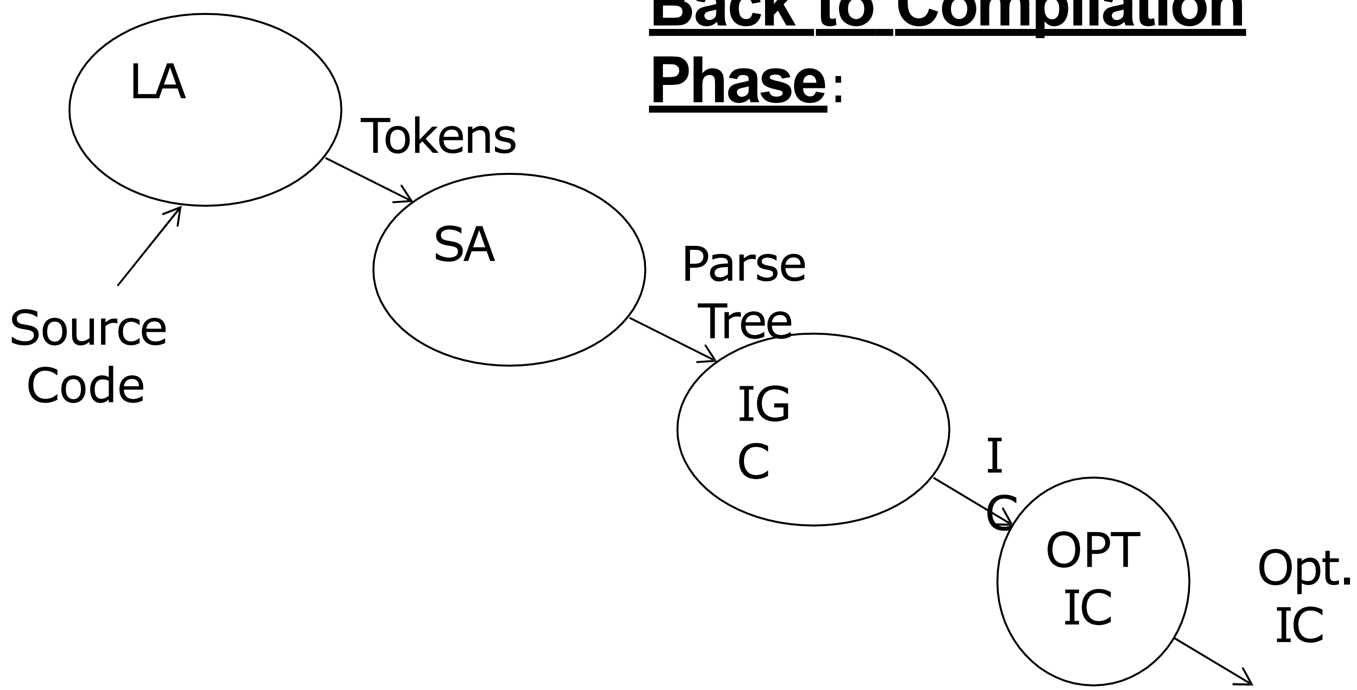
Chapter 6. Code Optimization

1. Introduction
2. Machine Independent Optimization
    1. Basic optimizations
    2. loop Optimization
    3. Removing Redundant 3ACs
3. Machine Dependent Optimization
4. Data Flow Analysis

Mr. Param Venkat Vivek Kesireddy
Teaching Associate
Mail: pkesireddy@fullerton.edu

Back to Compilation Phase:



6.1 Introduction

Optimization is an attempt to simplify and to remove as many unnecessary and redundant 3AC/Assembly instructions

two types of optimization

- machine independent (optimization on IC)
- machine dependent (optimization on assembly)

E.g. machine independent Opt.

t = a +b
x = t

=> x = a+b (Simplify)

t1 = a*b
c = t1 + 1
t2 = a*b
d = t2 + 2

(t2 = a * b is not necessary)

=> d = t1 + 2

6.2 Machine independent optimizations

6.2.1 Basic Optimizations

a) constant folding: Pre-computation of the results at the compilation time

e.g.,

x = 2*3 => x = 6

b) constant propagation: replace all constant values during compilation time

e.g.,

tax_rate = 8.25 /* constant and tax_rate does not change */

=> change all tax_rate to 8.25 during compilation time

c) Reduction in strength: replace an expensive operator with a "cheaper" operator

e.g.,

$a = x^2$

=> $= x * x$

why?

$a = x^2$

=> in 3 AC

t1 = ln (x)

t2 = 2 * t1

a = exp (t2)

Expensive !

d) dead code elimination: eliminates code that is never executed or does nothing useful

e.g.,

```
if ( 5 < 3) .....                    /* never can be executed */
```

or

```
X =5;  /* assigned but never used */
…
X =7;  /* reassignment without the using previous assignment */
```

6.2.2. Loop Optimization

a) Loop invariant expressions:
take out the invariant expressions from the loop

e.g.,

```
for k = 1 to 100 do
    c[k] = 2 * (p-q*x) * (n-k+1)
```

=>

```
t1 = p-q*x
c[k] = t1 * (n-k+1)
```

b) loop unrolling: unroll nested loops to simple loops

e.g.,

```
for i = 1 to
    200 do  for j = 1
        to 2 do  write
            (x[i,j])
```

=>

```
for i = 1 to 200 do
write (x[i, 1], x[i,
    2])
```


which saves loop

6.2.3 Removing redundant 3 ACs

E.g.,
Source: x = (a+b) * (a+b)

3AC:
t1 = a+b
t2 = a+b
t3 = t1 * t2
x = t2
=>
t1 = a+b
x = t1 * t1
(from 4 instructions to 2 instructions)

How?

<u>3 steps</u>

Step 1:Generate modified DAG (Directed Acyclic Graph) of the

3AC Step 2: Sort the nodes in DAG Topologically

Step 3: Traverse the soft list backwards and generate 3 ACs

e.g.,

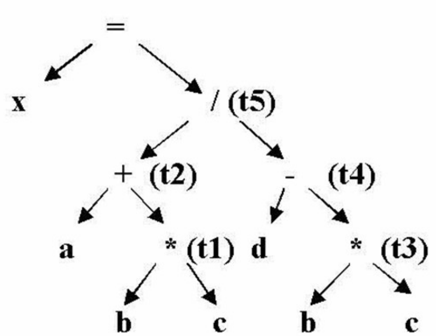
source: x = (a+ b*c) / (d - b*c)

3ACs:

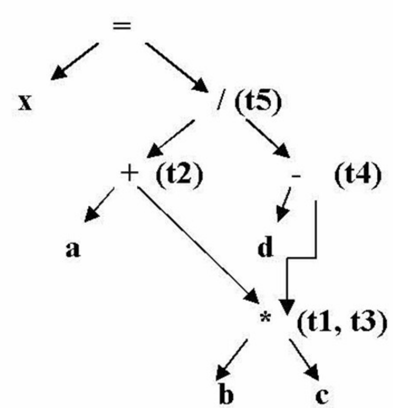
1. t1 = b*c
2. t2 = a + t1
3. t3 = b*c (not necessary)
4. t4 = d -t3
5. t5 = t2 / t4 (not necessary)
6. x = t5 (x = t4/t2)

Step 1: Create modified DAG of the 3 ACs
Directed Acyclic Graph: AST but one sub-tree per expression

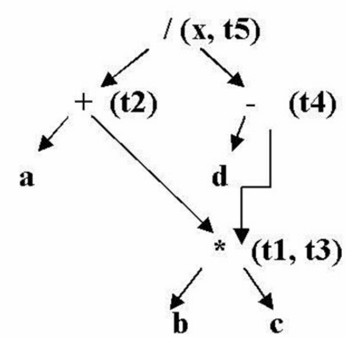
a) AST



b) DAG
Eliminate same expressions



c) modified DAG

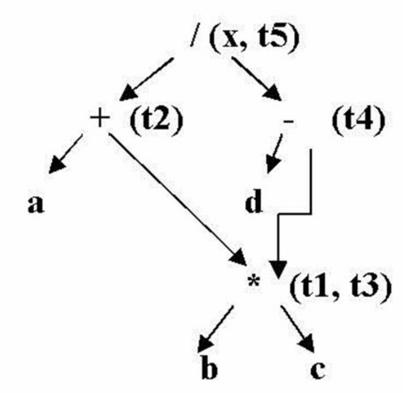


Step 2: Topological Sort

Set n =1
repeat

- select any source favoring left child & assign the number n
-

=>
n = 1 x, t5
n = 2 t2
n = 3 a
n = 4 t4
n = 5 d
n = 6 t1, t3
n = 7 b
n = 8 c



Step 3: Traverse the nodes (n) backwards and generate 3 ACs

```
1. t1 = b * c
2. t4 = d - t1
3. t2 = a + t1
4. x =  t2 / t4
```

Result: two instructions are eliminated

6.3 Machine Dependent Optimization

Peep-hole optimization

Look at the generated assembly code thru a small moving window (peep-hole) over few instructions at a time and do small scale optimization within that window

1. Redundant load/store elimination

a = b+c **a = b+c & d = a+e**

LOAD R0 = b
ADD R0 = c
STORE a, R0

2. Flow of control optimization: Avoid jumps on jumps

2. Jump-Over-Jump

Source:
repeat
...
until i = limit

```
loop-start
......

L       Rx, i
CMP     Rx, limit
je      loop-exit
jump    loop-start

loop-exit
```

=>

```
loop-start
......

L       Rx, i
CMP     Rx, limit
jne     loop-start

loop-exit
```

6.4 6.4 Data Flow Analysis (DFA)

We know how to optimize within a "basic block". (Ex. Removing a redundant 3Acs)
If we want optimize between blocks, we need to do DFA.

- Basic Block: A basic block is a sequence of 3 AC instructions that are executed sequentially, from the beginning till the end, i.e., no Branches

Ex. in terms of source code

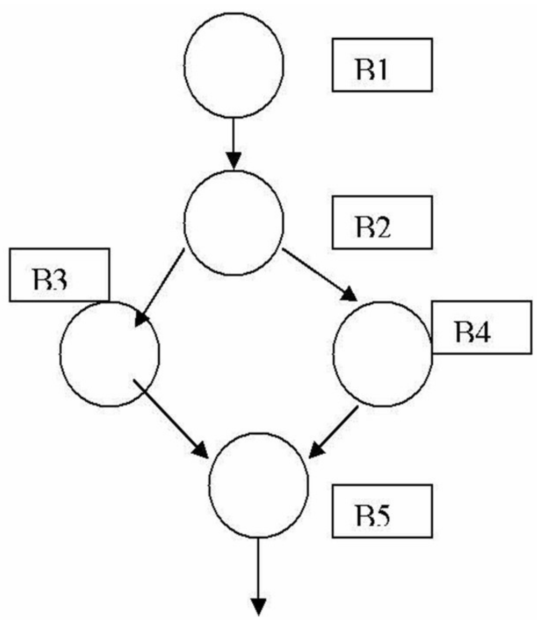
X1 B1
X2
If (a > b) then
B2
Begin B3
A1
A2
End
Else
b3

B4
Y1
B5

2. Definitions

We can express the blocks in terms of Flow Graph (FG)

FG is a directed graph in which each node is a basic block and the edges show the transfer off controls.



Predecessor of a node Bj: Bi is predecessor if Bj, if there is an edge from I to j

Also, then Bj is a successor of Bi.

Ex. B2 is a successor of B1 and B2 is a predecessor of B3 /B4

3. **<u>Forms of DFA</u>**

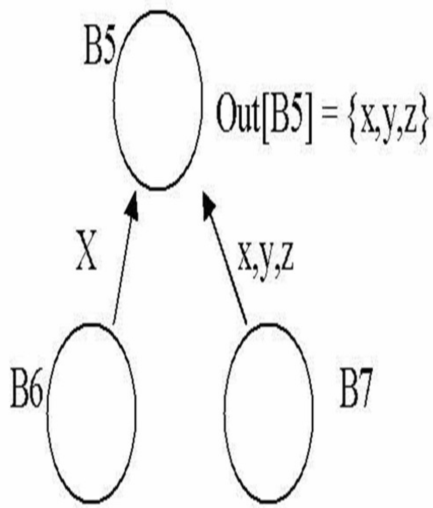
Forward Analysis : Examines Blocks from the First to the Last
Backward Analysis: Examines Block from the Last to First

In Each Flow Analysis: All Path and Any Path Analysis

⇒Total 4 different: Forward (Any and All Paths)
and Backward (Any and All Paths) Analysis.

•Each Analysis gives 3 unique equations

Example: Backward and Any Path Analysis



EQ1: OUT [Bj] = U IN [Bi]

Bi succ Bj

Ex.. IN[B6] = {x} IN[B7] = {x,y,z} => OUT [B5] = {x,y,z}

EQ2: IN[Bj] = USED[Bj] U (OUT[Bj] – Kill [Bj])

Kill: Expression is killed if operands is changed (ex. New assignment)

EQ3: OUT[Bn] = { }

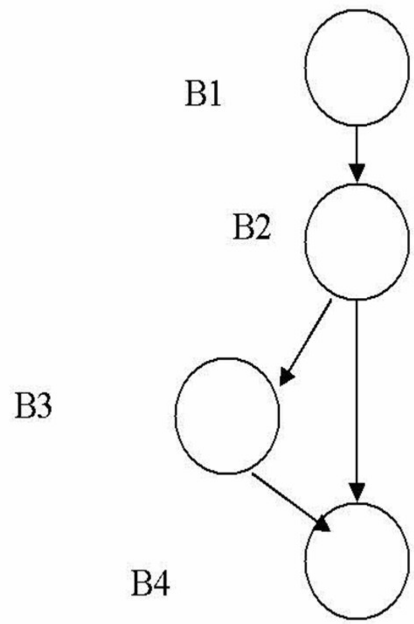
What is wrong with this code?

```
Procedure DFA
 Int a, b;

{
    get(a);
    if (a = 3)
        b = a;
    put (a, b);
    }
```

Use of the Backward-Any Path Analysis
(Find Initialized Variable)

Procedure DFA
Int a, b;

```
{
    get(a);          B1
    if (a = 3)
        b = a;       B2
    put (a, b);
}                    B3

                     B4
```



OUT[B4] = { }
IN [B4] = {a,b}

OUT[B3] = {a,b}
IN [B3] = {a} U ({a,b} – {b}) = {a}

OUT[B2] = {a} U {a,b} = {a,b}
IN [B2] = {a} U ({a,b} – { }) = {a,b}

OUT[B1] = {a,b}
IN [B1] = { } U ({a,b} – {a}) = {b}

⇨ IN[B1] = {b}
⇨ B is NOT defined prior to B1
⇨ Error

EQ1: OUT [Bj] = U IN [Bi]
Bi succ Bj

EQ2: IN[Bj] = USED[Bj] U (OUT[Bj] – Kill [Bj])
Kill: Expression is killed if operands is changed (ex. New assignment)

EQ3: OUT[Bn] = { }

E N D