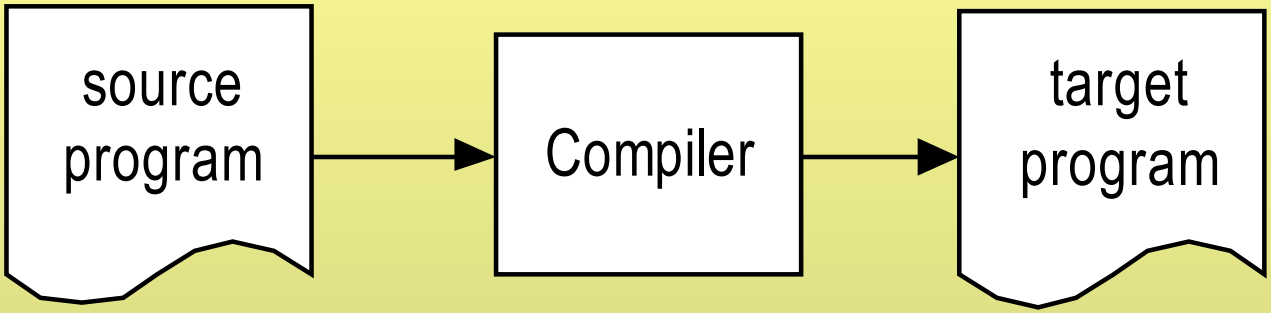
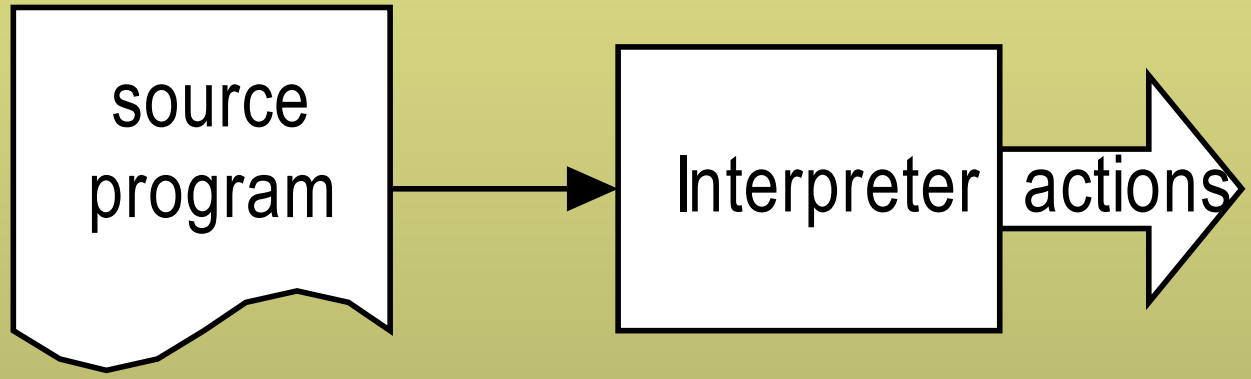
Compiler Optimization

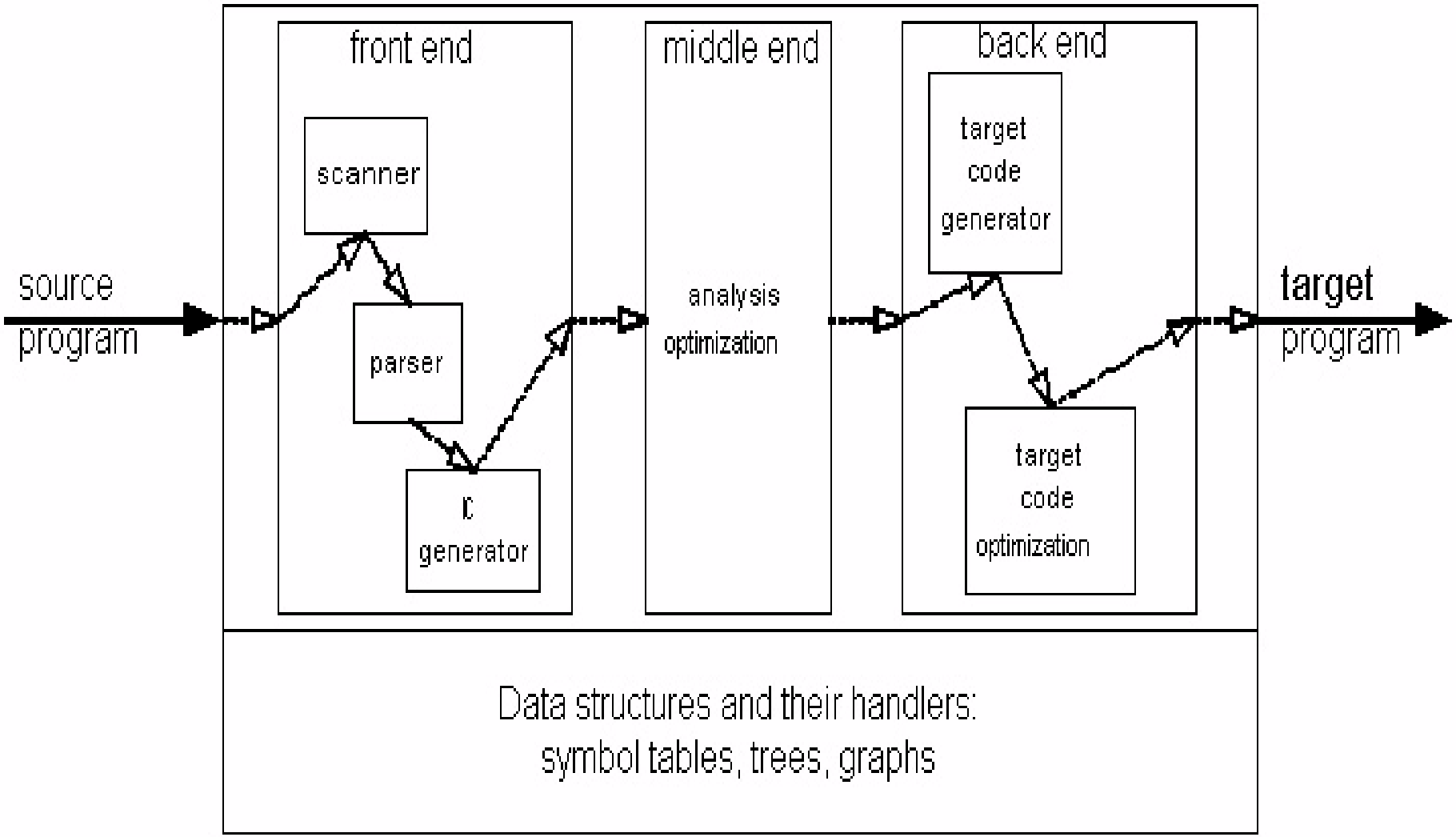
F. Franek

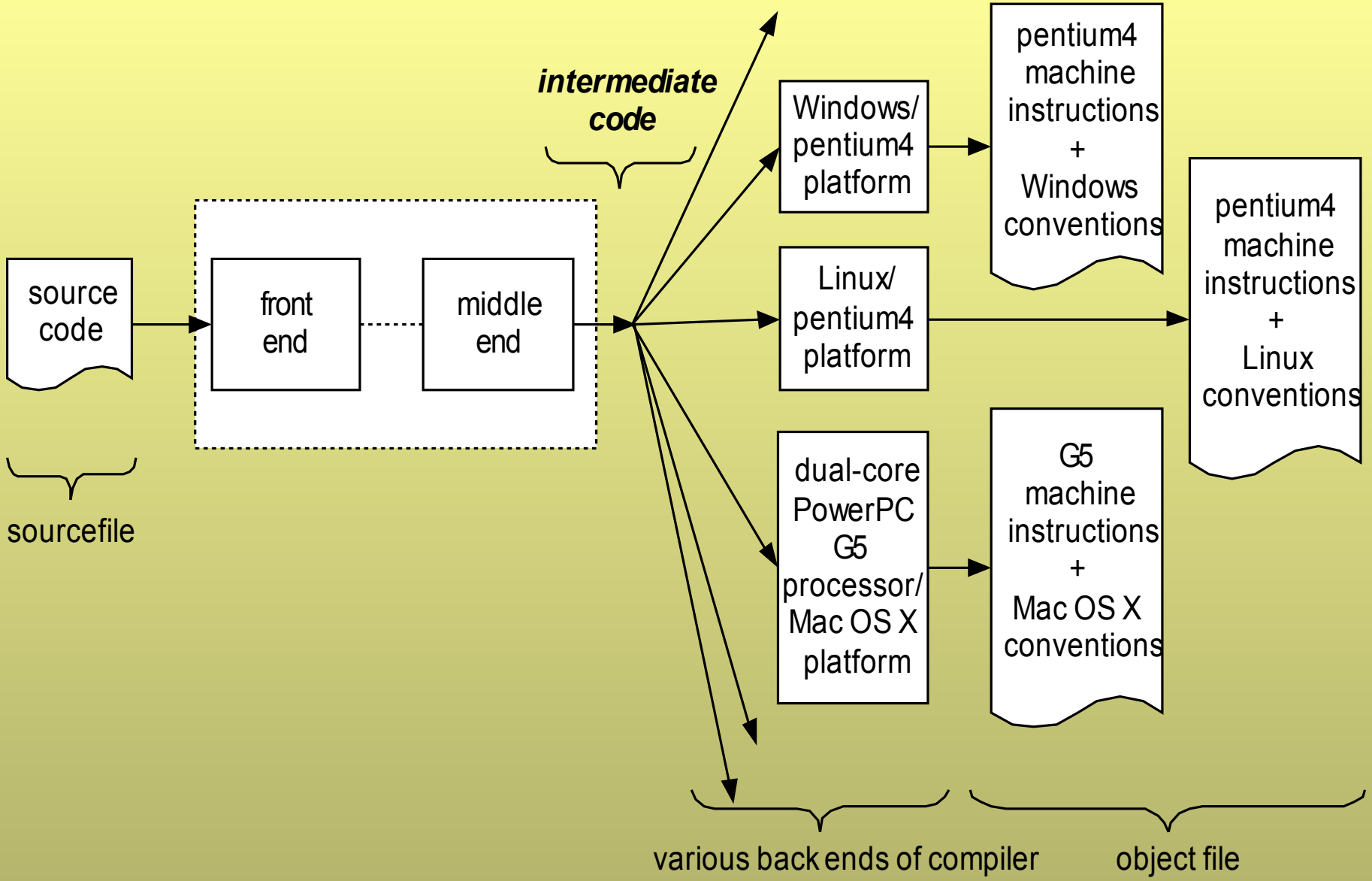
Dept. of Comp. and Software
McMaster University
Hamilton, Ontario, Canada

- **Compilation phases**
- **Intermediate code**
- **Control flow analysis**
- **Data flow analysis**
- **Various optimization techniques:**
  Peephole Optimization, Dead-code Elimination Unreachable-code Elimination, Straightening, If Simplification, Value Numbering, Copy Propagation, (SCCP) Sparse Conditional Constant Propagation, Common Subexpression Elimination, Loop-invariant Code Hoisting.


source program
Compiler
target program


source program
Interpreter
actions


front end
middle end
back end
scanner
parser
r
generator
analysis
optimization
target
code
generator
target
code
optimization
source
program
target
program
Data structures and their handlers:
symbol tables, trees, graphs


intermediate code
source code
front end
middle end
sourcefile
Windows/ pentium4 platform
Linux/ pentium4 platform
dual-core PowerPC G5 processor/ Mac OS X platform
pentium4 machine instructions + Windows conventions
pentium4 machine instructions + Linux conventions
G5 machine instructions + Mac OS X conventions
various back ends of compiler
object file

A good intermediate code (IC) should have the following qualities:

- **It should be relatively simple to generate from a syntax tree, otherwise its introduction would be too costly and error prone.**
- **It should be relatively simple to generate a target code from the IC for the same reasons.**
- **The semantics of the IC must be simple, clear, and unambiguous, so the optimization of the IC can be clearly specified and implemented.**

- **The syntax and semantics of the IC must be significantly less complex than that of the source code.**

So, what IC?

The simple answer is that IC is whatever a compiler designers decide to use.

Traditionally, there are three main approaches:

- ***Graphical representation*** **-- usually in the form of trees or graphs (often in the form of simplified syntax trees (e.g., A.W. Appel and his compilers for Tiger).**

- ***Stack-machine code*** **-- Java bytecode is an example of such an approach.**

- ***Three-address code*** **-- rudimentary Assembly-like instructions with two operands (hence two addresses) and a place to store the result (that is the third address).**

In this talk we are using as the IC a three-address code developed for MACS.

```
class Factorial {
 shared int fact(int n) {
   int i, res;
   if (n == 0) return 1;
   for(res=1,i=2; i <= n; res=res*i,i++);
   return res;
  }
}
```

MACS source code of method `fact()`

```
ic-start
   ...
   Fact:
   sec-start
      arg n {val}{int}
      t10 = 0
      t12 = n == t10
      t13 = !t12
      if t13 goto L1
      t14 = 1
      return t14
   L1:nop
   L2:t13 = 1
```

```
L2:t13 = 1
     res = t13
     t14 = 2
     i = t14
     t15 = i <= n
     t16 = !t15
     if t16 goto L3
  L4:nop
     t17 = res * i
     res = t17
     t18 = i + 1
     i = t18
     goto L2
  L3:nop
```

```
L3:nop
      return res
   sec-end
   ...
 ic-end
```

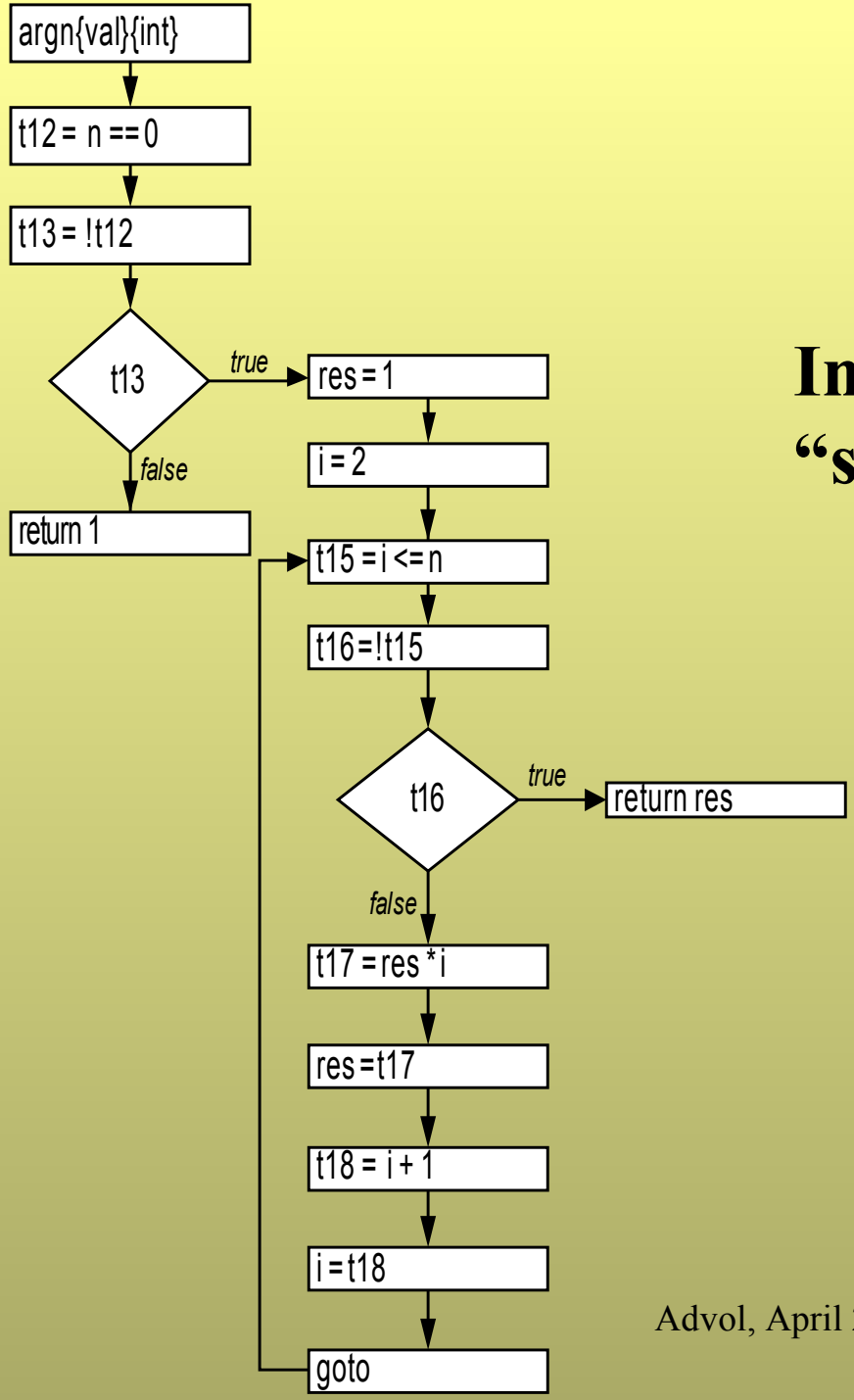
A raw MIC code of method `fact()`

```
ic-start
   ...
   fact:
   sec-start
      arg n {val}{int}
      t12 = n == 0
      t13 = !t12
      if t13 goto L2
      return 1
   L2:res = 1      (does this def. reach L3?)
      i = 2
      t15 = i <= n
      t16 = !t15
```

```
    t16 = !t15
    if t16 goto L3
L4: t17 = res * i
    res = t17
    t18 = i + 1
    i = t18
    goto L2
L3: return res
sec-end
...
ic-end
```

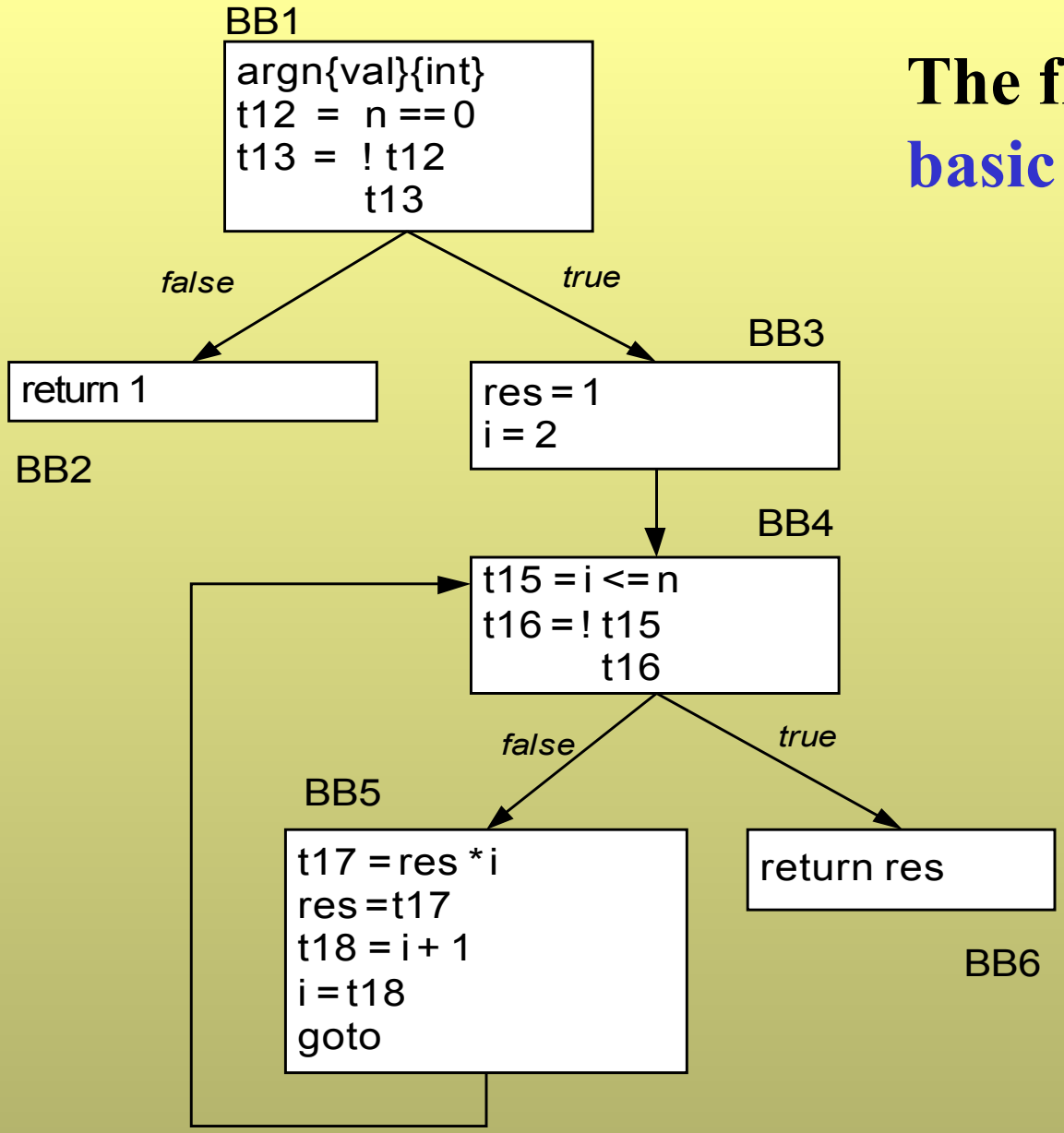
A streamlined MIC code of method `fact()`

It is hard to "see", for instance, the loop in the IC code.

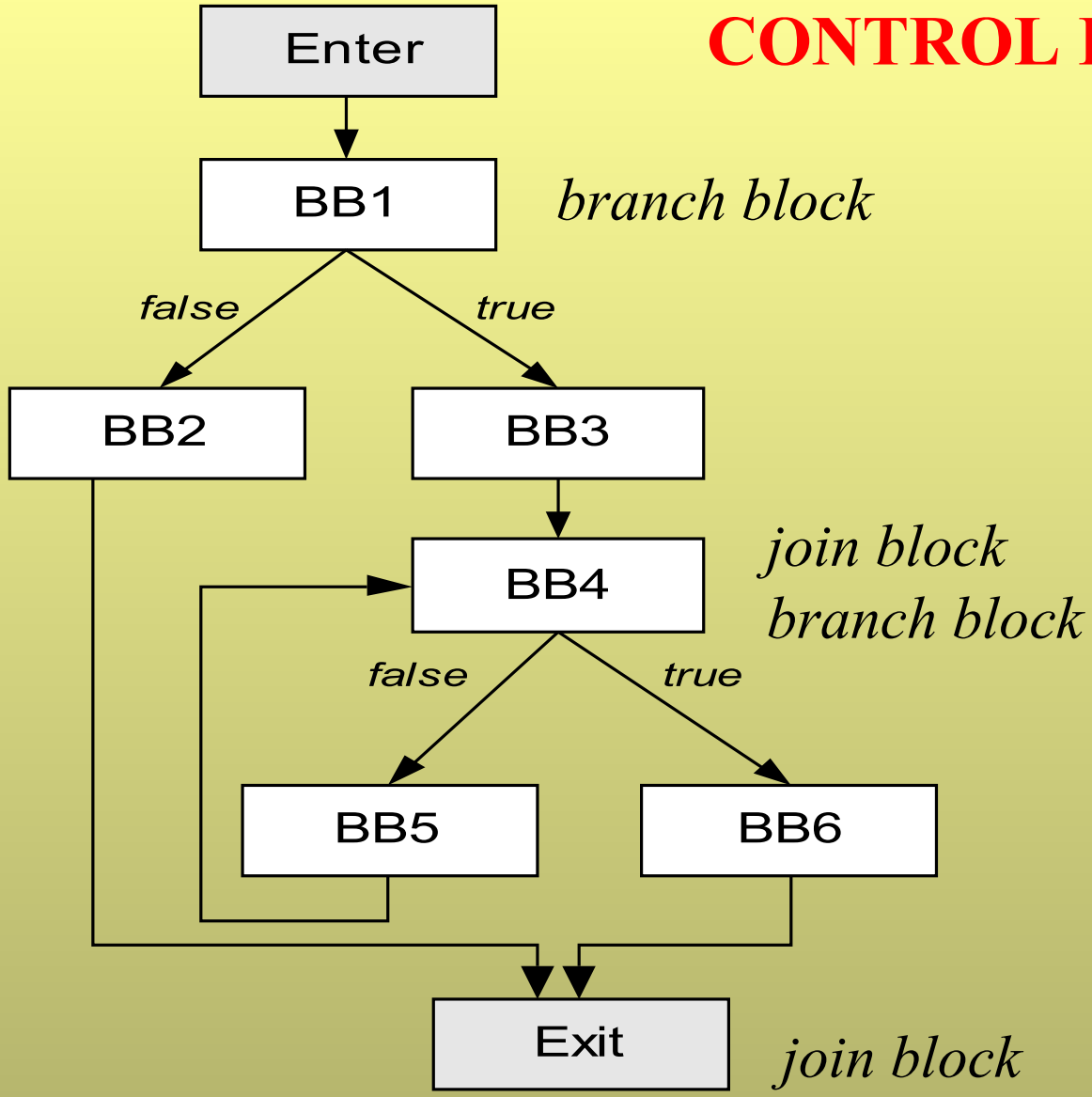


In the flowchart, we can "see" the loop again

The flowchart with basic blocks



CONTROL FLOW GRAPH

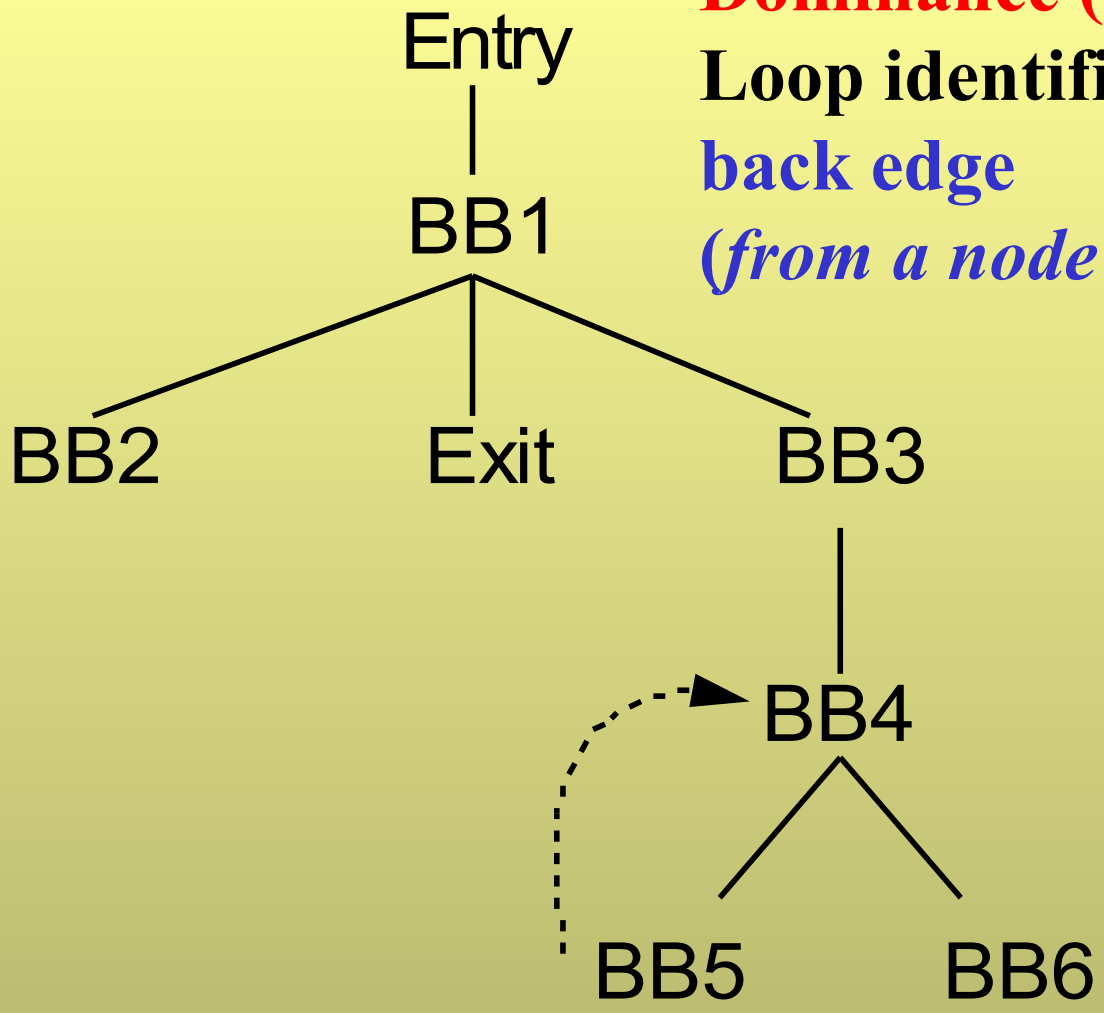


Dominance (dominator) tree

Loop identification via

back edge

(from a node to its dominator)

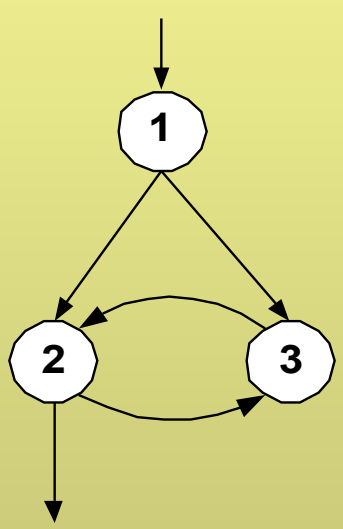


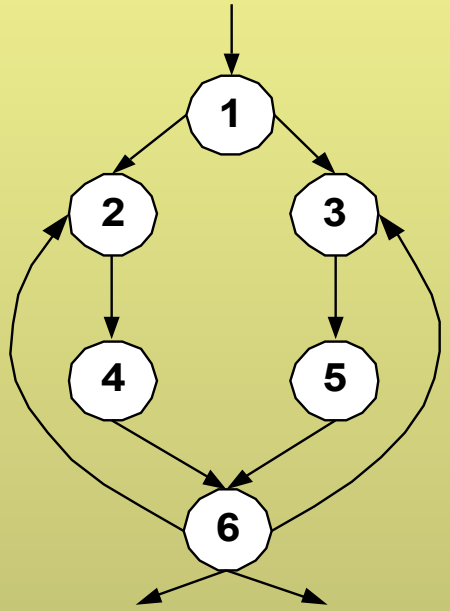
Calculations of dominators:

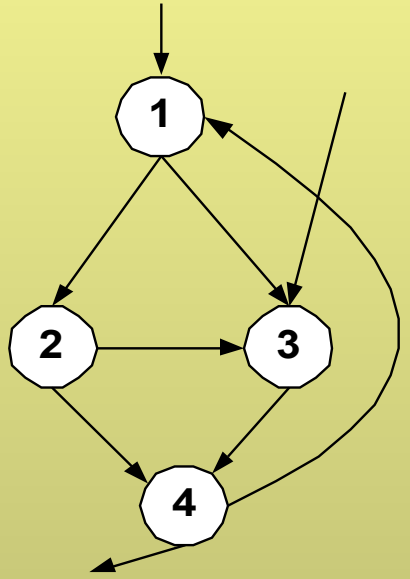
- **by a simple depth-first recursive algorithm based on the inductive definition of domination:**
  A dominates B if
  (a) A=B, or
  (b) A is a unique immediate predecessor of B, or
  (c) B has more than one immediate predecessor, and for every C immediate predecessor of B, A dominates C.

- **More efficient algorithms are due to Lengauer+Tarjan and Alstrup+Lauridsen**

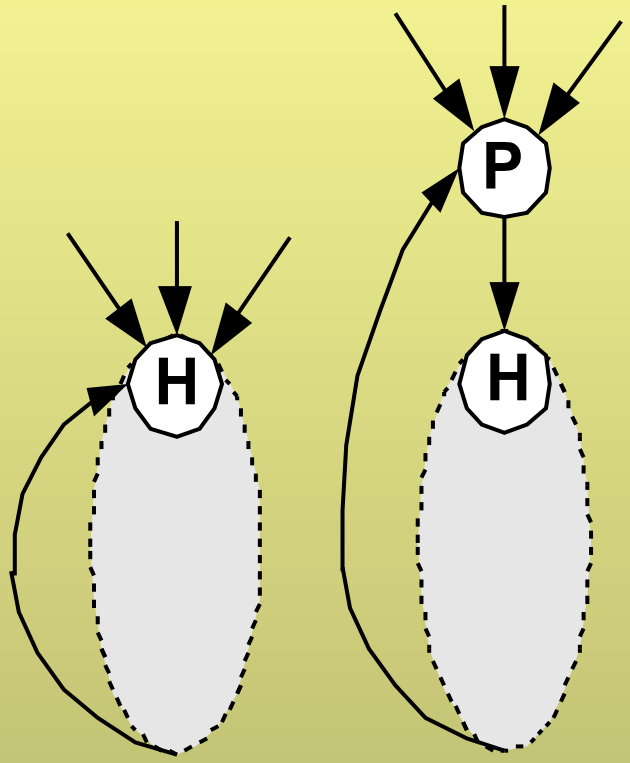
Dominators used for loop identification (to identify the back edge) and for a fast transformation to SSA.



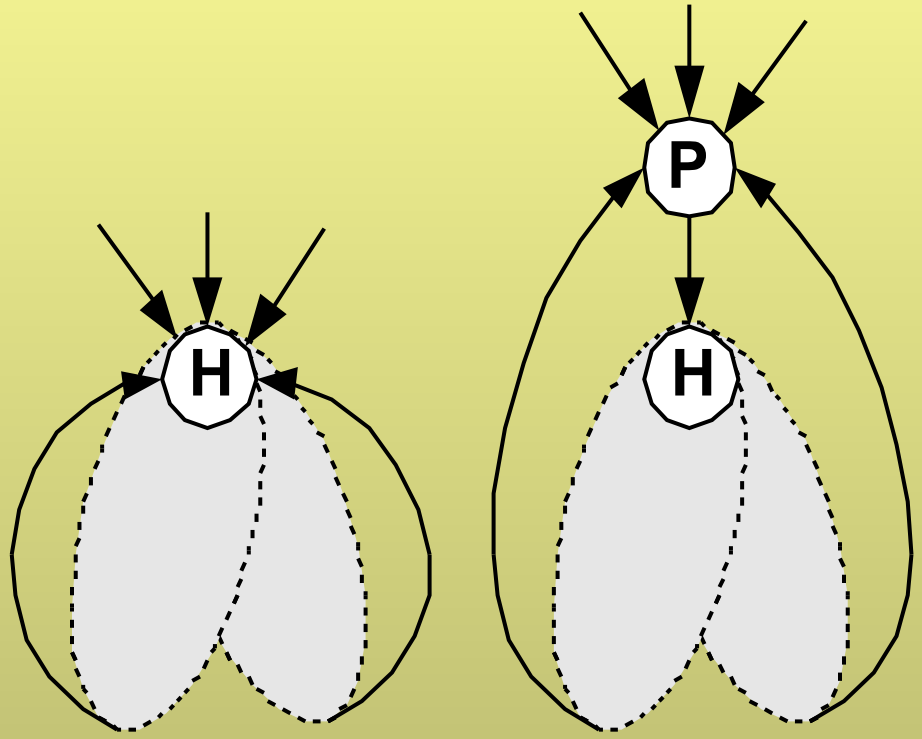




Non-loops with edges going back -- lacking domination



Introducing a preheader P to a natural loop with a header H



Introducing a preheader P to two natural loops sharing a header H

Constant folding is a typical example of data-flow analysis. Only by reasoning about the flow, we can replace `t12` by `5` in

```
t10 = 2
...
t11 = 3
...
t12 = t10 + t11
```

Terminology: *definition* (defining assignment)

use

a definition kills a subsequent definition

Problem of reaching definitions - data-flow equations

$$Rd_{en}(i) = \bigcup_{j \in Pred(i)} Rd_{ex}(j)$$

$$Rd_{ex}(i) = (Ad(i) \cup Rd_{en}(i)) - Kill(i)$$

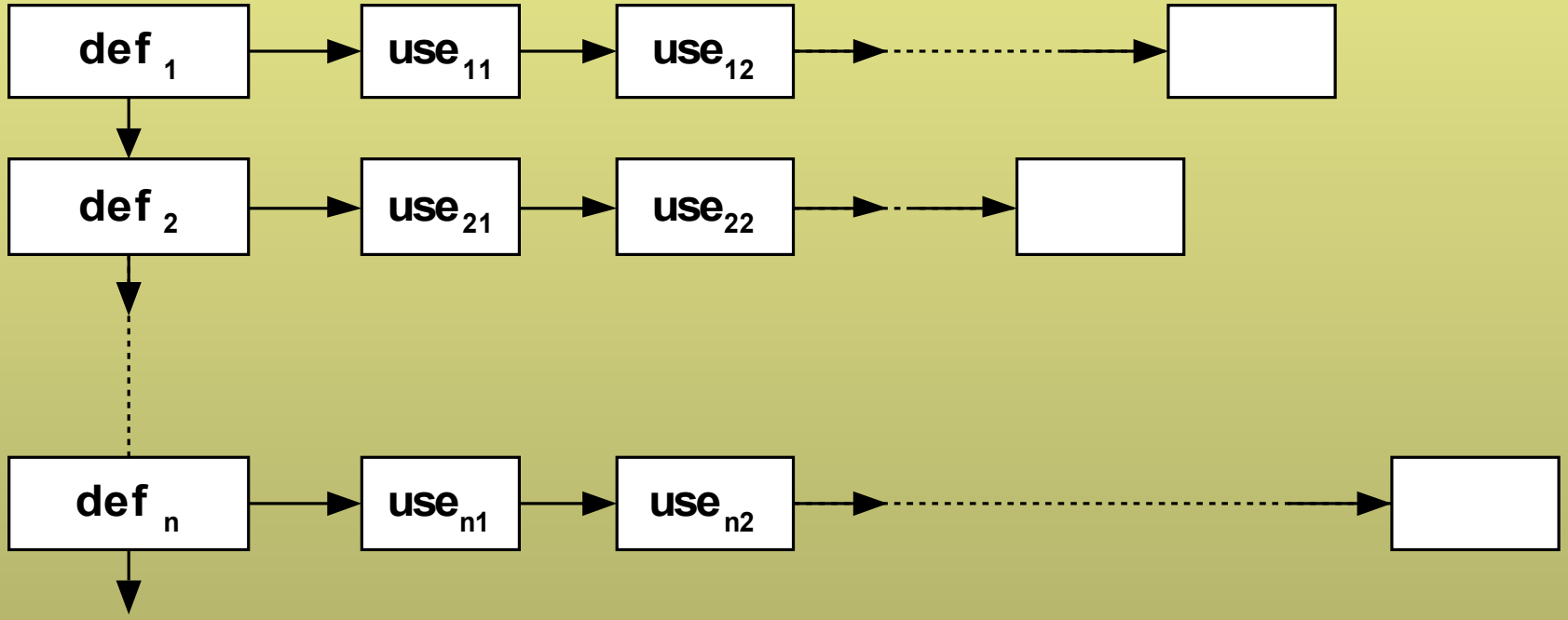
$$Ad(\texttt{Enter}) = Kill(\texttt{Enter}) = Rd_{en}(\texttt{Enter}) = RD_{ex}(\texttt{Enter}) = \emptyset$$

- **Computing $Ad(i)$ - traverse the block i and put in $Ad(i)$ any definition encountered.**
- **Computing $Kill(i)$ - After computing all $Ad(i)$, traverse the block i and for every assignment encountered, put in $Kill(i)$ any definition from $\bigcup Ad(i)$ that has the same left-hand side variable (except the current assignment, it does not kill itself).**

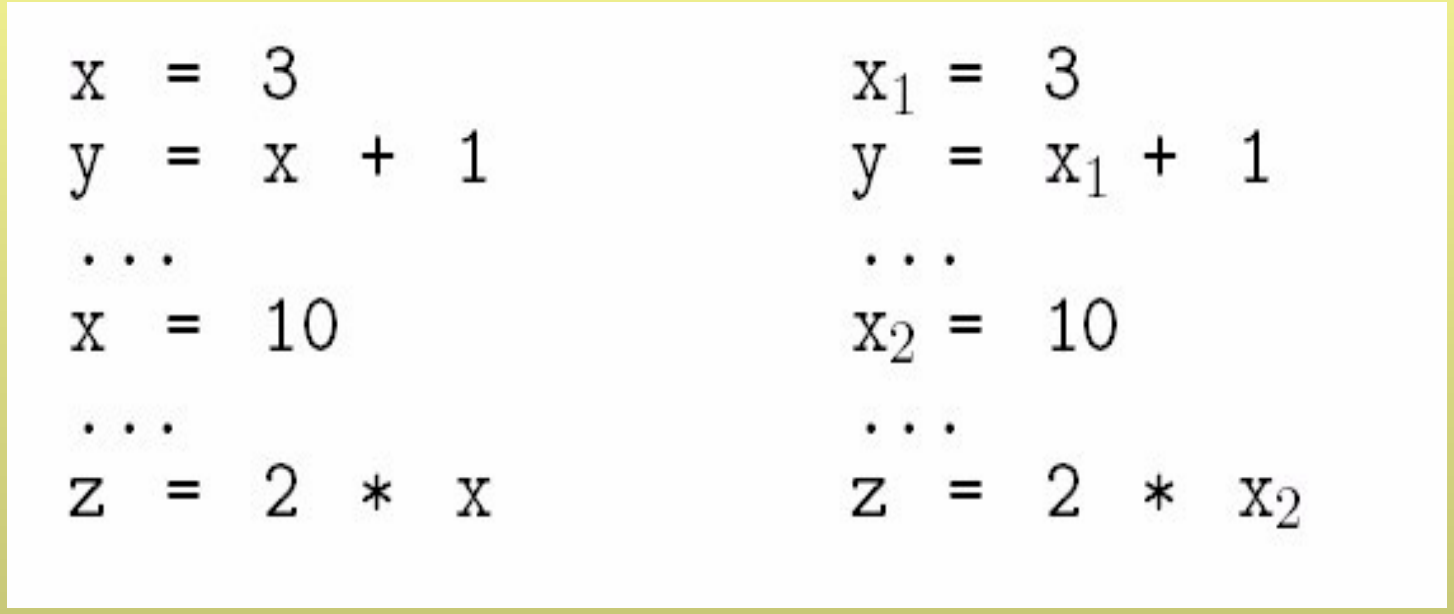
The equations are now solved iteratively traversing the flow graph in breadth-first fashion.

Sparse representations:

- **du-chains and ud-chains**
- **SSA (static single assignment)**

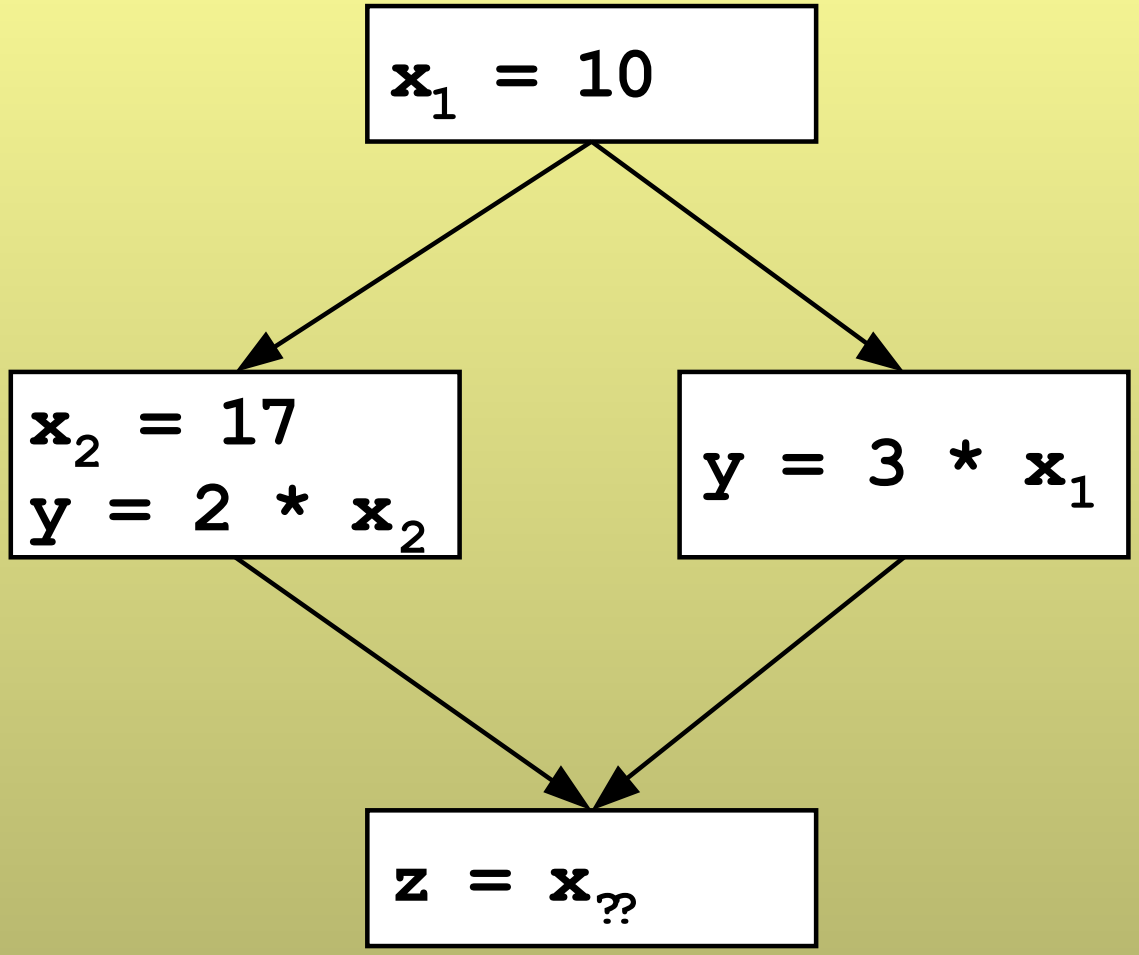


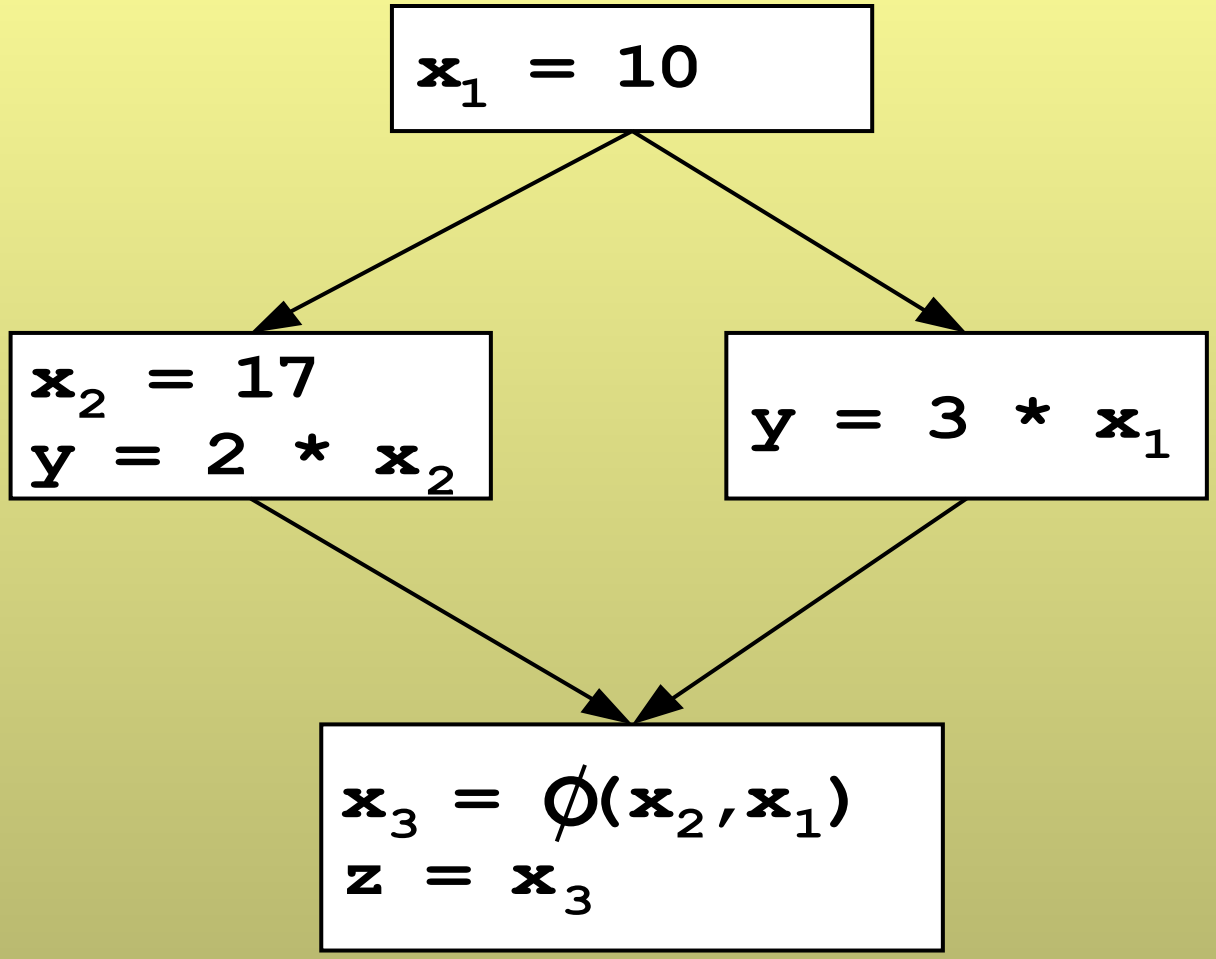
SSA can be regarded as systematic renaming of variables.

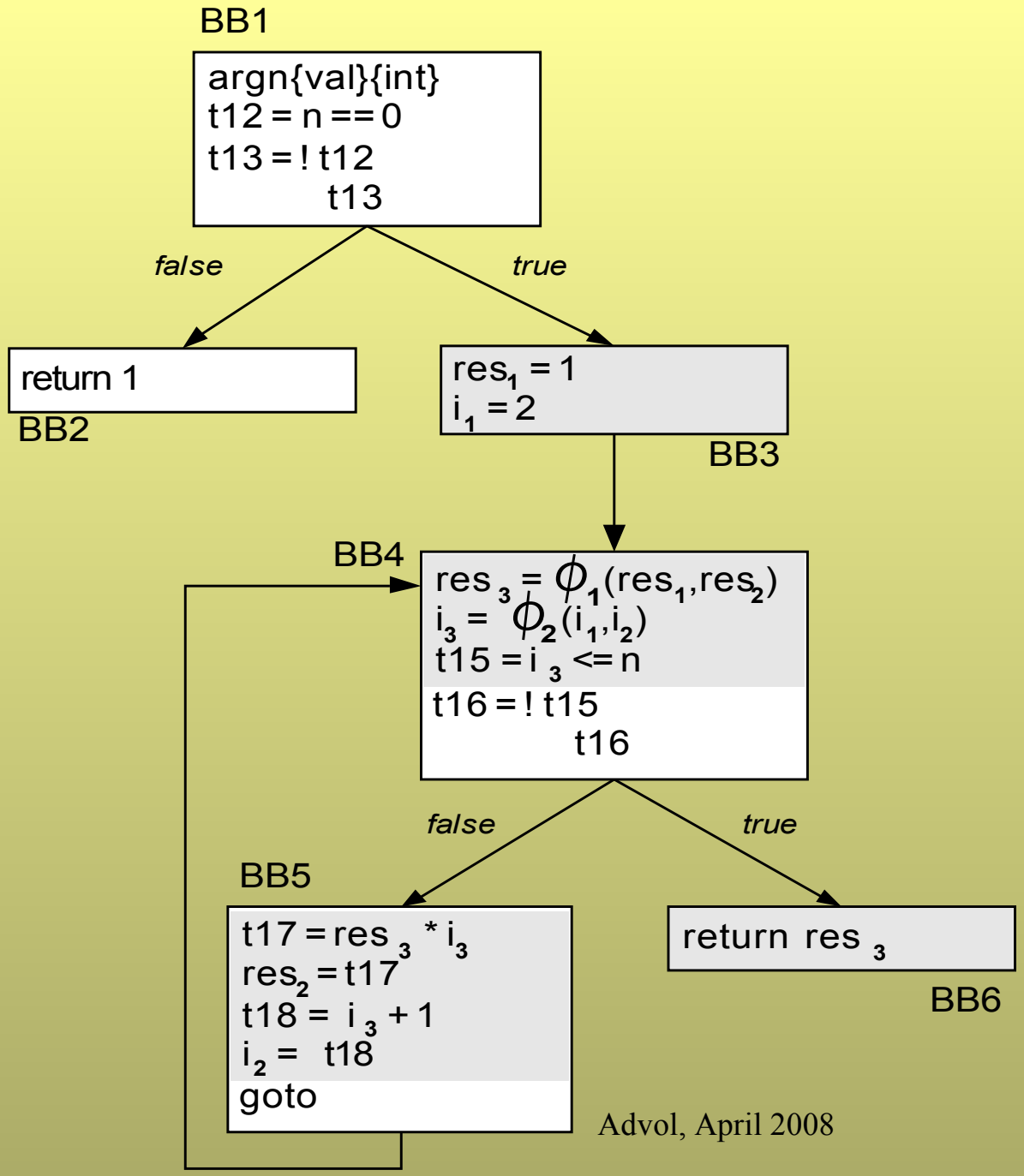


Not a problem within a simple basic block.

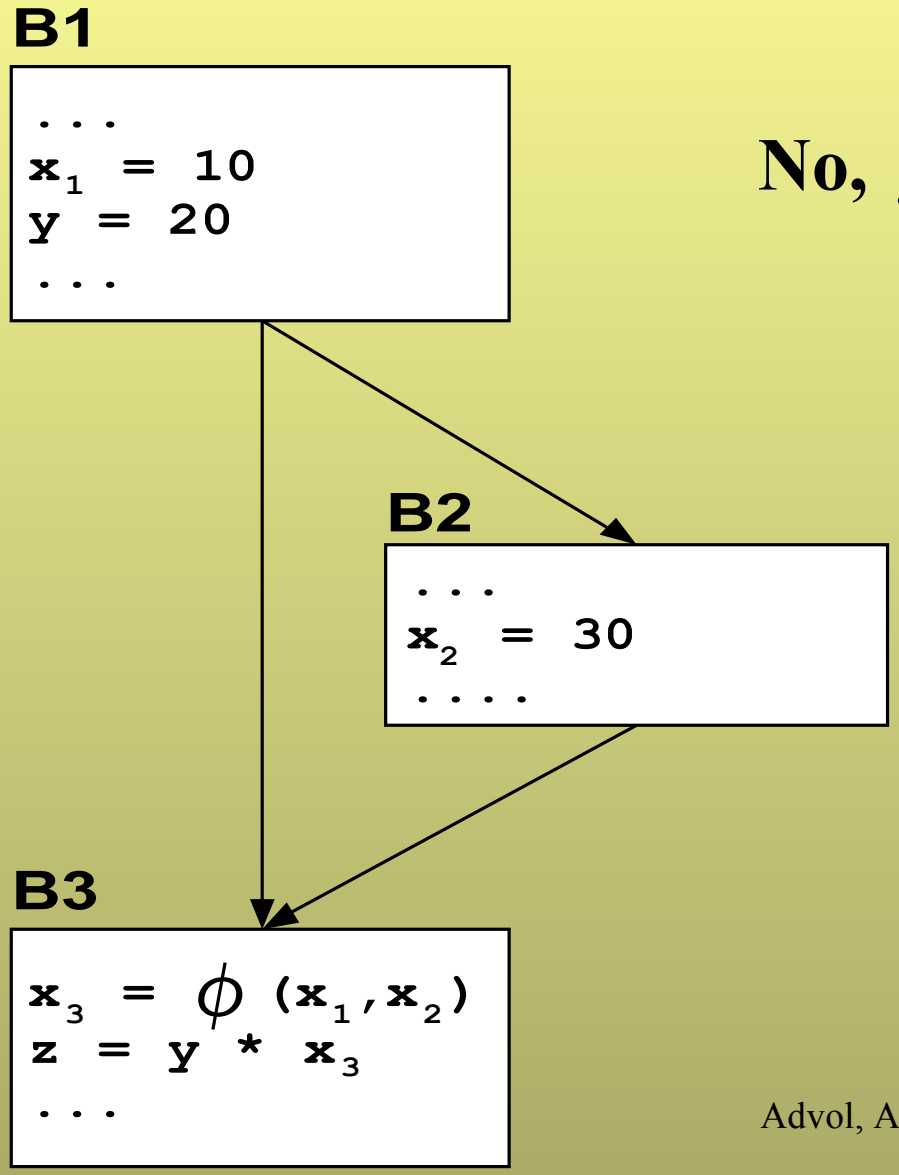
A problem in a join block.




$x_1 = 10$
$x_2 = 17$
$y = 2 * x_2$
$y = 3 * x_1$
$x_3 = \phi(x_2, x_1)$
$z = x_3$


BB1
argn{val}{int}
t12 = n == 0
t13 = ! t12
t13
false
true
return 1
BB2
res1 = 1
i1 = 2
BB3
BB4
res3 = ϕ1(res1,res2)
i3 = ϕ2(i1,i2)
t15 = i3 <= n
t16 = ! t15
t16
false
true
BB5
t17 = res3 * i3
res2 = t17
t18 = i3 + 1
i2 = t18
goto
return res3
BB6

Do we have to rename all variables ?



No, *y* need not be renamed.

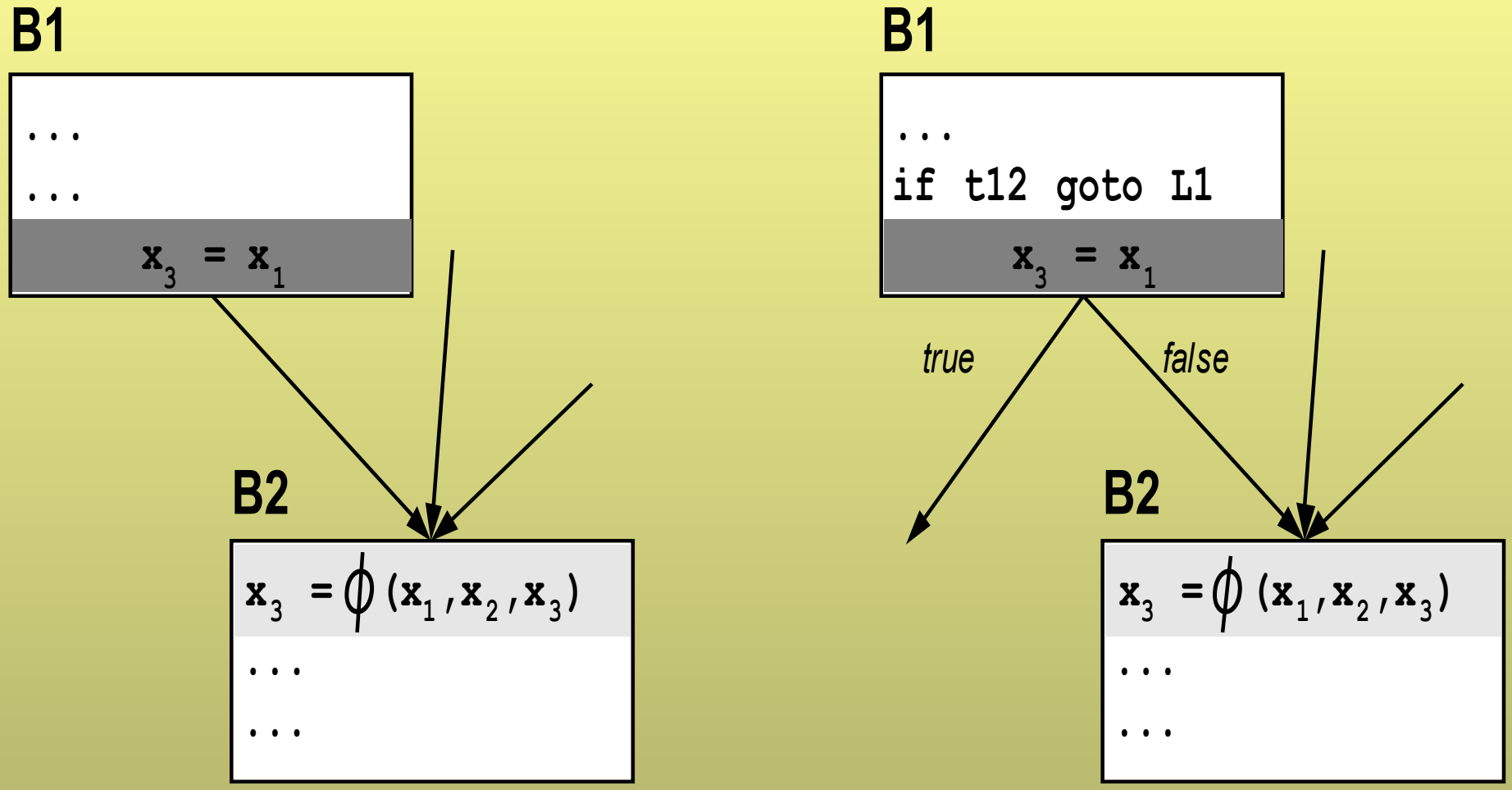
Path-convergence criteria for insertion of Φ

1. there are two distinct blocks `B1` and `B2` that both contain a definition of `x` that reaches to `J`, and
2. there is a path p_1 from `B1` to `J` and a path p_2 from `B2` to `J` that have no node in common except `J`, and
3. `J` can appear either in p_1 or in p_2 as an intermediate node (not the ending node), but not in both.

Numerous advantages for data analysis if the program is in SSA

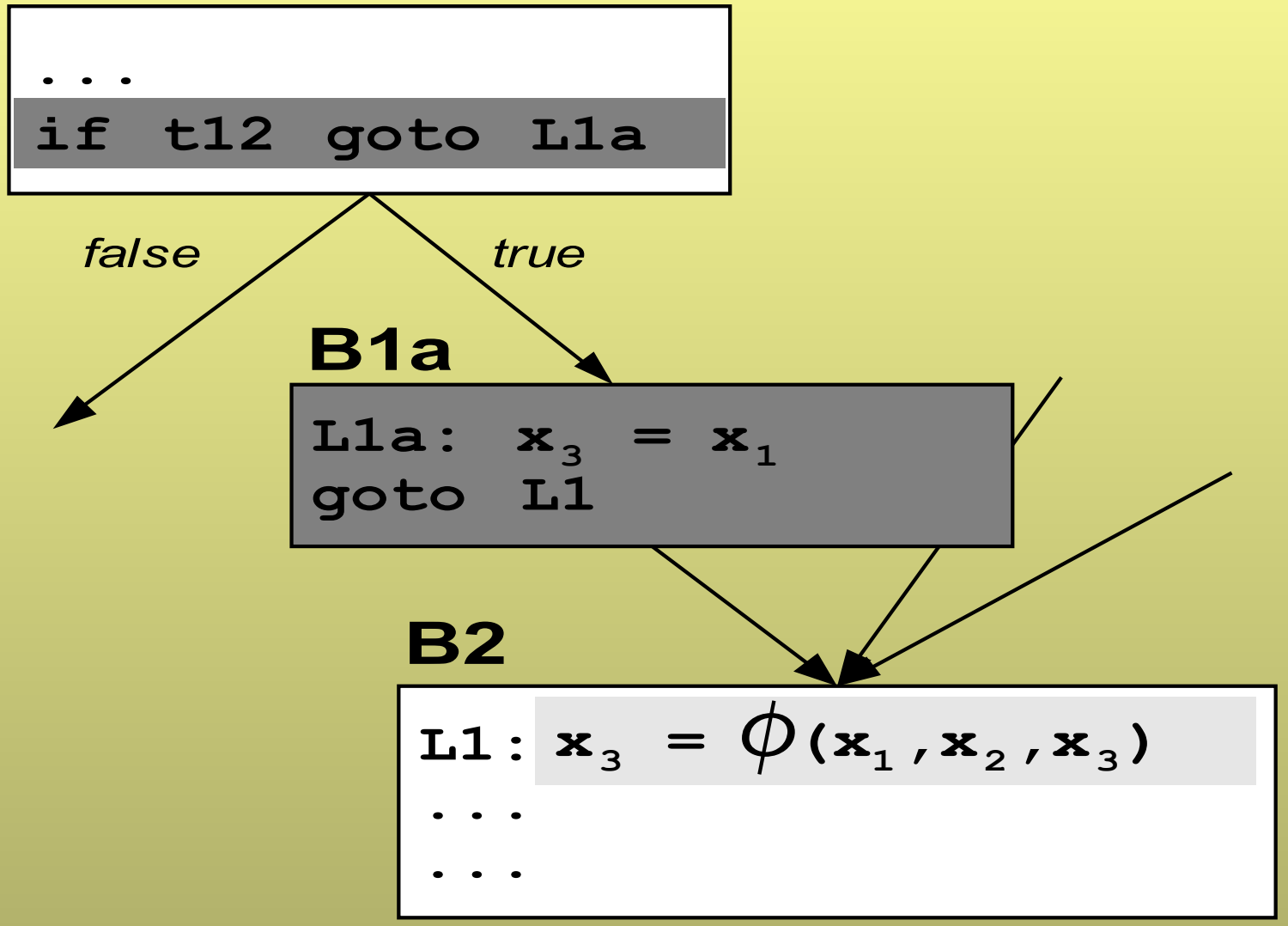
- **simplifies the analysis**
- **is almost linear in space (unlike do- and ud-chains)**
- **disassociates the parts of the code where the use of variable are just coincidental**

How to convert it back -- removing Φ



Replacing Φ with no edge splitting

B1 Replacing Φ with edge splitting



Peephole optimization

```
...                    ...
t17 = 2                f4 = 2
f4 = t17               ...
...
```

Very typical (in MACS `Icgen()`)

```
...                    ...
<label>:nop            <label>:<instruction>
<instruction>          ...
...
```

Peephole optimization vs. Delayed code emission

Dead-code elimination

```
...
t12 = x
x = t12 + 1
...
no further use of x
```

`x` is dead (not used after definition), while `t12` is not.

An instruction is dead if it only computes values not used in any instruction on any executable path leading from the instruction.

The dead-code elimination is an optimization method focused on detecting and eliminating dead instructions. It can be run on any level of IC but it is most suitable and effective on low level code including the target code.

A real care must be taken in evaluation of dead instructions; some real-world machine or assembly instructions are executed for their side-effects (like setting a condition code) and thus should not be considered dead even if they compute a dead value.

The basic approach to the detection of dead instructions is optimistic -- on a pass, mark some instructions as essential, and iterate the process in order to find the maximal set of essential instructions. The instructions that are not essential are considered dead and, consequently, eliminated.

Before starting the process of detection of essential instructions for the whole program, we identify the initial set of essential instructions by some other means, for instance for MIC -- I/O's, alloc(), transfer assignment, strstore(), etc. are deemed essential.

Unreachable-code elimination

```
    if (x | !x)
        goto A
    else
        goto B
    ...
A:  ...
B:  ...
```

Common error -- B will never be executed

clearly "visible" as `(x | !x)` is a tautology

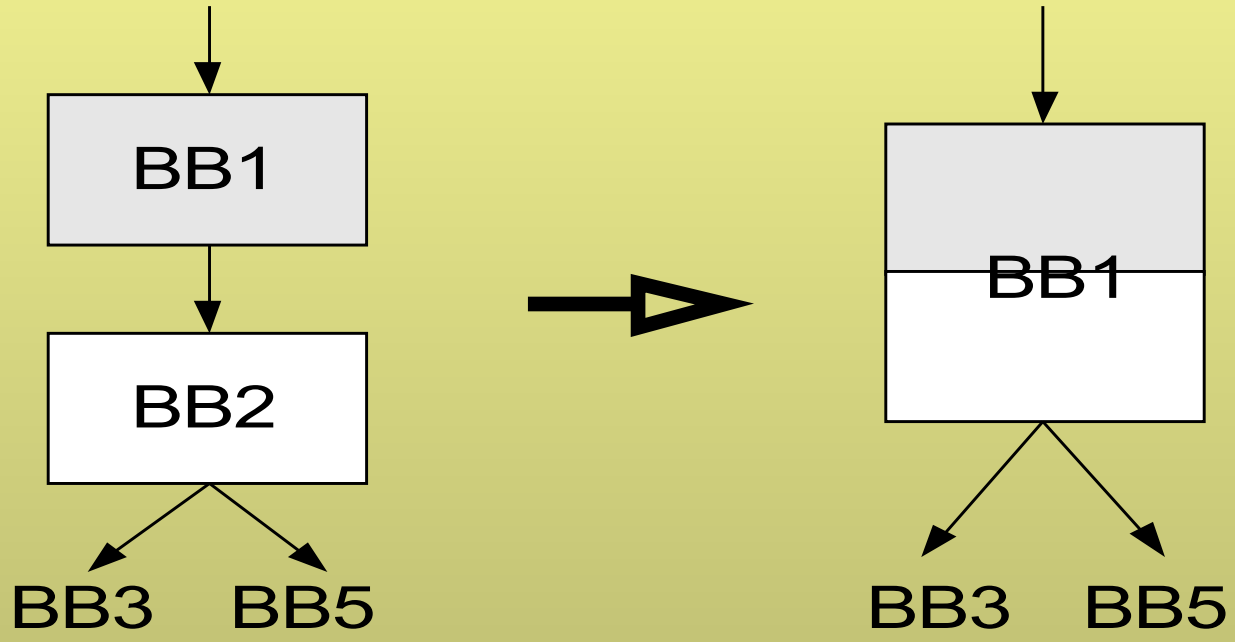
No longer “visible” in IC

```
        ...
        t13  =  !x
        t14  =  x or t13
        ...
        t15  =  !t14
        if t15 goto L1
        goto A
        goto L2
L1:goto B
L2:nop
        ...
 A:...
 B:...
```

The elimination of the unreachable code does not improve the performance as we are eliminating code that is NOT executed under any conditions. It can be run on any level of IC or target code.

In the graph-theoretical terms, the elimination of unreachable blocks boils down to the identifying the largest connected component of CFG containing the `Entry` block.

Straightening



If simplification

Example 1

```
    if true goto L1                goto L1
    if false goto L1
```

Example 2

```
    t12 = !t11
    if t12 goto L1
    goto L2                        if t11 goto L2
L1: ...                            L1: ...
    ...                                ...
L2: ...                            L2: ...
```

Example 3

```
    if t12 goto L1                 if t12 goto L2
    ...                            ...
    goto L2
L1: goto L2
L2: ...                            L2: ...
```

Value numbering

```
10        x  = y + t0
11        y  = x * z
12        t1 = y + t0
13        z1   = x * z
```

Clearly, we can simplify it to

```
10        x  = y + t0
11        y  = x * z
12        t1 = y + t0
13        z1   = y
```

Can we replace `t1=y+t0` with `x` ? NO

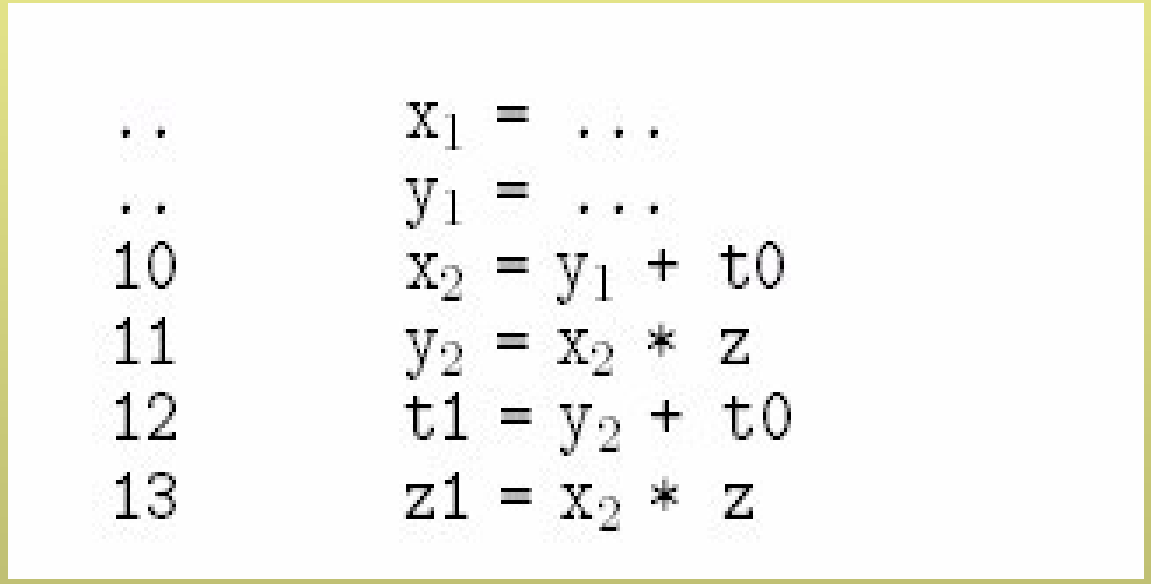
Value numbering is a procedure that discerns what can and what cannot be replaced.

Value numbering in a basic block -- the basic idea is based on hashing -- if two expressions hash to the same, one can replace the other.

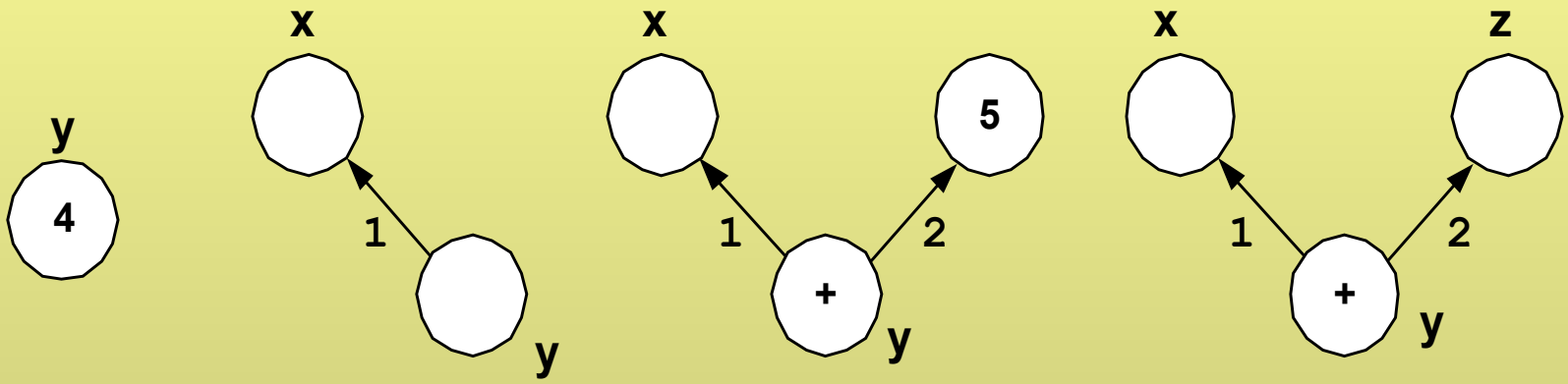
```
                4       3    2      1

10      x  = y  +  t0
          7      4    6    5

11      z  = x  *  z1
          8      7    2      1

12      t1 = y  +  t0
```

Hence we can replace `z = x * z` with `z1 = y`

More important and more complex is global value numbering (global means in a procedure). This process is simplified by having the code for the procedure in SSA.

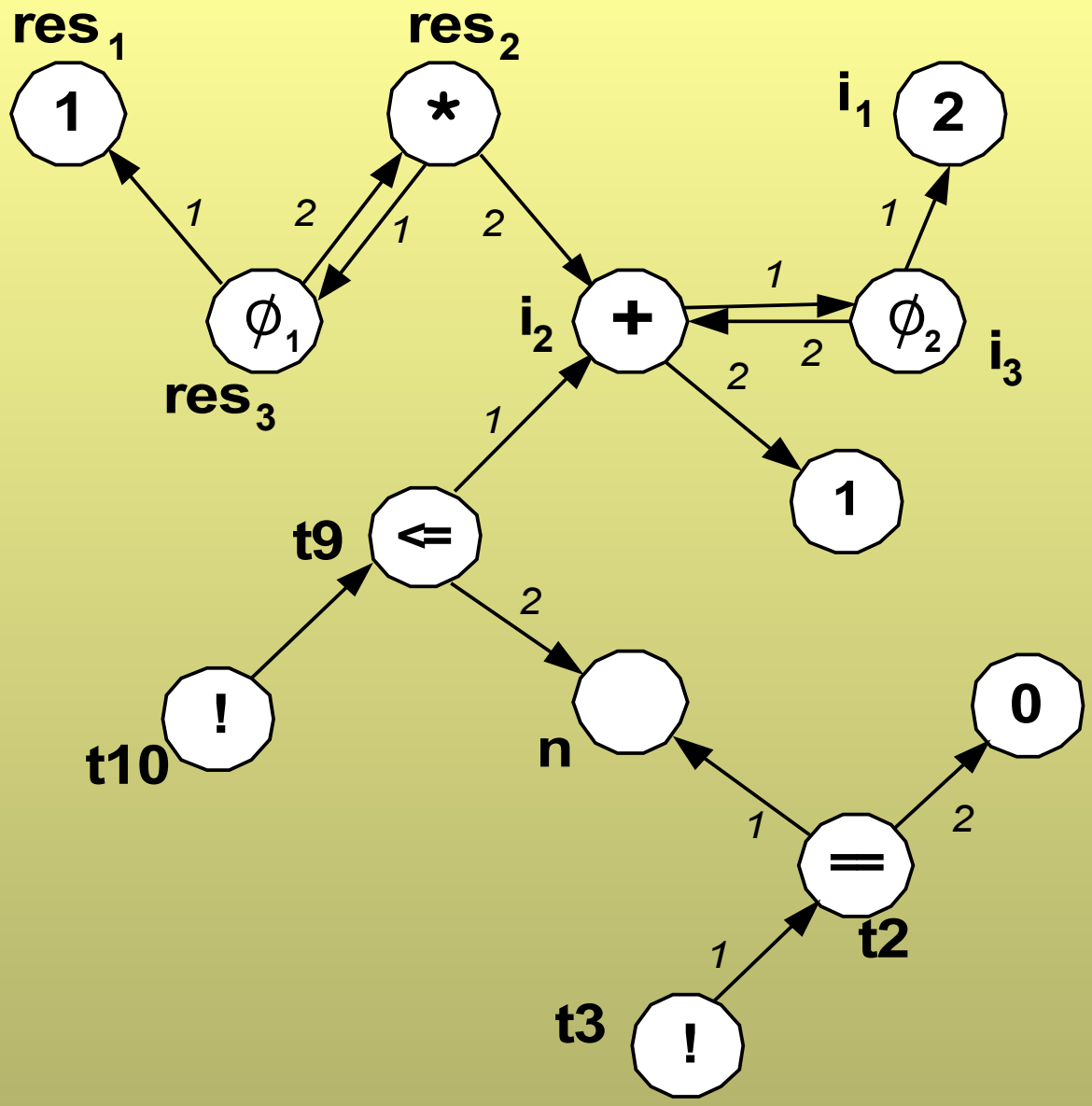


Note that in SSA form it is clear that we can replace `z1 = x * z` with `z1 = y`



`y = 4` `y = x` `y = x + 5` `y = x + z`

Building a value graph



Value graph for `fact()`

The basic idea is to define a congruence (indicating similarity) between variables and propagate it.

The *congruence* is a maximal relation on the value graph such that

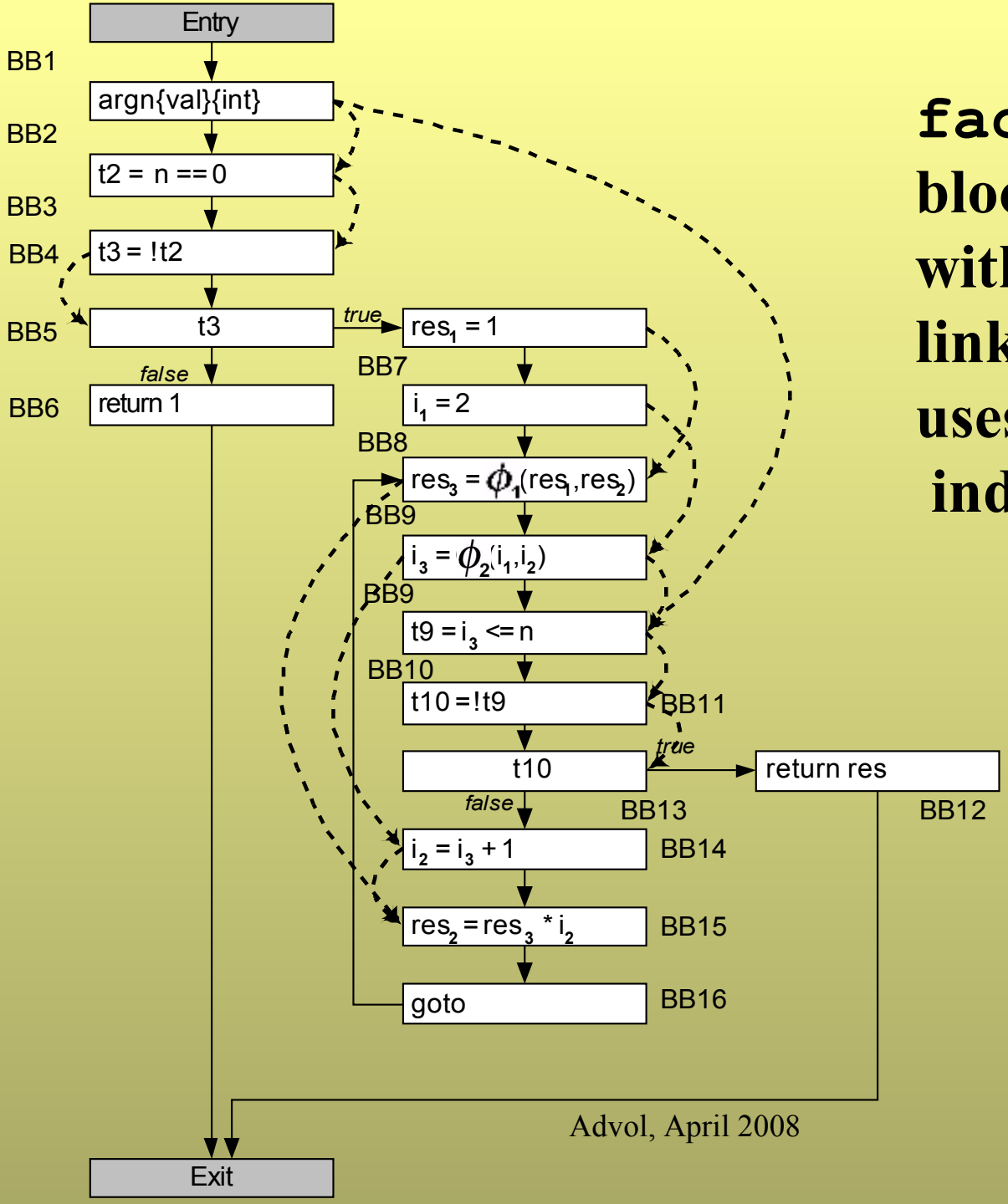
1. every node is *congruent* with itself,
2. two nodes labeled by the same constant are *congruent*,
3. two nodes with the same label are *congruent* if they have the same operators and the operators are *congruent*.

Typically, Aho, Hopcroft, Ullman's Partitioning Algorithm is used.

Copy propagation (or Copy folding)

Copy assignment `x = y` means that we can replace a use of `x` by a use of `y`.

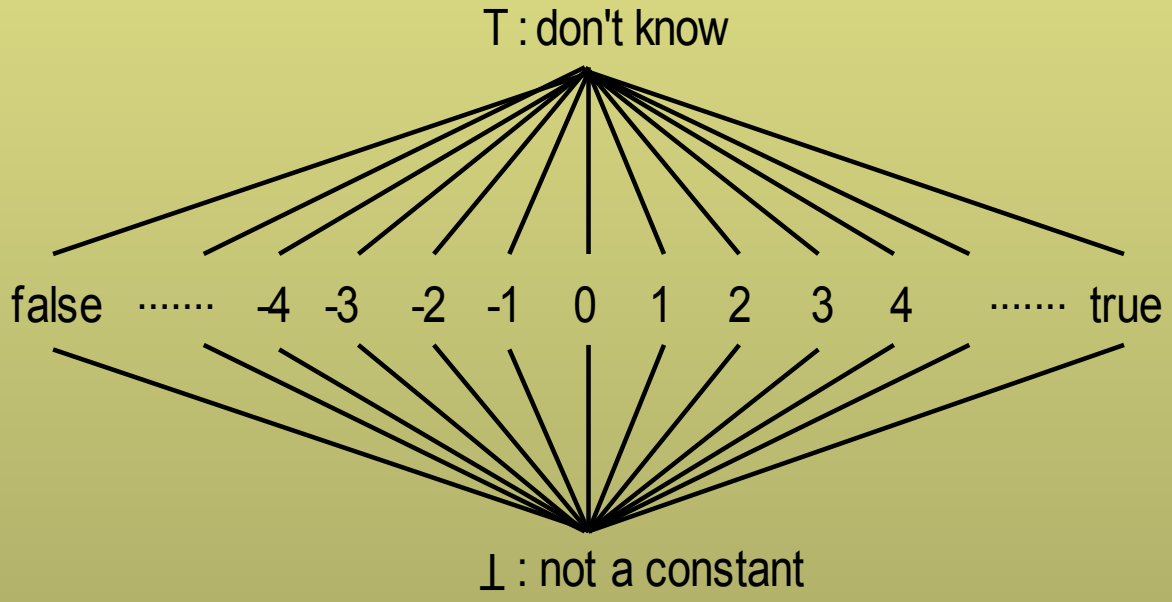
To do copy propagation, the program need not be in SSA form. If it is, it is straightforward as every variable has just a single defining assignment. Thus, when `x = y` is encountered, `y`'s defining assignment had been encountered before, and this is the defining assignment for `x`, so it is guaranteed that neither `y` nor `x` can change in a later instruction.



`fact()` is single block SSA form with SSA edges linking defs with uses indicated by ----

Sparse Conditional Constant Propagation

Deals with redundancy of the form `x = const` when we replace every use of `x` with `const`. We work with SSA form (sparse). It takes a form of symbolic computation using Constant Propagation Lattice.



```
t0 = 2
t1 = 3
t2 = t1+t2
```

```
t0 = 2
t1 = 3
t2 = 5
```

Can always be performed for integers and booleans

The same is not true for reals as the machine running the compiler may have a different floating-point arithmetic than the target machine. The problem of numerous exceptions must be considered. In simple terms, *the effort to include a constant expression evaluation in the compiler is just too extensive and generally, real constant folding is neither considered nor recommended.*

There are many computational rules for CPL,

just for an illustration:

The rules for CPL concerning $\wedge$ are:

$$a \wedge \top = a$$
$$a \wedge \bot = \bot$$
$$c_1 \wedge c_2 = c_1, \text{ if } c_1 = c_2$$
$$c_1 \wedge c_2 = \bot, \text{ if } c_1 \neq c_2$$

The rule for ϕ: $\phi(a_1, a_2, ..., a_n) = a_1 \wedge a_2 \wedge ... \wedge a_n$.

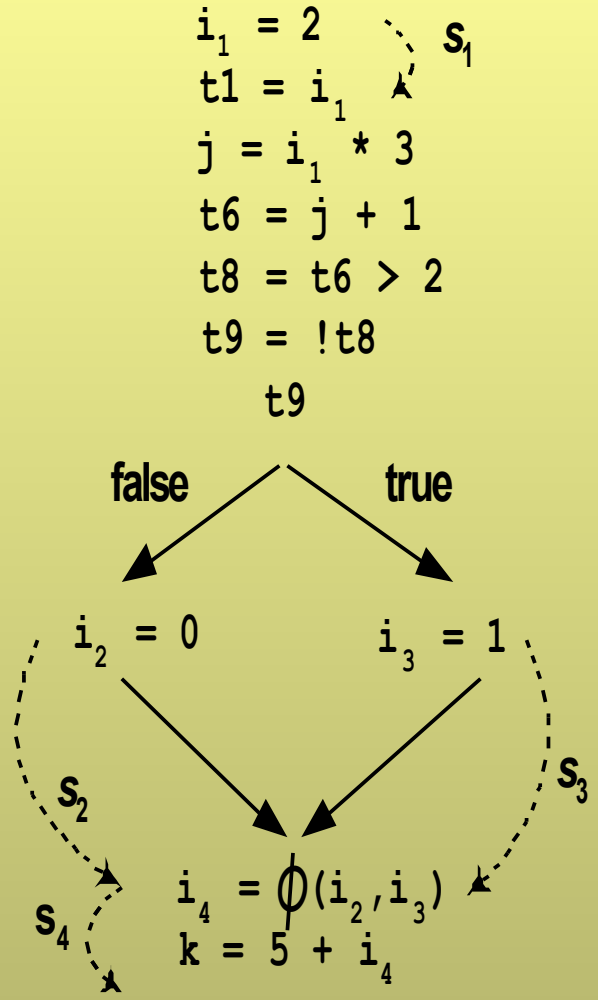
```
i = 2;
j = i * 3;
if (j + 1 > 2)
   i = 0;
else
   i = 1;
k = 5 + i;
```

A careful examination of the code will reveal that it can be reduced to

```
j = 6;
i = 0;
k = 5;
```

After peephole optimization, the streamlined code:

```
    i = 2
    t1 = i
    j = i * 3
    t6 = j + 1
    t8 = t6 > 2
    t9 = !t8
    if t9==true goto L2
L1: i = 0
    goto L3
L2: i = 1
L3: k = 5 + i
```



1) evaluates **i** to 2 and adds the SSA edge **S**
2) evaluates **t**1 to 2 and adds an SSA edge (not shown)
3) evaluates **j** to 6 and adds an SSA edge (not shown)
4) evaluates **t6** to 7 and adds an SSA edge (not shown)
5) evaluates **t8** to **true** and adds an SSA edge (not shown)
6) evaluates **t9** to **false** and adds an SSA edge (not shown)
7) marks the **false** branch as executable and leaves the **true** branch as non-executable

8) evaluates **i** to 0 and adds the SSA edge **S**

Common-subexpression elimination

```
t12 = x * y
...
t13 = x * y
```

can be replaced by

```
t14 = x * y
t12 = t14
...
t13 = t14
```

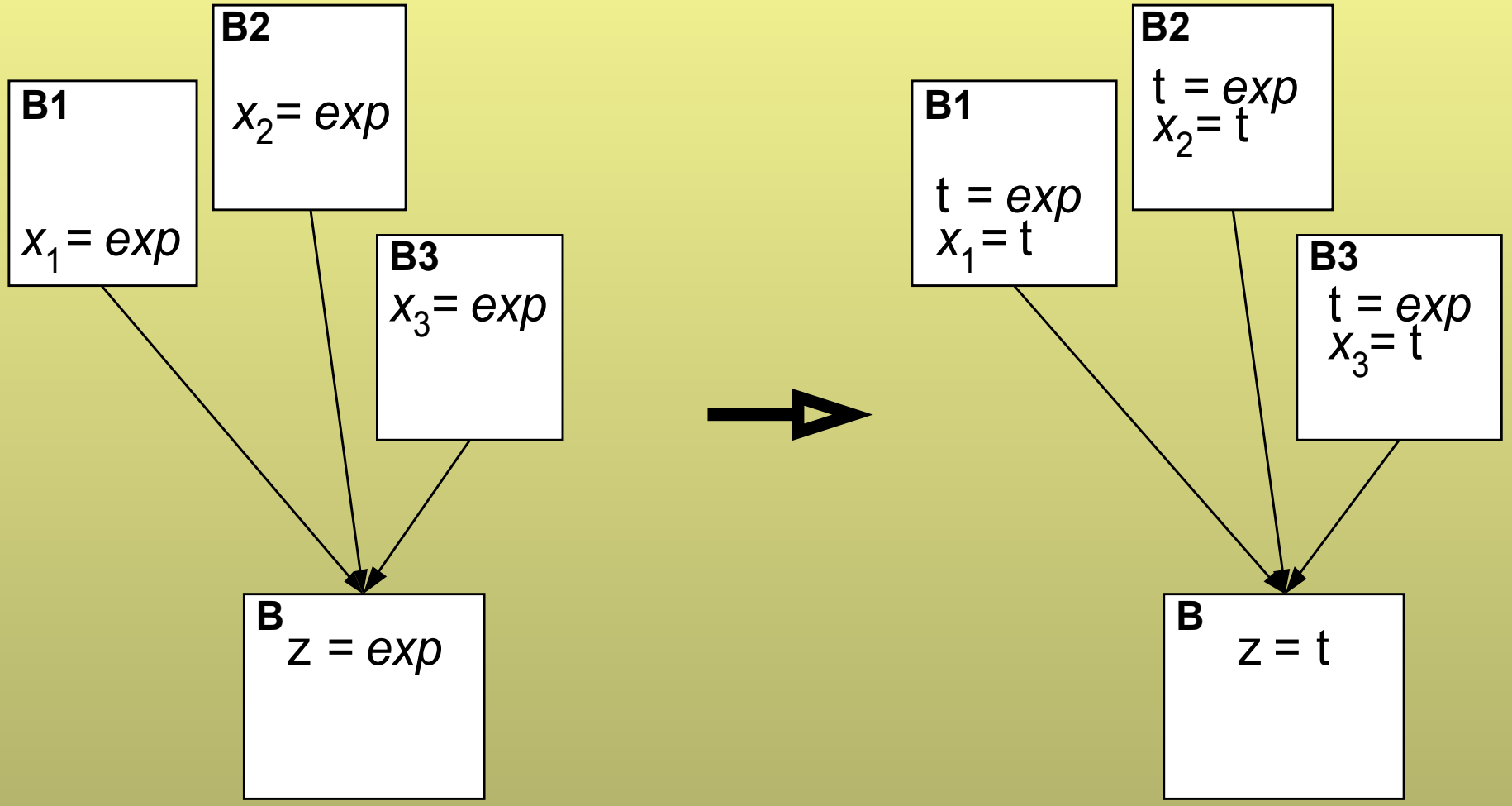
provided that the values of `x` and `y` have not changed between `t12 = x * y` and `t13 = x * y`.

For a basic block it is simple (similar ideas as in value numbering).

More involved is global elimination -- computing *available expressions*.

An expression *exp* is <u>available</u> at the entry to a basic block `B` if on every path in the flow graph from the `Entry` block to `B` there is an evaluation of *exp* that is not subsequently killed by having one or both of its operands reassigned (hence the value of *exp* stays the same).

Available expression and global common-subexpression elimination



We have to solve data flow equations

$$Avail_{en}(i) = \bigcap_{j \in Pred(i)} Avail_{ex}(j)$$

$$Avail_{ex}(i) = Avail(i) \cup \left(Avail_{en}(i) - Kill(i)\right)$$

The purpose of all of this computation is to obtain `Avail`$_{en}$`(i)` for all `i`.

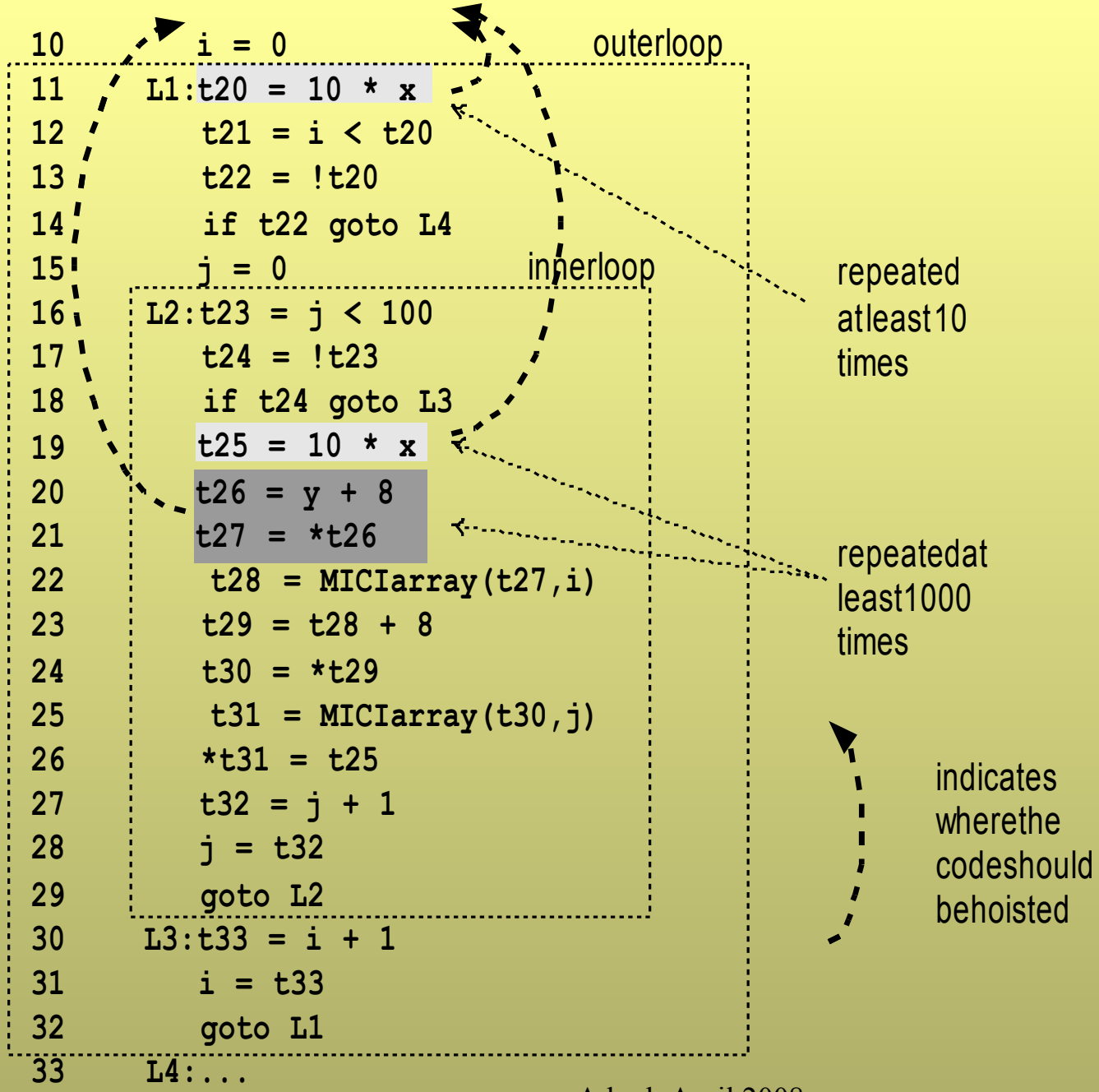
Loop-invariant code hoisting

```
...
for(i = 0; i < 10 * x; i++)
  for(j = 0; j < 100; j++)
    y[i][j] = 10 * x;
...
```

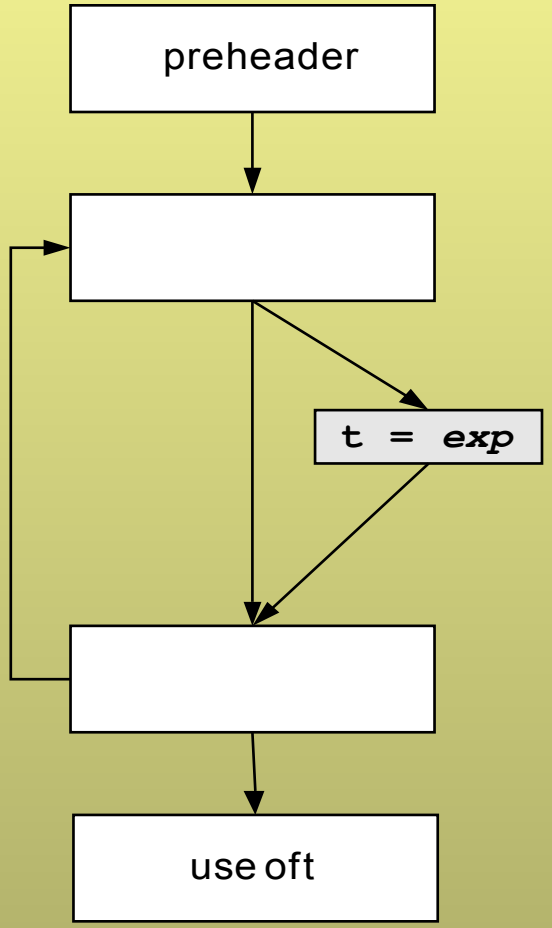
bad

```
...
z = 10 * x;
for(i = 0; i < z; i++)
  for(j = 0; j < 100; j++)
    y[i][j] = 10 * x;
...
```

Better
looks OK

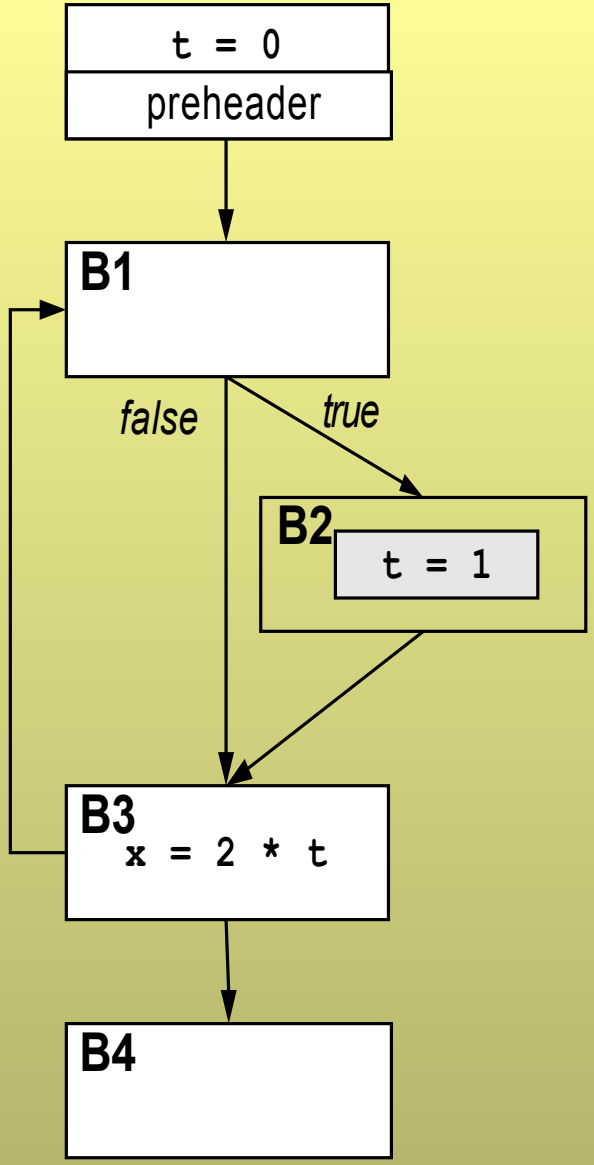

10 i = 0
11 L1:t20 = 10 * x
12 t21 = i < t20
13 t22 = !t20
14 if t22 goto L4
15 j = 0
16 L2:t23 = j < 100
17 t24 = !t23
18 if t24 goto L3
19 t25 = 10 * x
20 t26 = y + 8
21 t27 = *t26
22 t28 = MICIarray(t27,i)
23 t29 = t28 + 8
24 t30 = *t29
25 t31 = MICIarray(t30,j)
26 *t31 = t25
27 t32 = j + 1
28 j = t32
29 goto L2
30 L3:t33 = i + 1
31 i = t33
32 goto L1
33 L4:...
outerloop
innerloop
repeated atleast10 times
repeatedat least1000 times
indicates wherethe codeshould behoisted

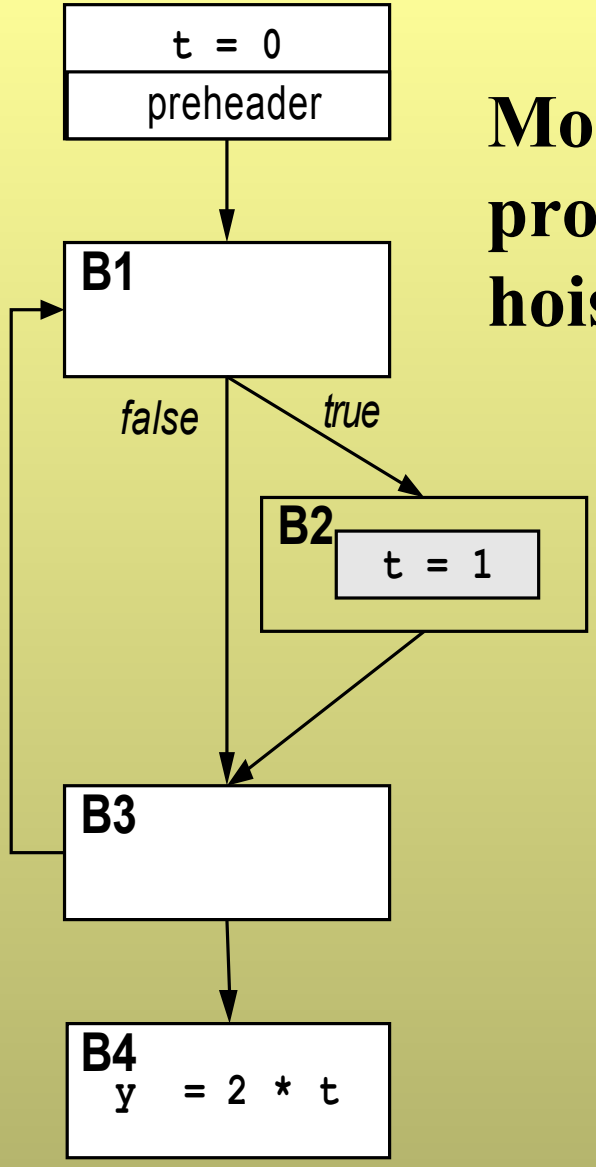
Definition of invariant is inductive, so again we work with a fix-point problem.



Problem with 'simple' hoisting

still better to hoist than not.





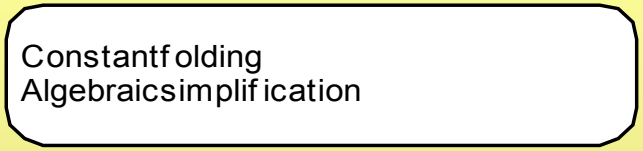
More complex problem with hoisting

An assignment that is a candidate to be hoisted to the preheader, must satisfy additional constraints:

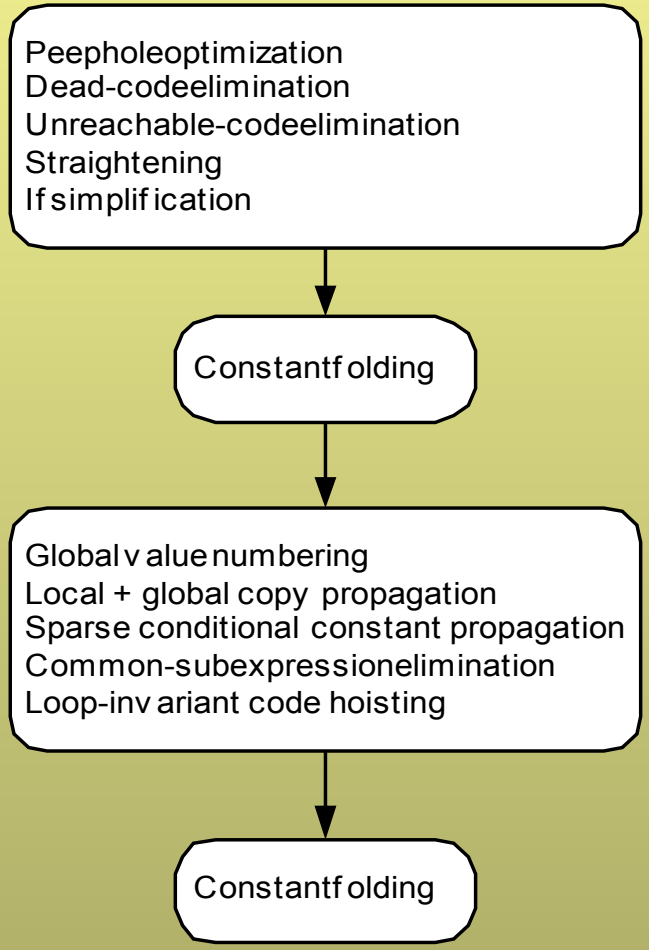
- **The assignment must be in a basic block that dominates all uses of the left-hand variable in the loop (this takes care of the situation depicted on the left as B2 does not dominate B3 with the use of t).**

- **The assignment must be in a basic block that dominates all exit blocks of the loop (this takes care to the situation depicted on the right as B2 does not dominate the exit block B3).**

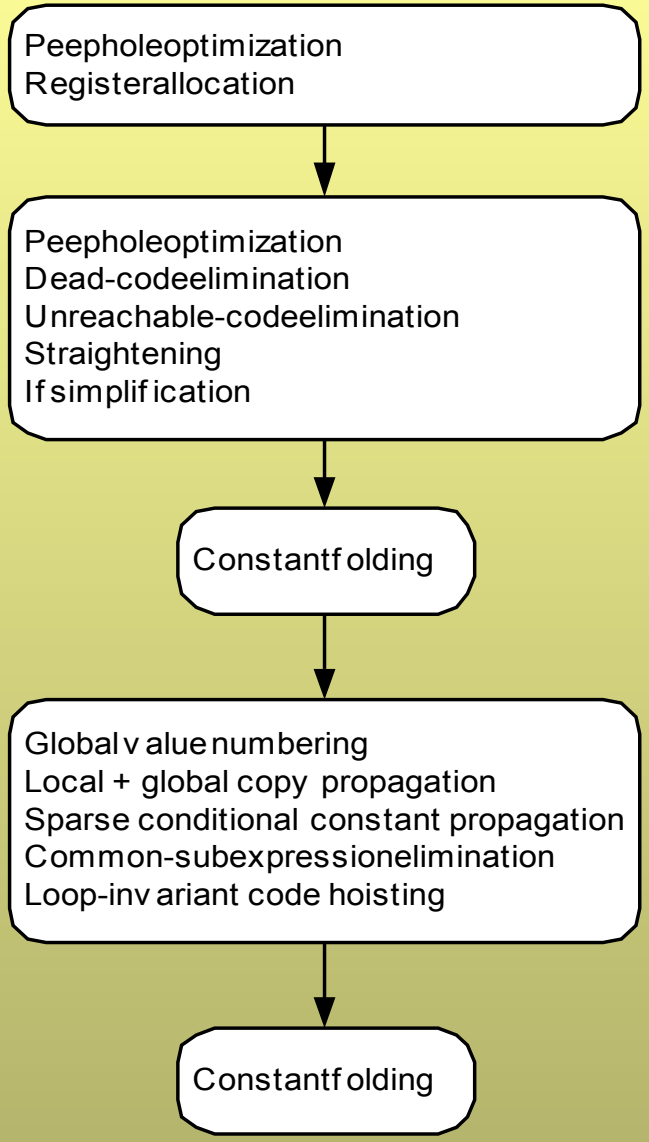
on *MACS* source code level



on *IC* level



on *MI* level



Thank you!