - a. Code motion
      - b. Induction variable elimination
      - c. Reduction in strength
      - d. Loop Jamming
      - e. Loop Unrolling
  - 6. Illustrate the optimization of basic blocks with examples
    - Structure preserving transformation

1. List out the examples of function preserving transformations

- Common subexpression elimination
- Copy Propagation
- Dead code elimination
- Constant folding

2. What are the actions performed by a simple code generator for a typical three address statement of the form x: = y op z.

- **Find Storage Location (getReg Function)**:
    - **Call `getReg` for Result Storage**:
    - Determine the appropriate location (usually a register) to store the result of `y op z`. This location is referred to as `L`.
- **Determine Current Location of `y`**:
    - **Check Address Descriptor for `y`**:
        - An address descriptor keeps track of where each variable's current value is stored (in memory or registers).

        - Prefer using the register if `y` is stored in both memory and a register due to faster access speeds.
    - **Generate Move Instruction if Necessary**:
        - If `y` is not already in the preferred location `L`, generate an instruction to move `y` to `L`. For example, if `y` is in memory but needs to be in a register, emit an instruction like `LOAD R1, y_addr` to move `y` into register `R1`.
- **Generate the Operation Instruction**:
    - **Generate Operation `op`**:
        - Determine the current location of `z` using the address descriptor. Prefer using a register if `z` is in both memory and a register.
        - Generate the machine instruction that performs the operation `op` on `y` and `z`, storing the result in `L`. For instance, if `op` is addition and `L` is a register `R3`, generate an instruction like `ADD R3, R1, R2`, where `R1` and `R2` are registers holding `y` and `z`, respectively.
    - **Update Address Descriptor**:
        - Update the address descriptor to indicate that `x` is now stored in `L`.
        - Ensure that `x` is not marked as being stored in any other location (invalidate previous locations).
- **Update Register Usage**:
    - **Free Unused Registers**:
        - Check the next use information for `y` and `z` to determine if they are needed later in the code.
        - If `y` and/or `z` have no further uses (i.e., they are not live after this point), update the register descriptor to free the registers they occupy. This makes these registers available for future use.

3. What is the role of peephole optimization in the compilation process

- **Machine dependent optimization** focuses on making improvements directly to the code that runs on the hardware, called the **target code**.
- This type of optimization tailors the code to make the best use of a specific computer or processor.

- Unlike machine independent optimizations, which are applied to intermediate code and are more general, machine dependent optimizations are applied after the target code is generated.

Peephole optimization

- **Peephole optimization** is a common machine dependent optimization technique.
- It involves looking at small sections of the target code (the "peephole") and finding ways to make those sections more efficient.
- The "peephole" is like a small moving window that examines a few instructions at a time.

Purpose of peephole

- The goal is to replace inefficient sequences of instructions with better ones that do the same thing but faster or using less memory.

Steps in peephole optimization

- **Redundant Load/Store:**
  - Removing unnecessary instructions that load or store data when it's not needed.

```
MOV R0,X
MOV X,R0
```

- **Remove Unreachable Code:**
  - Eliminating code that will never be executed.

```
void fun()
        {
                int a=10, b=20, c;
                c = a * 10;
                return c;
                b = b * 15;
                return b;
        }
```

- When `return c` is reached, it wont execute the remaining 2 lines, making it unreachable code

- **Flow of Control Optimization:**
  - Simplifying jumps and branches to make the control flow of the program more efficient.

In this code, label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

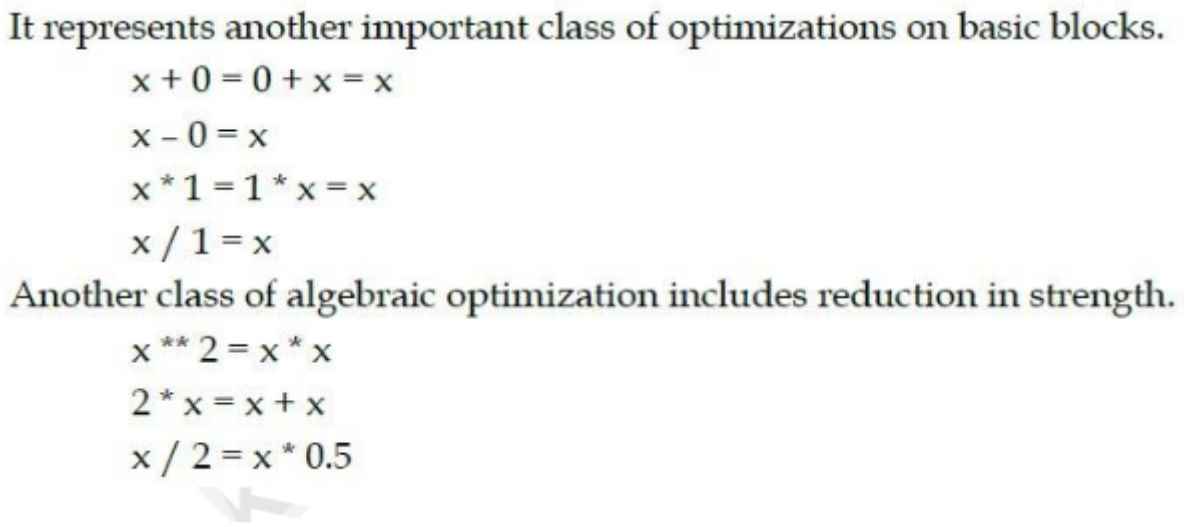
```
MOV R1, R2
GOTO L1

...

L1: GOTO L2
L2: INC R1
```

```
MOV R1,R2
GOTO L2
...
L2:INC R1
```

- **Algebraic Simplifications:**
  - Replacing complex arithmetic operations with simpler ones
  - (e.g., changing `x = x * 1` to `x = x`).



- **Use of Machine Idioms:**

- Using specific machine-level instructions that are particularly efficient on the target machine.
- some machines have auto-increment and auto-decrement addressing modes

4. What are the issues in the design of a code generator

Input to code generator

- The code generator takes two main inputs:
- **Intermediate Representation (IR):**
  - This is a simpler version of the source code that the compiler front end has already processed.
  - Types of IR include:
    - **Postfix Notation:** Where operators come after their operands.
    - **Three-Address Code (TAC):** Each instruction uses at most three addresses (e.g., `a = b + c`).
    - **Virtual Machine Code:** Like code for a stack-based machine.
    - **Graphical Representations:** Such as syntax trees (which show the structure of the code) and DAGs (Directed Acyclic Graphs).
- **Symbol Table:**
  - This contains information about all the variables, functions, etc., in the program. It tells the code generator things like types and memory locations.
  The code generation phase proceeds on the assumption that its
  input is free of errors

Target Program

- The goal of the code generator is to produce the target program, which can come in different forms:
  - **Absolute Machine Language:**
    - **Definition:** Code that can be loaded directly into a specific memory location and executed.
    - **Advantage:** It can be immediately executed without further processing.
  - **Relocatable Machine Language:**

        - **Definition:** Code that can be loaded into different memory locations.
        - **Advantage:** Allows separate compilation of subprograms. These can be linked together later by a linker before execution.
    - **Assembly Language:**
        - **Definition:** A low-level language that is easier to read and write than machine language.
        - **Advantage:** Simplifies the process of code generation. Assembly code is then converted to machine language by an assembler.
- **Target Machine Architectures**
    - The type of machine architecture significantly influences how the code generator works and the quality of the generated code. The main types of architectures are:
    - **RISC (Reduced Instruction Set Computer):**
        - **Features**
            - Many registers.
            - Three-address instructions (e.g., `add r1, r2, r3`).
            - Simple addressing modes.
            - Simple and uniform instruction set.
        - **Impact:** Easier to generate efficient code due to simplicity and consistency.
    - **CISC (Complex Instruction Set Computer):**
        - **Features:**
            - Few registers.
            - Two-address instructions (e.g., `add r1, r2`).
            - Multiple addressing modes.
            - Different classes of registers.
            - Variable length instructions.
        - **Impact:** More complex to generate efficient code because of the variety and complexity of instructions.
    - **Stack-Based Machines:**
        - **Features:**
            - Operations are performed on values at the top of a stack.
            - Operands are pushed onto the stack and operations use these topmost operands.
            - The top of the stack is often kept in registers for efficiency.

- **Impact:** Code generation focuses on managing the stack operations and keeping frequently used values in registers for performance.

Memory Management

- **Mapping Variable Names to Addresses:**
  - This is a joint effort by the front end and the code generator of the compiler.
  - The symbol table, which the front end creates, provides the necessary information.
- **Details:**
  - **Name and width are obtained from Symbol Table:**
    - **Name:** The variable's identifier.
    - **Width:** The amount of storage needed for the variable (e.g., how many bytes).
- **During Code Generation:**
  - **Translating Three-Address Code:**
    - Each instruction in the intermediate representation (three-address code) is converted into actual machine instructions and addresses.
  - **Relative Addressing:**
    - Instructions are assigned addresses relative to a starting point, not absolute memory addresses. This allows for flexibility in where the program can be loaded into memory.

Instruction Selection

- **Purpose:**
  - The code generator translates the intermediate representation (IR) of a program into machine code that the target machine can execute.
- **Factors Affecting Instruction Selection:**
  - **Level of the IR:**
    - **High-Level IR:** Simple and close to the original source code. Each IR statement is translated into several machine instructions using code templates. This often results in less efficient code.
    - **Low-Level IR:** Contains details closer to machine instructions, allowing the code generator to produce more efficient code sequences.
  - **Instruction-Set Architecture:**

        - The design and features of the target machine's instruction set significantly affect how easily and efficiently instructions can be selected and generated.
        - **Uniformity and Completeness:** If the instruction set is uniform (consistent) and complete (covers all necessary operations), instruction selection is easier.
    - **Desired Quality of Code:**
        - Quality is measured in terms of speed (execution time) and size (memory usage).
        - More efficient code generation often requires considering machine-specific details and optimizing beyond the straightforward translation.
- **Example: Simple Instruction Selection:**
    - Consider the statement `x = y + z`:
    - The code generator might produce:
        - MOV y, R0
        - ADD z, R0
        - MOV R0, x
- **Example: Optimized Instruction Selection:**
    - Consider a sequence of statements:
        - a = b + c
        - d = a + e
    - Without optimization
        - MOV b, R0
        - ADD c, R0
        - MOV R0, a
        - MOV a, R0 ; Redundant
        - ADD e, R0
        - MOV R0, d
    - With optimization recognizing that `a` is already in `R0`:
        - MOV b, R0
        - ADD c, R0
        - MOV R0, a
        - ADD e, R0
        - MOV R0, d

- **Using Special Instructions:**
    - If the target machine has specialized instructions, use them for efficiency.
    - For example, instead of:
        - MOV a, R0
        - ADD #1, R0
        - MOV R0, a
    - We can use
        - INC a

Register Allocation

- **What It Is:**
    - Register allocation is about deciding which program values (variables) should be kept in the limited number of registers available in the target machine. Registers are the fastest storage locations, so using them efficiently is crucial for performance.
- **Why It Matters:**
    - **Registers are Fast:** Accessing values in registers is much quicker than accessing values in memory.
    - **Limited Number:** Most machines have only a small number of registers, so careful management is needed to use them effectively.
- **Two Main Tasks:**
    - **Register Allocation:**
        - **Goal:** Decide which variables should be kept in registers throughout the execution of the program.
        - **How:** Determine at each point in the program which variables are most beneficial to keep in registers, based on their usage.
    - **Register Assignment:**
        - **Goal:** Choose which specific register each of these variables will use.
        - **How:** Assign the selected variables to specific registers, ensuring no conflicts.

Evaluation order

- **What It Is:**

        - Choosing the order in which operations are evaluated in the code can impact how efficiently the target code runs.
- **Why It Matters:**
    - Different evaluation orders can affect the number of registers and instructions needed in the target code.
- **Key Points:**
    - **Efficiency Impact:**
        - Some evaluation orders require fewer resources (like registers and instructions) than others, leading to more efficient code.
    - **Complexity:**
        - Selecting the best evaluation order is a complex problem.
        - In technical terms, it's an NP-complete problem, which means finding the optimal solution can be very time-consuming.
    - **Code Optimization:**
        - Even though finding the best evaluation order is difficult, code optimization techniques can help.
        - During optimization, the order of instructions in the code may change based on various factors, including the available resources and the characteristics of the target machine.

5. Describe the principal sources of optimization

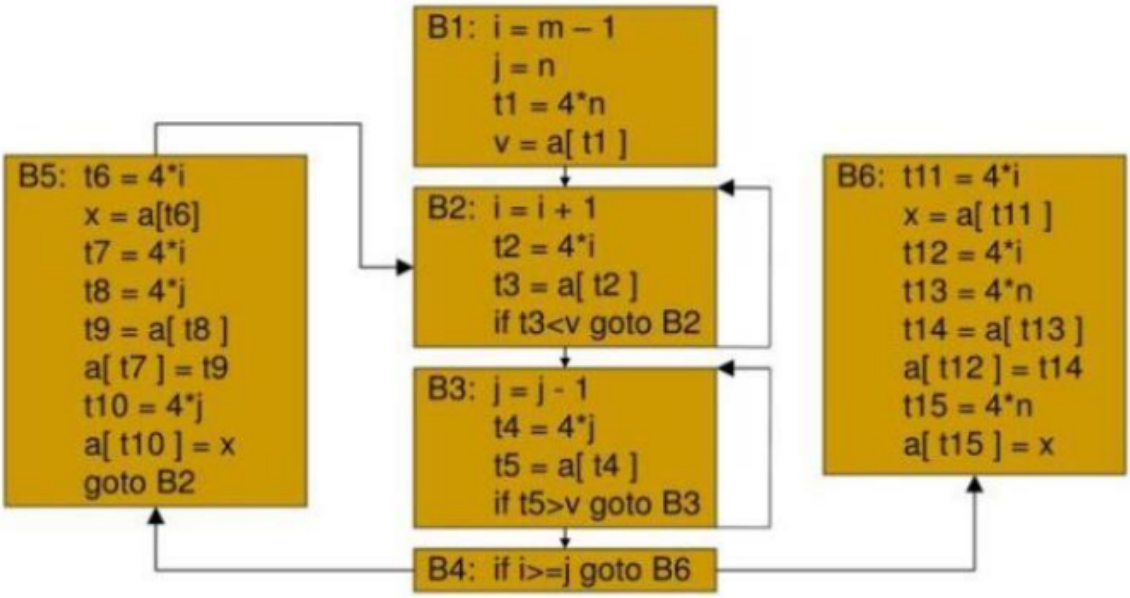
- A Transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise its called global. Many transformations can be performed at both local and global levels
- There are 2 of them, Function preserving transformations and loop optimization

Function Preserving transformations

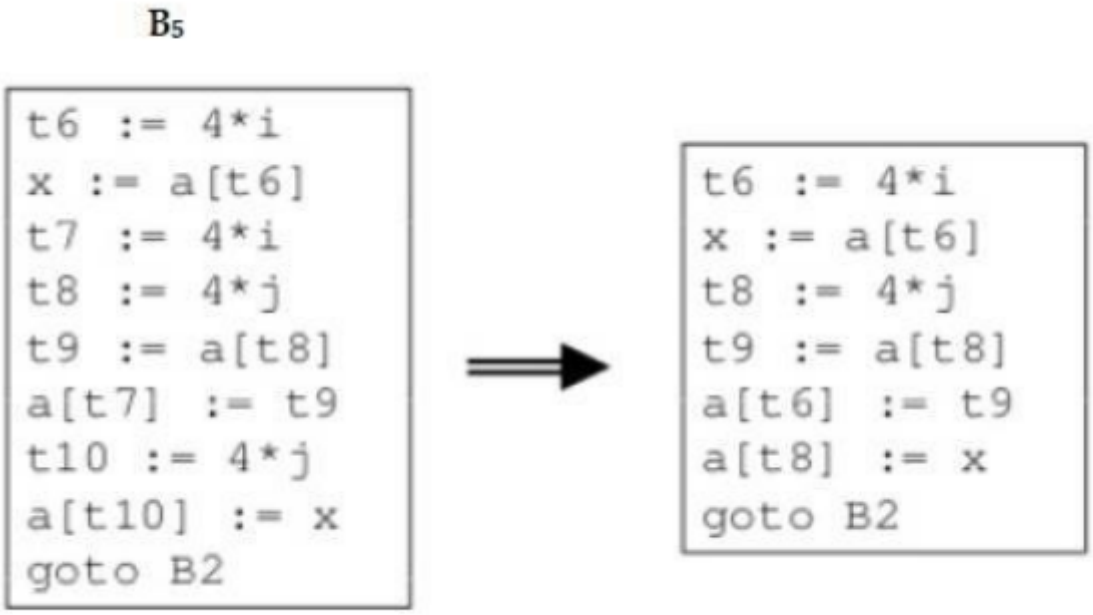
- There are a number of ways in which a compiler can improve a program without changing the function it computes
- Some function preserving transformations example are given below

a. Common subexpression elimination

- An occurence of 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
- Conside this flow graph example of quicksort



- Here We can eliminate some things inside B5
  - There are expressions which has the same values like
    - t6 = 4 * i
    - t7 = 4 * i
    - t8 = 4 * j
    - t10 = 4 * j
  - these excess values can be eliminated like so



-

b. Copy propagation

- Assignments of the form f=g called copy statements, or copies for short
- The idea behind copy progagation tranformation is to use g for f, whenever possible after the copy statement f = g
- Example
  - x:=t3 in block B5 of Flow Graph is a copy

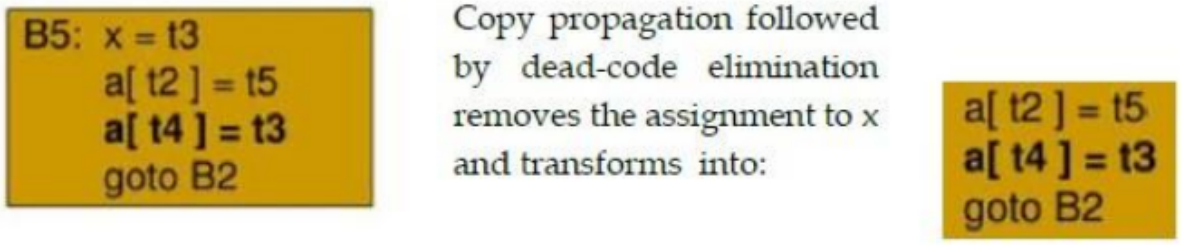
  -
  - This change gives us the opportunity to eliminate x, since its not used anywhere after the change
- Advantage of copy propogation
  - It often turns the copy statement into dead code
  - For example x = t3 is now dead code, since its not used anywhere else in the code

c. Dead code elimination

- A variable is live at a point in the program if its value can be used subsequently, otherwise its dead at that ppoint

- Dead code are statements which computes values but never get used
- Programmer is unlikely it introduce this intentionally, but can happen as a result of any transformation
    - For example, in the copy propogation example, we did a transformation and `x=t3` became a dead code

Consider B5 of flow graph.



-
- Here x = t3 is eliminated

d. Constant folding

- We can avoid evaluating expression at compile time if the operands used are constants
- For example consider this expression
    - a = 3.14157 / 2
    - Here the operands are 3.14153 and 2, which are constants
    - When dividing we get 1.570
    - So since we already know the answer we can remove the division operation and do the following
    - a = 1.570
    - This improves run time performance and reduces code size

Loop optimization

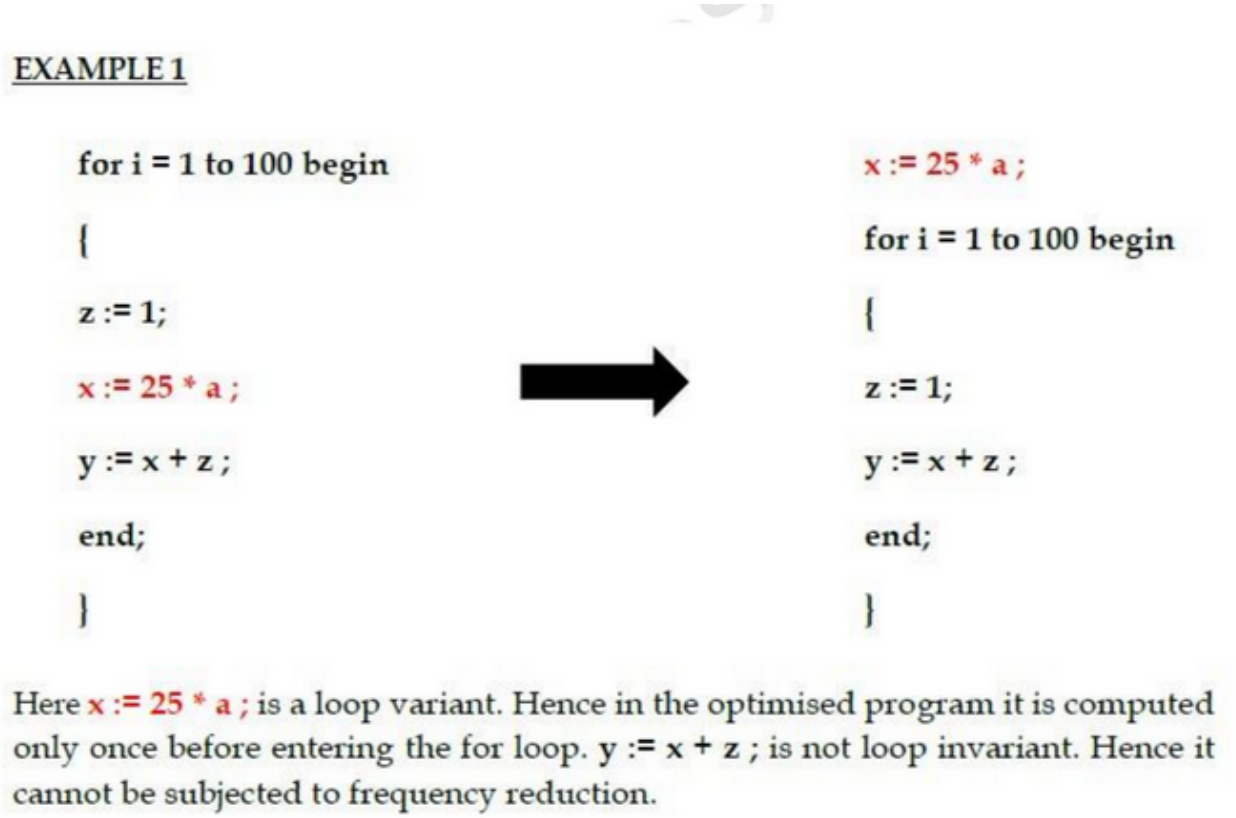
The running time of a program may be improved if the number of instruction in an inner loop is decreased, even if we increase the amount of code outside loop

a. Code motion

- Execution time of a program can be reduced by moving code from part of program which is executed very frequently to another part of the program which is executed fewer times

- We will be moving a code named **Loop invariant code**
  - **Loop invariant code** is a code that resides inside the loop and computes the same value of each iteration

Example



b. Induction variable elimination

- Loops are usually processed inside out. For example consider the loop around B3

```
B3: j = j - 1
        t4 = 4*j
        t5 = a[t4]
        if t5>v goto B3
```

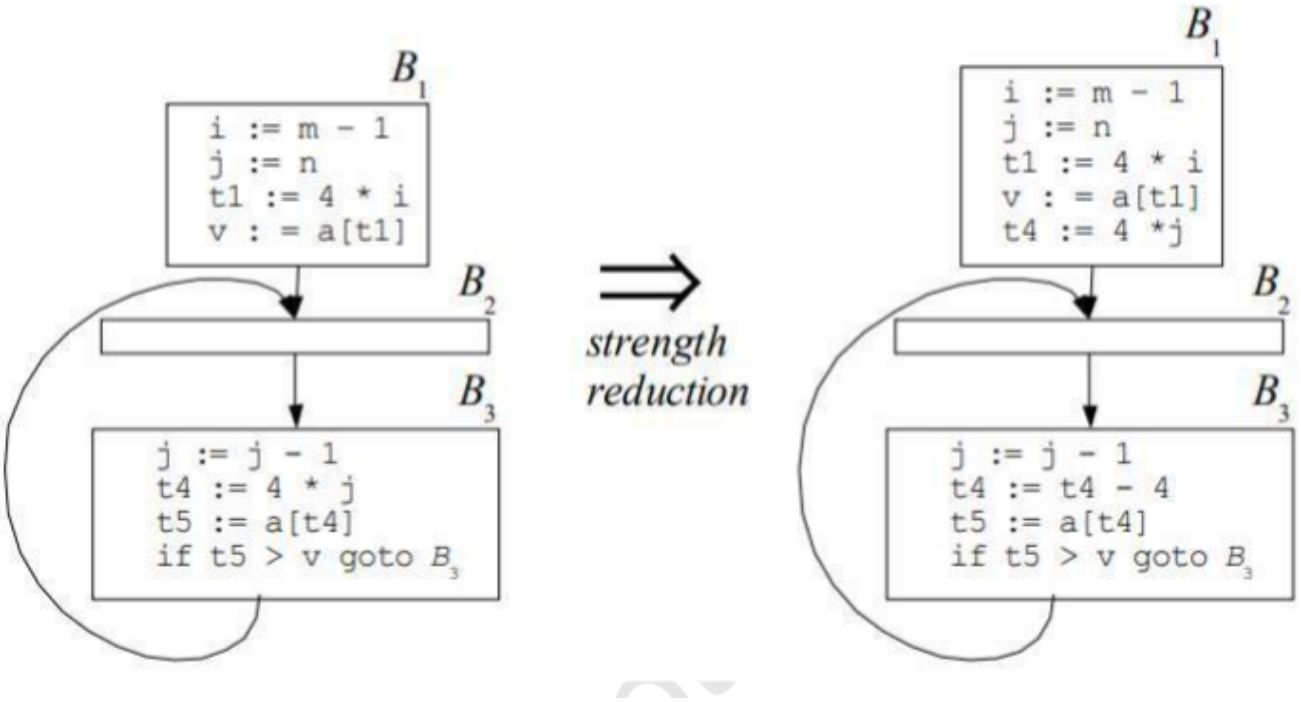
- Note that the value of j and t4 remain in lock-step
- Every time the value of j decreases by 1, that of t4 decreases by 4 because 4 * j is assigned to t4. Such identifier are called induction variables

c. Reduction in strength

- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction variable elimination.
- For the inner loop around B3 we cant get rid of either j or t4 completely; t4 is used in B4 and j in B4
- However, we can illustrate reduction in stength, eventually j will be eliminated when B2-B5 is considered

We may therefore replace the assignment t4:= 4*j by t4:= t4-4. The only problem is that t4 does not have a value when we enter block B3 for the first time.

Since we must maintain the relationship t4=4*j on entry to the block B3, we place an initializations of t4 at the end of the block where j itself is initialized, shown by the dashed addition to block B1 in Figure



- Here we replace t4 = t * j with t4 = t4 - 4
- This way, multiplication is replaced with subtraction which is more efficient

d. Loop Jamming

- If the operations performed can be done in a single loop then merge or combine the loops

```
// Program with multiple Loops

int main()
{
        for(i=0;i<n;i++)
                A[i] = i + 1
        for(j=0;j<n;j++)
                B[j] = j - 1
}

return 0;
```

This program can be converted to

```
// Program with multiple Loops

int main()
{
        for(i=0;i<n;i++)
                A[i] = i + 1
                B[i] = j - 1
}

return 0;
```

e. Loop Unrolling

- If there exists, simple code which can reduce the number of times the loop executes then, the loop can be replaced with these codes

```
// Program with loops
int main()
{
        for(i=0;i<3;i++)
                cout << "Cd";
        return 0;
}
```

- This can be converted to

```
// Program with loops
int main()
{
        cout << "Cd";
        cout << "Cd";
        cout << "Cd";
        return 0;
}
```

6. Illustrate the optimization of basic blocks with examples

There are two types of basic block optimizations:

1. Structure preserving transformations
2. Algebraic transformations

Structure preserving transformation

Copy Propagation

Its of two types, Variable and Constant Propagation

Variable Propagation

```
x = y               =>  z = y + 2
z = x + 2
```

Constant Propagation

```
x = 3               =>  z = 3 + a
z = x + a
```

Strength Reduction

- Replace expensive statement / instruction with cheaper ones

```
x = 2 * y (costly) => x = y + y (cheaper)
```

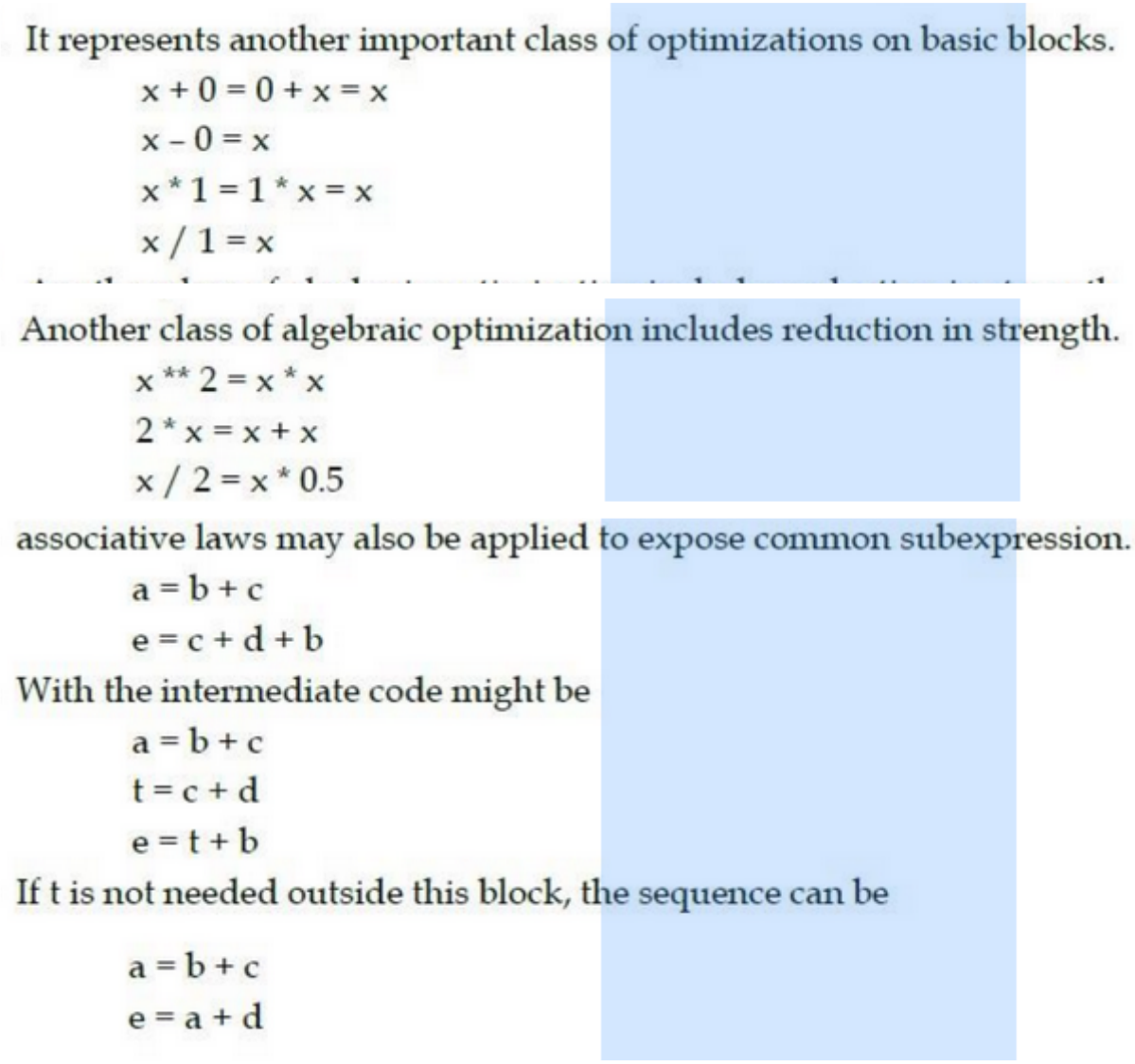
Constant Folding

- Solve the numeric expression and use the result in its place, so compiler need not calculate the result all the time

```
x = 2 * 3 + y => x = 6 + y (Optimized code)
```

Algebraic Transformations

We can simplify statements like



7. Write the Code Generation Algorithm and explain the getreg function

- **Code generation** is like the final step in making a recipe where you take all your prepared ingredients (**instructions**) and put them together to make the final dish (**machine code**). Here, we're converting intermediate code (a simplified version of the program) into machine code (the language the computer understands).
- The code generation algorithm works with **three-address statements**. Think of these as simple instructions in a cooking recipe. Each statement typically looks like:
  - `x := y op z`
  - This means "calculate `y op z` and store the result in `x`".

Steps of the algorithm

- For each three-address statement, the algorithm performs several actions:
- **Find Storage Location (getReg Function)**
  - The algorithm calls a function called `getReg` to figure out where to store the result of `y op z`. This could be in a register (a small, fast storage area in the CPU) or in memory.
- **Determine Current Location of `y`**
  -
    - The algorithm checks where `y` is currently stored. This is done using something called an **address descriptor**.
  - If `y` is both in memory and a register, it prefers the register because accessing data from a register is faster.
  - If `y` is not already in the chosen location `L`, it generates an instruction to move `y` to `L`.
- **Generate the Operation Instruction**
  - The algorithm then generates the instruction to perform the operation `op` on `z` and store the result in `L`.
  - It checks the current location of `z` and prefers using a register if `z` is both in memory and a register.
  - After generating this instruction, it updates the address descriptor to show that `x` is now in location `L` and not anywhere else.
- **Update Register Usage**
  - If `y` or `z` are no longer needed (i.e., they have no next uses or are not live after this point), the algorithm updates the register descriptor to show that these registers are now free for use by other variables.

getreg function

The `getReg` function helps the algorithm determine which register to use. Here's how it works:

1. **Check if `y` is Already in a Register**
   - If `y` is already in a register `R`, it uses that register.
2. **Find an Available Register**
   - If no specific register is required, it looks for any available register.
3. **Choose the Least Expensive Register**
   - If no registers are free, it picks a register that requires the fewest instructions to free up. This means moving the register's current contents to memory so the register can be used for `y`.

Example

`x := y + z`

1. **Invoke `getReg`**:
   - Determine where to store the result of `y + z`. Let's say it decides on register `R1`.
2. **Check Location of `y`**:
   - If `y` is in memory and register `R2`, and `R2` is free, it keeps `y` in `R2`. If not, it moves `y` to `R1`.
3. **Generate Operation Instruction**:
   - The algorithm generates an instruction to add `z` (from `R3`) to `y` (now in `R1`), storing the result in `R1`.
   - It generates the instruction: `ADD R3, R1`
     - This means: "Add the value of `z` (in `R3`) to the value in `R1` (which is now `y`), and store the result back in `R1`."
4. **Update Register Usage**:
   - If `y` or `z` are no longer needed, mark their registers as free.

8. Generate target code sequence for the following statement d := (a-b)+(a-c)+(a-c).

Step 1: Make Three Address Code Sequence

- We have d := (a-b)+(a-c)+(a-c)
- We need to make some equations, like
  - **t1:=a-b**
  - **t2:=a-c**
  - **t3:=t1+t2**
  - **d:=t3+t2**

Step 2: Make the table

- Initially all registers are empty
- First statement is t1 = a - b
  - We invoke getreg function to find location L, where the result is to be stored, Here L is R0
    - Initially we will consider R0 and R1
    - getreg function will return R0
  - We need to get current value of Y
    - a is not in any register
    - Check if a is not in R0
    - Moving a to R0
    - `MOV a,R0`
  - Generate the instruction op z',L
    - Check whether b is already in register or not
    - b is in memory, not in register
    - SUB b, R0
  - Update Register Descriptor
    - Here we describe what the register contains
    - R0 contains t1
  - Update Address descriptor
    - t1 in R0
- Second statement is t2 = a-c
  - Check a is in any register
    - R0 doesnt contain A, it contains t1
    - MOV a,R1
  - Check C is any register

        - SUB c, R1
    - Update Register descriptor
        - R0 contains t1
        - R1 contains t2
- Third statement is t3 = t1 + t2
    - Register descriptor says
        - R0 contains t1
        - R1 contains t2
    - ADD R1,R0
    - Update Register descriptor
- d = t3 + t2
    -

Statements	Code Generator	Register Descriptor	Address Descriptor
		Registers are empty	
t1 = a-b	MOV a,R0 SUB b,R0	R0 contains t	t in R0
t2 = a-c	MOV a,R1 SUB c,R1	R0 contains t1 R1 contains t2	t1 in R0 t2 in R1
t3 = t1 + t2	ADD R1,R0	R0 contains t3 R1 contains t2	t3 in R0 t2 in R1
d = t3 + t2	ADD R1,R0 MOV R0,d	R0 contains d	d in R0 d in R0 and memory