CD-MID

UNIT-4 QUESTIONS

1) What are the issues in design of code generator and describe it?

A) A code generator is the phase of a compiler that converts the intermediate code into target machine code. The design of an efficient code generator is complex and involves several important issues.

Input to the Code Generator

- The code generator receives intermediate code from previous phases.
- The form of this input (three-address code, syntax tree, etc.) affects the design and efficiency.

Target Language

- The type of target machine (RISC or CISC) influences instruction selection.
- The generator should create correct and efficient code for that machine.

Memory Management

- Proper allocation of registers and memory locations is important.
- Frequently used variables should be kept in registers for faster access.

Instruction Selection

- Selecting efficient machine instructions to perform a task is crucial.
- The generator must choose the best instructions based on speed and size.

Register Allocation

- Since registers are limited, use them wisely.
- Try to reduce memory access by reusing registers.

Instruction Ordering

- The order of instructions affects performance and execution.
- The code generator must order instructions to reduce delays.

Optimization

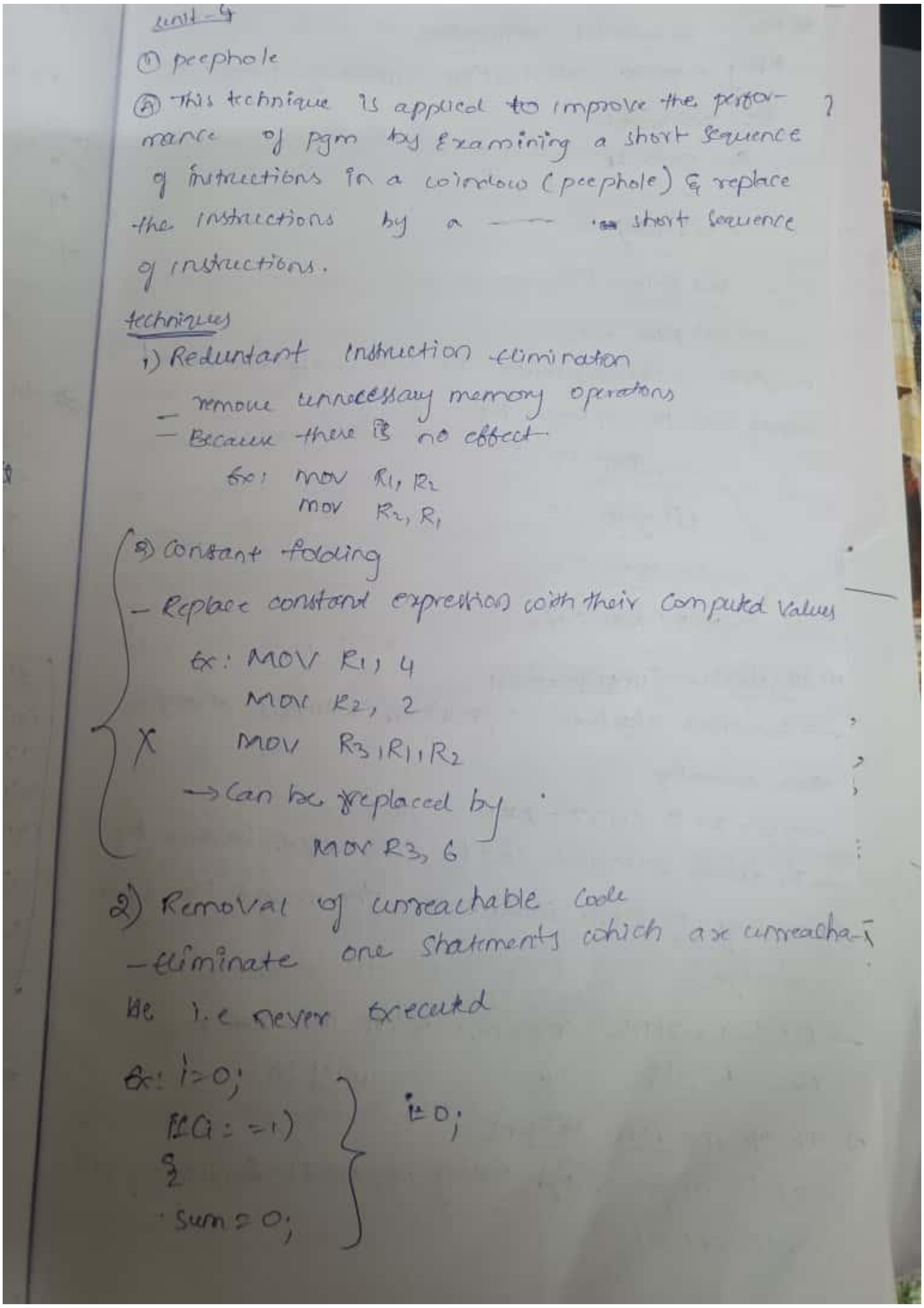
- The code should run faster and use fewer resources.
- Optimization can be at machine level, loop level, or register level.

Error Handling

- The code generator should be able to detect and report errors in the intermediate code or resource allocation.

Designing an efficient code generator means balancing correctness, speed, and machine requirements. A good code generator produces fast, small, and error-free machine code that runs correctly on the target system.

2) Explain about peephole optimization techniques?



unit - 4

① peephole

Ⓐ This technique is applied to improve the performance of pgm by Examining a short sequence of instructions in a window (peephole) & replace the instructions by a —— short sequence of instructions.

techniques

1) Redundant instruction elimination

- remove unnecessary memory operations
- Because there is no effect

```
Ex: mov R1, R2
    mov R2, R1
```

3) Constant folding

- Replace constant expression with their computed values

```
Ex: MOV R1, 4
    MOV R2, 2
    MOV R3, R1, R2
```

→ Can be replaced by

```
    MOV R3, 6
```

2) Removal of unreachable code

- Eliminate one statements which are unreachable i.e never executed

```
Ex: i=0;
    if(i==1)
    {
    sum=0;
    }
```

```
i=0;
```

3) Flow-of-control optimizations

– using peephole optimization unnessary jumps can be Eliminated

Ex: goto L1

```
      ----
L1: goto L2
      ----
L2: goto L3
      ----
L3: MOV a, R0
```

multiple jumps can make the code inefficient Above code can be replaced by

```
    goto L3
      ---
L1: goto L3
      ---
L2: goto L3
      ---
L3: MOV a, R0
```

4) Algebraic Simplifications

→ Simplifies algebraic expressions without changing their meaning

→ $x = x + 0$ (or) $x = x * 1$

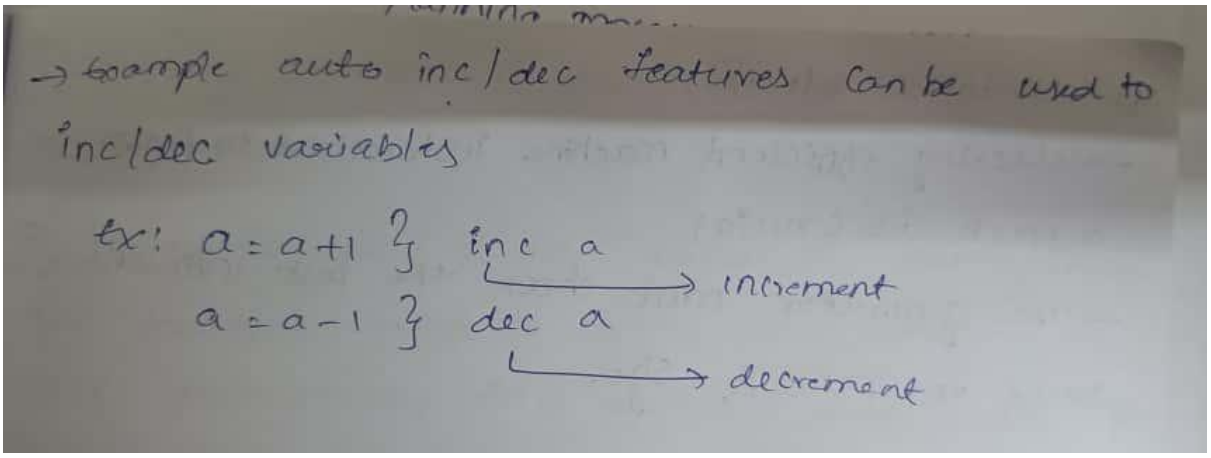
→ the above statement can be Eliminated because by executing of those statements the result 'x', can't change.

→ Ex: $a = x \wedge 2$ replaced with $a = x * x$

Ex: $b = y / 8$ " " $b = y >> 3$

5) use of machine Idioms

It is a process of using powerful features of CPU instructions



→ Example auto inc/dec features can be used to
inc/dec variables

Ex: $a = a+1$ } inc a $\xrightarrow{}$ increment
$a = a-1$ } dec a $\xrightarrow{}$ decrement

9) Briefly discuss about the garbage collection to avoid the manual memory management?

a) Garbage Collection

- Garbage collection is an automatic memory management technique used in programming languages like Java, Python, and C#.
- It automatically frees memory that is no longer in use, so the programmer does not have to manually deallocate it.

Key Principles

❖ In traditional manual memory management, programmers must allocate and free memory explicitly, which can cause bugs like memory leaks, double-free errors, and dangling pointers.

❖ Garbage collector automatically removes objects that are no longer accessible by the program, freeing up their memory.

Working of Garbage Collection

1. **Allocation:** When a program creates objects, memory is allocated from the heap area.
2. **Root Identification**: The garbage collector begins with a set of "roots" from which all reachable objects can be found.
3. **Marking (Reachability):**
    - The garbage collector finds all objects that are still reachable by the program (in use).
4. **Sweeping (Reclaiming):**
    - After marking, the collector goes through the memory and frees space used by objects that are not marked (not reachable).

Garbage Collection Algorithms

1. Mark and Sweep Algorithm

- Marks all reachable (in-use) objects.
- Sweeps (deletes) all unmarked (unused) objects from memory.

2. Generational GC:

- Divides heap into generations
- Short-lived objects are collected frequently from younger generations, improving efficiency

3. Copying Collector

- Copies live (active) objects from one memory area to another and frees the old area.

4. Reference Counting (used in some systems):

- Each object keeps a counter of references. If it becomes zero, the object can be collected.

Advantages

- No need for manual memory management.
- Prevents memory leaks and dangling pointers.
- Makes programs safer and more reliable.

Disadvantages

- Uses some CPU time for collection.
- Programmer has less control over when memory is freed.

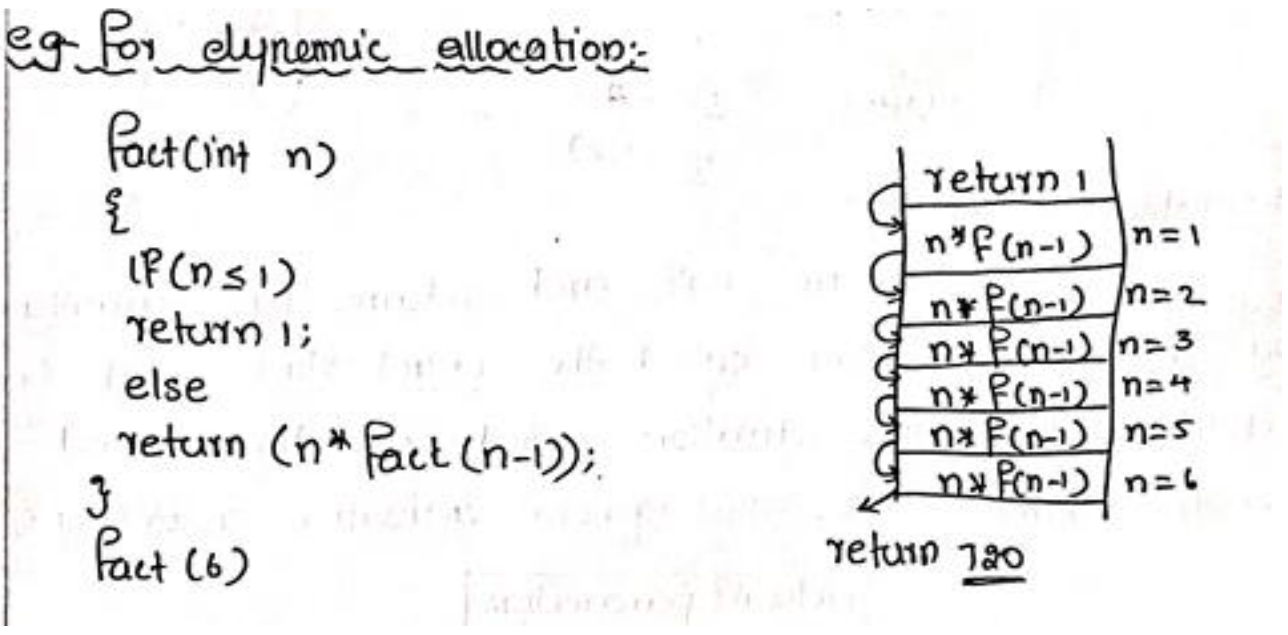
Conclusion

Garbage collection automatically manages memory by removing unused data. It prevents problems like memory leaks and makes programs faster, safer, and easier to handle.

3) Discuss Various stack allocation space with example?

a)

- Stack allocation is commonly known as **dynamic allocation**.
- Dynamic allocation refers to the allocation of memory at the time of execution (run time) rather than at compile time.
- The stack works on the LIFO (Last-In-First-Out) principle.
- Each time a function is called, an activation record is pushed onto the stack.
- Local variables are created when the function starts and automatically deleted when it ends.
- At run time, an activation record can be allocated by incrementing the 'top' of the stack by the size of the record and deallocated by decrementing 'top' similarly.

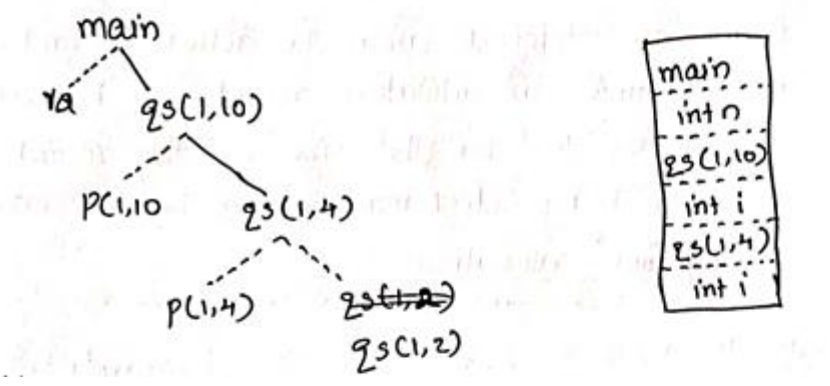


eg- For dynamic allocation:-

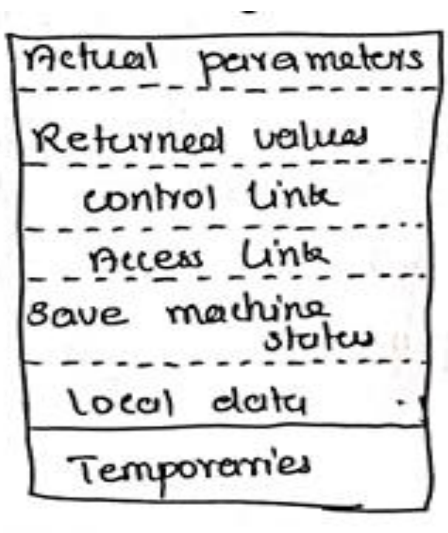
```
Fact(int n)
{
 if(n≤1)
 return 1;
 else
 return (n* Fact(n-1));
}
Fact(6)
```

Activation Trees

- The use of the run-time stack is enabled by relationships between the activation tree and the behavior of the program.
- The sequence of procedure calls corresponds to a preorder traversal of the activation tree, and sequence of returns to a postorder traversal.
- At any moment, nodes that are marked "live" (open) in the activation tree are those currently on the stack (from current procedure and its ancestors).



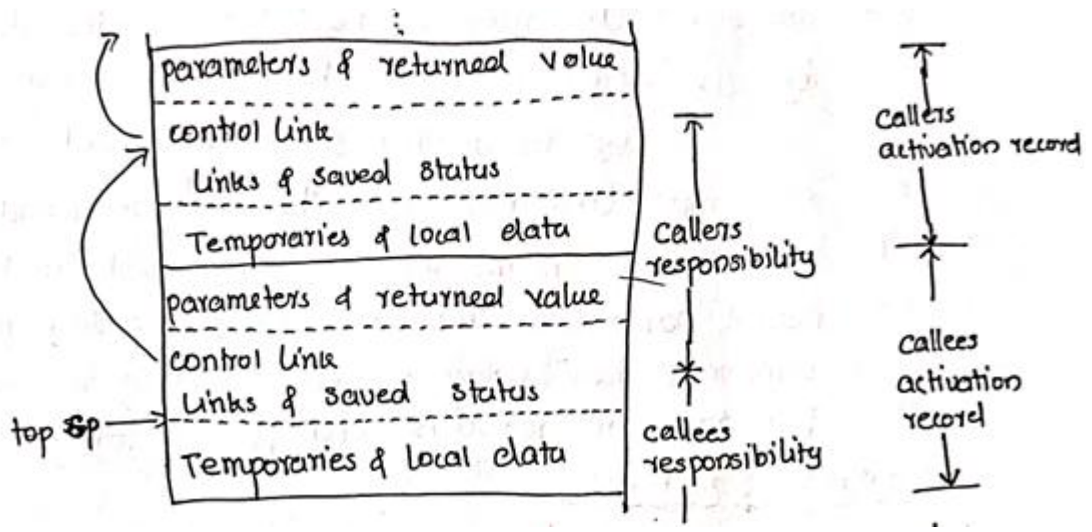
Activation Record:



An activation record (sometimes called a frame) includes:

- Actual parameters
- Returned value
- Control link (points to caller's activation record)
- Access link (if needed, to reach non-local data)
- Saved machine status (register/flags before call)
- Local data (for the procedure)
- Temporaries (values from expressions)

Calling Sequences:

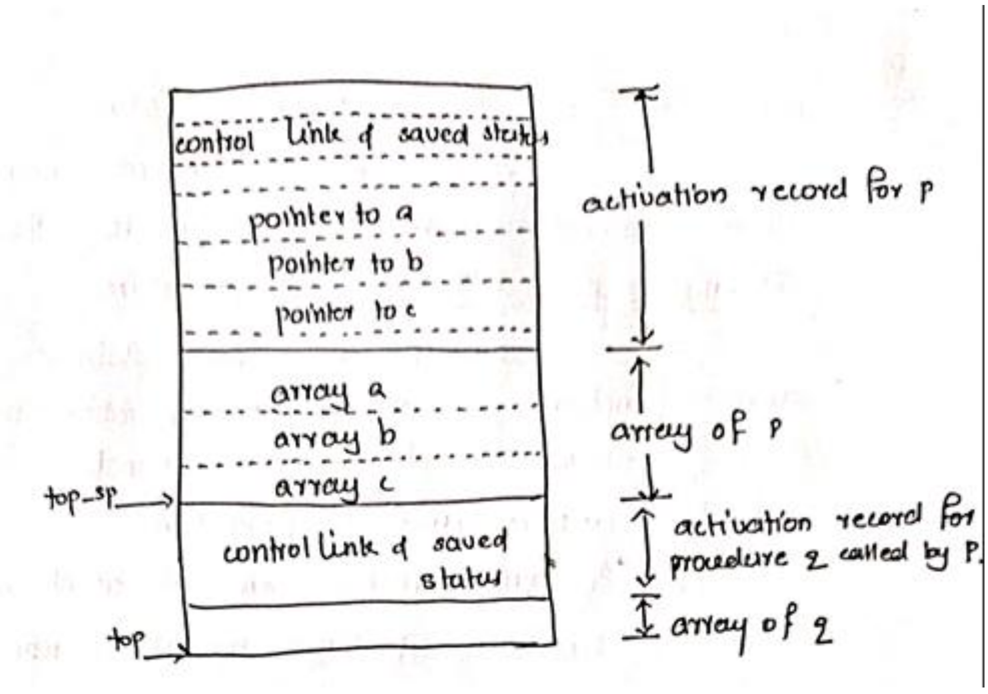


- Calling sequence involves code to allocate activation record on the stack and initialize its fields.

    1. Caller evaluates actual parameters.
    2. Caller stores return address and old stack top value into callee's activation record and increments top-sp.
    3. Callee saves register values and initializes local data.
    4. Callee begins execution.

Variable-Length Data on Stack:

- In modern languages, objects whose size is not determinable at compile time are allotted space in the heap rather than the stack.
- On the stack, we may only store pointers to such arrays. This avoids the overhead of garbage collection and simplifies access.
- In the activation record, only pointers to heap-allocated arrays are kept.



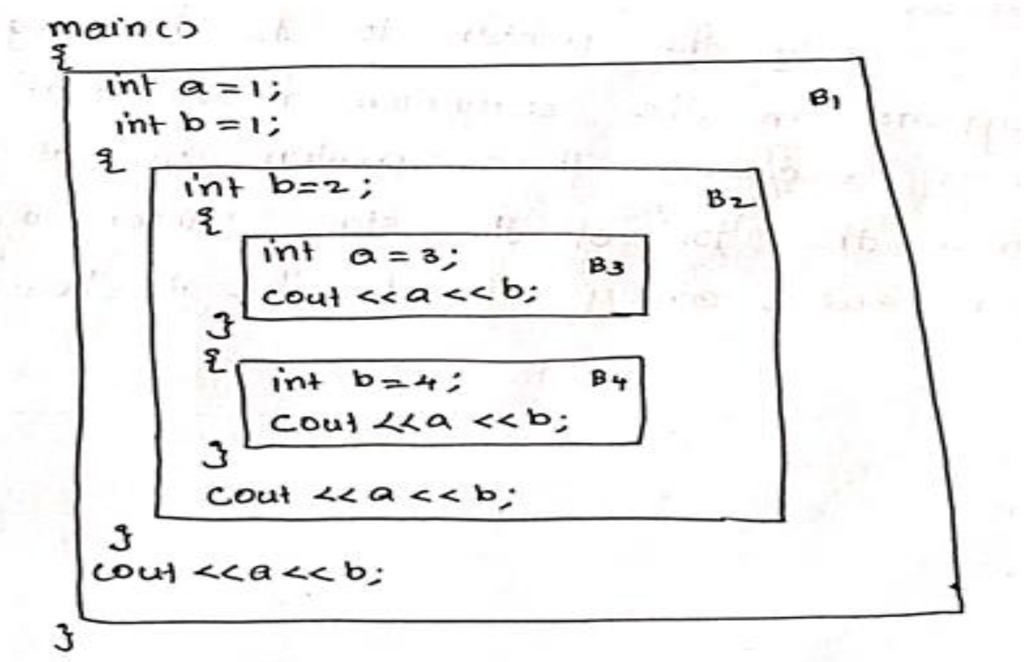
8) How to access Non-Local Data on stack with procedures?

a) Access become more complicated in languages where procedure can declare inside other procedures

1. Data Access Without Nested Procedures:

- All variables are either local to a function or global.
- Global variables are allocated in static storage, meaning their locations are fixed and known at compile time.
- *Any other variable* must be local to the activation at the top of the stack—these can be accessed via the stack pointer.

2. Issues With Nested Procedure:



```
main()
{
  int a=1;
  int b=1;
  {
    int b=2;
    {
      int a=3;
      cout<<a<<b;
    }
    {
      int b=4;
      cout<<a<<b;
    }
    cout<<a<<b;
  }
  cout<<a<<b;
}
```

- When procedures are nested, variable access must resolve which activation record contains the target variable.
- Example (see notes): A block structure with main, B1, B2, B3, B4, where inner blocks can declare variables of the same name, shadowing outer ones.

3. A Language with Nested Procedures:

- Most programming languages do not allow you to define a function inside another function. However, some modern languages do, such as ML (a functional language).
- ML allows you to declare procedures inside other procedures.
- Variable and Function Declaration in ML:
  - To define a variable and assign a value, you use:

    val <name> = <expression>
  - To define a function, you use:

    fun <name>(<arguments>) = <body>

4.Nesting Depth:

- Functions not nested within any other procedure are at nesting depth 1.
- If a procedure is defined within another, its nesting depth increases accordingly.

Access Links:

Access Link is a pointer added to each activation record to help nested functions follow the static scope rule.

- If procedure P is nested inside procedure Q, then in any activation of P, the access link points to the most recent activation of Q.
- These access links form a chain from the current to all outer activations at lower nesting levels.
- This chain helps the program access variables and data of all the functions that are visible to the currently executing procedure.

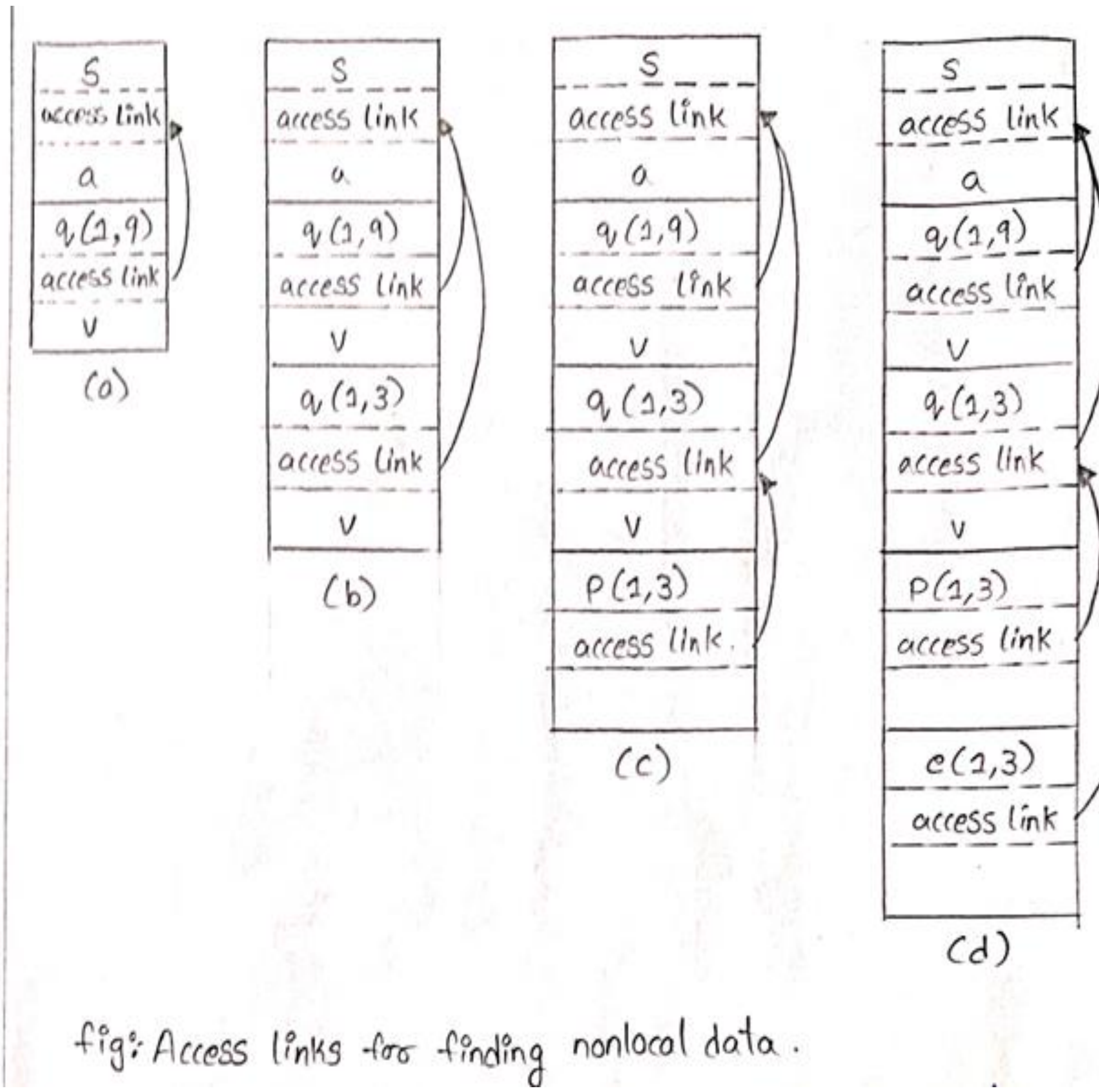


fig: Access links for finding nonlocal data.

This figure shows how access links help a function reach variable that are not local to it. Each activation record has an access link that points to the nearest outer function's record. When a nested function needs a variable from an outer function, it follows these access links step by step until it finds that variable. Thus, access links allow inner functions to use data from outer functions easily.

5) Explain in detail about simple code generator?

a) A code generator is the part of a compiler that translates the intermediate code of a source program into machine code (target program). It produces instructions that the CPU can directly execute.

a) Register and Address Descriptors

- The code generation algorithm checks each three-address instruction, decides which operands need to be loaded into registers, and then generates the operation.
- If the result needs to be stored in memory, it also generates that instruction.

1. Register Descriptor

- It keeps track of which variable's value is stored in which register. Only registers available within a basic block are used.
- Initially, all register descriptors are empty.
- As code generation proceeds, each register holds one or more variable values.

2. Address Descriptor

- It keeps track of all locations (register, memory, or stack) where the current value of a variable can be found.
- This information is stored in the symbol table entry for that variable.

b) Code Generation Algorithm

The code generation algorithm forms the core of the compiler. It uses register and address descriptors to generate machine instructions for three-address code statements.

1) Machine Instructions for Operations

For a three-address instruction like x = y + z:

- Use getReg(x = y + z) to select registers for x, y, and z → say Rx, Ry, Rz.
- If y is not in Ry, issue LD Ry, y' (load y into register Ry).
- If z is not in Rz, issue LD Rz, z'.
- Generate the operation → ADD Rx, Ry, Rz.

2) Machine Instructions for Copy Statements

For a copy statement like x = y:

- Choose the same register for both x and y.
- If y is not already in a register, load it using LD Ry, y.
- Update the register descriptor so that x is stored in Ry.
- No operation is needed if y is already in Ry.

3) Ending the Basic Block

- At the end of a block, temporary variables can be discarded.
- If a variable is live, its value must be saved in memory.
- Otherwise, its register can be reused.

4) Managing Register and Address Descriptors

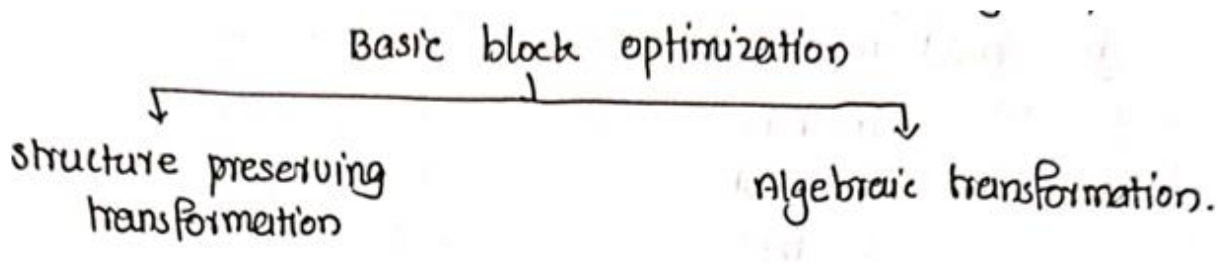
As the code is generated, the descriptors must be updated:

1. For instruction LD R, x:
   - Register descriptor for R holds only x.
   - Address descriptor for x adds register R.
2. For instruction ST x, R:
   - Address descriptor for x includes its memory location.
3. For ADD Rx, Ry, Rz (x = y + z):
   - Register descriptor for Rx holds only x.
   - Address descriptor for x's location is Rx.
4. For copy statement x = y:
   - Update x to use y's register.
   - Address descriptor for x becomes Ry.

The simple code generator uses register and address descriptors to efficiently manage variables and generate machine code. It ensures minimal memory access, proper register usage, and optimized code generation for three-address instructions.

7) Explain in detail about Optimization of Basic Blocks?

a) Optimization of basic blocks is done after the intermediate code generation phase of a compiler. It is the process of improving the code so that the program runs faster and uses fewer resources.


Basic block optimization
Structure preserving transformation
Algebraic transformation.

DAG Representation of Basic Blocks

A Directed Acyclic Graph (DAG) is used to represent a basic block. It helps to perform several code-improving transformations on the block.

Using DAG, we can:

a) **Eliminate Local Common Subexpressions:**

- Remove duplicate expressions that have already been computed.

b) **Eliminate Dead Code:**

- Remove instructions that compute values which are never used.

c) **Reorder Independent Statements:**

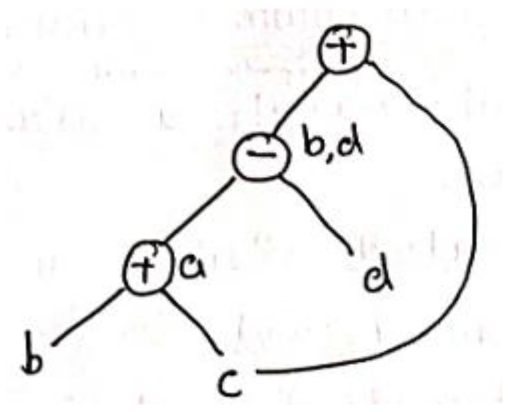
- Change the order of statements that don't depend on each other to improve efficiency.

d) **Apply Algebraic Laws:**

- Rearrange operands or simplify expressions using algebraic rules.

Finding Local Common Subexpressions

- When constructing the DAG, we check if a new node with the same operator and operands already exists.
- If it exists, we reuse it instead of creating a new one.
- **Example:**



a = b + c

b = a - d

c = b + c

d = a - d

Here, b + c is a common subexpression and computed only once.

Dead Code Elimination

- A variable is *live* if its value will be used later.
- If a variable's value is never used, such statements are called **dead code**.
- These statements are removed to make the program smaller and faster.

Example:

x = a
y = a * b
z = a * c

If y is never used, y = a * b is dead code and can be removed.

Use of Algebraic Identities

Algebraic identities help to simplify computations.

Examples:

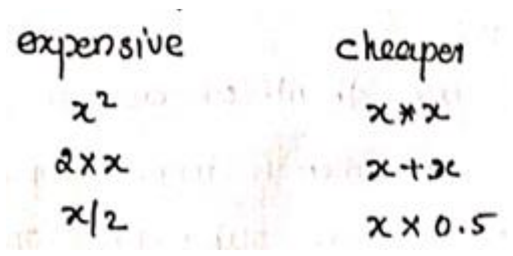
x + 0 = x

x - 0 = x

x * 1 = x

x / 1 = x

We can also replace expensive operations with cheaper ones:



expensive	cheaper
x^2	$x*x$
$2 \times x$	$x+x$
$x/2$	$x \times 0.5$

This improves execution speed.

Constant Folding

If an expression contains only constants, it is evaluated at compile time instead of run time.

Example: 2 * 3.14 → replaced by 6.28