Tutorial No. 6

1) Explain following machine independent transformation technique:

→ a) Common sub expression and dead code elimination:

- Common sub expression elemination:
  - A common subexpression is 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 resource and time.
  - Thus is better be eliminated

Consider an example

a = a+b		a = b+c
b = a-d		b = a-d
c = b+c	⇒	c = b+c
d = a-d		d = b
a = a+c		a = a+c
c = a-d		c = a-d

- 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 remove 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 recorded as dead code.
- Consider the below dead code:

a = b+c		a = b+c
b = a-d		b = a-d
c = b+c	⇒	c = b+c
d = b		
a = a+c		a = a+c
c = a-b		c = a-b

b) Copy propagation and constant folding

- Copy propagation:
  - Copy propagation suggests to use one variable instead of other, in cases where assignment 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 propagation.
  - Consider the below copy propagation.

a = b+c		a = b+c
b = a-d		b = a-d
c = b+c	⇒	c = b+c
d = b		d = b
a = a+c		a = a+c
c = a-d		c = a-b

- 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 run time, but if we replace them with their values they need not be evaluated at runtime, saving time.
- e.g.

a = 4*2	⇒	a = 8
b = a+3		b = a+3

2) Explain how code motion and frequency reduction for loop optimization?

→ - Frequency reduction is a type in loop optimization proces which to machine independent.
- In frequency reduction code inside a loop is optimized to improve the running time of program.
- Frequency reduction is used to decrease the amount of code in a loop.
- A statement or expression, which can be removed outside the loop body without affecting the semantics of the program, is moved outside the loop.
- Frequency Reduction is also called code Motion.

* Objectives of frequency Reduction
- To reduce the evalution frequency of expression
- To bring loop invariant statements out of the loop.
- Below is the example of frequency reduction

① Initial code

```
while (i<100)
{
  a = sin(x) / cos(x) + i;
}
```

② Optimized code

```
t = sin(x) / cos(x)
while (i<100)
{
  a = t + i;
  i++;
}
```

3) Explain global data flow analysis using data flow equations

→ • To efficiently optimize the code compiler collects all the information about the program and distribute the information about to each block of 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 achieve 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 same optimization.

For example, consider the following code.

```
x = a+b;
x = 6*3
```

• In this code, the first assignment of x useless. The value computed 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, conside the following code:

```
a=1;  b=2;  ;  c=3:
if (condition)
  x = a+5;
else
  x = b+4;
c = x+1;
```

In this code, at line 1 the initial assignment is useless and x+1 expression can be simplified as 7. The data flow analysis can be performed on the program's control flow graph. The control flow graph of a program 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.

4) Explain different code generation issue with example?

→ a) The target machine
- A byte addressable machine with four bytes to a word and n general purpose registers.
- Two address instructions
  - OP source, destination.
- Six addressing modes

• absolute	M	M	1
• registers	R	R	0
• indexed	c(R)	c+content(R)	1
• Indirect register	*R	contents(R)	0
• Indirect indexed	*c(R)	contents(c+contents(R))	1
• Literal	#c	N/A	1

b) Instruction selection
- Instruction selection is important to obtain efficient code.
- Choose appropriate target machine instructions to implement the IR statements
- Suppose we translate 3AC x:=y+z to

```
MOV y, R0
ADD z, R0      a: a+1
MOV R0, z
```

```
MOV a, R0
ADD #1, R0
MOV R0, a
cost = 6
```

Better ⇓
```
ADD #1, a
cost=3
```

Best ⇓
```
INC, a
cost=2
```

Instruction selection: Utilizing Addressing Modes
- Suppose we translate a=b+c into
```
MOV b, R0
ADD c, R0.
MOV R0, a
```
- Assuming addresses a, b and c are stored in R0, R1 and R2.
```
MOV *R1, *R0
ADD *R2, *R0
```
- Assuming R1 and R2 contain values of b & c.
```
ADD R2, R1
MOV R1, a.
```

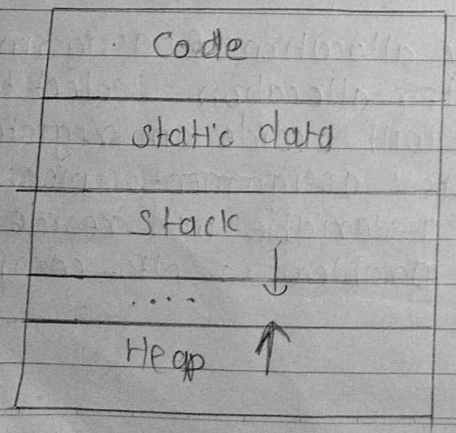
c) Register allocation & assignment:
- Register allocation: select the set of variables that will reside on registers.
- Register assignment: pick the specific register that a variable will reside in
- The problem is NP-complete.

Example:

```
t = a*b                t = a*b
t = t+a                t = t+a
t = t/d                t = t/d
  ↓ {R1=t}               ↓ {R0=a, R1=t}
MOV a, R1              MOV a, R0
MUL b, R1              MOV R0, R1
ADD a, R1              MUL b, R1
DIV d, R1              ADD R0, R1
MOV R1, t              DIV d, R1
                       MOV R1, t
```

d) Distinct Regions of Memory

- Code space - Instructions to be executed
  - Best if read only
- Static (or Global) - Variables that retain their value over the lifetime of the program
- Stack - Variables that is only as long as the block within which they are defined (local)
- Heap - Variables that are defined by calls to the system storage allocator.



5) Explain labeling algorithm with example.

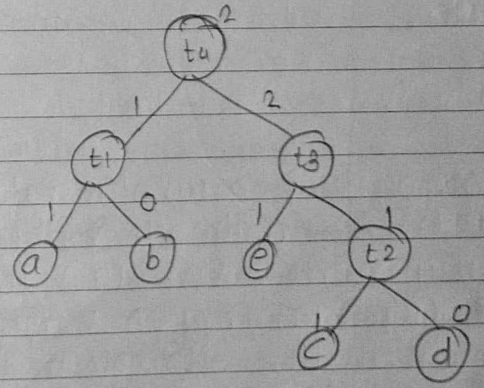
→ # Labeling algorithm

```
if n is a leaf then
    if n is the leftmost child of its parent then
        label(n) := 1
    else
        label(n) := 0
else begin
    let n1, n2, ---, nk be the children.
    labels so that label(n1) ≥ label(n2) ≥ ---
                ≥ label(nk);
    label(n) := max 1≤i≤k (label(ni) + i - 1).
```

Example:

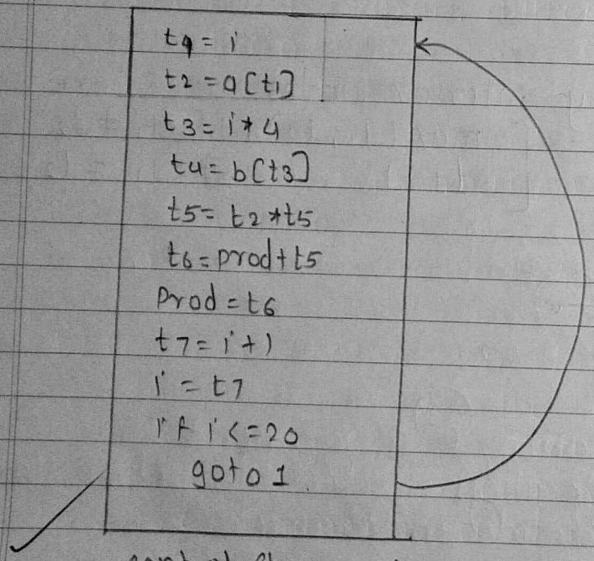
For binary interior nodes:

$$label(n) = \begin{cases} \max(l_1, l_2), & \text{if } l_1 \neq l_2 \\ l_1 + 1, & \text{if } l_1 = l_2 \end{cases}$$

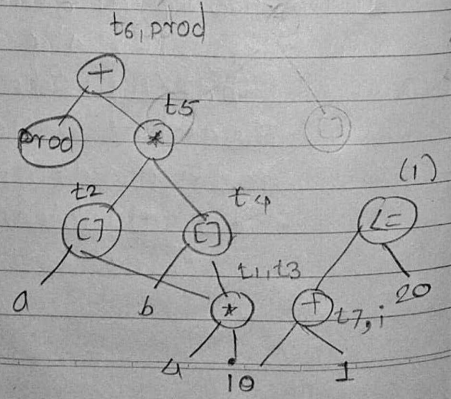


6) For the following 3A statements shown below, construct flow graph and DAG:

1) $t_1 = 4 * i$
2) $t_2 = a[t_1]$
3) $t_3 = 4 * i$
4) $t_4 = a[t_3]$
5) $t_5 = t_2 * t_4$
6) $t_6 = prod + t_5$
7) $prod = t_6$
8) $t_7 = i + 1$
9) $i = t_7$
10) IF $i <= 20$ goto (1)



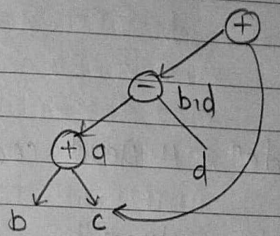
control flow graph



DAG

7) Draw the DAG for the following

$d = b * c$ $e = a + b$ $b = b * c$ $a = e - d$.



3AC:

1) $t_1 = b + c$
2) $a = t_1$
3) $t_2 = a - d$
4) $b = t_2$
5) $t_3 = b + c$
6) $c = t_3$
7) $t_4 = a - d$
8) $d = t_4$

8) Explain structure preserving transformation techniques with example.

→ The structure preserving transformation of basic block includes:

a) Dead code elimination
b) common subexpression elimination
c) Renaming of Temporary variables
d) Interchange of two independent adjacent statements.

a) Dead code elimination:

Dead code elimination is defined as that part of the code that never executes during the program execution so, for optimization, such code or dead code is eliminated. The code which is never executed during the program (Dead code) takes time so, for optimization and speed, it is eliminated from the code. Eliminating the dead code increases the speed of the program as the compiler does not have to translate the dead code.

Example:

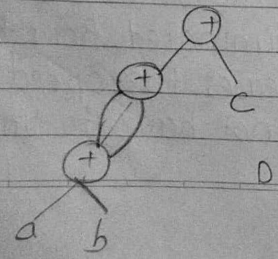
```
// Program with Dead code
int main()
{
    x=2
    if(x>2)
        cout << "code"; // Dead code
    else
        cout << "optimization";
    return 0;
}

// Optimized program without dead code
int main()
{
    x=2;
    cout << "optimization" ; // Dead code eliminated
    return 0;
}
```

b) Common Subexpression Elimination:

In this technique, the sub expression which are common are used frequently are calculated only once and reused when needed. DAG (Directed Acyclic Graph) is used to eliminate common subexpressions.

Example:



6 Nodes
6 Edges

DAG representation

c) Renaming of Temporary Variables:

Statements containing instances of a temporary variable can be changed to instances of new temporary variable without changing the basic block value.

Example:-

Statement $t = a+b$ can be changed to $x = a+b$ where t is a temporary variable and x is a new temporary variable without changing the value of the basic block.

d) Interchange of Two Independent Adjacent Statements:

If a block has two adjacent statements which are independent can be interchanged without affecting the basic block value.

Example:

$t_1 = a+b$

$t_2 = c+d$

These two independent statements of a block can be interchanged without affecting the value of the block.

9) Explain register allocation and assignment with the suitable example.

→ Register allocation and assignment

- Efficient utilization of the limited set of registers is important to generate good code
- Registers are assigned by
  - Register allocation to select the set of variables that will reside in registers at a point in the code.
  - Register assignment to pick the specific register that a variable will be reside in
- Finding an optimal register assignment in general is NP-complete.

Example:

```
t := a*b
t = t+a
t = t/d
↓{R1 = t}
MOV a, R1
MUL b, R1
ADD a, R1
DID d, R1
MOV R1, t
```

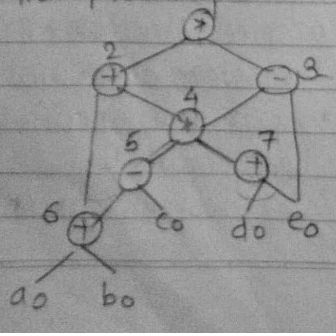
```
t := a*b
t = a+b
t := t/d
↓ {R0 = a, R1 = t}
MOV a, R0
MOV R0, R1
MOV b, R1
ADD R0, R1
DIV d, R1
MOV R1, t
```

10) Explain node listing and labeling algorithm with example.

→ • Node listing algorithm

```
while unlisted interior nodes remain do begin
    select an unlisted node n, all of those parents
    have been listed;
    list n;
    while the leftmost child m of n has no unlisted
    parents and is not a leaf do begin
        list m;
        n := m;
    end
end
```

Example:




```
t7 = d+e
t6 = a+b
t5 = t6-c
t4 = t5*t7
t3 = t4-e
t2 = t6+t4
t1 = t2*t3
```

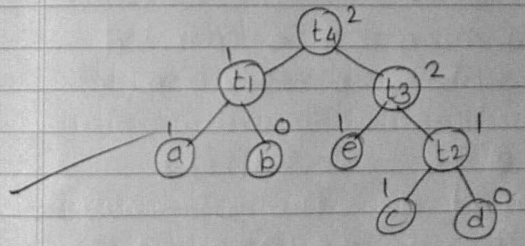
• The labeling algorithm:

```
if n is a leaf then
    if n is the leftmost child of its parent then
        label(n) := 1
    else
        laben(n) := 0
else begin
    let n1, n2, ... nk be the children of n ordered
    by label so that label(n1) ≥ label(n2) ≥ ... ≥ label(nk);
    label(n) := max 1≤i≤k (label(ni) + i - 1).
end
```

Example:

for binary interior nodes

$$label(n) = \begin{cases} \max(l_1, l_2) & , \text{if } l_1 \neq l_2 \\ l_1 + 1 & , \text{if } l_1 = l_2 \end{cases}$$

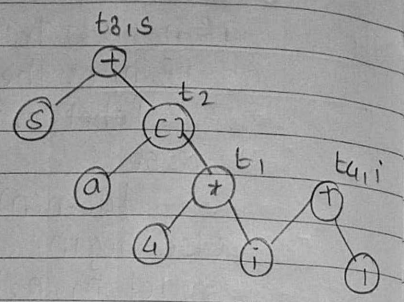


11) Construct DAG and 3AC statements for following C program:

```
i = 1;
s = 0;
while (i <= 10)
{
  s = s + a[i][b1]
  i = i + 1;
}
```

Three address code:

1) i=0
2) s=0
3) t1=i*4
4) t2=a[t1]
5) t3=s+t2
6) s=t3
7) t4=i+1
8) i=t4
9) if (i<10) goto (L1)



12) Generate DAG representation of following code and list out the application of DAG.

```
i=1;
while (i<=10)
DO
 sum1=a[i];
```

→ 1) i=1
2) i<=10 goto
3) goto
4) t1=i*4
5) t2=a[t1]
6) sum1=a[i]

- Application of DAG

* Easy to determine:
- common subexpression
- names used in the block but evaluated outside the block
- names whose values could be used outside the block.
- Leaves labeled by unique identifiers
- Interior nodes labeled by operator symbol
- Interior node optionally given a sequence of identifies, having the value represented by the nodes

14) Discuss in detail about global data flow analysis

→ Global data flow analysis:

To efficient 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.

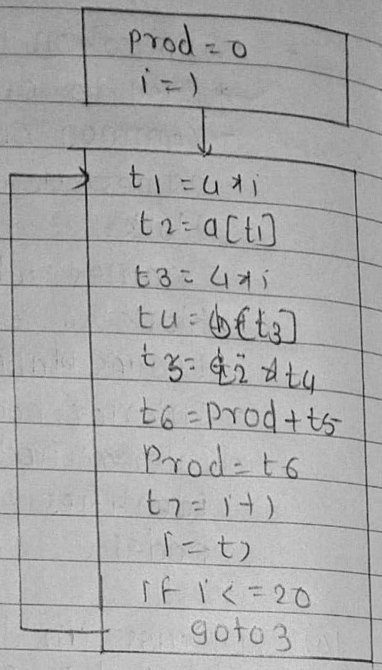
- A flow graph is directed graph
- The nodes in the graph are basic blocks
- There is an edge from B_1 to B_2 iff B_2 immediately follows B_1 in some execution sequence
  - there is jump from B_1 to B_2
  - B_2 immediately follows B_1 in a program text
- B_1 is a predecessor of B_2, B_2 is a successor of B_1.

Name :
Date : / /

Example:

3AC

1) prod := 0
2) i = 1
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 <= 20 goto 3



Data flow information can be collected by setting up and solving systems of equations of the form:

Out[S] = gen[S] ∪ (in[S] - kill[S])

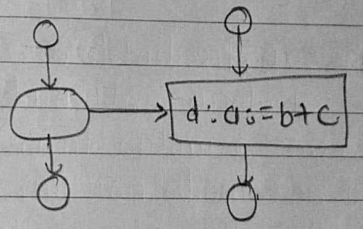
This eqn can read as "the information at the end of statement, or enters at the beginning and is not killed as control flow through the statement." such equations are called data-flow equation.

- The details of how data flow equations are set and solved depend on three factors. The notions of generating and killing depend on the desired information. for some problems, instead of proceeding along with flow of control & defining out[S] in terms of in[S], we need to proceed backwards and define in[S] in terms of out[S].

- Since data flows along control paths, data flow analysis is affected by the constructs in a program. In fact, when we write out[S] we implicitly assume that there is unique end point where control leaves the statement; in general, eqn's are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
- There are subtleties that go along with such statements as procedure calls, assignments thr pointer variables, and even assignments to array variables.

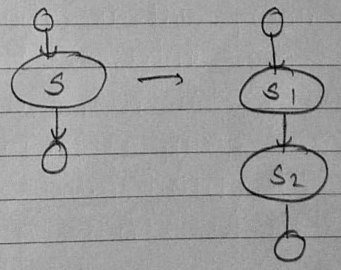
Example :

i)



gen[S] = {d}
kill[S] = Da - d
out[S] = gen[S] ∪ (in[S] - kill[S])

ii)



in[S1] = in[S]
in[S2] = out[S1]
out[S2] = out[S2]

gen[S] = gen[S2] (gen[S1] - kill[S2])
kill[S] = kill[S2] ∪ (kill[S1] - gen[S2]).

13) Explain principles sources of optimization

→ 1) Common-subexpression elimination
2) Copy propagation
3) Dead code elimination
4) Constant folding.