

# Code Generation through Passing Jump Targets

# Feedback

- Theoretical
- Record lectures?
- Cover more material

```
static int k = 0;
```

```
static boolean action(int si,  
                      boolean ob,  
                      int sm, int pr) {  
    if (sm + 2*pr > 10 &&  
        !(si <= 5 && ob)) {  
        k++; return true;  
    } else {  
        return false;  
    }  
}
```

# Code Compiled with javac

```
0:  iload_2  
1:  iconst_2  
2:  iload_3  
3:  imul  
4:  iadd  
5:  bipush 10  
7:  if_icmple 29  
10: iload_0  
11: iconst_5  
12: if_icmpgt 19  
15: iload_1  
16: ifne 29  
19: getstatic #2; //Field k  
22: iconst_1  
23: iadd  
24: putstatic #2; //Field k  
27: iconst_1  
28: ireturn  
29: iconst_0  
30: ireturn
```

## Compared to our current translation:

if 'sm+2\*pr > 10' false, immediately ireturns  
if 'si > 5' is true, immediately goes to 'then' part  
no intermediate result for if condition - do  
branches directly  
negation sign eliminated and pushed through  
only one iconst\_0 and one iconst\_1

# Translate This While Loop using Rules that Explicitly Put Booleans on Stack

```
static void count(int from,  
                 int to,  
                 int step) {  
    int counter = from;  
    while (counter < to) {  
        counter = counter + step;  
    }  
}
```

```
nbegin: iload #counter  
        iload #to  
        if_icmplt ntrue  
        iconst_0  
        goto nafter  
ntrue:  iconst_1  
nafter: ifeq nexit  
        iload #counter  
        iload #step  
        iadd  
        istore #counter  
        goto nbegin  
nexit:
```

# Towards More Efficient Translation

# Macro 'branch' instruction

Introduce an imaginary big instruction

**branch(c,nTrue,nFalse)**

Here

**c** is a potentially *complex* Java boolean expression,  
*that* is the main reason why branch is not a real instruction

**nTrue** is label to jump to when **c** evaluates to true

**nFalse** is label to jump to when **c** evaluates to false

no "fall through" – always jumps (symmetrical)

We show how to:

- use **branch** to compile if, while, etc.
- expand **branch** recursively into concrete bytecodes

# Using **branch** in Compilation

```
[ if (c) t else e ] =  
    branch(c,nTrue,nFalse)  
nTrue: [ t ]  
    goto nAfter  
nFalse: [ e ]  
nAfter:
```

```
[ while (c) s ] =  
test: branch(c,body,exit)  
body: [ s ]  
    goto test  
exit:
```

# Decomposing **branch**

**branch(!c,nThen,nElse) =**  
    **branch(c,nElse,nThen)**

**branch(c1 && c2,nThen,nElse) =**  
    **branch(c1,nNext,nElse)**  
nNext: **branch(c2,nThen,nElse)**

**branch(c1 || c2,nThen,nElse) =**  
    **branch(c1,nThen,nNext)**  
nNext: **branch(c2,nThen,nElse)**

**branch(true,nThen,nElse) =**  
    **goto nThen**

**branch(false,nThen,nElse) =**  
    **goto nElse**

**boolean var b with slot N**

**branch(b,nThen,nElse) =**  
    **iload\_N**  
    **ifeq nElse**  
    **goto nThen**

# Compiling Relations

branch(e1 R e2,nThen,nElse) =

[ e1 ]

R can be <,>==,!=,<=,>=,...

[ e2 ]

**if\_icmp**R nThen

**goto** nElse

# Putting boolean variable on the stack

Consider storing  $x = c$

where  $x, c$  are boolean and  $c$  has **&&**, **||**

How to put result of **branch** on stack to allow **istore**?

**[ b = c ] = branch(c, nThen, nElse)**

nThen: **iconst\_1**

**goto** nAfter

nElse: **iconst\_0**

nAfter: **istore #b**

# Compare Two Translations of This While Loop

```
while (counter < to) {  
    counter = counter + step;  
}
```

old one:

```
nbegin: iload #counter  
        iload #to  
        if_icmplt ntrue  
        iconst_0  
        goto nafter  
ntrue:  iconst_1  
nafter: ifeq nexit  
        iload #counter  
        iload #step  
        iadd  
        istore #counter  
        goto nbegin  
nexit:
```

new one:

```
test:   iload #counter  
        iload #to  
        if_icmplt body  
        goto exit  
body:   iload #counter  
        iload #step  
        iadd  
        istore #counter  
        goto test  
exit:
```

# Complex Boolean Expression: Example

Generate code for this:

```
if ((x < y)&& !((y < z) && ok))      branch(x<y,n1,else)
  return                            n1:  branch(y<z,n2,then)
else                                  n2:  branch(ok,else,then)
  y = y + 1                          then: return
                                     goto after
                                     else: iload #y
                                     iconst_1
                                     iadd
                                     istore #y
                                     after:
```

Compare to the  
old translation.

# Implementing **branch**

- Option 1: emit code using **branch**, then rewrite
- Option 2: **branch** is a just a function in the compiler that expands into instructions

**branch**(c,nTrue,nFalse)



```
def compileBranch(c:Expression,  
    nTrue : Label, nFalse : Label) : List[Bytecode] =  
{ ... }
```

The function takes **two destination labels**.

# More Complex Control Flow

# Destination Parameters in Compilation

- To compilation functions [...] pass a **label** to which instructions should jump **after** they finish.
  - No fall-through

```
[ x = e ] after =           // new parameter 'after'  
[ e ]  
istore #x  
goto after                // at the end jump to it
```

```
[ s1 ; s2 ] after =
```

```
freshL: [ s1 ] freshL  
[ s2 ] after
```

we could have any junk in here  
because ([s1] freshL) ends in a jump

# Translation of **if**, **while**, **return** with one 'after' parameter

[ **if** (c) t **else** e ] after =

**branch**(c,nTrue,nFalse)

nTrue: [ t ] after

nFalse: [ e ] after

[ **while** (c) s ] after =

test: **branch**(c,body,after)

body: [ s ] test

[ **return** exp ] after =

[ exp ]

**ireturn**

# Generated Code for Example

[ if (x < y) return; else y = 2; ] after =

**iload #x**

**iload #y**

**if\_icmp\_lt nTrue**

**goto nFalse**

nTrue: **return**

nFalse: **iconst\_2**

**istore #y**

**goto after**

Note: no **goto** after **return** because

- translation of 'if' does not generate goto as it did before, since it passes it to the translation of the body
- translation of 'return' knows it can ignore the 'after' parameter

# break statement

A common way to exit from a loop is to use a 'break' statement e.g.

```
while (true) {  
  code1  
  if (cond) break  
  cond2  
}
```

Consider a language that has expressions, assignments, the {...} blocks, 'if' statements, while, and a 'break' statement.

**The 'break' statement exits the innermost loop and can appear inside arbitrarily complex blocks and if conditions.**

How would translation scheme for such construct look like?

# Two Destination Parameters

[ s1 ; s2 ] after brk =  
                  [ s1 ] freshL brk  
freshL: [ s2 ] after brk

[ x = e ] after brk =  
[ e ]  
**istore #x**  
**goto after**

[ **return** exp ] after brk =  
[ exp ]  
**ireturn**

[ **break** ] after brk =  
**goto brk**

[ **while** (c) s ] **after** brk =  
test: **branch**(c,body,after)  
body: [ s ] test **after**

this is where the second  
parameter gets bound to  
the exit of the loop



# if with two parameters

[ **if** (c) t **else** e ] after brk =

**branch**(c,nTrue,nFalse)

nTrue: [ t ] after brk

nFalse: [ e ] after brk

# **break** and **continue** statements?

Three parameters!

[ **break** ] after brk **cont** =  
**goto** brk

[ **continue** ] after brk **cont** =  
**goto cont**

[ **while** (c) s ] after brk **cont** =  
**test:** **branch**(c,body,after)  
body: [ s ] test after **test**

# Some High-Level Instructions for JVM

# Method Calls

Invoking methods (arguments pushed onto stack)

**invokestatic**

**invokevirtual**

Returning value from methods:

**ireturn** – take integer from stack and return it

**areturn** – take reference from stack and return it

**return** – return from a method returning **'void'**

# invokestatic

invokestatic

indexbyte1

indexbyte2

..., [arg1, [arg2 ...]] → ...

The unsigned **indexbyte1** and **indexbyte2** are used to construct an index into the run-time **constant pool** of the current class (§2.6), where the value of the index is **(indexbyte1 << 8) | indexbyte2**. The run-time constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3). The resolved method must not be an instance initialization method (§2.9) or the class or interface initialization method (§2.9). It must be static, and therefore cannot be abstract.

On successful resolution of the method, the class that declared the resolved method is initialized (§5.5) if that class has not already been initialized.

The operand stack must contain nargs argument values, where the number, type, and order of the values must be consistent with the descriptor of the resolved method.

If the method is synchronized, the monitor associated with the resolved Class object is entered or reentered as if by execution of a monitorenter instruction (§monitorenter) in the current thread.

If the method is not native, the **nargs argument values are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The nargs argument values are consecutively made the values of local variables of the new frame, with arg1 in local variable 0 (or, if arg1 is of type long or double, in local variables 0 and 1) and so on.** Any argument value that is of a floating-point type undergoes value set conversion (§2.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java Virtual Machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

If the method is native and the platform-dependent code that implements it has not yet been bound (§5.6) into the Java Virtual Machine, that is done. The nargs argument values are popped from the operand stack and are passed as parameters to the code that implements the method. Any argument value that is of a floating-point type undergoes value set conversion (§2.8.3) prior to being passed as a parameter. The parameters are passed and the code is invoked in an implementation-dependent manner. When the platform-dependent code returns, the following take place:

If the native method is synchronized, the monitor associated with the resolved Class object is updated and possibly exited as if by execution of a monitorexit instruction (§monitorexit) in the current thread.

If the native method returns a value, the return value of the platform-dependent code is converted in an implementation-dependent way to the return type of the native method and pushed onto the operand stack.

# invokevirtual

invokevirtual

indexbyte1

indexbyte2

..., objectref, [arg1, [arg2 ...]] →...

Description

The unsigned indexbyte1 and indexbyte2 are used to construct an index into the run-time constant pool of the current class (§2.6), where the value of the index is **(indexbyte1 << 8) | indexbyte2**.

The run-time constant pool item at that index must be a symbolic reference to a method (§5.1), which gives the name and descriptor (§4.3.3) of the method as well as a symbolic reference to the class in which the method is to be found. The named method is resolved (§5.4.3.3). The resolved method must not be an instance initialization method (§2.9) or the class or interface initialization method (§2.9). Finally, if the resolved method is protected (§4.6), and it is a member of a superclass of the current class, and the method is not declared in the same run-time package (§5.3) as the current class, then the class of objectref must be either the current class or a subclass of the current class.

If the resolved method is not signature polymorphic (§2.9), then the invokevirtual instruction proceeds as follows.

Let C be the class of objectref. The actual method to be invoked is selected by the following lookup procedure:

If C contains a declaration for an instance method m that overrides (§5.4.5) the resolved method, then m is the method to be invoked, and the lookup procedure terminates.

Otherwise, if C has a superclass, this same lookup procedure is performed recursively using the direct superclass of C; the method to be invoked is the result of the recursive invocation of this lookup procedure.

Otherwise, an AbstractMethodError is raised.

The objectref must be followed on the operand stack by nargs argument values, where the number, type, and order of the values must be consistent with the descriptor of the selected instance method.

If the method is synchronized, the monitor associated with objectref is entered or reentered as if by execution of a monitorenter instruction (§monitorenter) in the current thread.

If the method is not native, the **nargs argument values and objectref are popped from the operand stack. A new frame is created on the Java Virtual Machine stack for the method being invoked. The objectref and the argument values are consecutively made the values of local variables of the new frame, with objectref in local variable 0, arg1 in local variable 1 (or, if arg1 is of type long or double, in local variables 1 and 2), and so on.** Any argument value that is of a floating-point type undergoes value set conversion (§2.8.3) prior to being stored in a local variable. The new frame is then made current, and the Java Virtual Machine pc is set to the opcode of the first instruction of the method to be invoked. Execution continues with the first instruction of the method.

...

# Translating Method Calls: Example

[ x = objExpr.myMethodName(e1,e2) ] =

[ objExpr ]

[ e1 ]

[ e2 ]

invokevirtual #13

istore #x

constant pool area:

0: "hello, world"

1:

...

13: className.myMethodName/(II)I

...



# Rule for Method Call Translation

[ objExpr.myMethodName(e1,...,en) ] =

[ objExpr ]

[ e1 ]

...

[ en ]

**invokevirtual** #constantPoolAddr

# Objects and References

**ifnull** label - consume top-of-stack reference and jump if it is null

**ifnonnull** label - consume top-of-stack reference, jump if *not* null

**new** #className - create fresh object of class pointed to by the offset  
#className in the constant pool  
(does not invoke any constructors)

**getfield** #field – consume object reference from stack,  
**obj.field** then dereference the field of that object given  
by (field,class) stored in the #field pointer in the constant pool  
and put the value of the field on the stack

**putfield** #field - consume an object reference obj and a value v  
**obj.field= v** from the stack and store v it in the #field of obj

“If the field descriptor type is boolean, byte, char, short, or int, then the value must be an int.”

# Array Manipulation

a = reference - “address” arrays

i = int arrays (and some other int-like value types)

Selected array manipulation operations:

**newarray, anewarray, multianewarray** – allocate an array object and put a reference to it on the stack

**aaload, iaload** – take: a reference to array and index from stack  
load the value from array and push it onto the stack

**aastore, iastore** – take: a reference to array, an index, a value from stack, store the value into the array index

**arraylength** – retrieve length of the array

Java arrays store the size of the array and its type, which enables run-time checking of array bounds and object types.

# There are Floating Point Operations...

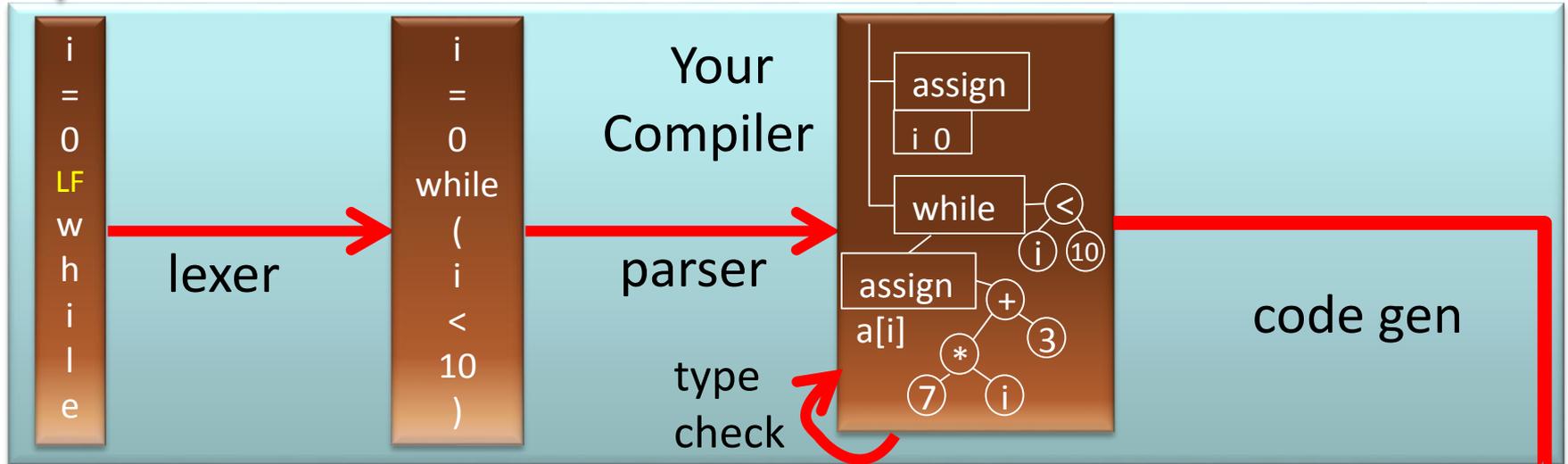
- fadd
- faload (for floating point arrays)
- fastore (for floating point arrays)
- fcmp<op>
- fconst\_<f>
- fdiv
- fload
- fload\_<n>
- fmul
- fneg
- frem
- freturn
- fstore
- fstore\_<n>
- fsub

When needed,  
READ THE JVM Spec 😊

Covered!

```
i=0  
while (i < 10) {  
  a[i] = 7*i+3  
  i = i + 1 }  
}
```

source code  
simplified Java-like  
language



characters

words

trees

# Java Virtual Machine (JVM) Bytecode

```
21: iload_2  
22: iconst_2  
23: iload_1  
24: imul  
25: iadd  
26: iconst_1  
27: iadd  
28: istore_2
```

**OPTIONAL MATERIAL:**

**Abstract Interpretation**

(Cousot, Cousot 1977)

**also known as**

**Data-Flow Analysis**

(Kildall 1973)

# Example: Constant propagation

## Here is why it is useful

```
int a, b, step, i;
boolean c;
a = 0;
b = a + 10;
step = -1;
if (step > 0) {
    i = a;
} else {
    i = b;
}
c = true;
while (c) {
    print(i);
    i = i + step; // can emit decrement
    if (step > 0) {
        c = (i < b);
    } else {
        c = (i > a); // can emit better instruction here
    } // insert here (a = a + step), redo analysis
}
```

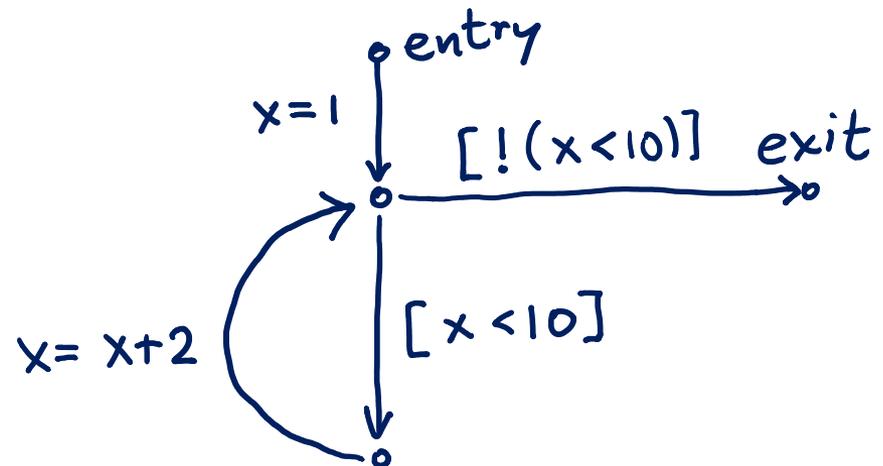
# Goal of Data-Flow Analysis

Automatically compute information about the program

- Use it to report errors to user (like type errors)
- Use it to optimize the program

Works on control-flow graphs:  
(like flow-charts)

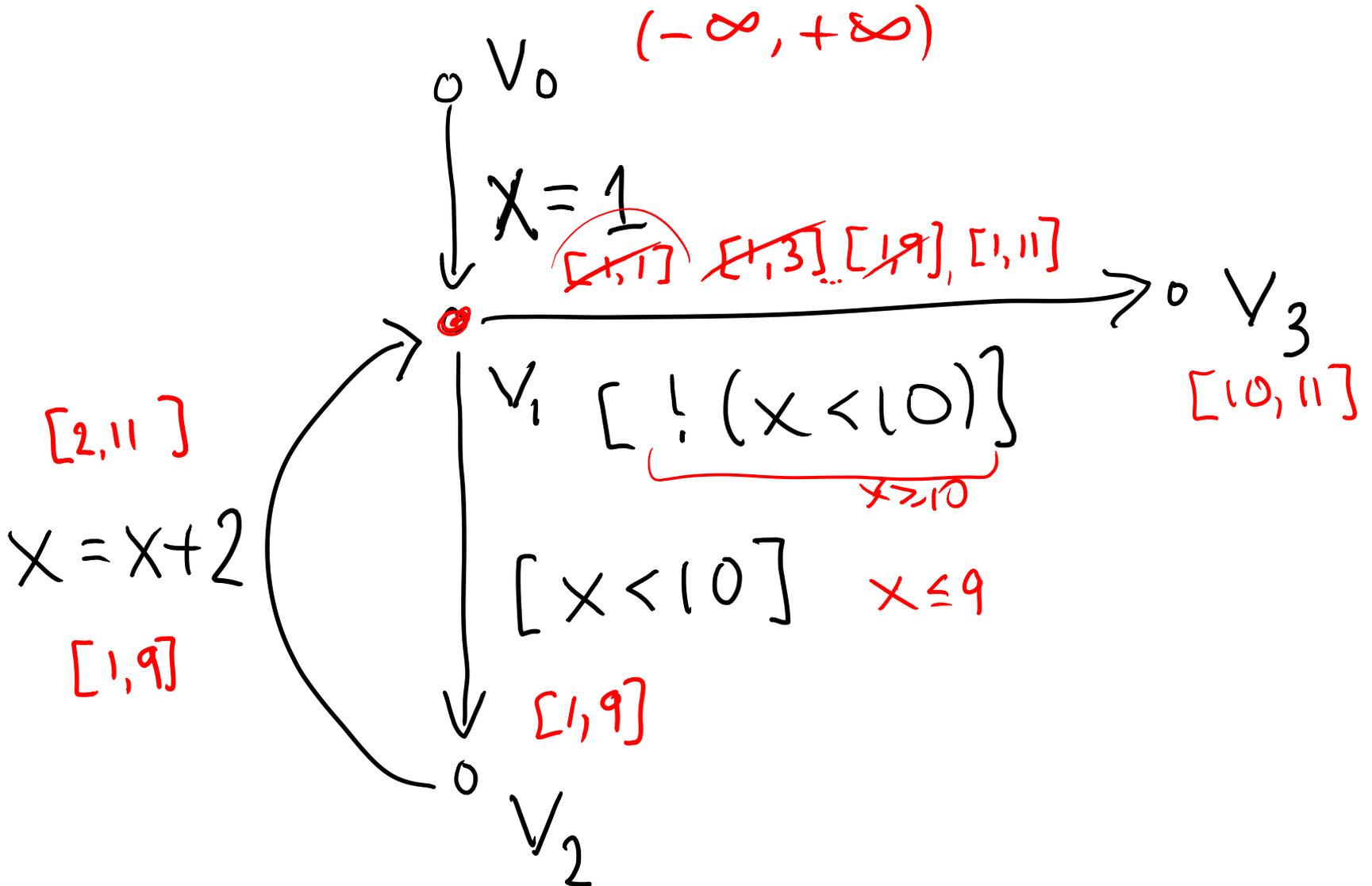
```
x = 1  
while (x < 10) {  
  x = x + 2  
}
```



# Interpretation and Abstract Interpretation

- Control-Flow graph is similar to AST
- We can
  - interpret control flow graph
  - generate machine code from it (e.g. LLVM, gcc)
  - abstractly interpret it: do not push values, but **approximately compute supersets of possible values** (e.g. intervals, types, etc.)

# Compute Range of x at Each Point



# What we see in the sequel

1. How to compile abstract syntax trees into control-flow graphs
2. Lattices, as structures that describe abstractly sets of program states (facts)
3. Transfer functions that describe how to update facts
4. Basic idea of fixed-point iteration

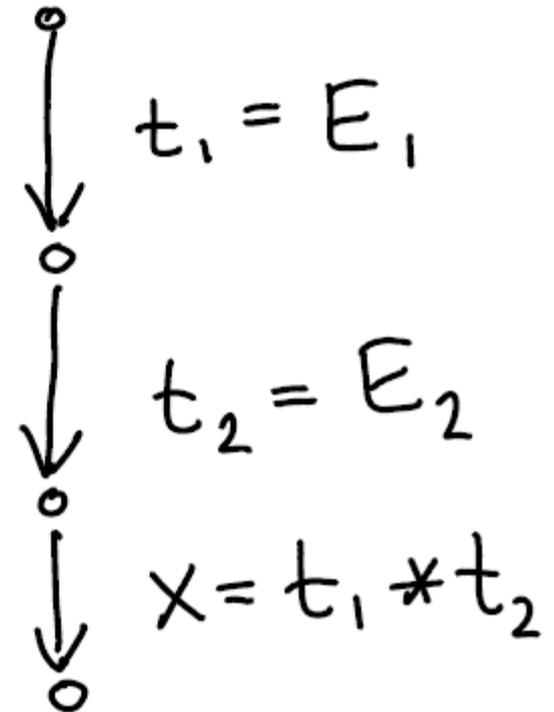
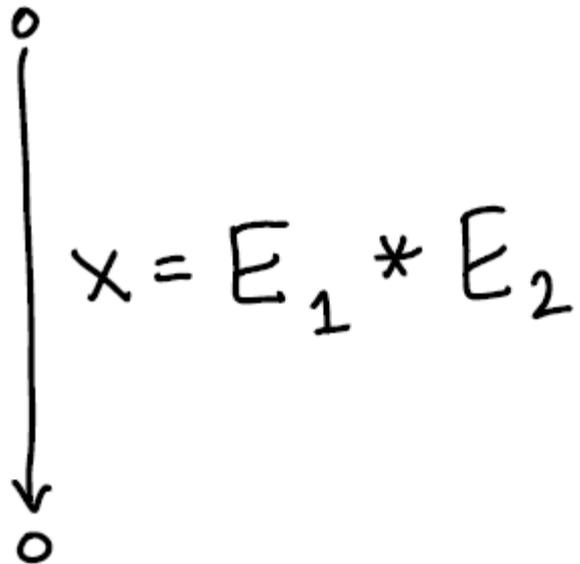
# Generating Control-Flow Graphs

- Start with graph that has one entry and one exit node and label is entire program
- Recursively decompose the program to have more edges with simpler labels
- When labels cannot be decomposed further, we are done

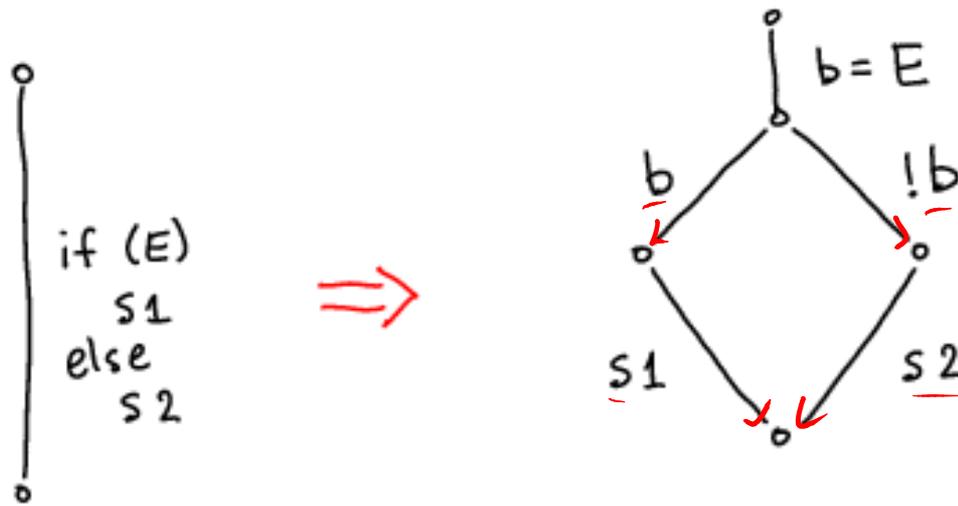
# Flattening Expressions

for simplicity and ordering of side effects

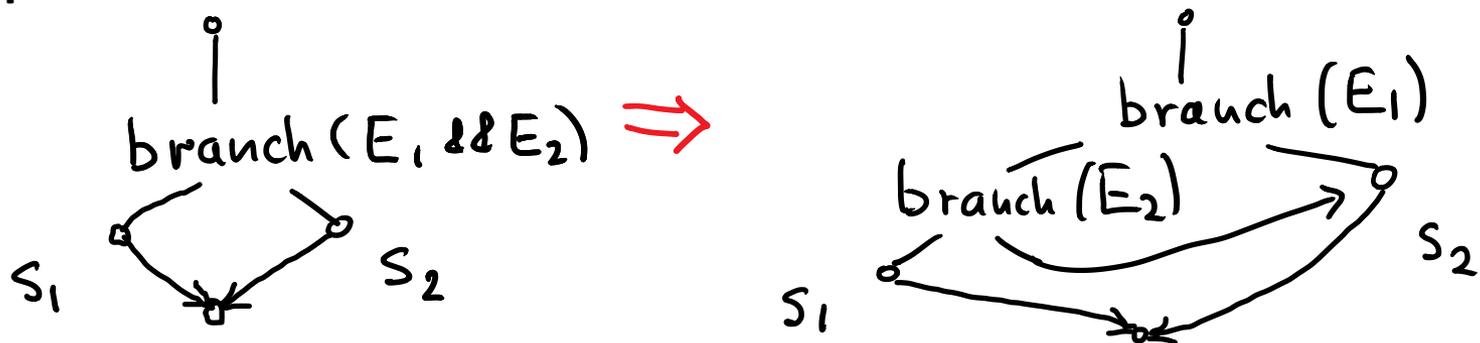
$E_1, E_2$  - complex expressions  
 $t_1, t_2$  - fresh variables



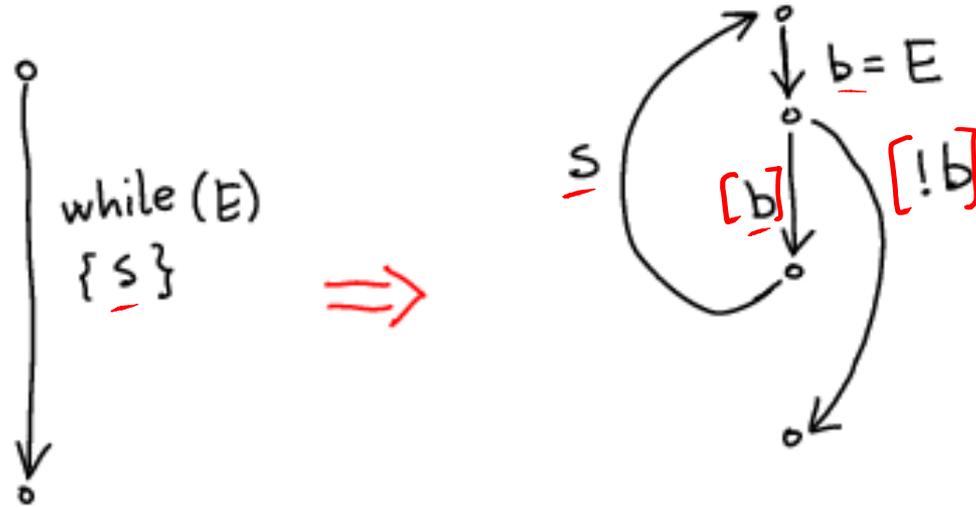
# If-Then-Else



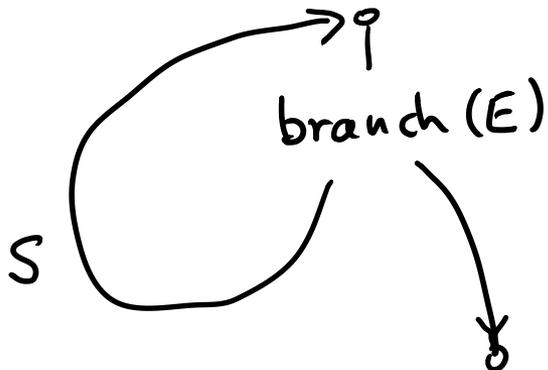
Better translation uses the "branch" instruction approach: have two destinations



# While



Better translation uses the "branch" instruction



# Example 1: Convert to CFG

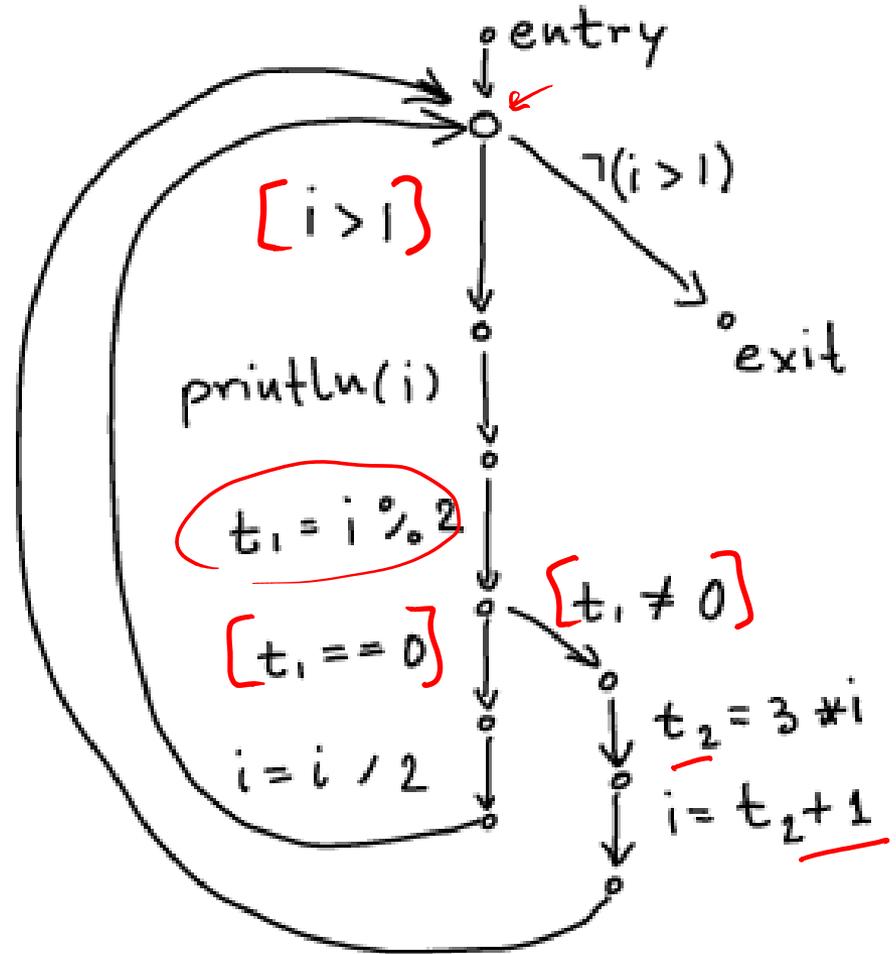
```
while (i < 10) {  
    println(j);  
    i = i + 1;  
    j = j + 2*i + 1;  
}
```

## Example 2: Convert to CFG

```
int i = n;  
while (i > 1) {  
    println(i);  
    if (i % 2 == 0) {  
        i = i / 2;  
    } else {  
        i = 3*i + 1;  
    }  
}
```

# Example 2 Result

```
int i = n;  
while (i > 1) {  
  println(i);  
  if (i % 2 == 0) {  
    i = i / 2;  
  } else {  
    i = 3 * i + 1;  
  }  
}
```



# Translation Functions

$$\begin{aligned} [s_1 ; s_2] v_{\text{source}} v_{\text{target}} &= \\ [s_1] v_{\text{source}} v_{\text{fresh}} & \\ [s_2] v_{\text{fresh}} v_{\text{target}} & \end{aligned}$$

**insert** ( $v_s, \text{stmt}, v_t$ ) =  
cfg = cfg + ( $v_s, \text{stmt}, v_t$ )

**[ branch( $x < y$ ) ]**  $v_{\text{source}} v_{\text{true}}$   
 $v_{\text{false}} =$

**insert**( $v_{\text{source}}, [x < y], v_{\text{true}}$ );  
**insert**( $v_{\text{source}}, [!(x < y)], v_{\text{false}}$ )

**[  $x = y + z$  ]**  $v_s v_t =$  **insert**( $v_s, x = y + z, v_t$ )

*when  $y, z$  are constants or variables*

# Analysis Domain (D)

## Lattices

# Lattice

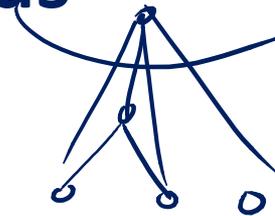
**Partial order:** binary relation  $\leq$  (subset of some  $D^2$ ) which is

- reflexive:  $x \leq x$
- anti-symmetric:  $x \leq y \wedge y \leq x \rightarrow x = y$
- transitive:  $x \leq y \wedge y \leq z \rightarrow x \leq z$

**Lattice** is a partial order in which every **two-element** set has **least** among its upper bounds and **greatest** among its lower bounds

- Lemma: if  $(D, \leq)$  is lattice and  $D$  is finite, then lub and glb exist for every finite set

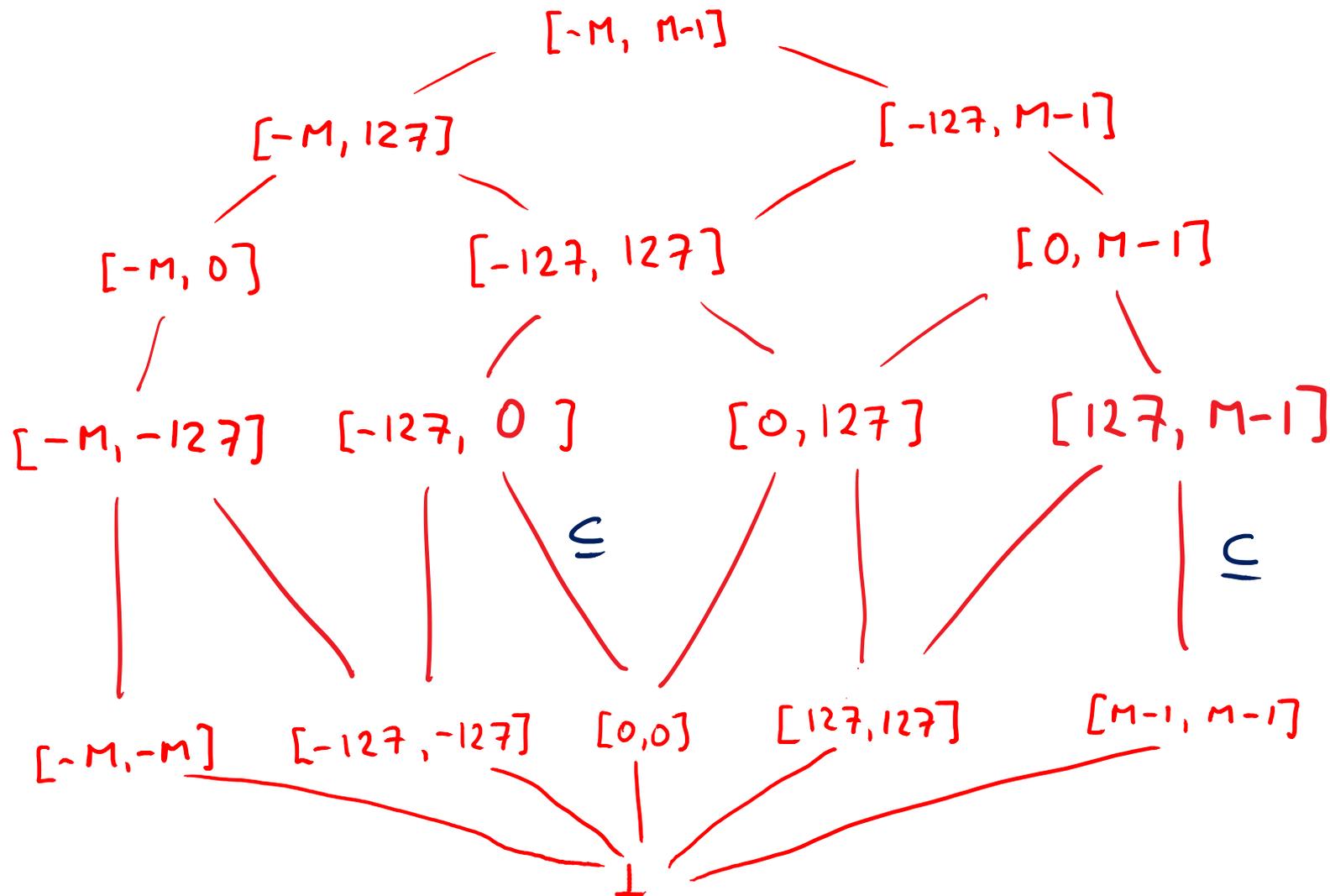
$$\cap \quad \sqcup \quad \sqcup \{a, b, c\}$$



# Graphs and Partial Orders

- If the domain is finite, then partial order can be represented by directed graphs
  - if  $x \leq y$  then draw edge from  $x$  to  $y$
- For partial order, no need to draw  $x \leq z$  if  $x \leq y$  and  $y \leq z$ . So we only draw non-transitive edges
- Also, because always  $x \leq x$ , we do not draw those self loops
- Note that the resulting graph is acyclic: if we had a cycle, the elements must to be equal

# Domain of Intervals $[a,b]$ where $a,b \in \{-M, -127, 0, 127, M-1\}$



# Defining Abstract Interpretation

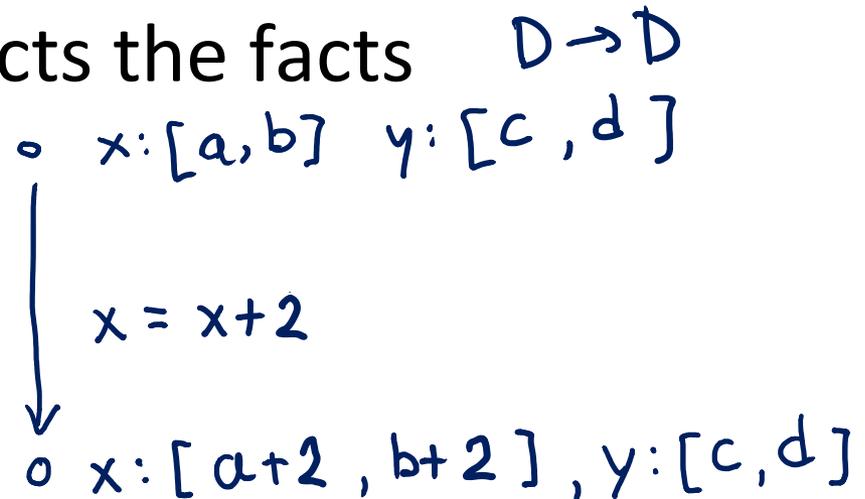
**Abstract Domain D** describing which information to compute – this is often a lattice

- inferred types for each variable:  $x:T1, y:T2$
- interval for each variable  $x:[a,b], y:[a',b']$

**Transfer Functions,  $[[st]]$**  for each statement **st**, how this statement affects the facts  $D \rightarrow D$

- Example:

$$\begin{aligned} & [[x = x + 2]](x:[a,b], \dots) \\ & = (x:[a+2, b+2], \dots) \end{aligned}$$



# For now, we consider arbitrary integer bounds for intervals

- Really 'Int' should be BigInt, as in e.g. Haskell
- Often we must analyze machine integers
  - need to correctly represent (and/or warn about) overflows and underflows
  - fundamentally same approach as for unbounded integers
- For efficiency, many analysis do not consider arbitrary intervals, but only a subset of them
- We consider as the domain
  - empty set (denoted  $\perp$ , pronounced “bottom”)
  - all intervals  $[a,b]$  where  $a,b$  are integers and  $a \leq b$ , or where we allow  $a = -\infty$  and/or  $b = \infty$
  - set of all integers  $[-\infty, \infty]$  is denoted  $T$ , pronounced “top”

# Find Transfer Function: Plus

Suppose we have only two integer variables:  $x, y$

◦  $x: [a, b] \quad y: [c, d]$   
↓  
◦  $x: [a', b'] \quad y: [c', d']$

$x = x + y$

If  $a \leq x \leq b \quad c \leq y \leq d$

and we execute  $x = x + y$

then  $x' = x + y$   
 $y' = y$

so

$a + c \leq x' \leq$

$b + d$   
 $c \leq y' \leq d$

So we can let

$a' = a + c \quad b' = b + d$

$c' = c \quad d' = d$

# Find Transfer Function: Minus

Suppose we have only two integer variables:  $x, y$

$$\begin{array}{l} \circ \\ x: [a, b] \quad y: [c, d] \\ \downarrow \\ y = x - y \\ \circ \\ x: [a', b'] \quad y: [c', d'] \end{array}$$

If

and we execute  $y = x - y$

then

So we can let

$$\begin{array}{ll} a' = a & b' = b \\ c' = a - d & d' = b - c \end{array}$$

# Transfer Functions for Tests

Tests e.g.  $[x > 1]$  come from translating if, while into CFG

$x: [-10, 10]$

if ( $x > 1$ ) {

$x:$

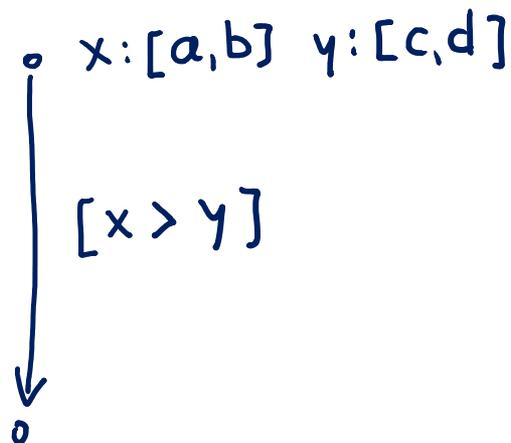
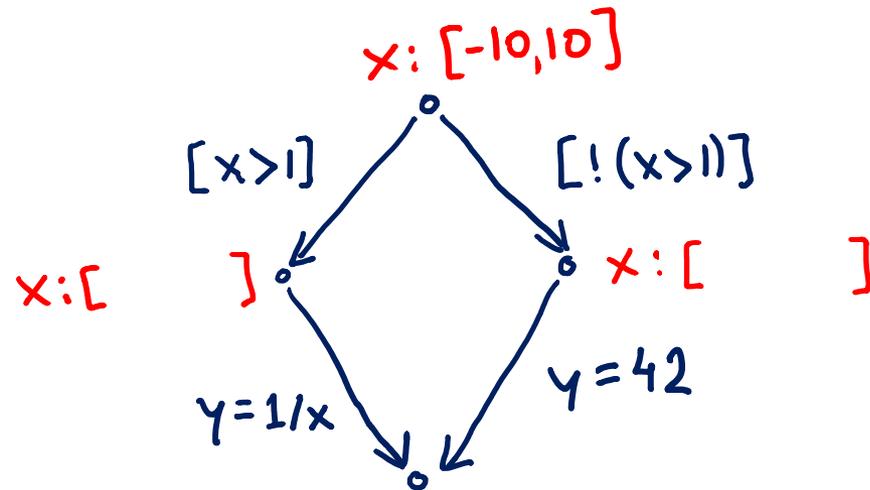
$y = 1 / x$

} else {

$x:$

$y = 42$

}



# Joining Data-Flow Facts

$x: [-10, 10]$   $y: [-1000, 1000]$

if ( $x > 0$ ) {

$x:$

$y:$

$y = x + 100$

$x:$

$y:$

} else {

$x:$

$y:$

$y = -x - 50$

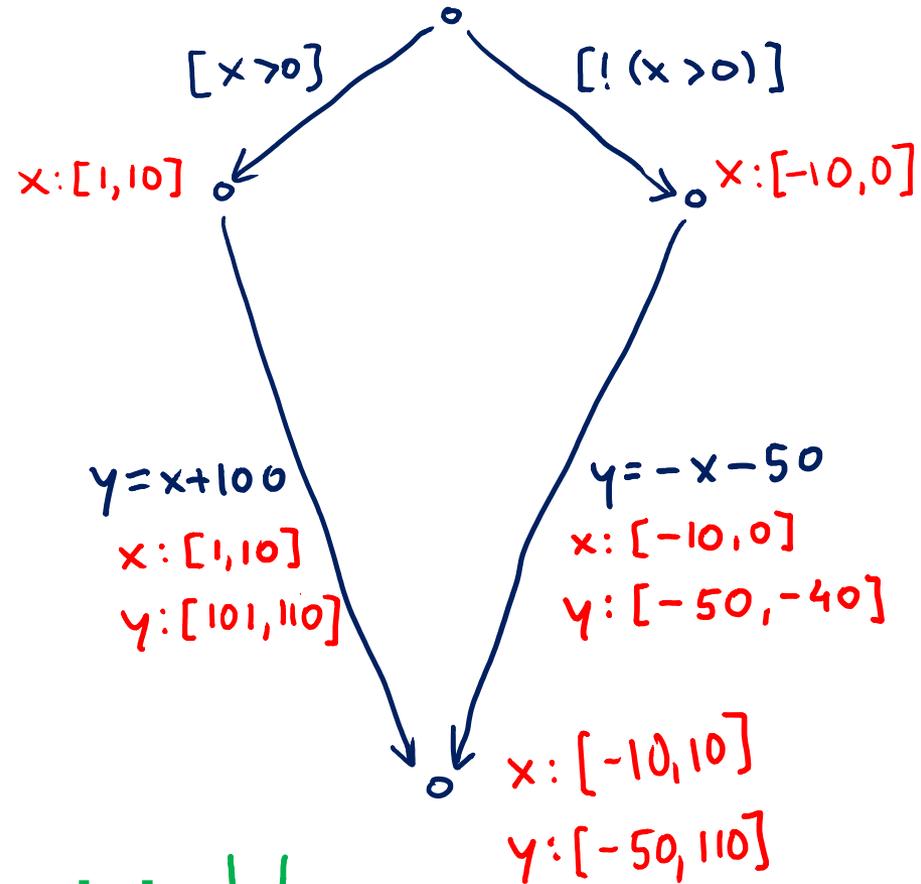
$x:$

$y:$

}

$x:$

$y:$



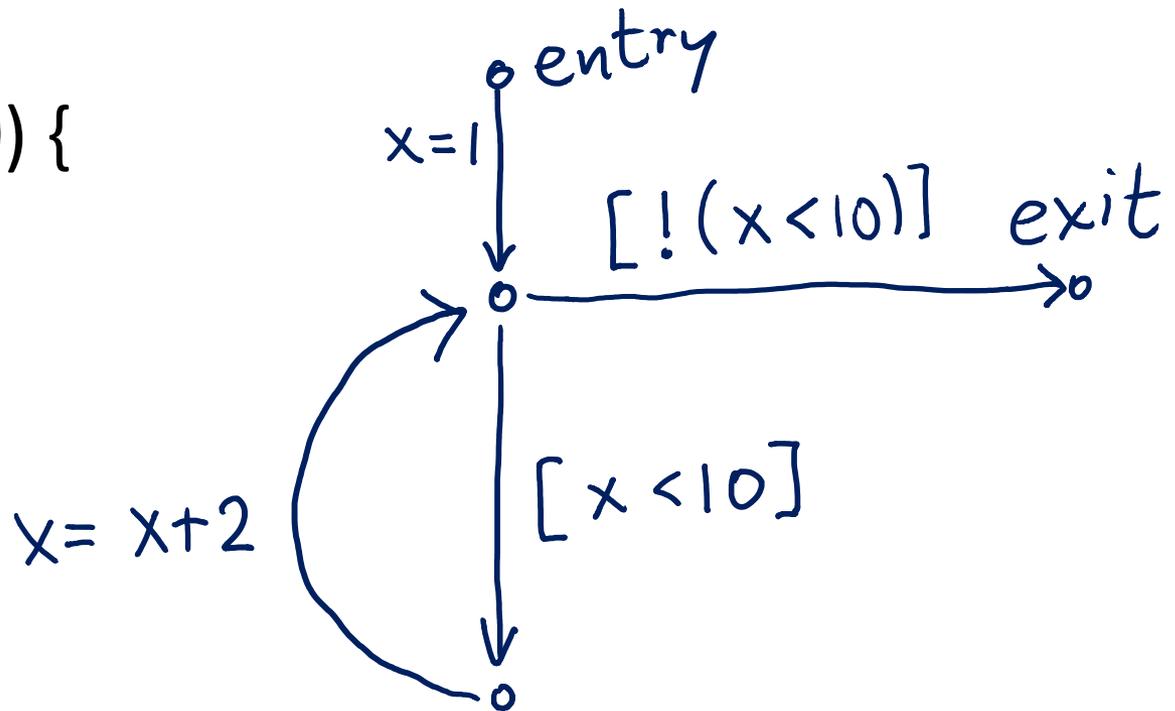
# Handling Loops: Iterate Until Stabilizes

$x = 1$

while ( $x < 10$ ) {

$x = x + 2$

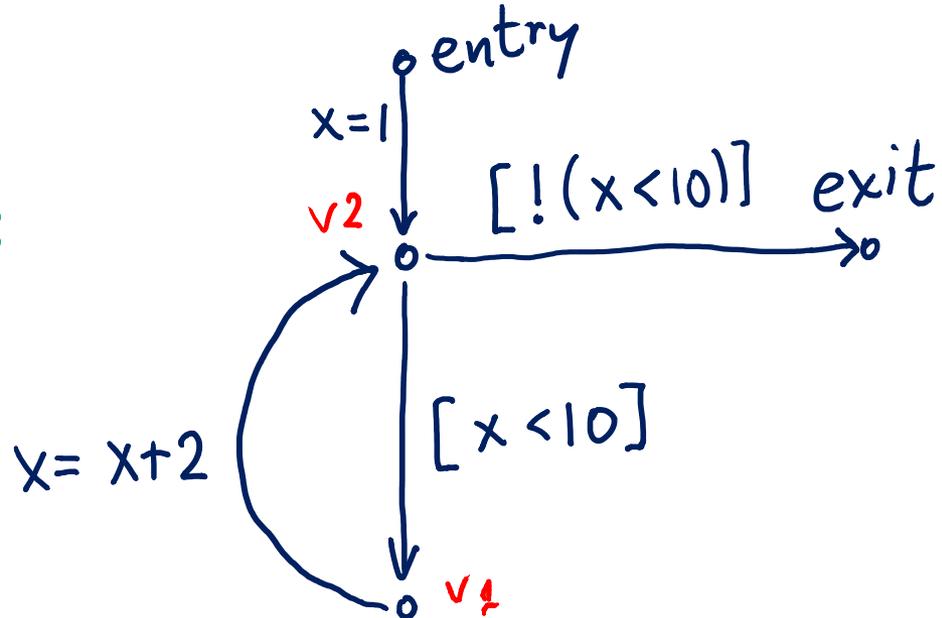
}



# Analysis Algorithm

```
var facts : Map[Node,Domain] = Map.withDefault(empty)
facts(entry) = initialValues
while (there was change)
  pick edge (v1,statmt,v2) from CFG
  such that facts(v1) has changed
  facts(v2)=facts(v2) join transferFun(statmt, facts(v1))
}
```

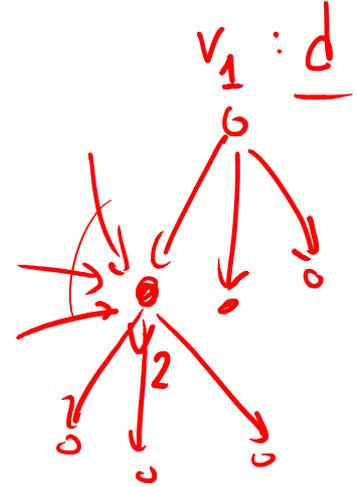
Order does not matter for the end result, as long as we do not permanently neglect any edge whose source was changed.



```

var facts : Map[Node,Domain] = Map.withDefault(empty)
var worklist : Queue[Node] = empty
def assign(v1:Node,d:Domain) = if (facts(v1)!=d) {
  facts(v1)=d
  for (stmt,v2) <- outEdges(v1) { worklist.add(v2) }
}
assign(entry, initialValues)
while (!worklist.isEmpty) {
  var v2 = worklist.getAndRemoveFirst
  update = facts(v2)
  for (v1,stmt) <- inEdges(v2)
    { update = update join transferFun(facts(v1),stmt) }
  assign(v2, update)
}

```



Work List Version

# Exercise: Run range analysis, prove that **error** is unreachable

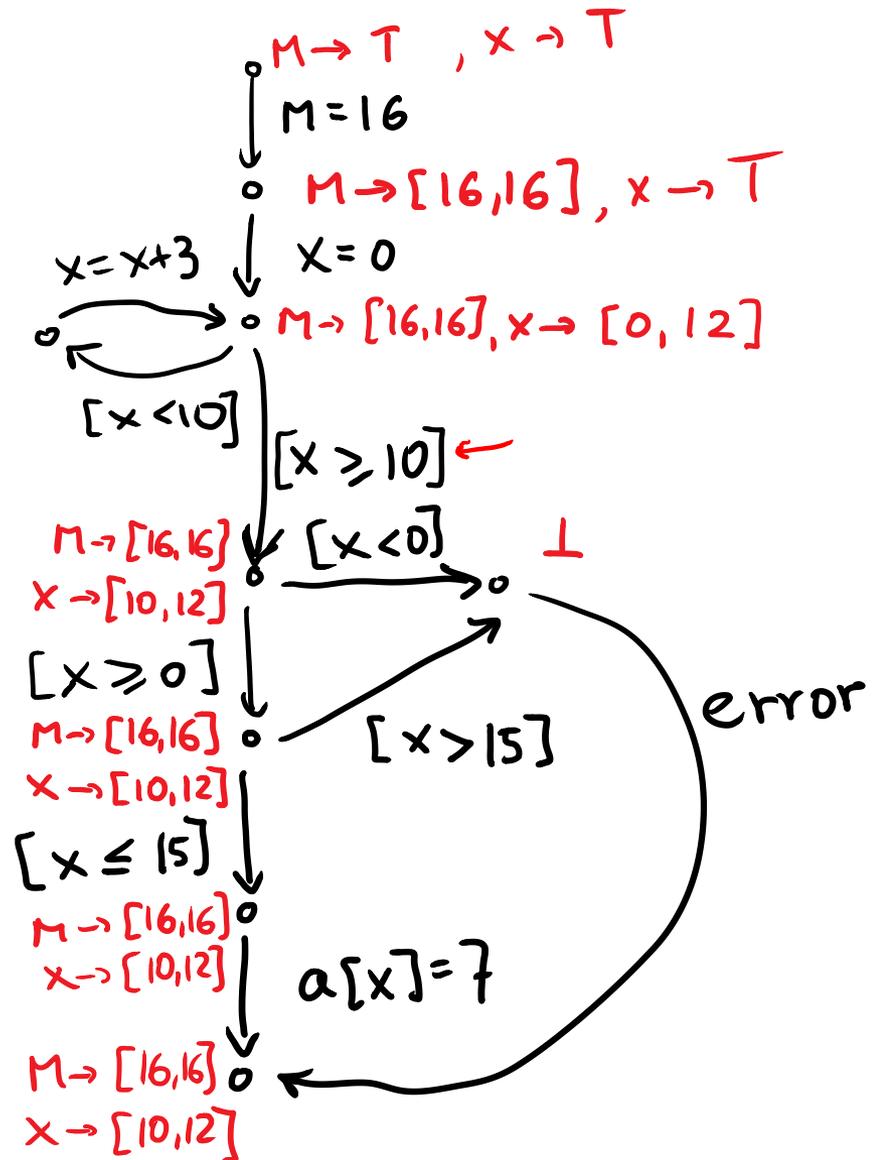
```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
  checks array accesses
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
  error;
}
```

# Range analysis results

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
  error;
}
```

checks array accesses

$M \rightarrow [16, 16]$   
 $x \rightarrow [0, 9]$

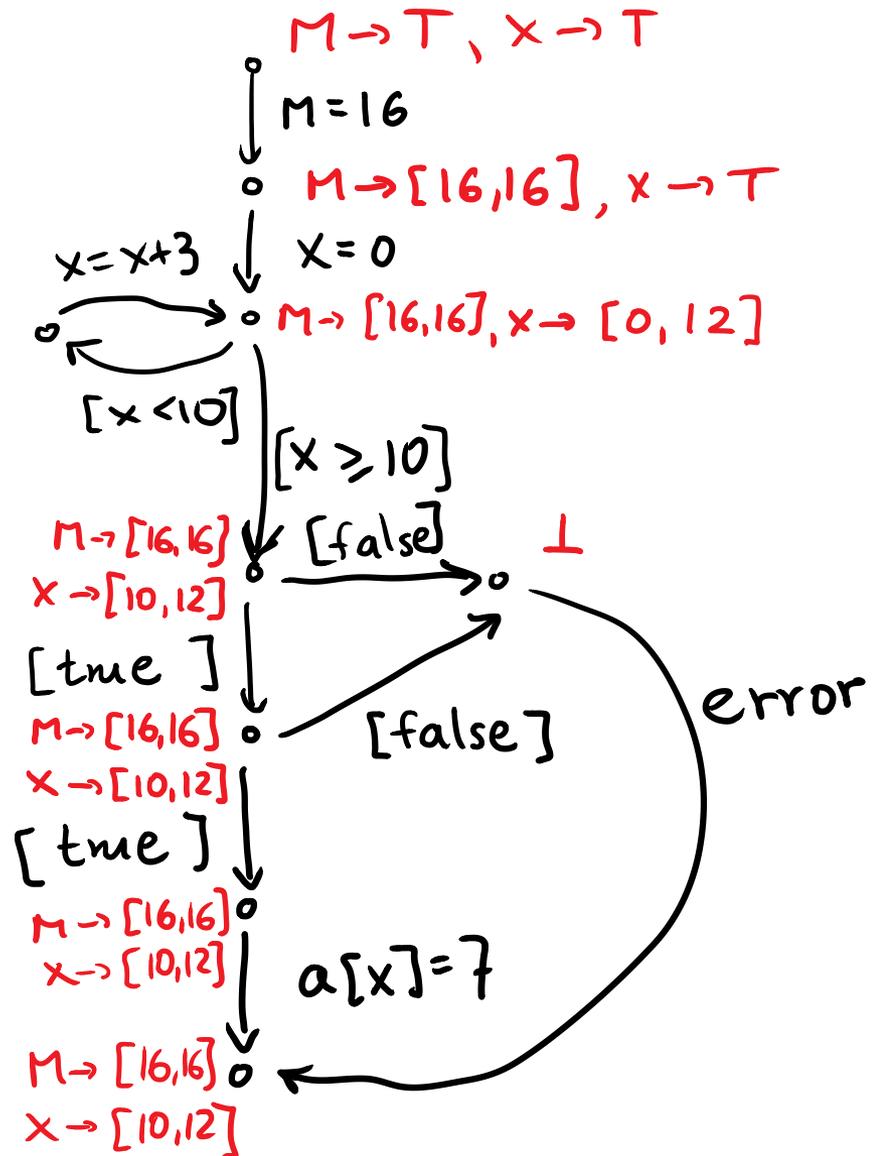


# Simplified Conditions

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
  error;
}
```

checks array accesses

$M \rightarrow [16, 16]$   
 $x \rightarrow [0, 9]$

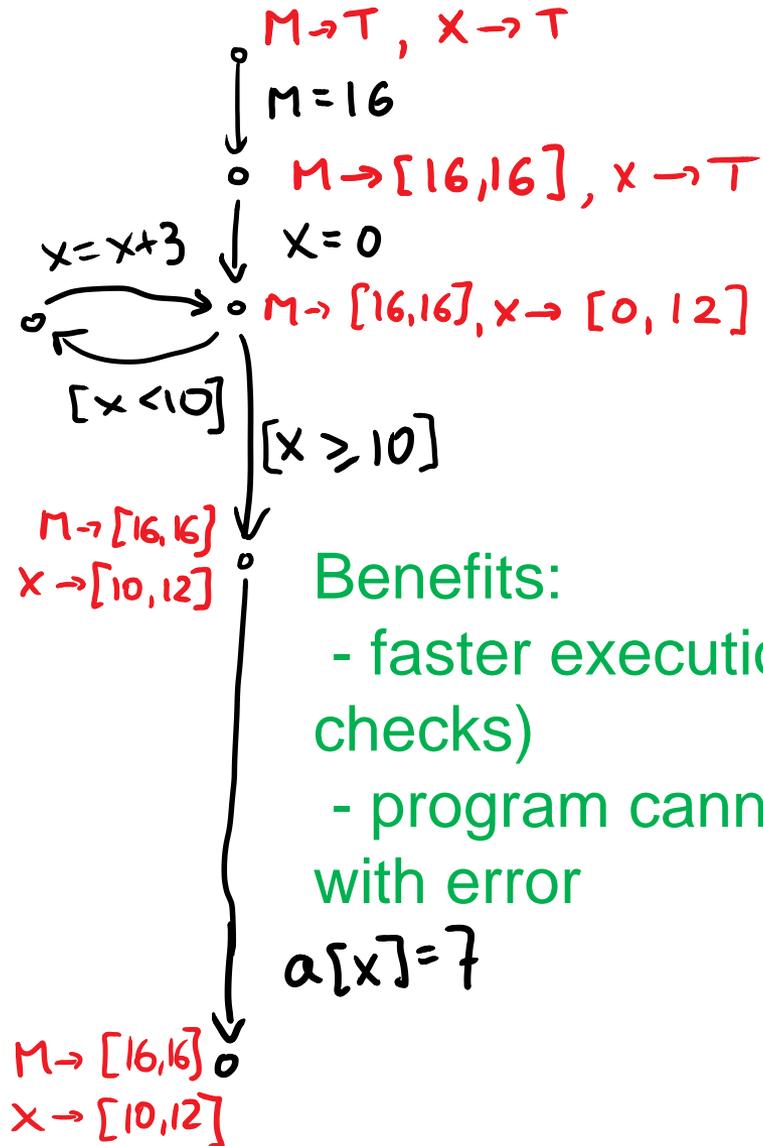


# Remove Trivial Edges, Unreachable Nodes

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
  error;
}
```

checks array accesses

$M \rightarrow [16, 16]$   
 $x \rightarrow [0, 9]$



Benefits:

- faster execution (no checks)
- program cannot crash with error

$a[x] = 7$