**1.)**

1 Define Synthasized attributes and inherited attributes

S L.L|L L -> LBB B 0|1

Design an L-attributed SDD to compute S.val, the decimal-number value of an input string. For example, the translation of string 101.101 should be the decimal number 5.625.

Hint: use an inherited attribute L. Side that tells which side of the decimal point a bit is on

====

**Synthesized and Inherited Attributes**

In syntax-directed definitions (SDDs), attributes are associated with grammar symbols to hold information about the parse tree. These attributes can be of two types: **synthesized attributes** and **inherited attributes**.

1. **Synthesized Attributes**:
   * Synthesized attributes are those whose values are computed from the attribute values of the children (or subnodes) in the parse tree.
   * They flow upwards in the parse tree.
2. **Inherited Attributes**:
   * Inherited attributes are those whose values are assigned from the attribute values of the parent and/or siblings.
   * They flow downwards or horizontally in the parse tree.
3. **Example Grammar and SDD**
4. Given the grammar:

S -> L.L

L -> L B

L -> B

B -> 0 | 1

We want to design an L-attributed SDD to compute S.val, the decimal value of the input string. Let's add attributes:

* S.val: synthesized attribute representing the final decimal value.
* L.val: synthesized attribute representing the decimal value of the substring on the left side of the decimal point.
* L.frac\_val: synthesized attribute representing the fractional value of the substring on the right side of the decimal point.
* L.side: inherited attribute indicating whether L is on the left ("left") or right ("right") side of the decimal point.
* B.val: synthesized attribute representing the binary value (0 or 1).

**Semantic Rules**

Let's define the semantic rules to compute the attributes:

1. **Grammar: S -> L.L**
   * S.val = L1.val + L2.frac\_val
     + Where L1 is the L on the left side of the decimal point.
     + Where L2 is the L on the right side of the decimal point.
   * L1.side = "left"
   * L2.side = "right"
2. **Grammar: L -> L B**
   * For L.side == "left":
     + L1.val = 2 \* L2.val + B.val
     + L1.side = L.side
   * For L.side == "right":
     + L1.frac\_val = L2.frac\_val + B.val \* 2^(-L2.length - 1)
     + L1.length = L2.length + 1
     + L1.side = L.side
3. **Grammar: L -> B**
   * For L.side == "left":
     + L.val = B.val
   * For L.side == "right":
     + L.frac\_val = B.val \* 2^(-1)
     + L.length = 1
4. **Grammar: B -> 0 | 1**
   * B.val = 0 if B -> 0
   * B.val = 1 if B -> 1

**Attribute Computation Example**

Let's compute the value for 101.101:

**Step-by-Step Derivation**

1. Parse tree for 101.101:

css

Copy code

S

/ \

L L

/|\ /|\

L B L B B

| | | | |

B BBBB

| | | | |

1 0 1 1 0 1

1. **Attribute values**:
   * For the left L (101):
     + B1.val = 1
     + L2.val = 2 \* 1 + 0 = 2
     + L1.val = 2 \* 2 + 1 = 5
   * For the right L (101):
     + B1.val = 1 \* 2^-1 = 0.5
     + L2.frac\_val = 0.5 + 0 \* 2^-2 = 0.5
     + L1.frac\_val = 0.5 + 1 \* 2^-3 = 0.5 + 0.125 = 0.625
   * S.val = 5 + 0.625 = 5.625

Thus, the translation of 101.101 to its decimal value is 5.625.

---------------------------------------------------------------------------------------------------------------

2.)

Suppose that we have a production A→BCD. Each of the four non- terminals A, B, C, and D have two attributes: s is a synthesized attribute, and i is an inherited attribute. For each of the sets of rules below, tell whether

(i) the rules are consistent with an S-attributed definition

(ii) the rules are consistent with an L-attributed definition, and

(iii) whether the rules are consistent with any evaluation order at all?

(iv) Draw dependency graph for each set of rules

a) A.s= Bi +C.s

b) A .s=B.i+C.s and D .i=A.i+B.s.

c) A .s=B.s+D.s.

d) A .s=D.i B .i=A.s+C.s. Ci =B.s. and D .i=B.i+C.i.

=====

### Consistency with S-attributed and L-attributed Definitions

In the context of syntax-directed definitions (SDDs):

- \*\*S-attributed definitions\*\* use only synthesized attributes.

- \*\*L-attributed definitions\*\* allow both inherited and synthesized attributes, but inherited attributes can only depend on the parent and siblings to the left in the production.

### Evaluating the Rules

Let's examine each set of rules for the production \( A \to BCD \) and determine their consistency with S-attributed and L-attributed definitions, as well as whether they are consistent with any evaluation order.

#### Set (a): \( A.s = B.i + C.s \)

1. \*\*S-attributed\*\*:

- \*\*No\*\*, because \( A.s \) depends on \( B.i \), an inherited attribute.

2. \*\*L-attributed\*\*:

- \*\*Yes\*\*, because \( A.s \) can be computed after \( B.i \) and \( C.s \) are computed, and \( B.i \) is inherited from the parent.

3. \*\*Any evaluation order\*\*:

- \*\*Yes\*\*, the dependencies can be resolved.

4. \*\*Dependency graph\*\*:

- \( B.i \to A.s \)

- \( C.s \to A.s \)

#### Set (b): \( A.s = B.i + C.s \) and \( D.i = A.i + B.s \)

1. \*\*S-attributed\*\*:

- \*\*No\*\*, because \( A.s \) depends on \( B.i \) (inherited) and \( D.i \) depends on \( A.i \) (inherited) and \( B.s \).

2. \*\*L-attributed\*\*:

- \*\*Yes\*\*, because the inherited attributes only depend on the parent and siblings to the left.

- \( D.i \) depends on \( A.i \) and \( B.s \), which is valid for L-attributed.

3. \*\*Any evaluation order\*\*:

- \*\*Yes\*\*, the dependencies can be resolved.

4. \*\*Dependency graph\*\*:

- \( B.i \to A.s \)

- \( C.s \to A.s \)

- \( A.i \to D.i \)

- \( B.s \to D.i \)

#### Set (c): \( A.s = B.s + D.s \)

1. \*\*S-attributed\*\*:

- \*\*Yes\*\*, because all attributes involved are synthesized.

2. \*\*L-attributed\*\*:

- \*\*Yes\*\*, because it is also consistent with the L-attributed definition.

3. \*\*Any evaluation order\*\*:

- \*\*Yes\*\*, the dependencies can be resolved.

4. \*\*Dependency graph\*\*:

- \( B.s \to A.s \)

- \( D.s \to A.s \)

#### Set (d): \( A.s = D.i \), \( B.i = A.s + C.s \), \( C.i = B.s \), and \( D.i = B.i + C.i \)

1. \*\*S-attributed\*\*:

- \*\*No\*\*, because \( A.s \) and \( B.i \) depend on inherited attributes.

2. \*\*L-attributed\*\*:

- \*\*No\*\*, because there are cyclic dependencies that violate the L-attributed rule.

3. \*\*Any evaluation order\*\*:

- \*\*No\*\*, due to cyclic dependencies.

4. \*\*Dependency graph\*\*:

- \( D.i \to A.s \)

- \( A.s \to B.i \)

- \( C.s \to B.i \)

- \( B.s \to C.i \)

- \( B.i \to D.i \)

- \( C.i \to D.i \)

### Summary

- \*\*Set (a)\*\*: L-attributed and consistent with any evaluation order.

- \*\*Set (b)\*\*: L-attributed and consistent with any evaluation order.

- \*\*Set (c)\*\*: S-attributed, L-attributed, and consistent with any evaluation order.

- \*\*Set (d)\*\*: Not consistent with S-attributed, L-attributed, or any evaluation order due to cyclic dependencies.

### Dependency Graphs

Here are the dependency graphs for each set of rules:

\*\*Set (a):\*\*

B.i → A.s ← C.s

\*\*Set (b):\*\*

B.i → A.s ← C.s

A.i → D.i ← B.s

\*\*Set (c):\*\*

B.s → A.s ← D.s

\*\*Set (d):\*\*

D.i → A.s → B.i

↑ ↓

B.i ← C.s C.i ← B.s

↑ ↓

← D.i

The dependency graph for set (d) shows a cycle, indicating the dependencies cannot be resolved.

**--------------------------------------------------------------------------**

**3**

a) In this task we are given a Java-like language where methods can have locally defined methods. Furthermore, it is possible to declare variables and methods at the other most program level. That is supposed to work as usual in languages with static scoping. The following is a program in this language. {

class C {

void ml () {

void f ()\ \ ;

f ()

}

void m2 () {

int i;

void g () {

int j;

j = i

};

i = 1

rC .ml()

};};

Crc;

void main() {

}

rc= new C\ \ ;rC .m2\ \

}

Draw the call stack in the situation where the activation record for f is on top of the stack for the first time. Draw the stack including variables, access-links, and control-links, but without access-links for methods which are directly declared in a class.

**======**

To address the problem, let's break down the situation and draw the call stack when the activation record (AR) for the method `f` is on top of the stack for the first time. This requires understanding how the methods are invoked and the context in which `f` is called. Here is the provided code in a more readable format:

```java

class C {

void m1() {

void f() {

// method body

}

f(); // calling f

}

void m2() {

int i;

void g() {

int j;

j = i; // accessing i from m2

}

i = 1;

this.m1(); // calling m1

}

};

C rc;

void main() {

// main method

}

rc = new C();

rc.m2();

### Execution Flow and Call Stack Analysis

1. \*\*`main` method\*\*:

- Creates an instance of class `C`.

- Calls the method `m2` on the instance `rc`.

2. \*\*`m2` method\*\*:

- Declares a local variable `i`.

- Defines a local method `g`.

- Initializes `i` to 1.

- Calls `m1`.

3. \*\*`m1` method\*\*:

- Defines a local method `f`.

- Calls `f`.

At the point where `f` is called, the activation records on the call stack are:

- \*\*`main`\*\*

- \*\*`m2`\*\*

- \*\*`m1`\*\*

- \*\*`f`\*\*

### Call Stack and Activation Records

Let's draw the call stack including variables, access-links, and control-links at the point where `f` is on top of the stack.

\*\*Stack Frame for `f`\*\*:

- \*\*Variables\*\*: None (assuming `f` has no local variables)

- \*\*Access-link\*\*: Points to `m1`'s activation record

- \*\*Control-link\*\*: Points to `m1`'s activation record (return address to `m1` after `f` completes)

\*\*Stack Frame for `m1`\*\*:

- \*\*Variables\*\*: None (assuming `m1` has no local variables)

- \*\*Access-link\*\*: Points to `m2`'s activation record

- \*\*Control-link\*\*: Points to `m2`'s activation record (return address to `m2` after `m1` completes)

\*\*Stack Frame for `m2`\*\*:

- \*\*Variables\*\*: `i = 1`

- \*\*Access-link\*\*: Points to the global frame (assuming methods declared in a class are directly accessible)

- \*\*Control-link\*\*: Points to the global frame (return address to `main` after `m2` completes)

\*\*Stack Frame for `main`\*\*:

- \*\*Variables\*\*: None (assuming `main` has no local variables)

- \*\*Access-link\*\*: Points to the global frame (as `main` is at the outermost level)

- \*\*Control-link\*\*: Points to the global frame (return address to the system or program entry point)

### Visualization of the Call Stack

+---------------------------------+

| Stack Frame for `f` |

|---------------------------------|

| Variables: None |

| Access-link: AR of `m1` |

| Control-link: AR of `m1` |

+---------------------------------+

| Stack Frame for `m1` |

|---------------------------------|

| Variables: None |

| Access-link: AR of `m2` |

| Control-link: AR of `m2` |

+---------------------------------+

| Stack Frame for `m2` |

|---------------------------------|

| Variables: i = 1 |

| Access-link: Global Frame |

| Control-link: Global Frame |

+---------------------------------+

| Stack Frame for `main` |

|---------------------------------|

| Variables: None |

| Access-link: Global Frame |

| Control-link: Global Frame |

+---------------------------------+

| Global Frame |

|---------------------------------|

| Variables: rc = instance of C |

+---------------------------------+

### Notes on the Call Stack:

1. \*\*Access-link\*\*: Points to the activation record of the method in which the current method was defined, ensuring proper variable scope access in static scoping.

2. \*\*Control-link\*\*: Points to the activation record of the caller method, facilitating the return control flow.

This call stack structure supports static scoping by allowing each method to access its surrounding lexical environment correctly through access-links and control-links, ensuring that variable `i` in `m2` is accessible in `g` and `m1` correctly calls `f` within its defined scope.

**------------------------------------------------------------------------------------------------------------------------**

**3.b)**

To demonstrate the difference between static (lexical) and dynamic scoping, let's write a small program and explain how the variable values differ based on the scoping mechanism used.

### Program Example

```java

int x = 10;

void function1() {

int x = 20;

function2();}

void function2() {

print(x);}

void main() {

function1();

}

### Explanation

- \*\*Static Scoping (Lexical Scoping)\*\*:

- In static scoping, the scope of a variable is determined by the program's structure, specifically the block in which the variable is declared. The compiler resolves each variable reference to the most closely nested scope where the variable is declared.

- When `function2` is called, it looks for the nearest declaration of `x` in the environment where `function2` itself is defined. In this case, `function2` is defined at the global scope where `x` is 10. Therefore, the output will be `10`.

- \*\*Dynamic Scoping\*\*:

- In dynamic scoping, the scope of a variable is determined by the call stack at runtime. The interpreter or runtime system resolves each variable reference by searching the call stack from the current function backwards until it finds a variable with the same name.

- When `function2` is called, it looks for the nearest declaration of `x` in the call stack. At runtime, `function1` is on the call stack, and it has a local `x` with the value `20`. Therefore, the output will be `20`.

### Expected Output Based on Scoping Mechanism

- \*\*Static Scoping (Lexical Scoping)\*\*:

- The output of the program will be `10`.

- \*\*Dynamic Scoping\*\*:

- The output of the program will be `20`.

### Code Explanation

Here's the breakdown of what happens in each scoping mechanism:

1. \*\*Static Scoping\*\*:

- `main` calls `function1`.

- `function1` declares a local `x` with the value `20` and then calls `function2`.

- `function2` looks for `x` in its defining environment (the global scope) and finds `x = 10`.

- `function2` prints `10`.

2. \*\*Dynamic Scoping\*\*:

- `main` calls `function1`.

- `function1` declares a local `x` with the value `20` and then calls `function2`.

- `function2` looks for `x` in the runtime call stack. It finds the local `x` declared in `function1` with the value `20`.

- `function2` prints `20`.

### Conclusion

The program demonstrates how the value of `x` printed by `function2` differs based on whether the language uses static or dynamic scoping. Static scoping resolves `x` in the lexical scope of `function2`'s definition, while dynamic scoping resolves `x` based on the runtime call stack.

**-------------------------------------------------------------------------------------------------------------**

**4a)** Consider the following object-oriented program in Java style:

class A { inta,b; };

class B extends A { int c, d1, d2, d3,d4, d5, d6,d7,d8,d9; };

static void f(Ax) { x.a = 1 }

static void g(B x) { c = 2 }

static void h() { p = new A(); f(p); g(p); }

static void k() { B p = new B(); f(p); g(p); }

static void main() { h(); k () ; }

Explain the run-time structure of values of type A and B. Indicate a constraint on the layout of these structures needed to support inheritance.

=====

### Run-Time Structure of Values of Type A and B

In Java and many other object-oriented languages, objects are allocated on the heap, and their memory layout supports inheritance. Let's explore the structure of objects of type `A` and `B` and the necessary constraints to support inheritance.

### Class Definitions and Memory Layout

Given the class definitions:

```java

class A {

int a, b;}

class B extends A {

int c, d1, d2, d3, d4, d5, d6, d7, d8, d9;}

#### Object Layout

- \*\*Class A\*\*: Contains two integer fields, `a` and `b`.

- \*\*Class B\*\*: Inherits from `A`, so it contains `a` and `b` from `A`, plus its own fields: `c`, `d1`, `d2`, `d3`, `d4`, `d5`, `d6`, `d7`, `d8`, and `d9`.

The memory layout for instances of `A` and `B` can be visualized as follows:

- \*\*Instance of A\*\*:

+-------+

| a |

+-------+

| b |

+-------+

- \*\*Instance of B\*\*:

+-------+

| a | (inherited from A)

+-------+

| b | (inherited from A)

+-------+

| c |

+-------+

| d1 |

+-------+

| d2 |

+-------+

| d3 |

+-------+

| d4 |

+-------+

| d5 |

+-------+

| d6 |

+-------+

| d7 |

+-------+

| d8 |

+-------+

| d9 |

+-------+

### Constraint on the Layout to Support Inheritance

The primary constraint to support inheritance is that the memory layout of a subclass must extend the layout of its superclass. Specifically:

1. \*\*Prefix Constraint\*\*: The memory layout of `B` must start with the memory layout of `A`. This ensures that any `B` object can be treated as an `A` object, allowing polymorphism and correct method dispatch.

- This means that the fields of `A` (`a` and `b`) must be at the same offset in both `A` and `B`.

### Explanation of the Program

Let's walk through the program execution:

```java

static void f(A x) { x.a = 1; }

static void g(B x) { x.c = 2; }

static void h() {

A p = new A();

f(p); // p.a = 1

g(p); // Error: g(B) cannot be called with an A reference}

static void k() {

B p = new B();

f(p); // p.a = 1 (B instance is treated as an A instance)

g(p); // p.c = 2}

static void main() {

h(); // Error in g(p)

k(); // Works fine}

### Details of Method Execution

- \*\*Method `f(A x)`\*\*:

- Sets `x.a` to 1.

- Can be called with an instance of either `A` or `B` due to polymorphism. When called with an instance of `B`, `x.a` refers to the `a` field inherited from `A`.

- \*\*Method `g(B x)`\*\*:

- Sets `x.c` to 2.

- Can only be called with an instance of `B`, since `c` is not a member of `A`.

### Execution Flow

- \*\*Method `h`\*\*:

- Creates an instance of `A`.

- Calls `f(p)` which sets `p.a` to 1.

- Attempts to call `g(p)` with an `A` instance, leading to a compile-time error because `g` expects a `B` instance.

- \*\*Method `k`\*\*:

- Creates an instance of `B`.

- Calls `f(p)` which sets `p.a` to 1. Here, `p` is treated as an instance of `A` due to inheritance.

- Calls `g(p)` which sets `p.c` to 2.

### Conclusion

To support inheritance, the memory layout of a subclass must extend that of its superclass. The fields of the superclass must appear at the beginning of the subclass's layout. This allows methods expecting a superclass type to operate correctly on subclass instances. This is crucial for polymorphism and method dispatch to work correctly in object-oriented languages like Java.

**------------------------------------------------------------------------------------------------------------------------**

**4.b**)### Memory Leak in Reference Counting Garbage Collectors

\*\*Reference counting garbage collectors\*\* manage memory by keeping a count of references to each object. When an object's reference count drops to zero, the object is deallocated. However, this approach can suffer from a significant problem: \*\*circular references\*\*.

### Circular References Problem

A circular reference occurs when two or more objects reference each other, forming a cycle. In such cases, the reference counts of the objects in the cycle never drop to zero, even if they are no longer accessible from the rest of the program. This leads to memory leaks because the garbage collector never reclaims the memory used by these objects.

#### Example of Circular References

Consider the following pseudocode:

```java

class Node {

Node next;

Node prev;}

Node node1 = new Node();

Node node2 = new Node();

node1.next = node2;

node2.prev = node1;

// At this point, node1 and node2 form a circular reference.

node1 = null;

node2 = null;

In this example:

- `node1` references `node2`.

- `node2` references `node1`.

Even if we set `node1` and `node2` to `null`, the reference counts of both objects (`node1` and `node2`) remain non-zero due to their mutual references. Thus, they are not collected, leading to a memory leak.

### Addressing Circular References

To address the problem of circular references in reference counting garbage collectors, various techniques can be used:

1. \*\*Cycle Detection Algorithms\*\*:

- Implement algorithms to detect cycles in the object graph. When a cycle is detected, the collector can decrease the reference counts of all objects in the cycle, breaking the cycle and allowing the objects to be collected.

- \*\*Example Algorithm\*\*: A common cycle detection algorithm is the \*\*Bacon-Rajan cycle collector\*\*, which periodically searches for cycles among the objects and breaks them.

2. \*\*Hybrid Garbage Collection\*\*:

- Combine reference counting with other garbage collection techniques, such as mark-and-sweep, which can collect cycles.

- \*\*Example Approach\*\*: Use reference counting for immediate collection of non-cyclic garbage and occasionally invoke a tracing garbage collector to handle cycles.

3. \*\*Deferred Reference Counting\*\*:

- Use deferred or lazy reference counting to reduce the overhead of maintaining reference counts, and combine it with periodic tracing to detect and collect cycles.

- \*\*Example Approach\*\*: Only decrement reference counts for local variables when certain conditions are met (e.g., during specific garbage collection phases), and use a mark-and-sweep collector for full collection.

4. \*\*Weak References\*\*:

- Use weak references for certain references that should not contribute to the reference count. This approach can help avoid creating strong circular dependencies.

- \*\*Example Usage\*\*: Use weak references for back-references or caches, where the existence of one object should not prevent the garbage collection of another.

### Practical Implementation

Here is an example of how a hybrid approach might be implemented in a simple garbage collector:

```java

class Node {

Node next;

Node prev;

int refCount;

Node() {

refCount = 1; // Initial reference count }

void addReference() {

refCount++; }

void removeReference() {

refCount--;

if (refCount == 0) {

// Perform deferred reference count adjustments

if (next != null) next.removeReference();

if (prev != null) prev.removeReference();

// Here we would also check for and handle cycles if necessary}}}

// Assume we have a mark-and-sweep phase integrated

void markAndSweep() {

// Mark reachable objects

// Sweep unmarked objects, handling cycles if detected}

### Conclusion

While reference counting garbage collectors are simple and efficient for many scenarios, they are prone to memory leaks due to circular references. Addressing this issue requires additional mechanisms like cycle detection algorithms, hybrid garbage collection strategies, deferred reference counting, or the use of weak references. Each of these techniques helps mitigate the problem of circular references and ensures that memory is reclaimed effectively.

**------------------------------------------------------------------------------------------------------------------------**

**5**

Define basci block and control flow graph.Discuss the algorithm to

construct basic blocks from the given three address code

for i = 0 ;i<n; i ++ for ( j = 0 ; j < n j++) c[i][j] = 0 for ( i = 0; i< n ;i++) for ( j = 0; j < n ;j++) for ( k = 0 ; k<n; k++) c[i][j] = c[i][j] + a[i][k] \* b[k][j]

Construct the control flow graph for the above code

======

### Definitions

#### Basic Block

A \*\*basic block\*\* is a sequence of consecutive statements in a program with the following properties:

1. \*\*Single entry point\*\*: The control flow enters the basic block only at the beginning.

2. \*\*Single exit point\*\*: The control flow leaves the basic block only at the end, without halting or branching except at the end.

#### Control Flow Graph (CFG)

A \*\*control flow graph (CFG)\*\* is a directed graph used to represent all paths that might be traversed through a program during its execution. Each node in the CFG represents a basic block, and edges between nodes represent the flow of control from one block to another.

### Algorithm to Construct Basic Blocks from Three-Address Code

1. \*\*Identify Leaders\*\*:

- The first instruction is a leader.

- Any instruction that is the target of a jump (label) is a leader.

- Any instruction that immediately follows a jump instruction is a leader.

2. \*\*Form Basic Blocks\*\*:

- A basic block starts with a leader and includes all subsequent instructions up to, but not including, the next leader or the end of the program.

### Example Code

```java

for (i = 0; i< n; i++)

for (j = 0; j < n; j++)

c[i][j] = 0;

for (i = 0; i< n; i++)

for (j = 0; j < n; j++)

for (k = 0; k < n; k++)

c[i][j] = c[i][j] + a[i][k] \* b[k][j];

### Three-Address Code

Assuming `n` is a predefined variable, the equivalent three-address code might be:

1. `i = 0`

2. `if i>= n goto end1`

3. `j = 0`

4. `if j >= n goto next\_i1`

5. `c[i][j] = 0`

6. `j = j + 1`

7. `goto 4`

8. `next\_i1:`

9. `i = i + 1`

10. `goto 2`

11. `end1:`

12. `i = 0`

13. `if i>= n goto end2`

14. `j = 0`

15. `if j >= n goto next\_i2`

16. `k = 0`

17. `if k >= n gotonext\_j`

18. `c[i][j] = c[i][j] + a[i][k] \* b[k][j]`

19. `k = k + 1`

20. `goto 17`

21. `next\_j:`

22. `j = j + 1`

23. `goto 15`

24. `next\_i2:`

25. `i = i + 1`

26. `goto 13`

27. `end2:`

### Identifying Leaders

- `i = 0` (first instruction)

- `if i>= n goto end1` (target of jump)

- `if j >= n goto next\_i1` (target of jump)

- `goto 4` (instruction following a jump)

- `goto 2` (instruction following a jump)

- `i = 0` (first instruction of second loop)

- `if i>= n goto end2` (target of jump)

- `if j >= n goto next\_i2` (target of jump)

- `if k >= n gotonext\_j` (target of jump)

- `goto 17` (instruction following a jump)

- `goto 15` (instruction following a jump)

- `goto 13` (instruction following a jump)

### Basic Blocks

1. \*\*Block B1\*\*:

- `i = 0`

- `if i>= n goto end1`

2. \*\*Block B2\*\*:

- `j = 0`

- `if j >= n goto next\_i1`

3. \*\*Block B3\*\*:

- `c[i][j] = 0`

- `j = j + 1`

- `goto 4`

4. \*\*Block B4\*\*:

- `next\_i1:`

- `i = i + 1`

- `goto 2`

5. \*\*Block B5\*\*:

- `end1:`

- `i = 0`

- `if i>= n goto end2`

6. \*\*Block B6\*\*:

- `j = 0`

- `if j >= n goto next\_i2`

7. \*\*Block B7\*\*:

- `k = 0`

- `if k >= n gotonext\_j`

8. \*\*Block B8\*\*:

- `c[i][j] = c[i][j] + a[i][k] \* b[k][j]`

- `k = k + 1`

- `goto 17`

9. \*\*Block B9\*\*:

- `next\_j:`

- `j = j + 1`

- `goto 15`

10. \*\*Block B10\*\*:

- `next\_i2:`

- `i = i + 1`

- `goto 13`

11. \*\*Block B11\*\*:

- `end2:`

### Control Flow Graph (CFG)

The nodes in the CFG correspond to the basic blocks. The edges indicate the possible flow of control between the blocks.

+------+

| B1 |

+------+

|

v

+------+

| B2 |

+------+

|

v

+------+

| B3 |

+------+

|

v

+------+

| B4 |

+------+

^ |

| v

| +------+

| | B5 |

| +------+

| |

+-------v

+------+

| B6 |

+------+

|

v

+------+

| B7 |

+------+

|

v

+------+

| B8 |

+------+

|

v

+------+

| B9 |

+------+

|

v

+------+

| B10 |

+------+

|

v

+------+

| B11 |

+------+

### Explanation of the CFG

- \*\*B1\*\*: Initializes `i` and checks the outer loop condition.

- \*\*B2\*\*: Initializes `j` and checks the inner loop condition.

- \*\*B3\*\*: Sets `c[i][j] = 0` and increments `j`.

- \*\*B4\*\*: Increments `i` and loops back to `B1`.

- \*\*B5\*\*: Re-initializes `i` and checks the outer loop condition for the second set of nested loops.

- \*\*B6\*\*: Initializes `j` and checks the inner loop condition.

- \*\*B7\*\*: Initializes `k` and checks the innermost loop condition.

- \*\*B8\*\*: Performs the matrix multiplication step and increments `k`.

- \*\*B9\*\*: Increments `j` and loops back to `B6`.

- \*\*B10\*\*: Increments `i` and loops back to `B5`.

- \*\*B11\*\*: End of the program.

This CFG captures the control flow through the nested loops and matrix initialization and multiplication steps, accurately reflecting the program structure.

**-----------------------------------------------------------------------------------------------------------------**

**6**

void quicksort(int m, int n)

/\* recursively sorts a[m] through a[n] \*/

{int i, j;

int v, x;

if (n <= m) return;

/\* fragment begins here \*/ i = m - 1 j = n v = a[n] ;

while (1) { doii = i + 1 while (a[i] < v) j = j - 1 while (a[j] > v) if (i>= j) break; x = a[i] a[i] = a[j] ; a[j] = x; /\* swap a[i], a[j] \*/

} x = a[i] a[i] = a[n] ; a[n] = x /\* swap a[i], a[n] / / fragment ends here \*/ quicksort(m,j); quicksort (i + 1, n) ;

}

Explain machine independent optimization and convert the above code into three address code and optimize the generated three address code.

**====================**

### Machine Independent Optimization

\*\*Machine independent optimization\*\* refers to the process of improving the performance and efficiency of code at a high level, without considering the specifics of the underlying hardware. These optimizations are performed by the compiler and focus on improving the intermediate representation (IR) of the code. Common techniques include:

1. \*\*Constant Folding\*\*: Evaluate constant expressions at compile time.

2. \*\*Constant Propagation\*\*: Replace variables that have constant values with those values.

3. \*\*Common Subexpression Elimination\*\*: Reuse previously computed values.

4. \*\*Dead Code Elimination\*\*: Remove code that does not affect the program’s outcome.

5. \*\*Strength Reduction\*\*: Replace expensive operations with cheaper ones.

6. \*\*Loop Optimizations\*\*: Improve the performance of loops, such as loop unrolling and loop-invariant code motion.

### Converting the Quicksort Fragment to Three-Address Code

We will convert the fragment of the quicksort function to three-address code, which involves breaking down operations into simple, atomic instructions.

#### Original Fragment

```c

/\* fragment begins here \*/

i = m - 1;

j = n;

v = a[n];

while (1) {

do i = i + 1 while (a[i] < v);

do j = j - 1 while (a[j] > v);

if (i>= j) break;

x = a[i];

a[i] = a[j];

a[j] = x; /\* swap a[i], a[j] \*/

}

x = a[i];

a[i] = a[n];

a[n] = x; /\* swap a[i], a[n] \*/

/\* fragment ends here \*/

#### Three-Address Code

1. `i = m - 1`

2. `j = n`

3. `v = a[n]`

4. `L1:`

5. `i = i + 1`

6. `if a[i] >= v goto L2`

7. `goto L1`

8. `L2:`

9. `j = j - 1`

10. `if a[j] <= v goto L3`

11. `goto L2`

12. `L3:`

13. `if i>= j goto L4`

14. `x = a[i]`

15. `a[i] = a[j]`

16. `a[j] = x`

17. `goto L1`

18. `L4:`

19. `x = a[i]`

20. `a[i] = a[n]`

21. `a[n] = x`

### Optimized Three-Address Code

1. \*\*Constant Propagation\*\*: No obvious constants to propagate.

2. \*\*Dead Code Elimination\*\*: All code is used.

3. \*\*Common Subexpression Elimination\*\*: No common subexpressions.

4. \*\*Strength Reduction\*\*: No expensive operations to replace.

5. \*\*Loop Optimization\*\*: Optimize the loop by merging redundant comparisons.

We notice that the loop structure can be slightly optimized by combining some of the comparisons directly in the loop conditions:

```c

i = m - 1

j = n

v = a[n]

L1:

i = i + 1

L5:

if a[i] < v goto L5

L2:

j = j - 1

L6:

if a[j] > v goto L6

L3:

if i>= j goto L4

x = a[i]

a[i] = a[j]

a[j] = x

goto L1

L4:

x = a[i]

a[i] = a[n]

a[n] = x

### Explanation

- \*\*L1\*\*: Start of the loop.

- \*\*L5\*\*: Find the first element from the left that is not less than `v`.

- \*\*L2\*\*: Continue to find the first element from the right that is not greater than `v`.

- \*\*L6\*\*: Find the first element from the right that is greater than `v`.

- \*\*L3\*\*: Check if the indices have crossed.

- \*\*L4\*\*: Perform the final swap to place the pivot element in the correct position.

### Final Optimized Three-Address Code

1. `i = m - 1`

2. `j = n`

3. `v = a[n]`

4. `L1: i = i + 1`

5. `if a[i] < v goto L1`

6. `L2: j = j - 1`

7. `if a[j] > v goto L2`

8. `if i>= j goto L4`

9. `x = a[i]`

10. `a[i] = a[j]`

11. `a[j] = x`

12. `goto L1`

13. `L4: x = a[i]`

14. `a[i] = a[n]`

15. `a[n] = x`

### Explanation of the Optimized Code

- Lines 4 and 5: The increment and condition check for `i` are merged into a single loop (`L1`).

- Lines 6 and 7: The decrement and condition check for `j` are merged into a single loop (`L2`).

- This reduces the number of jumps and simplifies the control flow, making the code more efficient.