UNIT - 5 : DEFINITION OF BASIC BLOCK & CONTROL FLOW GRAPHS

Basic Block:

Basic Block is a straight-line code sequence that has no branches in and out branches except to the entry and at the end respectively. Basic block contains a sequence of statement. The flow of control enters at the beginning of the statement and leave at the end without any halt (except may be the last instruction of the block).

The following sequence of three address statements forms a basic block:

t1:= x * x

t2:= x * y

t3:= 2 * t2

t4:= t1 + t3

t5:= y * y

t6:= t4 + t5

Basic Block Construction:

Algorithm: Partition into basic blocks

Input: It contains the sequence of three address statements

Output: it contains a list of basic blocks with each three address statement in exactly one block

Method: First identify the leader in the code. The rules for finding leaders are as follows:

- The first statement is a leader.
- Statement L is a leader if there is an conditional or unconditional goto statement like: if....goto L or goto L
- Instruction L is a leader if it immediately follows a goto or conditional goto statement like: if goto B or goto B

For each leader, its basic block consists of the leader and all statement up to. It doesn't include the next leader or end of the program.

Consider the following source code for dot product of two vectors a and b of length 10:

```
begin
        prod :=0;
i:=1;
do begin
prod :=prod+ a[i] * b[i];
i :=i+1;
end
while i <= 10
end
```

The three address code for the above source program is given below:

B1:

(1) prod := 0

(2) i := 1

B2:

(3) t1 := 4* i

(4) t2 := a[t1]

(5) t3 := 4* i

(6) t4 := b[t3]

(7) t5 := t2*t4

(8) t6 := prod+t5

(9) prod := t6

(10) t7 := i+1

(11) i := t7

(12) if i<=10 goto (3)

Basic block B1 contains the statement (1) to (2)

Basic block B2 contains the statement (3) to (12)

Control Flow Graph:

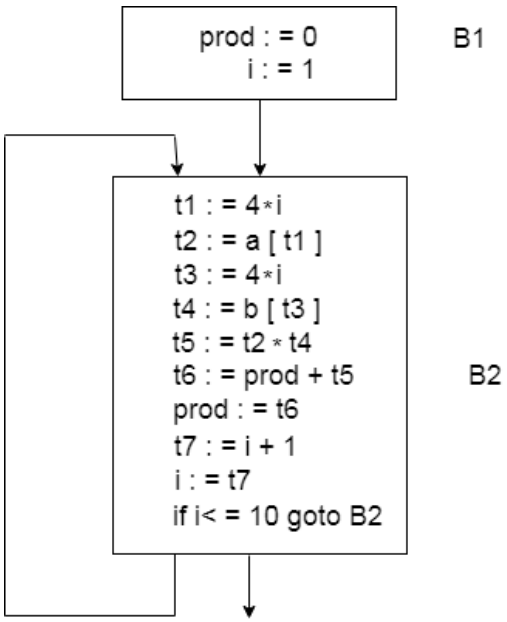
Control Flow Graph (CFG) is a directed graph. It contains the flow of control information for the set of basic block.

In the context of basic blocks:

- **Nodes (Vertices):** Each node represents a basic block.
- **Edges (Arcs):** Directed edges connect basic blocks to represent the possible flow of control from one block to another. An edge from Block A to Block B indicates that it's possible for the program to transition from executing Block A to executing Block B.

A control flow graph is used to depict that how the program control is being parsed among the blocks. It is useful in the loop optimization.

Flow graph for the vector dot product is given as follows:



- Block B1 is the initial node. Block B2 immediately follows B1, so from B2 to B1 there is an edge.
- The target of jump from last statement of B1 is the first statement B2, so from B1 to B2 there is an edge.
- B2 is a successor of B1 and B1 is the predecessor of B2.

DAG Representation of Basic Block:

• A DAG for a basic block is a directed acyclic graph with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

• DAGs are useful data structures for implementing transformations on basic blocks.

• It gives a picture of how the value computed by a statement is used in subsequent statements.

• It provides a good way of determining common sub - expressions.

Algorithm for Construction of DAG:

Input: A basic block

Output: A DAG for the basic block containing the following information:

- A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
- For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

Case (iii) x : = y

Method:

Step 1:

If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

Step 2:

For the case(i), create a node(OP) whose left child is node(y) and right child is node(z). (Checking for common sub expression). Let n be this node.

For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.

For case(iii), node n will be node(y).

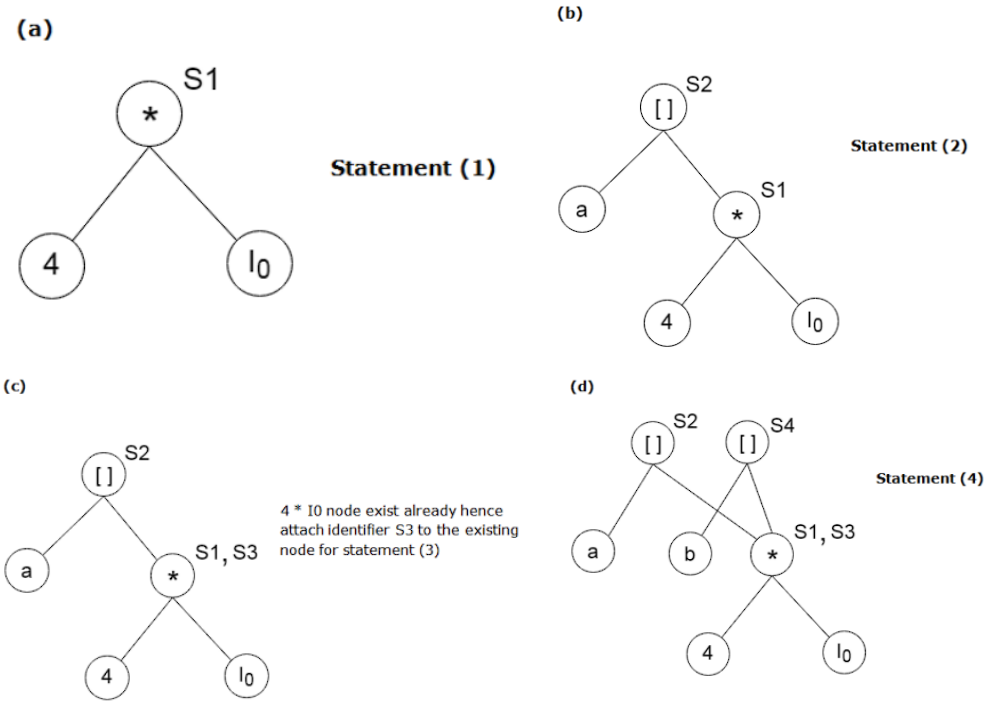
Step 3:

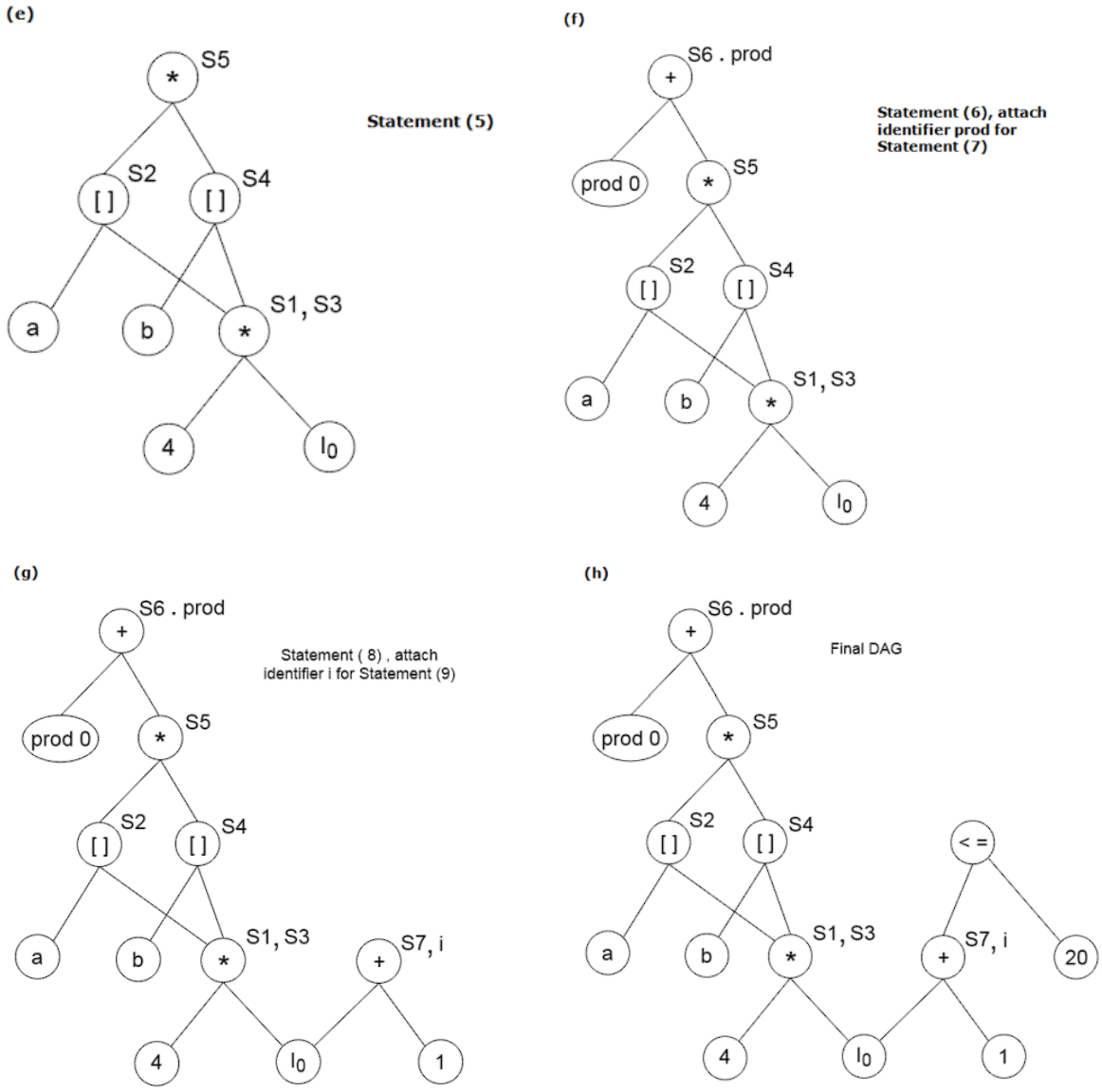
Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements

1. S1:= 4 * i
2. S2:= a[S1]
3. S3:= 4 * i
4. S4:= b[S3]
5. S5:= s2 * S4
6. S6:= prod + S5
7. Prod:= s6
8. S7:= i+1
9. i := S7
10. **if** i<= 20 **goto** (1)

Stages in DAG Construction:





Advantages of DAG:

A Directed Acyclic Graph (DAG) offers several advantages in compiler design, particularly in the context of optimizing and improving the efficiency of code generation. Here are the key benefits:

1. **Common Subexpression Elimination**: DAGs help in identifying and eliminating common subexpressions by representing them as single nodes. This reduces redundant calculations and optimizes the use of computational resources.
2. **Optimal Instruction Selection**: By using DAGs, the compiler can more effectively select the most efficient instructions, as the DAG representation makes it easier to identify the dependencies and the optimal sequence of operations.

3. **Efficient Register Allocation**: DAGs can assist in determining which variables can share the same register, as nodes that are not connected by an edge can potentially be stored in the same register, reducing the overall number of registers required.
4. **Code Motion Optimization**: DAGs allow the movement of invariant code outside loops (loop-invariant code motion), reducing the amount of code executed within loops and improving runtime efficiency.
5. **Simplification of Complex Expressions**: Complex expressions are broken down into simpler components in a DAG, making it easier for the compiler to optimize and generate efficient machine code.
6. **Improved Dead Code Elimination**: Nodes in a DAG that are not connected to the final output can be easily identified as dead code and eliminated, further optimizing the compiled program.

Sources of Optimization:

Optimization in compilers can be applied at various stages to enhance the performance and efficiency of the generated code. Here are the primary sources of optimization:

1. Peephole Optimization: A localized optimization technique that examines and improves a small set of instructions (a "peephole") in the code.

- **Examples**:
  - **Redundant Instruction Elimination**: Removing unnecessary instructions that do not affect the program's outcome.
  - **Instruction Combination**: Replacing multiple instructions with a more efficient single instruction.
  - **Constant Folding**: Evaluating constant expressions at compile-time rather than at runtime.

2. Loop Optimization: Techniques focused on improving the performance of loops, which are often performance bottlenecks.

- **Examples**:
  - **Loop Unrolling**: Reducing the overhead of loop control by duplicating the loop body multiple times.
  - **Loop-Invariant Code Motion**: Moving calculations that do not change within the loop outside the loop to avoid repeated computation.
  - **Loop Fusion**: Combining adjacent loops that iterate over the same range to reduce loop overhead.

3. Data Flow Analysis: Analyzing the flow of data within a program to optimize variable usage and instruction execution.

- **Examples**:
  - **Constant Propagation**: Replacing variables that have constant values with those values.
  - **Dead Code Elimination**: Removing code that does not affect the program's output.
  - **Live Variable Analysis**: Identifying variables that are used before being redefined to avoid unnecessary computations.

4. Inline Expansion: Replacing a function call with the body of the function to eliminate the overhead of the call.

- **Examples**:
  - **Inlining Small Functions**: Directly incorporating the function's code into the calling location to reduce call overhead.
  - **Reducing Function Call Overhead**: Minimizing the performance penalty associated with frequent function calls.

5. Register Allocation: Efficiently assigning variables to CPU registers to reduce memory access overhead.

- **Examples**:
  - **Graph Coloring**: Using graph coloring techniques to minimize the number of registers used.
  - **Spilling**: Deciding which variables to store in registers and which to keep in memory when registers are limited.

6. Control Flow Optimization: Improving the flow of control in a program to reduce unnecessary branches and jumps.

- **Examples**:
  - **Branch Prediction Optimization**: Arranging code to minimize branch mispredictions.
  - **Eliminating Unnecessary Jumps**: Removing jumps that do not contribute to program logic, streamlining control flow.

7. Memory Optimization: Reducing memory usage and access time to improve runtime performance.

- **Examples**:

- **Memory Alignment**: Aligning data in memory to match the architecture's requirements for faster access.
- **Cache Optimization**: Organizing data to maximize cache hits and minimize cache misses.

8. Instruction-Level Parallelism (ILP): Leveraging parallel execution of instructions to improve performance.

- **Examples**:
  - **Pipelining**: Breaking down instruction execution into stages that can be executed concurrently.
  - **Superscalar Execution**: Issuing multiple instructions per clock cycle in parallel.

Loop Optimization:

Loop Optimization is the process of increasing execution speed and reducing the overheads associated with loops. It plays an important role in improving cache performance and making effective use of parallel processing capabilities. Most execution time of a scientific program is spent on loops.

Here are some common loop optimization techniques:

1. Loop Unrolling:

Loop unrolling is a technique that involves expanding the loop body multiple times to reduce the overhead of loop control and increase instruction-level parallelism.

Example:

```
// Before Unrolling
for (int i = 0; i < 4; i++)
    arr[i] = arr[i] * 2;
}

// After Unrolling
arr[0] = arr[0] * 2;
arr[1] = arr[1] * 2;
arr[2] = arr[2] * 2;
arr[3] = arr[3] * 2;
```

- **Benefit**: Reduces loop overhead (such as increment and test operations) and can lead to better performance on modern processors that benefit from fewer branch instructions.

2. Loop-Invariant Code Motion:

Moving computations that do not change within the loop (loop-invariant) outside the loop to avoid repeated execution.

Example:

```
// Before Optimization
for (int i = 0; i < n; i++) {
    int temp = a * b; // Invariant code
    arr[i] = arr[i] + temp;
}

// After Optimization
int temp = a * b;
for (int i = 0; i < n; i++) {
    arr[i] = arr[i] + temp;
}
```

- **Benefit**: Reduces the number of operations executed within the loop, leading to improved runtime performance.

3. Loop Fusion (Merging):

Combining adjacent loops that iterate over the same range into a single loop to reduce loop overhead.

Example:

```
// Before Fusion
for (int i = 0; i < n; i++) {
    arr1[i] = arr1[i] + 1;
}
for (int i = 0; i < n; i++) {
    arr2[i] = arr2[i] + 2;
}

// After Fusion
for (int i = 0; i < n; i++) {
    arr1[i] = arr1[i] + 1;
    arr2[i] = arr2[i] + 2;
}
```

- **Benefit**: Reduces the loop overhead and may improve cache performance by accessing related data in a single pass.

4. Loop Fission (Loop Splitting):

Breaking a single loop that performs multiple independent operations into separate loops to improve cache performance or enable parallel execution.

Example:

```
// Before Fission
for (int i = 0; i < n; i++) {
    arr1[i] = arr1[i] + 1;
    arr2[i] = arr2[i] + 2;
}

// After Fission
for (int i = 0; i < n; i++) {
    arr1[i] = arr1[i] + 1;
}
for (int i = 0; i < n; i++) {
    arr2[i] = arr2[i] + 2;
}
```

- **Benefit**: Can improve performance by reducing cache conflicts or enabling better parallelization.

5. Loop Blocking (Tiling):

Breaking down loops operating on large data sets into smaller blocks to improve cache performance by keeping data in cache during computation.

Example:

```
// Before Blocking
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        arr[i][j] = arr[i][j] + 1;
    }
}

// After Blocking
int blockSize = 64; // Example block size
for (int i = 0; i < n; i += blockSize) {
    for (int j = 0; j < n; j += blockSize) {
        for (int ii = i; ii < i + blockSize; ii++) {
            for (int jj = j; jj < j + blockSize; jj++) {
                arr[ii][jj] = arr[ii][jj] + 1;
            }
        }
    }
}
```

- **Benefit**: Improves cache utilization by operating on smaller chunks of data that can fit into cache.

6. Loop Interchange:

Swapping the order of nested loops to improve cache performance or exploit parallelism.

Example:

```
// Before Interchange
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        arr[i][j] = arr[i][j] + 1;
    }
}

// After Interchange
for (int j = 0; j < m; j++) {
    for (int i = 0; i < n; i++) {
        arr[i][j] = arr[i][j] + 1;
    }
}
```

- **Benefit**: Can lead to better memory access patterns, reducing cache misses and improving performance.

7. Loop Vectorization:

Converting loop operations to use vector (SIMD) instructions, which process multiple data points in parallel.

Example:

```
// Before Vectorization
for (int i = 0; i < n; i++) {
    arr[i] = arr[i] * 2;
}

// After Vectorization (pseudo-code)
for (int i = 0; i < n; i += 4) {
    vec = load(arr[i]);   // Load 4 elements into vector register
    vec = vec * 2;        // Multiply all 4 elements in parallel
    store(arr[i], vec);   // Store the result back
}
```

- **Benefit**: Takes advantage of modern CPU features to perform multiple operations in a single instruction, significantly speeding up computation.

Loop Invariant Computation:

Loop-invariant computation is a compiler optimization technique that identifies calculations or expressions within a loop that produce the same result in every iteration of the loop. By moving these computations outside the loop, the compiler can reduce the number of operations performed during the loop's execution, leading to improved efficiency.

How Loop-Invariant Computation Works:

1. **Identify Invariant Computations**: The compiler analyzes the loop to determine which expressions or calculations yield the same result regardless of the loop's iteration count. These are identified as loop-invariant.
2. **Hoist Invariant Computations**: Once identified, these loop-invariant computations are moved, or "hoisted," outside of the loop, so they are executed only once before the loop begins.
3. **Modify the Loop**: The loop is then modified to remove the redundant computations, leading to a reduced workload for each iteration.

Example of Loop-Invariant Computation:

Before Optimization:

```
for (int i = 0; i < n; i++) {
    int temp = x * y;  // Loop-invariant computation
    arr[i] = arr[i] + temp;
}
```

In this example, x * y is computed in every iteration of the loop, but its value does not change, making it a loop-invariant computation.

After Optimization:

```
int temp = x * y;  // Hoisted outside the loop
for (int i = 0; i < n; i++) {
    arr[i] = arr[i] + temp;
}
```

Now, x * y is calculated only once, before the loop starts, and its value temp is reused in every iteration.

Benefits of Loop-Invariant Computation:

- **Performance Improvement**: By reducing the number of redundant calculations inside the loop, the overall execution time of the loop can be significantly decreased, especially for loops with a large number of iterations.
- **Reduced CPU Utilization**: Moving computations outside the loop reduces the number of instructions that the CPU needs to execute during each iteration, leading to more efficient use of resources.
- **Power Efficiency**: In scenarios where power consumption is a concern (e.g., in embedded systems or mobile devices), reducing unnecessary computations can also lead to lower power consumption.

Considerations:

- **Complex Expressions**: Some loop-invariant expressions may involve complex data structures or functions. The compiler needs to ensure that hoisting these expressions does not introduce side effects or alter the program's behavior.
- **Code Size Increase**: In some cases, hoisting loop-invariant computations might increase the code size slightly, as the computation is moved outside the loop. However, the performance benefits typically outweigh this downside.

Peephole Optimization:

Peephole optimization is a simple yet powerful technique used in compiler design to optimize small sequences of instructions within a program's machine code or intermediate representation. It involves examining a "peephole" (a small window) of a few adjacent instructions, identifying patterns that can be optimized, and then replacing these patterns with more efficient sequences.

Key Features of Peephole Optimization:

1. **Localized Optimization**: Peephole optimization focuses on a small section of code at a time, typically just a few instructions. This localized approach makes it relatively easy to implement and apply during the final stages of compilation.
2. **Pattern Matching**: The technique works by recognizing specific patterns or sequences of instructions that can be improved. Once a pattern is identified, it is replaced with a more efficient set of instructions.
3. **Platform-Specific**: Peephole optimizations are often tailored to the specific architecture or platform for which the code is being compiled. This allows for optimizations that take advantage of particular features or quirks of the target machine.

Common Types of Peephole Optimizations:

1. **Redundant Instruction Elimination**: This optimization technique removes instructions that do not contribute to the final result. Redundant instructions are those that duplicate previous operations or perform actions that are effectively no-ops.

   Example:

   Before Optimization:

```
int a = 5;
a = a + 0;  // Redundant operation
a = a + 0;  // Another redundant operation
```

   After Optimization:

```
int a = 5;
```

2. **Constant Folding:** This optimization evaluates expressions with constant values at compile-time rather than at runtime. It combines multiple constant operations into a single operation.

   Example:
   Before Optimization:

```
int x = 5;
int y = 3;
int z = x + y;  // Addition of constants
```

   After Optimization:

```
int z = 8;  // Directly use the result of 5 + 3
```

3. **Strength Reduction:** This technique replaces expensive operations (like multiplication) with cheaper operations (like bit shifts) when possible. It's often used for operations with constant factors.

   Example:
   Before Optimization:

```
int x = 4;
int y = x * 8;  // Multiplication
```

   After Optimization:

```
int y = x << 3;  // Multiplication by 8 is equivalent to left shift by 3
```

4. **Algebraic Simplification:** Simplifies algebraic expressions by eliminating operations that do not change the result or are redundant.

   Example:

   Before Optimization:

```
int x = 7;
int y = x - 0;  // Subtracting zero
```

   After Optimization:

```
int y = 7;  // Directly use the value of x
```

5. **Branch Optimization:** This technique removes unnecessary branch instructions, such as jumps to locations with no operations or to locations that are always executed.

   Example:

   Before Optimization:

```
if (true) {
    // do something
}
```

   After Optimization:

```
// do something  // Directly execute since condition is always true
```

6. **Null Sequence Elimination:** Removes sequences of instructions that cancel each other out or have no effect, reducing unnecessary operations.

   Example:

   Before Optimization:

```
int x = 5;
x = x - x;  // Sets x to 0
x = x + 10; // Adds 10 to x
```

   After Optimization:

```
int x = 10;  // Directly assign the result
```

Issues in the Design of Code Generator:

Code Generation:

The final phase in compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program. The code generation techniques presented below can be used whether or not an optimizing phase occurs before code generation.

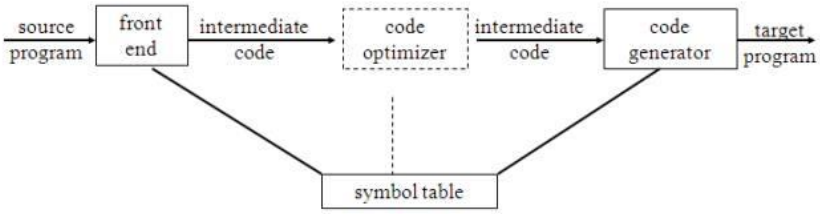


Fig. 4.1 Position of code generator

Issues in the Design of a Code Generator:

The following issues arise during the code generation phase:

1. Input to Code Generator:

- The input to the code generator contains the intermediate representation of the source program and the information of the symbol table. The source program is produced by the front end.
- Intermediate representation has the several choices:
  a) Postfix notation
  b) Syntax tree
  c) Three address code
- We assume front end produces low-level intermediate representation i.e. values of names in it can directly manipulated by the machine instructions.
- The code generation phase needs complete error-free intermediate code as an input requires.

2. Target Program:

The target program is the output of the code generator. The output can be:

a) **Assembly language:** It allows subprogram to be separately compiled.

b) **Relocatable machine language:** It makes the process of code generation easier.

c) **Absolute machine language:** It can be placed in a fixed location in memory and can be executed immediately.

3. Memory Management:

- During code generation process the symbol table entries have to be mapped to actual p addresses and levels have to be mapped to instruction address.
- Mapping name in the source program to address of data is co-operating done by the front end and code generator.
- Local variables are stack allocation in the activation record while global variables are in static area.

4. Instruction Selection:

- Nature of instruction set of the target machine should be complete and uniform.
- When you consider the efficiency of target machine then the instruction speed and machine idioms are important factors.
- The quality of the generated code can be determined by its speed and size.

Example:

The Three address code is:

a:= b + c

d:= a + e

Inefficient assembly code is:

MOV b, R0 R0→b

ADD c, R0 R0 c + R0

MOV R0, a a → R0

MOV a, R0 R0→ a

ADD e, R0 R0 → e + R0

MOV R0, d d → R0

5. Register Allocation:

Register can be accessed faster than memory. The instructions involving operands in register are shorter and faster than those involving in memory operand.

The following sub problems arise when we use registers:

Register allocation: In register allocation, we select the set of variables that will reside in register.

Register assignment: In Register assignment, we pick the register that contains variable.

Certain machine requires even-odd pairs of registers for some operands and result.

For example:

Consider the following division instruction of the form:

D x, y

Where,

x is the dividend even register in even/odd register pair

y is the divisor

Even register is used to hold the reminder.

Old register is used to hold the quotient.

6. Evaluation Order:

The efficiency of the target code can be affected by the order in which the computations are performed. Some computation orders need fewer registers to hold results of intermediate than others.

A Simple Code Generator:

A code generator generates target code for a sequence of three- address statements and effectively uses registers to store operands of the statements.

For example: consider the three-address statement a := b+c It can have the following sequence of codes:

ADD Rj, Ri Cost = 1

(or)

ADD c, Ri Cost = 2

(or)

MOV c, Rj Cost = 3

ADD Rj, Ri

Register and Address Descriptors:

• A register descriptor is used to keep track of what is currently in each register. The register descriptors show that initially all the registers are empty.

• An address descriptor stores the location where the current value of the name can be found at run time.

A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z, perform the following actions:

1. Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.

2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y' , L to place a copy of y in L.

3. Generate the instruction OP z' , L where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.

4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z

Generating Code for Assignment Statements:

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

```
t : = a - b
u : = a - c
v:= t+ u
d : = v + u
```
with d live at the end.

Code sequence for the example is:

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

d : = v + u	ADD R1, R0
	
	MOV R0, d
		R0 contains d	d in R0
			d in R0 and memory

Generating Code for Indexed Assignments:

The table shows the code sequences generated for the indexed assignmen a:= b[i] and a[i]:= b

Statements	Code Generated	Cost
a : = b[i]	MOV b(Ri), R	2
a[i] : = b	MOV b, a(Ri)	3

Generating Code for Pointer Assignments:

The table shows the code sequences generated for the pointer assignments a : = *p and *p : = a

Statements	Code Generated	Cost
a : = *p	MOV *Rp, a	2
*p : = a	MOV a, *Rp	2

Generating Code for Conditional Statements

Statement	Code
if x < y goto z	CMP x, y CJ< z /* jump to z if condition code is negative */
x : = y +z	MOV y, R0

if x < 0 goto z ADD z, R0

MOV R0,x

CJ< z

Code Generation From DAG:

The advantage of generating code for a basic block from its DAG representation is that from a DAG we can easily see how to rearrange the order of the final computation sequence than we can start from a linear sequence of three-address statements or quadruples.

Rearranging the order:

The order in which computations are done can affect the cost of resulting object code. For example, consider the following basic block:

t1 : = a + b

t2 : = c + d

t3 : = e - t2

t4 : = t1 - t3

Generated code sequence for basic block:

MOV a , R0

ADD b , R0

MOV c , R1

ADD d , R1

MOV R0 , t1

MOV e , R0

SUB R1 , R0

MOV t1 , R1

SUB R0 , R1

MOV R1 , t4

Rearranged basic block:

Now t1 occurs immediately before t4.

t2 : = c + d

t3 : = e - t2

t1 : = a + b

t4 : = t1 - t3

Revised code sequence:

MOV c , R0

ADD d , R0

MOV a , R0

SUB R0 , R1

MOV a , R0

ADD b , R0

SUB R1 , R0

MOV R0 , t4

In this order, two instructions **MOV R0 , t1 and MOV t1 , R1** have been saved.

A Heuristic ordering for DAGs:

The heuristic ordering algorithm attempts to make the evaluation of a nod the evaluation of its leftmost argument. The algorithm shown below produces the ordering in reverse.

Algorithm:

1) while unlisted interior nodes remain do begin

2) select an unlisted node n, all of whose parents have been listed;

3) list n;

4) while the leftmost child m of n has no unlisted parents and is not a leaf do

begin

5) list m;

6) n : = m

end

end

Example: Consider the DAG shown below

Initially, the only node with no unlisted parents is 1 so set n=1 at line (2) and list 1 at line (3). Now, the left argument of 1, which is 2, has its parents listed, so we list 2 and set n=2 at line (6). Now, at line (4) we find the leftmost child of 2, which is 6, has an unlisted parent 5. Thus, we select a new n at line (2), and node 3 is the only candidate. We list 3 and proceed down its left chain, listing 4, 5 and 6. This leaves only 8 among the interior nodes so we list that. The resulting list is 1234568 and the order of evaluation is 8654321.

Code sequence:

t8 : = d + e

t6 : = a + b

t5 : = t6 - c

t4 : = t5 * t8

t3 : = t4 - e

t2 : = t6 + t4

t1 : = t2 * t3

This will yield an optimal code for the DAG on machine whatever be the number of registers.

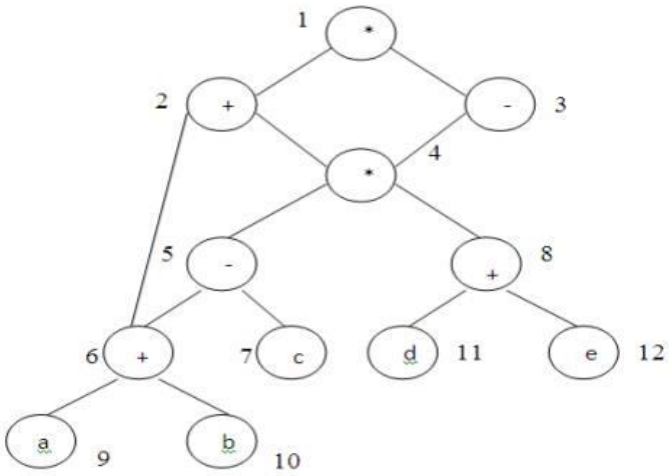
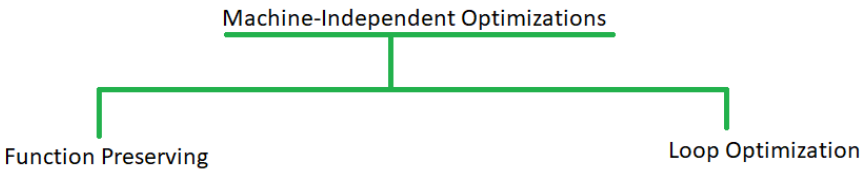


Fig. 4.7 A DAG

Machine Independent Optimization:

Machine Independent code optimization tries to make the intermediate code more efficient by transforming a section of code that doesn't involve hardware components like CPU registers or any absolute memory location. Generally, it optimizes code by eliminating redundancies, reducing the number of lines in code, eliminating useless code or reducing the frequency of repeated code. Thus, can be used on any processor irrespective of machine specifications.

Machine independent code optimization can be achieved using the following methods:



Function Preserving Optimization:

Function Preserving optimizations deals with the code in a given function in an attempt of reducing the computational time. It can be achieved by following methods:

1. Common Subexpression elimination
2. Folding
3. Dead code elimination
4. Copy Propagation

1. Common Subexpression Elimination:

A common subexpression is the one which was computed and doesn't change after it last computation, but is often repeated in the program. The compiler evaluates its value even if it does not change. Such evaluations result in wastage of resources and time. Thus, it better be eliminated. Consider an example:

```
//Code snippet in three address code format
t1=x*z;
t2=a+b;
t3=p%t2;
t4=x*z;     //t4 and t1 is same expression
            //but evaluated twice by compiler.
t5=a-z;

// after Optimization
t1=x*z;
t2=a+b;
t3=p%t2;
t5=a-z;
```

It is troublesome if a common subexpression is often repeated in a program. Thus, it needs to be eliminated.

2. Constant Folding:

Constant Folding is a technique where the expression which is computed at compile time is replaced by its value. Generally, such expressions are evaluated at runtime, but if we replace them with their values they need not be evaluated at runtime, saving time.

```
//Code segment
int x= 5+7+c;

//Folding applied
int x=12+c;
```

Folding can be applied on boolean, integers as well as on floating point numbers but one should be careful with floating point numbers. Constant folding is often interleaved with constant propagation.

Constant Propagation:

If any variable is assigned a constant value and used in further computations, constant propagation suggests using the constant value directly for further computations. Consider the below example

```
// Code segment
int a = 5;
int c = b * 2;
int z = a;

//Applying constant propagation once
int c = 5 * 2;
int z = a;

//Applying constant propagation second time
int c = 10;
int z = a;

//Applying constant propagation last time
int z = a[10];
```

3. Dead Code Elimination:

Dead code is a program snippet that is never executed or never reached in a program. It is a code that can be efficiently removed from the program without affecting any other part of the program. In case, a value is obtained and never used in the future, it is also regarded as dead code. Consider the below dead code:

```
//Code
int x= a+23;  //the variable x is never used
              //in the program. Thus it is a dead code.
z=a+y;
printf("%d,%d".z,y);

//After Optimization
z=a+y;
printf("%d,%d".z,y);
```

Another example of dead code is assign a value to a variable and changing that value just before using it. The previous value assignment statement is dead code. Such dead code needs to be deleted in order to achieve optimization.

4. Copy Propagation:

Copy Propagation suggests to use one variable instead of other, in cases where assignments of the form x=y are used. These assignments are copy statements. We can efficiently use y at all required place instead of assign it to x. In short, elimination of copies in the code is Copy Propagation.

```
//Code segment
----;
a=b;
z=a+x;
x=z-b;
----;

//After Optimization
----;
z=b+x;
x=z-b;
----;
```

Another kind of optimization**, loop optimization** deals with reducing the time a program spends inside a loop. **(Explained Earlier)**

Global Data Flow Analysis:

- To efficiently optimize the code compiler collects all the information about the program and distribute this information to each block of the flow graph. This process is known as data-flow graph analysis.
- Certain optimization can only be achieved by examining the entire program. It can't be achieved by examining just a portion of the program.
- For this kind of optimization user defined chaining is one particular problem.
- Here using the value of the variable, we try to find out that which definition of a variable is applicable in a statement.

Based on the local information a compiler can perform some optimizations. For example, consider the following code:

```
x = a + b;

 x = 6 * 3
```

- In this code, the first assignment of x is useless. The value computer for x is never used in the program.
- At compile time the expression 6*3 will be computed, simplifying the second assignment statement to x = 18;

Some optimization needs more global information. For example, consider the following code:

```
a = 1;

  b = 2;

  c = 3;

  if (....) x = a + 5;

  else x = b + 4;

  c = x + 1;
```

In this code, at line 3 the initial assignment is useless and x +1 expression can be simplified as 7.

But it is less obvious that how a compiler can discover these facts by looking only at one or two consecutive statements. A more global analysis is required so that the compiler knows the following things at each point in the program:

- Which variables are guaranteed to have constant values
- Which variables will be used before being redefined

Data flow analysis is used to discover this kind of property. The data flow analysis can be performed on the program's control flow graph (CFG).

The control flow graph of a program is used to determine those parts of a program to which a particular value assigned to a variable might propagate.

Liveness Analysis:

Liveliness Analysis consists of a specified technique that is implemented to optimize register space allocation, for a given piece of code and facilitate the procedure for dead-code elimination. As any machine has a limited number of registers to hold a variable or data which is being used or manipulated, there exists a need to balance out the efficient allocation of memory to reduce cost and also enable the machine to handle complex code and a considerable amount of variables at the same time. The procedure is carried out during the compilation of an input code by a compiler itself.

Key Concepts in Liveness Analysis:

Live Variable: A variable is *live* at any instant of time, during the process of compilation of a program if its value is being used to process a computation as the evaluation of an arithmetic operation at that instant or it holds a value that will be used in the future without the variable being re-defined at any intermediate step.

Live Range: The *live range* of a variable is defined as the portion of code for which a variable was *live.* The *live range* of a variable might be continuous or distributed across different portions of the code. This implies that a variable might be *live* in an instant and *dead* in the next and might be *live* again for a certain portion.

Def-Use Chain: This is a relationship that connects a definition of a variable (where it is assigned a value) with its uses (where its value is read). It helps in understanding the flow of data and performing optimizations.

Live-In Set: The set of variables that are live at the entry of a basic block (a single entry, single exit segment of code).

Live-Out Set: The set of variables that are live at the exit of a basic block.

How Liveness Analysis Works:

1. **Initialization**: Start with an initial guess. Typically, you assume that all variables are live at the end of the program.
2. **Iterative Process**:

- For each basic block, calculate the Live-Out set.
- Compute the Live-In set for each block based on the Live-Out set and the definitions and uses within the block.
- Update the Live-In and Live-Out sets until they converge (i.e., do not change with further iterations).

3. **Algorithm**:

- **Live-Out(B)**: The union of Live-In sets of all successor blocks of block B.
- **Live-In(B)**: The union of variables that are used in B (and not defined in B), plus the Live-Out set of B.

Example: Consider the following simple code snippet:

```
1:  int a = 5;
2:  int b = a + 2;
3:  int c = b * 3;
4:  int d = c - a;
5:  print(d);
```

Let's analyze liveness at each line:

- **Line 1**: a is defined. a is not live at this point but becomes live in subsequent lines.
- **Line 2**: b is defined and used in line 3. Hence, b is live here.
- **Line 3**: c is defined and used in line 4. Hence, c is live here.
- **Line 4**: d is defined and used in the print statement. Hence, d is live here.
- **Line 5**: d is used, so d is live here.

In this example, you would track the live variables and ensure that each variable is stored or moved appropriately to avoid unnecessary computations or memory usage.

Benefits:

- **Register Allocation**: Helps in efficient allocation of registers by understanding which variables need to be kept in registers.
- **Dead Code Elimination**: Identifies and removes code that computes values that are never used.