Single Static Assignment and Unboxing

Compiler Temporaries

Code Generation

- $e_1 + e_2$: Store e_1 while computing e_2.

Common Sub-Expression Elimination

- Store sub-expression for reuse later.

Loop Invariants

- Compute and store expression outside of loop.

Storage Locations

Memory

- **Pro**: plentiful.
- **Con**: slow.
- **Con**: must update indices (or waste space) after code elimination.

Registers

- **Pro**: very fast.
- **Con**: only 8 (16) available.
- **Con**: reuse limits code motion.

Storage Locations

Memory

- **Pro**: plentiful.
- **Con**: slow.
- **Con**: must update indices (or waste space) after code elimination.

Registers

- **Pro**: very fast.
- **Con**: only 8 (16) available.
- **Con**: reuse limits code motion.

We want it all: ***no reuse***, ***no indexing***, and ***plentiful*** storage.

- Also, fast would be nice.

Static Single Assignment (SSA)

Add “temporary locations” to intermediate representation.

- ***Infinite number*** (effectively) of these.
- Each location assigned ***exactly once***.
- Implement these as registers later (register allocation).

3-Address IR: x becomes $x_1, x_2, \ldots$

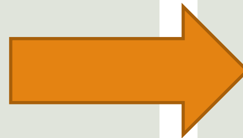
For Cool ASM IR: add registers `t0`, `t1`, …

SSA and a Basic Block

```
mul r1 <- r1 r1
li r2 <- 1
add r0 <- r0 r2
add r0 <- r0 r1
```

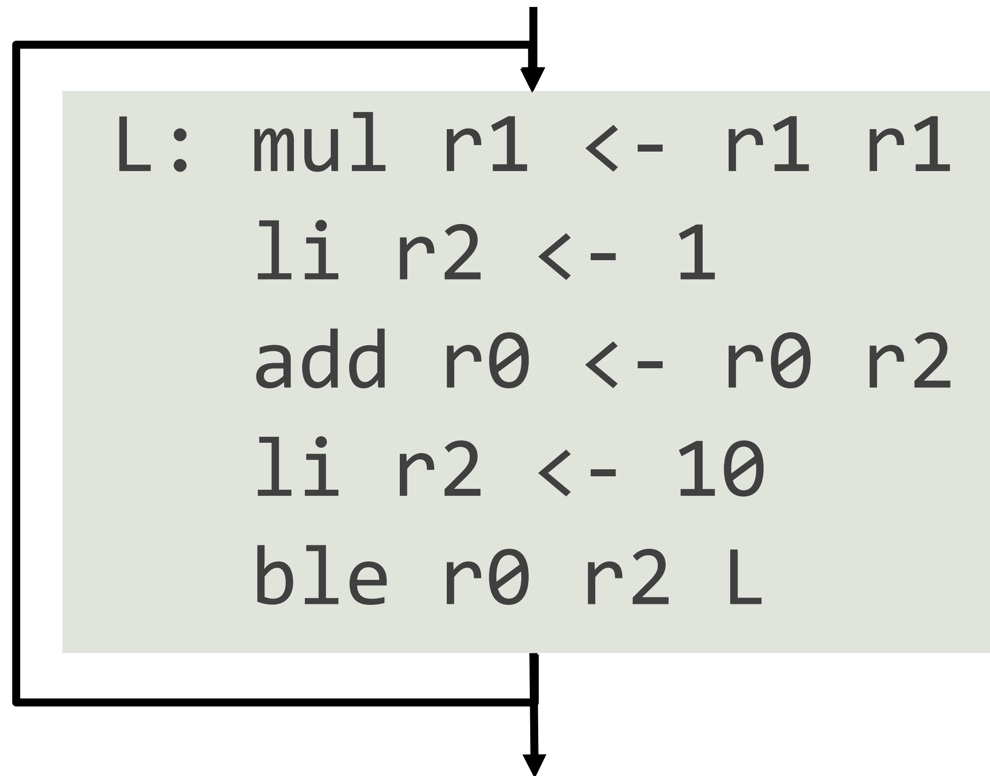
SSA and a Basic Block

```
mul r1 <- r1 r1
li r2 <- 1
add r0 <- r0 r2
add r0 <- r0 r1
```

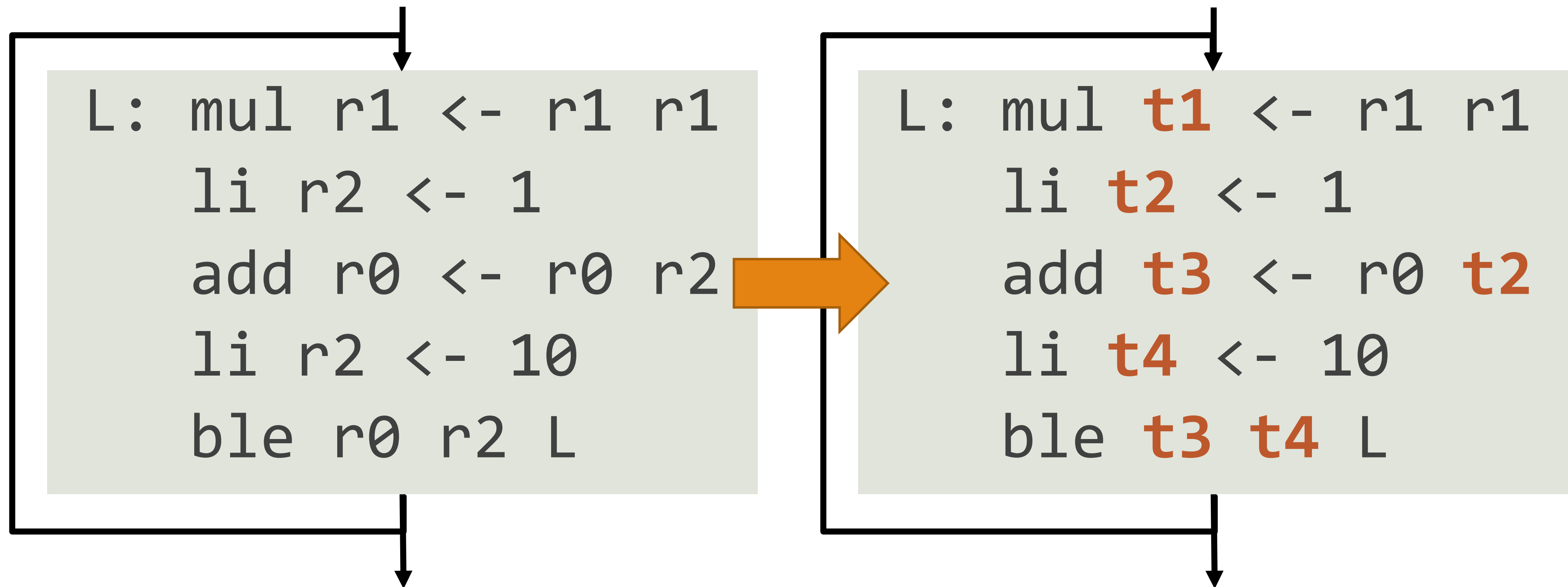


```
mul t1 <- r1 r1
li t2 <- 1
add t3 <- r0 t2
add t4 <- t3 t1
```

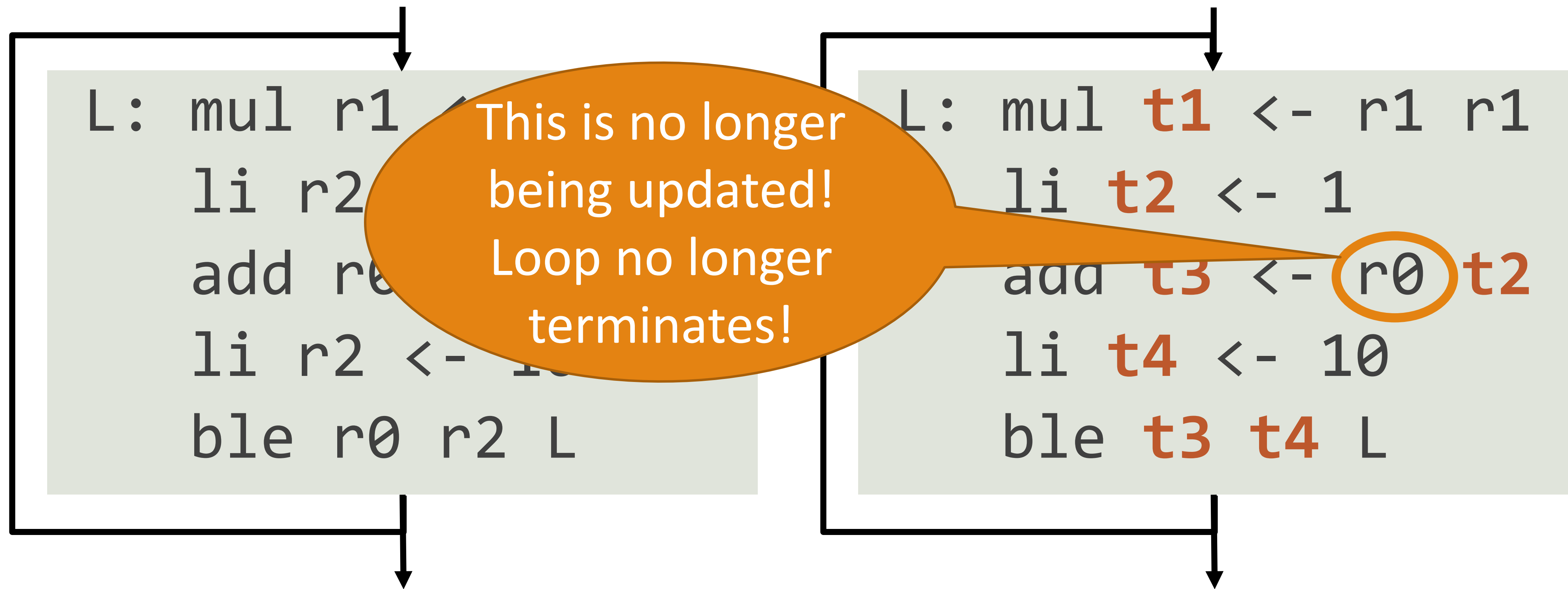
SSA and a Control Flow Graph (CFG)



SSA and a Control Flow Graph (CFG)



SSA and a Control Flow Graph (CFG)

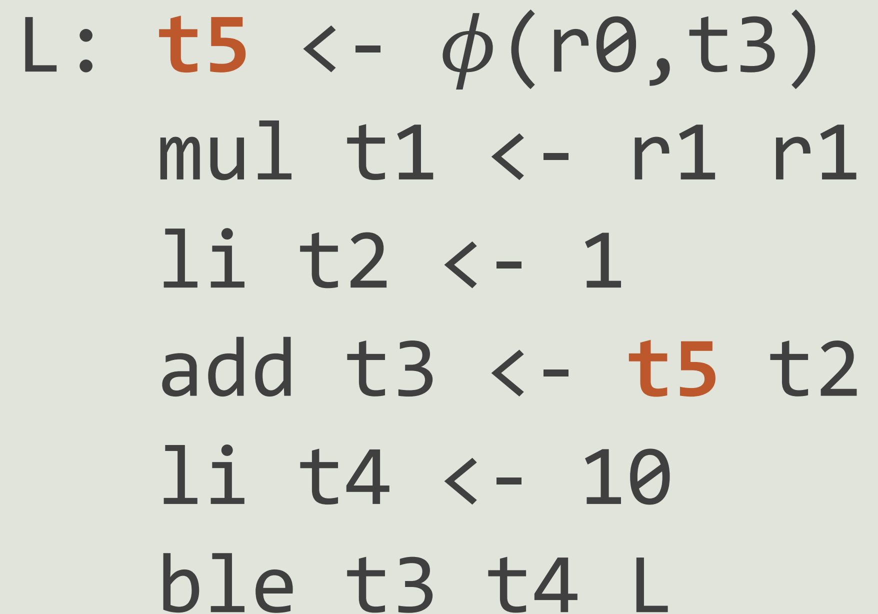


ϕ-Functions

In general, ***one logical value*** may reach a basic block along ***more than one path***.

Insert a ***ϕ-function*** to "merge" those values.

Not 3-address: one argument per incoming path.



Converting CFG to SSA Form

General Algorithm:

1. Find ***dominators*** for every CFG node.
   - Every path to node *N* goes through its dominators.
2. Compute ***dominance frontiers*** for every CFG node.
   - Set of nodes just barely ***not*** dominated by N.
3. Place ϕ-functions at frontiers.
4. Rename assignments and subsequent uses.

Converting CFG to SSA Form

General Algorithm:

- Required if your language uses gotos.
- Algorithmically efficient (data-flow, plus two tree walks).

Dominators:

- Also useful for identifying natural loops (gotos again).

Generating SSA Form Directly

```
program  ::= [[class;]]+
class    ::= class TYPE [inherits TYPE] { [[feature;]]* }
feature  ::= ID( [ formal [[, formal]]* ] ) : TYPE { expr }
           | ID : TYPE [ <- expr ]
formal   ::= ID : TYPE
expr     ::= ID <- expr
           | expr[@TYPE].ID( [ expr [[, expr]]* ] )
           | ID( [ expr [[, expr]]* ] )
           | if expr then expr else expr fi
           | while expr loop expr pool
           | { [[expr;]]+ }
           | let ID : TYPE [ <- expr ] [[, ID : TYPE [ <- expr ]]]* in expr
           | case expr of [[ID : TYPE => expr;]]+ esac
           | new TYPE
           | isvoid expr
           | expr + expr
           | expr − expr
           | expr ∗ expr
           | expr / expr
           | ˜expr
           | expr < expr
           | expr <= expr
           | expr = expr
           | not expr
           | (expr)
           | ID
           | integer
           | string
           | true
           | false
```

Generating SSA Form Directly

```
program ::= [[class;]]+
class   ::= class TYPE [inherits TYPE] { [[feature;]]* }
feature ::= ID( [ formal [[, formal]]* ] ) : TYPE { expr }
         |  ID : TYPE [ <- expr ]
formal  ::= ID : TYPE
expr    ::= ID <- expr
         |  expr[@TYPE].ID( [ expr [[, expr]]* ] )
         |  ID( [ expr [[, expr]]* ] )
         |  if expr then expr else expr f
         |  while expr loop expr p
         |  { [[expr;]]+ }
         |  let ID : TYPE [ <- expr ] [[, ID : TYPE [ <- expr ]]* in expr
         |  case expr of [[ID : TY E => expr; ]]+esac
         |  new TYPE
         |  isvoid expr
         |  expr + expr
         |  expr − expr
         |  expr ∗ expr
         |  expr / expr
         |  ˜expr
         |  expr < expr
         |  expr <= expr
         |  expr = expr
         |  not expr
         |  (expr)
         |  ID
         |  integer
         |  string
         |  true
         |  false
```

Most expressions: store result in temporary.

Generating SSA Form Directly

```
program ::= [[class;]]+
  class ::= class TYPE [inherits TYPE] { [[feature;]]* }
feature ::= ID( [ formal [[, formal]]* ] ) : TYPE { expr }
          | ID : TYPE [ <- expr ]
 formal ::= ID : TYPE
   expr ::= ID <- expr
          | expr[@TYPE].ID( [ expr [[, expr]]* ] )
          | ID( [ expr [[, expr]]* ] )
          | if expr then expr else expr fi
          | while expr loop expr pool
          | { [[expr;]]+ }
          | let ID : TYPE [ <- expr ] [[, ID : TYPE [ <- ex
          | case expr of [[ID : TYPE => expr;]]+ esac
          | new TYPE
          | isvoid expr
          | expr + expr
          | expr − expr
          | expr ∗ expr
          | expr / expr
          | ~expr
          | expr < expr
          | <= expr
          | true
          | false
```

Control-flow:
insert ϕ-function
for assignments
in body.

SSA Code Generation of Control-Flow

```
(*local x*)
if b then
  x <- 1
else
  x <- 2
fi
```

SSA Code Generation of Control-Flow

```
(*local x*)
if b then
  x <- 1
else
  x <- 2
fi
```

```
brz t1 L1
; true branch
call Int..new
mov t2 <- r1
li t3 <- 1
st t2[2] <- t3
jmp L2
```

```
L1:
; false branch
call Int..new
mov t4 <- r1
li t5 <- 2
st t4[2] <- t5
L2:
t6 <- φ(t2,t4)
```

SSA Code Generation of Control-Flow

`if` and `case` expressions:

- Also insert ϕ-functions for the expression value.
- `x <- if b then new Int else new String fi`

`while` expressions:

- Insert ϕ-function at top for variables used then modified.
- `while x < 10 loop x <- x + 1 pool`

Auto-Unboxing

Boxed Types

All Cool values are objects.

- Fewer special cases in language.
  - Handle for any value of any type can be stored in a (integer) register.
  - No need for void methods: everything satisfies type Object.
- Simpler generic data structures and methods.
  - Java7 has ***9*** `System.out.print()` methods.
  - `java.util.List` holds `Integers`, but not `ints`.

Unboxed Types

Many uses of Int (or Bool or String) do not use objects.

- Paying for flexibility we do not use!

$x + y$	**Instructions**
Unboxed	1 ALU
Boxed	1 ALU + 3 Memory + 1 constructor call*

We want to work with raw values wherever possible.

Unboxing: Intuition

Define type UnboxedInt

- Integer literals have type UnboxedInt
- Instruction `boxi` : UnboxedInt → Int
- Instruction `unboxi` : Int → UnboxedInt
- Arithmetic operators apply to UnboxedInt

```
x + y ⇒ box(unbox(x) + unbox(y))
```

$(a + b) + c$

(a + b) + c

```
unboxi t1 <- r2
unboxi t2 <- r3
add t3 <- t1 t2
boxi t4 <- t3
unboxi t5 <- t4
unboxi t6 <- r4
add t7 <- t5 t6
boxi t8 <- r7
```

(a + b) + c

```
unboxi t1 <- r2
unboxi t2 <- r3
add t3 <- t1 t2
boxi t4 <- t3
unboxi t5 <- t4
unboxi t6 <- r4
add t7 <- t5 t6
boxi t8 <- r7
```

Peephole optimization: Can save constructor call, plus 2 memory ops.

(a + b) + c

```
unboxi t1 <- r2
unboxi t2 <- r3
add t3 <- t1 t2
mov t5 <- t3
unboxi t6 <- r4
add t7 <- t5 t6
boxi t8 <- r7
```

Was that safe?

Something a Little More Complicated

```
let i : Int <- 1 in
while i < 10 loop {
  x <- x * 2;
  i <- i + 1;
} pool;
x
```

Something a Little More Complicated

```
li t1 <- 1
boxi t2 <- t1
L1:
unboxi t3 <- t2
li t4 <- 10
boxi t5 <- t4
unboxi t6 <- t5
ble t6 t3 L2
```

```
unboxi t7 <- r0
li t8 <- 2
boxi t9 <- t8
unboxi t10 <- t9
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
boxi t15 <- t14
unboxi t16 <- t15
```

```
add t17 <- t13 t16
boxi t18 <- t17
jmp L1
L2:
mov r1 <- t12
```

Something a Little More Complicated

```
li t1 <- 1
boxi t2 <- t1
L1:
t19 <- ϕ(t2, t18)
t20 <- ϕ(r0, t12)
unboxi t3 <- t19
li t4 <- 10
boxi t5 <- t4
unboxi t6 <- t5
ble t6 t3 L2
```

```
unboxi t7 <- t20
li t8 <- 2
boxi t9 <- t8
unboxi t10 <- t9
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
boxi t15 <- t14
unboxi t16 <- t15
```

```
add t17 <- t13 t16
boxi t18 <- t17
jmp L1
L2:
mov r1 <- t12
```

Something a Little More Complicated

```
li t1 <- 1
boxi t2 <- t1
L1:
t19 <- ϕ(t2, t18)
t20 <- ϕ(r0, t12)
unboxi t3 <- t19
li t4 <- 10
boxi t5 <- t4
unboxi t6 <- t5
ble t6 t3 L2
```

```
unboxi t7 <- t20
li t8 <- 2
boxi t9 <- t8
unboxi t10 <- t9
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
boxi t15 <- t14
unboxi t16 <- t15
```

```
add t17 <- t13 t16
boxi t18 <- t17
jmp L1
L2:
mov r1 <- t12
```

Something a Little More Complicated

```
li t1 <- 1
boxi t2 <- t1
L1:
t19 <- ϕ(t2, t18)
t20 <- ϕ(r0, t12)
unboxi t3 <- t19
li t4 <- 10
mov t6 <- t4
ble t6 t3 L2
```

```
unboxi t7 <- t20
li t8 <- 2
mov t10 <- t8
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
mov t16 <- t14
```

```
add t17 <- t13 t16
boxi t18 <- t17
jmp L1
L2:
mov r1 <- t12
```

A Data-Flow for Unboxing

Direction: Backward

Meet operator: Diamond

Value	Meaning
$\top$	Unknown
used	Value is used (arithmetic or method call)
converted	Value is eventually converted.
$\perp$	Value is used and converted.

Something a Little More Complicated

```
li t1 <- 1
boxi t2 <- t1
L1:
t19 <- ϕ(t2, t18)
t20 <- ϕ(r0, t12)
unboxi t3 <- t19
li t4 <- 10
mov t6 <- t4
ble t6 t3 L2
```

```
unboxi t7 <- t20
li t8 <- 2
mov t10 <- t8
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
mov t16 <- t14
```

```
add t17 <- t13 t16
boxi t18 <- t17
jmp L1
L2:
mov r1 <- t12
```

Something a Little More Complicated

```
li t1 <- 1
mov t2 <- t1
L1:
t19 <- ϕ(t2, t18)
t20 <- ϕ(r0, t12)
mov t3 <- t19
li t4 <- 10
mov t6 <- t4
ble t6 t3 L2
```

```
unboxi t7 <- t20
li t8 <- 2
mov t10 <- t8
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
mov t16 <- t14
```

```
add t17 <- t13 t16
mov t18 <- t17
jmp L1
L2:
mov r1 <- t12
```

Something a Little More Complicated

```
li t1 <- 1
mov t2 <- t1
L1:
t19 <- ϕ(t2, t18)
t20 <- ϕ(r0, t12)
mov t3 <- t19
li t4 <- 10
mov t6 <- t4
ble t6 t3 L2
```

```
unboxi t7 <- t20
li t8 <- 2
mov t10 <- t8
mul t11 <- t7 t10
boxi t12 <- t11
unboxi t13 <- t2
li t14 <- 1
mov t16 <- t14
```

```
add t17 <- t13 t16
mov t18 <- t17
jmp L1
L2:
mov r1 <- t12
```

Unboxing Summary

1. Insert type-casts to represent boxing and unboxing.
2. Use data-flow analysis to identify wasteful boxing.
   - ***Argument values*** must be boxed (for now).
   - ***Return values*** must be boxed (for now).
   - ***Self objects*** must be boxed.

Dominance

X ***dominates*** Y ($X \geq Y$)

- If ***every*** path to Y goes through X.
- Note: $X \geq X$

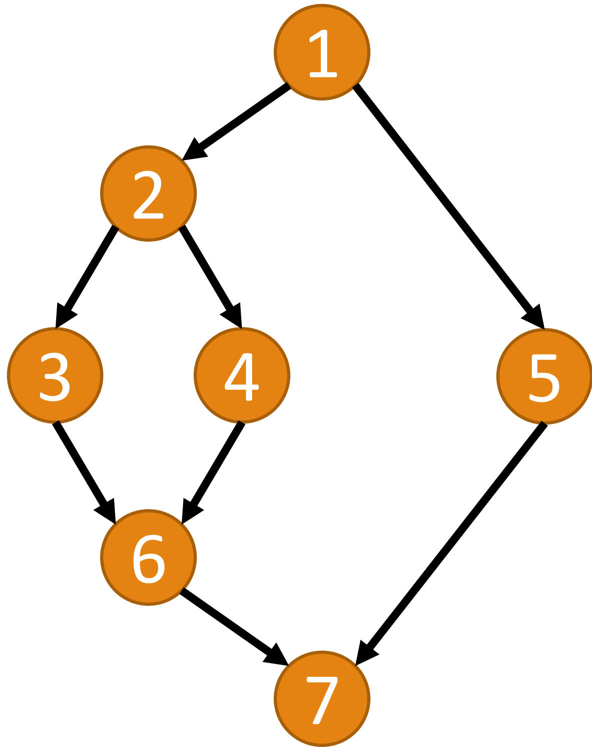
X ***strictly dominates*** Y ($X > Y$)

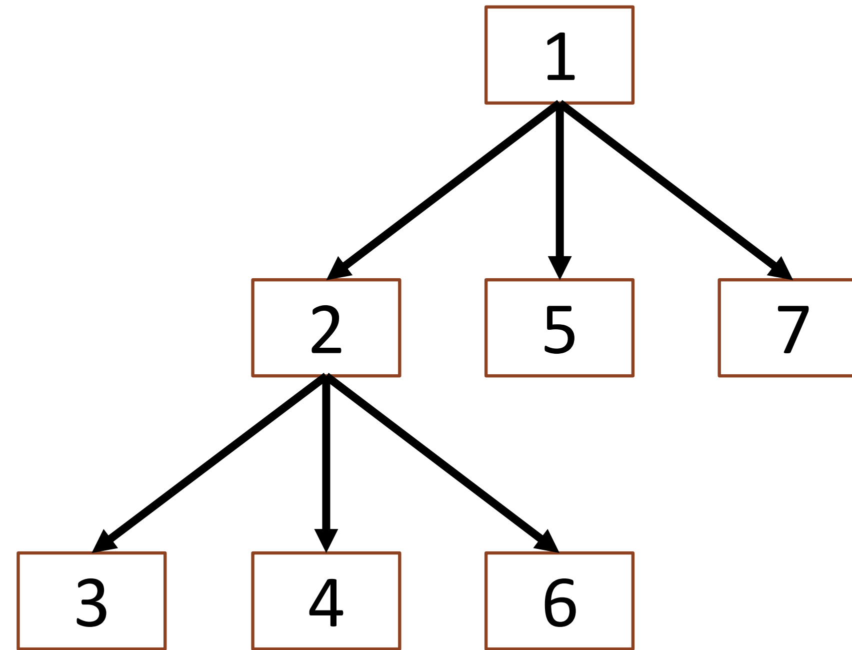
- If $X \geq Y$, but $X \neq Y$.

Find dominators with data-flow algorithm

- (Dragon Book, p658).

Dominance Trees





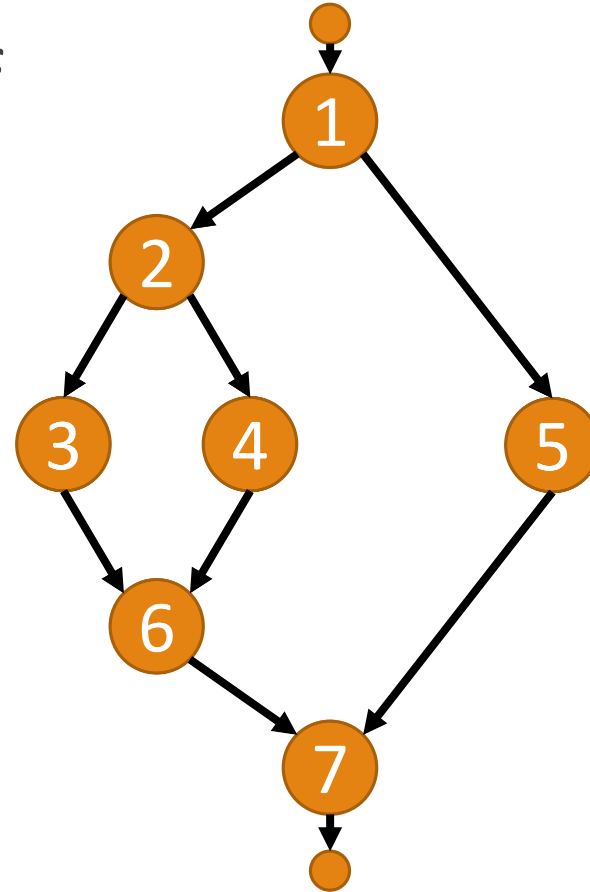
Dominance and ϕ-Functions

Place ϕ-function at node *N* if

- 2 non-empty CFG paths define *v*,
- And both paths meet at *N*.

Note: ϕ-function defines *v*.

I.e., place ϕ-function along dominance frontier.



DF(1) = {1}

DF(2) = {7}

DF(3) = {6}

DF(4) = {6}

DF(5) = {1,7}

DF(6) = {7}

DF(7) = {}