Chapter 4

Code Optimization

LEARNING OBJECTIVES

- *Code optimization basics*
- *Principle sources of optimization*
- *Loop invariant code motion*
- *Strength reduction on induction variables*
- *Loops in flow graphs*
- *Pre-header*
- *Global data flow analysis*
- *Definition and usage of variables*
- *Use-definition (u-d) chaining*
- *Data flow equations*

Code Optimization Basics

The process of improving the intermediate code and the target code in terms of both speed and the amount of memory required for execution is known as code optimization.

Compilers that apply code–improving transformations are called optimizing compilers.

Properties of the transformations of an optimizing compiler are

1. A transformation must preserve the meaning of programs.
2. It must speed up programs by a measurable amount.
3. A transformation must be worth the effort.

Places for improvements

1. Source Code:
   User can – profile a program
   – change an algorithm
   – transform loops
2. Intermediate code can be improved by improving
   – Loops
   – Procedure calls
   – Address calculations
3. Target code can be improved by
   – Using registers
   – Selecting instructions
   – Peephole transformations

Optimizing compiler organization

This applies
- Control flow analysis
- Data flow analysis
- Transformations

Issues in design of code optimization The issues in the design of code optimization are
1. Target machine characteristics
2. Target CPU architecture
3. Functional units

Target machine Optimization is done, according to the target machine characteristics. Altering the machine description parameters, one can optimize single piece of compiler code.

Target CPU architecture The issues to be considered for the optimization with respect to CPU architecture
1. Number of CPU registers
2. RISC Instruction set
3. CISC instruction set
4. Pipelining

Functional units Based on number of functional units, optimization is done. So that instructions can be executed simultaneously.

Principle Sources of Optimization

Some code improving transformation is Local transformations and some are Global transformations.

Local Transformations can be performed by looking only at a statement in a basic block. Otherwise it is global transformation.

Function Preserving Transformations

These transformations improve the program without changing the function it computes. Some of these transformations are

1. Common sub expression elimination
2. Copy propagation
3. Dead-code elimination
4. Loop optimization
   - Code motion
   - Induction variable elimination
   - Reduction in strength

Common sub expression elimination The process of identifying common sub expressions and eliminating their computation multiple times is known as common sub expression elimination.

Example: Consider the following program segment:

```
int sum_n, sum_n2, sum_n3;
int sum (int n)
{
Sum_n = ((n)*(n+1))/2;
sum_n2 = ((n)*(n+1)*(2n+1))/6;
sum_n3 = (((n)*(n+1))/2)*(((n)*(n+1))/2;
}
```

Three Address code for the above input is

(0) Proc-begin sum
(1) t_0: = n + 1
(2) t_1: = n * t_0
(3) t_2: = t_1/2
(4) sum_n = t_2
(5) t_3: = n + 1
(6) t_4: = n * t_3
(7) t_5: 2 * n
(8) t_6: = t_5 + 1
(9) t_7: = t_4 * t_6
(10) t_8: = t_7/6
(11) sum_n_2: = t_8
(12) t_9: = n + 1
(13) t_{10}: n * t_9
(14) t_{11}: t_{10}/2
(15) t_{12}: = n + 1
(16) t_{13}: = n * t_{12}
(17) t_{14}: = t_{13}/2
(18) t_{15}: = t_{11} * t_{14}
(19) sum_n_3: = t_{15}
(20) label L_o
(21) Proc end sum

The computations made in quadruples

(1) – (3), (12) – (14), (15) – (17) are essentially same. That is, ((n)*(n + 1))/2 is computed.

It is the common sub expression.

This common sub expression is computed four times in the above example.

It is possible to optimize the code to have common sub expressions computed only once and then reuse the computed values further.

∴ Optimized intermediate code will be

(0) proc-begin sum
(1) t_0: = n + 1
(2) t_1: = n * t_0
(3) sultan: = t_1/2
(4) t_5: = 2 * n
(5) t_6: = t_5 + 1
(6) t_7: = t_1 * t_6
(7) sum_n2: = t_7/6
(8) sum_n3: sum_n * sum_n
(9) proc-end sum

Constant folding The constant expressions in the input source are evaluated and replaced by the equivalent values at the time of compilation.

For example 10*3, 6 + 101 are constant expressions and they are replaced by 30, 107 respectively.

Example: Consider the following '*C*' code:

```
int arr1 [10];
int main ( )
{
   arr1 [0] = 3;
   arr1 [1] = 4;
}
```

Unoptimized three address code equivalent to the above '*C*' code is

(0) proc-begin main
(1) t_0: = 0*4
(2) t_1: = &arr1
(3) t_1 [t_0]: = 3
(4) t_2: = 1*4
(5) t_3: = &arr1
(6) t_3 [t_2]: = 4
(7) Label L_0
(8) Proc – end main

In the above code, 0*4 is a constant expression its value = 0. 1*4 is a constant expression, its value = 4.

∴ After applying constant folding, optimized code will be

(0) proc-begin main
(1) t_0: = 0
(2) t_1: = &arr1
(3) t_1 [t_0]: = 3
(4) t_2: = 4

(5) t_3: = &arr1
(6) t_3 [t_2]: = 4
(7) label L_0
(8) proc – end main

Copy propagation In copy propagation, if there is an expression $x = y$ then use the variable 'y' instead of 'x'. This propagated in the statements following $x = y$.

Example: In the previous example, there are two copy statements.

(1) $t_0 = 0$
(2) $t_2 = 4$

After applying copy propagation, the optimized code will be

(0) proc-begin main
(1) t_0: = 0
(2) t_1: = &arr1
(3) t_1 [0]: = 3
(4) t_2: = 4
(5) t_3: = &arr1
(6) t_3 [4]: = 4
(7) Label L_0
(8) proc-end main

In the three address code shown above, quadruples (1) and (4) are no longer used in any of the following statements.

∴ (1) and (4) can be eliminated.

Three address code after dead store elimination

(0) proc-begin main
(1) t_1: = &arr1
(2) t_1 [0]: = 3
(3) t_3: = &arr1
(4) t_3 [4]: = 4
(5) Label L_0
(6) proc-end main

In the above example, we are propagating constant values. It is also known as constant propagation.

Variable propagation Propagating another variable instead of the existing one is known as variable propagation.

Example: int func(int *a*, int *b*, int *c*)

```
{
    int d, e, f;
    d = a;
    If (a > 10)
    {
       e = d + b;
    }
    Else
    {
       e = d + c;
    }
    f = d*e;
    return (f);
}
```

Three address code (unoptimized):

(0) proc-begin func
(1) $d := a$
(2) if a >10 goto L_0
(3) goto L_1
(4) label : L_0
(5) $e := d + b$
(6) goto L_2
(7) label : L_1
(8) $e := d + c$
(9) label : L_2
(10) $f := d * e$
(11) return f
(12) goto L_3
(13) label : L_3
(14) proc-end func

Three address code after variable (copy) propagation:

(0) proc-begin func
(1) d: = a
(2) If a >10 goto .L_0
(3) goto L_1
(4) label: L_0
(5) e: = $a + b$
(6) goto L_2
(7) label: L_1
(8) e: = $a + c$
(9) label: L_2
(10) f: = a*e
(11) return f
(12) goto L_3
(13) label: L_3
(14) proc-end func

After dead store elimination:
In the above code (1) d: = a is no more used
∴ Eliminate the dead store d: = a

(0) proc-begin func
(1) If a > 10 goto L_0
(2) goto L_1
(3) label: L_0
(4) e: = $a + b$
(5) goto L_2
(6) label: L_1
(7) e: $a + c$
(8) label: L_2
(9) f: = a*e
(10) return f
(11) goto L_3
(12) label: L_3
(13) proc-end func

Dead code elimination Eliminating the code that never gets executed by the program is known as Dead code elimination. It reduces the memory required by the program

Example: Consider the following Unoptimized Intermediate code:

(0) proc-begin func
(1) debug: = 0
(2) If debug = = 1 goto L_0
(3) goto L_1
(4) label: L_0
(5) param c
(6) param b
(7) param a
(8) param lc1
(9) call printf 16
(10) retrieve to
(11) label: L_1
(12) t_1: = $a + b$
(13) t_2: = $t_1 + c$
(14) v_1: = t_2
(15) Return v_1
(16) goto L_2
(17) label: L_2
(18) proc-end func

In copy propagation, debug is replaced with 0, wherever debug is used after that assignment.

∴ Statement 2 will be changed as

If 0 = = 1 goto L_0

0 = = 1, always returns false.

∴ The control cannot flow to label: L_0

This makes the statements (4) through (10) as dead code. (2) Can also be removed as part of dead code elimination. (1) Cannot be eliminated, because 'debug' is a global variable. The optimized code after elimination of dead code is shown below.

(0) proc-begin func
(1) debug: = 0
(2) goto L_1
(3) label: L_1
(4) t_1: = $a + b$
(5) t_2: = $t_1 + c$
(6) v_1: = t_2
(7) return v_1
(8) goto L_2
(9) label: L_2
(10) proc-end func

Algebraic transformations We can use algebraic identities to optimize the code further. For example

Additive Identity: $a + 0 = a$

Multiplicative Identity: a*1 = a

Multiplication with 0: a*0 = 0

Example: Consider the following code fragment:

```
struct mystruct
{
int a [20];
int b;
} xyz;
int func(int i)
{
xyz.a[i] = 34;
}
```

The Unoptimized three address code:

(0) proc-begin func
(1) t_0: = &xyz
(2) t_1: = 0
(3) t_2: = i*4
(4) t_1: = $t_2 + t_1$
(5) t_0 [t_1] = 34
(6) label: L_0
(7) proc-end func

Optimized code after copy propagation and dead code elimination is shown below:

The statement t_1: = 0 is eliminated.

(0) proc-being func
(1) t_0 =: = &xyz
(2) t_2: = i*4
(3) t_1 : = $t_2 + 0$
(4) t_0 [t_1]: = 34
(5) label: L_0
(6) proc-end func

After applying additive identity:

(0) proc-begin func
(1) t_0: = &xyz
(2) t_2: = i*4
(3) t_1 : = t_2
(4) t_0 [t_1]: = 34
(5) label: L_0
(6) proc-end func

After copy propagation and dead store elimination:

(0) proc-begin func
(1) t_0: = &xyz
(2) t_2 : = i*4
(3) t_0 [t_2]: = 34
(4) label: L_0
(5) proc-end func

Strength reduction transformation This transformation replaces expensive operators by equivalent cheaper ones on the target machine.

For example $y := x*2$ is replaced by $y := x + x$ as addition is less expensive than multiplication.

Similarly
Replace $y := x*32$ by $y := x << 5$
Replace $y := x/8$ by $y := x >> 3$

Loop optimization We can optimize loops by

(1) Loop invariant code motion transformation.
(2) Strength reduction on induction variable transformation.

Loop invariant code motion

The statements within a loop that compute value, which do not vary throughout the life of the loop are called loop invariant statements.

Consider the following program fragment:

```
int a [100];
int func(int x, int y)
{
int i;
int n1, n2;
i = 0;
n1 = x*y;
n2 = x - y;
while (a[i] > (n1*n2))
i = i + 1;
return(i);
}
```

The Three Address code for above program is

(0) proc-begin func
(1) $i := 0$
(2) $n_1 := x*y$
(3) $n_2 := x - y$
(4) label : L_0
(5) $t_2 := i*4$
(6) $t_3 := \&arr$
(7) $t_4 := t_3[t_2]$
(8) $t_5 := n_1 * n_2$
(9) if $t_4 > t_5$ goto L_1
(10) goto L_2
(11) label : L_1
(12) $i := i + 1$
(13) goto L_0
(14) label : L_2
(15) return i
(16) goto L_3
(17) label : L_3
(18) proc-end func

In the above code statements (6) and (8) are invariant.

After loop invariant code motion transformation the code will be

(0) proc-begin func
(1) $i := 0$
(2) $n_1 := x*y$
(3) $n_2 := x-y$
(4) $t_3 := \&arr$
(5) $t_5 := n_1*n_2$
(6) label : L_0
(7) $t_2 := i*4$
(8) $t_4 := t_3[t_2]$
(9) if $t_4 > t_5$ goto L_1
(10) goto L_2
(11) label : L_1
(12) $i := i + 1$
(13) goto L_0
(14) label : L_2
(15) return i
(16) goto L_3
(17) label : L_3
(18) proc-end func

Strength reduction on induction variables

Induction variable: A variable that changes by a fixed quantity on each of the iterations of a loop is an induction variable.

Example: Consider the following code fragment:

```
int i;
int a[20];
int func( )
{
    while(i<20)
    {
    a[i] = 10;
    i = i + 1;
    }
}
```

The three-address code will be

(0) proc-begin func
(1) label : L_0
(2) if $i < 20$ goto L_1
(3) goto L_2
(4) label : L_1
(5) $t_0 := i*4$
(6) $t_1 := \&a$
(7) $t_1[t_0] := 10$
(8) $i := i + 1$
(9) goto L_0
(10) label : L_2
(11) label : L_3
(12) proc-end func

After reduction of strength the code will be
Here (5) $t_0 = i*4$ is moved out of the loop and (8) is followed by $t_0 = t_0 + 4$.

(0) proc-begin func
(0a) $t_0 := i*4$
(1) label : L_0
(2) if $i < 20$ goto L_1
(3) goto L_2
(4) label:L_1
(5)
(6) $t_1 := \&a$
(7) $t_1[t_0] := 10$
(8) $i := i + 1$
(8a) $t_0 := t_0 + 4$
(9) goto L_0
(10) label : L_2
(11) label : L_3
(12) proc-end func

Loops in Flow Graphs

Loops in the code are detected during the data flow analysis by using the concept called 'dominators' in the flow graph.

Dominators

A node d of a flow graph dominates node n, if every path from the initial node to 'n' goes through 'd'.

It is represented as d dom n.

Notes:

1. Each and every node dominates itself.
2. Entry of the loop dominates all nodes in the loop.

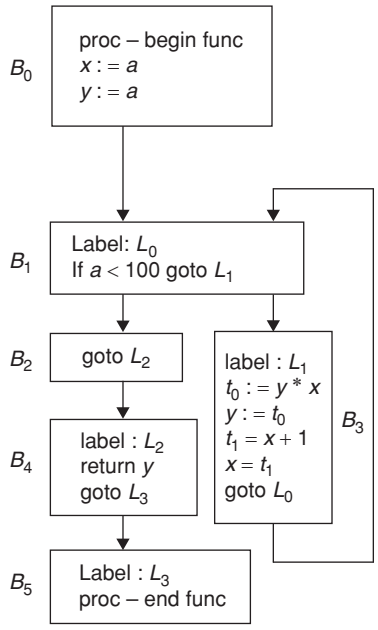
Example: Consider the following code fragment:

```
int func(int a)
{
int x, y;
x = a;
y = a;
While (a < 100)
{
y = y*x;
x = x+1;
}
return(y);
}
```

The Three Address code after local optimization will be

(0) proc-begin func
(1) $x := a$
(2) $y := a$
(3) label: L_0
(4) if $a < 100$ goto L_1
(5) goto L_2
(6) label: L_1
(7) $t_0 := y*x$
(8) $y := t_0$
(9) $t_1 := x + 1$
(10) $x := t_1$
(11) goto L_0
(12) label: L_2
(13) return y
(14) goto L_3
(15) label: L_3
(16) proc-end func

The Flow Graph for above code will be:



To reach B_2, it must pass through B_1
$\therefore$ B_1 dominates B_2. Also B_0 dominates B_2.
dominators $[B_1] = \{B_0, B1\}$ (or) dominators $[1] = \{0, 1\}$

The dominators for each of the nodes in the flow graph are

dominators [0] = {0}
dominators [1] = {0, 1}
dominators [2] = {0, 1, 2}
dominators [3] = {0, 1, 3}
dominators [4] = {0, 1, 2, 4}
dominators [5] = {0, 1, 2, 4, 5}

Edge

An edge in a flow graph represents a possible flow of control.

In the flow graph, B_0 to B_1 edge is represented as $0 \rightarrow 1$.

Head and tail: In the edge $a \rightarrow b$, the node b is called head and the node a is called as tail.

Back edges: There are some edges in which dominators [tail] contains the head.

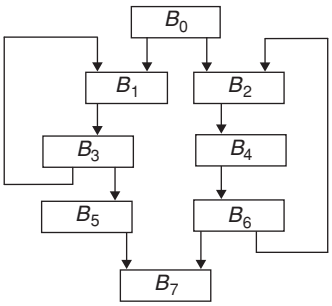
The presence of a back edge indicates the existence of a loop in a flow graph.

In the previous graph, $3 \rightarrow 1$ is a back edge.

Consider the following table:

Edge	Head	Tail	Dominators [head]	Dominators [tail]
$0 \rightarrow 1$	1	0	{0, 1}	{0}
$1 \rightarrow 2$	2	1	{0, 1, 2}	{0, 1}
$1 \rightarrow 3$	3	1	{0, 1, 3}	{0, 1}
$3 \rightarrow 1$	1	3	{0, 1}	{0, 1, 3}
$2 \rightarrow 4$	4	2	{0, 1, 2, 4}	{0, 1, 2}
$4 \rightarrow 5$	5	4	{0, 1, 2, 4, 5}	{0, 1, 2, 4}

Example: Consider below flow graph:



The dominators of each node are

dominators [0] = {0}
dominators [1] = {0, 1}
dominators [2] = {0, 2}
dominators [3] = {0, 1, 3}
dominators [4] = {0, 2, 4}
dominators [5] = {0, 1, 3, 5}
dominators [6] = {0, 2, 4, 6}
dominators [7] = {0, 7}

Edge	Head	Tail	Dominators [head]	Dominators [tail]
$0 \rightarrow 1$	1	0	{0, 1}	{0}
$0 \rightarrow 2$	2	0	{0, 2}	{0}
$1 \rightarrow 3$	3	1	{0, 1, 3}	{0, 1}
$3 \rightarrow 1$	1	3	{0, 1}	{0, 1, 3}
$3 \rightarrow 5$	5	3	{0, 1, 3, 5}	{0, 1, 3} Backedge
$5 \rightarrow 7$	7	5	{0, 7}	{0, 1, 3, 5}
$2 \rightarrow 4$	4	2	{0, 2, 4}	{0, 2}
$6 \rightarrow 2$	2	6	{0, 2}	{0, 2, 4, 6} Backedge
$4 \rightarrow 6$	6	4	{0, 2, 4, 6}	{0, 2, 4}
$6 \rightarrow 7$	7	6	{0, 7}	{0, 2, 4, 6}

Here $\{B_6, B_2, B_4\}$ form a loop (L_1), $\{B_3, B_1\}$ form another loop (L_2)

In a loop, the entry of the loop dominates all nodes in the loop.

Header of the loop The entry of the loop is also called as the header of the loop.

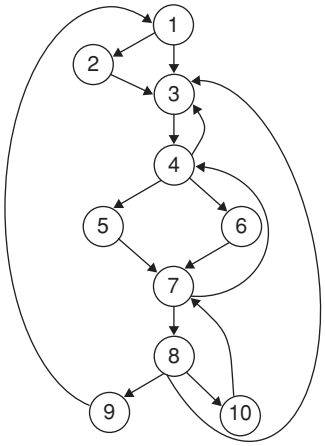
Loop exit block In loop L_1 can be exited from the basic block B_6. It is called loop exit block. The block B_3 is the loop exit block for the loop L_2. It is possible to have multiple exit blocks in a loop.

Dominator tree

A tree, which represents dominate information in the form of tree is a dominator tree. In this,

- The initial node is the root.
- Each node d dominates only its descendents in the tree.

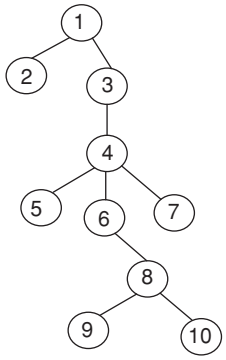
Consider the flow graph



The dominators of each node are

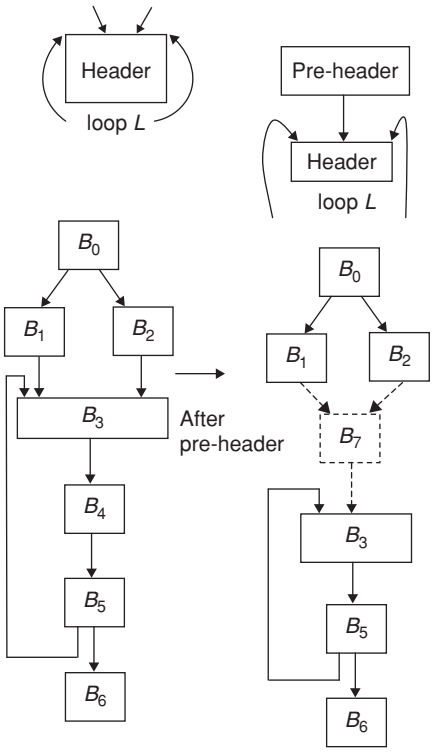
dominators [1] = {1}
dominators [2] = {1, 2}
dominators [3] = {1, 3}
dominators [4] = {1, 3, 4}
dominators [5] = {1, 3, 4, 5}
dominators [6] = {1, 3, 4, 6}
dominators [7] = {1, 3, 4, 7}
dominators [8] = {1, 3, 4, 7, 8}
dominators [9] = {1, 3, 4, 7, 8, 9}
dominators [10] = {1, 3, 4, 7, 8, 10}

The dominator tree will be:



Pre-header

A pre-header is a basic block introduced during the loop optimization to hold the statements that are moved from within the loop. It is a predecessor to the header block.



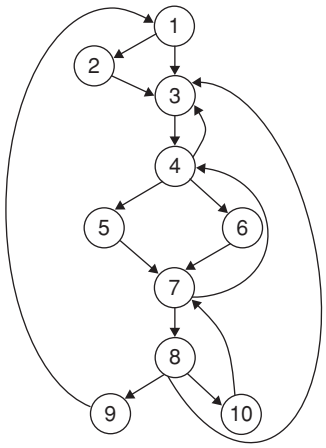
Reducible Flow Graphs

A flow graph G is reducible if and only if we can partition the edges into two disjoint groups:

(1) Forward edges
(2) Backward edges with the following properties.

(i) The forward edges form an acyclic graph in which every node can be reached from the initial node of G.
(ii) The back edges consist only of edges whose heads dominates their tails.

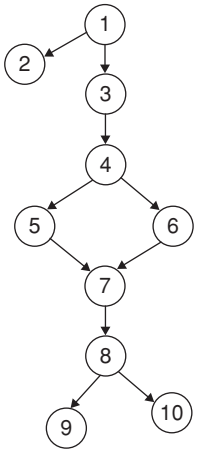
Example: Consider previous flow graph



In the above flow graph, there are five back edges

$$4 \to 3,\ 7 \to 4,\ 8 \to 3,\ 9 \to 1 \text{ and } 10 \to 7$$

Remove all backedges.
The remaining edges must be the forward edges.
The remaining graph is acyclic.



$\therefore$ It is reducible.

Global Dataflow Analysis

Point: A point is a place of reference that can be found at

1. Before the first statement in a basic block.
2. After the last statement in a basic block.
3. In between two adjacent statements within a basic block.

Example 1:

$$\boxed{\begin{array}{l} a = 10 \\ b = 20 \\ c = a * b \end{array}}\ B_1$$

Here, In B_1 there are 4 points

Example 2:

B_1
- $P_1 - B_1$ proc-begin func
- $P_2 - B_1$ $v_3 = v_1 + v_2$
- $P_3 - B_1$ if $c > 100$ goto L_0
- $P_4 - B_1$

There is 4 point in the basic block B_1, given by $P_1 - B_1$, $P_2 - B_1$, $P_3 - B_1$ and $P_4 - B_1$.

Path: A path is a sequence of points in which the control can flow.

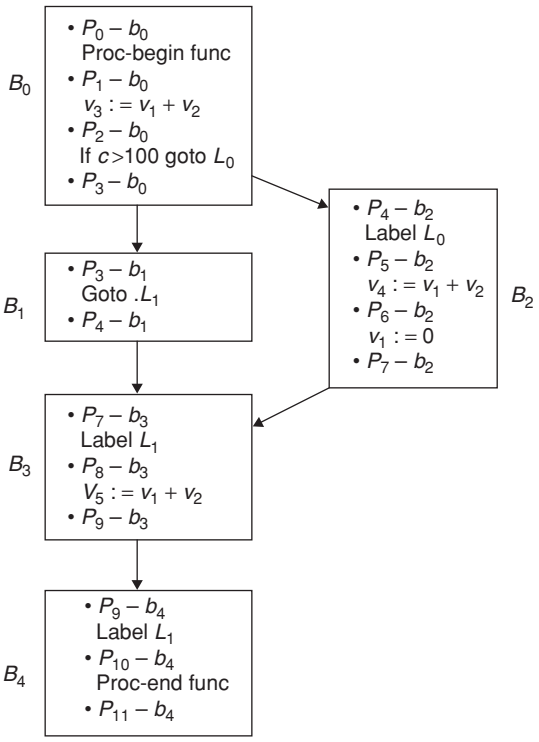
A path from P_1 to P_n is a sequence of points $P_1, P_2, \ldots, P_n$ such that for each i between 1 and n-1, either

(a) P_i is the point immediately preceding a statement and P_{i+1} is the point immediately following that statement in the same block.

(OR)

(b) P_i is the end of some block and P_{i+1} is the beginning of a successor block.

Example:



Path is between the points $P_0 - b_0$ and $P_6 - b_2$:

The sequence of points $P_0 - b_0$, $P_1 - b_0$, $P_2 - b_0$, $P_3 - b_0$, $P_4 - b_2$, $P_5 - b_2$ and $P_6 - b_2$.

Path between $P_3 - b_1$ and $P_6 - b_2$: There is no sequence of points.

Path between $P_0 - b_0$ and $P_7 - b_3$: There are two paths.

(1) Path 1 consists of the sequence of points, $P_0 - b_0, P_1 - b_0$, $P_2 - b_0, P_3 - b_0, P_3 - b_0, P_4 - b_1$ and $P_7 - b_3$.

(2) Path 2 consists of the sequence of points $P_0 - b_0, P_1 - b_0$, $P_2 - b_0, P_3 - b_0, P_4 - b_2, P_5 - b_2, P_6 - b_2, P_7 - b_2$ and $P_7 - b_3$

Definition and Usage of Variables

Definitions

It is either an assignment to the variable or reading of a value for the variable.

Use

Use of identifier x means any occurrence of x as an operand.

Example: Consider the statement

$$x = y + z;$$

In this statement some value is assigned to x. It defines x and used y and z values.

Global Data-Flow-Analysis

Data Flow Analysis (DFA) is a technique for gathering information about the possible set of values calculated at various points in a program.

- An example of a data-flow analysis is reaching definitions.
- A single way to perform data-flow analysis of program is to setup data flow equations for each node of the control flow graph.

Use definition (U-d) chaining

The use of a value is any point where that variable or constant is used in the right hand side of an assignment or is evaluating an expression.

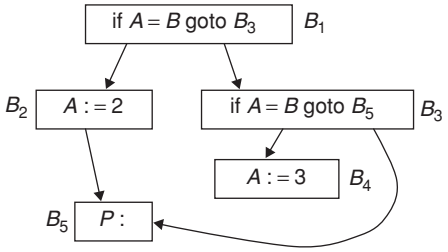
The definition of a value occurs implicitly at the beginning of the whole program for a variable.

A point is defined either prior to or immediately after a statement.

Reaching definitions

A definition of a variable A reaches a point P if there is a path in the flow graph from that definition to P, such that no other definitions of A appear on the path.

Example:



The definition $A: = 3$ can reach point p in B_5.

To determine the definitions that can reach a given program first assign distinct numbers to each definition, since it is associated with a unique quadruple.

- For each simple variable A, make a list of all definitions of A anywhere in the program.
- Compute two sets for each basic block B.

Gen [B] is the set of generated definitions within block B and that reach the end of the block.

1. Kill [B], which is the set of definitions outside of B that define identifiers that also have definitions within B.
2. IN [B], which are all definitions reaching the point just before B's first statement.

Once this is known, the definitions reaching any use of A within B are found by:

Let u be the statement being examined, which uses A.

1. If there are definitions of A within B before u, the last is the only one reaching u.
2. If there is no definition of A within B prior to u, those reaching u are in IN [B].

Data Flow Equations

1. For all blocks B,

$$\text{OUT}[B] = (\text{IN}[B] - \text{KILL}[B]) \cup \text{GEN}[B]$$

A definition d, reaches the end of B if

(a) $d \in$ IN [B] and is not killed by B.
(or)
(b) d is generated in B and is not subsequently redefined here.

2. IN [B] = U OUT [P]
$\forall P$ preceding B
A definition reaches the beginning of B iff it reaches the end of one of its predecessors.

Computing U-d Chains

If a use of variable 'a' is preceded in its block by a definition of 'a', this is the only one reaching it.

If no such definition precedes its use, all definitions of 'a' in IN [B] are on its chain.

Uses of U-d Chains

1. If the only definition of 'a' reaching this statement involves a constant, we can substitute that constant for 'a'.
2. If no definitions of 'a' reaches this point, a warning can be given.
3. If a definition reaches nowhere, it can be eliminated. This is part of dead code elimination.

EXERCISES

Practice Problems I

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. Replacing the expression 2 * 3.14 by 6.28 is
(A) Constant folding
(B) Induction variable
(C) Strength reduction
(D) Code reduction

2. The expression (a*b)*c op … where 'op' is one of '+', '*' and '↑' (exponentiation) can be evaluated on CPU with a single register without storing the value of (a*b) if
(A) 'op' is '+' or '*'
(B) 'op' is '↑' or '+'
(C) 'op' is '↑' or '*'
(D) not possible to evaluate without storing

3. Machine independent code optimization can be applied to
(A) Source code
(B) Intermediate representation
(C) Runtime output
(D) Object code

4. In block B if S occurs in B and there is no subsequent assignment to y within B, then the copy statement $S : x = y$ is
(A) Generated (B) Killed
(C) Blocked (D) Dead

5. If E was previously computed and the value of variable in E have not changed since previous computation, then an occurrence of an expression E is
(A) Copy propagation
(B) Common sub expression
(C) Dead code
(D) Constant folding

6. In block B, if x or y is assigned there and s is not in B, then $s : x = y$ is
(A) Generated (B) Killed
(C) Blocked (D) Dead

7. Given the following code
$A = x + y$;
$B = x + y$;
Then the corresponding optimized code as
_____
_____
$C = x + y$;
_____
$A = C$;
_____
$B = C$;
When will be optimized code pose a problem?
(A) When C is undefined.
(B) When memory is consideration.
(C) C may not remain same after some statements.
(D) Both (A) and (C).

8. Can the loop invariant $X = A - B$ from the following code be moved out?
For i = 1 to 10
```
{
A = B * C;
X = A - B;
}
```
(A) No
(B) Yes
(C) $X = A - B$ is not invariant
(D) Data insufficient

9. If every path from the initial node goes through a particular node, then that node is said to be a
(A) Header (B) Dominator
(C) Parent (D) Descendant

Common data for questions 10 and 11: Consider the following statements of a block:
$a: = b + c$
$b: = a - d$
$c: = b + c$
$d: = a - d$

10. The above basic block contains, the value of b in 3rd statement is
(A) Same as b in 1st statement
(B) Different from b in 1st statement
(C) 0
(D) 1

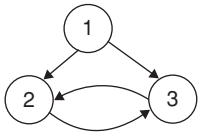
11. The above basic block contains
(A) Two common sub expression
(B) Only one common sub expression
(C) Dead code
(D) Temporary variable

12. Find the induction variable from the following code:
$A = -0.2$;
$B = A + 5.0$;
(A) A
(B) B
(C) Both A and B are induction variables
(D) No induction variables

13. The analysis that cannot be implemented by forward operating data flow equations mechanism is
(A) Interprocedural
(B) Procedural
(C) Live variable analysis
(D) Data

14. Which of the following consist of a definition, of a variable and all the uses, U, reachable from that definition without any other intervening definitions?
(A) Ud-chaining (B) Du-chaining
(C) Spanning (D) Searching

15. Consider the graph



The graph is
(A) Reducible graph
(B) Non-reducible graph
(C) Data insufficient
(D) None of these

Practice Problems 2

Directions for questions 1 to 15: Select the correct alternative from the given choices.

1. In labeling algorithm, let n is a binary node and its children have L_1 and L_2, if $L_1 = L_2$ then LABEL (n):
(A) $L_1 - 1$ (B) $L_2 + 1$
(C) $L_1 + L_1$ (D) $L_1 + 1$

2. The input for the code generator is a:
(A) Tree at lexical level
(B) Tree at semantic level
(C) Sequence of assembly language instructions
(D) Sequence of machine idioms

3. In labeling algorithm, let n is a binary node and its children have i_1 and i_2, LABEL (n) if $i_1 \neq i_2$ is
(A) Max (i_1, i_2)
(B) $i_2 + 1$
(C) $i_2 - 1$
(D) $i_2 - i_1$

4. The following tries to keep frequently used value in a fixed register throughout a loop is:
(A) Usage counts
(B) Global register allocation
(C) Conditional statement
(D) Pointer assignment

5. Substitute y for x for copy statement $s : x = y$ if the following condition is met
(A) Statements s may be the only definition of x reaching u
(B) x is dead
(C) y is dead
(D) x and y are aliases

6. Consider the following code

```
for (i=0; i<m; i++)
{
for (j=0; j<m; j ++)
If (i%2)
{
a = a + (14*j+5*i);
b = b + (9 + 4*j);
}
}
```

Which of the following is false?
(A) There is a scope of common reduction in this code
(B) There is a scope of strength reduction in this code.
(C) There is scope of dead code elimination in this code
(D) Both (A) and (C)

7. S_1: In dominance tree, the initial node is the root.
S_2: Each node d dominates only its ancestors in the tree.
S_3: if $d \neq n$ and d dom n then d dom m.
Which of the statements is/are true?
(A) S_1, S_2 are true
(B) S_1, S_2 and S_3 are true

(C) Only S_3 is true
(D) Only S_1 is true

8. The specific task storage manager performs:
(A) Allocation/Deallocation of storage to programs
(B) Protection of storage area allocated to a program from illegal access by other programs in the system
(C) The status of each program
(D) Both (A) and (B)

9. Concept which can be used to identify loops is:
(A) Dominators
(B) Reducible graphs
(C) Depth first ordering
(D) All of these

10. A point cannot be found:
(A) Between two adjacent statements
(B) Before the first statement
(C) After the last statement
(D) Between any two statements

11. In the statement, $x = y*10 + z$; which is/are defined?
(A) x (B) y
(C) z (D) Both (B) and (C)

12. Consider the following program:

```
void main ( )
{
    int x, y;
    x = 3; y = 7;
    --------
    --------
    if (x<y)
    {
    int x;
    {
    int y;
    y = 9;
    ------
    x = 2*y;
    }
    -------
    -------
    x = x + y;
    printf ("%d", x);
    }
    ------
    printf ("%d", x);
    }
```

The output is
(A) 3 – 25 (B) 25 – 3
(C) 3 – 3 (D) 25 – 25

13. The evaluation strategy which delays the evaluation of an expression until its value is needed and which avoids repeated evaluations is:
(A) Early evaluation (B) Late evaluation
(C) Lazy evaluation (D) Critical evaluation

14. If two or more expressions denote same memory address, then the expressions are:
(A) Aliases (B) Definitions
(C) Superiors (D) Inferiors

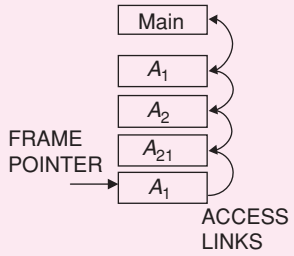
15. Operations that can be removed completely are called:
(A) Strength reduction
(B) Null sequences
(C) Constant folding
(D) None of these

Previous Years' Questions

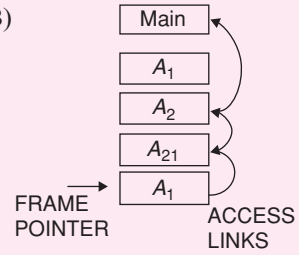
1. In a compiler, keywords of a language are recognized during: **[2011]**
(A) parsing of the program
(B) the code generation
(C) the lexical analysis of the program
(D) dataflow analysis

2. Consider the program given below, in a block structured pseudo-language with lexical scoping and nesting of procedures permitted. **[2012]**
Program main;
Var …
Procedure A_1;
Var …
Call A_2;
End A_1
Procedure A_2;
Var …
Procedure A_{21};
Var …
Call A_1;
End A_{21}
Call A_{21};
End A_2
Call A_1;
End main
Consider the calling chain: Main $\rightarrow A_1 \rightarrow A_2 \rightarrow A_{21} \rightarrow A_1$
The correct set of activation records along with their access links is given by:

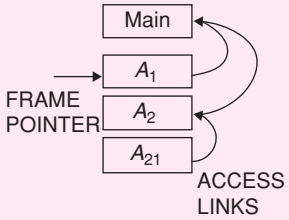
(A)



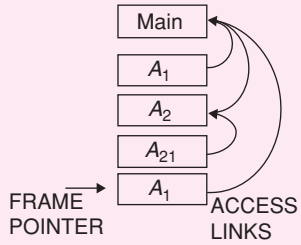
(B)



(C)



(D)



Common data for questions 3 and 4: The following code segment is executed on a processor which allows only register operands in its instructions. Each instruction can have atmost two source operands and one destination operand. Assume that all variables are dead after this code segment.

```
c = a + b;
d = c * a;
e = c + a;
x = c * c;
If (x > a) {
   y = a * a;
}
Else {
      d = d * d;
      e = e * e;
}
```

3. What is the minimum number of registers needed in the instruction set architecture of the processor to compile this code segment without any spill to memory? Do not apply any optimization other than optimizing register allocation. **[2013]**

(A) 3 (B) 4
(C) 5 (D) 6

4. Suppose the instruction set architecture of the processor has only two registers. The only allowed compiler optimization is code motion, which moves statements from one place to another while preserving correctness. What is the minimum number of spills to memory in the compiled code? **[2013]**

(A) 0 (B) 1
(C) 2 (D) 3

5. Which one of the following is NOT performed during compilation? **[2014]**

(A) Dynamic memory allocation
(B) Type checking
(C) Symbol table management
(D) Inline expansion

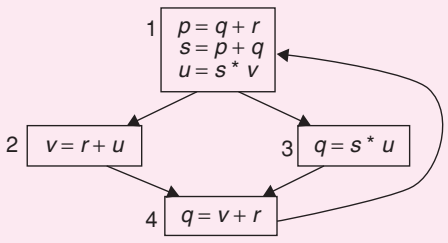
6. Which of the following statements are CORRECT? **[2014]**

(i) Static allocation of all data areas by a compiler makes it impossible to implement recursion.
(ii) Automatic garbage collection is essential to implement recursion.
(iii) Dynamic allocation of activation records is essential to implement recursion.
(iv) Both heap and stack are essential to implement recursion.

(A) (i) and (ii) only (B) (ii) and (iii) only
(C) (iii) and (iv) only (D) (i) and (iii) only

7. A variable x is said to be live at a statement S_i in a program if the following three conditions hold simultaneously: **[2015]**

1. There exists a statement S_j that uses x
2. There is a path from S_i to S_j in the flow graph corresponding to the program.
3. The path has no intervening assignment to x including at S_i and S_j.



The variables which are live both at the statement in basic block 2 and at the statement in basic block 3 of the above control flow graph are

(A) p, s, u (B) r, s, u
(C) r, u (D) q, v

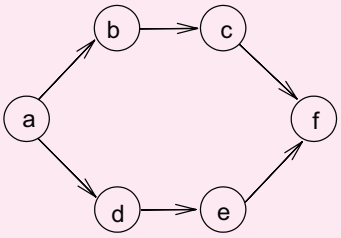
8. Match the following **[2015]**

P. Lexical analysis	1. Graph coloring
Q. Parsing	2. DFA minimization
R. Register allocation	3. Post-order traversal
S. Expression evaluation	4. Production tree

(A) P–2, Q–3, R–1, S–4 (B) P–2, Q–1, R–4, S–3

(C) P–2, Q–4, R–1, S–3 (D) P–2, Q–3, R–4, S–1

9. Consider the following directed graph:



The number of different topological orderings of the vertices of the graph is _____ . **[2016]**

10. Consider the following grammar:

stmt – > **if** expr **then** expr **else** expr; stmt | Ò
expr – > term **relop** term | term
term – > id | number
id – > **a** | **b** | **c**
number – > [**0** – **9**]

where **relop** is a relational operator (e.g., <, >,…), Ò refers to the empty statement, and **if, then, else** are terminals.

Consider a program *P* following the above grammar containing ten **if** terminals. The number of control flow paths in *P* is_____. For example, the program

if e_1 **then** e_2 **else** e_3

has 2 control flow paths, $e_1 \rightarrow e_2$ and $e_1 \rightarrow e_3$. **[2017]**

11. Consider the expression $(a—1) * (((b + c)/3) + d)$. Let X be the minimum number of registers required by an *optimal* code generation (without any register spill) algorithm for a load/store architecture, in which *(i) only load and store instruction can have memory operands and (ii) arithmetic instructions can have only register or immediate operands*. The value of X is _____. **[2017]**

12. Match the following according to input (from the left column) to the compiler phase (in the right column) that processes it: **[2017]**

(P) Syntax tree	(i) Code generator
(Q) Character stream	(ii) Syntax analyzer
(R) Intermediate representation	(iii) Semantic analyzer
(S) Token stream	(iv) Lexical analyzer

(A) P → (ii), Q → (iii), R → (iv), S → (i)

(B) P → (ii), Q → (i), R → (iii), S → (iv)

(C) P → (iii), Q → (iv), R → (i), S → (ii)

(B) P → (i), Q → (iv), R → (ii), S → (iii)

Answer Keys

Exercises

Practice Problems 1

1. A **2.** C **3.** B **4.** A **5.** B **6.** B **7.** C **8.** B **9.** B **10.** B
11. B **12.** D **13.** C **14.** B **15.** B

Practice Problems 2

1. D **2.** B **3.** A **4.** B **5.** A **6.** D **7.** D **8.** D **9.** D **10.** D
11. A **12.** B **13.** C **14.** A **15.** B

Previous Years' Questions

1. C **2.** D **3.** B **4.** B **5.** A **6.** D **7.** C **8.** C **9.** 6 **10.** 1024
11. 2 **12.** C