CFG (Control flow graph)

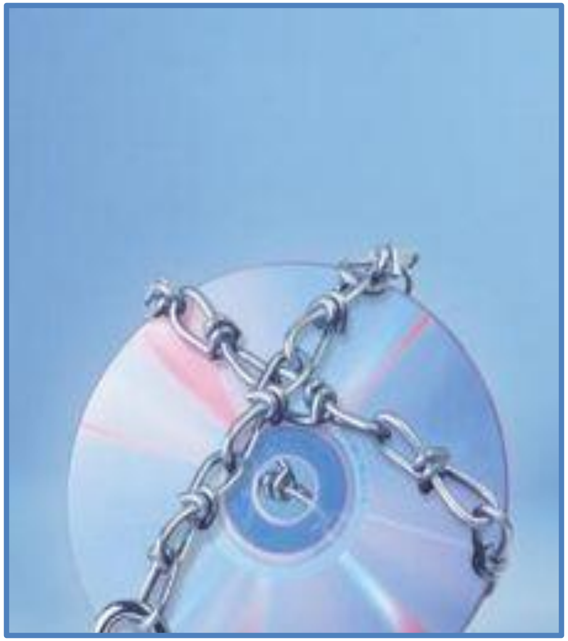
Class B T12

오지은 **200814189** 신승우 **201011340** 이종선 **200811448**

Contents

Contents

Introduction to CFG (1/17)



CFGs help ensure that software works correctly.

✓ Control flow graphs are one technique to ensure that computer programs work correctly.

✓ Computer code is complicated and is worked and reworked before it is ready for final release.

✓ There may be code that will never be executed and there may be code that will lead to loops from which you can never exit.

✓ Control flow graphs are one way of finding this bad code.

✓ First the code is broken up into control blocks, then graphs — trees.

✓ They are constructed to ensure that every block is reachable and that no block loops endlessly.

Introduction to CFG (2/17)

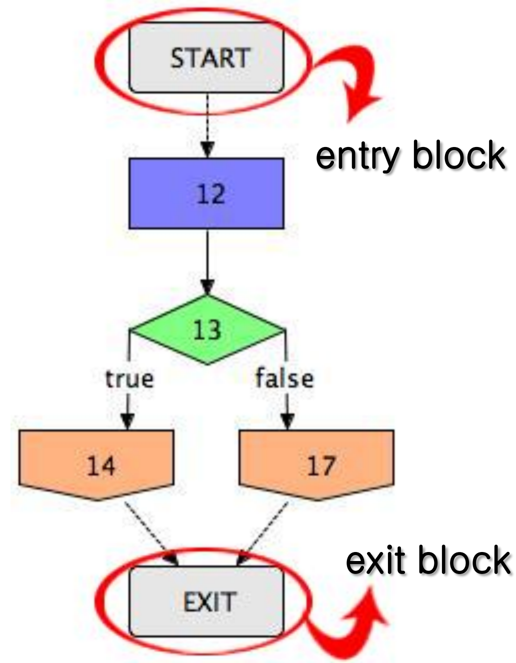
✓ In a control flow graph each node in the graph represents a basic block, i.e. a straight-line piece of code without any jumps or jump targets.

✓ Jump targets start a block, and jumps end a block.

✓ Directed edges are used to represent jumps in the c ontrol flow.

✓ There are, in most presentations, two specially designated blocks:
the **entry block**, through which control enters into the flow graph, and the **exit block**, through which all control flow leaves.

✓ The CFG is essential to many compiler optimizations and static analysis tools.



Introduction to CFG (3/17)

Consider the following fragment of code:

```
0: (A) t0 = read_num
1: (A) if t0 mod 2 == 0
2: (B) print t0 + " is odd."
3: (B) goto 5
4: (C) print t0 + " is even."
5: (D) end program
```

In the above, we have 4 basic blocks: A from 0 to 1, B from 2 to 3, C at 4 and D at 5. In particular, in this case, A is the "entry block", D the "exit block" and lines 4 and 5 are jump targets.
A graph for this fragment has edges from A to B, A to C, B to D and C to D.

Introduction to CFG (4/17)

Basic Blocks (BB)

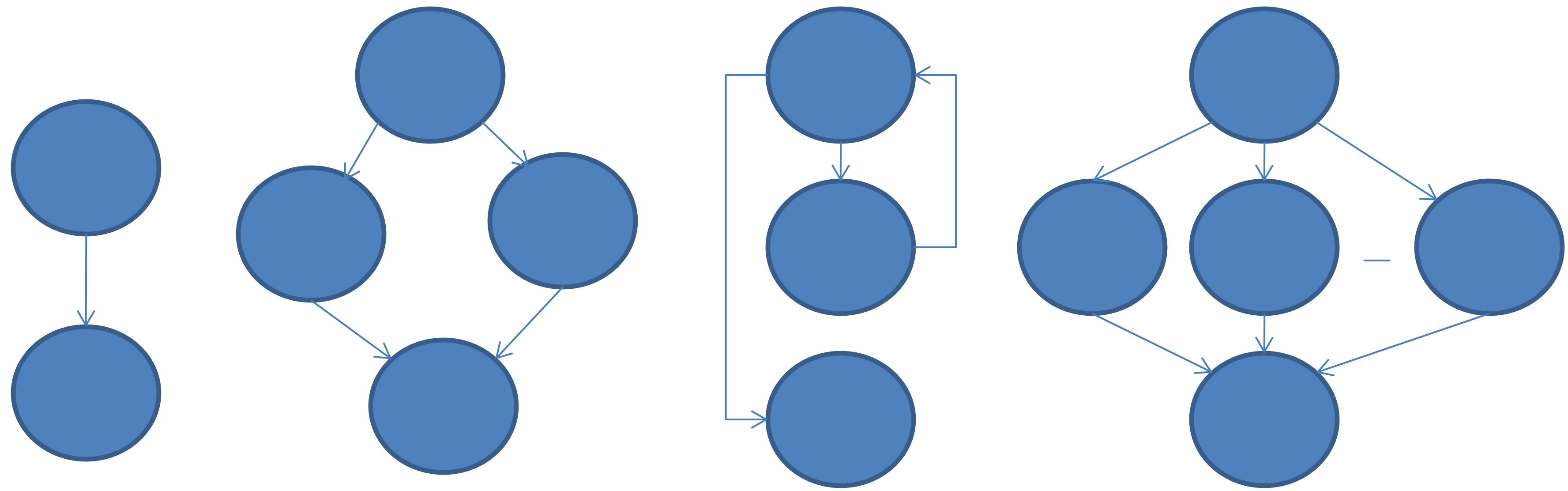
Compilers usually decompose programs into their basic blocks as a first step in the analysis process.
Basic blocks form the vertices or nodes in a control flow graph.

- ✓ Meaning
  A group of instructions applied with same performing condition.

- ✓ Definition
  1. one entry point, meaning no code within it is the destination of a jump instruction anywhere in the program.
  2. one exit point, meaning only the last instruction can cause the program to begin executing code in a different basic block.

Introduction to CFG (5/17)

A sequence of instructions forms a basic block

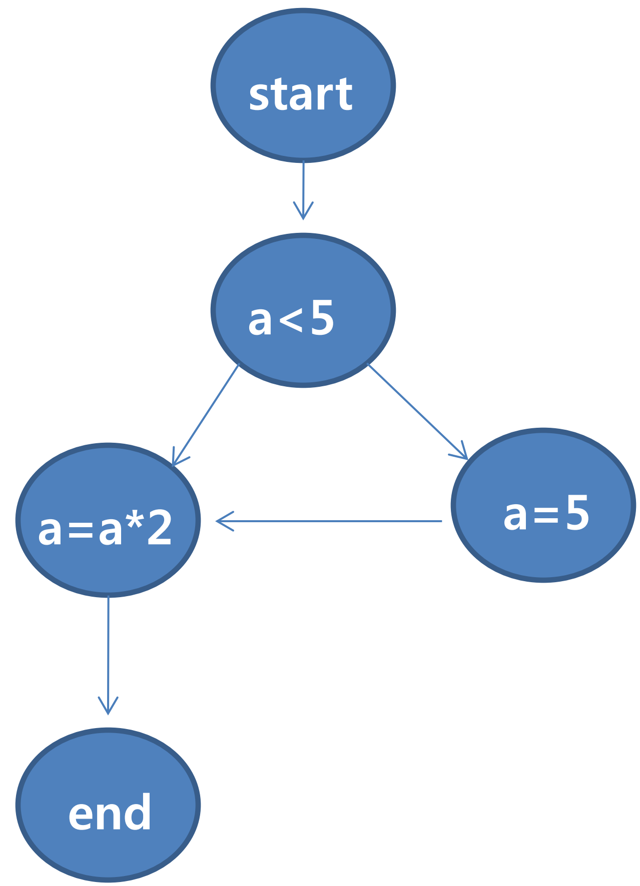


Sequence **IF** **while** **case**

Introduction to CFG (6/17)

If statement

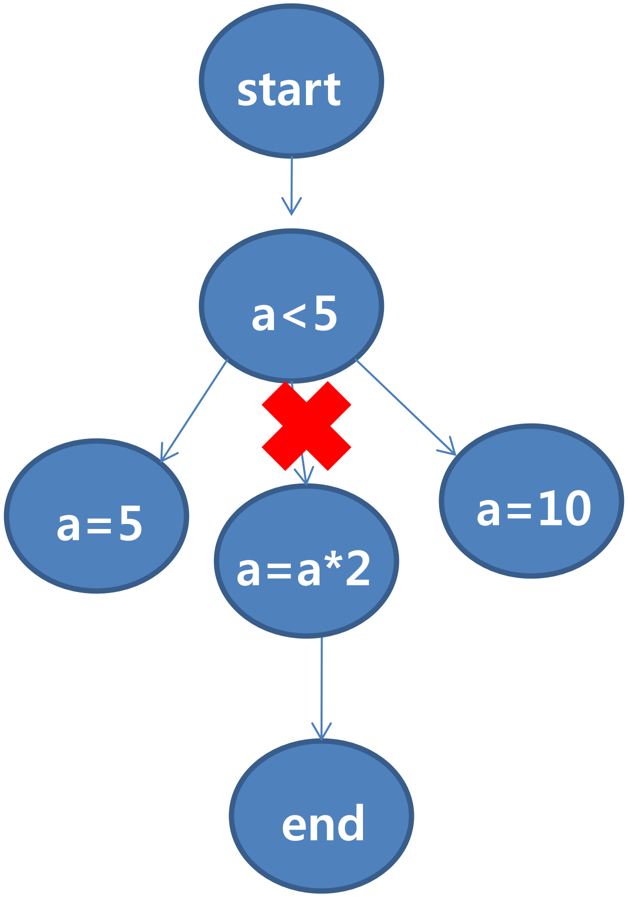
```
void if TestMethod (int a)
{
      if (a < 5)
      {
            a = 5;
      }
       a= a*2;
}
```



Introduction to CFG (7/17)

If else statement

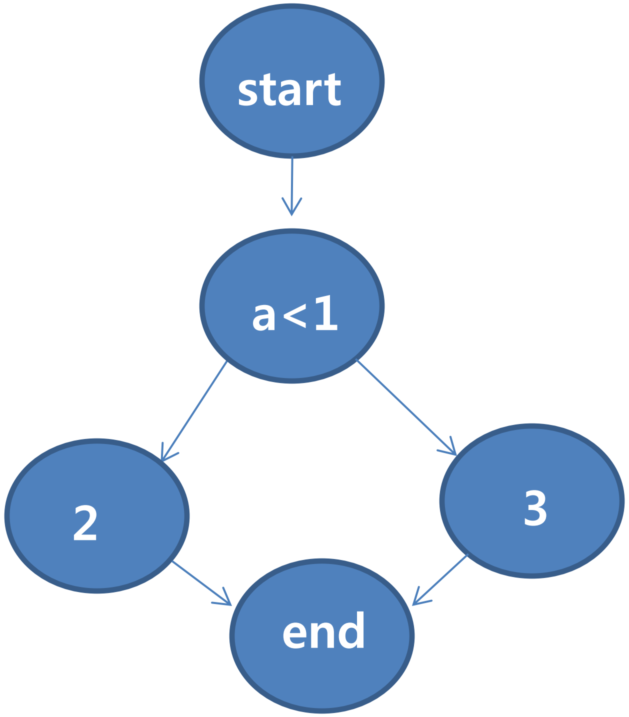
```
void if TestMethod (int a)
{
      if (a < 5)
      {
            a = 5;
      }
       a= a*2;
}
```



Introduction to CFG (8/17)

Ternary operator

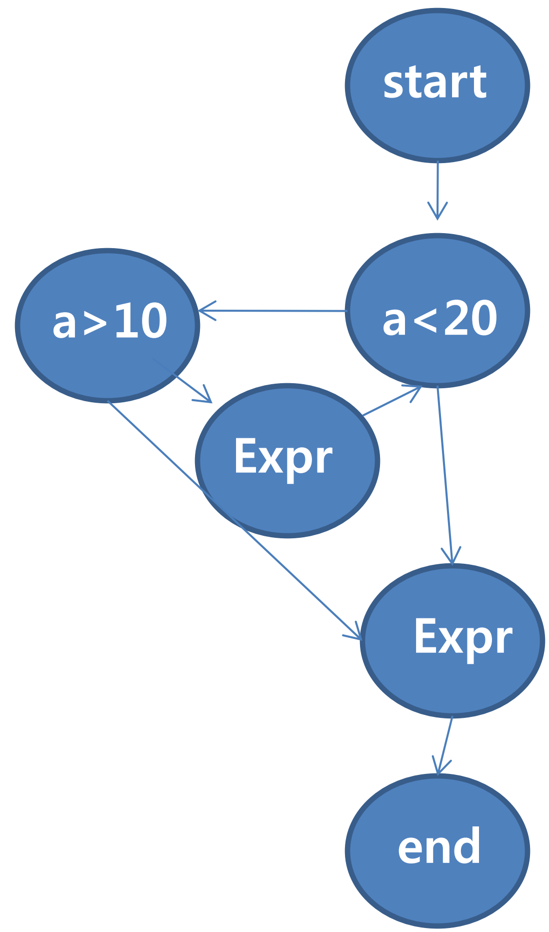
```
void ternaryTestMethod(int b, int a)
{
	b = (a<1) ? 2 : 3;
}
```



Introduction to CFG (9/17)

For statement

```
void forTestMethod(int a)
{
   for (a = 1; a < 20; )
   {
      if (a > 10)
      {
         break;
      }
      else
      {
         a = a + 2;
      }
   }
   a= a*2;
}
```

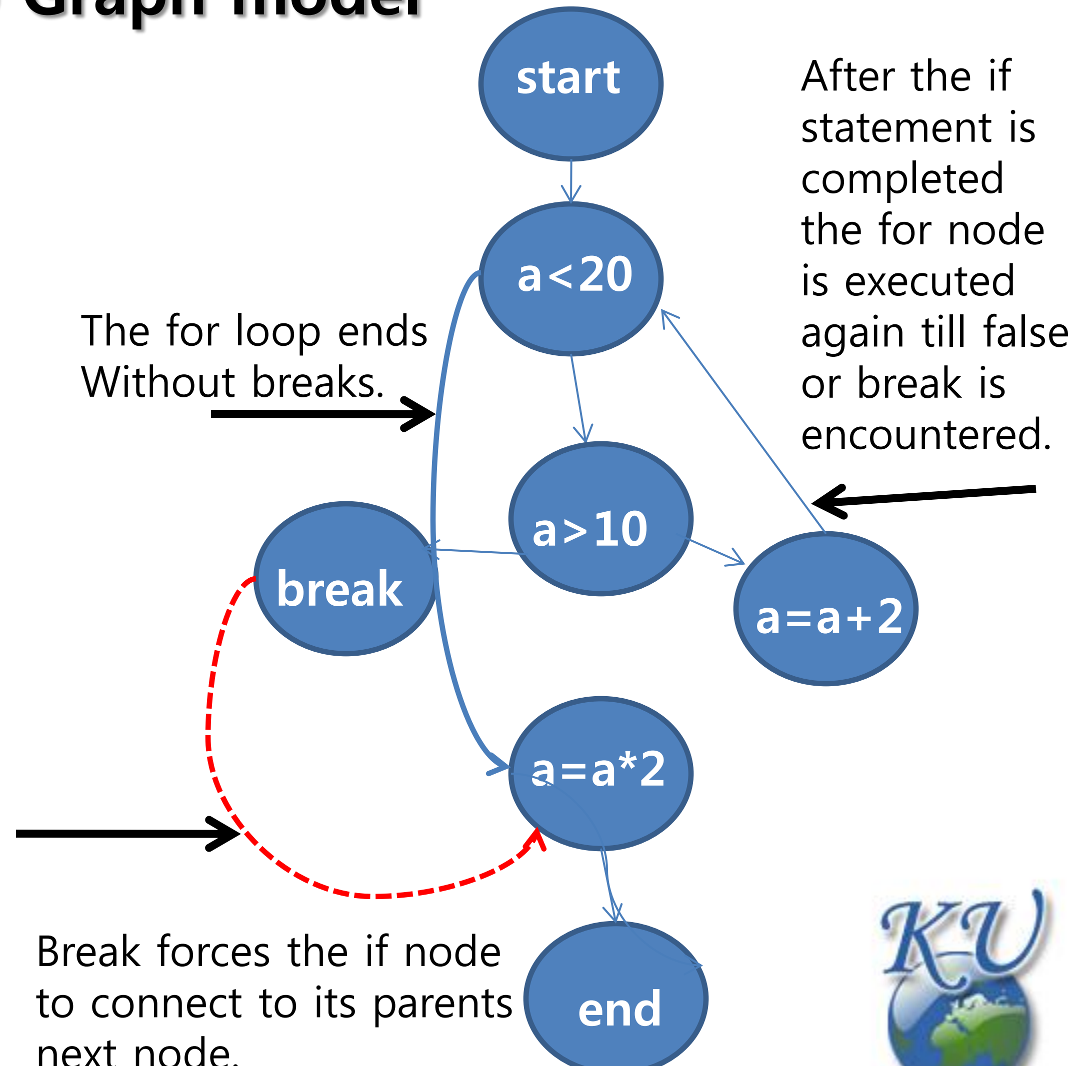


Expr which is an abbreviation of very Long Expression Statements.

Introduction to CFG (10/17)

For statement - Tree to Graph model

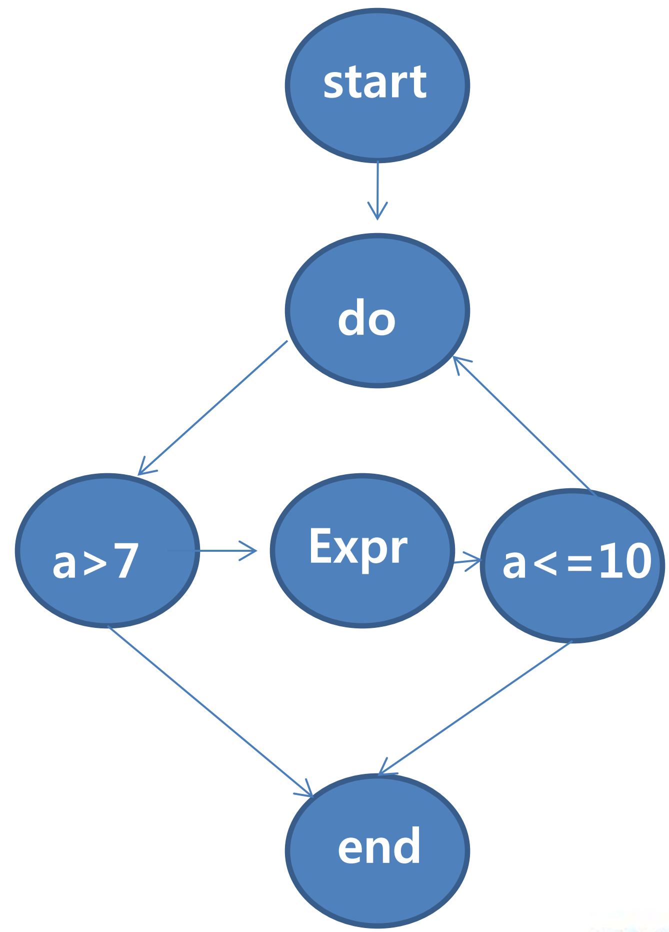
```
void forTestMethod(int a)
{
   for (a = 1; a < 20; )
  {
     if (a > 10)
     {
         break;
      }
     else
     {
        a = a + 2;
      }
   }
   a= a*2;
}
```



Introduction to CFG (11/17)

Do-while statement

```
Void doWhileTestMethod(int a)
{
 do
 {
   if (a > 7)
   {
     break;
   }
   else
   {
     a = a + 2;
   }
 } while (a <= 10);
}
```

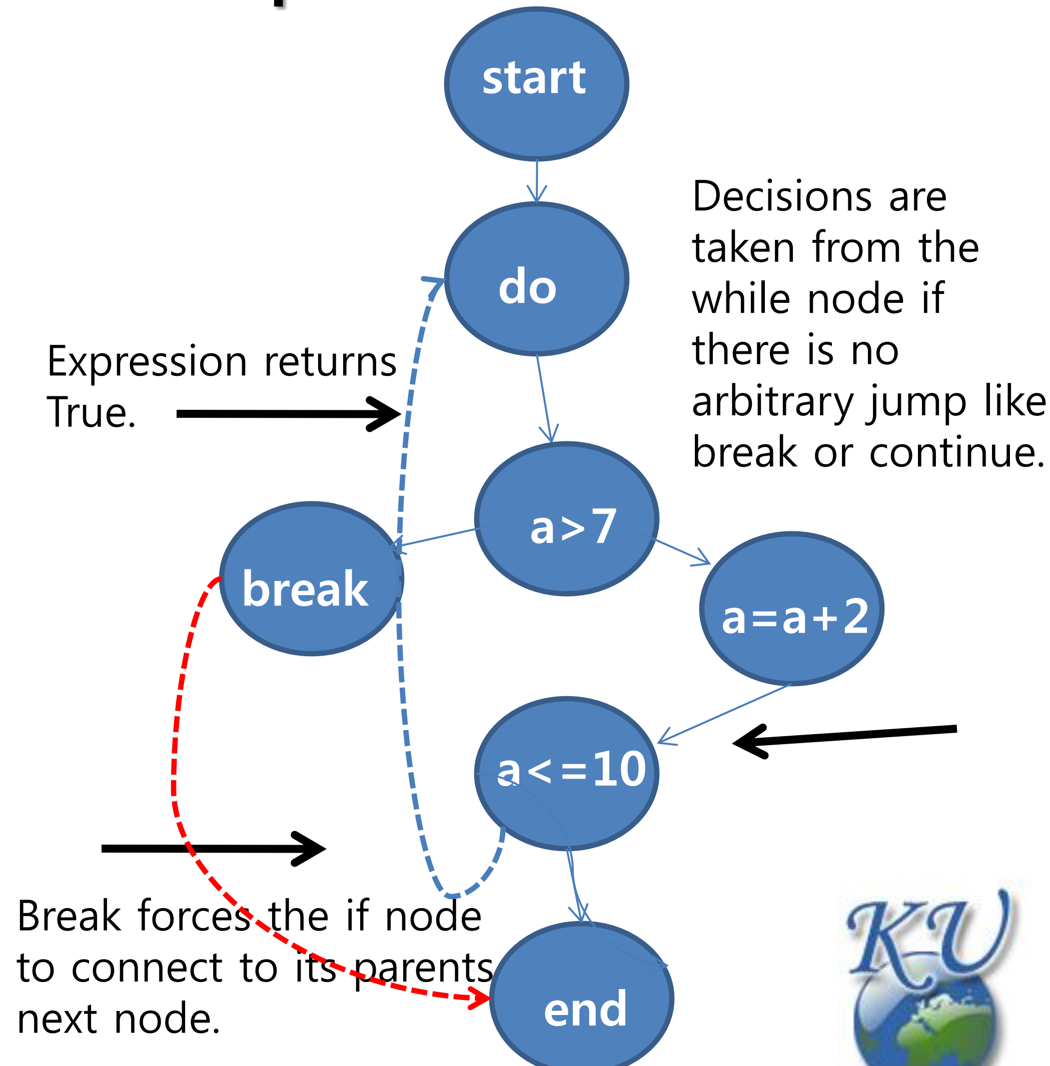


Expr which is an abbreviation of very Long Expression Statements.

Introduction to CFG (12/17)

Do-while statement - Tree to Graph model

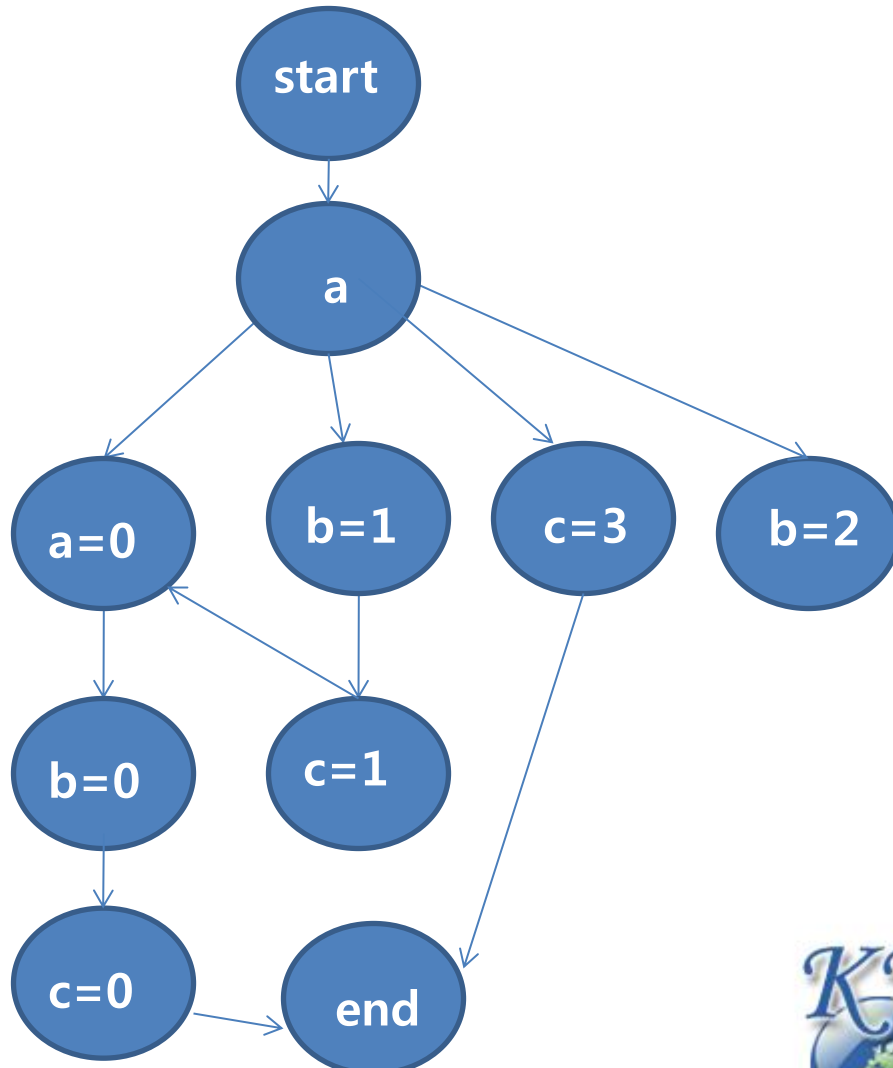
```
Void doWhileTestMethod(int a)
{
 do
 {
   if (a > 7)
   {
     break;
   }
   else
   {
     a = a + 2;
   }
 } while (a <= 10);
}
```



Introduction to CFG (13/17)

Switch case statement

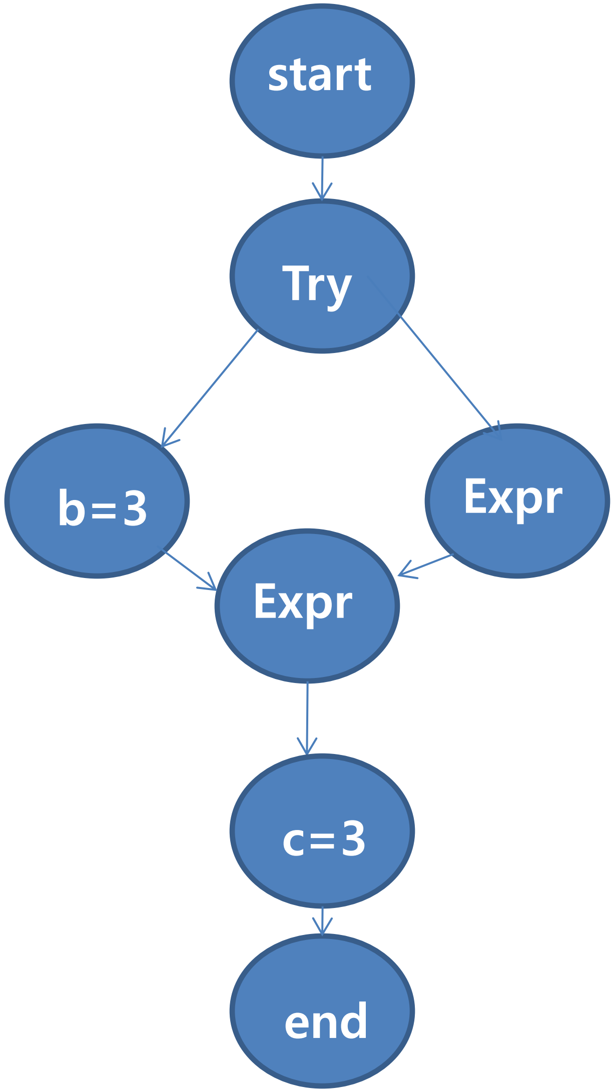
```
void switchTestMethod
(int a, int b, int c)
{
   switch (a)
  {
   case 1:
       b = 2;
   case 2:
       c = 3;  break;
    case 3:
       b = 1;  c = 1;
    default:
       a = 0;  b = 0;
       c = 0;
   }
}
```



Introduction to CFG (14/17)

Try-catch statement

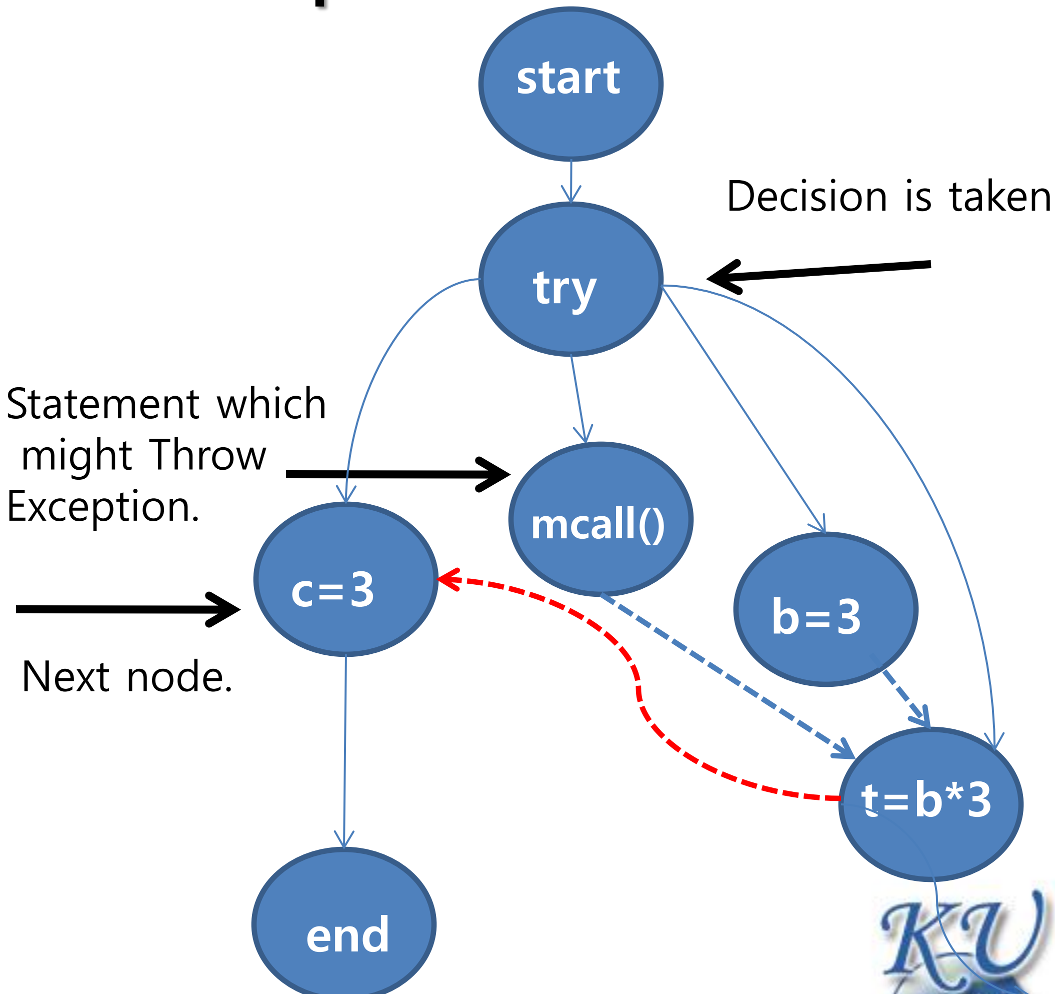
```
void tryCatchTestMethod
(int b, int c, int t)
 {
  try {
 mightThrowAnException(b);
   } catch (Exception e)
   {
     b = 3;
   }
  finally
  {
   t = b*3;
  }
    c= 3;
}
```



Introduction to CFG (15/17)

Try-catch statement - Tree to Graph model

```
void tryCatchTestMethod
(int b, int c, int t)
 {
  try {
 mightThrowAnException(b);
   } catch (Exception e)
   {
     b = 3;
   }
  finally
  {
   t = b*3;
  }
    c= 3;
}
```



-- The dotted lines represent the new connections added by the view algorithm.

Introduction to CFG (16/17)

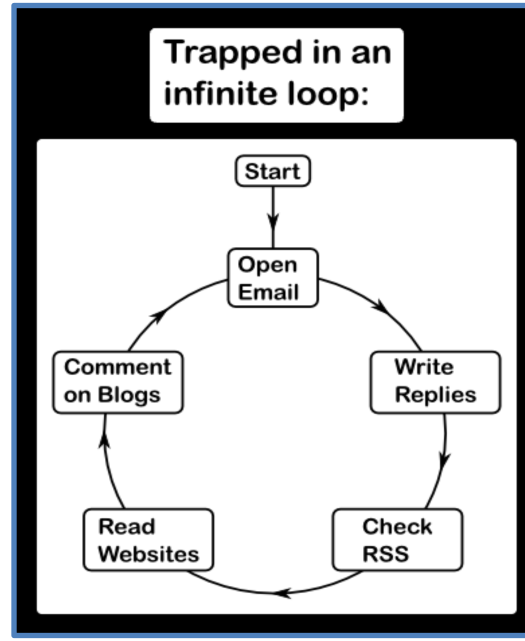
Reachability

✓ If a block/subgraph is not connected from the subgraph containing the entry block, that block is unreachable during any execution, and so is unreachable code; it can be safely removed.

✓ If the exit block is unreachable from the entry block, it indicates an infinite loop. Not all infinite loops are detectable, of course; see Halting problem.

✓ Dead code and some infinite loops are possible.

✓ Even if the programmer didn't explicitly code that way: optimizations like constant propagation and constant folding followed by jump threading could collapse multiple basic blocks into one, cause edges to be removed from a CFG, etc., thus possibly disconnecting parts of the graph.



Introduction to CFG (17/17)

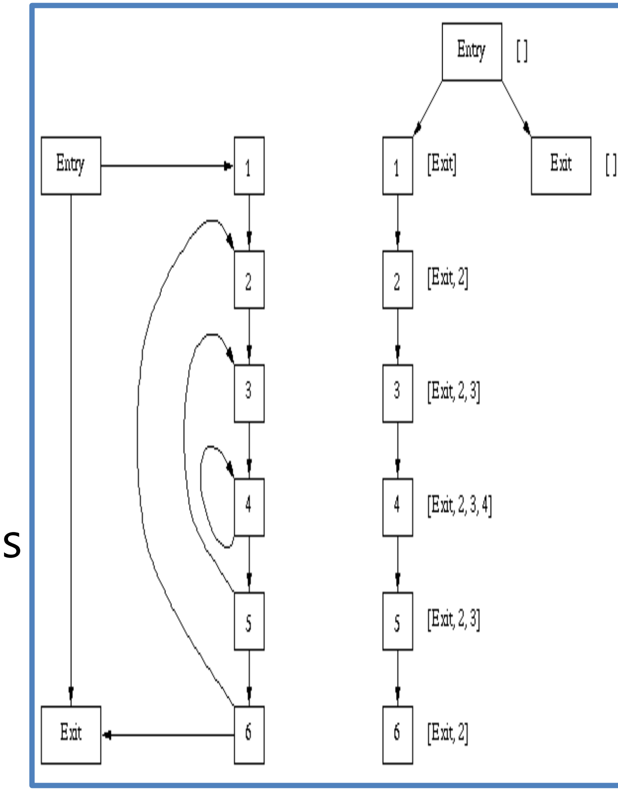
Domination relationship Dominator (graph theory)

✓ A block M dominates a block N if every path from the entry that reaches block N has to pass through block M. The entry block dominates all blocks.

✓ In the reverse direction, block M postdominates block N if every path from N to the exit has to pass through block M. The exit block postdominates all blocks.

✓ It is said that a block M immediately dominates block N if M dominates N, and there is no intervening block P such that M dominates P and P dominates N. In other words, M is the last dominator on all paths from entry to N.

✓Each block has a unique immediate dominator.



* **The dominance frontier**
-entry, exit

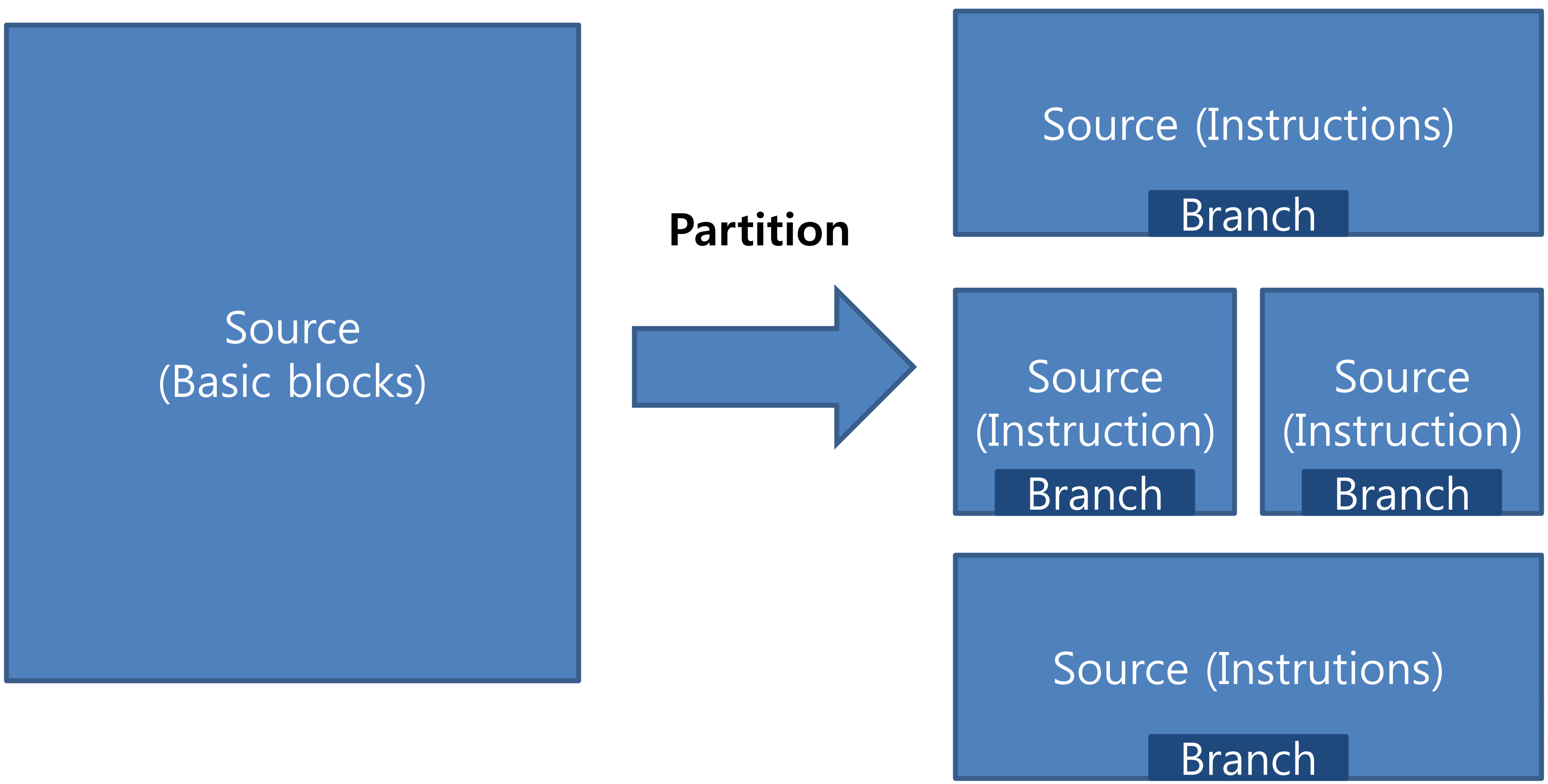
* **6 dominates 2**

Contents

Algorithm to construct Control Flow Graph

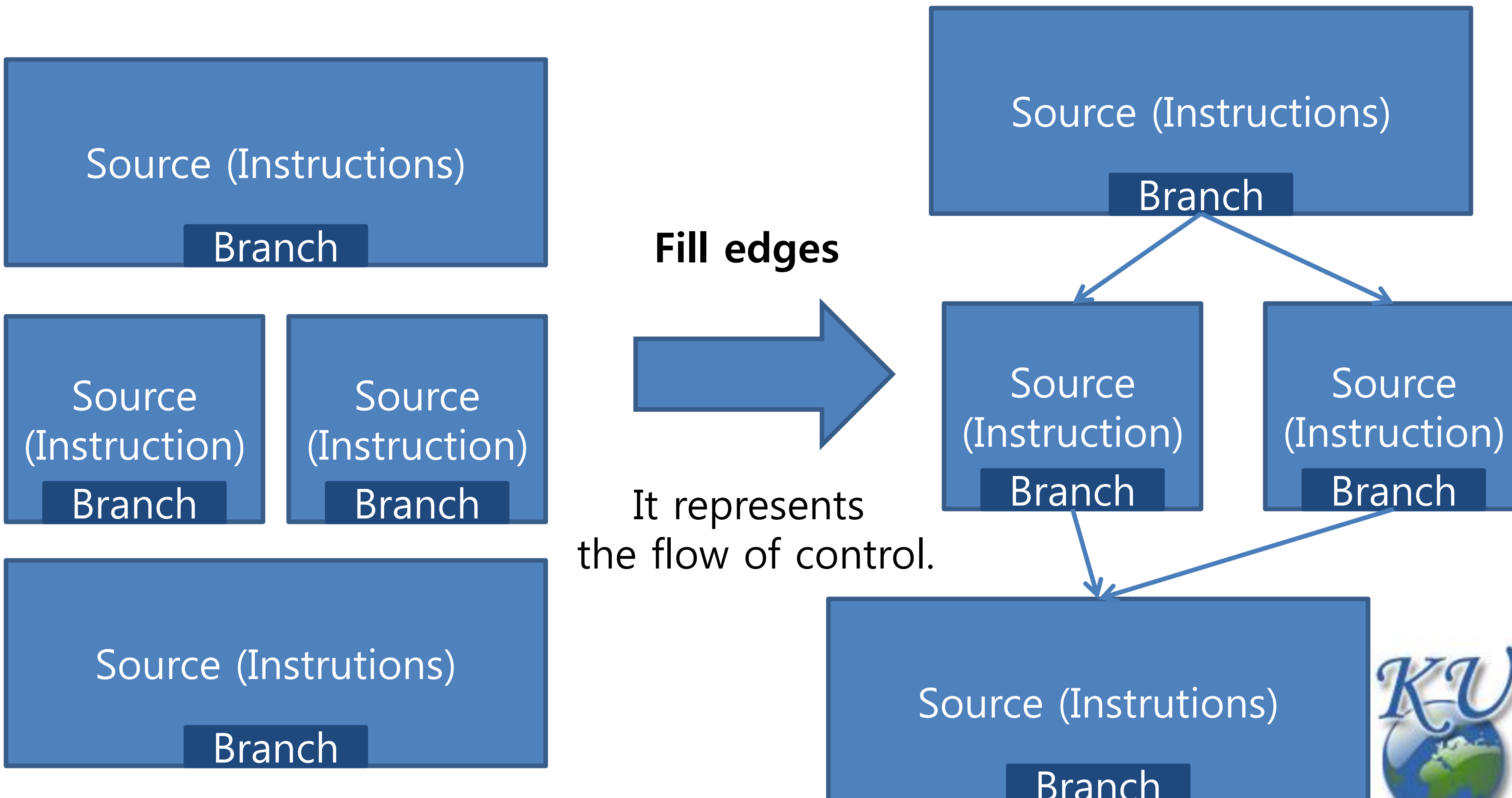
(1/10)

Partition the code into a set of basic blocks



Algorithm to construct Control Flow Graph

Look at the branches in the code and fill in the cfg's edges to represent the flow of control



Algorithm to construct Control Flow Graph

It’s a base Algorithm to construct CFG

```
block_list = initial list of blocks
for each block b in block_list
    remove b from block_list
    branch_found = false
    for each instruction i in b
        if i is a branch
            let branch_found = true
            let countdown = branch-latency
            break
    if branch_found
        for each instruction p in b after i
            decrement countdown
            if countdown = 0 break
    if countdown = 0
        split b at p
        let b′ = remainder of b
        add b′ to block_list
        add edges from b to targets of i
        if b is conditional add edge to b′
  if not branch_found or countdown > 0
    add edge from b to fallthrough of b
```

Algorithm to construct Control Flow Graph

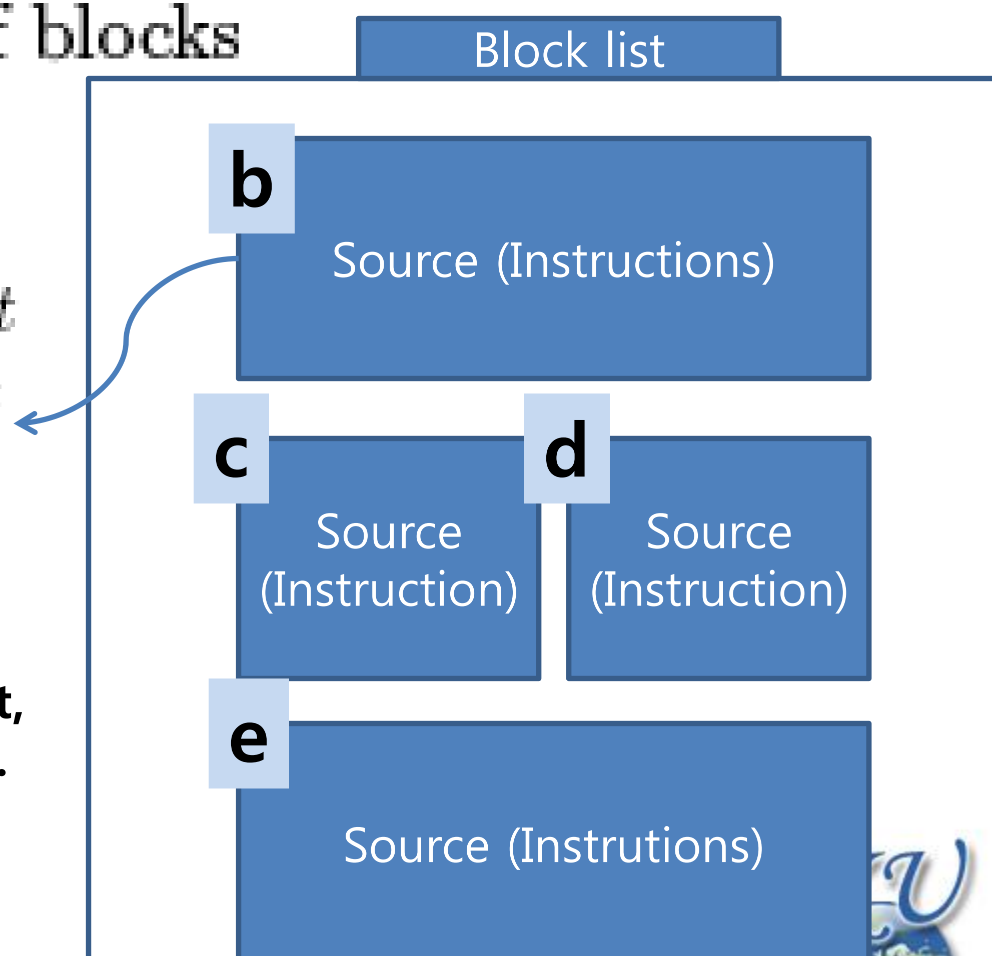
It's a base Algorithm to construct CFG

$$block\_list = \text{initial list of blocks}$$

block_list is list that includes set of blocks.

$$\text{for each block } b \text{ in } block\_list$$
$$\quad \text{remove } b \text{ from } block\_list$$
$$\quad branch\_found = \text{false}$$

It start with block b in block_list. So it remove block b from block_list, and set branch_found variable false.



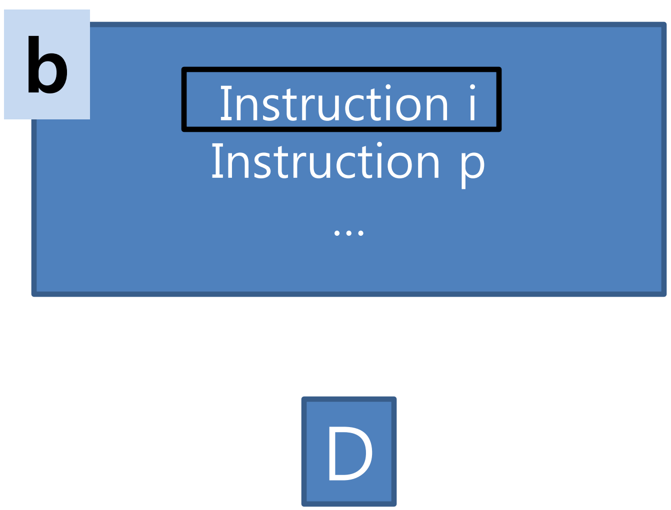
Algorithm to construct Control Flow Graph
(5/10)

It's a base Algorithm to construct CFG

for each instruction i in b
    if i is a branch
        let $branch\_found$ = true
        let $countdown$ = branch-latency
        break

We assume the target of i's branch is 'D' and i has one branch.

Search a branch in each instructions this block.



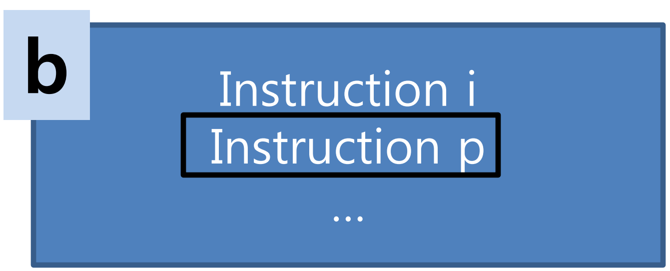
Instruction i is branch?
→ branch_found = true
→ countdown = 1(branch_latency)

Algorithm to construct Control Flow Graph
(6/10)

It's a base Algorithm to construct CFG

if *branch_found*
    for each instruction p in b after i
        decrement *countdown*
        if *countdown* $= 0$ break

Look a after instruction that has branches.



Instruction p,
countdown--;

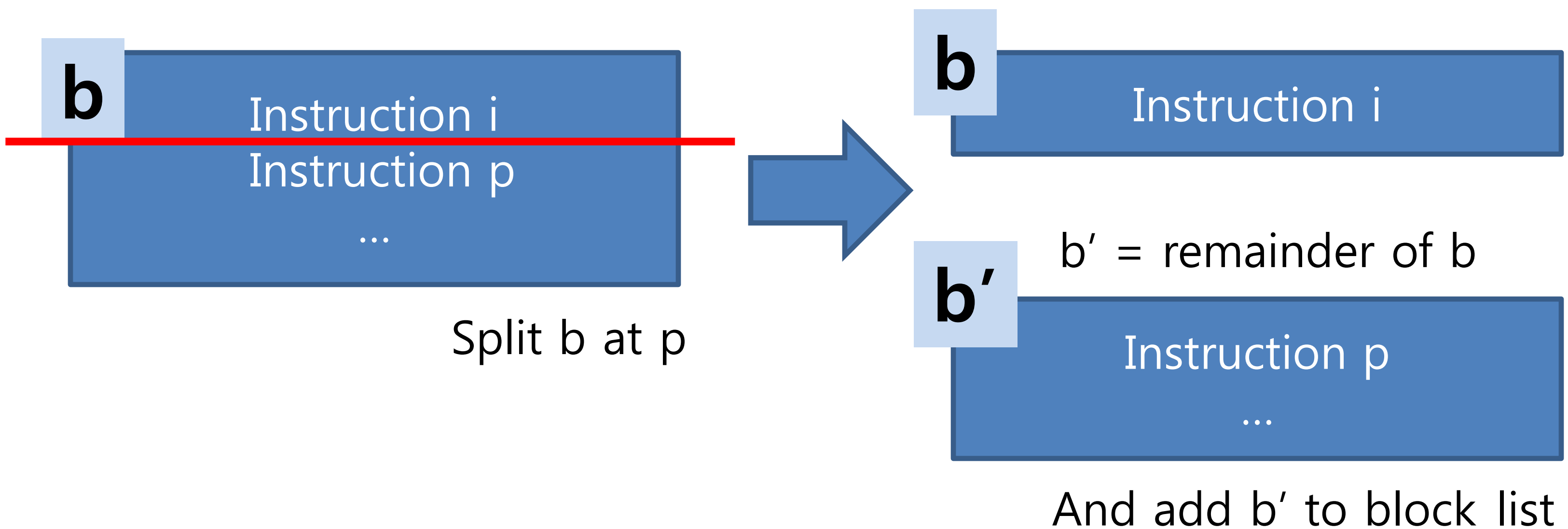
Continue to decrement the countdown variable until it reaches the zero (0) since branch is hidden as much as Branch Latency.

Algorithm to construct Control Flow Graph

It's a base Algorithm to construct CFG

if $countdown = 0$
  split b at p
  let $b' =$ remainder of b
  add b' to $block\_list$
  add edges from b to targets of i
  if b is conditional add edge to b'

It's a case countdown variable is 'zero' when it counts all branches.

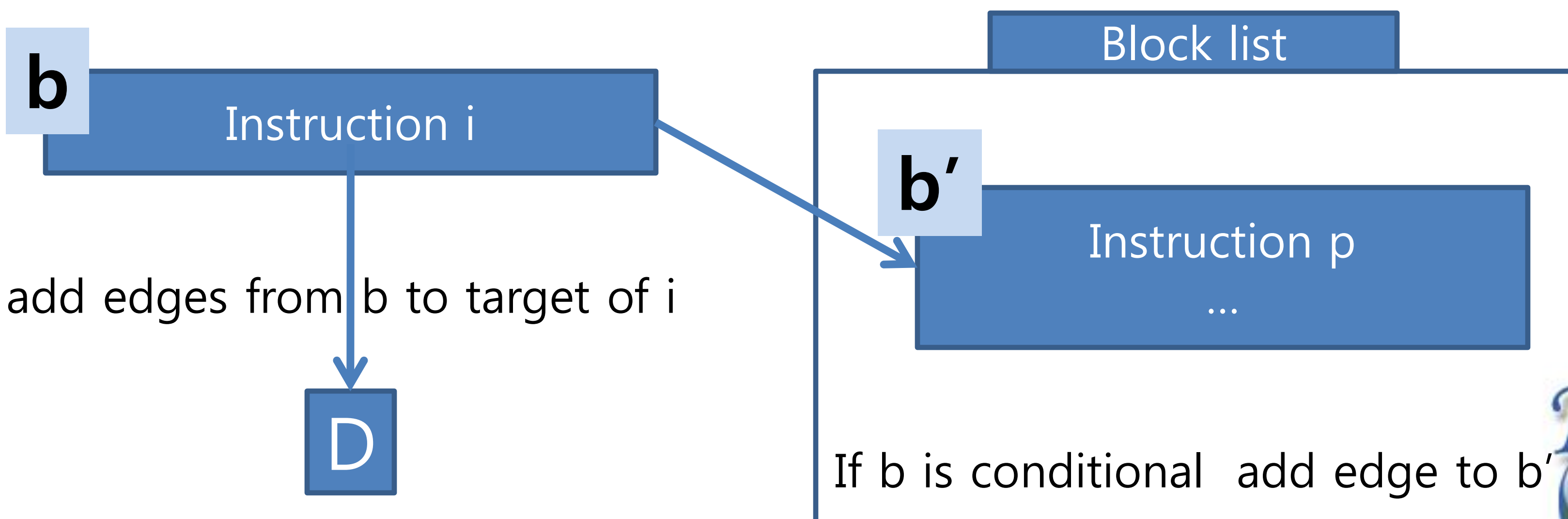


Algorithm to construct Control Flow Graph

It's a base Algorithm to construct CFG

```
if countdown = 0
    split b at p
    let b' = remainder of b
    add b' to block_list
    add edges from b to targets of i
    if b is conditional add edge to b'
```

It's a case countdown variable is 'zero' when it counts all branches.



Algorithm to construct Control Flow Graph

(9/10)

It's a base Algorithm to construct CFG

if not $branch\_found$ or $countdown > 0$
    add edge from b to fallthrough of b

If you do not find branch or do not reduce the number of branch found, draw the basic edge of the next of the block b.

Algorithm to construct Control Flow Graph

(10/10)

It's a base Algorithm to construct dominate relations in CFG

a) $\mathrm{Dom}(n_o) = \{n_o\}$

b) $\mathrm{Dom}(n) = \left(\bigcap_{p \in \mathrm{preds}(n)} \mathrm{Dom}(p) \right) \bigcup \{n\}$

a) Start node n_o only dominates itself, n_o

b) Node n dominates itself (n) and nodes that p dominates.

(p is predecessor node of n)

```
// dominator of the start node is the start itself
Dom(n0) = {n0}
// for all other nodes, set all nodes as the dominators
for each n in N - {n0}
    Dom(n) = N;
// iteratively eliminate nodes that are not dominators
while changes in any Dom(n)
    for each n in N - {n0}:
        Dom(n) = {n} union with intersection over all p in pred(n) of Dom(p)
```

Contents

Statement of purpose

Control Flow Graph

✓ Transfer Source code to Control Flow Graph.
✓ The program detects and traces variables to indicates their flow.
✓ A variable is indicated when it is assigned first time.
✓ If a variable is assigned value, it is indicated by node of rectangle.
✓ If a variable's flow is split by condition, it is indicated by node of rhomboid.
✓ Nodes of one same function are able to be presented by one node.

Contents