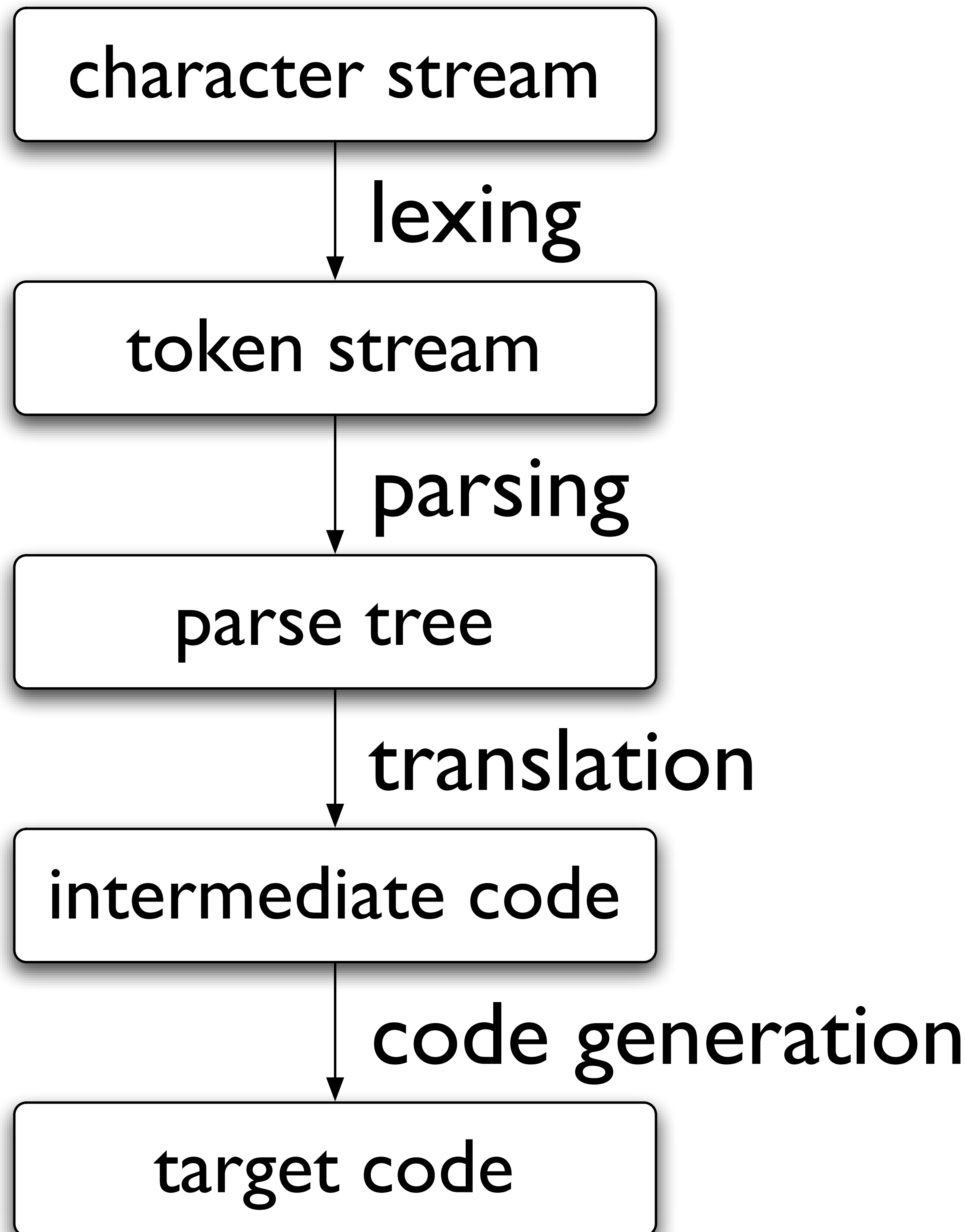
Optimising Compilers

Computer Science Tripos Part II

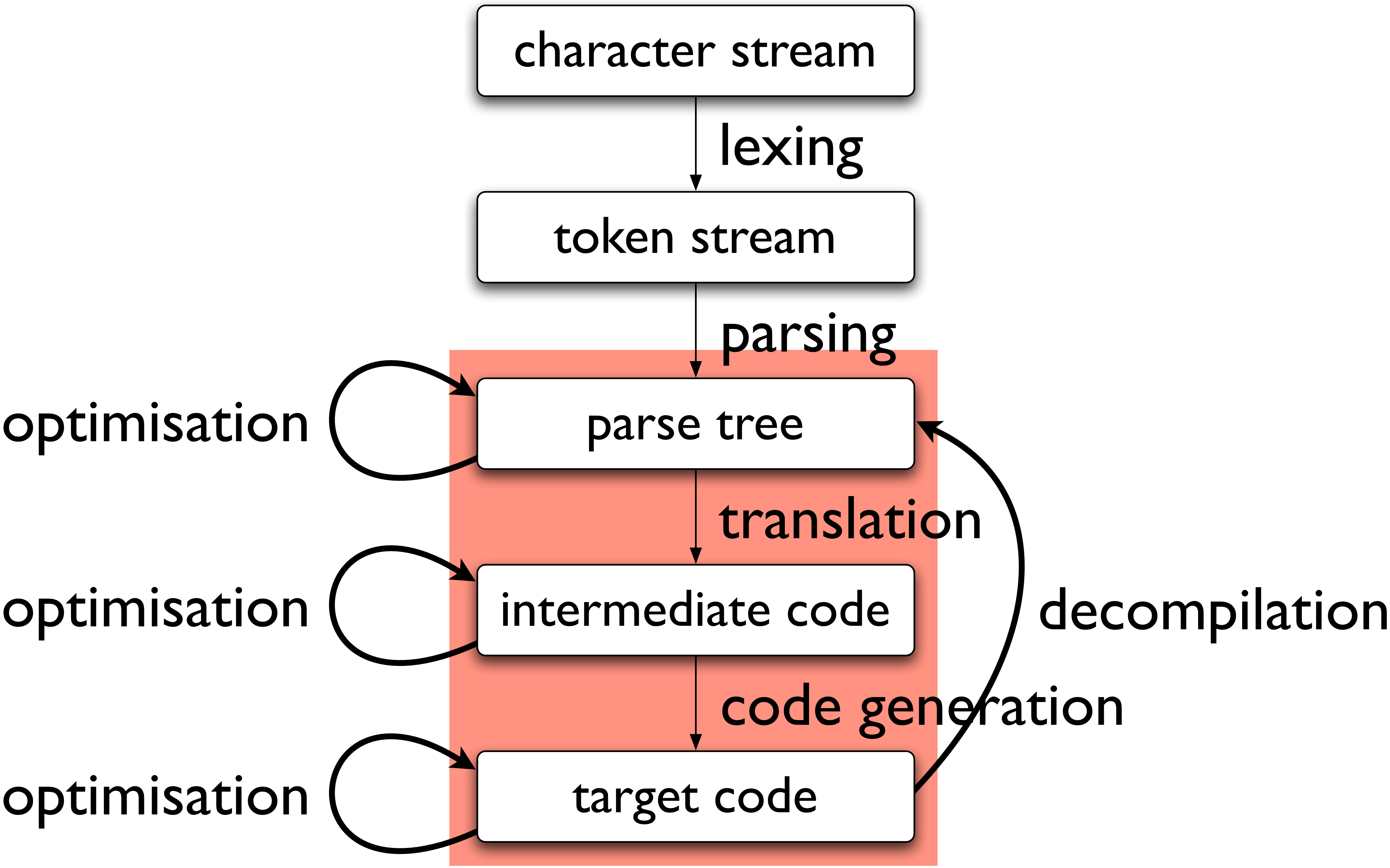
Timothy Jones

Lecture 1
Introduction

A non-optimising compiler



An optimising compiler



Optimisation
(really "amelioration"!)

Good humans write simple, maintainable, *general* code.

Compilers should then *remove unused generality*, and hence hopefully make the code:

- Smaller
- Faster
- Cheaper (e.g. lower power consumption)

Optimisation

=

Analysis

+

Transformation

Analysis + Transformation

- Transformation *does something dangerous*.
- Analysis determines *whether it's safe*.

Analysis + Transformation

- An analysis shows that your program has some property...
- ...and the transformation is designed to be safe for all programs with that property...
- ...so it's safe to do the transformation.

Analysis + Transformation

```
int main(void)
{
  return 42;
}

int f(int x)
{
  return x * 2;
}
```

Analysis + Transformation

```
int main(void)
{
  return 42;
}
```

✓

Analysis + Transformation

```c
int main(void)
{
  return f(21);
}

int f(int x)
{
  return x * 2;
}
```

Analysis + Transformation

```
int main(void)
{
  return f(21);
}
```

Analysis + Transformation

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

Analysis + Transformation

```
int t = k * 2;
while (i <= t) {
  j = j * i;
  i = i + 1;
}
```

Analysis + Transformation

```
while (i <= k*2) {
  k = k - i;
  i = i + 1;
}
```

Analysis + Transformation

```
int t = k * 2;
while (i <= t) {
  k = k - i;
  i = i + 1;
}
```

Stack-oriented code

```
iload 0
iload 1
iadd
iload 2
iload 3
iadd
imul
ireturn
```

?

3-address code

```
MOV t32,arg1
MOV t33,arg2
ADD t34,t32,t33
MOV t35,arg3
MOV t36,arg4
ADD t37,t35,t36
MUL res1,t34,t37
EXIT
```

C into 3-address code

```
int fact (int n) {
  if (n == 0) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}
```

C into 3-address code

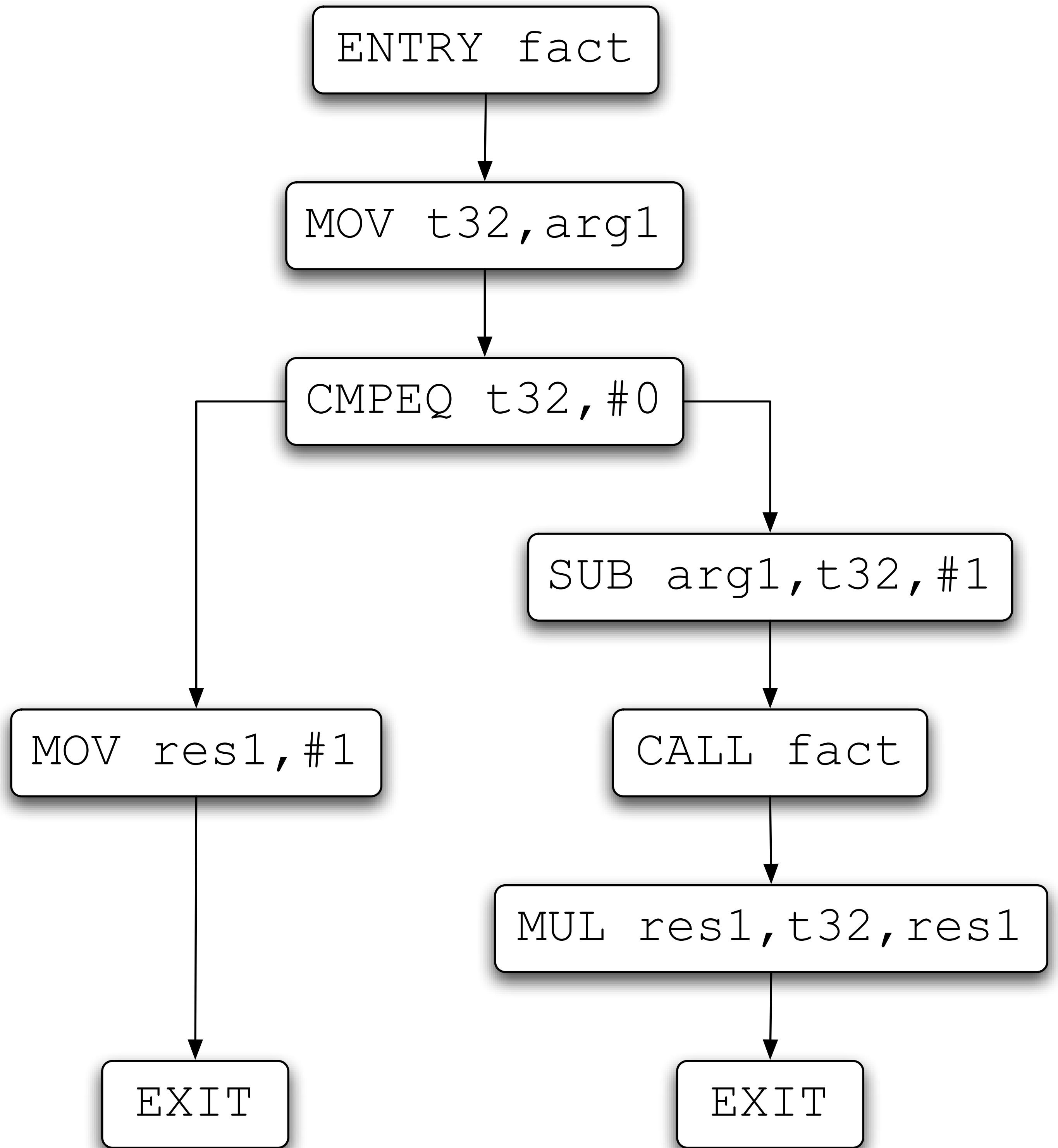
```
       ENTRY fact
       MOV t32,arg1
       CMPEQ t32,#0,lab1
       SUB arg1,t32,#1
       CALL fact
       MUL res1,t32,res1
       EXIT
lab1:  MOV res1,#1
       EXIT
```

Flowgraphs

- A graph representation of a program
- Each node stores 3-address instruction(s)
- Each edge represents (potential) control flow:

$$\begin{aligned} pred(n) &= \{n' \mid (n', n) \in edges(G)\} \\ succ(n) &= \{n' \mid (n, n') \in edges(G)\} \end{aligned}$$

Flowgraphs



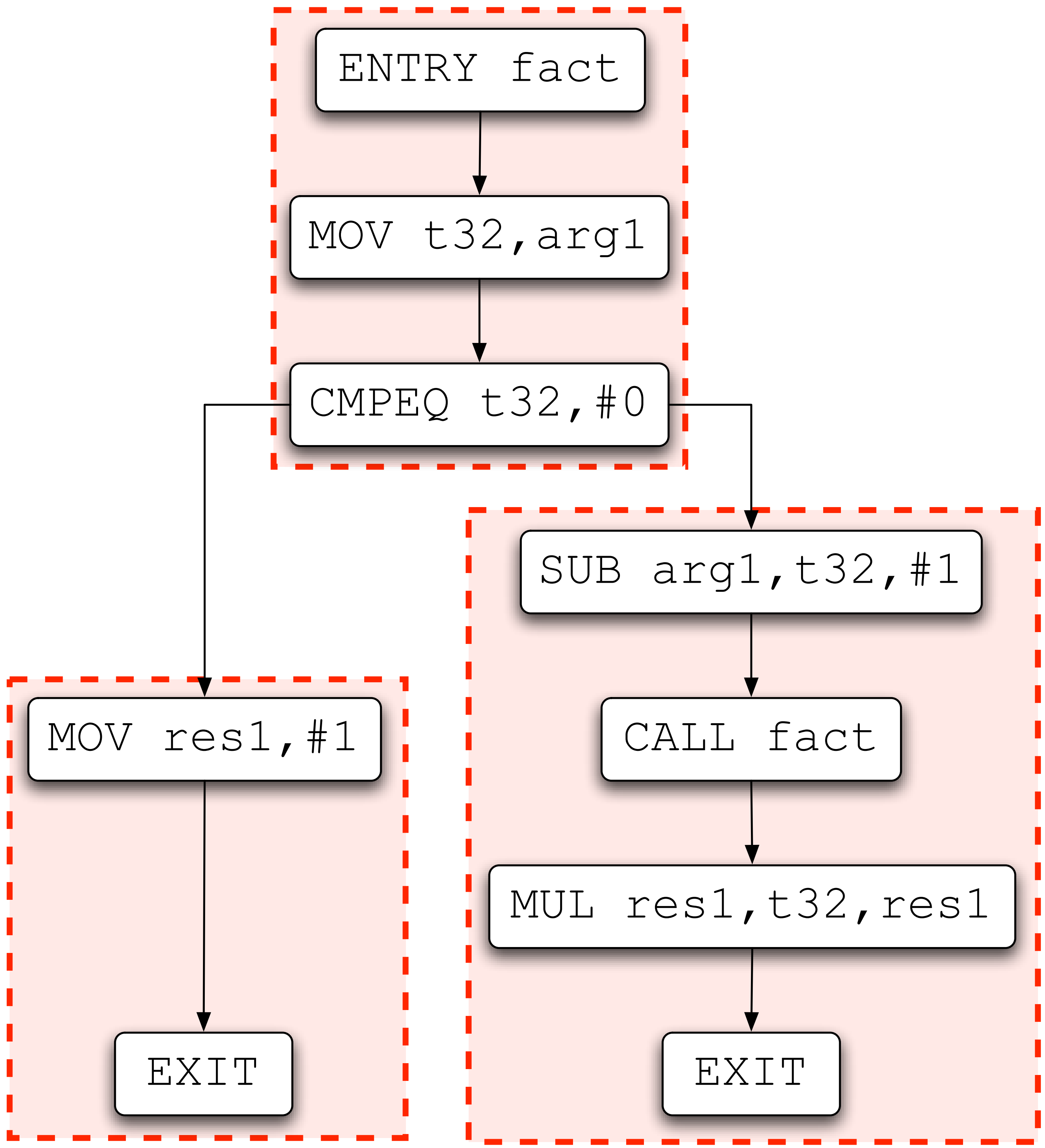
ENTRY fact
MOV t32,arg1
CMPEQ t32,#0
SUB arg1,t32,#1
MOV res1,#1
CALL fact
MUL res1,t32,res1
EXIT
EXIT

Basic blocks

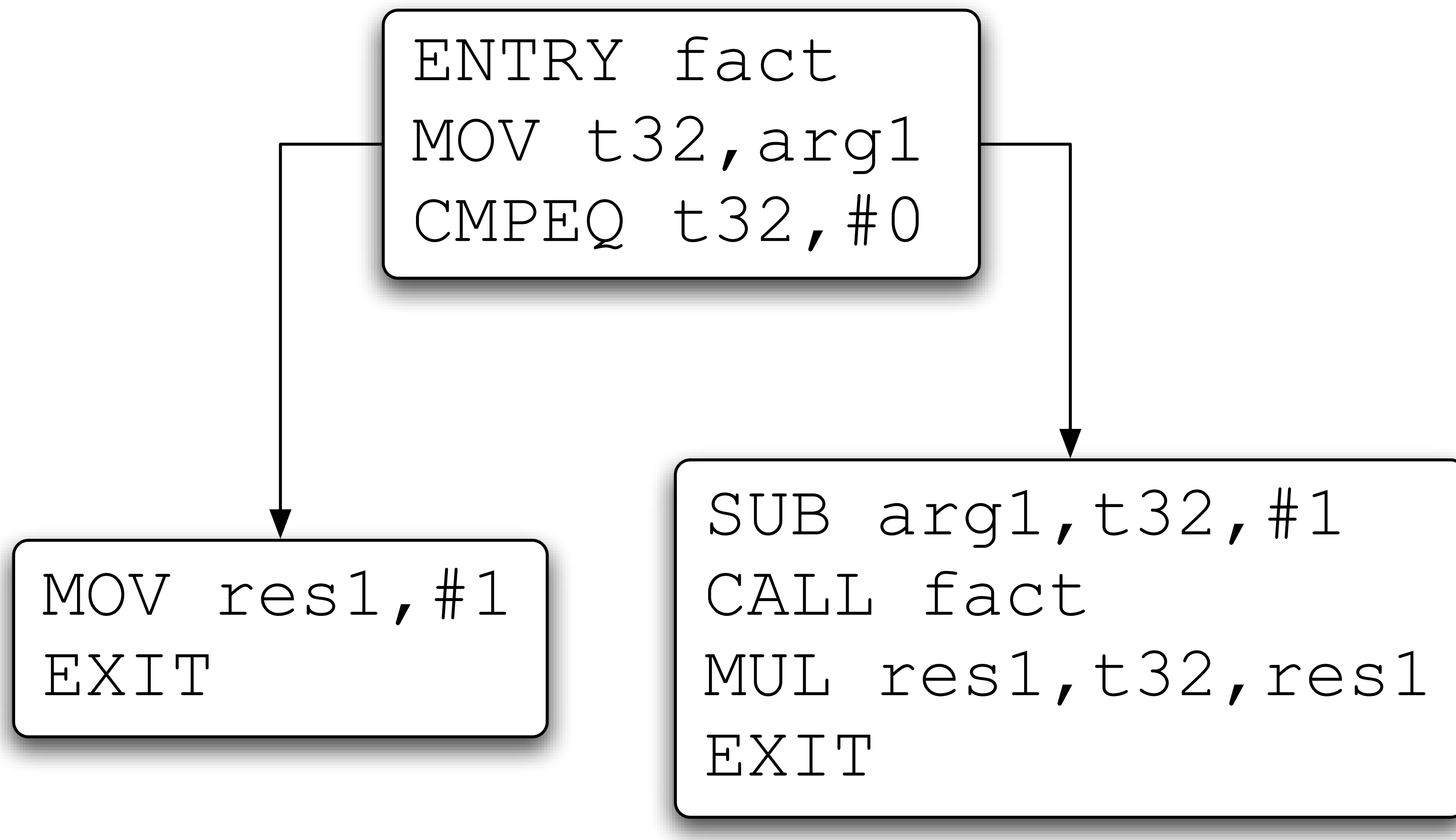
A maximal sequence of instructions $n_1, \ldots, n_k$ which have

- exactly one predecessor (except possibly for n_1)
- exactly one successor (except possibly for n_k)

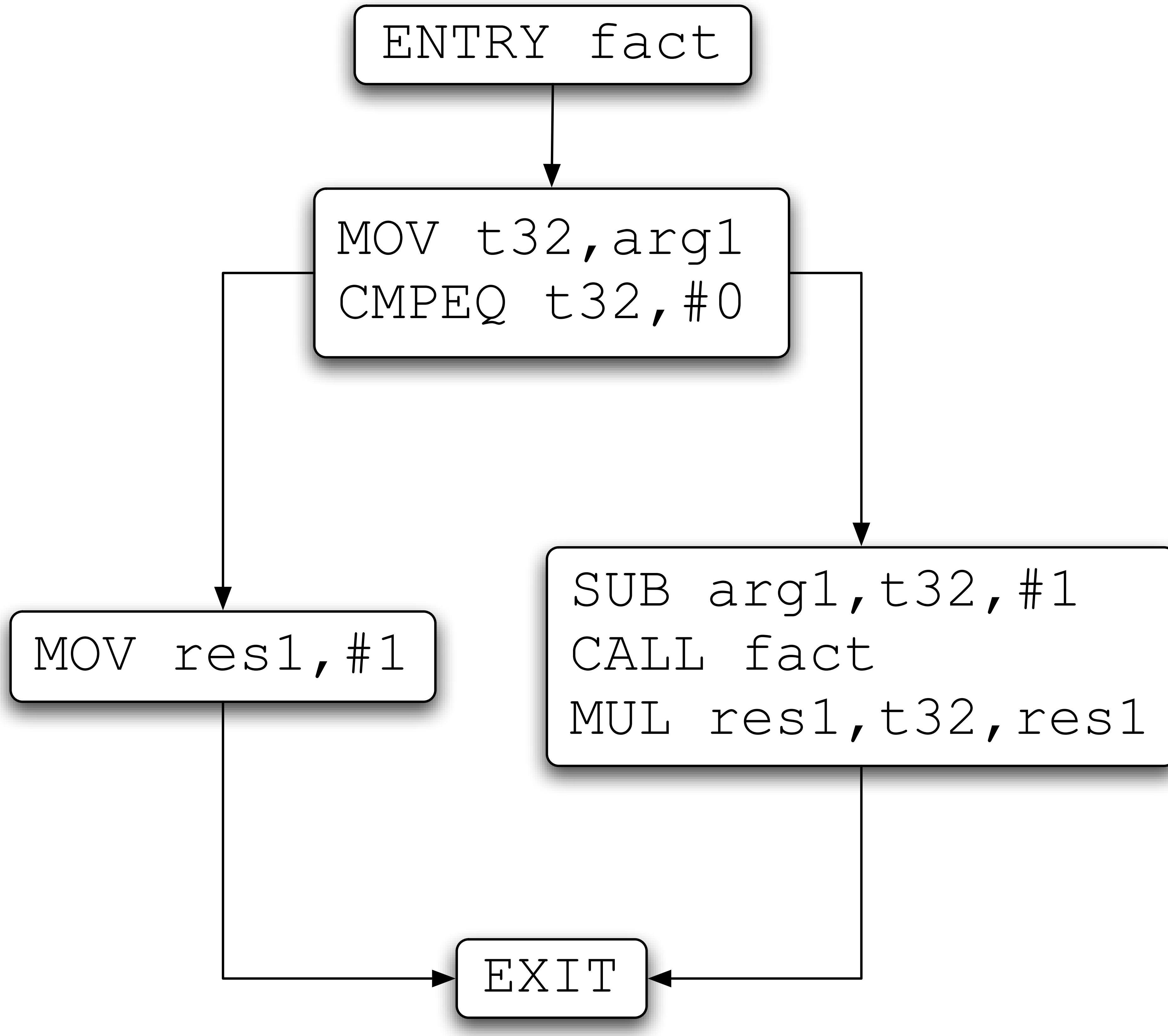
Basic blocks



Basic blocks



Basic blocks



Basic blocks

A basic block doesn't contain any interesting control flow.

Basic blocks

Reduce time and space requirements
for analysis algorithms
by calculating and storing data flow information

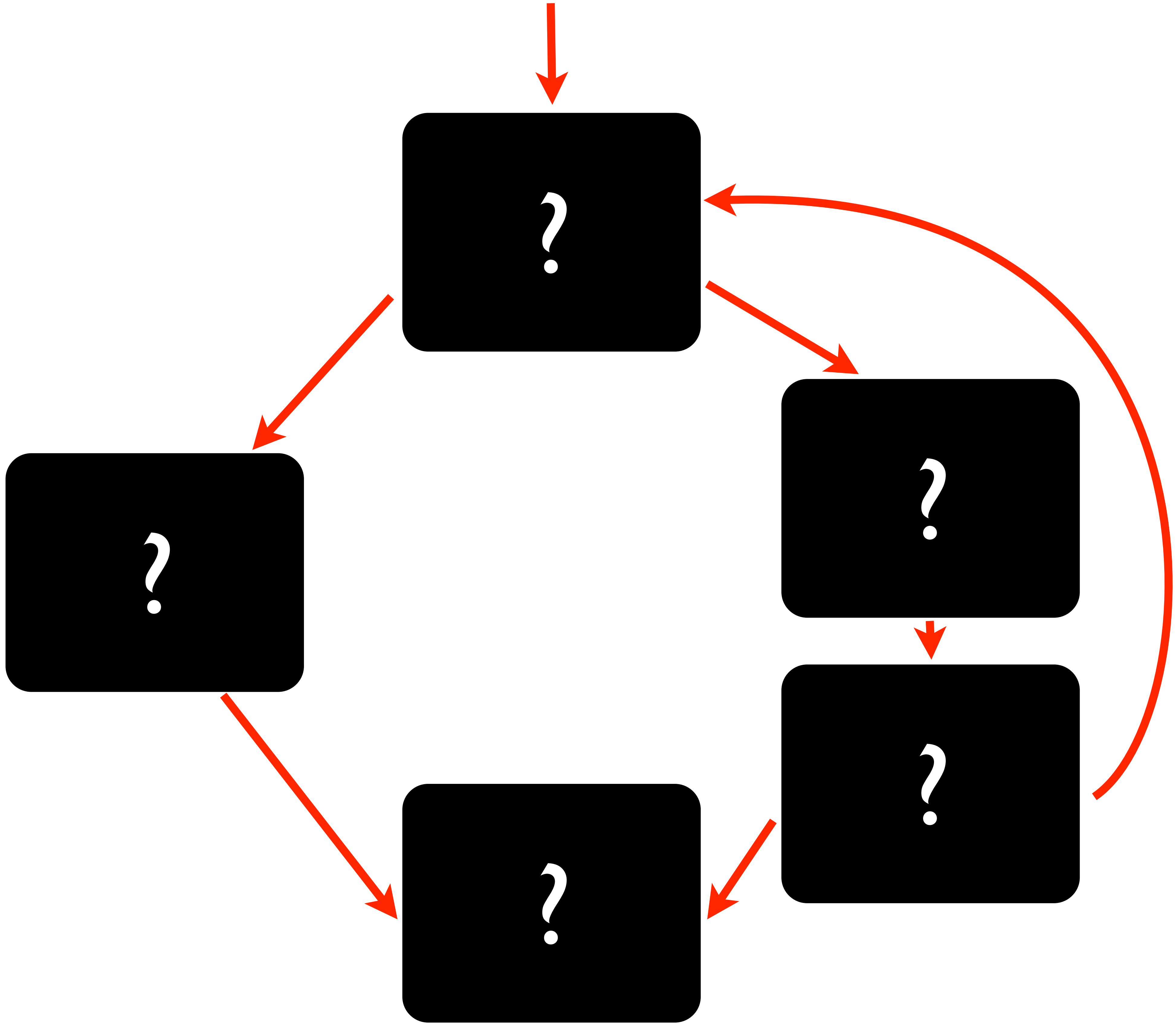
once per block

(and recomputing within a block if required)
instead of

once per instruction.

Basic blocks

Basic blocks



Types of analysis
(and hence optimisation)

Scope:

- Within basic blocks (“local” / “peephole”)
- Between basic blocks (“global” / “intra-procedural”)
  - e.g. live variable analysis, available expressions
- Whole program (“inter-procedural”)
  - e.g. unreachable-procedure elimination

Peephole optimisation

```
ADD t32,arg1,#1
MOV r0,r1
MOV r1,r0
MUL t33,r0,t32
```

matches

replace

```
MOV x,y
MOV y,x
```

with

```
MOV x,y
```

```
ADD t32,arg1,#1
MOV r0,r1
MUL t33,r0,t32
```

Types of analysis
(and hence optimisation)

Type of information:

- Control flow
  - Discovering control structure (basic blocks, loops, calls between procedures)
- Data flow
  - Discovering data flow structure (variable uses, expression evaluation)

Finding basic blocks

1. Find all the instructions which are *leaders*:
    - the first instruction is a leader;
    - the target of any branch is a leader; and
    - any instruction immediately following a branch is a leader.
2. For each leader, its basic block consists of itself and all instructions up to the next leader.

Finding basic blocks

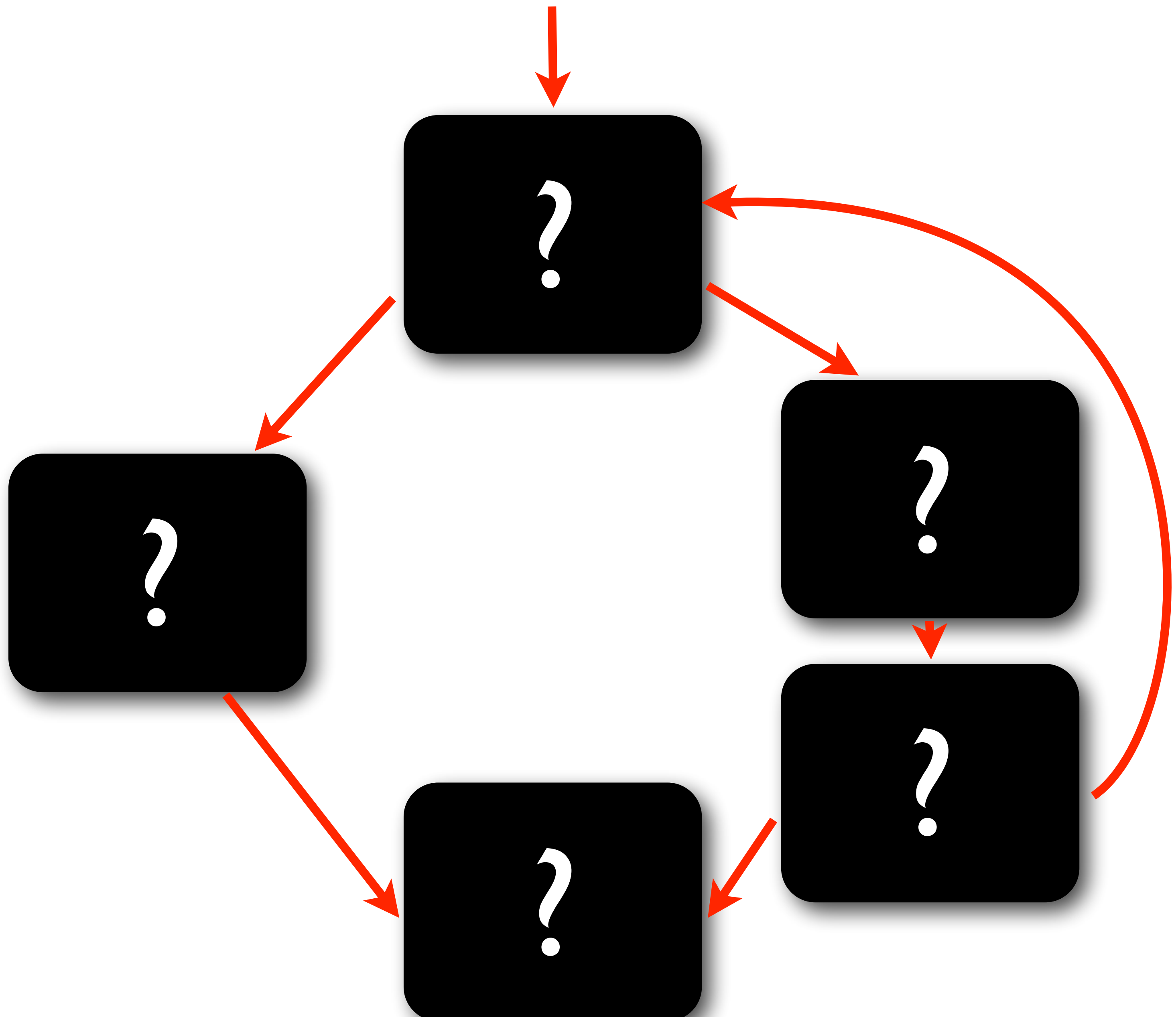
```
       ENTRY fact
       MOV t32,arg1
       CMPEQ t32,#0,lab1
       SUB arg1,t32,#1
       CALL fact
       MUL res1,t32,res1
       EXIT
lab1:  MOV res1,#1
       EXIT
```

Summary

- Structure of an optimising compiler
- Why optimise?
- Optimisation = Analysis + Transformation
- 3-address code
- Flowgraphs
- Basic blocks
- Types of analysis
- Locating basic blocks

Lecture 2
Unreachable-code &
-procedure elimination

Control-flow analysis



Discovering information about how *control* (e.g. the program counter) may move through a program.

Intra-procedural analysis

An *intra-procedural* analysis collects information about the code inside a single procedure.

We may repeat it many times (i.e. once per procedure), but information is only propagated within the boundaries of each procedure, not between procedures.

One example of an intra-procedural control-flow optimisation (an analysis and an accompanying transformation) is *unreachable-code elimination*.

Dead vs. unreachable code

```
int f(int x, int y) {
   int z = x * y;   DEAD
   return x + y;
}
```

Dead code computes unused values.
(Waste of time.)

Dead vs. unreachable code

```
int f(int x, int y) {
   return x + y;
   int z = x * y;
}
```

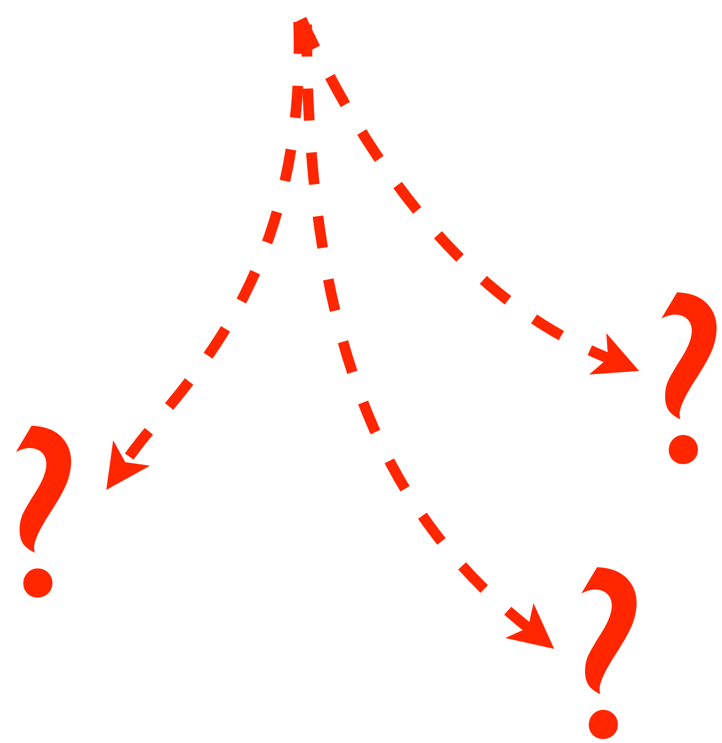
UNREACHABLE

Unreachable code cannot possibly be executed.
(Waste of space.)

Dead vs. unreachable code

Deadness is a *data-flow* property:
“May this data ever arrive anywhere?”

```
int f(int x, int y) {
    int z = x * y;
    ⋮
```



Dead vs. unreachable code

Unreachability is a *control-flow* property:
"May control ever arrive here?"

```
    ⋮
    int z = x * y;
}
```

Safety of analysis

```
int f(int x, int y) {
   if (g(x)) {
      int z = x * y;
   }
   return x + y;
}
```

UNREACHABLE?

```
bool g(int x) {
   return false;
}
```

✓

Safety of analysis

```
int f(int x, int y) {
   if (g(x)) {
      int z = x * y;
   }
   return x + y;
}
```

UNREACHABLE?

```
bool g(int x) {
   return ...x...;
}
```

?

Safety of analysis

```
int f(int x, int y) {
  if (g(x)) {
    int z = x * y;
  }
  return x + y;
}
```

UNREACHABLE?

In general, this is undecidable.
(Arithmetic is undecidable; cf. halting problem.)

Safety of analysis

- Many interesting properties of programs are undecidable and cannot be computed precisely...
- ...so they must be *approximated*.
- A broken program is much worse than an inefficient one...
- ...so we must err on the side of *safety*.

Safety of analysis

- If we decide that code is unreachable then we may do something dangerous (e.g. remove it!)...
- ...so the safe strategy is to *overestimate* reachability.
- If we can't easily tell whether code is reachable, we just assume that it is. (This is conservative.)
- For example, we assume
  - both branches of a conditional are reachable
  - and that loops always terminate.

Safety of analysis

Naïvely,

```
if (false) {
  int z = x * y;
}
```

this instruction is reachable,

```
while (true) {
  // Code without 'break'
}
int z = x * y;
```

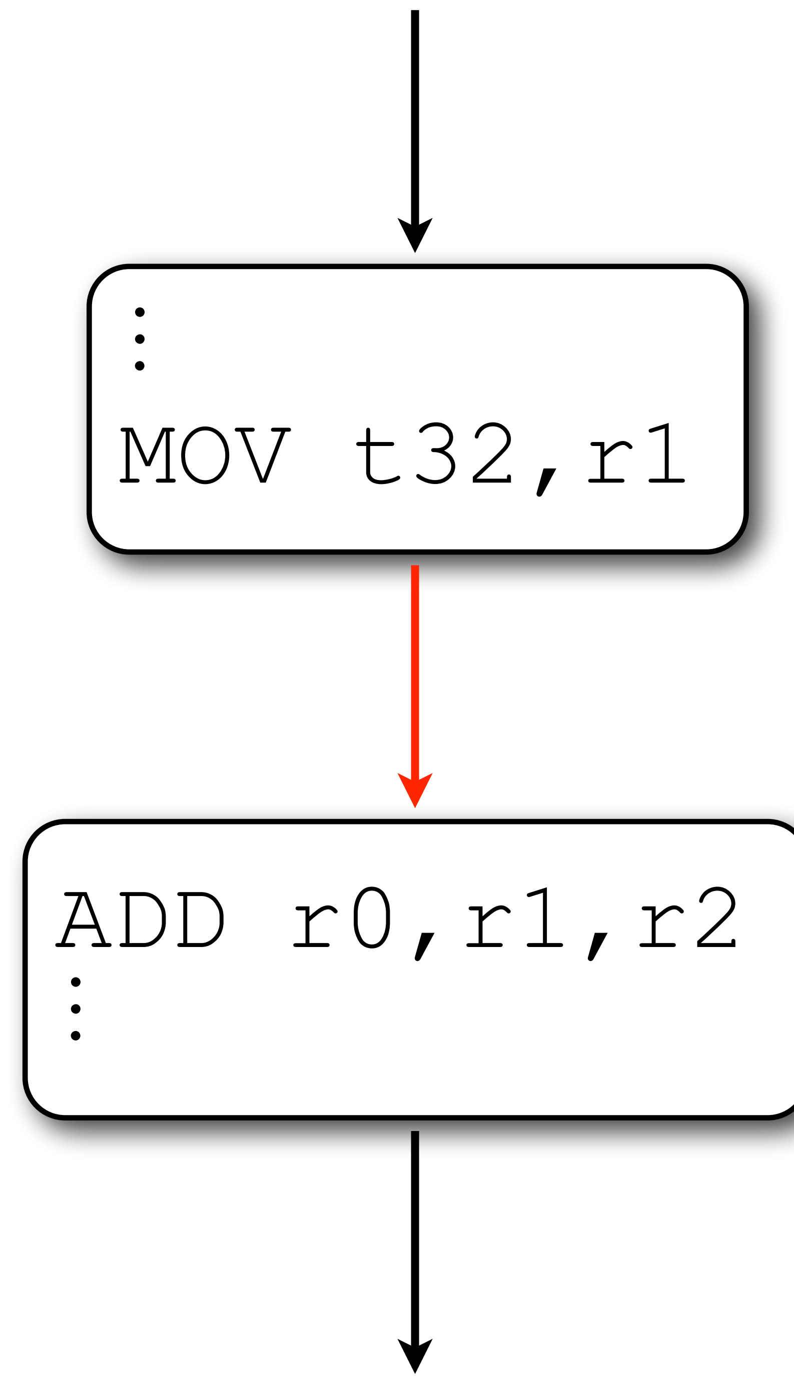
and so is this one.

Safety of analysis

Another source of uncertainty is encountered when constructing the original flowgraph: the presence of indirect branches (also known as "computed jumps").

Safety of analysis

```
       ⋮
       MOV t32,r1
       JMP lab1

       ⋮
lab1:  ADD r0,r1,r2
       ⋮
```

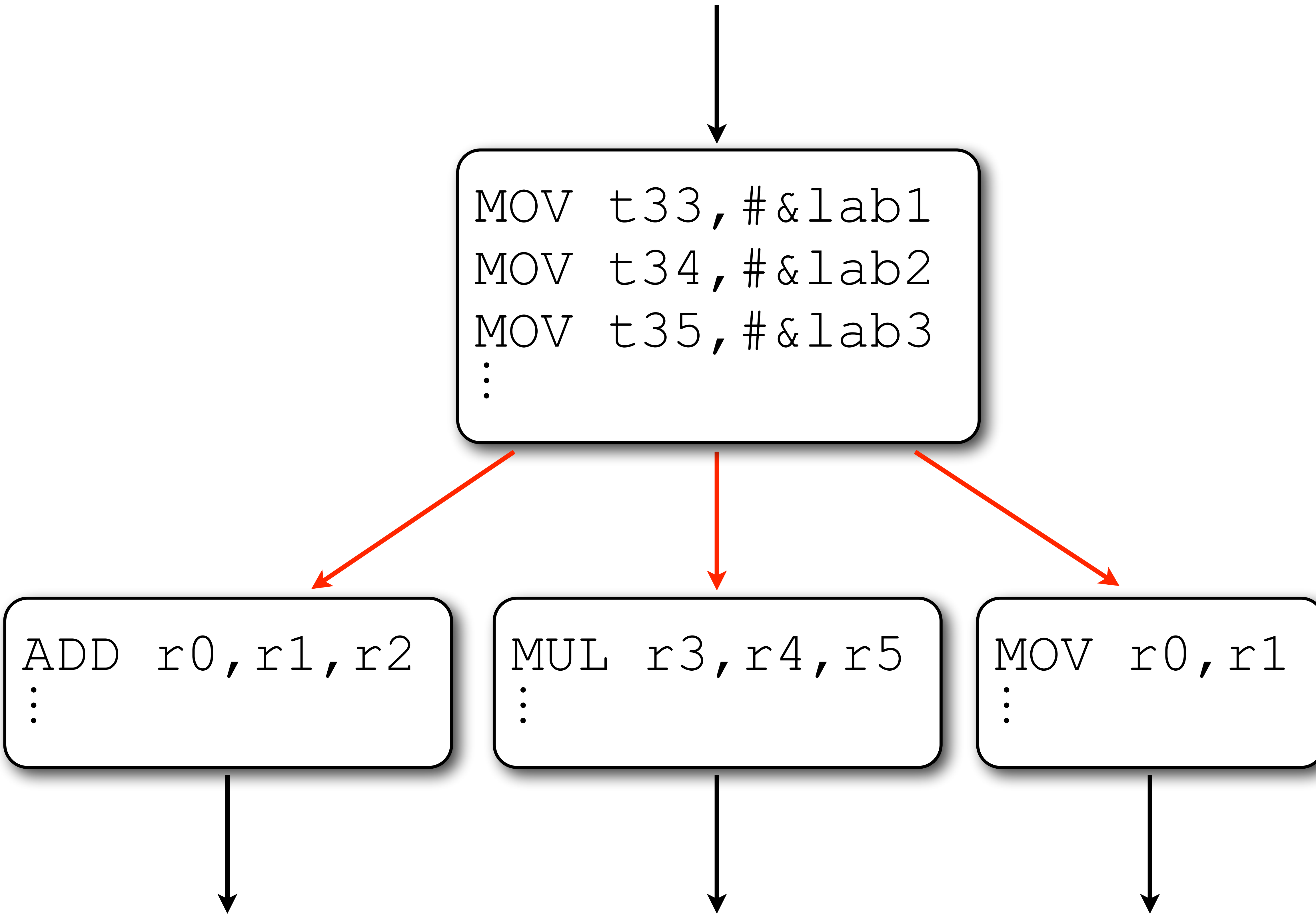


Safety of analysis

```
⋮
MOV t33,#&lab1
MOV t34,#&lab2
MOV t35,#&lab3

⋮
JMPI  t32

lab1: ADD r0,r1,r2
        ⋮
lab2: MUL r3,r4,r5
        ⋮
lab3: MOV r0,r1
        ⋮
```

?
?
?

Safety of analysis

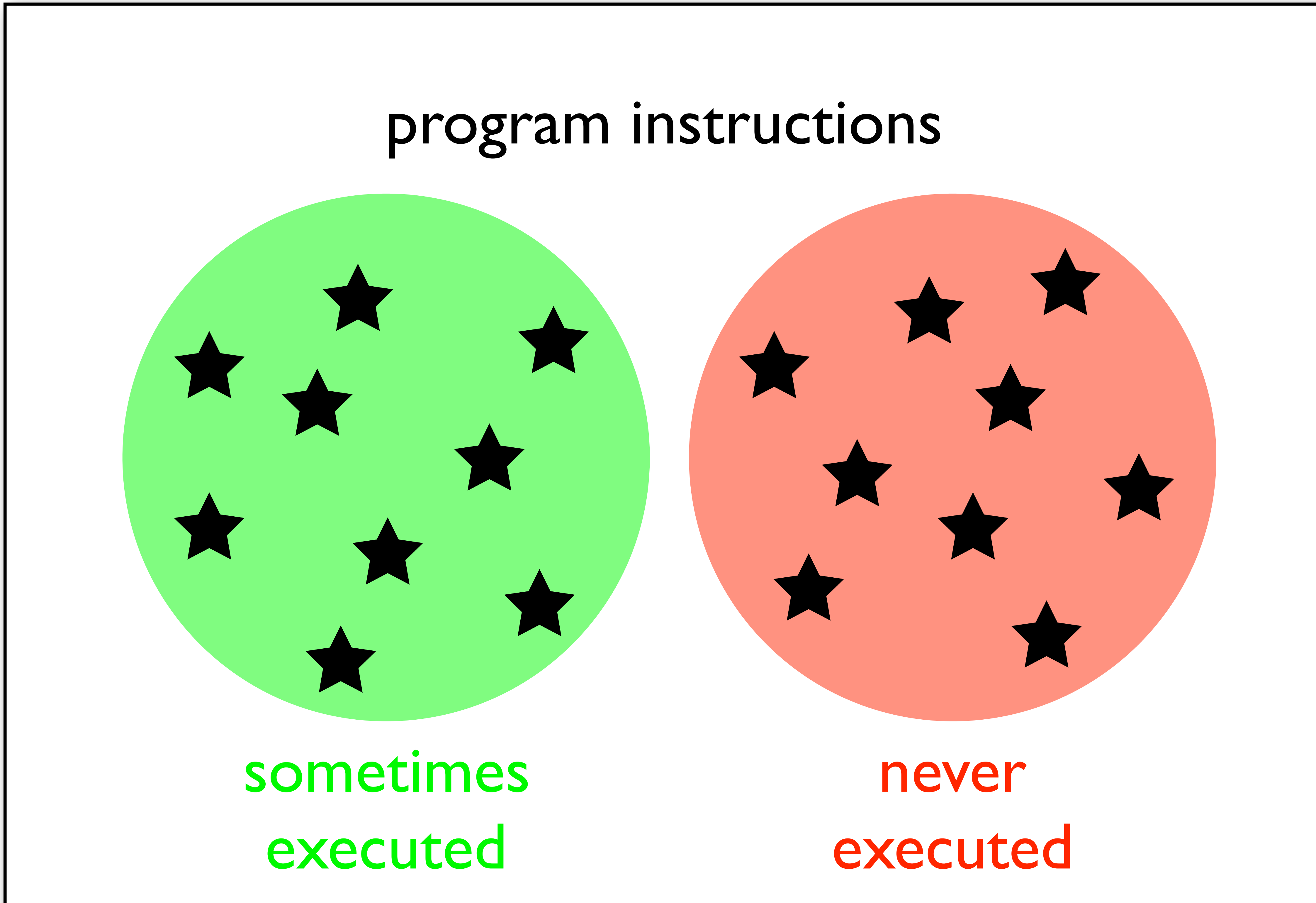


Safety of analysis

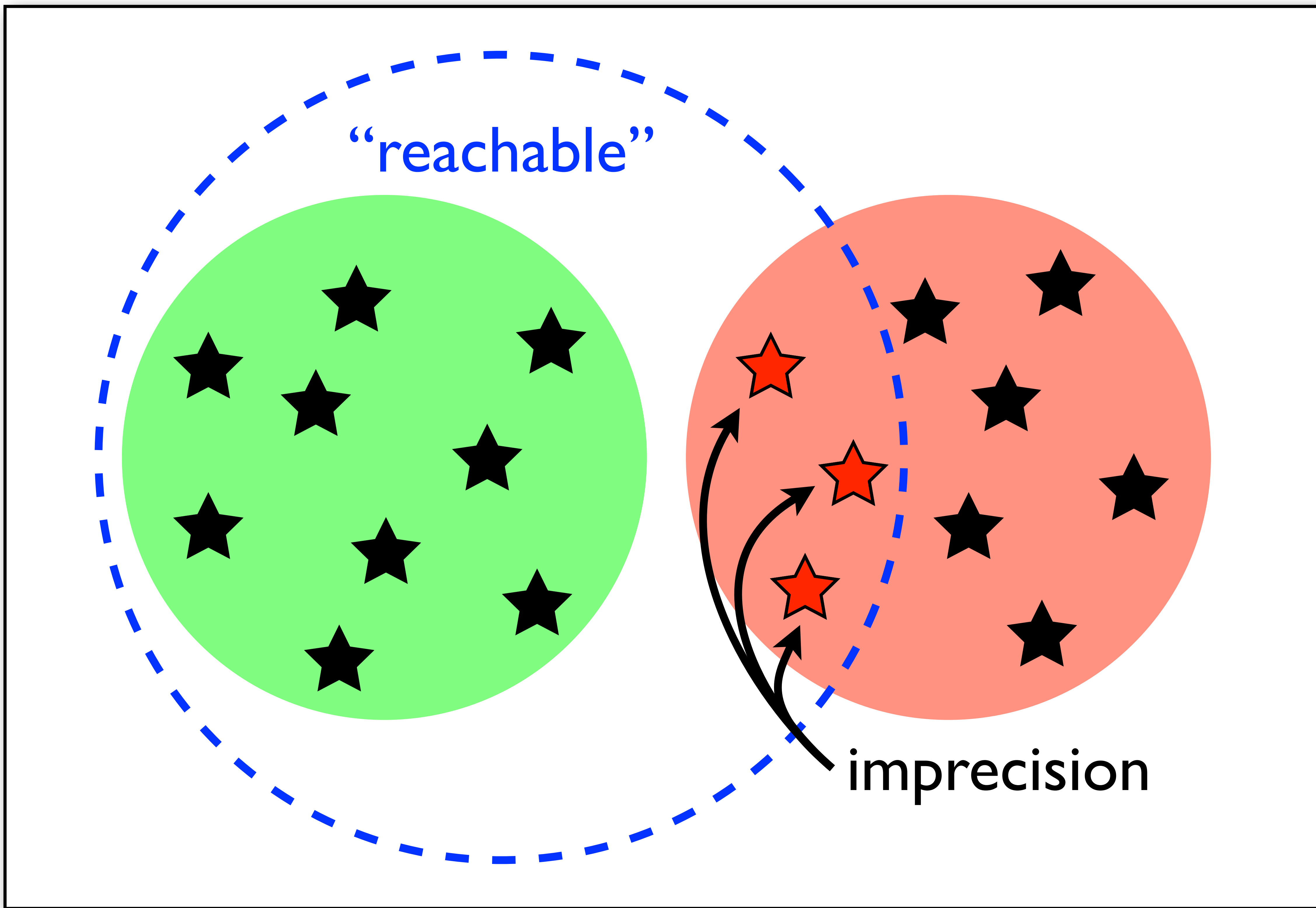
Again, this is a conservative *overestimation* of reachability.

In the worst-case scenario in which branch-address computations are completely unrestricted (i.e. the target of a jump could be absolutely anywhere), the presence of an indirect branch forces us to assume that *all* instructions are potentially reachable in order to guarantee safety.

Safety of analysis



Safety of analysis

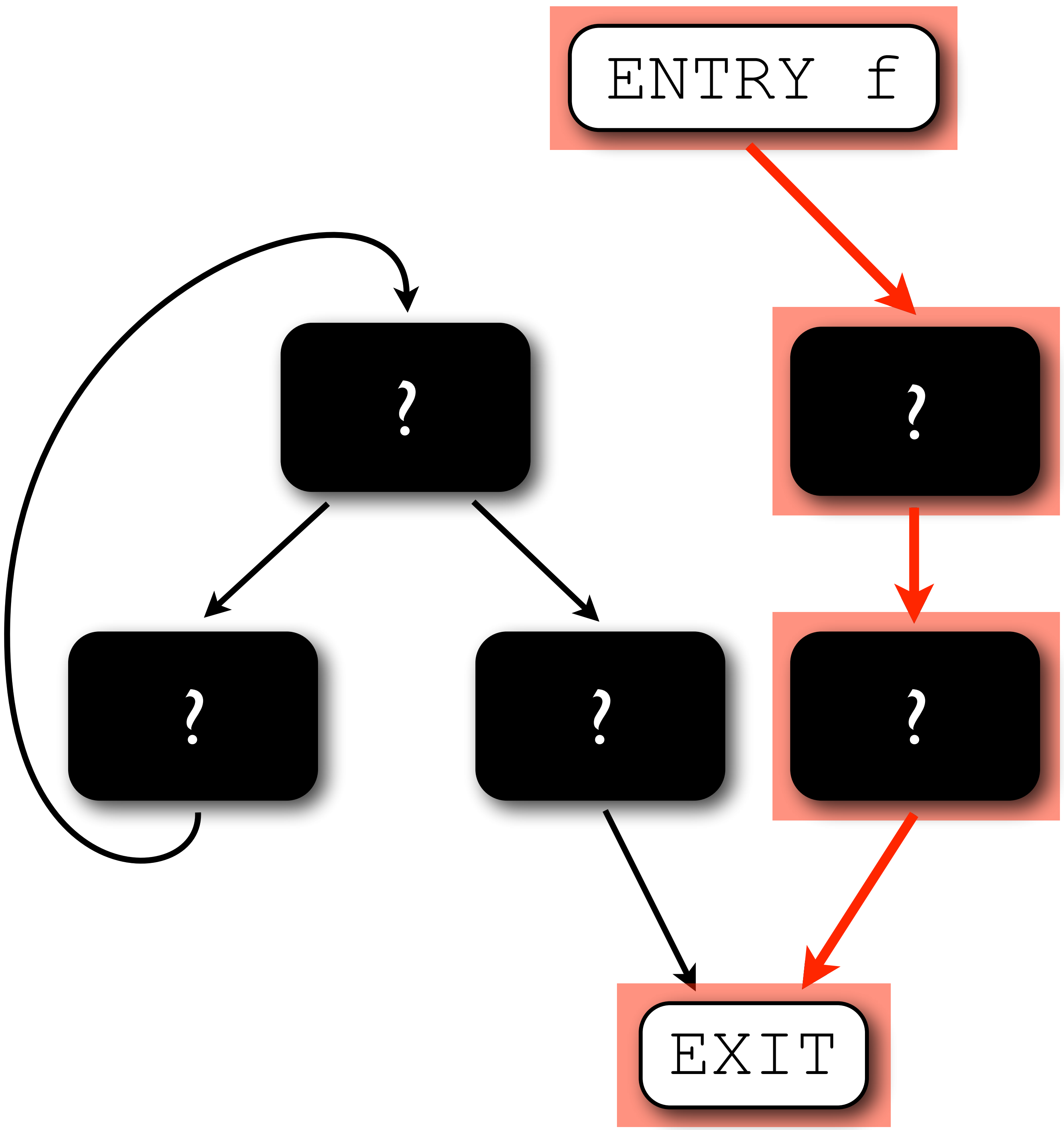


Unreachable code

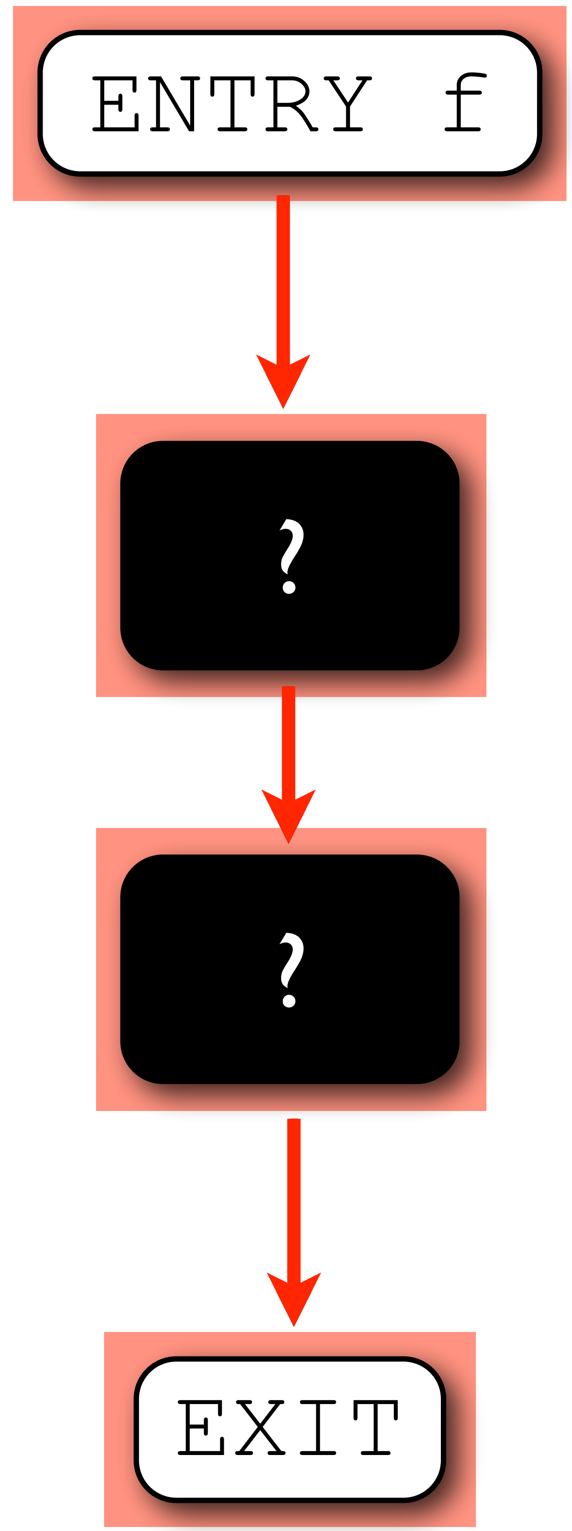
This naïve reachability analysis is simplistic, but has the advantage of corresponding to a very straightforward operation on the flowgraph of a procedure:

1. mark the procedure's entry node as reachable;
2. mark every successor of a marked node as reachable and repeat until no further marking is required.

Unreachable code



Unreachable code



Unreachable code

Programmers rarely write code which is
completely unreachable in this naïve sense.
Why bother with this analysis?

- Naïvely unreachable code may be introduced as a result of other optimising transformations.
- With a little more effort, we can do a better job.

Unreachable code

Obviously, if the conditional expression in an `if` statement is literally the constant "`false`", it's safe to assume that the statements within are unreachable.

```
if (false) {
    int z = x * y;
}
```

UNREACHABLE

But programmers never write code like that either.

Unreachable code

However, other optimisations might produce such code. For example, *copy propagation*:

```
bool debug = false;
⋮
if (debug) {
    int z = x * y;
}
```

Unreachable code

However, other optimisations might produce such code.
For example, *copy propagation*:

```
⋮
if (false) {
  int z = x * y;
}
```

UNREACHABLE

Unreachable code

We can try to spot (slightly) more subtle things too.

- `if (!true) {... }`
- `if (false && ...) {... }`
- `if (x != x) {... }`
- `while (true) {... } ...`
- ...

Unreachable code

Note, however, that the reachability analysis no longer consists simply of checking whether any paths to an instruction *exist* in the flowgraph, but whether any of the paths to an instruction are actually *executable*.

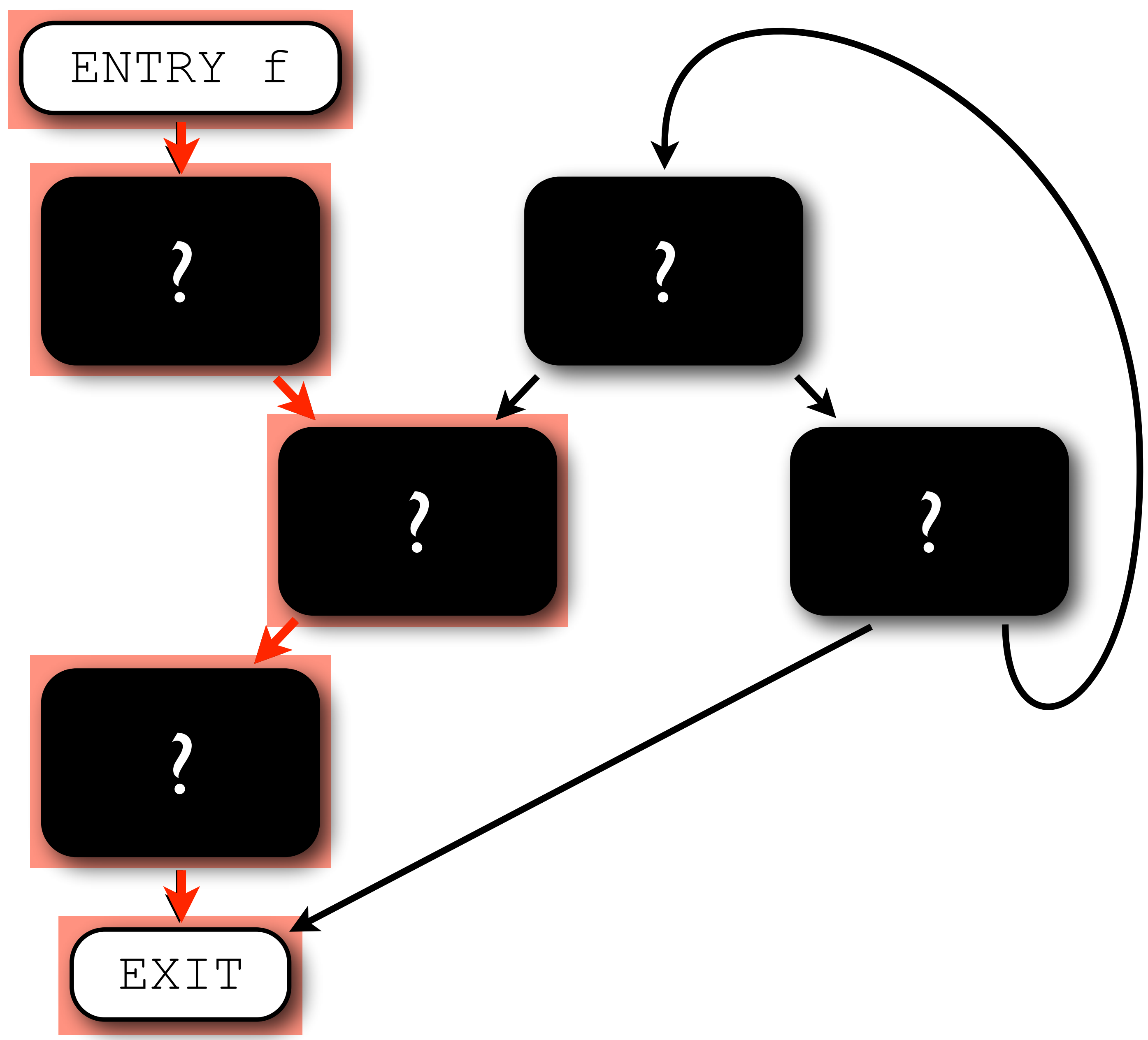
With more effort we may get arbitrarily clever at spotting non-executable paths in particular cases, but in general the undecidability of arithmetic means that we cannot always spot them all.

Unreachable code

Although unreachable-code elimination can only make a program *smaller*, it may enable other optimisations which make the program *faster*.

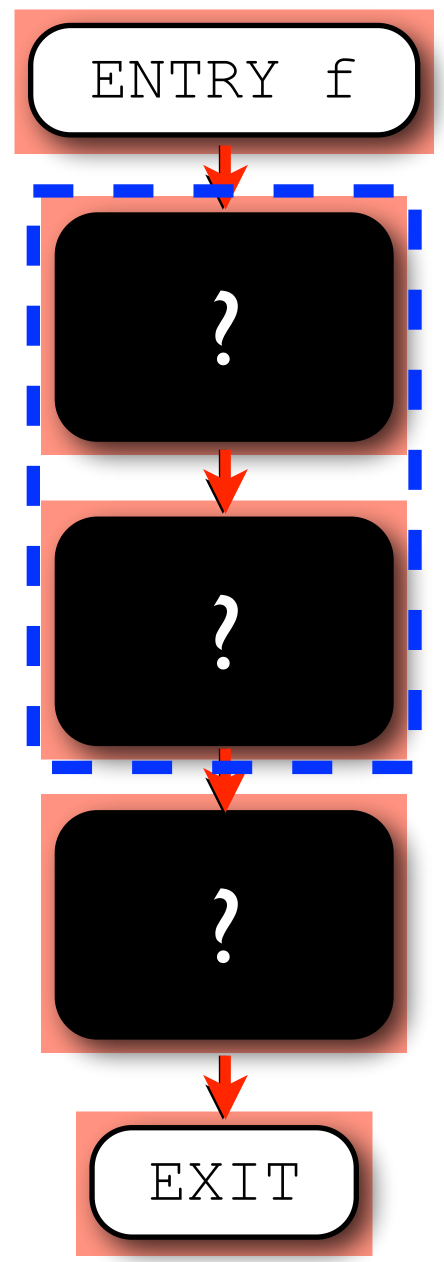
Unreachable code

For example, *straightening* is an optimisation which can eliminate jumps between basic blocks by coalescing them:



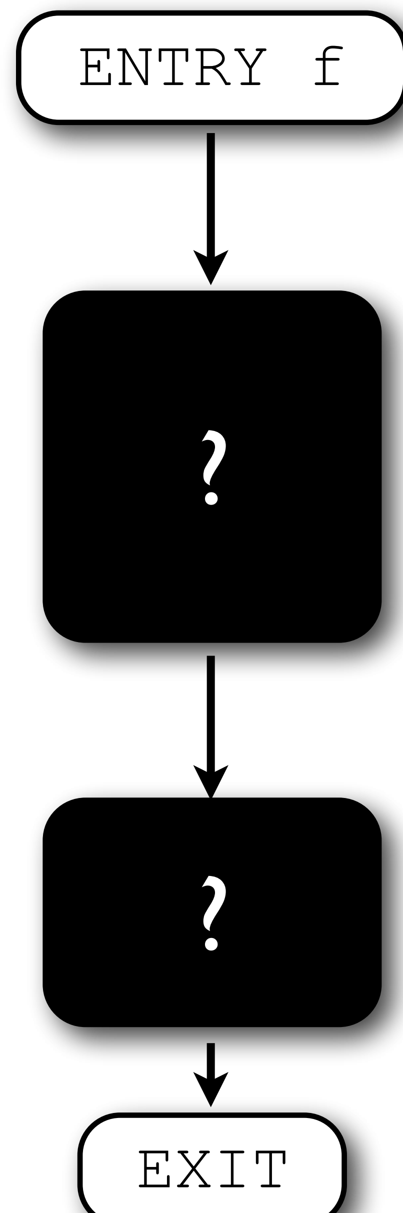
Unreachable code

For example, *straightening* is an optimisation which can eliminate jumps between basic blocks by coalescing them:



Unreachable code

For example, *straightening* is an optimisation which can eliminate jumps between basic blocks by coalescing them:



Straightening has removed a branch instruction, so the new program will execute faster.

Inter-procedural analysis

An *inter-procedural* analysis collects information about an entire program.

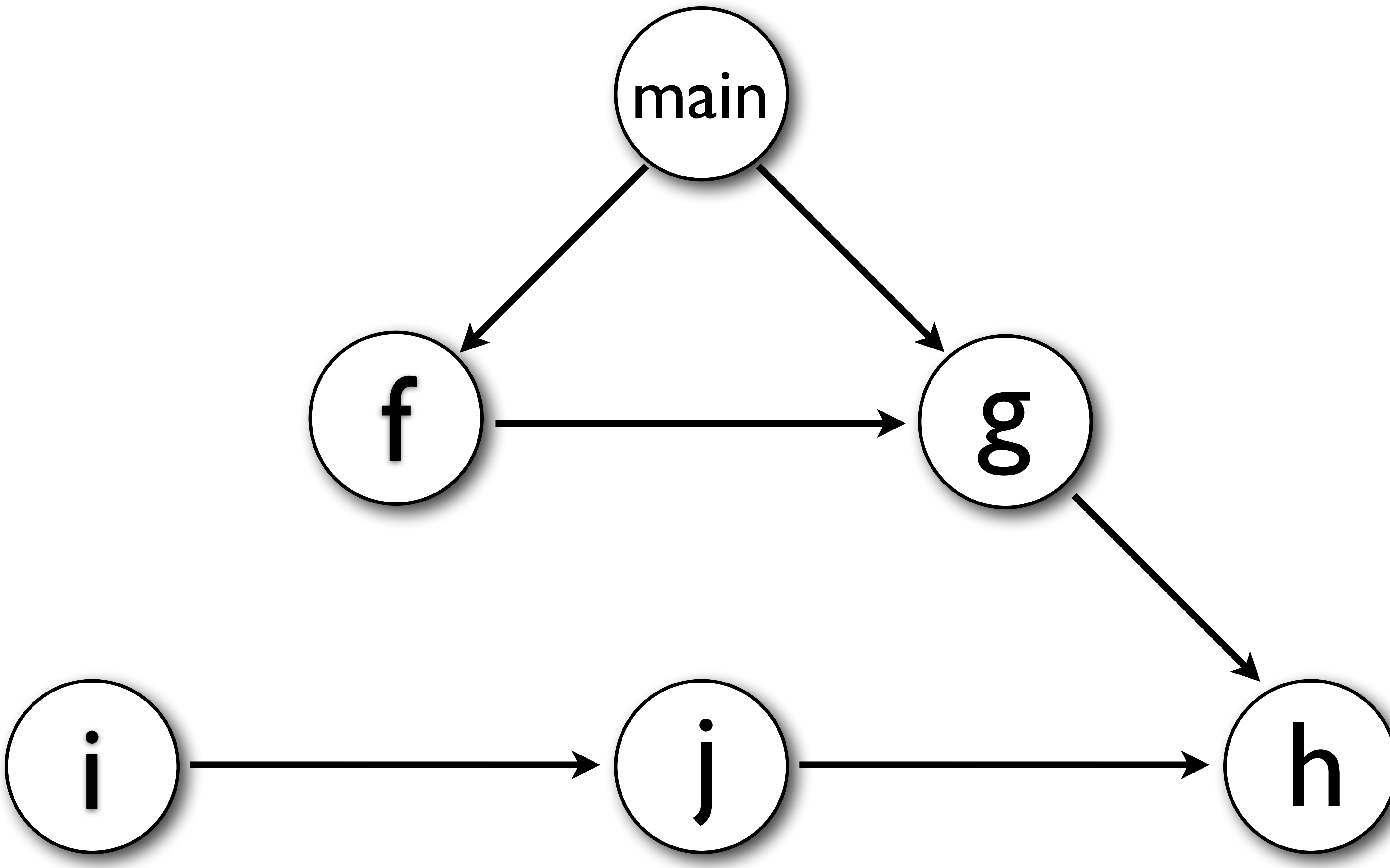
Information is collected from the instructions of each procedure and then propagated between procedures.

One example of an inter-procedural control-flow optimisation (an analysis and an accompanying transformation) is *unreachable-procedure elimination*.

Unreachable procedures

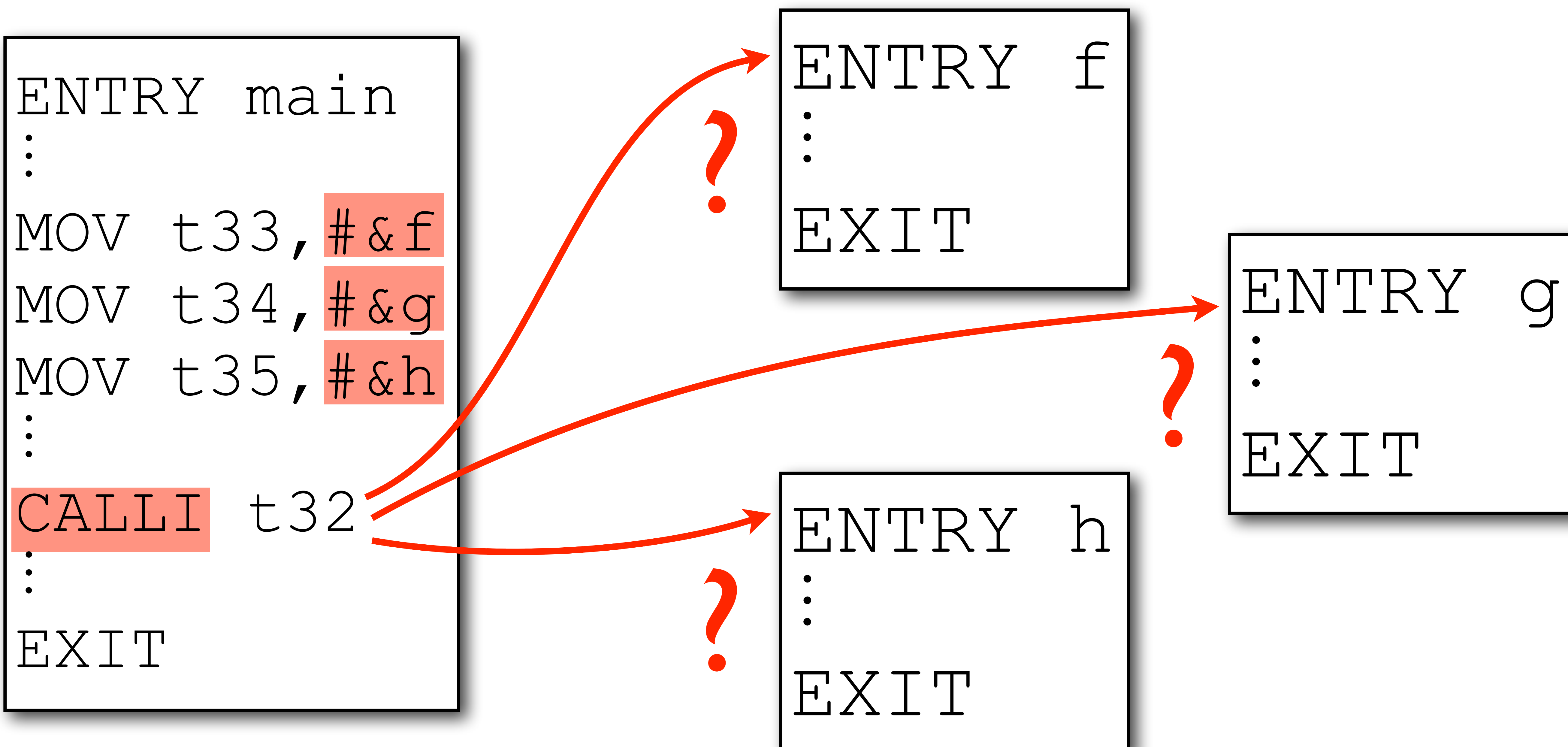
Unreachable-*procedure* elimination is very similar in spirit to unreachable-*code* elimination, but relies on a different data structure known as a *call graph*.

Call graphs



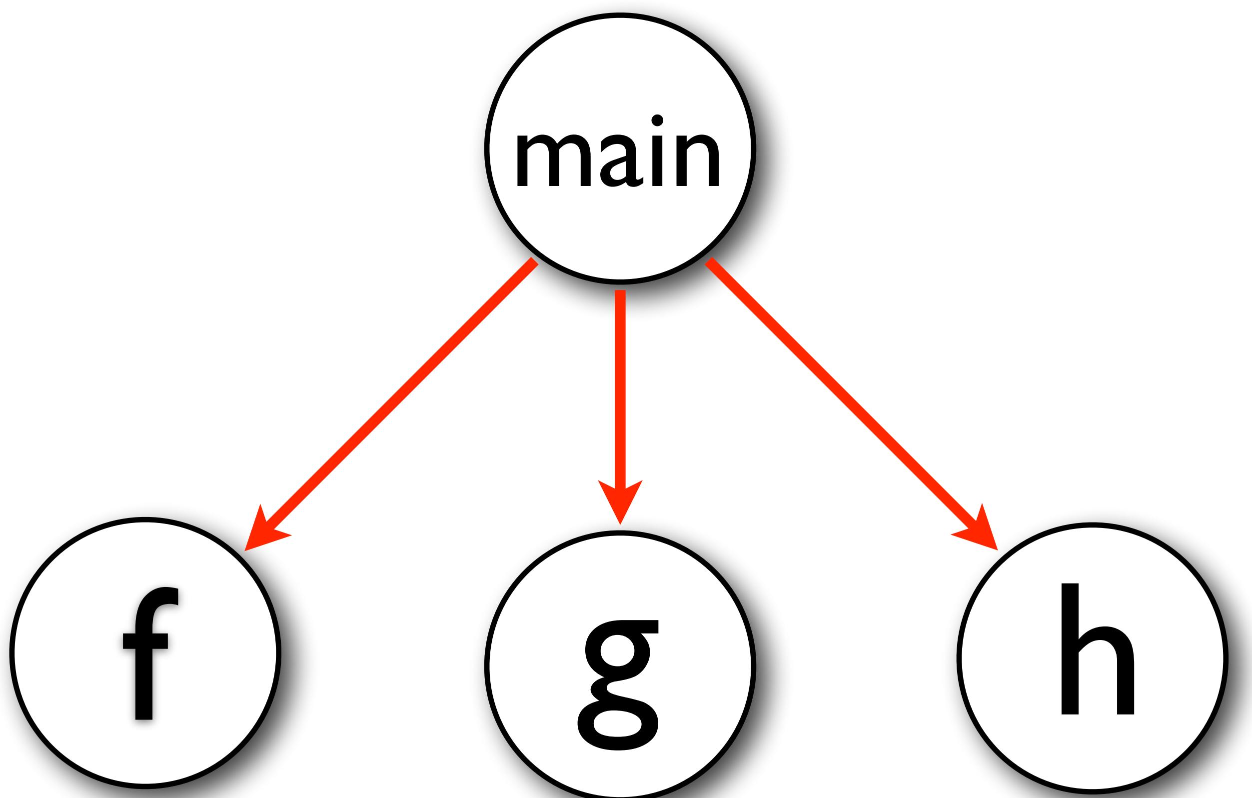
Call graphs

Again, the precision of the graph is compromised in the presence of *indirect calls*.



Call graphs

Again, the precision of the graph is compromised in the presence of *indirect calls*.



And as before, this is a *safe* overestimation of reachability.

Call graphs

In general, we assume that a procedure containing an indirect call has *all* address-taken procedures as successors in the call graph — i.e., it could call any of them.

This is obviously *safe*; it is also obviously *imprecise*.

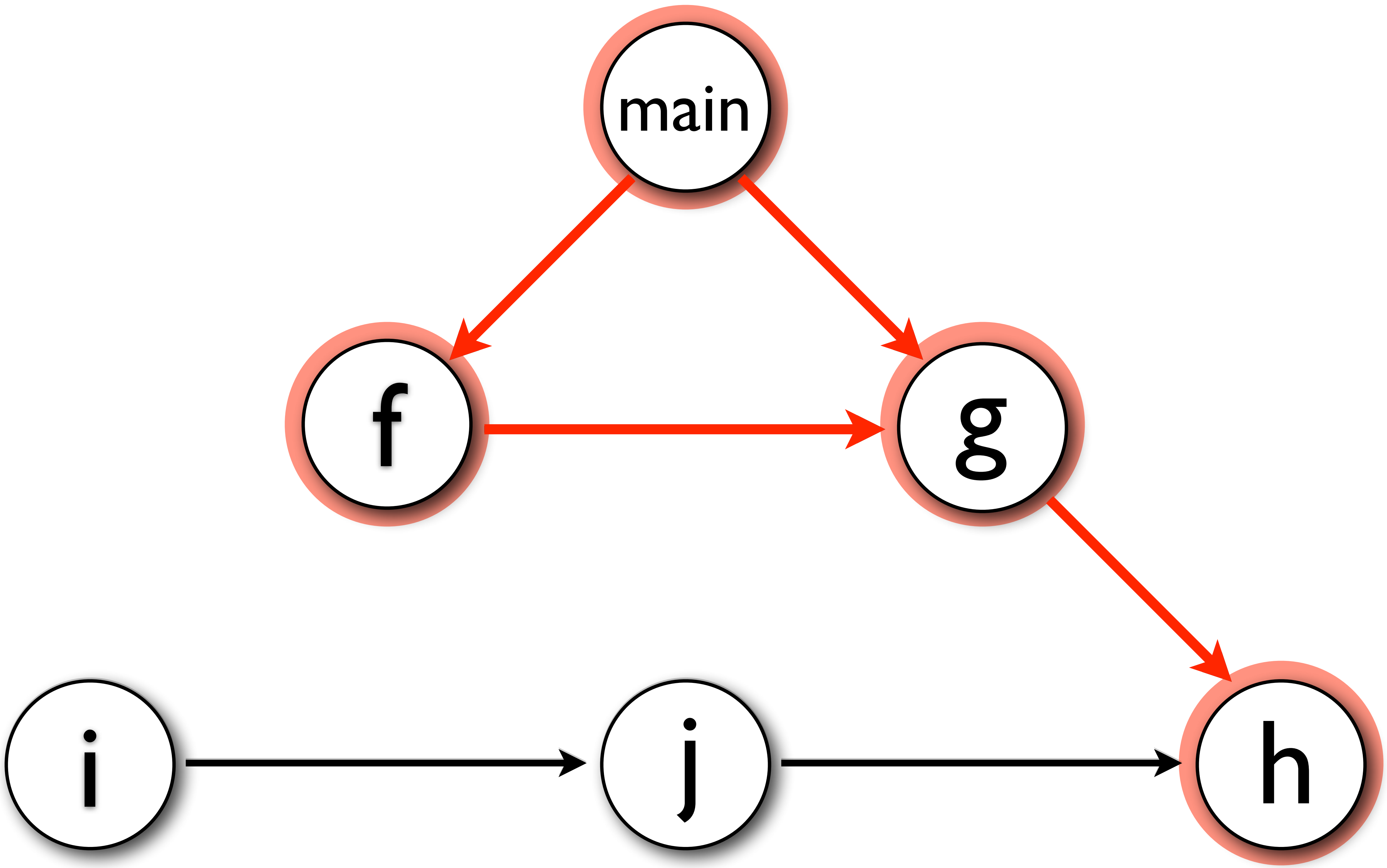
As before, it might be possible to do better by application of more careful methods (e.g. tracking data-flow of procedure variables).

Unreachable procedures

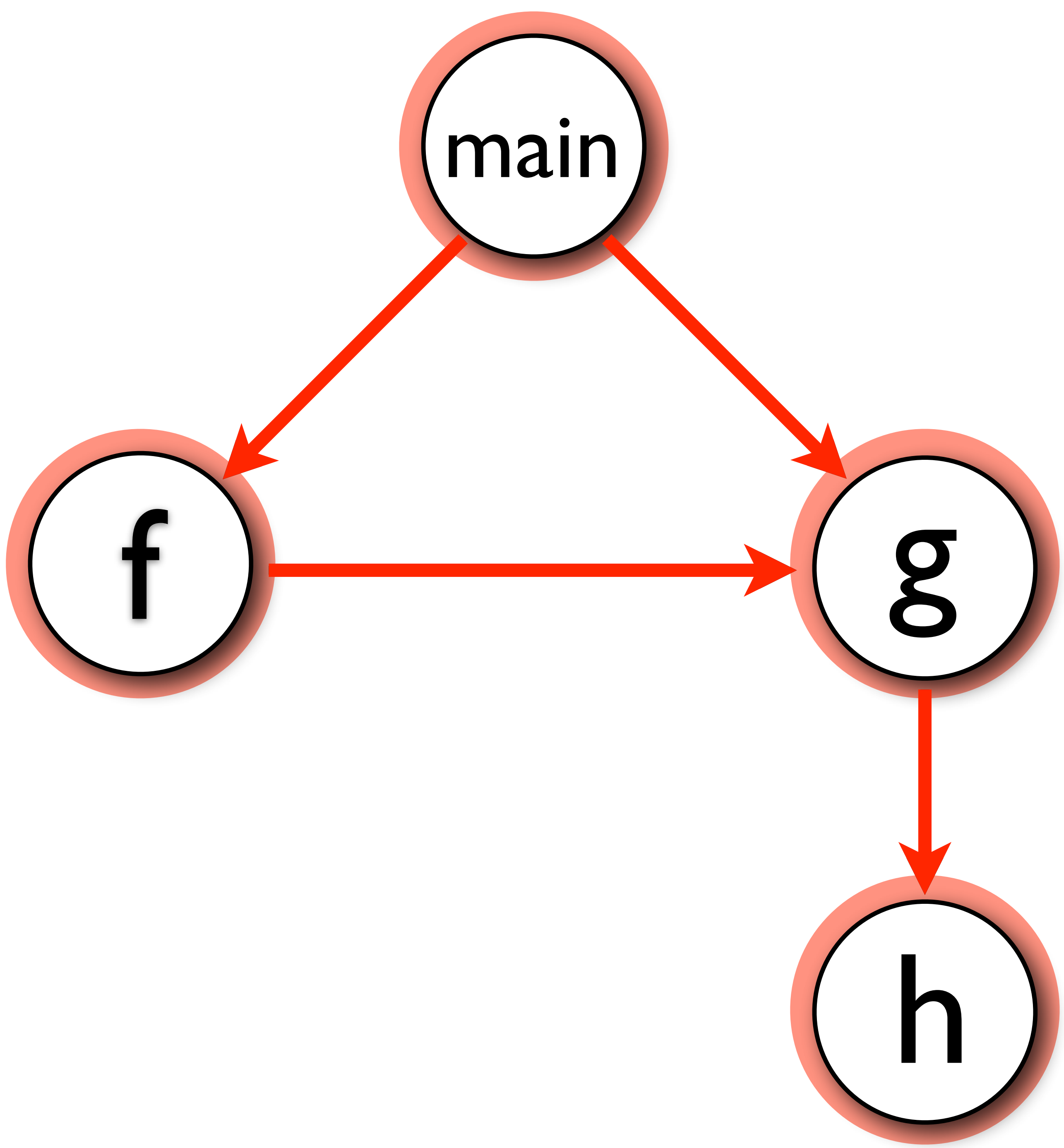
The reachability analysis is virtually identical to that used in unreachable-*code* elimination, but this time operates on the call graph of the entire program (vs. the flowgraph of a single procedure):

1. mark procedure `main` as callable;
2. mark every successor of a marked node as callable and repeat until no further marking is required.

Unreachable procedures



Unreachable procedures



Safety of transformations

- All instructions/procedures to which control may flow at execution time will *definitely* be marked by the reachability analyses...
- ...but not vice versa, since some marked nodes might never be executed.
- Both transformations will *definitely* not delete any instructions/procedures which are needed to execute the program...
- ...but they might leave others alone too.

If simplication

Empty `then` in `if-then`

```
if (f(x)) {
}
```

(Assuming that `f` has no side effects.)

If simplication

Empty `else` in `if-then-else`

```
if (f(x)) {
    z = x * y;
} else {
}
```

If simplication

Empty `then` in `if-then-else`

```
if (!f(x)) {
} else {
  z = x * y;
}
```

If simplication

Empty `then` and `else` in `if-then-else`

```
if (f(x)) {
} else {
}
```

If simplication

Constant condition

```
if (true) {
  z = x * y;
}
```

If simplication

Nested `if` with common subexpression

```
if (x > 3 && t) {
   ⋮
   if (x > 3) {
      z = x * y;
   } else {
      z = y - x;
   }
}
```

Loop simplification

```
int x = 0;
int i = 0;
while (i < 4) {
  i = i + 1;
  x = x + i;
}
```

Loop simplification

```
int x = 10;
int i = 4;
```

Summary

- Control-flow analysis operates on the control structure of a program (flowgraphs and call graphs)
- Unreachable-*code* elimination is an *intra*-procedural optimisation which reduces code size
- Unreachable-*procedur*e elimination is a similar, *inter*-procedural optimisation making use of the program's call graph
- Analyses for both optimisations must be imprecise in order to guarantee safety

Lecture 3
Live variable analysis

Data-flow analysis

```
MOV t32,arg1
MOV t33,arg2
ADD t34,t32,t33
MOV t35,arg3
MOV t36,arg4
ADD t37,t35,t36
MUL res1,t34,t37
```

Discovering information about how *data* (i.e. variables and their values) may move through a program.

Motivation

Programs may contain

- code which gets executed but which has no useful effect on the program's overall result;
- occurrences of variables being used before they are defined; and
- many variables which need to be allocated registers and/or memory locations for compilation.

The concept of *variable liveness* is useful in dealing with all three of these situations.

Liveness

Liveness is a data-flow property of variables:
"Is the value of this variable needed?" (cf. dead code)

```
int f(int x, int y) {
  int z = x * y;
  ⋮
```

? ? ?

Liveness

At each instruction, each variable in the program is either live or dead.

We therefore usually consider liveness from an instruction's perspective: each instruction (or node of the flowgraph) has an associated *set* of live variables.

```
   ⋮
n: int z = x * y;
   return s + t;
```

$live(n) = \{$ `s`, `t`, `x`, `y` $\}$

Semantic vs. syntactic

There are two kinds of variable liveness:

- Semantic liveness
- Syntactic liveness

Semantic vs. syntactic

A variable *x* is *semantically* live at a node *n* if there is *some execution sequence* starting at *n* whose (externally observable) behaviour can be affected by changing the value of *x*.

```
int x = y * z;
:
return x;
```

`x` LIVE

Semantic vs. syntactic

A variable *x* is *semantically* live at a node *n* if there is *some execution sequence* starting at *n* whose (externally observable) behaviour can be affected by changing the value of *x*.

```
int x = y * z;     x DEAD
⋮
x = a + b;
⋮
return x;
```

Semantic vs. syntactic

Semantic liveness is concerned with
the *execution behaviour* of the program.

This is undecidable in general.
(e.g. Control flow may depend upon arithmetic.)

Semantic vs. syntactic

A variable is *syntactically* live at a node if there is a path to the exit of the flowgraph along which its value may be used before it is redefined.

Syntactic liveness is concerned with properties of the *syntactic structure* of the program.

Of course, this *is* decidable.

So what's the difference?

Semantic vs. syntactic

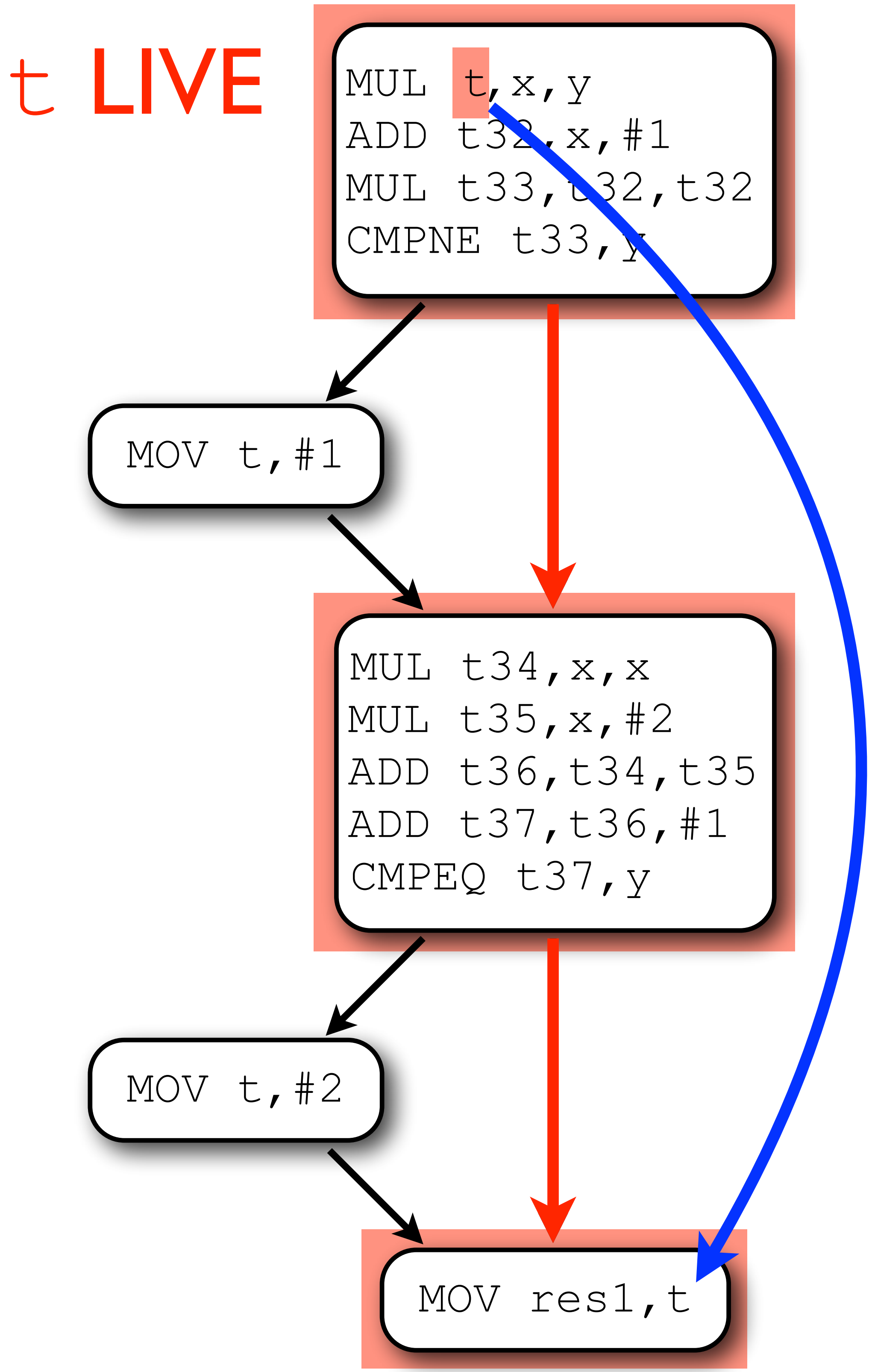
```
int t = x * y;   t DEAD
if ((x+1)*(x+1) == y) {
  t = 1;
}
if (x*x + 2*x + 1 != y) {
  t = 2;
}
return t;
```

Semantically: one of the conditions will be true, so on every execution path `t` is redefined before it is returned. The value assigned by the first instruction is never used.

Semantic vs. syntactic

```
        MUL t,x,y
        ADD t32,x,#1
        MUL t33,t32,t32
        CMPNE t33,y,lab1
        MOV t,#1
lab1:   MUL t34,x,x
        MUL t35,x,#2
        ADD t36,t34,t35
        ADD t37,t36,#1
        CMPEQ t37,y,lab2
        MOV t,#2
lab2:   MOV res1,t
```

Semantic vs. syntactic



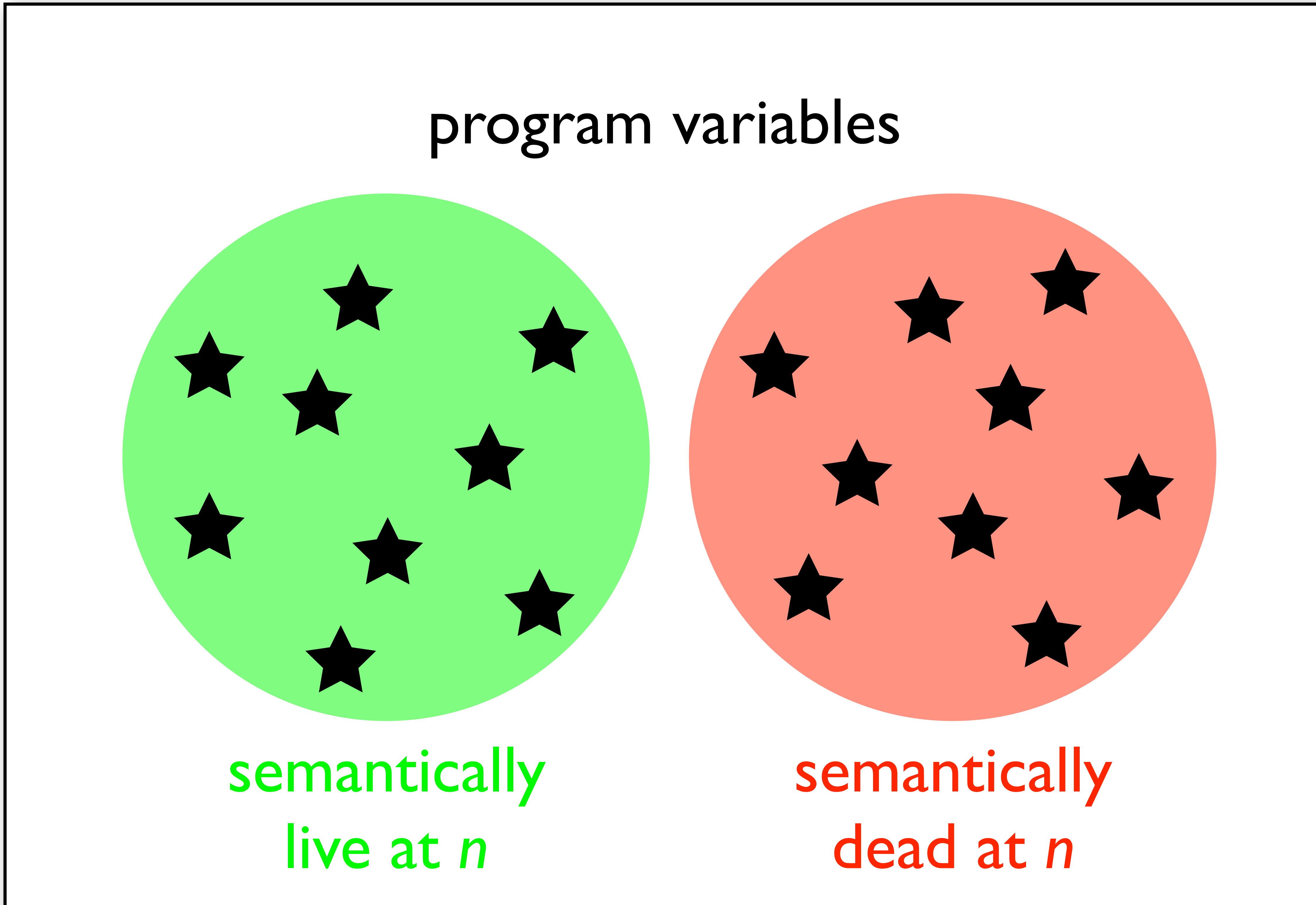
On *this* path through the flowgraph, `t` is not redefined before it's used, so `t` is *syntactically* live at the first instruction.

Note that this path never actually occurs during execution.

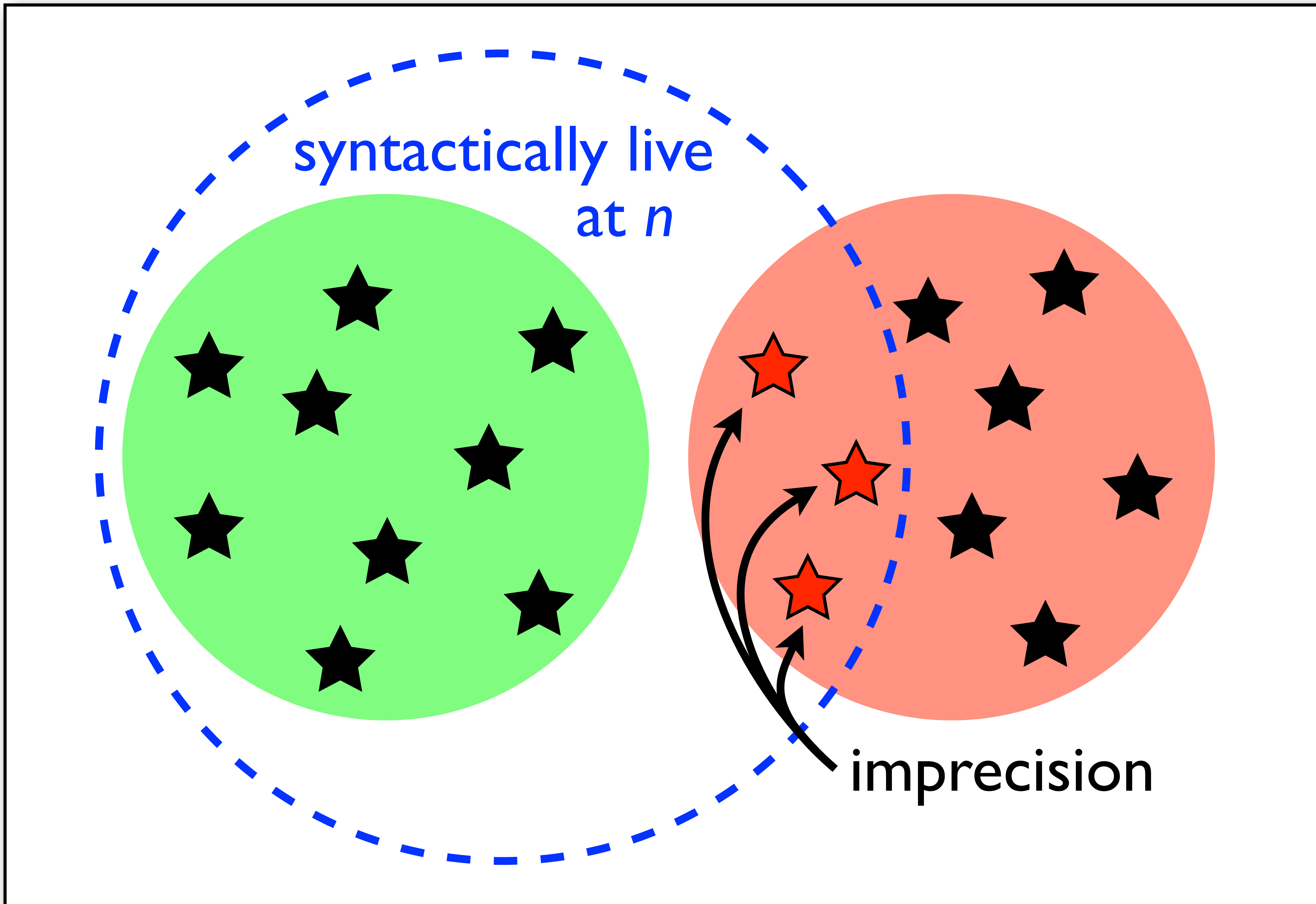
Semantic vs. syntactic

So, as we've seen before, *syntactic* liveness is a computable approximation of *semantic* liveness.

Semantic vs. syntactic



Semantic vs. syntactic



Semantic vs. syntactic

$$sem\text{-}live(n) \subseteq syn\text{-}live(n)$$

Using syntactic methods, we *safely overestimate* liveness.

Live variable analysis

LVA is a *backwards* data-flow analysis: usage information from *future* instructions must be propagated backwards through the program to discover which variables are live.

```
int f(int x, int y) {
    int z = x * y;
    ⋮
                print z;
int a = z*2;
              if (z > 5) {
```

Live variable analysis

Variable liveness flows (backwards) through the program in a continuous stream.

Each instruction has an effect on the liveness information as it flows past.

Live variable analysis

An instruction makes a variable live when it *references* (uses) it.

Live variable analysis

{ b, c, e, f }

```
a = b * c;
```
REFERENCE b, c

{ e, f }

```
d = e + 1;
```
REFERENCE e

{ f }

```
print f;
```
REFERENCE f

{ }

Live variable analysis

An instruction makes a variable dead
when it *defines* (assigns to) it.

Live variable analysis

{ }

```
a = 7;
```
DEFINE a

{ a }

```
b = 11;
```
DEFINE b

{ a, b }

```
c = 13;
```
DEFINE c

{ a, b, c }

Live variable analysis

We can devise functions *ref*(*n*) and *def*(*n*) which give the sets of variables referenced and defined by the instruction at node *n*.

ref(`x = 3`) = { }

def(`x = 3`) = { `x` }

ref(`print x`) = { `x` }

def(`print x`) = { }

ref(`x = x + y`) = { `x`, `y` }

def(`x = x + y`) = { `x` }

Live variable analysis

As liveness flows backwards past an instruction, we want to modify the liveness information by *adding* any variables which it references (they become live) and *removing* any which it defines (they become dead).

```
     { x, y }
ref( print x ) = { x }
     { y }
```

```
     { y }
def( x = 3 ) = { x }
     { x, y }
```

Live variable analysis

If an instruction both references and defines variables, we must remove the defined variables *before* adding the referenced ones.

$\{ x, y, z \}$

$x = x + y$

$\{ x, z \}$

$$def(x = x + y) = \{ x \}$$

$$ref(x = x + y) = \{ x, y \}$$

Live variable analysis

So, if we consider *in-live*(*n*) and *out-live*(*n*), the sets of variables which are live immediately *before* and immediately *after* a node, the following equation must hold:

$$\mathit{in\text{-}live}(n) = \Big(\mathit{out\text{-}live}(n) \setminus \mathit{def}(n)\Big) \cup \mathit{ref}(n)$$

Live variable analysis

$$\textit{in-live}(n) = \Big(\textit{out-live}(n) \setminus \textit{def}(n)\Big) \cup \textit{ref}(n)$$

$$\textit{in-live}(n) = \Big(\textit{out-live}(n) \setminus \textit{def}(n)\Big) \cup \textit{ref}(n)$$

$$= (\{\ \mathtt{x}, \mathtt{z}\ \} \setminus \{\ \mathtt{x}\ \}) \cup \{\ \mathtt{x}, \mathtt{y}\ \}$$

n: $\boxed{\mathtt{x\ =\ x\ +\ y}}$

$$= \{\ \mathtt{z}\ \} \cup \{\ \mathtt{x}, \mathtt{y}\ \} = \{\ \mathtt{x}, \mathtt{y}, \mathtt{z}\ \}$$

$$\textit{out-live}(n) = \{\ \mathtt{x}, \mathtt{z}\ \}$$

$\textit{def}(n) = \{\ \mathtt{x}\ \}$ $\qquad$ $\textit{ref}(n) = \{\ \mathtt{x}, \mathtt{y}\ \}$

Live variable analysis

So we know how to calculate *in-live*(*n*) from the values of *def*(*n*), *ref*(*n*) and *out-live*(*n*). But how do we calculate *out-live*(*n*)?

$$\mathit{in\text{-}live}(n) = \big(\mathit{out\text{-}live}(n) \setminus \mathit{def}(n)\big) \cup \mathit{ref}(n)$$

n: `x = x + y`

$$\mathit{out\text{-}live}(n) = ?$$

Live variable analysis

In straight-line code each node has a unique successor, and the variables live at the exit of a node are exactly those variables live at the entry of its successor.

Live variable analysis

l:

$out\text{-}live(l) = \{\ \mathtt{s}, \mathtt{t}, \mathtt{x}, \mathtt{y}\ \}$

$in\text{-}live(m) = \big(out\text{-}live(m) \setminus def(m)\big) \cup ref(m)$

m: `z = x * y;`

$out\text{-}live(m) = \{\ \mathtt{s}, \mathtt{t}, \mathtt{z}\ \}$

$in\text{-}live(n) = \big(out\text{-}live(n) \setminus def(n)\big) \cup ref(n)$

n: `print s + t;`

$out\text{-}live(n) = \{\ \mathtt{z}\ \}$

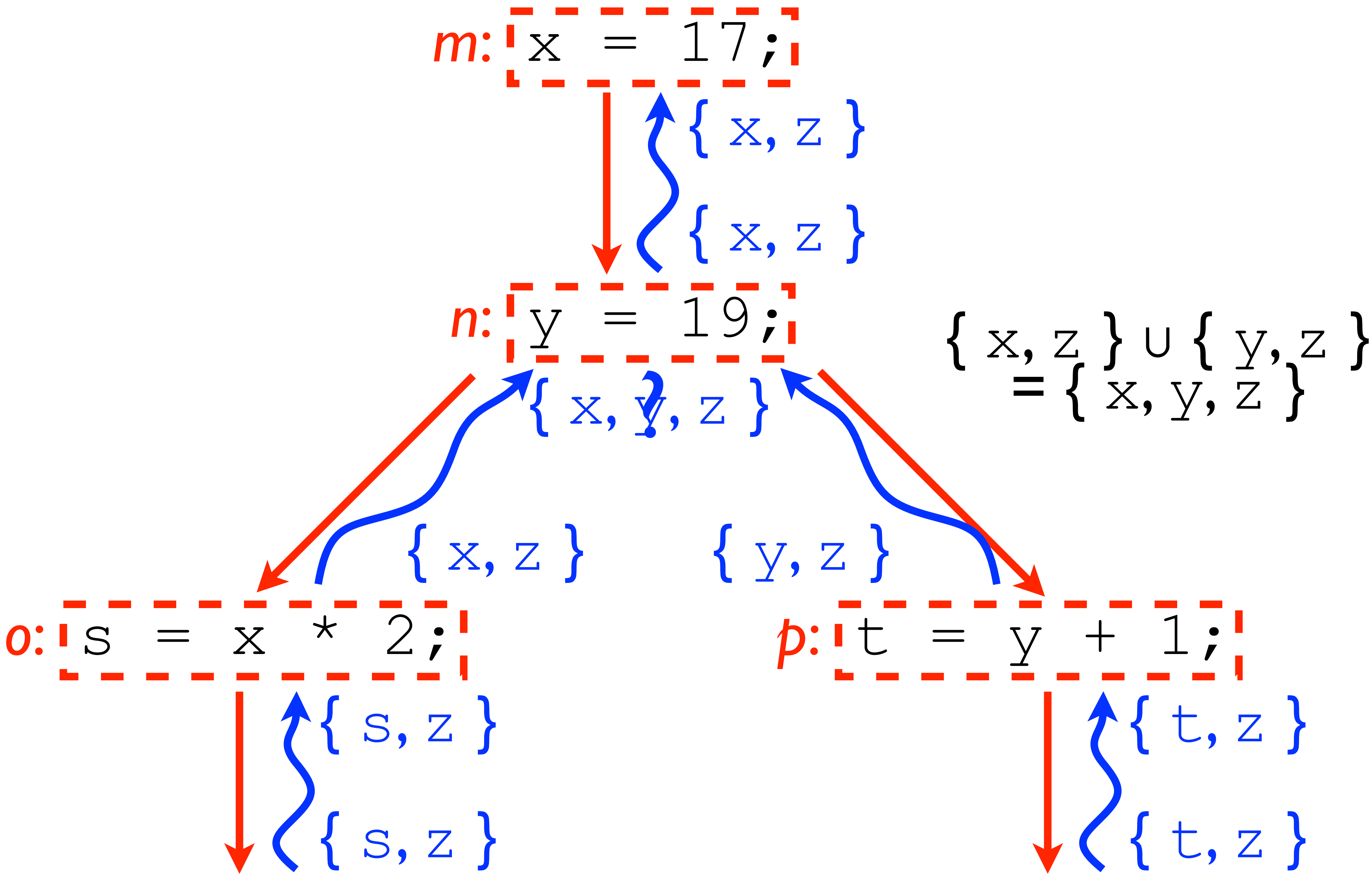
$in\text{-}live(o) = \big(out\text{-}live(o) \setminus def(o)\big) \cup ref(o)$

o:

Live variable analysis

In general, however, each node has an arbitrary number of successors, and the variables live at the exit of a node are exactly those variables live at the entry of *any* of its successors.

Live variable analysis



Live variable analysis

So the following equation must also hold:

$$\mathit{out\text{-}live}(n) = \bigcup_{s \in succ(n)} \mathit{in\text{-}live}(s)$$

Data-flow equations

These are the *data-flow equations* for live variable analysis, and together they tell us everything we need to know about how to propagate liveness information through a program.

$$\textit{in-live}(n) = \Big(\textit{out-live}(n) \setminus \textit{def}(n)\Big) \cup \textit{ref}(n)$$

$$\textit{out-live}(n) = \bigcup_{s \in \textit{succ}(n)} \textit{in-live}(s)$$

Data-flow equations

Each is expressed in terms of the other, so we can combine them to create one overall liveness equation.

$$live(n) = \left(\left(\bigcup_{s \in succ(n)} live(s) \right) \setminus def(n) \right) \cup ref(n)$$

Algorithm

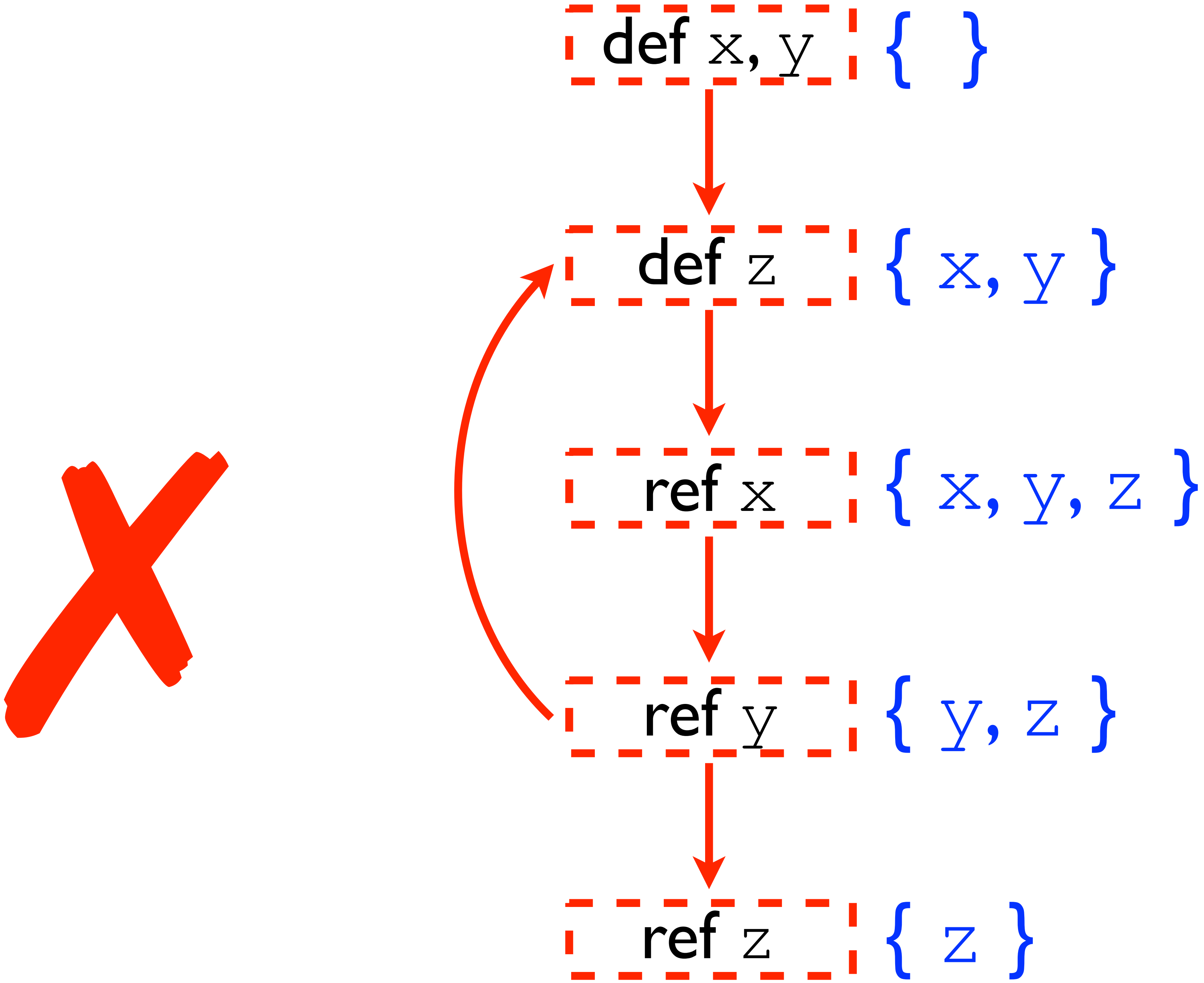
We now have a formal description of liveness, but we need an actual algorithm in order to do the analysis.

Algorithm

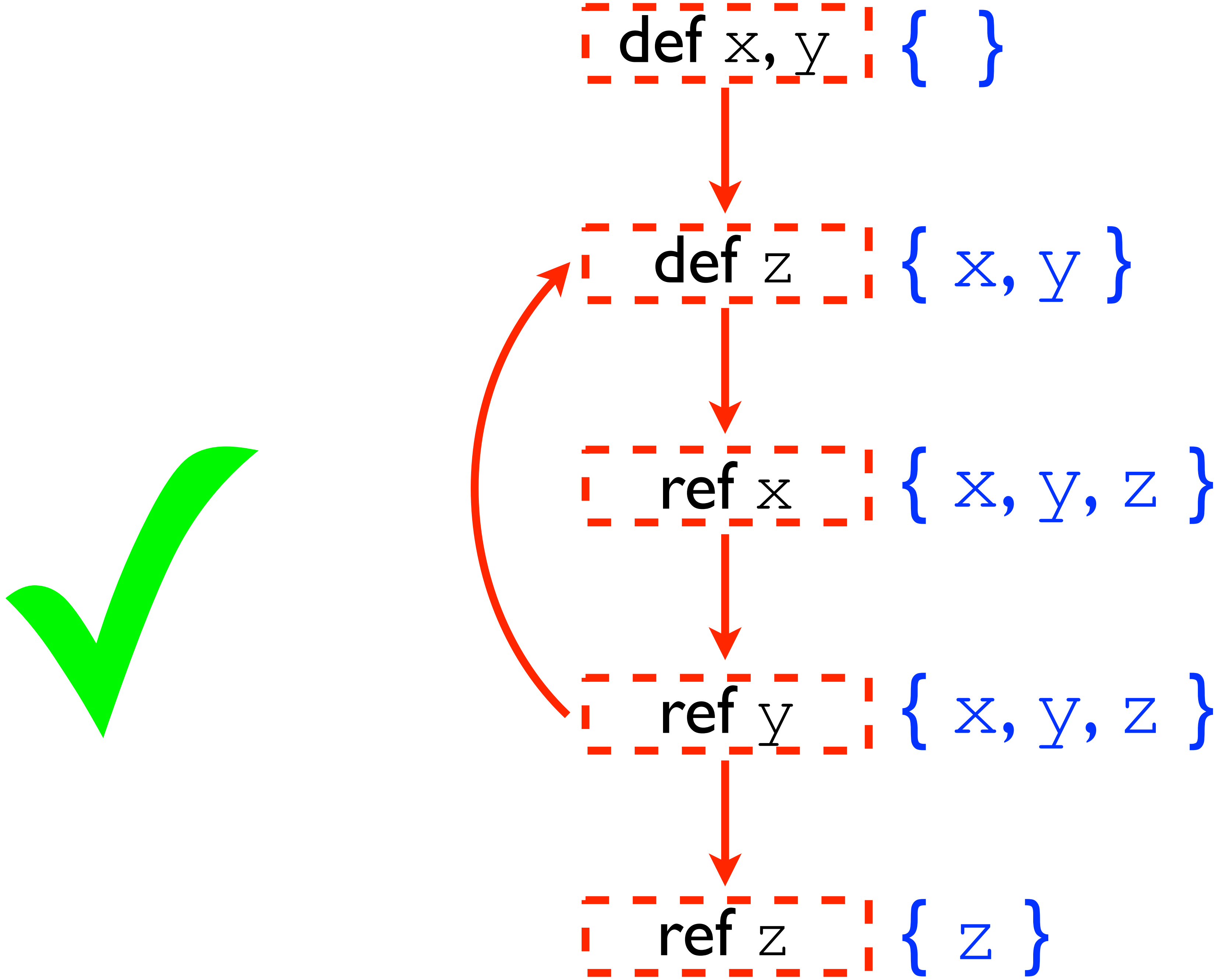
"Doing the analysis" consists of computing a value *live(n)* for each node *n* in a flowgraph such that the liveness data-flow equations are satisfied.

A simple way to solve the data-flow equations is to adopt an iterative strategy.

Algorithm



Algorithm



Algorithm

```
for i = 1 to n do live[i] := {}
while (live[] changes) do
   for i = 1 to n do
```

$$\texttt{live[i]} := \left(\left(\bigcup_{s \in succ(\texttt{i})} \texttt{live}[s]\right) \setminus def(\texttt{i})\right) \cup ref(\texttt{i})$$

Algorithm

This algorithm is guaranteed to terminate since there are a *finite* number of variables in each program and the effect of one iteration is *monotonic*.

Furthermore, although any solution to the data-flow equations is safe, this algorithm is guaranteed to give the *smallest* (and therefore most precise) solution.

(See the *Knaster-Tarski theorem* if you're interested.)

Algorithm

Implementation notes:

- If the program has *n* variables, we can implement each element of `live[]` as an *n*-bit value, with each bit representing the liveness of one variable.

- We can store liveness once per basic block and recompute inside a block when necessary. In this case, given a basic block *n* of instructions i_1, ..., i_k:

$$live(n) = \left(\bigcup_{s \in succ(n)} live(s) \right) \setminus def(i_k) \cup ref(i_k) \cdots \setminus def(i_1) \cup ref(i_1)$$

Safety of analysis

- Syntactic liveness safely overapproximates semantic liveness.
- The usual problem occurs in the presence of address-taken variables (cf. labels, procedures): *ambiguous* definitions and references. For safety we must
  - overestimate ambiguous references (assume all address-taken variables are referenced) and
  - underestimate ambiguous definitions (assume no variables are defined); this increases the size of the smallest solution.

Safety of analysis

```
MOV x,#1
MOV y,#2
MOV z,#3
MOV t32,#&x
MOV t33,#&y
MOV t34,#&z
  ⋮
m: STI t35,#7
  ⋮
n: LDI t36,t37
```

def(*m*) = { }
ref(*m*) = { t35 }

def(*n*) = { t36 }
ref(*n*) = { t37, x, y, z }

Summary

- Data-flow analysis collects information about how data moves through a program
- Variable liveness is a data-flow property
- Live variable analysis (LVA) is a backwards data-flow analysis for determining variable liveness
- LVA may be expressed as a pair of complementary data-flow equations, which can be combined
- A simple iterative algorithm can be used to find the smallest solution to the LVA data-flow equations

Lecture 4
Available expression analysis

Motivation

Programs may contain code whose result is needed, but in which some computation is simply a redundant repetition of earlier computation within the same program.

The concept of *expression availability* is useful in dealing with this situation.

Expressions

Any given program contains a finite number of expressions (i.e. computations which potentially produce values), so we may talk about the *set of all expressions* of a program.

```
int z = x * y;
print s + t;
int w = u / v;
⋮
```

program contains expressions { `x*y`, `s+t`, `u/v`, ... }

Availability

Availability is a data-flow property of expressions:
“Has the value of this expression already been computed?”

```
    ⋮
    int z = x * y;
}
```

Availability

At each instruction, each expression in the program is either available or unavailable.

We therefore usually consider availability from an instruction's perspective: each instruction (or node of the flowgraph) has an associated *set* of available expressions.

```
   int z = x * y;
   print s + t;
n: int w = u / v;
   ⋮
```

avail(*n*) = { `x*y`, `s+t` }

Availability

So far, this is all familiar from live variable analysis.

Note that, while expression availability and variable liveness share many similarities (both are simple data-flow properties), they do differ in important ways.

By working through the low-level details of the availability property and its associated analysis we can see where the differences lie and get a feel for the capabilities of the general data-flow analysis framework.

Semantic vs. syntactic

For example, availability differs from earlier examples in a subtle but important way: we want to know which expressions are *definitely* available (i.e. have already been computed) at an instruction, not which ones *may* be available.

As before, we should consider the distinction between *semantic* and *syntactic* (or, alternatively, *dynamic* and *static*) availability of expressions, and the details of the approximation which we hope to discover by analysis.

Semantic vs. syntactic

An expression is *semantically* available at a node *n* if its value gets computed (and not subsequently invalidated) along *every execution sequence* ending at *n*.

```
int x = y * z;
⋮
return y * z;
```

`y*z` AVAILABLE

Semantic vs. syntactic

An expression is *semantically* available at a node *n* if its value gets computed (and not subsequently invalidated) along *every execution sequence* ending at *n*.

```
int x = y * z;
⋮
y = a + b;
⋮
return y * z;
```

`y*z` UNAVAILABLE

Semantic vs. syntactic

An expression is *syntactically* available at a node *n* if its value gets computed (and not subsequently invalidated) along every path from the entry of the flowgraph to *n*.

As before, semantic availability is concerned with the *execution behaviour* of the program, whereas syntactic availability is concerned with the program's *syntactic structure*.

And, as expected, only the latter is decidable.

Semantic vs. syntactic

```
if ((x+1)*(x+1) == y) {
  s = x + y;
}
if (x*x + 2*x + 1 != y) {
  t = x + y;
}
return x + y;
```

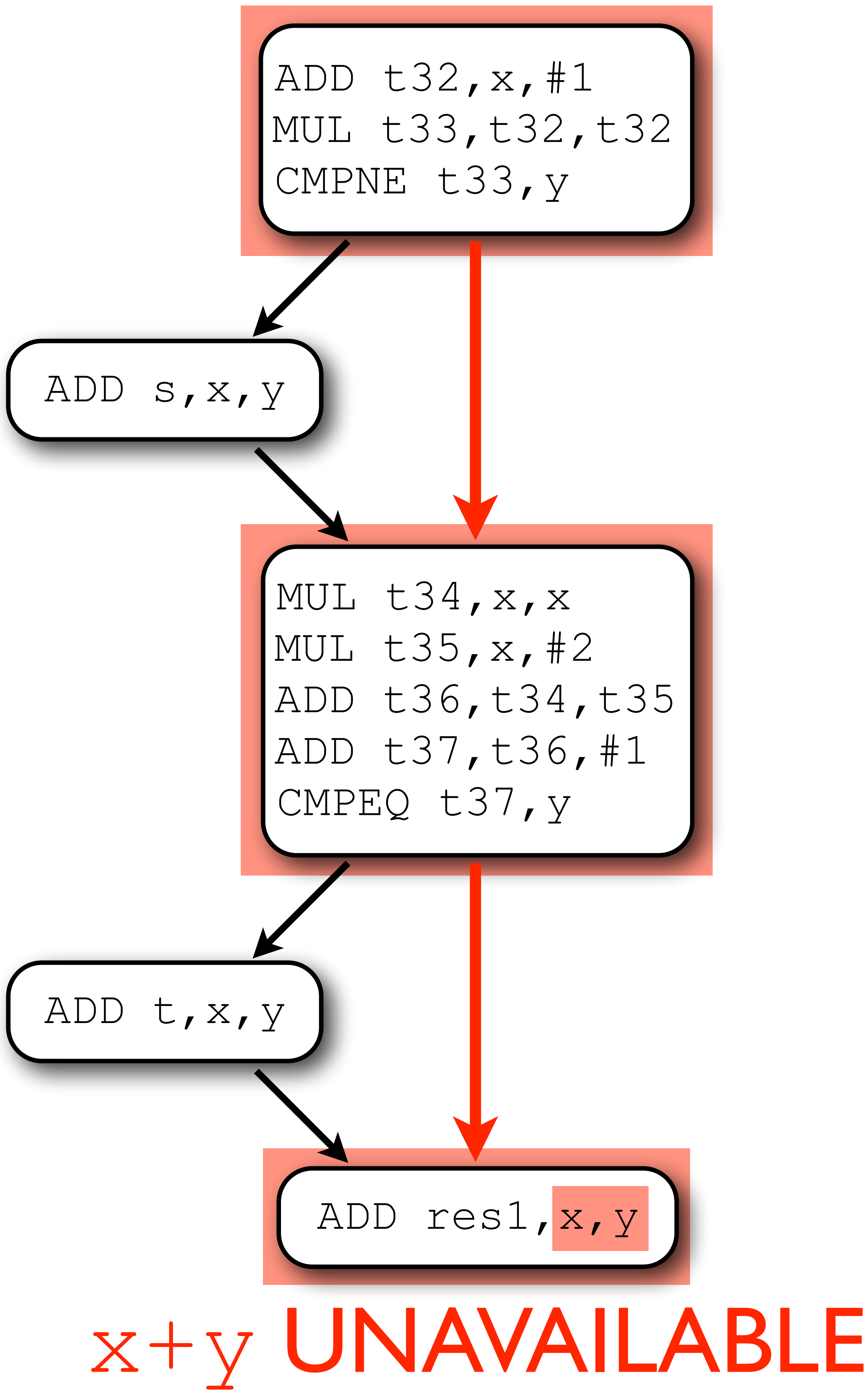
`x+y` AVAILABLE

Semantically: one of the conditions will be true, so on every execution path `x+y` is computed twice. The recomputation of `x+y` is redundant.

Semantic vs. syntactic

```
       ADD t32,x,#1
       MUL t33,t32,t32
       CMPNE t33,y,lab1
       ADD s,x,y
lab1:  MUL t34,x,x
       MUL t35,x,#2
       ADD t36,t34,t35
       ADD t37,t36,#1
       CMPEQ t37,y,lab2
       ADD t,x,y
lab2:  ADD res1,x,y
```

Semantic vs. syntactic



On *this* path through the flowgraph, `x+y` is only computed once, so `x+y` is *syntactically* unavailable at the last instruction.

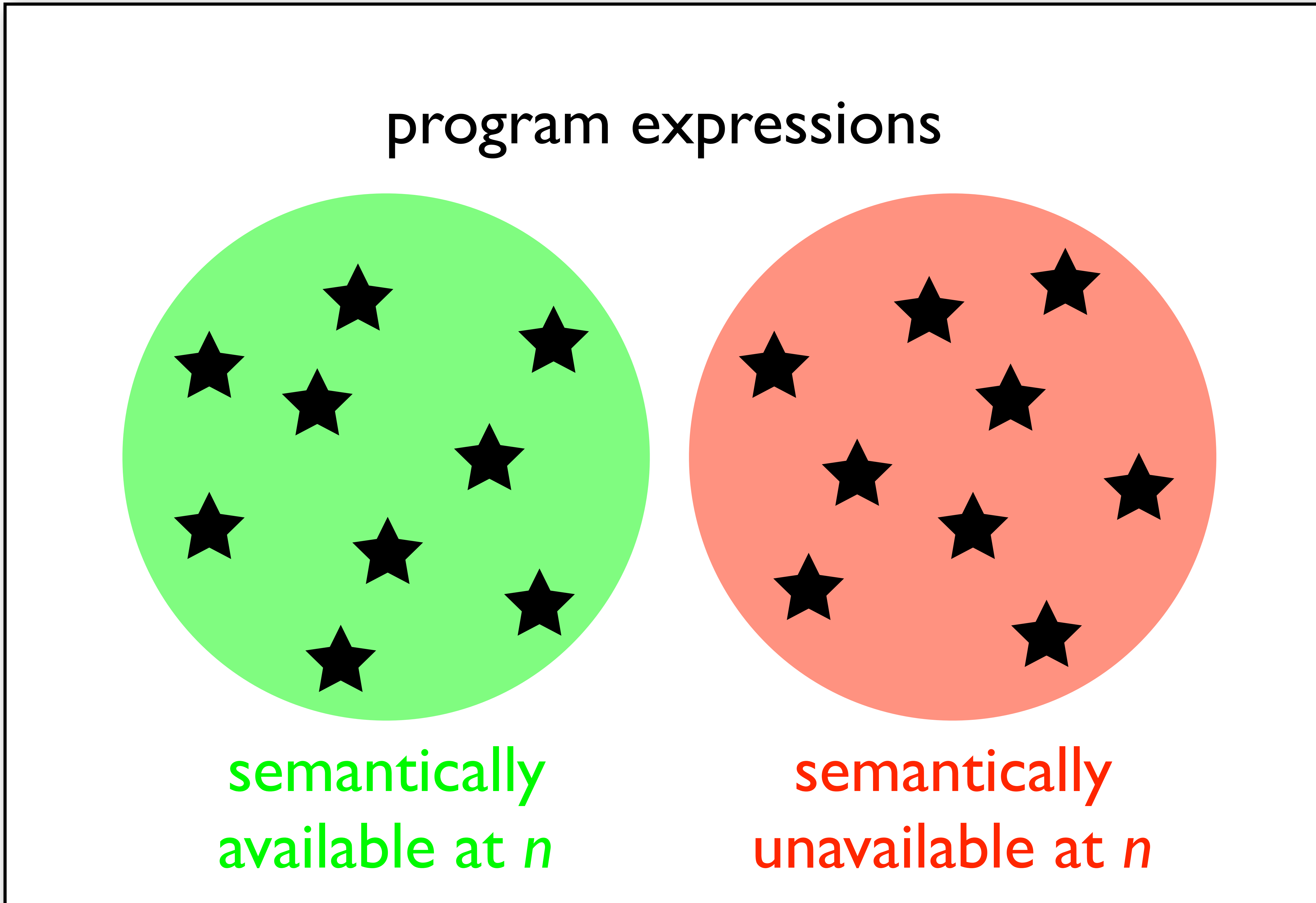
Note that this path never actually occurs during execution.

Semantic vs. syntactic

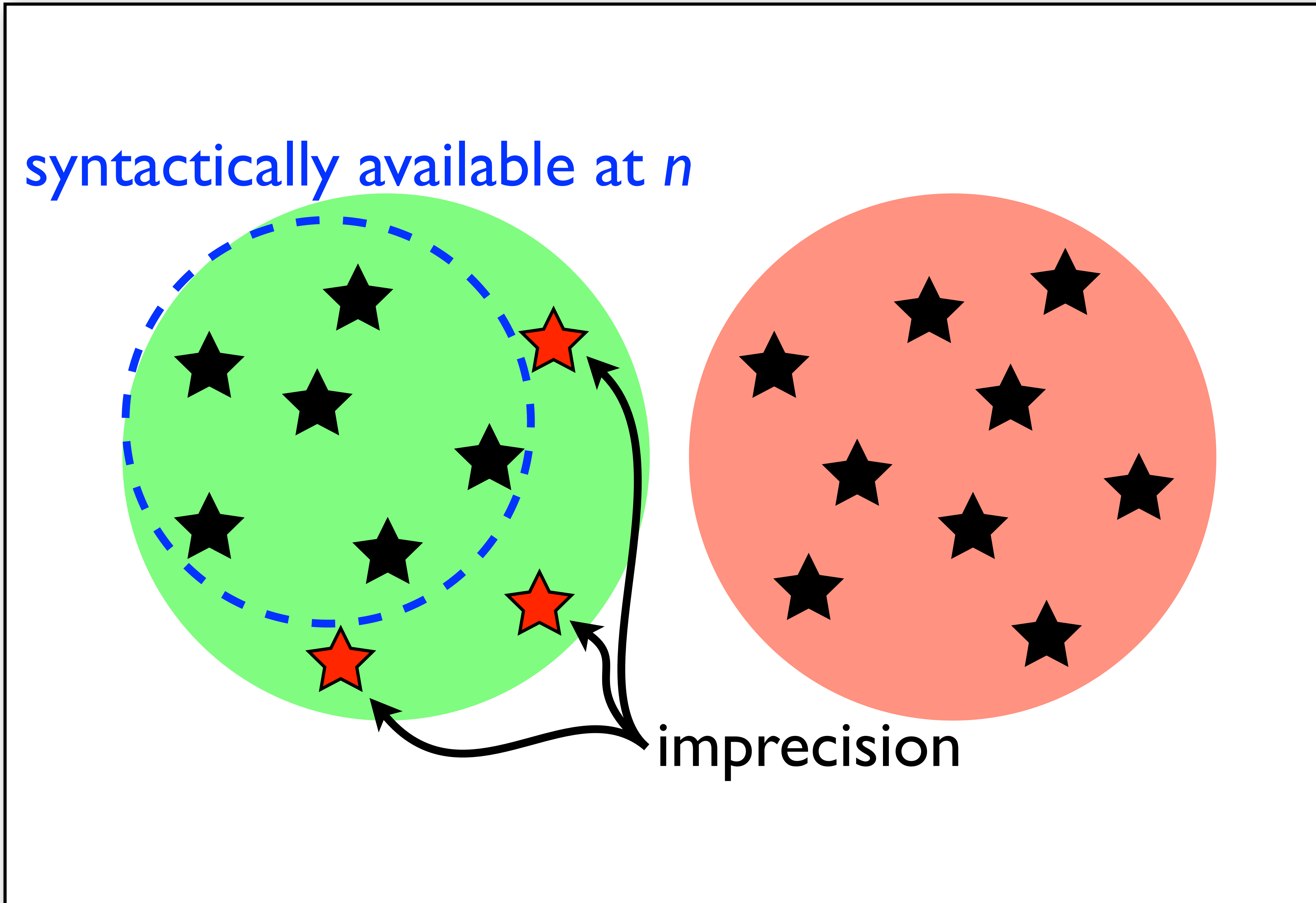
If an expression is deemed to be available, we may do something dangerous (e.g. remove an instruction which recomputes its value).

Whereas with live variable analysis we found safety in assuming that *more* variables were live, here we find safety in assuming that *fewer* expressions are available.

Semantic vs. syntactic



Semantic vs. syntactic



Semantic vs. syntactic

$$sem\text{-}avail(n) \supseteq syn\text{-}avail(n)$$

This time, we *safely underestimate* availability.

(cf. $sem\text{-}live(n) \subseteq syn\text{-}live(n)$)

Warning

Danger: there is a standard presentation of available expression analysis (textbooks, notes for this course) which is formally satisfying but contains an easily-overlooked subtlety.

We'll first look at an equivalent, more intuitive bottom-up presentation, then amend it slightly to match the version given in the literature.

Available expression analysis

Available expressions is a *forwards* data-flow analysis: information from past instructions must be propagated forwards through the program to discover which expressions are available.

```
                t = x * y;
print x * y;                    if (x*y > 0)

      ⋮
      int z = x * y;
    }
```

Available expression analysis

Unlike variable liveness, expression availability flows *forwards* through the program.

As in liveness, though, each instruction has an effect on the availability information as it flows past.

Available expression analysis

An instruction makes an expression available when it *generates* (computes) its current value.

Available expression analysis

{ }

`print a*b;` GENERATE `a*b`

{ a*b }

`c = d + 1;` GENERATE `d+1`

{ a*b, d+1 }

`e = f / g;` GENERATE `f/g`

{ a*b, d+1, f/g }

Available expression analysis

An instruction makes an expression unavailable when it *kills* (invalidates) its current value.

Available expression analysis

{ a*b, c+1, d/e, d-1 }

a = 7; KILL a*b

{ c+1, d/e, d-1 }

c = 11; KILL c+1

{ d/e, d-1 }

d = 13; KILL d/e, d-1

{ }

Available expression analysis

As in LVA, we can devise functions *gen*(n) and *kill*(n) which give the sets of expressions generated and killed by the instruction at node n.

The situation is slightly more complicated this time: an assignment to a variable x kills *all expressions in the program* which contain occurrences of x.

Available expression analysis

So, in the following, E_x is the set of expressions in the program which contain occurrences of x.

gen(`x = 3`) = { }

kill(`x = 3`) = E_x

gen(`print x+1`) = { `x+1` }

kill(`print x+1`) = { }

gen(`x = x + y`) = { `x+y` }

kill(`x = x + y`) = E_x

Available expression analysis

As availability flows forwards past an instruction, we want to modify the availability information by *adding* any expressions which it generates (they become available) and *removing* any which it kills (they become unavailable).

{ y+1 }

gen(`print x+1`) = { x+1 }

{ x+1, y+1 }

{ x+1, y+1 }

kill(`x = 3`) = E_x

{ y+1 }

Available expression analysis

If an instruction both generates and kills expressions, we must remove the killed expressions *after* adding the generated ones (cf. removing *def*(*n*) *before* adding *ref*(*n*)).

{ `x+1`, `y+1` }

```
x = x + y
```

{ `y+1` }

gen(`x = x + y`) = { `x+y` }

kill(`x = x + y`) = E_x

Available expression analysis

So, if we consider *in-avail*(*n*) and *out-avail*(*n*), the sets of expressions which are available immediately *before* and immediately *after* a node, the following equation must hold:

$$out\text{-}avail(n) = \Big(in\text{-}avail(n) \cup gen(n)\Big) \setminus kill(n)$$

Available expression analysis

$$out\text{-}avail(n) = \Big(in\text{-}avail(n) \cup gen(n)\Big) \setminus kill(n)$$

in-avail(*n*) = { `x+1`, `y+1` }

n: $\boxed{\texttt{x = x + y}}$

out-avail(*n*) = (*in-avail*(*n*) ∪ *gen*(*n*)) ∖ *kill*(*n*)

= ({ `x+1`, `y+1` } ∪ { `x+y` }) ∖ { `x+1`, `x+y` }

= { `x+1`, `x+y`, `y+1` } ∖ { `x+1`, `x+y` } = { `y+1` }

gen(*n*) = { `x+y` }   *kill*(*n*) = { `x+1`, `x+y` }

Available expression analysis

As in LVA, we have devised one equation for calculating *out-avail*(n) from the values of *gen*(n), *kill*(n) and *in-avail*(n), and now need another for calculating *in-avail*(n).

$\textit{in-avail}(n) = ?$

n: $\boxed{\texttt{x = x + y}}$

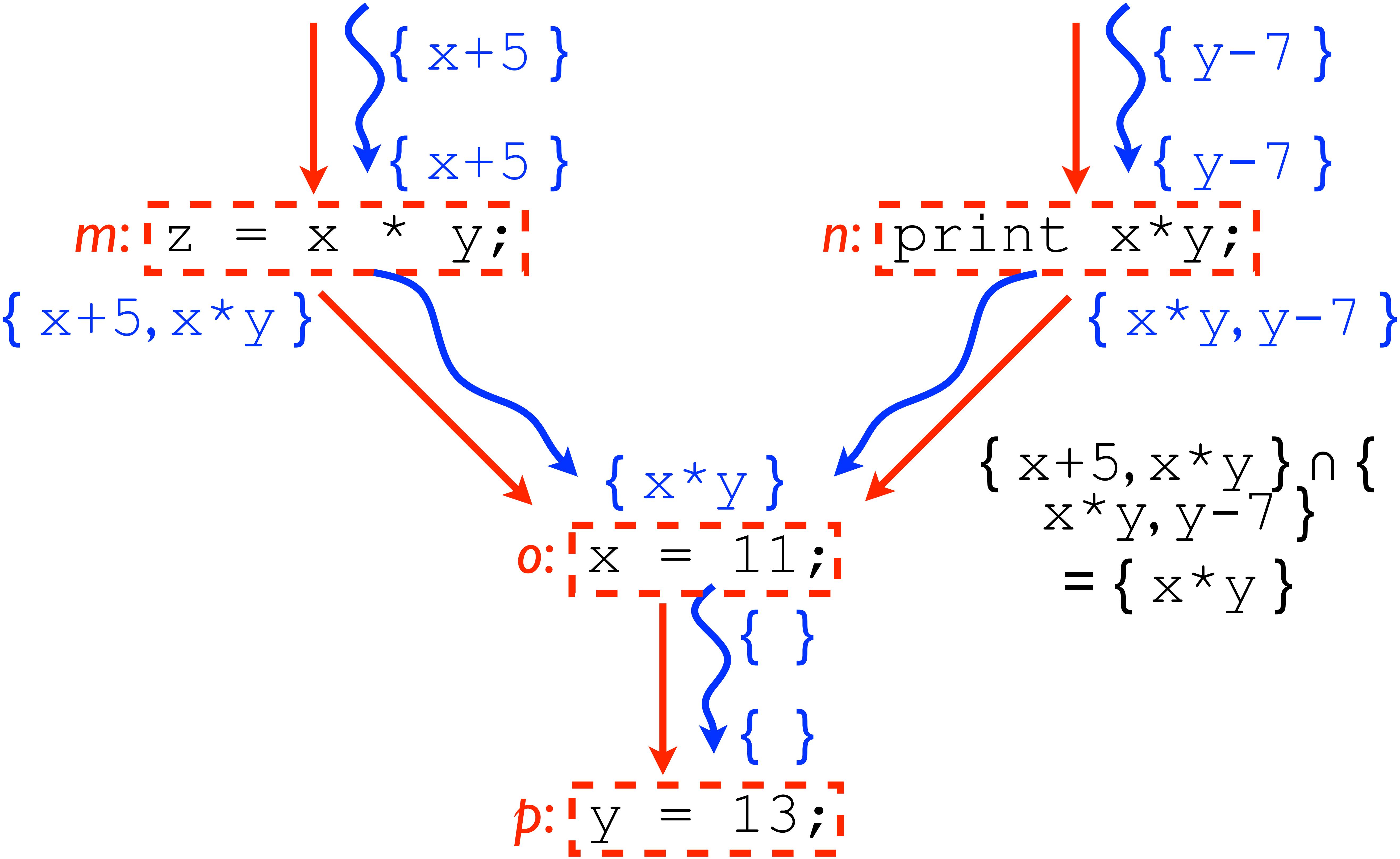
$\textit{out-avail}(n) = \big(\textit{in-avail}(n) \cup \textit{gen}(n)\big) \setminus \textit{kill}(n)$

Available expression analysis

When a node *n* has a single predecessor *m*, the information propagates along the control-flow edge as you would expect: *in-avail*(*n*) = *out-avail*(*m*).

When a node has multiple predecessors, the expressions available at the entry of that node are exactly those expressions available at the exit of *all* of its predecessors (cf. "*any* of its successors" in LVA).

Available expression analysis



Available expression analysis

So the following equation must also hold:

$$\mathit{in\text{-}avail}(n) = \bigcap_{p \in pred(n)} \mathit{out\text{-}avail}(p)$$

Data-flow equations

These are the *data-flow equations* for available expression analysis, and together they tell us everything we need to know about how to propagate availability information through a program.

$$\mathit{in\text{-}avail}(n) = \bigcap_{p \in \mathit{pred}(n)} \mathit{out\text{-}avail}(p)$$

$$\mathit{out\text{-}avail}(n) = \Big(\mathit{in\text{-}avail}(n) \cup \mathit{gen}(n)\Big) \setminus \mathit{kill}(n)$$

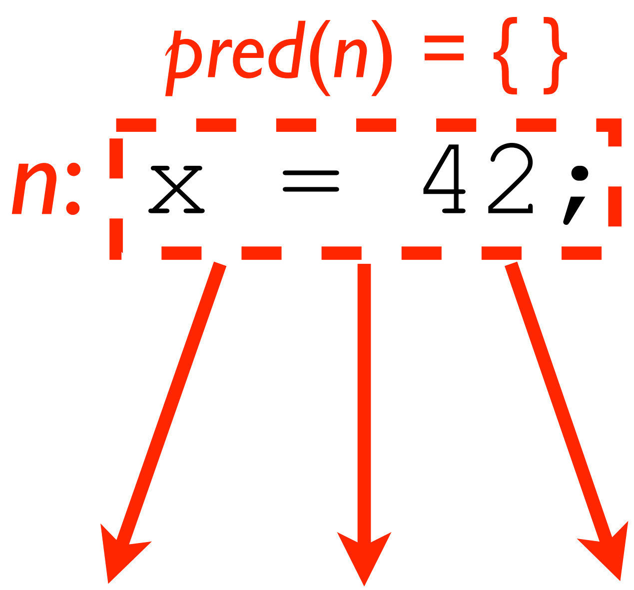
Data-flow equations

Each is expressed in terms of the other, so we can combine them to create one overall availability equation.

$$avail(n) = \bigcap_{p \in pred(n)} \Big((avail(p) \cup gen(p)) \setminus kill(p) \Big)$$

Data-flow equations

Danger: we have overlooked one important detail.



$$\mathit{avail}(n) = \bigcap_{p \in \mathit{pred}(n)} \big((\mathit{avail}(p) \cup \mathit{gen}(p)) \setminus \mathit{kill}(p)\big)$$

$$= \bigcap \{\,\}$$

$$= U$$ (i.e. *all* expressions in the program)

Clearly there should be *no* expressions available here, so we must stipulate explicitly that $\mathit{avail}(n) = \{\,\}$ if $\mathit{pred}(n) = \{\,\}$.

Data-flow equations

With this correction, our data-flow equation for expression availability is

$$avail(n) = \begin{cases} \bigcap_{p \in pred(n)} ((avail(p) \cup gen(p)) \setminus kill(p)) & \text{if } pred(n) \neq \{\,\} \\ \{\,\} & \text{if } pred(n) = \{\,\} \end{cases}$$

Data-flow equations

The functions and equations presented so far are correct, and their definitions are fairly intuitive.

However, we may wish to have our data-flow equations in a form which more closely matches that of the LVA equations, since this emphasises the similarity between the two analyses and hence is how they are most often presented.

A few modifications are necessary to achieve this.

Data-flow equations

$$in\text{-}live(n) = \Big(out\text{-}live(n) \setminus def(n)\Big) \cup ref(n)$$

$$out\text{-}live(n) = \bigcup_{s \in succ(n)} in\text{-}live(s)$$

These differences are inherent in the analyses.

$$in\text{-}avail(n) = \bigcap_{p \in pred(n)} out\text{-}avail(p)$$

$$out\text{-}avail(n) = \Big(in\text{-}avail(n) \cup gen(n)\Big) \setminus kill(n)$$

Data-flow equations

$$\mathit{in\text{-}live}(n) = \Big(\mathit{out\text{-}live}(n) \setminus \mathit{def}(n)\Big) \cup \mathit{ref}(n)$$

$$\mathit{out\text{-}live}(n) = \bigcup_{s \in \mathit{succ}(n)} \mathit{in\text{-}live}(s)$$

These differences are an arbitrary result of our definitions.

$$\mathit{in\text{-}avail}(n) = \bigcap_{p \in \mathit{pred}(n)} \mathit{out\text{-}avail}(p)$$

$$\mathit{out\text{-}avail}(n) = \Big(\mathit{in\text{-}avail}(n) \cup \mathit{gen}(n)\Big) \setminus \mathit{kill}(n)$$

Data-flow equations

We might instead have decided to define *gen*(*n*) and *kill*(*n*) to coincide with the following (standard) definitions:

- A node *generates* an expression e if it *must* compute the value of e and does not subsequently redefine any of the variables occuring in e.
- A node *kills* an expression e if it *may* redefine some of the variables occurring in e and does not subsequently recompute the value of e.

Data-flow equations

By the old definition:

gen(`x = x + y`) = { `x+y` }

kill(`x = x + y`) = E_x

By the new definition:

gen(`x = x + y`) = { }

kill(`x = x + y`) = E_x

(The new *kill*(*n*) may visibly differ when *n* is a basic block.)

Data-flow equations

Since these new definitions take account of which expressions are generated *overall* by a node (and exclude those which are generated only to be immediately killed), we may propagate availability information through a node by removing the killed expressions *before* adding the generated ones, *exactly as in LVA*.

$$\mathit{out\text{-}avail}(n) = \Big(\mathit{in\text{-}avail}(n) \setminus \mathit{kill}(n)\Big) \cup \mathit{gen}(n)$$

$$\mathit{in\text{-}live}(n) = \Big(\mathit{out\text{-}live}(n) \setminus \mathit{def}(n)\Big) \cup \mathit{ref}(n)$$

Data-flow equations

From this new equation for *out-avail*(*n*) we may produce our final data-flow equation for expression availability:

$$avail(n) = \begin{cases} \bigcap_{p \in pred(n)} ((avail(p) \setminus kill(p)) \cup gen(p)) & \text{if } pred(n) \neq \{\,\} \\ \{\,\} & \text{if } pred(n) = \{\,\} \end{cases}$$

This is the equation you will find in the course notes and standard textbooks on program analysis; remember that it depends on these more subtle definitions of *gen*(*n*) and *kill*(*n*).

Algorithm

- We again use an array, `avail[]`, to store the available expressions for each node.
- We initialise `avail[]` such that each node has *all* expressions available (cf. LVA: *no* variables live).
- We again iterate application of the data-flow equation at each node until `avail[]` no longer changes.

Algorithm

```
for i = 1 to n do avail[i] := U
while (avail[] changes) do
   for i = 1 to n do
      avail[i] := ⋂_{p∈pred(i)} ((avail[p] \ kill(p)) ∪ gen(p))
```

$$\texttt{avail[i]} := \bigcap_{p \in pred(\texttt{i})} ((\texttt{avail}[p] \setminus kill(p)) \cup gen(p))$$

Algorithm

We can do better if we assume that the flowgraph has a single entry node (the first node in `avail[]`).

Then `avail[1]` may instead be initialised to the empty set, and we need not bother recalculating availability at the first node during each iteration.

Algorithm

```
avail[1] := {}
for i = 2 to n do avail[i] := U
while (avail[] changes) do
   for i = 2 to n do
```

$$\texttt{avail[i]} := \bigcap_{p \in pred(\texttt{i})} ((\texttt{avail}[p] \setminus kill(p)) \cup gen(p))$$

Algorithm

As with LVA, this algorithm is guaranteed to terminate since the effect of one iteration is *monotonic* (it only removes expressions from availability sets) and an empty availability set cannot get any smaller.

Any solution to the data-flow equations is safe, but this algorithm is guaranteed to give the *largest* (and therefore most precise) solution.

Algorithm

Implementation notes:

- If we arrange our programs such that each assignment assigns to a distinct temporary variable, we may number these temporaries and hence number the expressions whose values are assigned to them.
- If the program has *n* such expressions, we can implement each element of `avail[]` as an *n*-bit value, with the m^{th} bit representing the availability of expression number *m*.

Algorithm

Implementation notes:

- Again, we can store availability once per basic block and recompute inside a block when necessary. Given each basic block *n* has k_n instructions $n[1], ..., n[k_n]$:

$$avail(n) = \bigcap_{p \in pred(n)} (avail(p) \setminus kill(p[1]) \cup gen(p[1]) \cdots \setminus kill(p[k_p]) \cup gen(p[k_p]))$$

Safety of analysis

- Syntactic availability safely underapproximates semantic availability.
- Address-taken variables are again a problem. For safety we must
  - underestimate ambiguous generation (assume no expressions are generated) and
  - overestimate ambiguous killing (assume all expressions containing address-taken variables are killed); this decreases the size of the largest solution.

Analysis framework

The two data-flow analyses we've seen, LVA and AVAIL, clearly share many similarities.

In fact, they are both instances of the same simple data-flow analysis framework: some program property is computed by iteratively finding the most precise solution to data-flow equations, which express the relationships between values of that property immediately before and immediately after each node of a flowgraph.

Analysis framework

$$in\text{-}live(n) = \Big(out\text{-}live(n) \setminus def(n)\Big) \cup ref(n)$$

$$out\text{-}live(n) = \bigcup_{s \in succ(n)} in\text{-}live(s)$$

$$in\text{-}avail(n) = \bigcap_{p \in pred(n)} out\text{-}avail(p)$$

$$out\text{-}avail(n) = \Big(in\text{-}avail(n) \setminus kill(n)\Big) \cup gen(n)$$

Analysis framework

LVA's data-flow equations have the form

$$in(n) = (out(n) \setminus \ldots) \cup \ldots \qquad out(n) = \bigcup_{s \in succ(n)} in(s)$$

union over successors

AVAIL's data-flow equations have the form

$$out(n) = (in(n) \setminus \ldots) \cup \ldots \qquad in(n) = \bigcap_{p \in pred(n)} out(p)$$

intersection over predecessors

Analysis framework

	∩	∪
pred	AVAIL	RD
succ	VBE	LVA

...and others

Analysis framework

So, given a single algorithm for iterative solution of data-flow equations of this form, we may compute all these analyses and any others which fit into the framework.

Summary

- Expression availability is a data-flow property
- Available expression analysis (AVAIL) is a forwards data-flow analysis for determining expression availability
- AVAIL may be expressed as two complementary data-flow equations, which may be combined
- A simple iterative algorithm can be used to find the largest solution to the data-flow equations
- AVAIL and LVA are both instances (among others) of the same data-flow analysis framework

Lecture 5
Data-flow anomalies and clash graphs

Motivation

Both human- and computer-generated programs sometimes contain *data-flow anomalies*.

These anomalies result in the program being worse, in some sense, than it was intended to be.

Data-flow analysis is useful in locating, and sometimes correcting, these code anomalies.

Optimisation vs. debugging

Data-flow anomalies may manifest themselves in different ways: some may actually "break" the program (make it crash or exhibit undefined behaviour), others may just make the program "worse" (make it larger or slower than necessary).

Any compiler needs to be able to report when a program is broken (i.e. "compiler warnings"), so the identification of data-flow anomalies has applications in both optimisation and bug elimination.

Dead code

Dead code is a simple example of a data-flow anomaly, and LVA allows us to identify it.

Recall that code is *dead* when its result goes unused; if the variable `x` is not live on exit from an instruction which assigns some value to `x`, then the whole instruction is dead.

Dead code

```
⋮
a = x + 11;
b = y + 13;
c = a * b;
⋮
print z;
```

c DEAD

⋮

{ x, y, z }

{ a, y, z }

{ a, b, z }

{ z }

⋮

{ z }

{ }

Dead code

For this kind of anomaly, an automatic remedy is not only feasible but also straightforward: dead code with no live side effects is useless and may be removed.

Dead code

⋮	⋮
	{ x, y, z }
`a = x + 11;`	
	{ y, z }
`b = y + 13;`	
	{ a, b, z }
`c = a * b;`	
	{ z }
⋮	⋮
	{ z }
`print z;`	
	{ }

Successive iterations may yield further improvements.

Dead code

The program resulting from this transformation will remain correct and will be both *smaller* and *faster* than before (cf. just smaller in *unreachable* code elimination), and no programmer intervention is required.

Uninitialised variables

In some languages, for example C and our 3-address intermediate code, it is syntactically legitimate for a program to read from a variable before it has definitely been initialised with a value.

If this situation occurs during execution, the effect of the read is usually *undefined* and depends upon unpredictable details of implementation and environment.

Uninitialised variables

This kind of behaviour is often undesirable, so we would like a compiler to be able to detect and warn of the situation.

Happily, the liveness information collected by LVA allows a compiler to see easily when a read from an undefined variable is possible.

Uninitialised variables

In a "healthy" program, variable liveness produced by later instructions is consumed by earlier ones; if an instruction demands the value of a variable (hence making it live), it is expected that an earlier instruction will define that variable (hence making it dead again).

Uninitialised variables

✓

```
x = 11;
y = 13;
z = 17;
⋮
print x;
print y;
```

$\{\ \}$

$\{x\}$

$\{x, y\}$

$\{x, y\}$

$\vdots$

$\{x, y\}$

$\{y\}$

$\{\ \}$

Uninitialised variables

If any variables are still live at the beginning of a program, they represent uses which are potentially unmatched by corresponding definitions, and hence indicate a program with potentially undefined (and therefore incorrect) behaviour.

Uninitialised variables

```
x = 11;
y = 13;
⋮
print x;
print y;
print z;
```

{ z }   z LIVE

{ x, z }

{ x, y, z }

⋮

{ x, y, z }

{ y, z }

{ z }

{ }

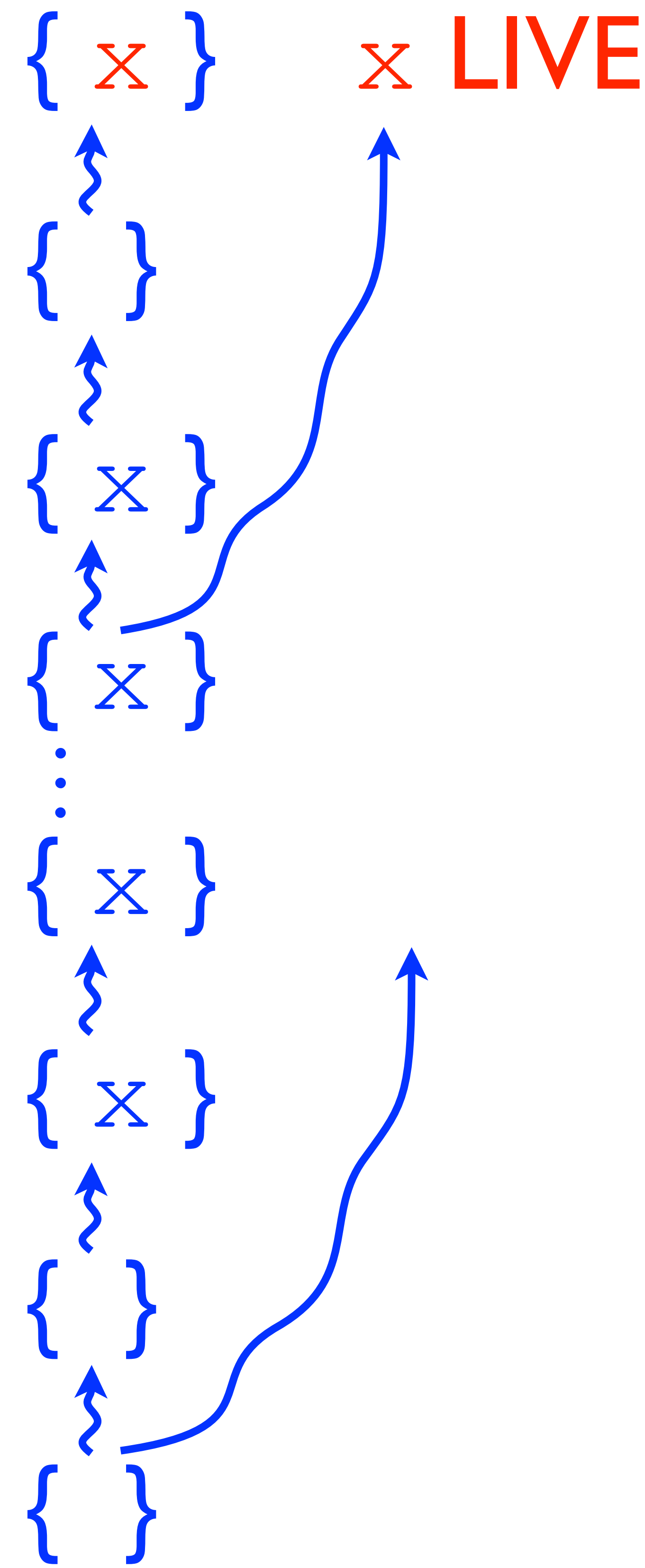
Uninitialised variables

In this situation, the compiler can issue a warning: “variable `z` may be used before it is initialised”.

However, because LVA computes a safe (syntactic) overapproximation of variable liveness, some of these compiler warnings may be (semantically) spurious.

Uninitialised variables

```
if (p) {
  x = 42;
}
⋮
if (p) {
  print x;
}
```



Note: intentionally ignoring p!

Uninitialised variables

Here the analysis is being *too* safe, and the warning is unnecessary, but this imprecision is the nature of our computable approximation to semantic liveness.

So the compiler must either risk giving unnecessary warnings about correct code (“false positives”) or failing to give warnings about incorrect code (“false negatives”). Which is worse?

Opinions differ.

Uninitialised variables

Although dead code may easily be remedied by the compiler, it's not generally possible to automatically fix the problem of uninitialised variables.

As just demonstrated, even the decision as to whether a warning indicates a genuine problem must often be made by the programmer, who must also fix any such problems by hand.

Uninitialised variables

Note that higher-level languages have the concept of (possibly nested) *scope*, and our expectations for variable initialisation in“healthy” programs can be extended to these.

In general we expect the set of live variables at the beginning of any scope to *not contain* any of the variables local to that scope.

Uninitialised variables

```
int x = 5;
int y = 7;
if (p) {
   int z;        { x, y, z } z LIVE
   ⋮
   print z;
}
print x+y;
```

Write-write anomalies

While LVA is useful in these cases, some similar data-flow anomalies can only be spotted with a different analysis.

Write-write anomalies are an example of this. They occur when a variable may be written twice with no intervening read; the first write may then be considered unnecessary in some sense.

```
x = 11;
x = 13;
print x;
```

Write-write anomalies

A simple data-flow analysis can be used to track which variables may have been written but not yet read at each node.

In a sense, this involves doing LVA in reverse (i.e. forwards!): at each node we should remove all variables which are referenced, then add all variables which are defined.

Write-write anomalies

$$in\text{-}wnr(n) = \bigcup_{p \in pred(n)} out\text{-}wnr(p)$$

$$out\text{-}wnr(n) = \Big(in\text{-}wnr(n) \setminus ref(n) \Big) \cup def(n)$$

$$wnr(n) = \bigcup_{p \in pred(n)} \Big((wnr(p) \setminus ref(p)) \cup def(p) \Big)$$

Write-write anomalies

```
x = 11;
y = 13;
z = 17;
⋮
print x;
y = 19;
⋮
```

$\{\ \}$

$\{x\}$

$\{x, y\}$

$\{x, y, z\}$

⋮

$\{x, y, z\}$

$\{y, z\}$

$\{y, z\}$

⋮

y is also dead here.

y is rewritten here without ever having been read.

Write-write anomalies

But, although the second write to a variable *may* turn an earlier write into dead code, the presence of a write-write anomaly doesn't *necessarily* mean that a variable is dead — hence the need for a different analysis.

Write-write anomalies

```
x = 11;
if (p) {
  x = 13;
}
print x;
```

`x` is live throughout this code, but if `p` is true during execution, `x` will be written twice before it is read. In most cases, the programmer can remedy this.

Write-write anomalies

```
if (p) {
  x = 13;
} else {
  x = 11;
}
print x;
```

This code does the same job, but avoids writing to `x` twice in succession on any control-flow path.

Write-write anomalies

```
if (p) {
  x = 13;
}
if (!p) {
  x = 11;
}
print x;
```

Again, the analysis may be too approximate to notice that a particular write-write anomaly may never occur during any execution, so warnings may be inaccurate.

Write-write anomalies

As with uninitialised variable anomalies, the programmer must be relied upon to investigate the compiler's warnings and fix any genuine problems which they indicate.

Clash graphs

The ability to detect data-flow anomalies is a nice compiler feature, but LVA's main utility is in deriving a data structure known as a *clash graph* (aka *interference graph*).

Clash graphs

When generating intermediate code it is convenient to simply invent as many variables as necessary to hold the results of computations; the extreme of this is "normal form", in which a new temporary variable is used on each occasion that one is required, with none being reused.

Clash graphs

```
x = (a*b) + c;
y = (a*b) + d;
```

lex, parse, translate

```
MUL t1,a,b
ADD x,t1,c
MUL t2,a,b
ADD y,t2,d
```

Clash graphs

This makes generating 3-address code as straightforward as possible, and assumes an imaginary target machine with an unlimited supply of "virtual registers", one to hold each variable (and temporary) in the program.

Such a naïve strategy is obviously wasteful, however, and won't generate good code for a real target machine.

Clash graphs

Before we can work on improving the situation, we must collect information about which variables actually *need* to be allocated to different registers on the target machine, as opposed to having been *incidentally* placed in different registers by our translation to normal form.

LVA is useful here because it can tell us which variables are *simultaneously live*, and hence *must* be kept in separate virtual registers for later retrieval.

Clash graphs

```
x = 11;
y = 13;
z = (x+y) * 2;
a = 17;
b = 19;
z = z + (a*b);
```

Clash graphs

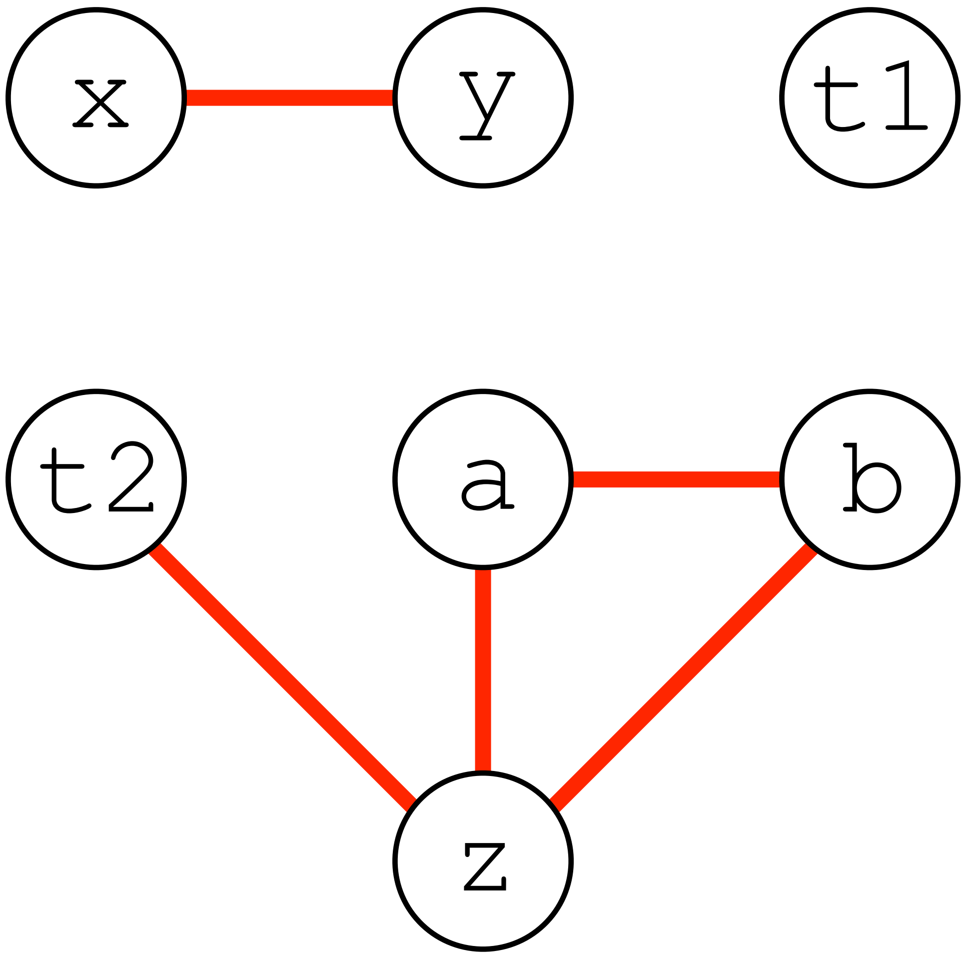
```
MOV x,#11
MOV y,#13
ADD t1,x,y
MUL z,t1,#2
MOV a,#17
MOV b,#19
MUL t2,a,b
ADD z,z,t2
```

Clash graphs

In a program's clash graph there is one vertex for each virtual register and an edge between vertices when their two registers are ever simultaneously live.

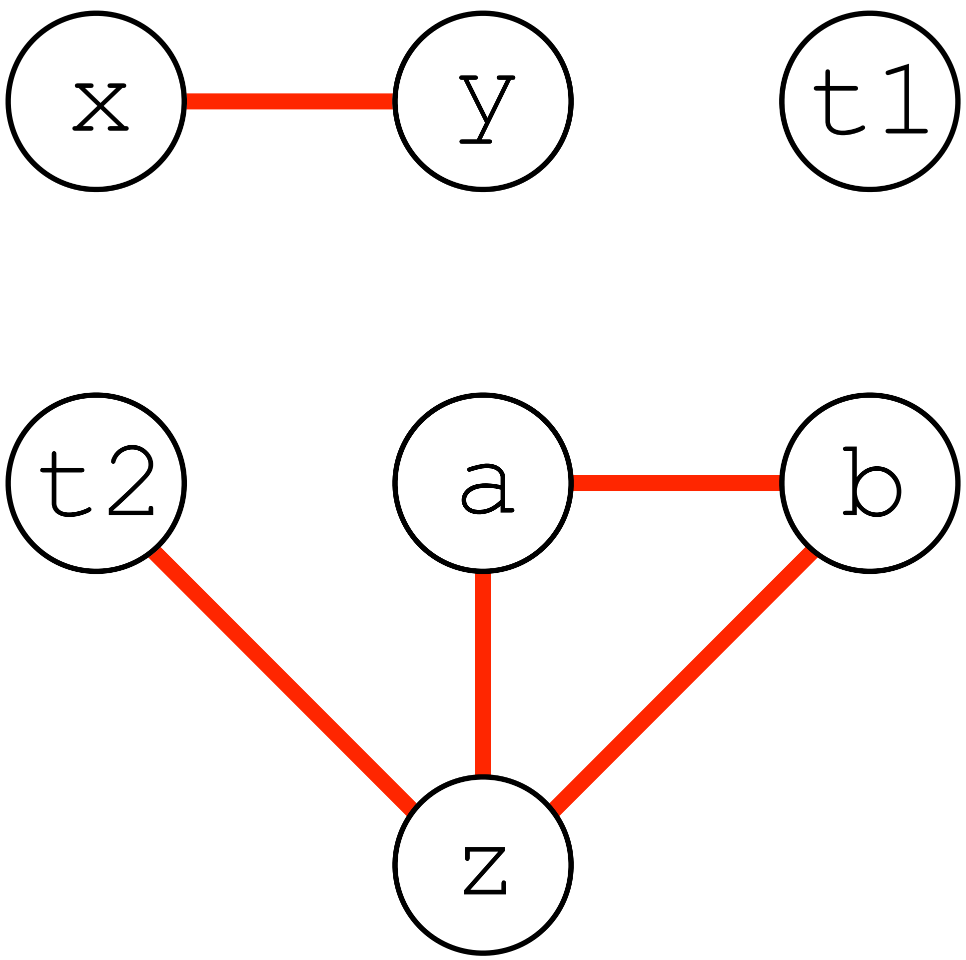
```
{ }
{ x }
{ x, y }
{ t1 }
{ z }
{ a, z }
{ a, b, z }
{ t2, z }
```

Clash graphs



{ }
{ x }
{ x, y }
{ t1 }
{ z }
{ a, z }
{ a, b, z }
{ t2, z }

Clash graphs




```
MOV a,#11
MOV b,#13
ADD a ,a,b
MUL z,a ,#2
MOV a,#17
MOV b,#19
MUL a ,a,b
ADD z,z,a
```

Summary

- Data-flow analysis is helpful in locating (and sometimes correcting) data-flow anomalies
- LVA allows us to identify dead code and possible uses of uninitialised variables
- Write-write anomalies can be identified with a similar analysis
- Imprecision may lead to overzealous warnings
- LVA allows us to construct a clash graph

Lecture 6
Register allocation

Motivation

Normal form is convenient for intermediate code.

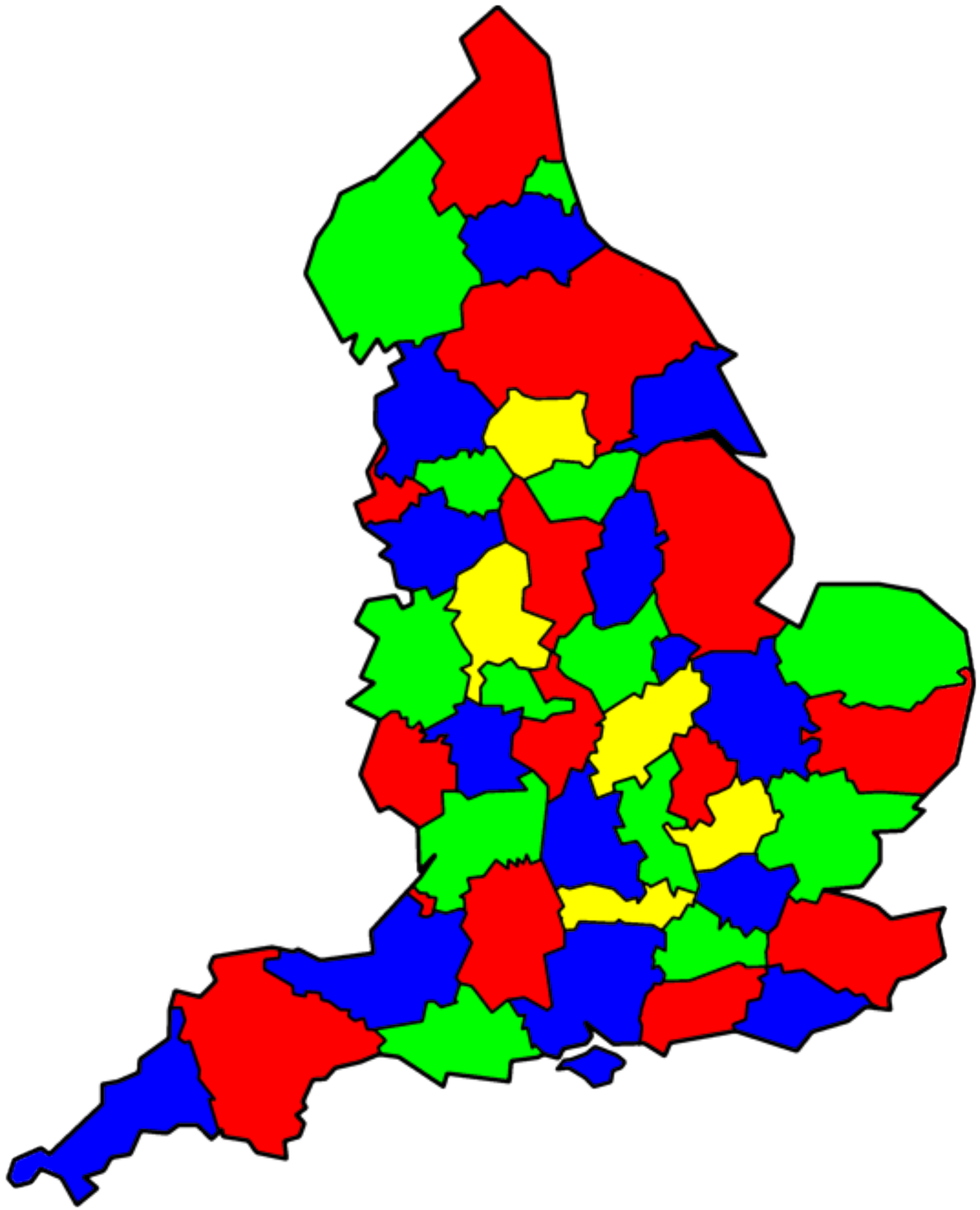
However, it's extremely wasteful.

Real machines only have a small finite number of registers, so at some stage we need to analyse and transform the intermediate representation of a program so that it only requires as many (architectural) registers as are really available.

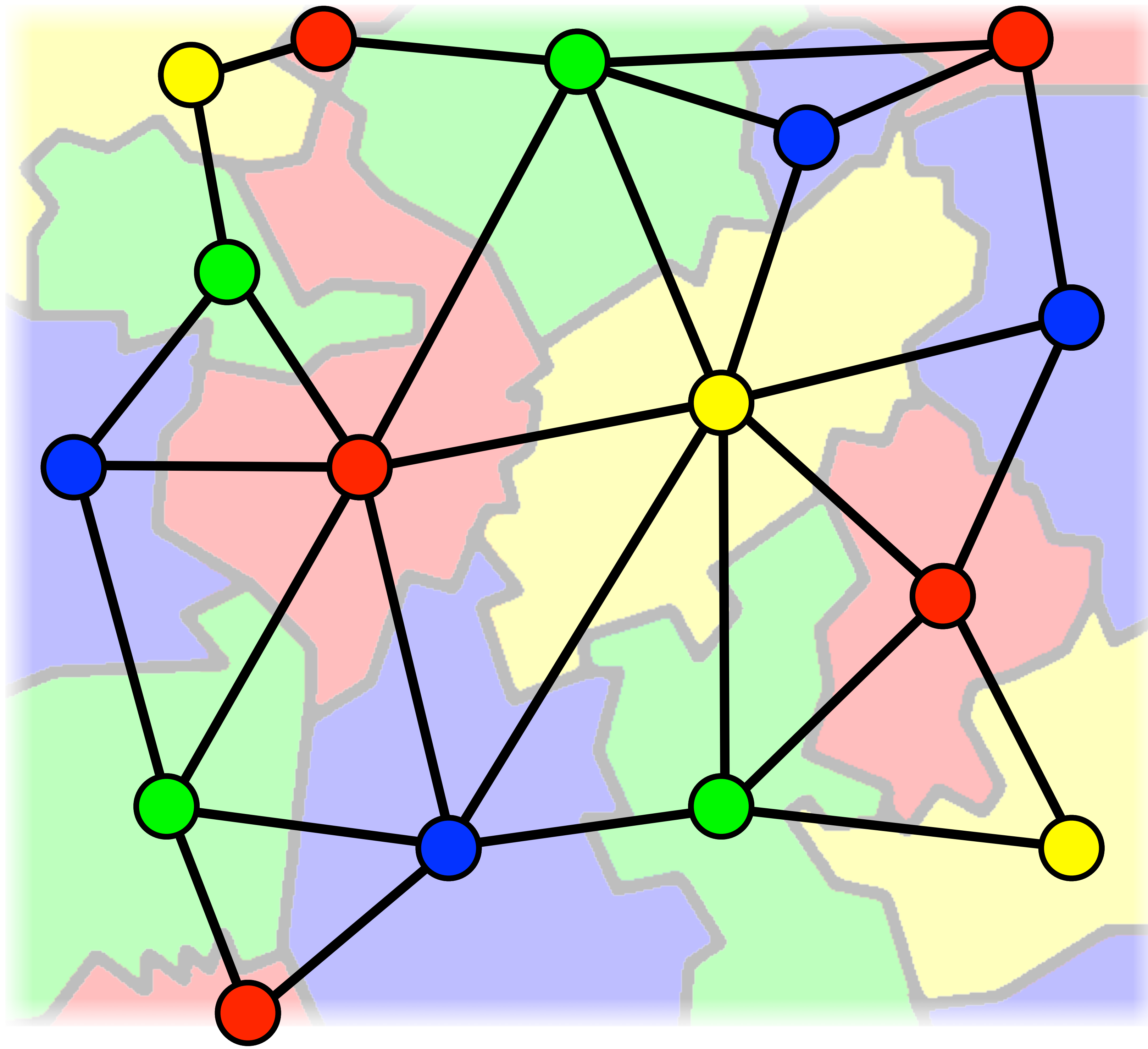
This task is called *register allocation*.

Graph colouring

Register allocation depends upon the solution of a closely related problem known as *graph colouring*.



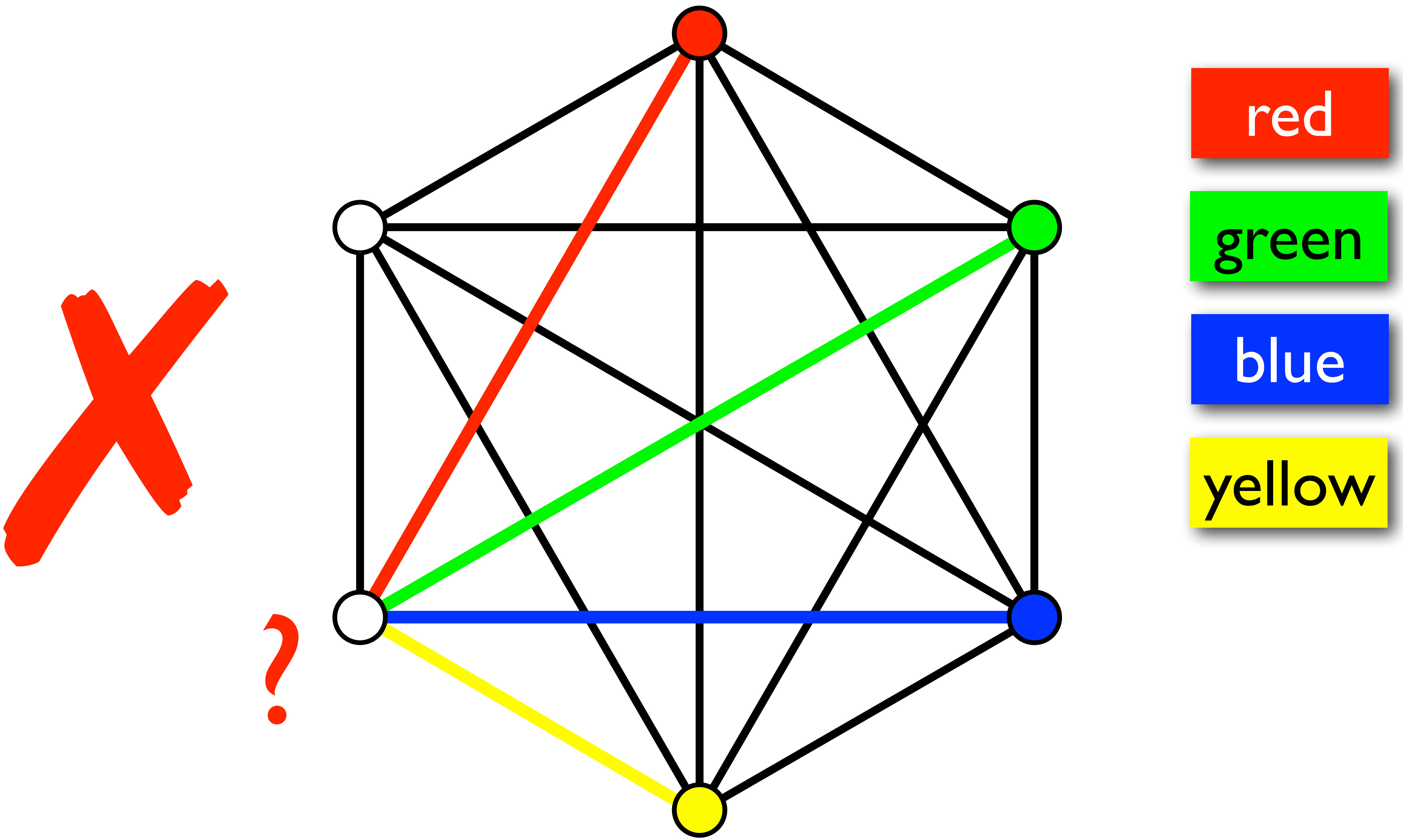
Graph colouring



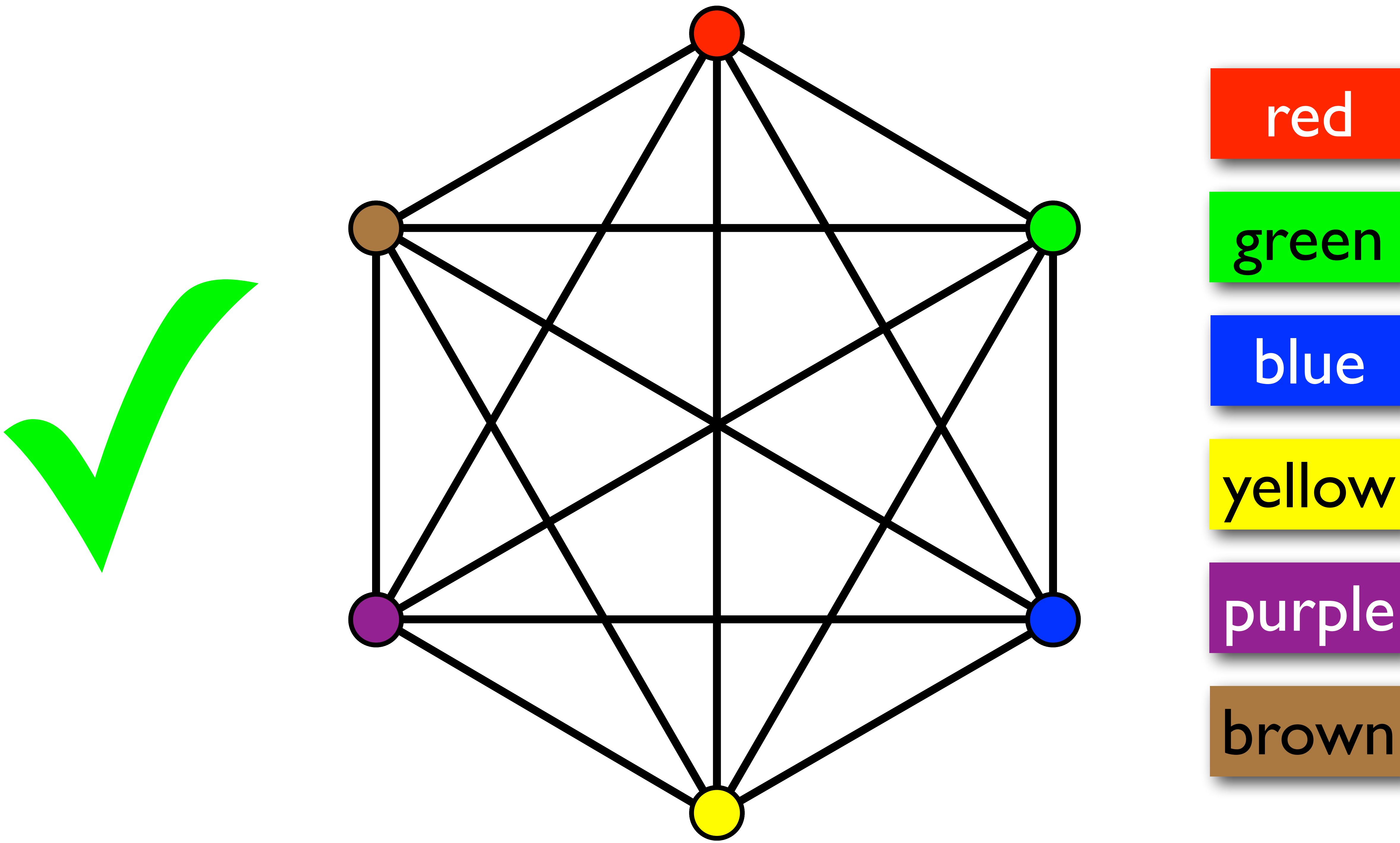
Graph colouring

For general (non-planar) graphs, however, four colours are not sufficient; there is no bound on how many may be required.

Graph colouring



Graph colouring



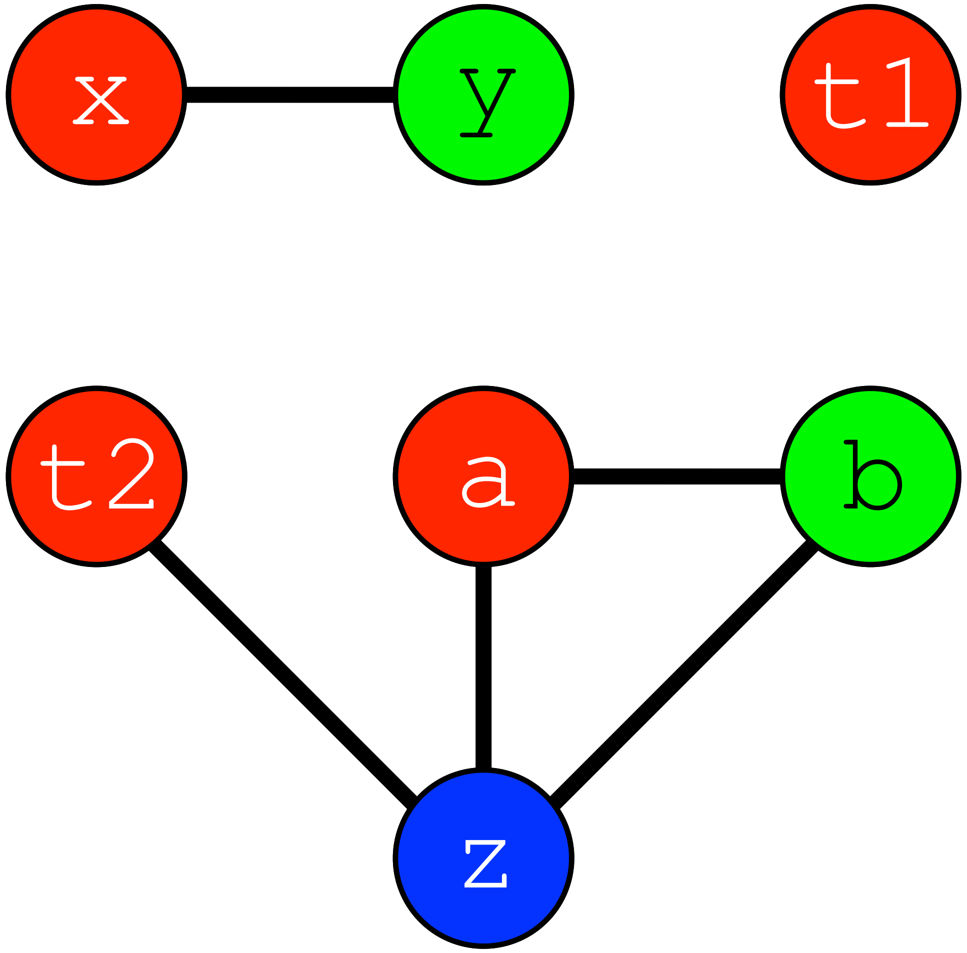
Allocation by colouring

This is essentially the same problem that we wish to solve for clash graphs.

- How many colours (i.e. *architectural* registers) are necessary to colour a clash graph such that no two connected vertices have the same colour (i.e. such that no two simultaneously live virtual registers are stored in the same arch. register)?
- What colour should each vertex be?

Allocation by colouring

```
MOV r0,#11
MOV r1,#13
ADD r0,r0,r1
MUL r2,r0,#2
MOV r0,#17
MOV r1,#19
MUL r0,r0,r1
ADD r2,r2,r0
```



Algorithm

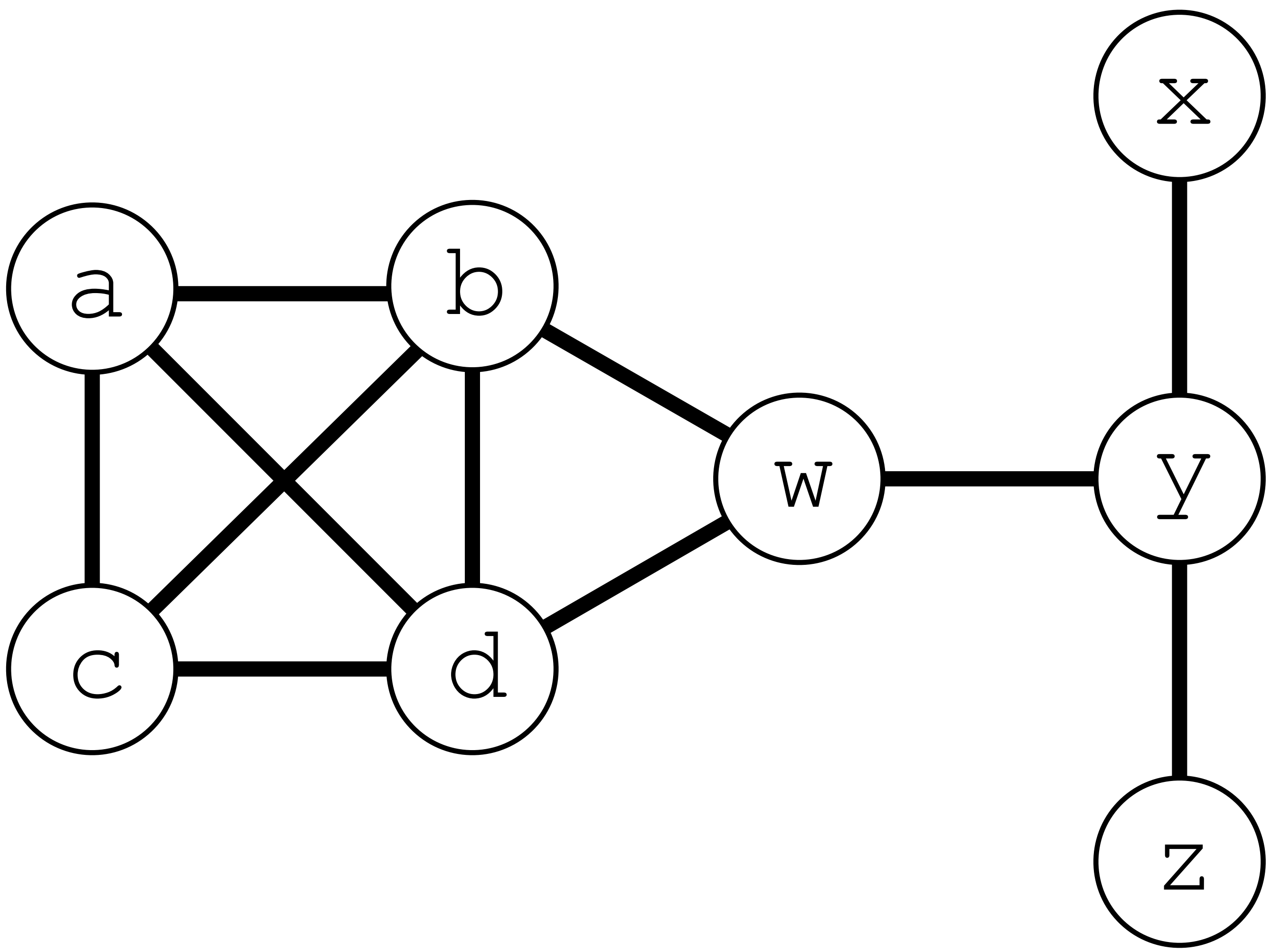
Finding the minimal colouring for a graph is NP-hard, and therefore difficult to do efficiently.

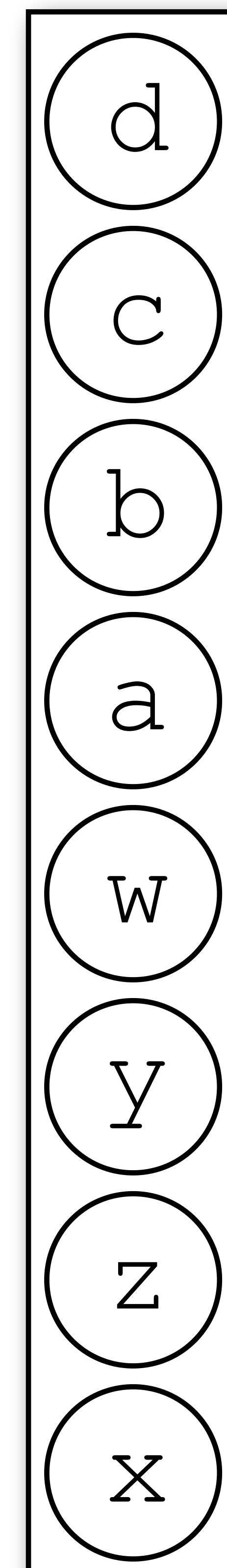
However, we may use a simple heuristic algorithm which chooses a sensible order in which to colour vertices and usually yields satisfactory results on real clash graphs.

Algorithm

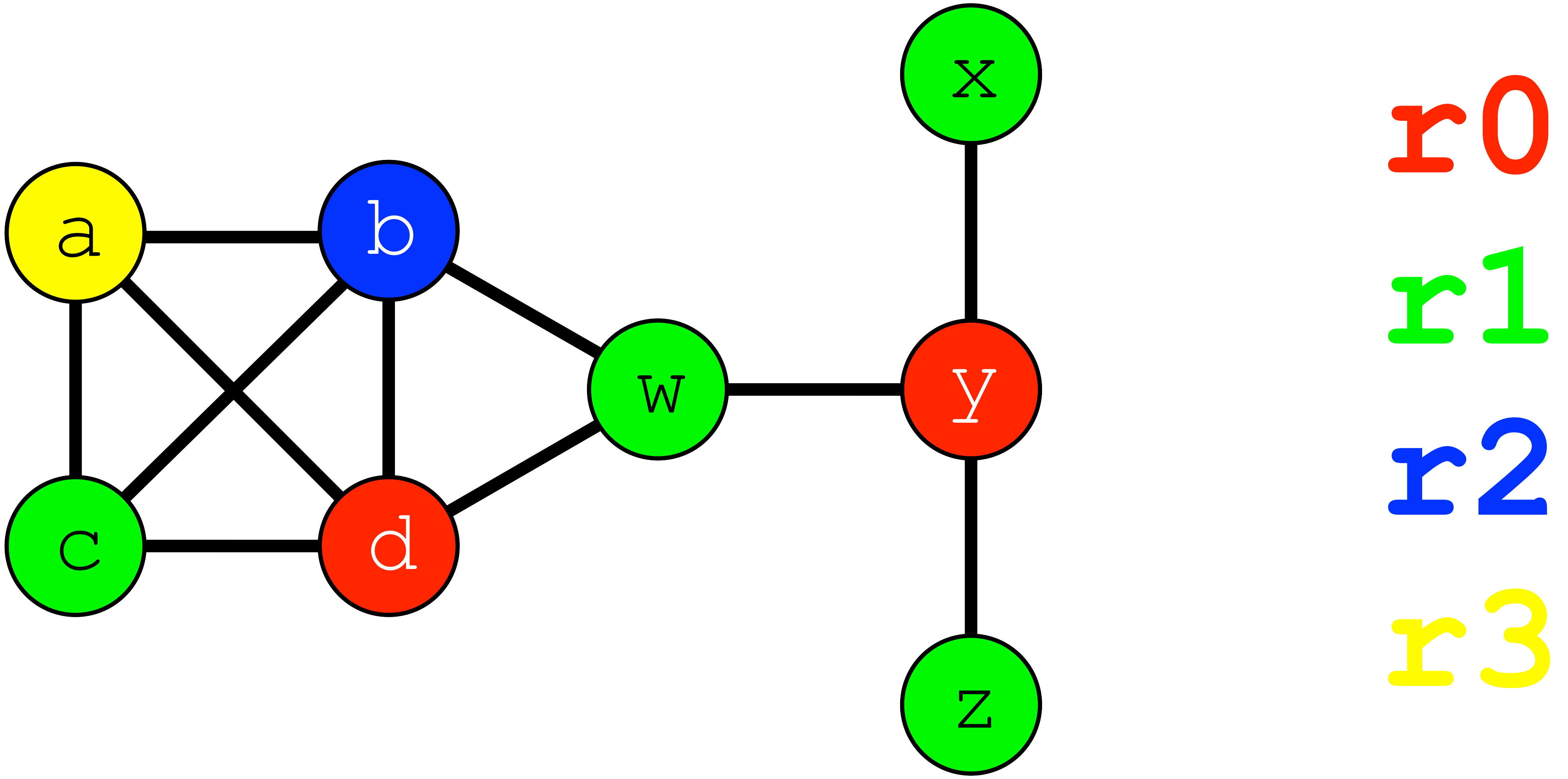
- Choose a vertex (i.e. virtual register) which has the least number of incident edges (i.e. clashes).
- Remove the vertex and its edges from the graph, and push the vertex onto a LIFO stack.
- Repeat until the graph is empty.
- Pop each vertex from the stack and colour it in the most conservative way which avoids the colours of its (already-coloured) neighbours.

Algorithm



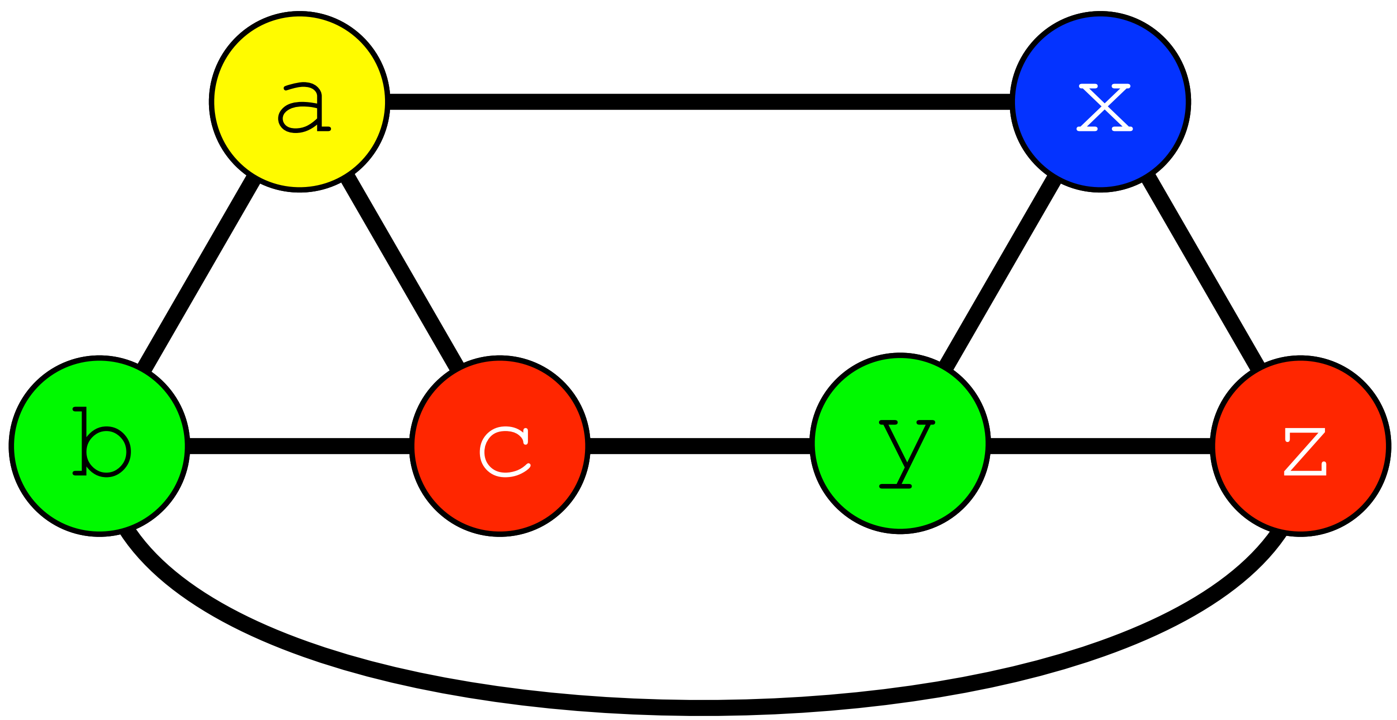


Algorithm



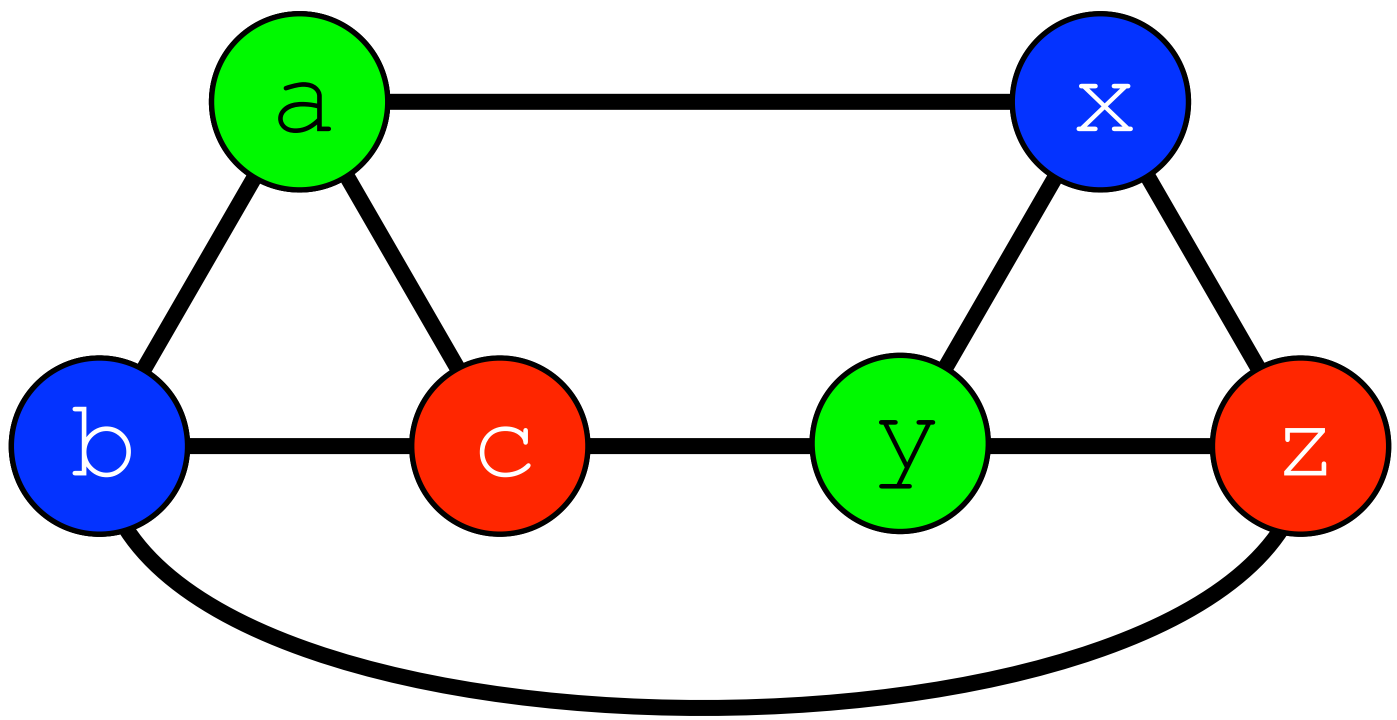
Algorithm

Bear in mind that this is only a heuristic.



Algorithm

Bear in mind that this is only a heuristic.



A better (more minimal) colouring may exist.

Spilling

This algorithm tries to find an approximately minimal colouring of the clash graph, but it assumes new colours are always available when required.

In reality we will usually have a finite number of colours (i.e. architectural registers) available; how should the algorithm cope when it runs out of colours?

Spilling

The quantity of architectural registers is strictly limited, but it is usually reasonable to assume that fresh memory locations will always be available.

So, when the number of simultaneously live values exceeds the number of architectural registers, we may *spill* the excess values into memory.

Operating on values in memory is of course much slower, but it gets the job done.

Spilling

```
ADD a,b,c
```

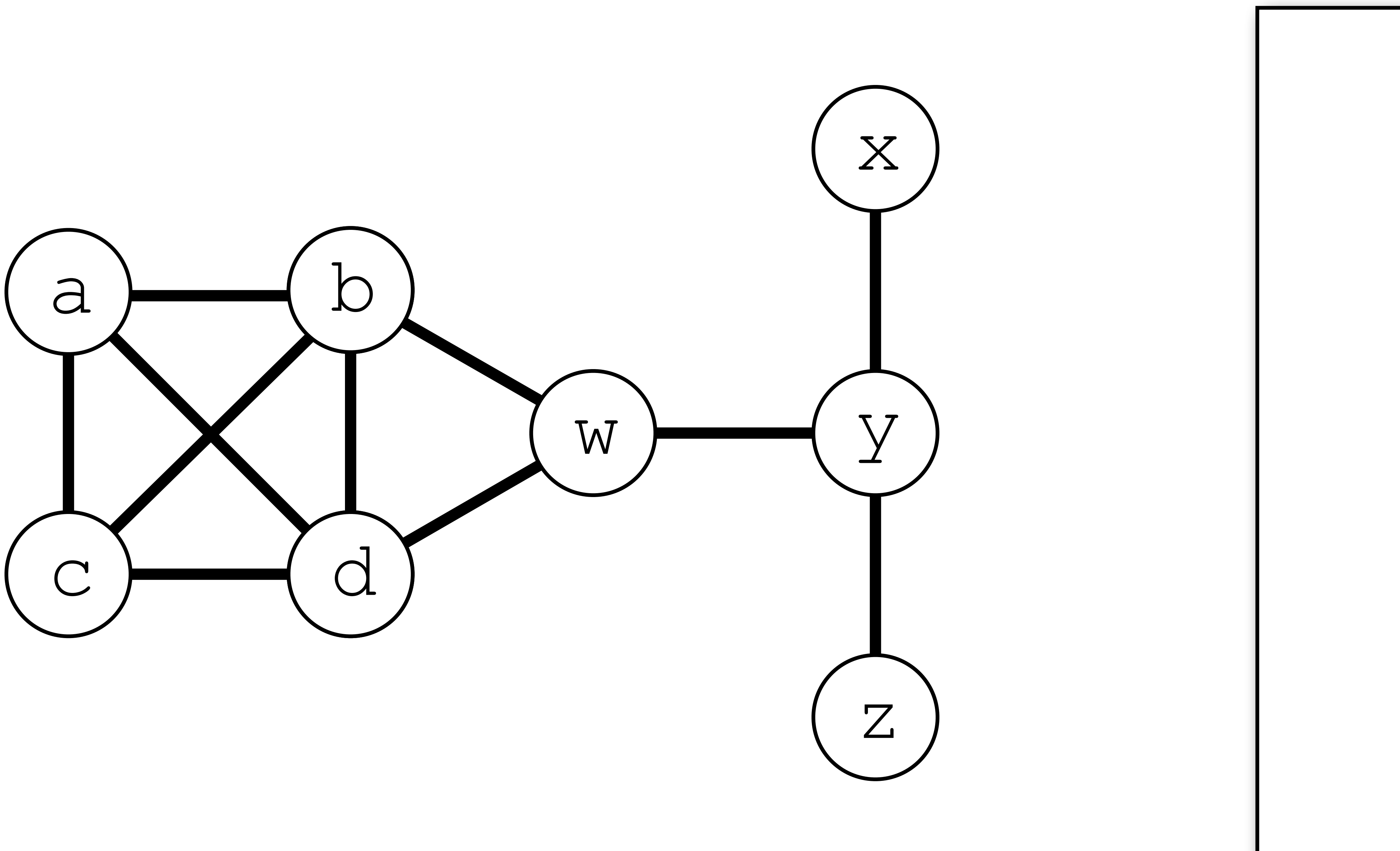
vs.

```
LDR t1,#0xFFA4
LDR t2,#0xFFA8
ADD t3,t1,t2
STR t3,#0xFFA0
```

Algorithm

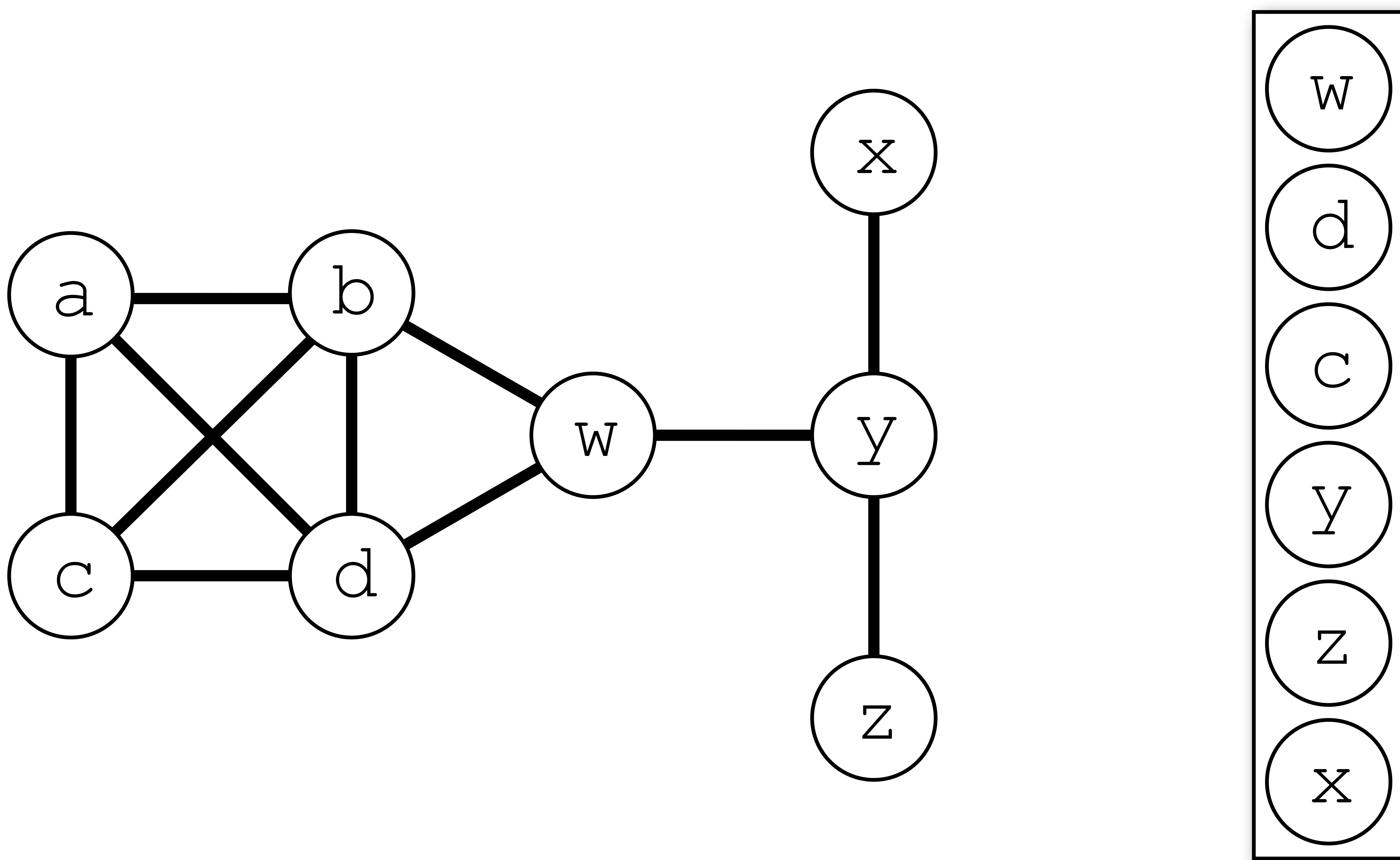
- Choose a vertex with the least number of edges.
- If it has fewer edges than there are colours,
    - remove the vertex and push it onto a stack,
    - otherwise choose a register to spill — e.g. the least-accessed one — and remove its vertex.
- Repeat until the graph is empty.
- Pop each vertex from the stack and colour it.
- Any uncoloured vertices must be spilled.

Algorithm

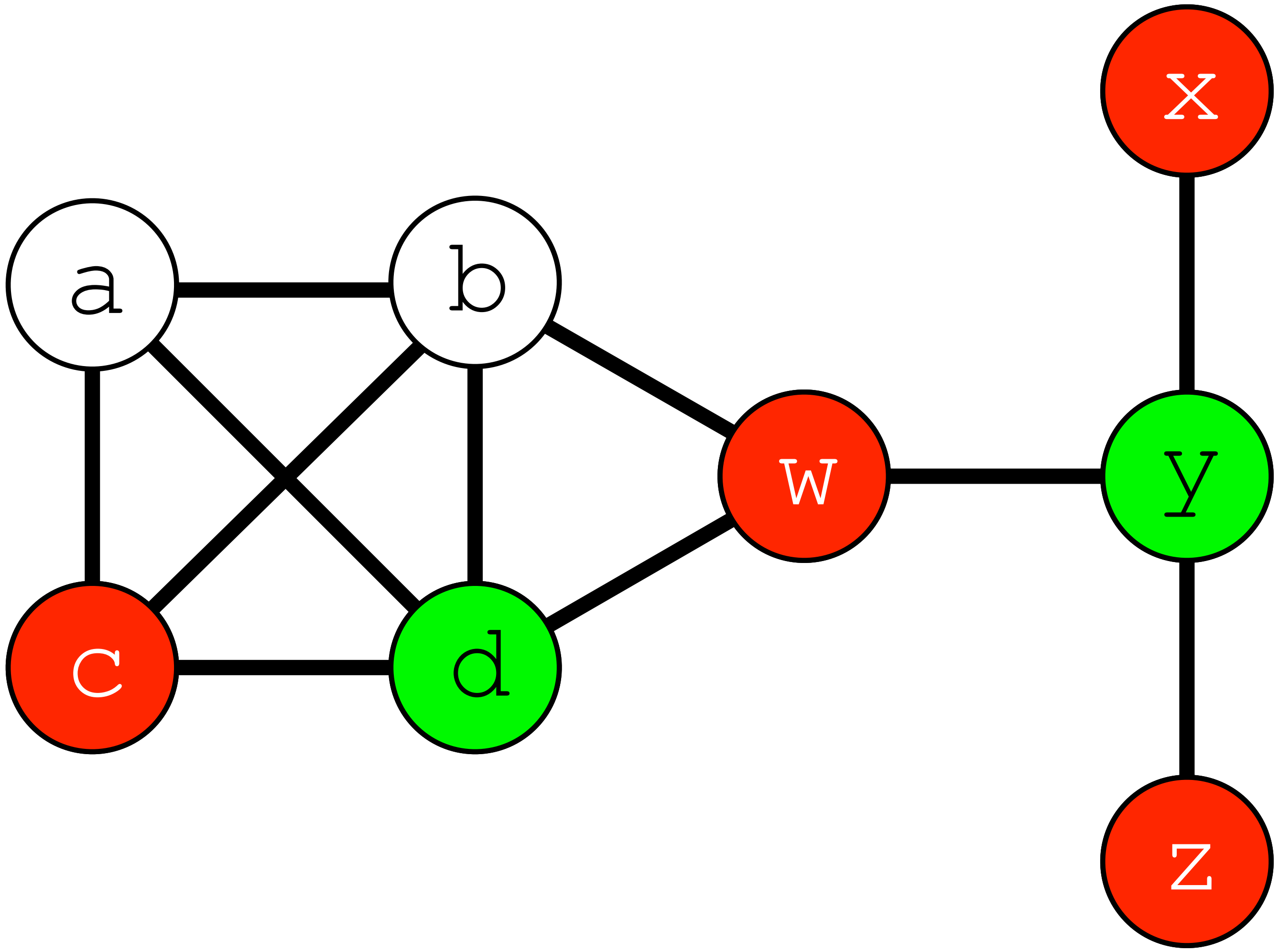


a: 3, b: 5, c: 7, d: 11, w: 13, x: 17, y: 19, z: 23

Algorithm



Algorithm



r0
r1

a and b
spilled to memory

a: 3, b: 5, c: 7, d: 11, w: 13, x: 17, y: 19, z: 23

Algorithm

Choosing the right virtual register to spill will result in a faster, smaller program.

The static count of "how many accesses?" is a good start, but doesn't take account of more complex issues like loops and simultaneous liveness with other spilled values.

One easy heuristic is to treat one static access inside a loop as (say) 4 accesses; this generalises to 4^n accesses inside a loop nested to level *n*.

Algorithm

"Slight lie": when spilling to memory, we (normally) need one free register to use as temporary storage for values loaded from and stored back into memory.

If any instructions operate on two spilled values simultaneously, we may need *two* such temporary registers to store both values.

So, in practise, when a spill is detected we may need to restart register allocation with one (or two) fewer architectural registers available so that these can be kept free for temporary storage of spilled values.

Algorithm

When we are popping vertices from the stack and assigning colours to them, we sometimes have more than one colour to choose from.

If the program contains an instruction "`MOV a,b`" then storing `a` and `b` in the same arch. register (as long as they don't clash) will allow us to delete that instruction.

We can construct a *preference graph* to show which pairs of registers appear together in `MOV` instructions, and use it to guide colouring decisions.

Non-orthogonal instructions

We have assumed that we are free to choose architectural registers however we want to, but this is simply not the case on some architectures.

- The x86 `MUL` instruction expects one of its arguments in the `AL` register and stores its result into `AX`.
- The VAX `MOVC3` instruction zeroes `r0`, `r2`, `r4` and `r5`, storing its results into `r1` and `r3`.

We must be able to cope with such irregularities.

Non-orthogonal instructions

We can handle the situation tidily by pre-allocating a virtual register to each of the target machine's arch. registers, e.g. keep `v0` in `r0`, `v1` in `r1`, ..., `v31` in `r31`.

When generating intermediate code in normal form, we avoid this set of registers, and use new ones (e.g. `v32`, `v33`, ...) for temporaries and user variables.

In this way, each architectural register is explicitly represented by a unique virtual register.

Non-orthogonal instructions

We must now do extra work when generating intermediate code:

- When an instruction requires an operand in a specific arch. register (e.g. x86 `MUL`), we generate a preceding `MOV` to put the right value into the corresponding virtual register.
- When an instruction produces a result in a specific arch. register (e.g. x86 `MUL`), we generate a trailing `MOV` to transfer the result into a new virtual register.

Non-orthogonal instructions

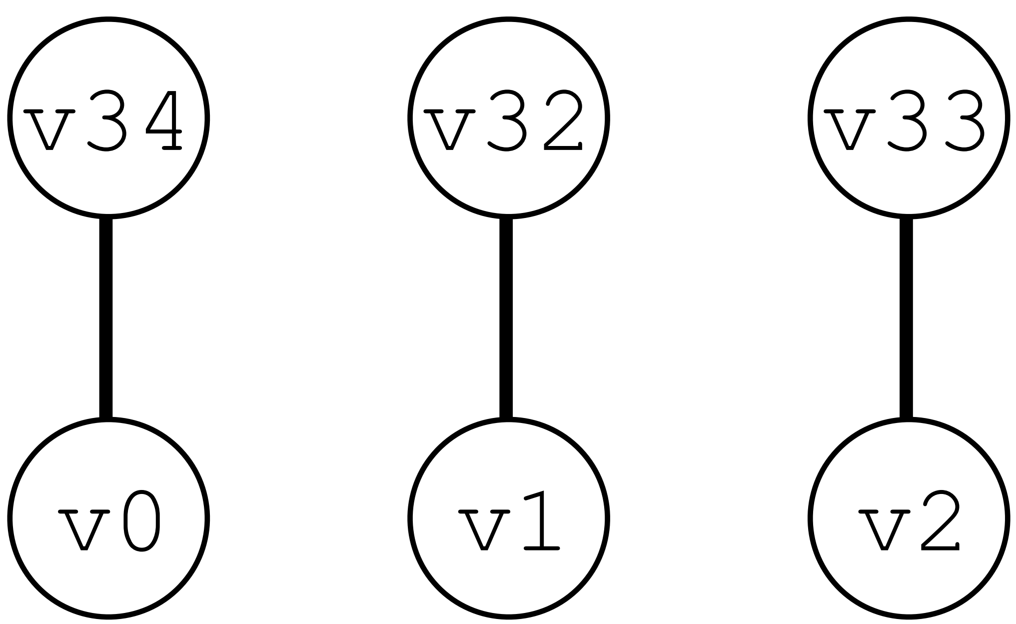
If (hypothetically) `ADD` on the target architecture can only perform `r0 = r1 + r2`:

```
x = 19;
y = 23;
z = x + y;
```

```
MOV v32,#19
MOV v33,#23
MOV v1,v32
MOV v2,v33
ADD v0,v1,v2
MOV v34,v0
```

Non-orthogonal instructions

This may seem particularly wasteful, but many of the `MOV` instructions will be eliminated during register allocation if a preference graph is used.

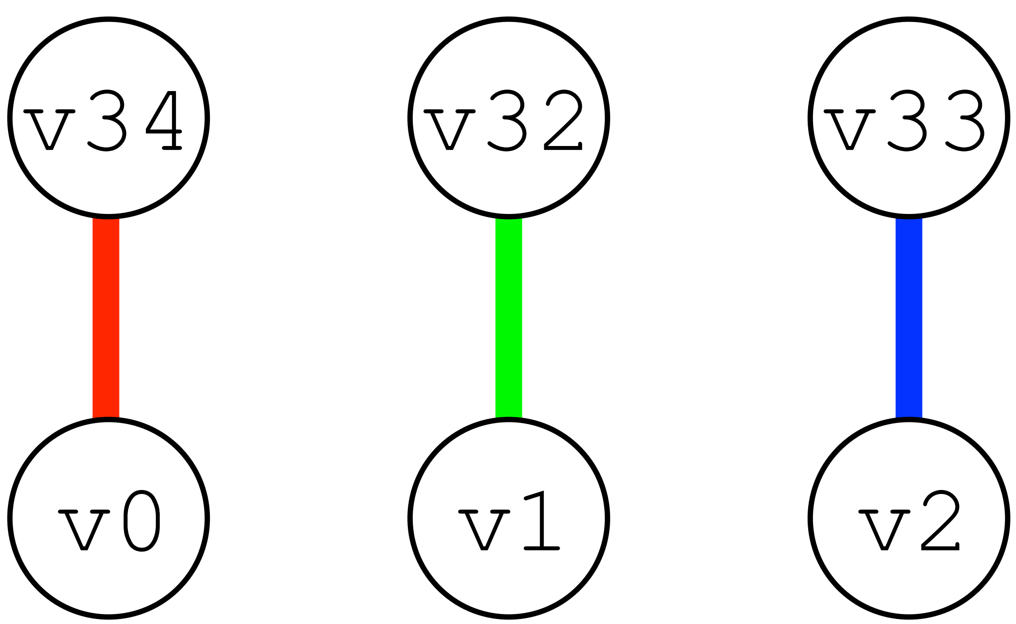


preference graph

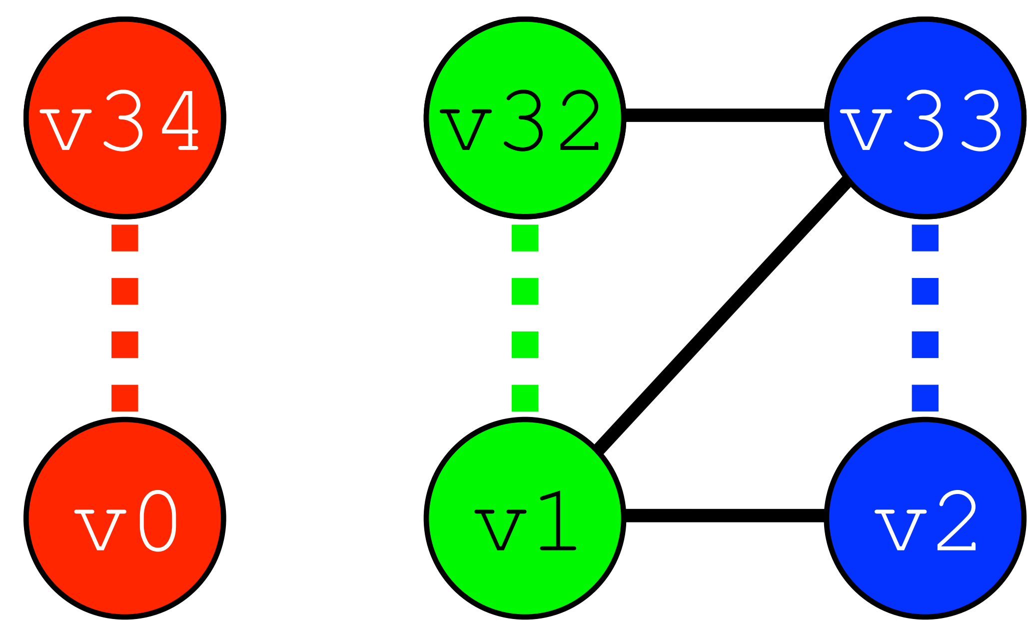
```
MOV v32,#19
MOV v33,#23
MOV v1,v32
MOV v2,v33
ADD v0,v1,v2
MOV v34,v0
```

Non-orthogonal instructions

This may seem particularly wasteful, but many of the `MOV` instructions will be eliminated during register allocation if a preference graph is used.



preference graph

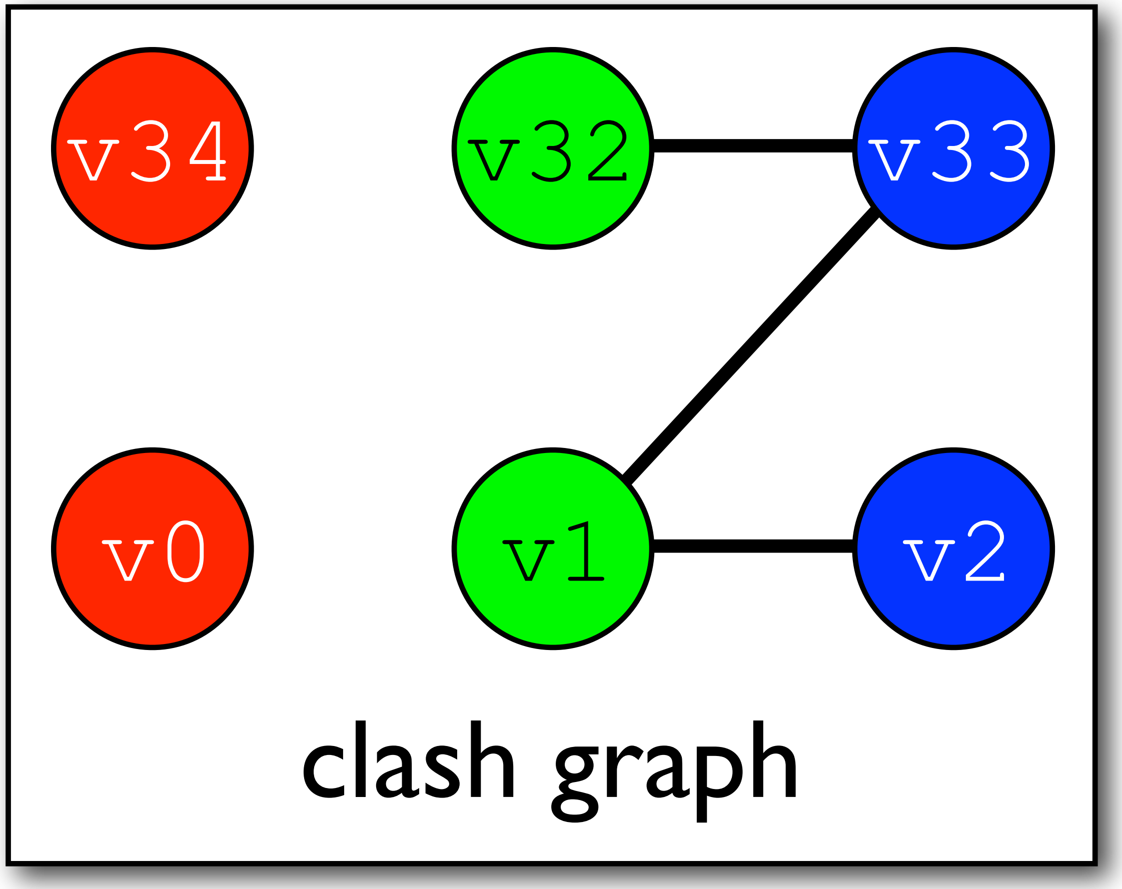


clash graph

Non-orthogonal instructions

This may seem particularly wasteful, but many of the `MOV` instructions will be eliminated during register allocation if a preference graph is used.

```
MOV r1,#19
MOV r2,#23
MOV r1,r1
MOV r2,r2
ADD r0,r1,r2
MOV r0,r0
```



Non-orthogonal instructions

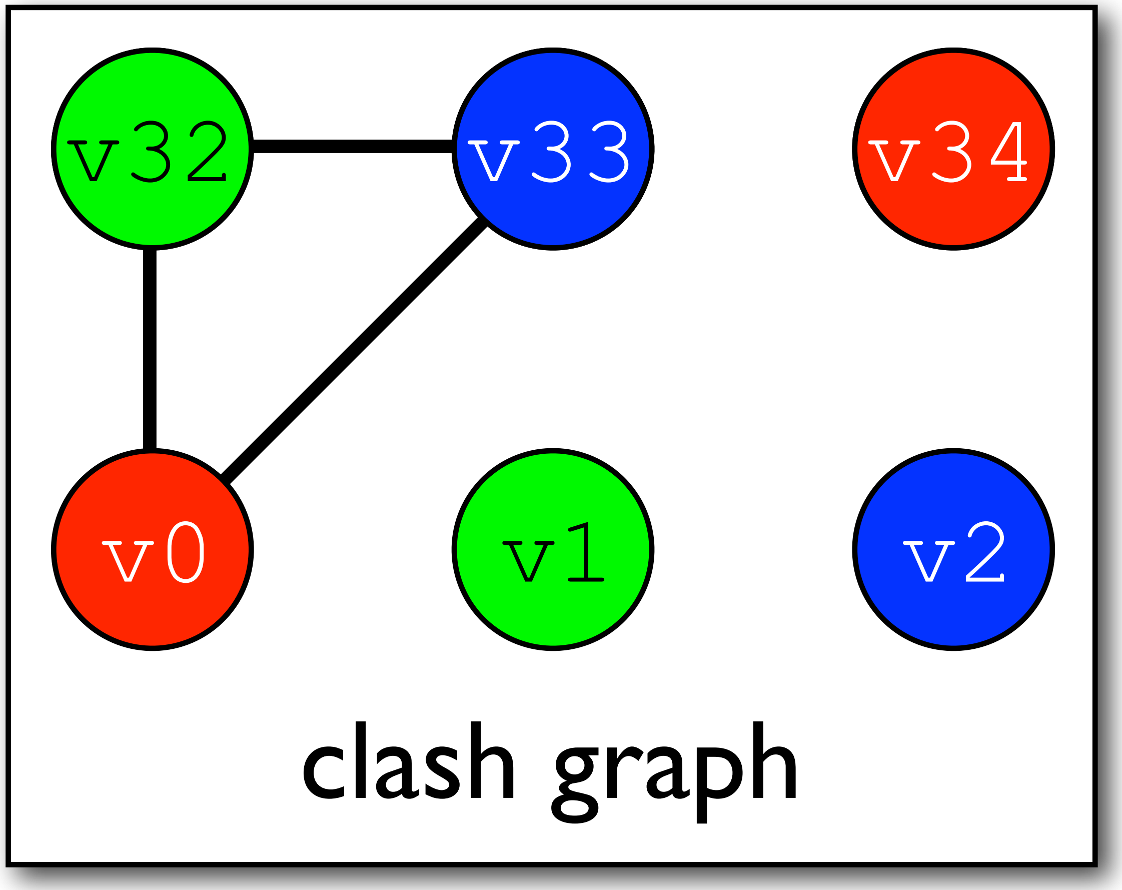
And finally,

- When we know an instruction is going to corrupt the contents of an architectural register, we insert an edge on the clash graph between the corresponding virtual register and all other virtual registers live at that instruction — this prevents the register allocator from trying to store any live values in the corrupted register.

Non-orthogonal instructions

If (hypothetically) `MUL` on the target architecture corrupts the contents of `r0`:

```
MOV r1,#6
MOV r2,#7
MUL r0,r1,r2
⋮
```



Procedure calling standards

This final technique of synthesising edges on the clash graph in order to avoid corrupted registers is helpful for dealing with the procedure calling standard of the target architecture.

Such a standard will usually dictate that procedure calls (e.g. `CALL` and `CALLI` instructions in our 3-address code) should use certain registers for arguments and results, should preserve certain registers over a call, and may corrupt any other registers if necessary.

Procedure calling standards

On the ARM, for example:

- Arguments should be placed in `r0`-`r3` before a procedure is called.
- Results should be returned in `r0` and `r1`.
- `r4`-`r8`, `r10` and `r11` should be preserved over procedure calls, and `r9` might be depending on the platform.
- `r12`-`r15` are special registers, including the stack pointer and program counter.

Procedure calling standards

Since a procedure call instruction may corrupt some of the registers (`r0`-`r3` and possibly `r9` on the ARM), we can synthesise edges on the clash graph between the corrupted registers and all other virtual registers live at the call instruction.

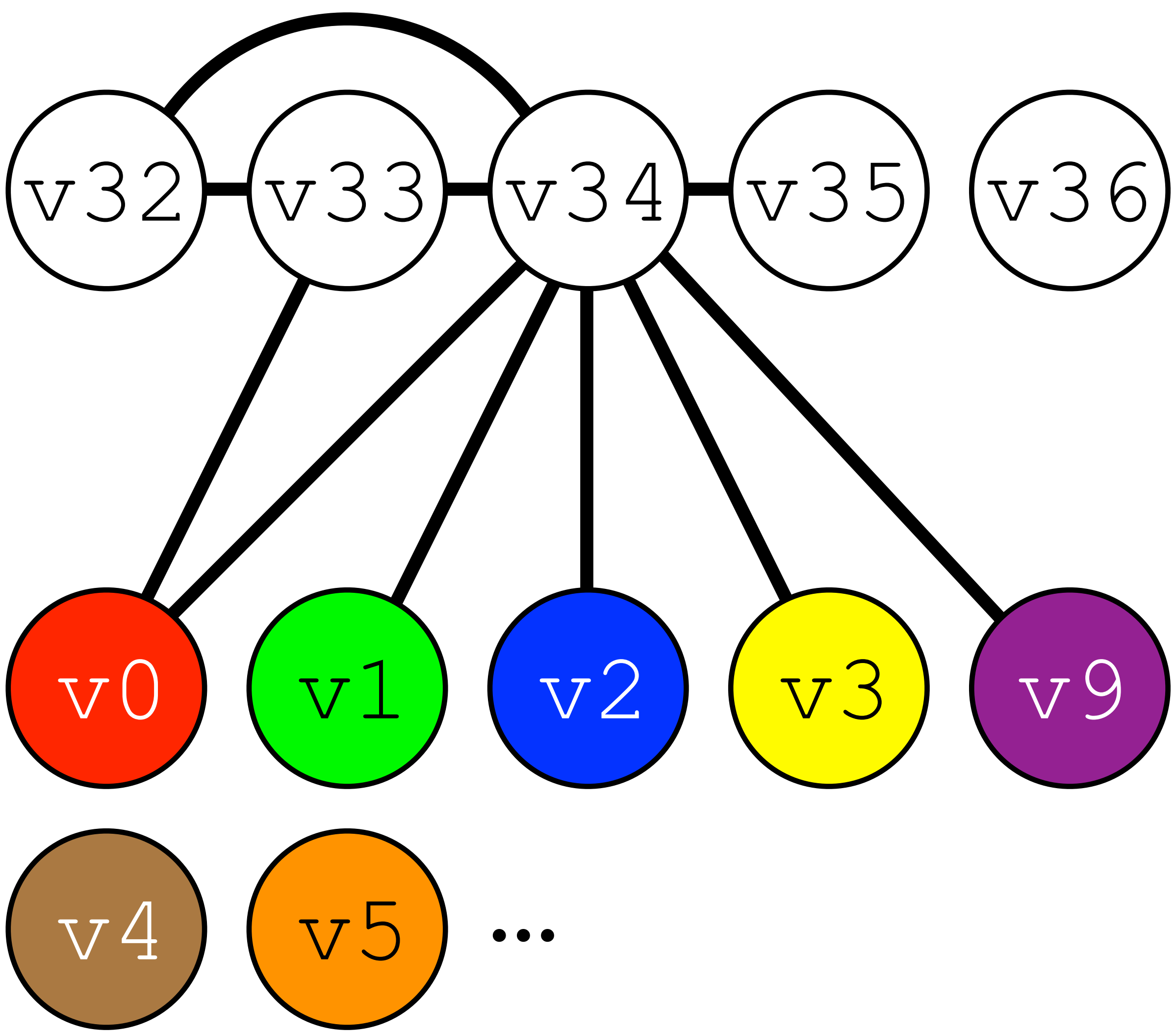
As before, we may also synthesise `MOV` instructions to ensure that arguments and results end up in the correct registers, and use the preference graph to guide colouring such that most of these `MOV`s can be deleted again.

Procedure calling standards

```
x = 7;
y = 11;
z = 13;
a = f(x,y)+z;
```

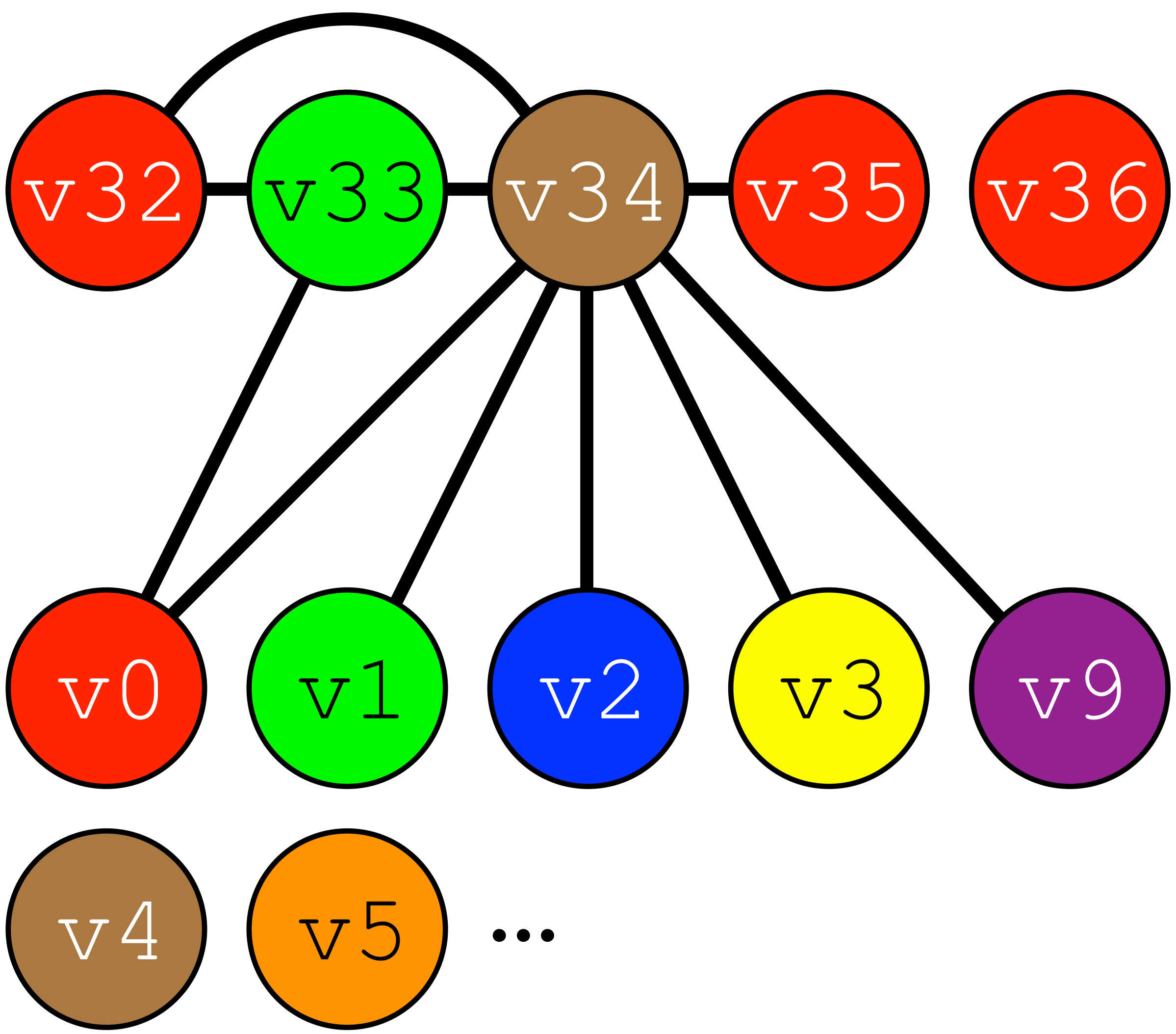
```
MOV v32,#7
MOV v33,#11
MOV v34,#13
MOV v0,v32
MOV v1,v33
CALL f
MOV v35,v0
ADD v36,v34,v35
```

Procedure calling standards




```
MOV v32,#7
MOV v33,#11
MOV v34,#13
MOV v0,v32
MOV v1,v33
CALL f
MOV v35,v0
ADD v36,v34,v35
```

Procedure calling standards




```
MOV r0,#7
MOV r1,#11
MOV r4,#13
MOV r0,r0
MOV r1,r1
CALL f
MOV r0,r0
ADD r0,r4,r0
```

Summary

- A register allocation phase is required to assign each virtual register to an architectural one during compilation
- Registers may be allocated by colouring the vertices of a clash graph
- When the number of arch. registers is limited, some virtual registers may be spilled to memory
- Non-orthogonal instructions may be handled with additional `MOV`s and new edges on the clash graph
- Procedure calling standards also handled this way

Lecture 7
Redundancy elimination

Motivation

Some expressions in a program may cause redundant recomputation of values.

If such recomputation is safely eliminated, the program will usually become faster.

There exist several *redundancy elimination* optimisations which attempt to perform this task in different ways (and for different specific meanings of "redundancy").

Common subexpressions

Common-subexpression elimination is a transformation which is enabled by available-expression analysis (AVAIL), in the same way as LVA enables dead-code elimination.

Since AVAIL discovers which expressions will have been computed by the time control arrives at an instruction in the program, we can use this information to spot and remove redundant computations.

Common subexpressions

Recall that an expression is *available* at an instruction if its value has definitely already been computed and not been subsequently invalidated by assignments to any of the variables occurring in the expression.

If the expression e is available on entry to an instruction which computes e, the instruction is performing a redundant computation and can be modified or removed.

Common subexpressions

We consider this redundantly-computed expression to be a *common subexpression*: it is common to more than one instruction in the program, and in each of its occurrences it may appear as a subcomponent of some larger expression.

```
x = (a*b)+c;
⋮
print a * b;
```

`a*b` AVAILABLE

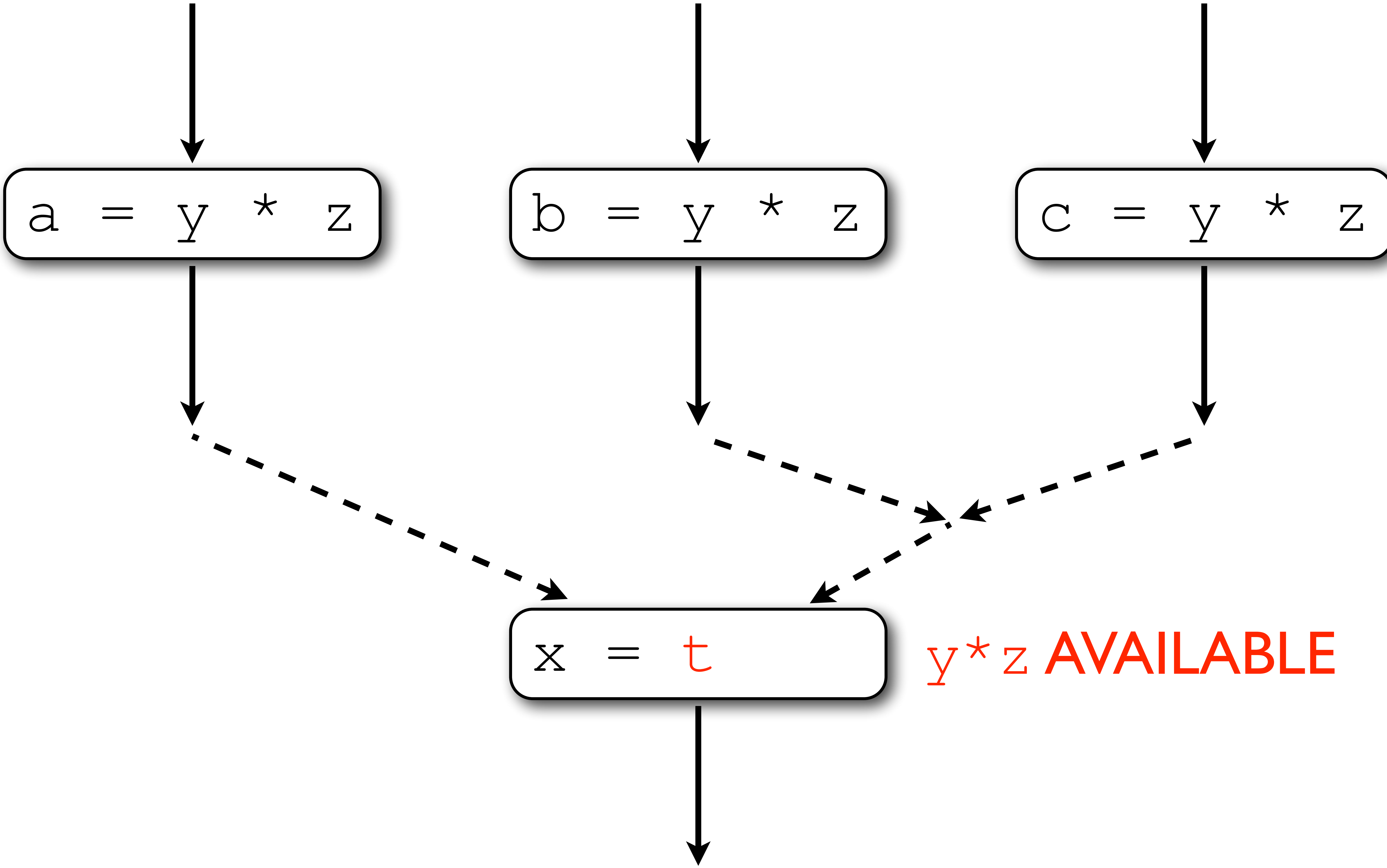
Common subexpressions

We can eliminate a common subexpression by storing its value into a new temporary variable when it is first computed, and reusing that variable later when the same value is required.

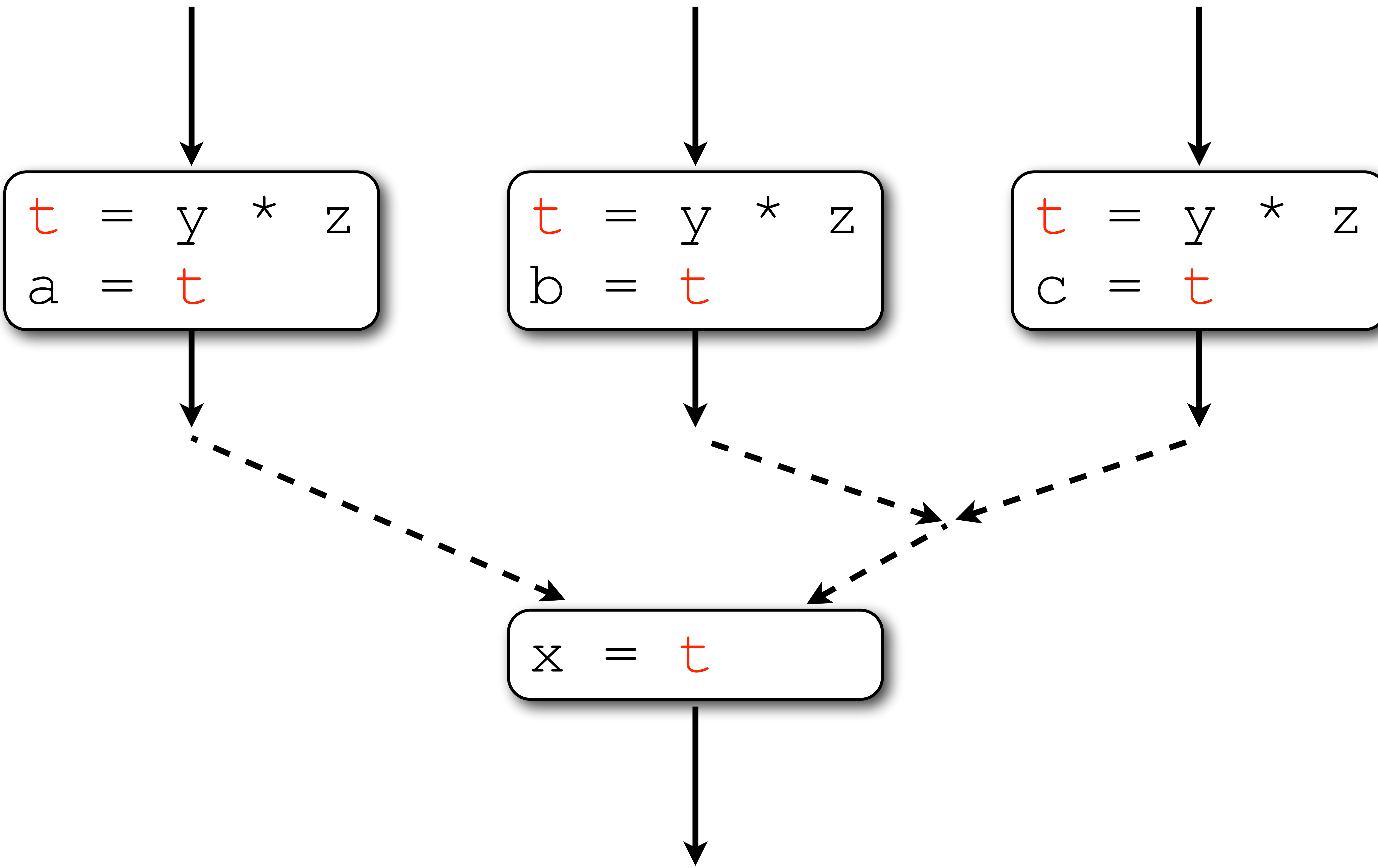
Algorithm

- Find a node *n* which computes an already-available expression *e*
  - Replace the occurrence of e with a new temporary variable *t*
  - On each control path backwards from *n*, find the first instruction calculating e and add a new instruction to store its value into *t*
- Repeat until no more redundancy is found

Algorithm



Algorithm



Common subexpressions

Our transformed program performs (statically) fewer arithmetic operations: `y*z` is now computed in three places rather than four.

However, three register copy instructions have also been generated; the program is now larger, and whether it is faster depends upon characteristics of the target architecture.

Common subexpressions

The program might have “got worse” as a result of performing common-subexpression elimination.

In particular, introducing a new variable increases register pressure, and might cause spilling.

Memory loads and stores are much more expensive than multiplication of registers!

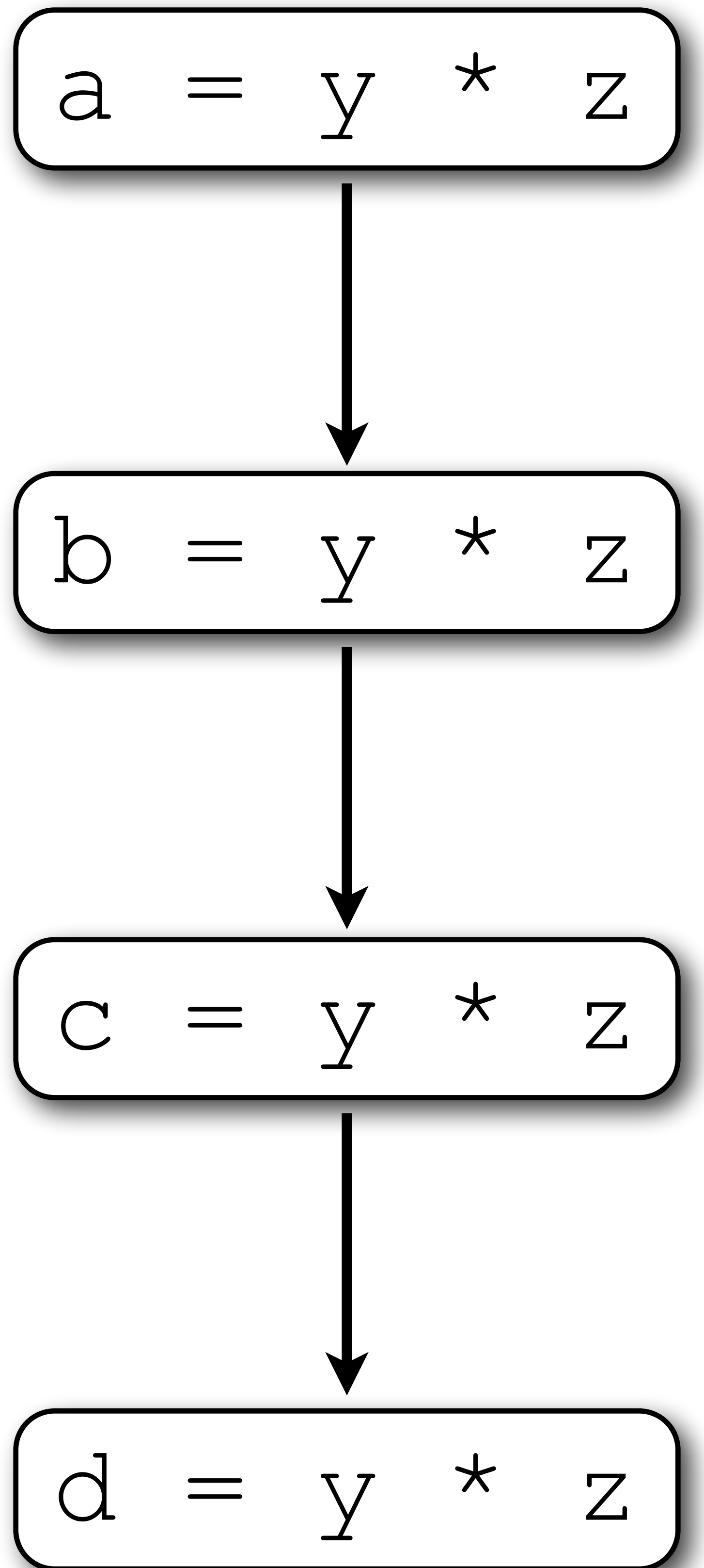
Copy propagation

This simple formulation of CSE is fairly careless, and assumes that other compiler phases are going to tidy up afterwards.

In addition to register allocation, a transformation called *copy propagation* is often helpful here.

In copy propagation, we scan forwards from an `x=y` instruction and replace `x` with `y` wherever it appears (as long as neither `x` nor `y` have been modified).

Copy propagation



Copy propagation

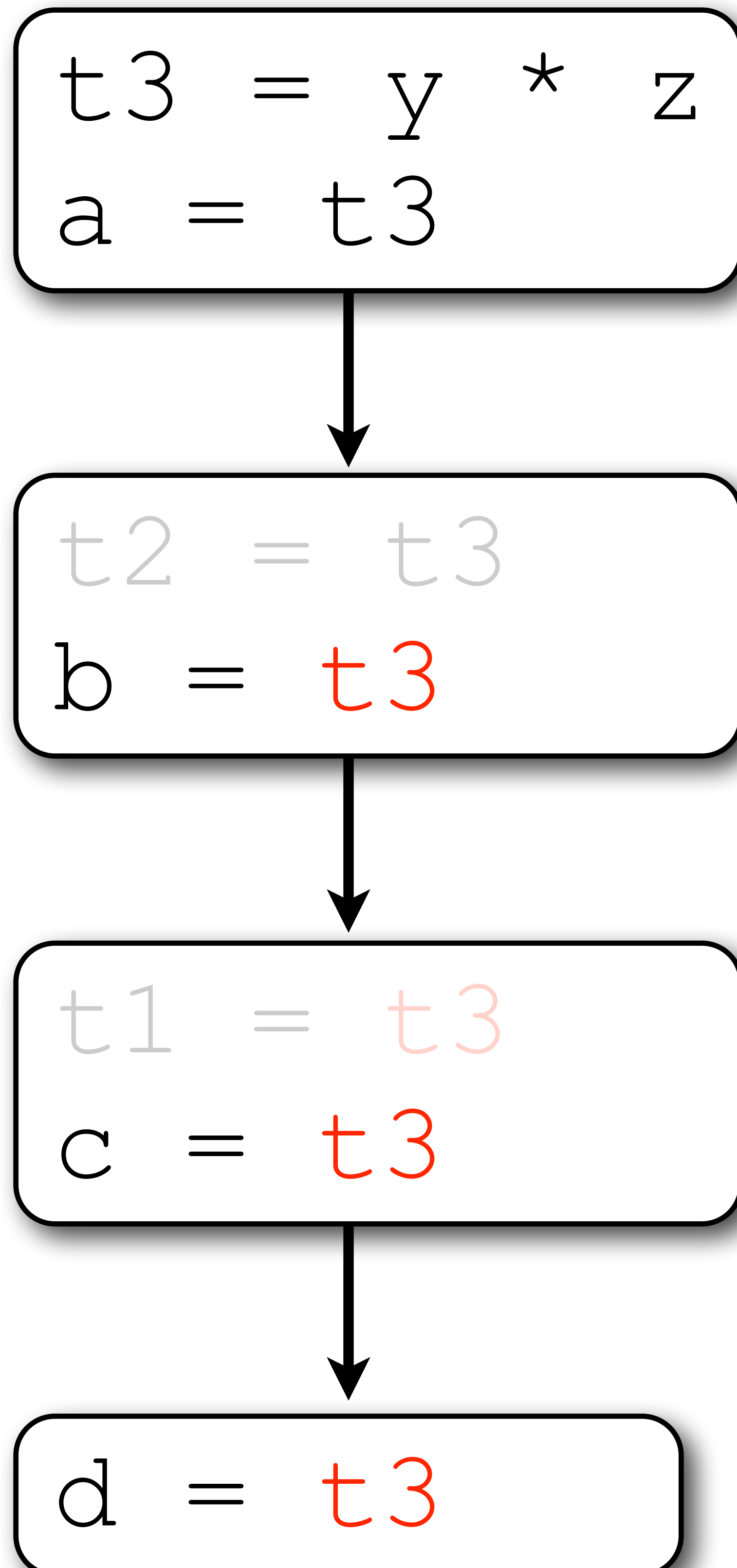
```
t3 = y * z
a = t3
```

```
t2 = y * z
b = t2
```

```
t1 = t2
c = t1
```

```
d = t1
```

Copy propagation



Code motion

Transformations such as CSE are known collectively as *code motion* transformations: they operate by moving instructions and computations around programs to take advantage of opportunities identified by control- and data-flow analysis.

Code motion is particularly useful in eliminating different kinds of redundancy.

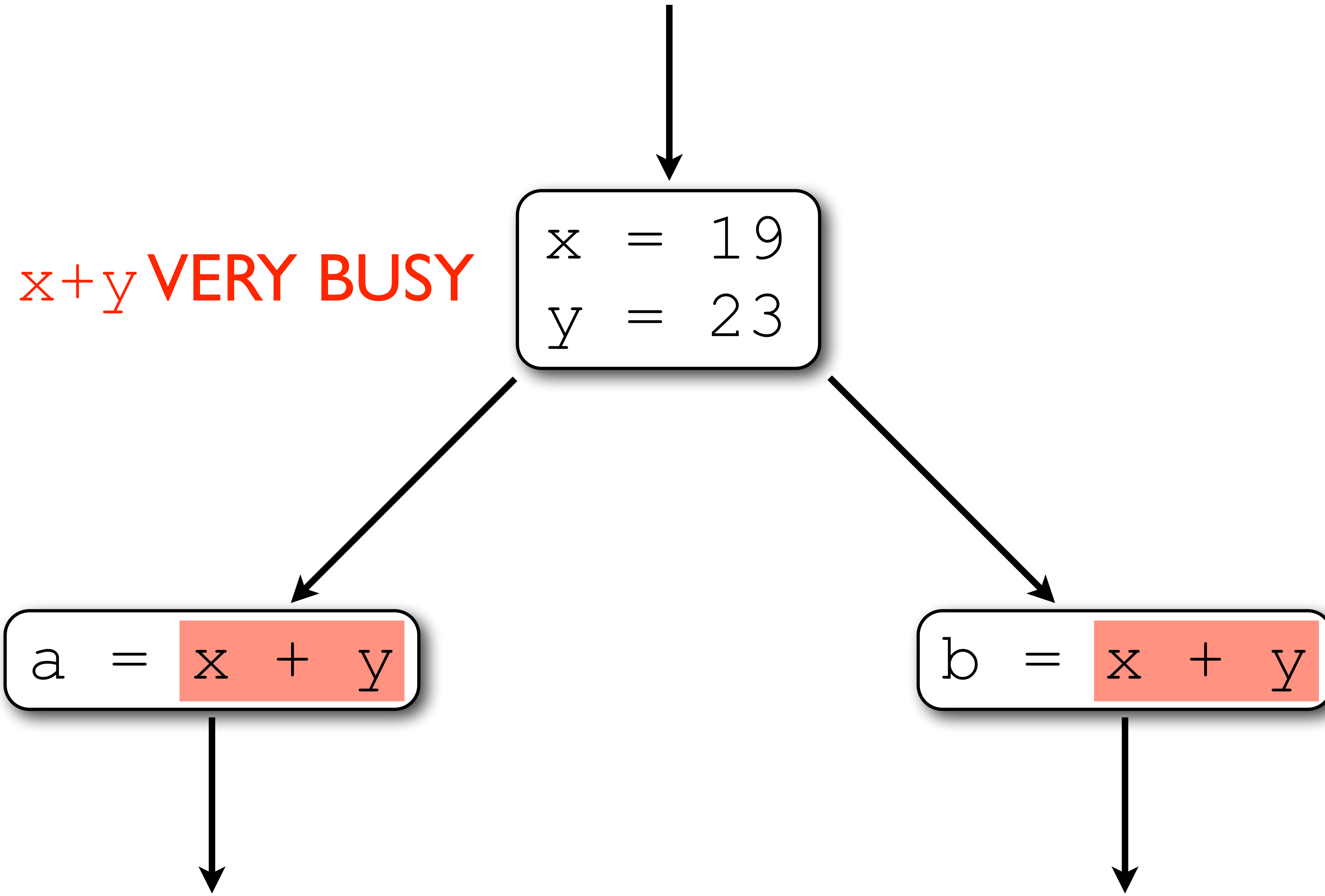
It's worth looking at other kinds of code motion.

Code hoisting

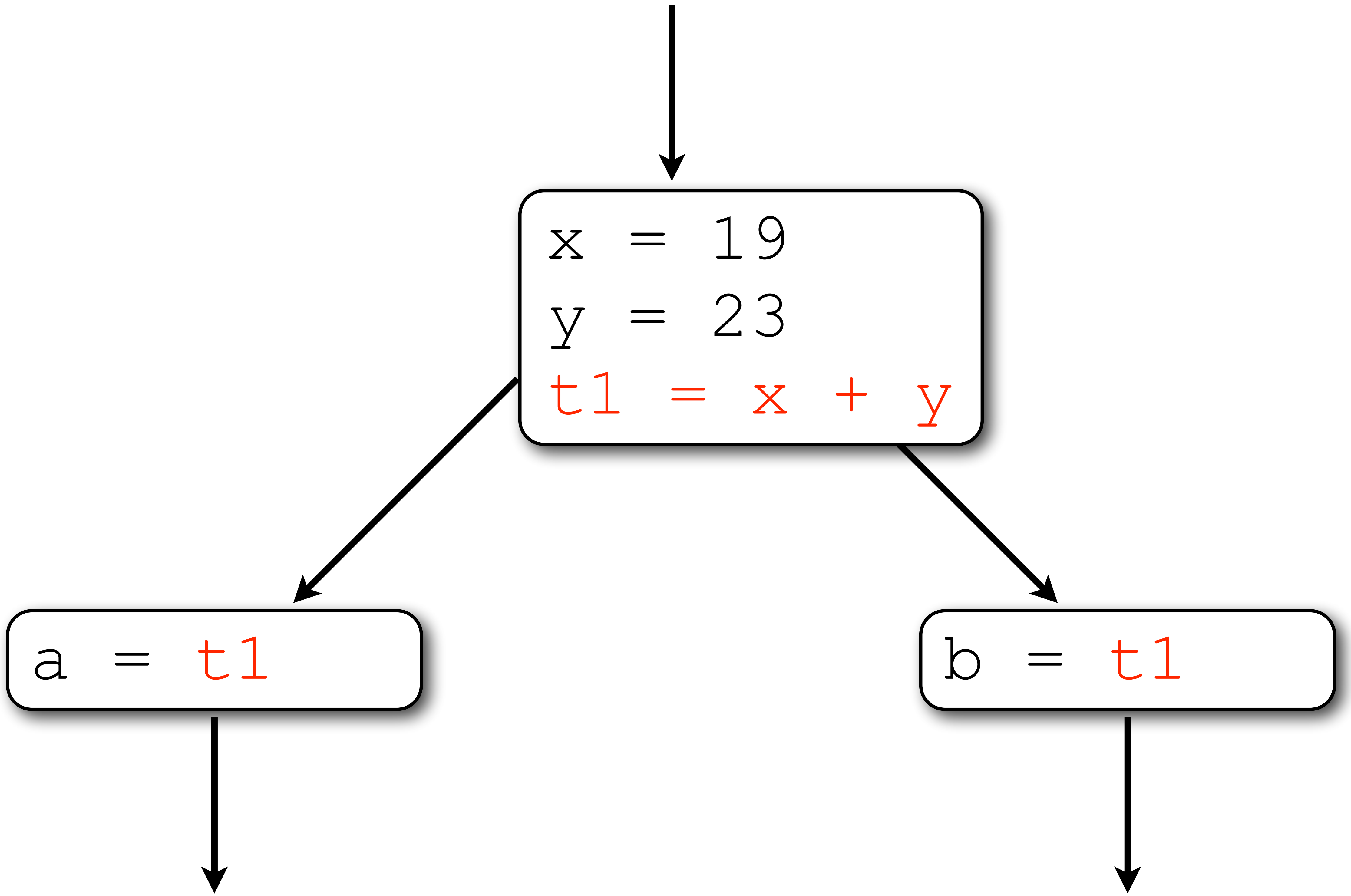
Code hoisting reduces the size of a program by moving duplicated expression computations to the same place, where they can be combined into a single instruction.

Hoisting relies on a data-flow analysis called *very busy expressions* (a backwards version of AVAIL) which finds expressions that are definitely going to be evaluated later in the program; these can be moved earlier and possibly combined with each other.

Code hoisting



Code hoisting



Code hoisting

Hoisting may have a different effect on execution time depending on the exact nature of the code. The resulting program may be slower, faster, or just the same speed as before.

Loop-invariant code motion

Some expressions inside loops are redundant in the sense that they get recomputed on every iteration even though their value never changes within the loop.

Loop-invariant code motion recognises these redundant computations and moves such expressions outside of loop bodies so that they are only evaluated once.

Loop-invariant code motion

```
a = ...;
b = ...;
x = a + b;
while (...) {
   ...
}
print x;
```

Loop-invariant code motion

This transformation depends upon a data-flow analysis to discover which assignments may affect the value of a variable ("reaching definitions").

If none of the variables in the expression are redefined inside the loop body (or are only redefined by computations involving other invariant values), the expression is invariant between loop iterations and may safely be relocated before the beginning of the loop.

Partial redundancy

Partial redundancy elimination combines common-subexpression elimination and loop-invariant code motion into one optimisation which improves the performance of code.

An expression is *partially redundant* when it is computed more than once on *some* (vs. all) paths through a flowgraph; this is often the case for code inside loops, for example.

Partial redundancy

```
a = ...;
b = ...;
... = a + b;
while (...) {
   ... = a + b;
   a = ...;
   ... = a + b;
}
```

Partial redundancy

This example gives a faster program of the same size.

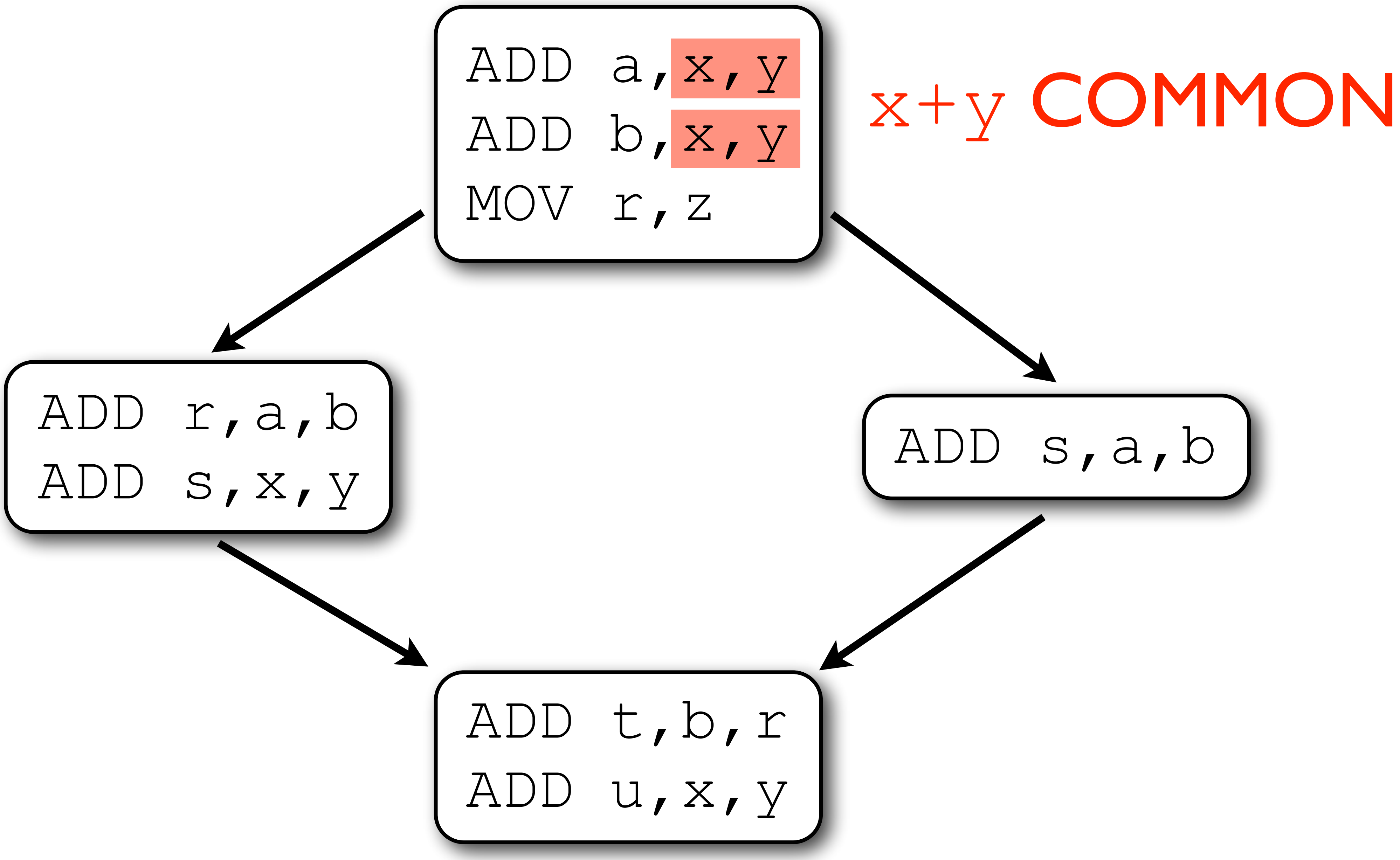
Partial redundancy elimination can be achieved in its own right using a complex combination of several forwards and backwards data-flow analyses in order to locate partially redundant computations and discover the best places to add and delete instructions.

Putting it all together

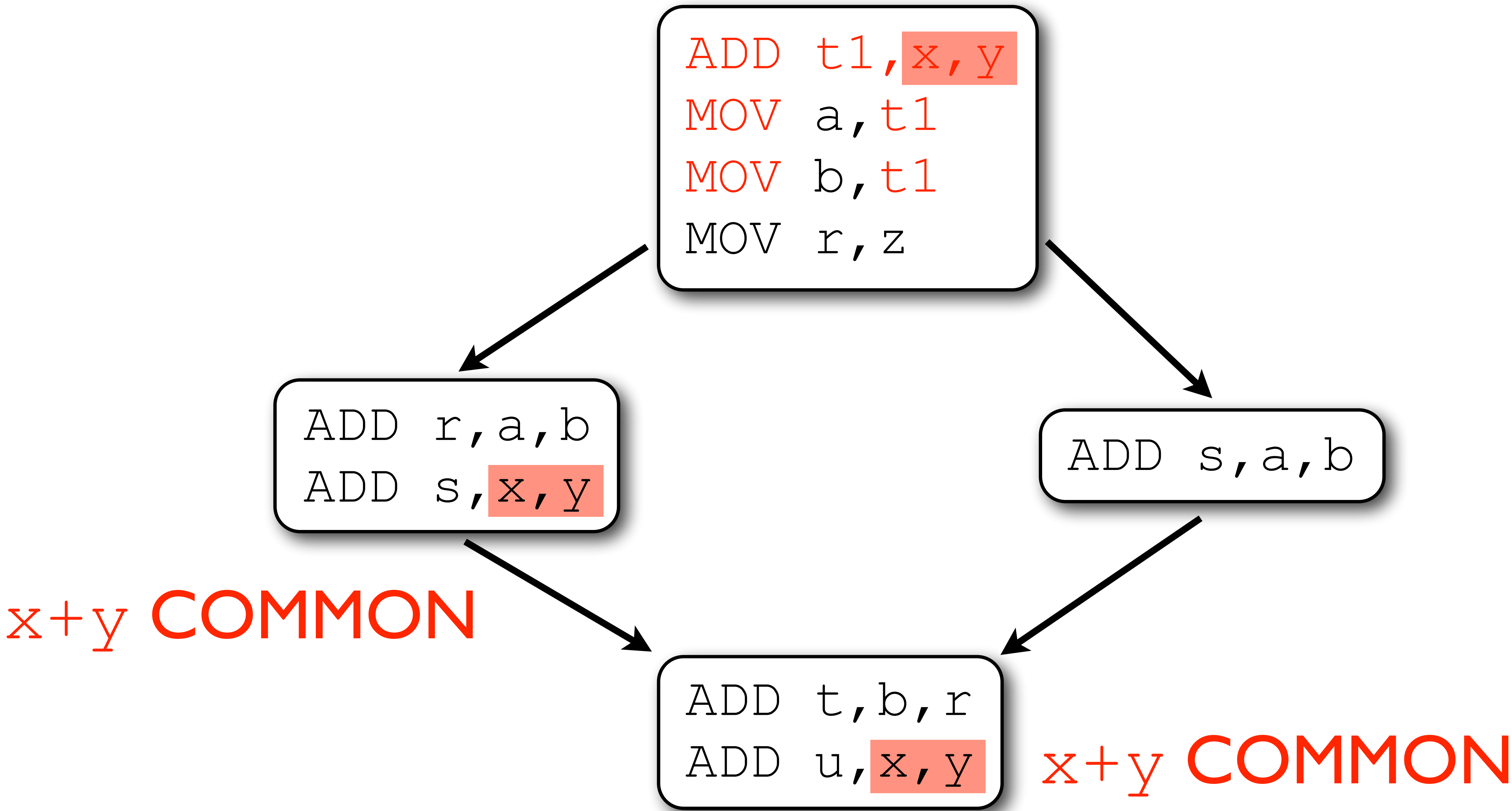
```
a = x + y;
b = x + y;
r = z;
if (a == 42) {
  r = a + b;
  s = x + y;
} else {
  s = a + b;
}
t = b + r;
u = x + y;
⋮
return r+s+t+u;
```

```
ADD a,x,y
ADD b,x,y
MOV r,z

ADD r,a,b
ADD s,x,y

ADD s,a,b

ADD t,b,r
ADD u,x,y
```

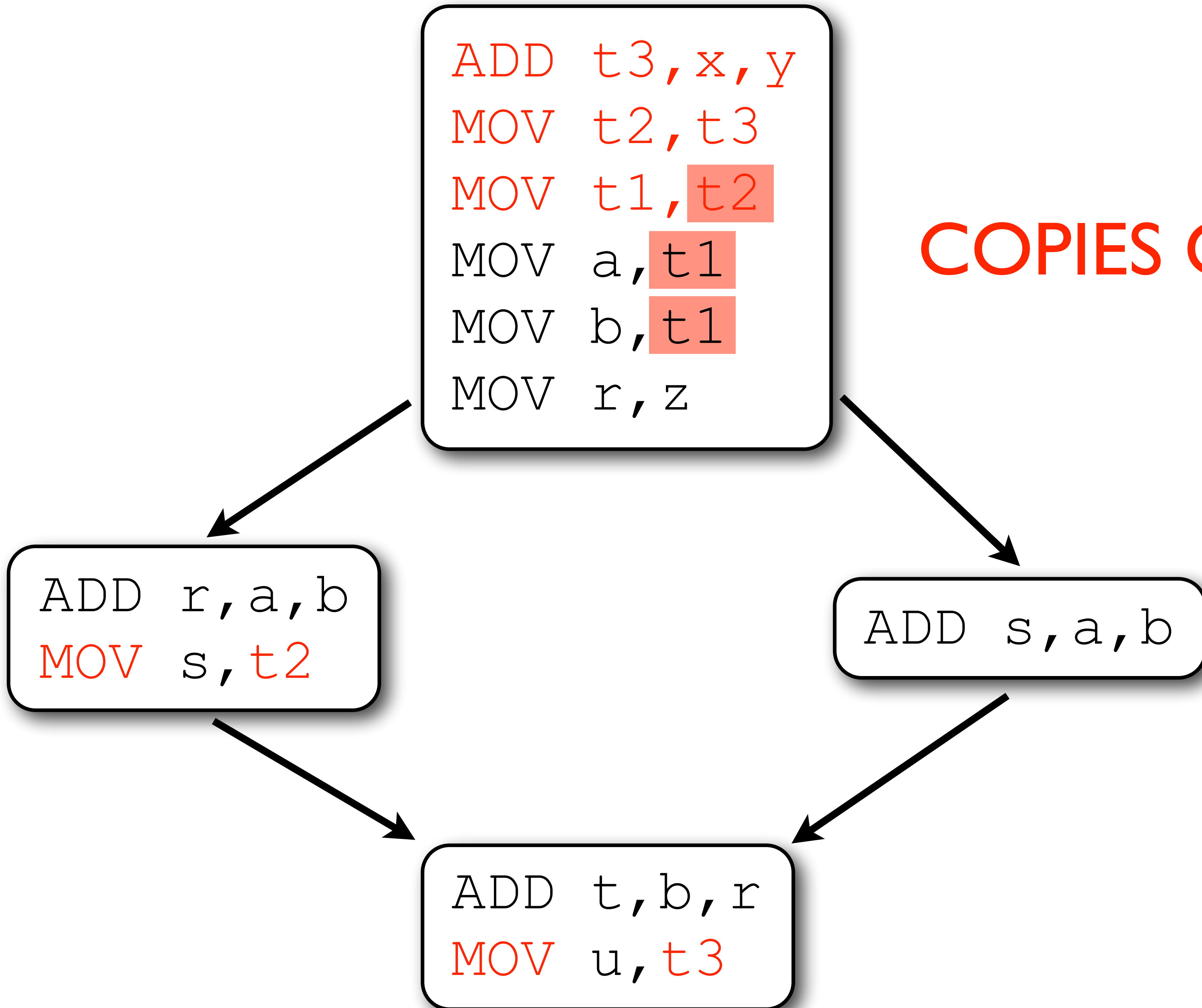
Putting it all together



Putting it all together

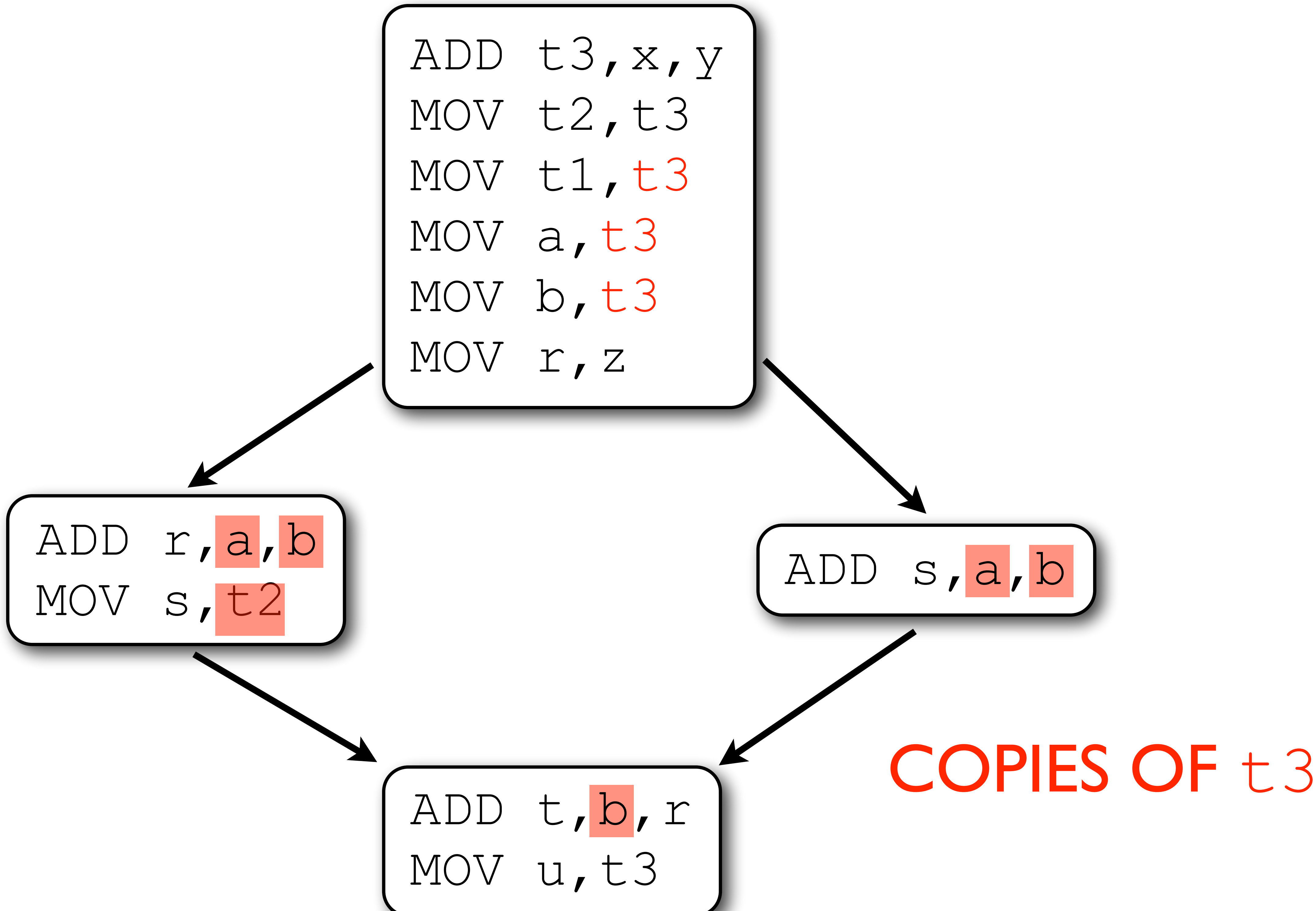


Putting it all together

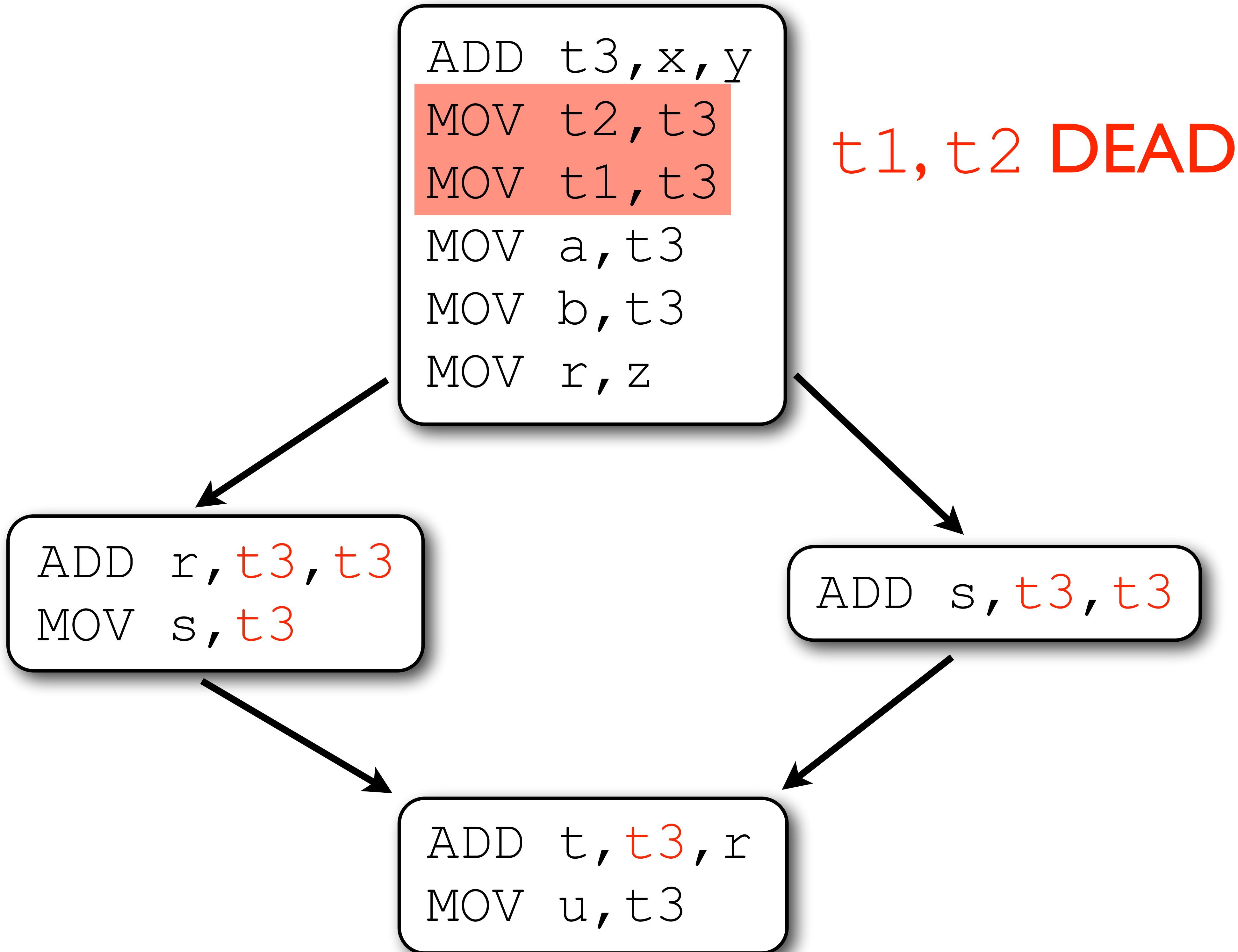


COPIES OF t3

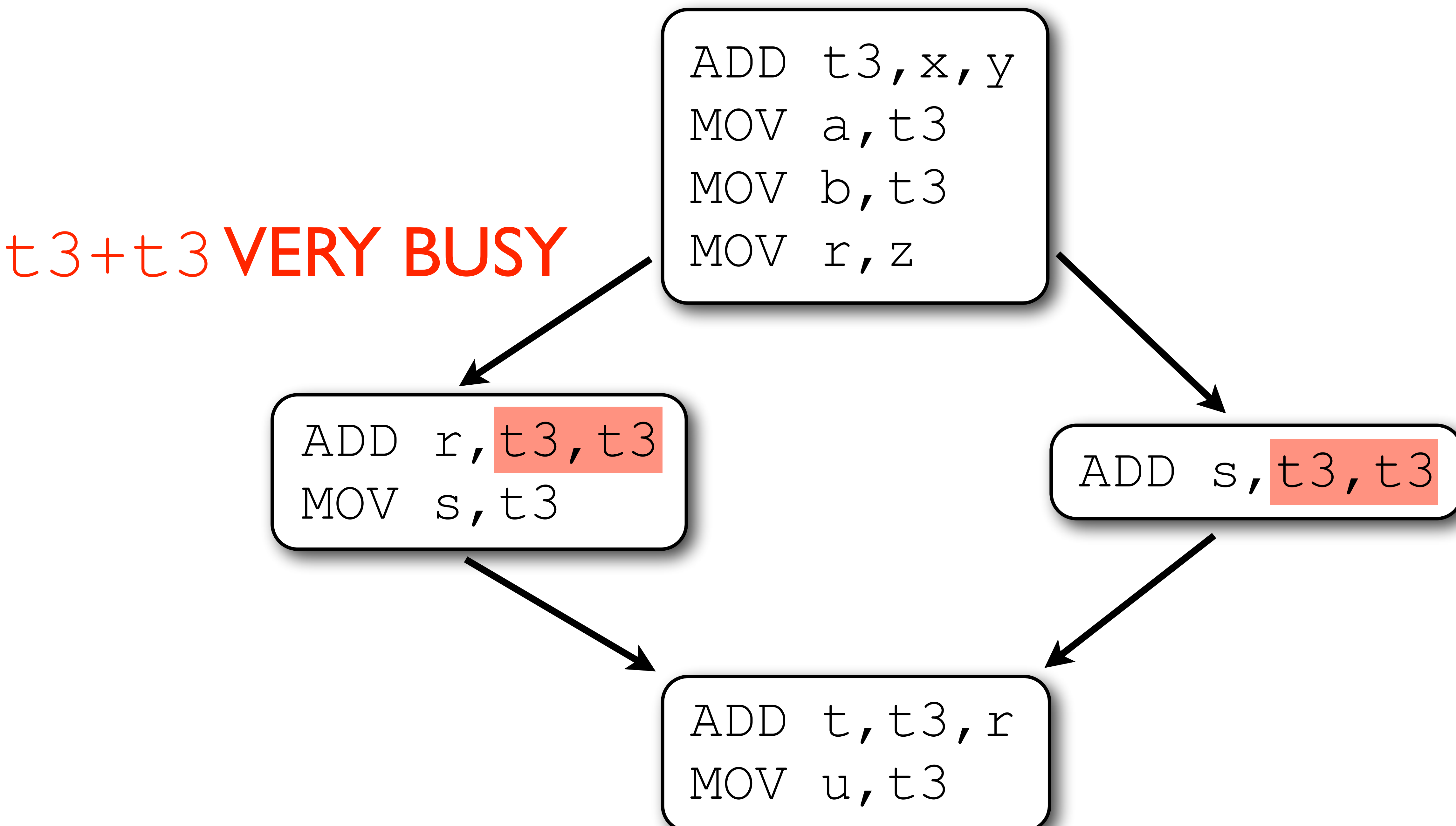
Putting it all together



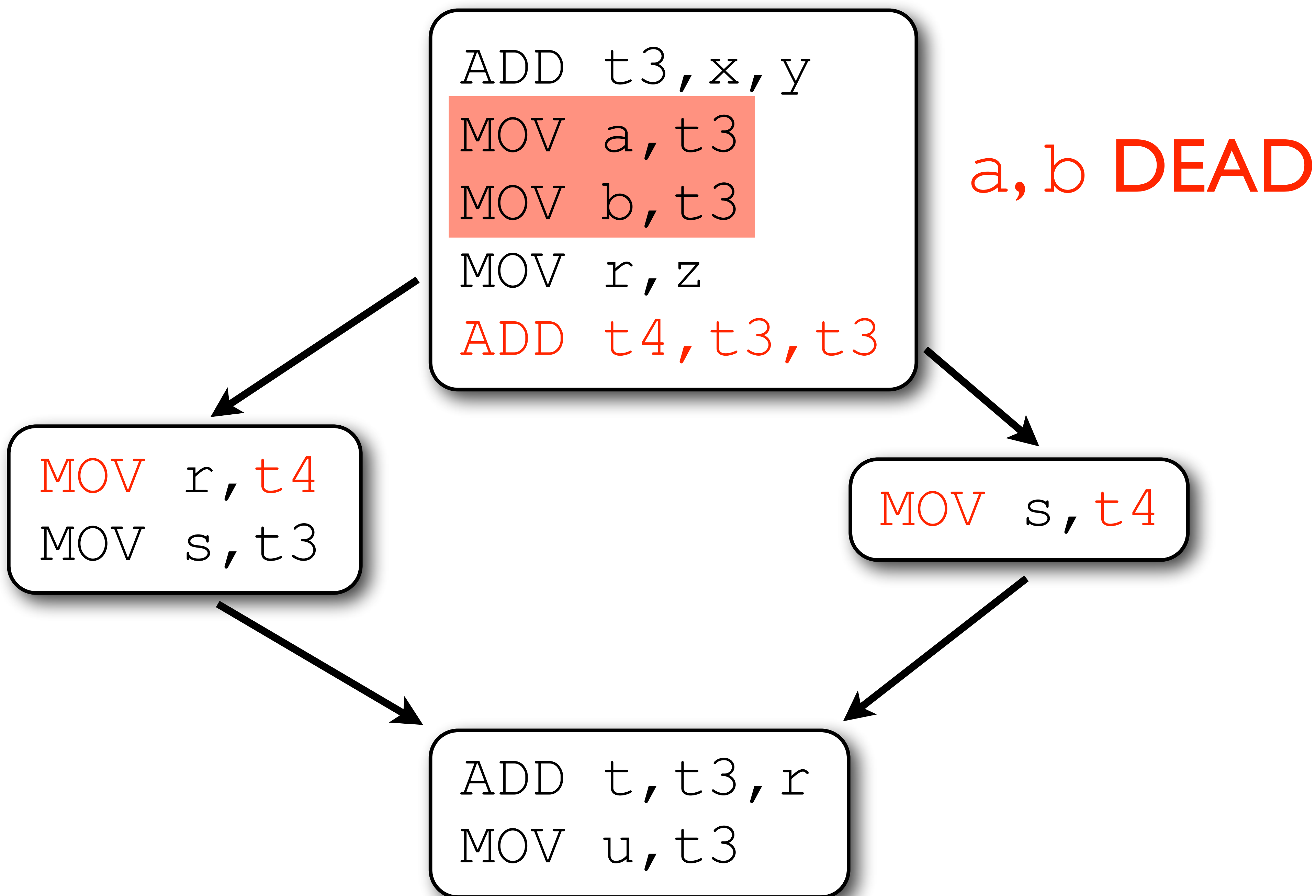
Putting it all together



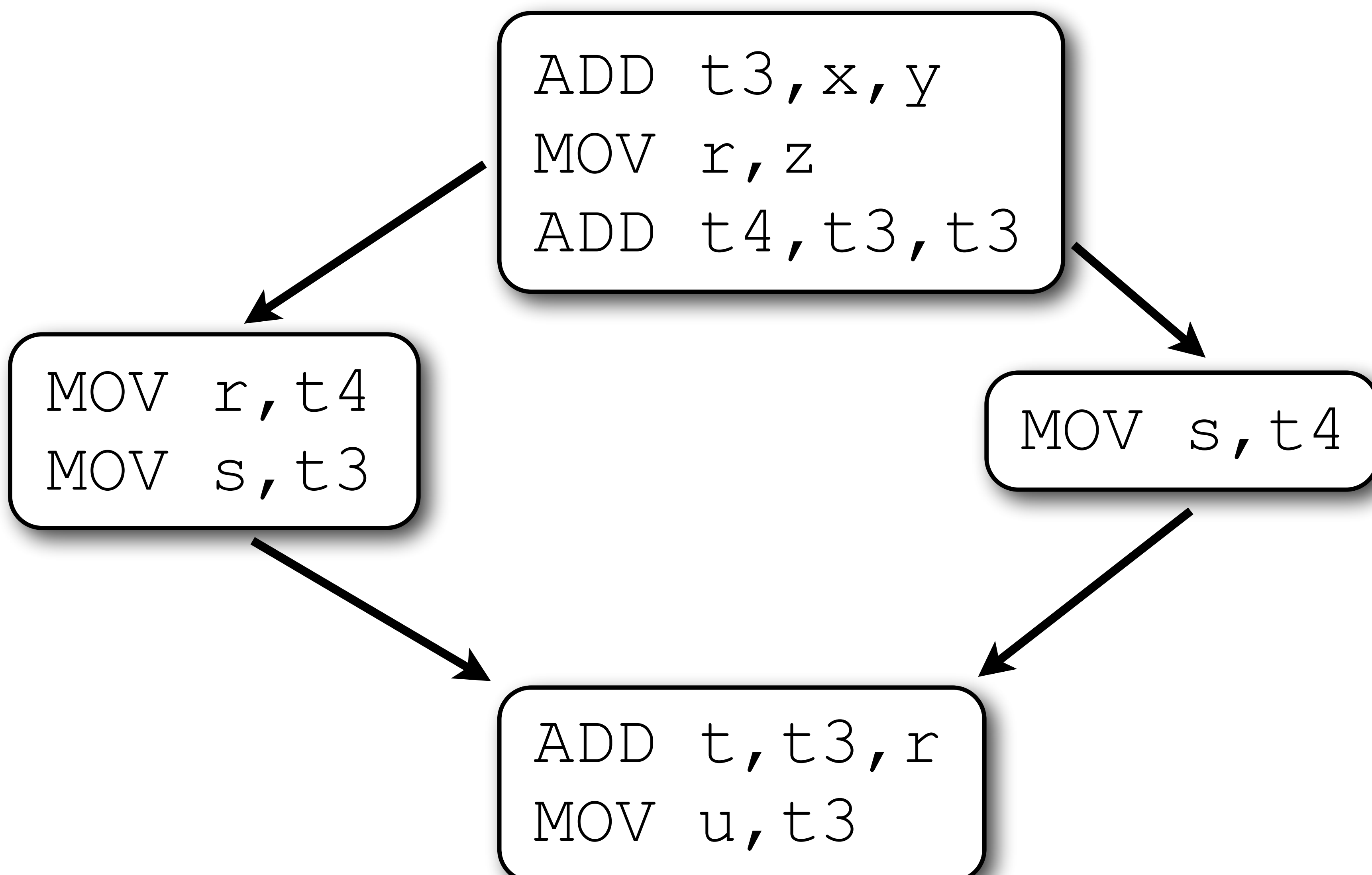
Putting it all together



Putting it all together



Putting it all together



Summary

- Some optimisations exist to reduce or remove redundancy in programs
- One such optimisation, common-subexpression elimination, is enabled by AVAIL
- Copy propagation makes CSE practical
- Other code motion optimisations can also help to reduce redundancy
- These optimisations work together to improve code

Lecture 8
Static single assignment and strength reduction

Motivation

Intermediate code in normal form permits maximum flexibility in allocating temporary variables to architectural registers.

This flexibility is not extended to user variables, and sometimes more registers than necessary will be used.

Register allocation can do a better job with user variables if we first translate code into *SSA form*.

Live ranges

User variables are often reassigned and reused many times over the course of a program, so that they become live in many different places.

Our intermediate code generation scheme assumes that each user variable is kept in a single virtual register throughout the entire program.

This results in each virtual register having a large *live range*, which is likely to cause clashes.

Live ranges

```
extern int f(int);
extern void h(int,int);
void g()
{
  int a,b,c;
  a = f(1); b = f(2); h(a,b);
  b = f(3); c = f(4); h(b,c);
  c = f(5); a = f(6); h(c,a);
}
```

Live ranges

```
a = f(1);
b = f(2);
h(a,b);
```

a, b CLASH

```
b = f(3);
c = f(4);
h(b,c);
```

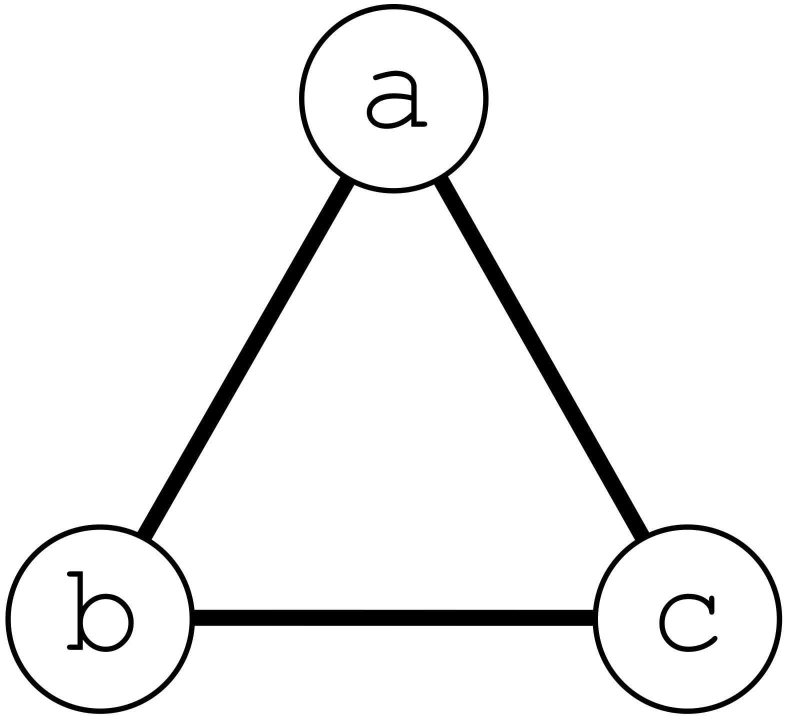
b, c CLASH

```
c = f(5);
a = f(6);
h(c,a);
```

c, a CLASH

Live ranges

```
a = f(1);
b = f(2);
h(a,b);

b = f(3);
c = f(4);
h(b,c);

c = f(5);
a = f(6);
h(c,a);
```



3 registers needed

Live ranges

We may remedy this situation by performing a transformation called *live range splitting*, in which live ranges are made smaller by using a different virtual register to store a variable's value at different times, thus reducing the potential for clashes.

Live ranges

```
extern int f(int);
extern void h(int,int);
void g()
{
  int a1,a2, b1,b2, c1,c2;
  a1 = f(1); b2 = f(2); h(a1,b2);
  b1 = f(3); c2 = f(4); h(b1,c2);
  c1 = f(5); a2 = f(6); h(c1,a2);
}
```

Live ranges

```
a1 = f(1);
b2 = f(2);
h(a1,b2);
```

a1,b2 CLASH

```
b1 = f(3);
c2 = f(4);
h(b1,c2);
```

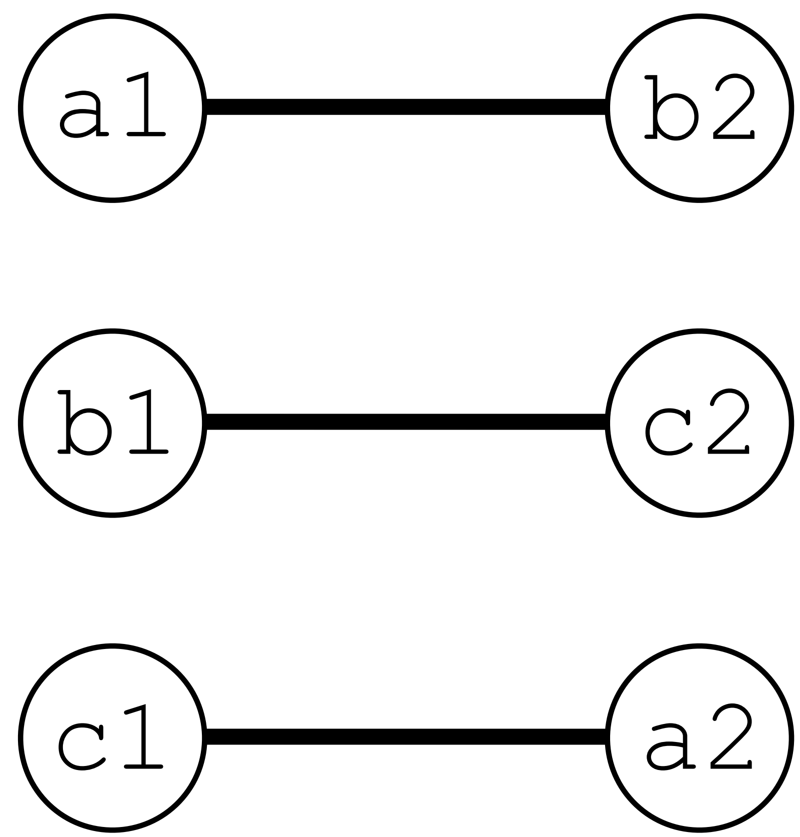
b1,c2 CLASH

```
c1 = f(5);
a2 = f(6);
h(c1,a2);
```

c1,a2 CLASH

Live ranges

```
a1 = f(1);
b2 = f(2);
h(a1,b2);

b1 = f(3);
c2 = f(4);
h(b1,c2);

c1 = f(5);
a2 = f(6);
h(c1,a2);
```



2 registers needed

Static single-assignment

Live range splitting is a useful transformation: it gives the same benefits for user variables as normal form gives for temporary variables.

However, if each virtual register is only ever assigned to once (statically), we needn't perform live range splitting, since the live ranges are already as small as possible.

Code in *static single-assignment* (SSA) form has this important property.

Static single-assignment

It is straightforward to transform straight-line code into SSA form: each variable is renamed by being given a subscript, which is incremented every time that variable is assigned to.

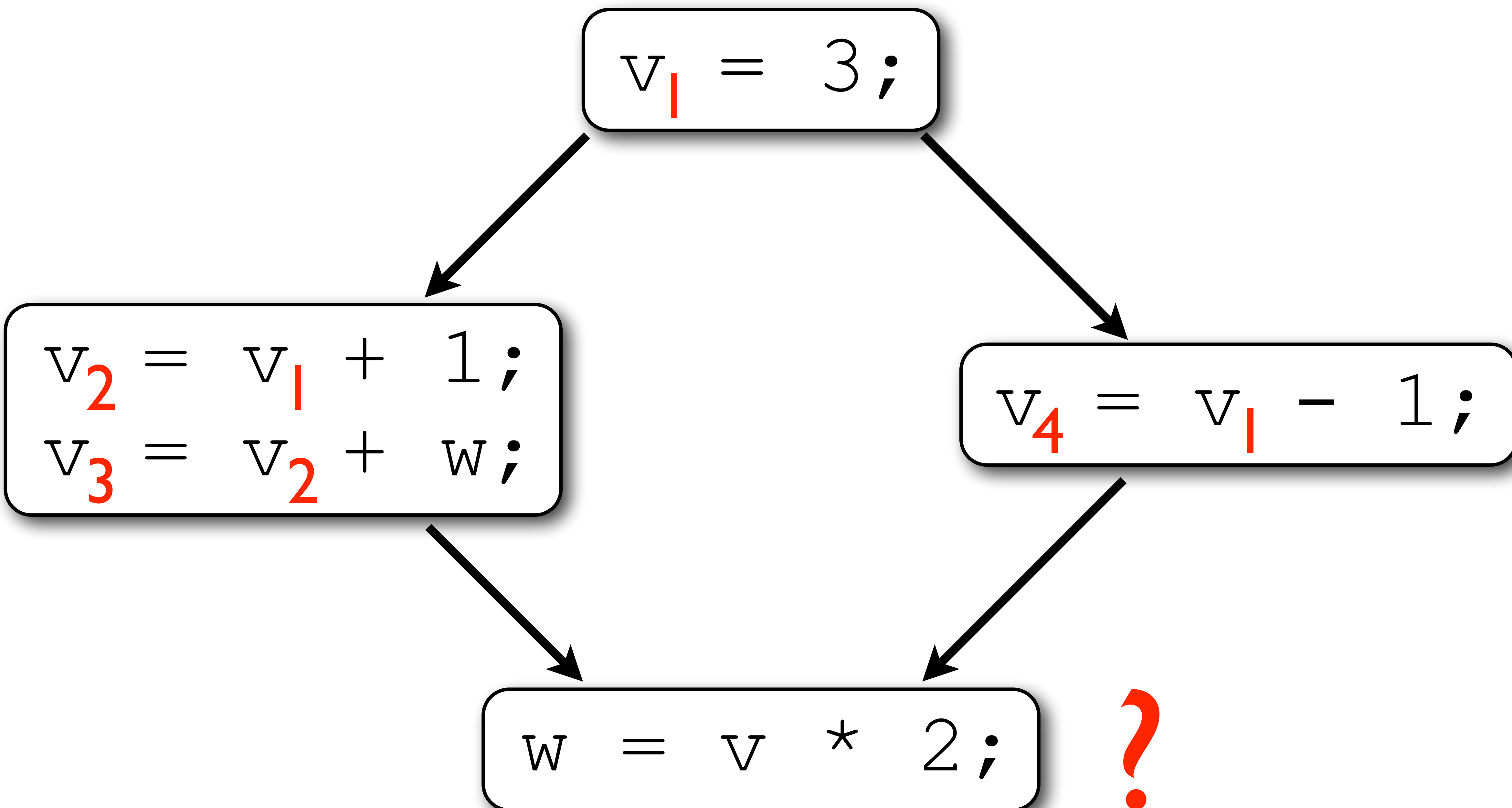
$$
\begin{aligned}
v_1 &= 3;\\
v_2 &= v_1 + 1;\\
v_3 &= v_2 + w_1;\\
w_2 &= v_3 + 2;
\end{aligned}
$$

Static single-assignment

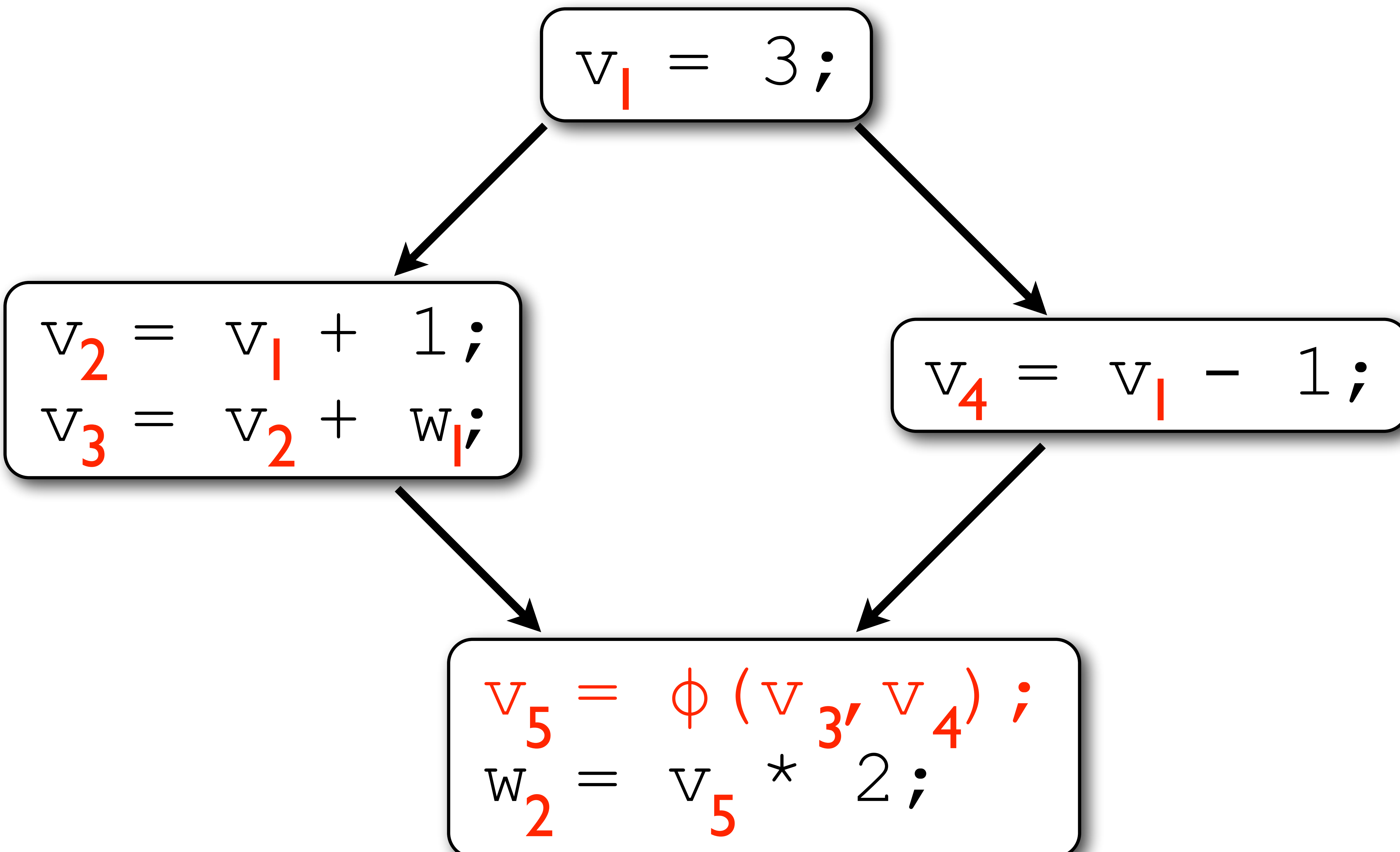
When the program's control flow is more complex, extra effort is required to retain the original data-flow behaviour.

Where control-flow edges meet, two (or more) differently-named variables must now be merged together.

Static single-assignment



Static single-assignment



Static single-assignment

The ϕ-functions in SSA keep track of which variables are merged at control-flow join points.

They are not executable since they do not record *which* variable to choose (cf. gated SSA form).

Static single-assignment

"Slight lie": SSA is useful for much more than register allocation!

In fact, the main advantage of SSA form is that, by representing data dependencies as precisely as possible, it makes many optimising transformations simpler and more effective, e.g. constant propagation, loop-invariant code motion, partial-redundancy elimination, and strength reduction.

Phase ordering

We now have many optimisations which we can perform on intermediate code.

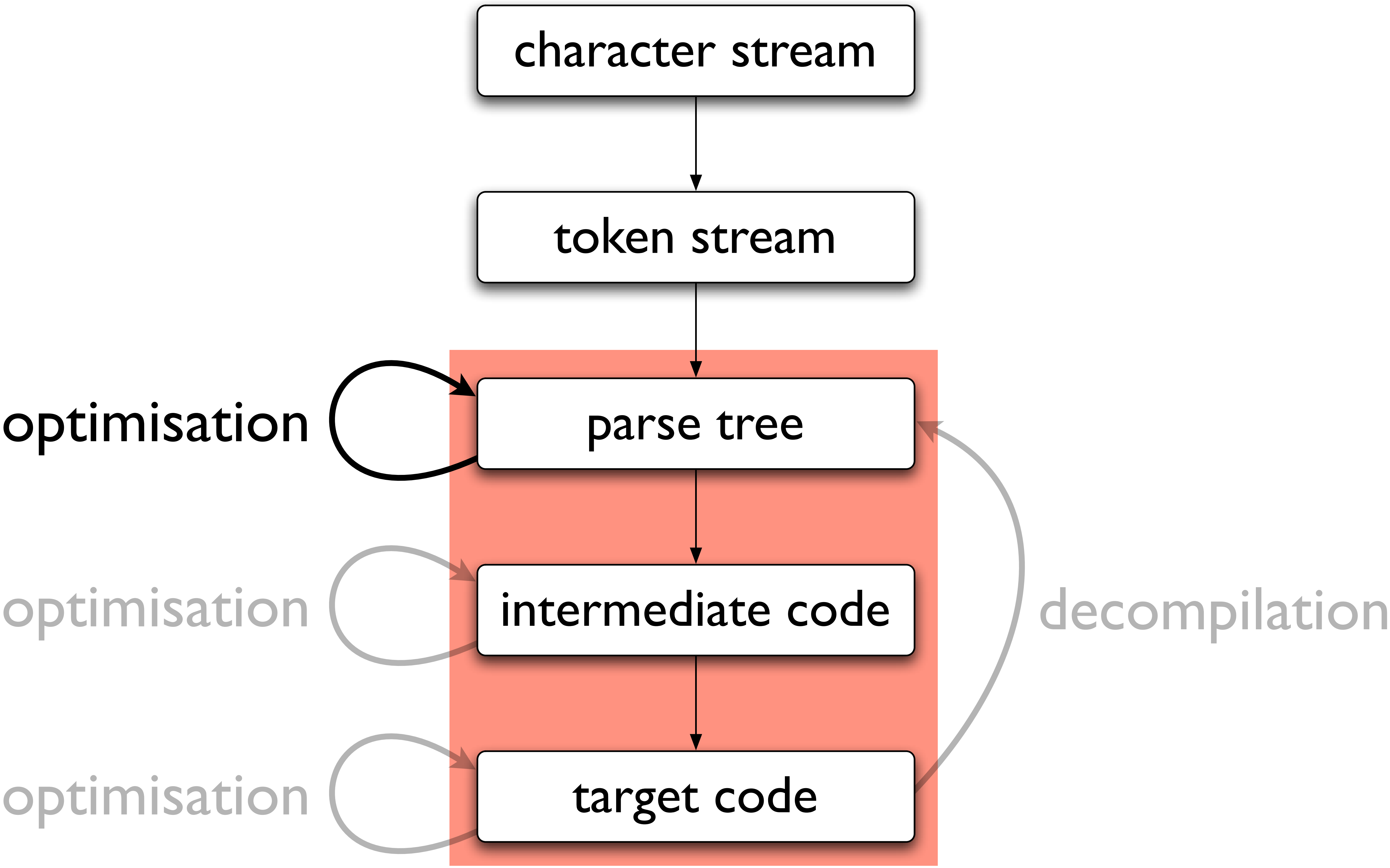
It is generally a difficult problem to decide in which *order* to perform these optimisations; different orders may be more appropriate for different programs.

Certain optimisations are antagonistic: for example, CSE may superficially improve a program at the expense of making the register allocation phase more difficult (resulting in spills to memory).

Part B

Higher-level optimisations

Higher-level optimisations



Higher-level optimisations

- More modern optimisations than those in Part A
  - Part A was mostly imperative
  - Part B is mostly functional
- Now operating on syntax of source language vs. an intermediate representation
- Functional languages make the presentation clearer, but many optimisations will also be applicable to imperative programs

Algebraic identities

The idea behind peephole optimisation of intermediate code can also be applied to abstract syntax trees.

There are many trivial examples where one piece of syntax is *always* (algebraically) equivalent to another piece of syntax which may be smaller or otherwise "better"; simple rewriting of syntax trees with these rules may yield a smaller or faster program.

Algebraic identities

... e + 0 ...

↓

... e ...

... (e + n) + m ...

↓

... e + (n + m) ...

Algebraic identities

These optimisations are boring, however, since they are always applicable to any syntax tree.

We're interested in more powerful transformations which may only be applied when some analysis has confirmed that they are safe.

Algebraic identities

In a lazy functional language,

`let x =` **e** `in if` **e′** `then` … `x` … `else` **e′′**

↓

`if` **e′** `then let x =` **e** `in` … `x` … `else` **e′′**

provided e′ and e′′ do not contain `x`.

This is still *quite* boring.

Strength reduction

More interesting analyses (i.e. ones that aren't purely syntactic) enable more interesting transformations.

Strength reduction is an optimisation which replaces expensive operations (e.g. multiplication and division) with less expensive ones (e.g. addition and subtraction).

It is most interesting and useful when done inside loops.

Strength reduction

For example, it may be advantageous to replace multiplication (`2*e`) with addition (`let x = e in x + x`) as before.

Multiplication may happen a lot inside loops (e.g. using the loop variable as an index into an array), so if we can spot a *recurring* multiplication and replace it with an addition we should get a faster program.

Strength reduction

```
int i;
for (i = 0; i < 100; i++)
{
  v[i] = 0;
}
```

Strength reduction

```
int i; char *p;
for (i = 0; i < 100; i++)
{
   p = (char *)v + 4*i;
   p[0] = 0; p[1] = 0;
   p[2] = 0; p[3] = 0;
}
```

Strength reduction

```
int i; char *p;
for (i = 0; i < 100; i++)
{
  p = (char *)v + 4*i;
  p[0] = 0; p[1] = 0;
  p[2] = 0; p[3] = 0;
}
```

Strength reduction

```
int i; char *p;
p = (char *)v;
for (i = 0; i < 100; i++)
{
  p[0] = 0; p[1] = 0;
  p[2] = 0; p[3] = 0;
  p += 4;
}
```

Strength reduction

```
int i; char *p;
p = (char *)v;
for (i = 0; p < (char *)v + 400; i++)
{
  p[0] = 0; p[1] = 0;
  p[2] = 0; p[3] = 0;
  p += 4;
}
```

Strength reduction

```
int i; int *p;
p = v;
for (i = 0; p < v + 100; i++)
{
  *p = 0;
  p++;
}
```

Strength reduction

```
int i; int *p;
p = v;
for (i = 0; p < v + 100; i++)
{
   *p = 0;
   p++;
}
```

Strength reduction

```
int *p;
for (p = v; p < v + 100; p++)
{
  *p = 0;
}
```

Multiplication has been replaced with addition.

Strength reduction

Note that, while this code is now almost optimal, it has obfuscated the intent of the original program.

Don't be tempted to *write* code like this!

For example, when targeting a 64-bit architecture, the compiler may be able to transform the original loop into fifty 64-bit stores, but will have trouble with our more efficient version.

Strength reduction

We are not restricted to replacing multiplication with addition, as long as we have

- induction variable: $\texttt{i} = \texttt{i} \oplus \texttt{c}$
- another variable: $\texttt{j} = \texttt{c}_2 \oplus (\texttt{c}_1 \otimes \texttt{i})$

for *some operations* $\oplus$ and $\otimes$ such that

$$\texttt{x} \otimes (\texttt{y} \oplus \texttt{z}) = (\texttt{x} \otimes \texttt{y}) \oplus (\texttt{x} \otimes \texttt{z})$$

Strength reduction

It might be easier to perform strength reduction on the intermediate code, but only if annotations have been placed on the flowchart to indicate loop structure.

At the syntax tree level, all loop structure is apparent.

Summary

- Live range splitting reduces register pressure
- In SSA form, each variable is assigned to only once
- SSA uses ϕ-functions to handle control-flow merges
- SSA aids register allocation and many optimisations
- Optimal ordering of compiler phases is difficult
- Algebraic identities enable code improvements
- Strength reduction uses them to improve loops

Lecture 9
Abstract interpretation

Motivation

We reason about programs statically, but we are really trying to make predictions about their dynamic behaviour.

Why not examine this behaviour directly?

It isn't generally feasible (e.g. termination, inputs) to run an *entire* computation at compile-time, but we can find things out about it by running a *simplified* version.

This is the basic idea of *abstract interpretation*.

Abstract interpretation

Warning: this will be a heavily simplified view of abstract interpretation; there is only time to give a brief introduction to the ideas, not explore them with depth or rigour.

Abstract interpretation

The key idea is to use an *abstraction*: a model of (otherwise unmanageable) reality, which

- discards enough detail that the model becomes manageable (e.g. small enough, computable enough), but
- retains enough detail to provide useful insight into the real world.

Abstract interpretation

For example, to plan a trip, you might use a map.

- A road map sacrifices a lot of detail —
  - trees, road conditions, individual buildings;
  - an entire dimension —
- but it retains most of the information which is important for planning a journey:
  - place names;
  - roads and how they are interconnected.

Abstract interpretation

Crucially, a road map is a useful abstraction because the route you plan is probably still valid back in reality.

- A cartographer creates an abstraction of reality (a map),
- you perform some computation on that abstraction (plan a route),
- and then you transfer the result of that computation back into the real world (drive to your destination).

Abstract interpretation

Trying to plan a journey by exploring the concrete world instead of the abstraction (i.e. driving around aimlessly) is either very expensive or virtually impossible.

A trustworthy map makes it possible — even easy.

This is a *real application* of abstract interpretation, but in this course we're more interested in computer programs.

Multiplying integers

A canonical example is the multiplication of integers.

If we want to know whether -1515×37 is positive or negative, there are two ways to find out:

- Compute in the concrete world (arithmetic), using the *standard interpretation* of multiplication. $-1515 \times 37 = -56055$, which is negative.
- Compute in an abstract world, using an *abstract interpretation* of multiplication: call it $\otimes$.

Multiplying integers

In this example the magnitudes of the numbers are insignificant; we care only about their sign, so we can use this information to design our abstraction.

$$(-) = \{ z \in \mathbb{Z} \mid z < 0 \}$$

$$(0) = \{ 0 \}$$

$$(+) = \{ z \in \mathbb{Z} \mid z > 0 \}$$

In the concrete world we have all the integers; in the abstract world we have only the values (−), (0) and (+).

Multiplying integers

We need to define the abstract operator ⊗.
Luckily, we have been to primary school.

$\otimes$	$(-)$	(0)	$(+)$
$(-)$	$(+)$	(0)	$(-)$
(0)	(0)	(0)	(0)
$(+)$	$(-)$	(0)	$(+)$

Multiplying integers

Armed with our abstraction, we can now tackle the original problem.

$$abs(-1515) = (-)$$

$$abs(37) = (+)$$

$$(-) \otimes (+) = (-)$$

So, without doing any *concrete* computation, we have discovered that -1515×37 has a negative result.

Multiplying integers

This is just a toy example, but it demonstrates the methodology: state a problem, devise an abstraction that retains the characteristics of that problem, solve the problem in the abstract world, and then interpret the solution back in the concrete world.

This abstraction has avoided doing arithmetic; in compilers, we will mostly be interested in avoiding expensive computation, nontermination or undecidability.

Safety

As always, there are important safety issues.

Because an abstraction discards detail, a computation in the abstract world will necessarily produce less precise results than its concrete counterpart.

It is important to ensure that this imprecision is *safe*.

Safety

We consider a particular abstraction to be safe if, whenever a property is true in the abstract world, it must also be true in the concrete world.

Our multiplication example is actually quite precise, and therefore trivially safe: the magnitudes of the original integers are irrelevant, so when the abstraction says that the result of a multiplication will be negative, it definitely will be.

In general, however, abstractions will be more approximate than this.

Adding integers

A good example is the *addition* of integers.
How do we define the abstract operator $\oplus$?

$\oplus$	$(-)$	(0)	$(+)$
$(-)$	$(-)$	$(-)$	$(?)$
(0)	$(-)$	(0)	$(+)$
$(+)$	$(?)$	$(+)$	$(+)$

Adding integers

When adding integers, their (relative) magnitudes *are* important in determining the sign of the result, but our abstraction has discarded this information.

As a result, we need a new abstract value: (?).

$$(-) = \{ z \in \mathbb{Z} \mid z < 0 \}$$

$$(0) = \{ 0 \}$$

$$(+) = \{ z \in \mathbb{Z} \mid z > 0 \}$$

$$(?) = \mathbb{Z}$$

Adding integers

(?) is less precise than (–), (0) and (+); it means "I don't know", or "it could be anything".

Because we want the abstraction to be safe, we must put up with this weakness.

Adding integers

$$abs(19 + 23) = abs(42) = (+)$$

$$abs(19) \oplus abs(23) = (+) \oplus (+) = (+)$$

$$\begin{tikzcd}
19, 23 \arrow[r, "+"] \arrow[d, "abs"'] & 42 \arrow[d, "abs"] \\
(+), (+) \arrow[r, "\oplus"'] & (+)
\end{tikzcd}$$

Adding integers

$$\mathit{abs}(-1515 + 37) = \mathit{abs}(-1478) = (-)$$

$$\mathit{abs}(-1515) \oplus \mathit{abs}(37) = (-) \oplus (+) = (?)$$

$$\begin{tikzcd}
-1515, 37 \arrow[r, "+"] \arrow[d, "\mathit{abs}"'] & -1478 \arrow[d, "\mathit{abs}"] \\
(-), (+) \arrow[r, "\oplus"'] & (?)
\end{tikzcd}$$

Safety

Here, safety is represented by the fact that (–) ⊆ (?):

$$\{ z \in \mathbb{Z} \mid z < 0 \} \subseteq \mathbb{Z}$$

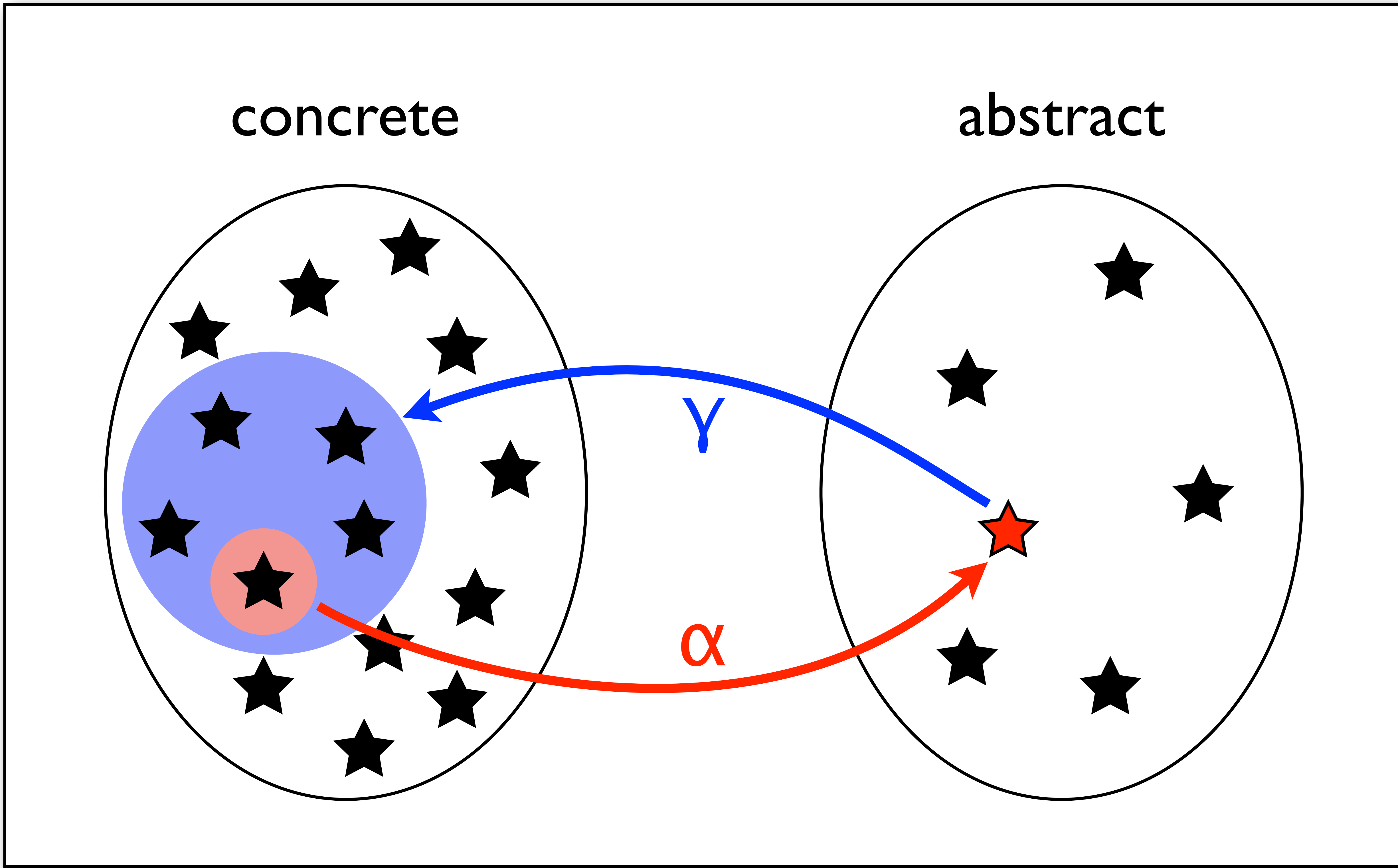
The result of doing the abstract computation is less precise, but crucially includes the result of doing the concrete computation (and then abstracting), so the abstraction is safe and hasn't missed anything.

Abstraction

Formally, an abstraction of some concrete domain *D* (e.g. $\wp(\mathbb{Z})$) consists of

- an abstract domain $D^{\#}$ (e.g. { (–), (0), (+), (?) }),
- an abstraction function $\alpha : D \rightarrow D^{\#}$ (e.g. *abs*), and
- a concretisation function $\gamma : D^{\#} \rightarrow D$, e.g.:
  - $(-) \mapsto \{ z \in \mathbb{Z} \mid z < 0 \}$,
  - $(0) \mapsto \{ 0 \}$, etc.

Abstraction



Abstraction

Given a function f from one concrete domain to another (e.g. $\hat{+} : \wp(\mathbb{Z}) \times \wp(\mathbb{Z}) \to \wp(\mathbb{Z})$), we require an abstract function $f^{\#}$ (e.g. $\oplus$) between the corresponding abstract domains.

$$\hat{+}(\{ 2, 5 \}, \{ -3, -7 \}) = \{ -1, -5, 2, -2 \}$$

$$\oplus((+), (-)) = (?)$$

Abstraction

So, for $D = \wp(\mathbb{Z})$ and $D^\# = \{\ (-), (0), (+), (?)\ \}$, we have:

$$\begin{tikzcd}
\mathbb{Z} \times \mathbb{Z} \arrow[r, "+"] \arrow[d] & \mathbb{Z} \arrow[d] \\
D \times D \arrow[r, "\hat{+}"] \arrow[d, shift right, "\alpha_1"'] & D \arrow[d, shift right, "\alpha_2"'] \\
D^\# \times D^\# \arrow[r, "\oplus"] \arrow[u, shift right, "\gamma_1"'] & D^\# \arrow[u, shift right, "\gamma_2"']
\end{tikzcd}$$

where $\alpha_{1,2}$ and $\gamma_{1,2}$ are the appropriate abstraction and concretisation functions.

Abstraction

These mathematical details are formally important, but are not examinable on this course.

Abstract interpretation can get very theoretical, but what's significant is the idea of using an abstraction to safely model reality.

Recognise that this is what we were doing in data-flow analysis: interpreting 3-address instructions as operations on *abstract values* — e.g. live variable sets — and then "executing" this abstract program.

Summary

- Abstractions are manageably simple models of unmanageably complex reality
- Abstract interpretation is a general technique for executing simplified versions of computations
- For example, the sign of an arithmetic result can be sometimes determined without doing any arithmetic
- Abstractions are approximate, but must be safe
- Data-flow analysis is a form of abstract interpretation

Lecture 10
Strictness analysis

Motivation

The operations and control structures of *imperative* languages are strongly influenced by the way most real computer hardware works.

This makes imperative languages relatively easy to compile, but (arguably) less expressive; many people use *functional* languages, but these are harder to compile into efficient imperative machine code.

Strictness optimisation can help to improve the efficiency of compiled functional code.

Call-by-value evaluation

Strict ("eager") functional languages (e.g. ML) use a *call-by-value* evaluation strategy:

$$\frac{e_2 \Downarrow v_2 \quad e_1[v_2/x] \Downarrow v_1}{(\lambda x.e_1)\; e_2 \Downarrow v_1}$$

- Efficient in space and time, but
- might evaluate more arguments than necessary.

Call-by-name evaluation

Non-strict ("lazy") functional languages (e.g. Haskell) use a *call-by-name* evaluation strategy:

$$\frac{e_1[e_2/x] \Downarrow v}{(\lambda x.e_1)\ e_2 \Downarrow v}$$

- Only evaluates arguments when necessary, but
- copies (and redundantly re-evaluates) arguments.

Call-by-need evaluation

One simple optimisation is to use *call-by-need* evaluation instead of call-by-name.

If the language has no side-effects, duplicated instances of an argument can be shared, evaluated once if required, and the resulting value reused.

This avoids recomputation and is better than call-by-name, but is still more expensive than call-by-value.

Call-by-need evaluation

```
plus(x,y) = if x=0 then y else plus(x-1,y+1)
```

Using call-by-value:

```
plus(3,4) ↦ if 3=0 then 4 else plus(3-1,4+1)
          ↦ plus(2,5)
          ↦ plus(1,6)
          ↦ plus(0,7)
          ↦ 7
```

Call-by-need evaluation

```
plus(x,y) = if x=0 then y else plus(x-1,y+1)
```

Using call-by-need:

```
plus(3,4) ↦ if 3=0 then 4 else plus(3-1,4+1)
          ↦ plus(3-1,4+1)
          ↦ plus(2-1,4+1+1)
          ↦ plus(1-1,4+1+1+1)
          ↦ 4+1+1+1
          ↦ 5+1+1
          ↦ 6+1
          ↦ 7
```

Replacing CBN with CBV

So why not just replace call-by-name with call-by-value?

Because, while replacing call-by-name with call-by-need never changes the semantics of the original program (in the absence of side-effects), replacing CBN with CBV does.

In particular, the program's *termination* behaviour changes.

Replacing CBN with CBV

Assume we have some nonterminating expression, Ω.

- Using CBN, the expression $(\lambda x.\, 42)\ \Omega$ will evaluate to 42.
- But using CBV, evaluation of $(\lambda x.\, 42)\ \Omega$ will not terminate: Ω gets evaluated first, even though its value is not needed here.

We should therefore try to use call-by-value wherever possible, but not when it will affect the termination behaviour of a program.

Neededness

Intuitively, it will be safe to use CBV in place of CBN whenever an argument is definitely going to be evaluated.

We say that an argument is *needed* by a function if the function will always evaluate it.

- $\lambda x,y.\ x+y$ needs both its arguments.
- $\lambda x,y.\ x+1$ needs only its first argument.
- $\lambda x,y.\ 42$ needs neither of its arguments.

Neededness

These needed arguments can safely be passed by value: if their evaluation causes nontermination, this will just happen sooner rather than later.

Neededness

In fact, neededness is too conservative:

$$\lambda x,y,z.\ \text{if } x \text{ then } y \text{ else } \Omega$$

This function might not evaluate y, so only x is *needed*.

But actually it's still safe to pass y by value: if y doesn't get evaluated then the function doesn't terminate anyway, so it doesn't matter if eagerly evaluating y causes nontermination.

Strictness

What we really want is a more refined notion:

It is safe to pass an argument by value when *the function fails to terminate whenever the argument fails to terminate*.

When this more general statement holds, we say the function is *strict* in that argument.

$$\lambda x,y,z.\ \text{if } x \text{ then } y \text{ else } \Omega$$

is strict in x and strict in y.

Strictness

If we can develop an analysis that discovers which functions are strict in which arguments, we can use that information to selectively replace CBN with CBV and obtain a more efficient program.

Strictness analysis

We can perform strictness analysis by abstract interpretation.

First, we must define a concrete world of programs and values.

We will use the simple language of *recursion equations*, and only consider integer values.

Recursion equations

$$\begin{aligned} F_1(x_1, \ldots, x_{k_1}) &= e_1 \\ \cdots &= \cdots \\ F_n(x_1, \ldots, x_{k_n}) &= e_n \end{aligned}$$

$$e ::= x_i \mid A_i(e_1, \ldots, e_{r_i}) \mid F_i(e_1, \ldots e_{k_i})$$

where each A_i is a symbol representing a built-in (predefined) function of arity r_i.

Recursion equations

For our earlier example,

```
plus(x,y) = if x=0 then y else plus(x-1,y+1)
```

we can write the recursion equation

$$plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$$

where *cond*, *eq*, 0, *sub1* and *add1* are built-in functions.

Standard interpretation

We must have some representation of nontermination in our concrete domain.

As values we will consider the integer results of terminating computations, $\mathbb{Z}$, and a single extra value to represent nonterminating computations: $\perp$.

Our concrete domain *D* is therefore $\mathbb{Z}_\perp = \mathbb{Z} \cup \{ \perp \}$.

Standard interpretation

Each built-in function needs a standard interpretation.

We will interpret each A_i as a function a_i on values in D:

$$\begin{aligned}
cond(\perp, x, y) &= \perp \\
cond(0, x, y) &= y \\
cond(p, x, y) &= x \quad \text{otherwise}
\end{aligned}$$

$$\begin{aligned}
eq(\perp, y) &= \perp \\
eq(x, \perp) &= \perp \\
eq(x, y) &= x =_{\mathbb{Z}} y \quad \text{otherwise}
\end{aligned}$$

Standard interpretation

The standard interpretation f_i of a user-defined function F_i is constructed from the built-in functions by composition and recursion according to its defining equation.

$$plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$$

Abstract interpretation

Our abstraction must capture the properties we're interested in, while discarding enough detail to make analysis computationally feasible.

Strictness is all about termination behaviour, and in fact this is all we care about: does evaluation of an expression *definitely not* terminate (as with Ω), or *may* it eventually terminate and return a result?

Our abstract domain $D^{\#}$ is therefore $\{ 0, 1 \}$.

Abstract interpretation

For each built-in function A_i we need a corresponding strictness function $a_i^{\#}$ — this provides the *strictness interpretation* for A_i.

Whereas the standard interpretation of each built-in is a function on concrete values from D, the strictness interpretation of each will be a function on abstract values from $D^{\#}$ (i.e. 0 and 1).

Abstract interpretation

A formal relationship exists between the standard and abstract interpretations of each built-in function; the mathematical details are in the lecture notes.

Informally, we use the same technique as for multiplication and addition of integers in the last lecture: we define the abstract operations using what we know about the behaviour of the concrete operations.

Abstract interpretation

x	y	$eq^{\#}(x,y)$
0	0	0
0	1	0
1	0	0
1	1	1

Abstract interpretation

p	x	y	$cond^{\#}(p,x,y)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Abstract interpretation

These functions may be expressed more compactly as boolean expressions, treating 0 and 1 from $D^\#$ as *false* and *true* respectively.

We can use Karnaugh maps (from IA DigElec) to turn each truth table into a simple boolean expression.

Abstract interpretation

x, y

p

$cond^{\#}$	0, 0	0, 1	1, 1	1, 0
0	0	0	0	0
1	0	1	1	1

$p \wedge y$ $\quad$ $p \wedge x$

$$cond^{\#}(p, x, y) = (p \wedge y) \vee (p \wedge x)$$

Abstract interpretation

$$\begin{array}{c|c|c|c|c}
 & & \multicolumn{2}{c}{y} & \\
 & eq^{\#} & 0 & 1 & \\
\hline
x & 0 & 0 & 0 & \\
\hline
 & 1 & 0 & \boxed{1} & x \wedge y
\end{array}$$

$$eq^{\#}(x, y) = x \wedge y$$

Strictness analysis

So far, we have set up

- a concrete domain, D, equipped with
  - a standard interpretation a_i of each built-in A_i, and
  - a standard interpretation f_i of each user-defined F_i;
- and an abstract domain, $D^\#$, equipped with
  - an abstract interpretation $a_i^\#$ of each built-in A_i.

Strictness analysis

The point of this analysis is to discover the missing piece: what is the strictness function $f_i^{\#}$ corresponding to each user-defined F_i?

These strictness functions will show us exactly how each F_i is strict with respect to each of its arguments — and that's the information that tells us where we can replace lazy, CBN-style parameter passing with eager CBV.

Strictness analysis

But recall that the recursion equations show us how to build up each user-defined function, by composition and recursion, from all the built-in functions:

$$plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$$

So we can build up the $f_i^{\#}$ from the $a_i^{\#}$ in the same way:

$$plus^{\sharp}(x, y) = cond^{\sharp}(eq^{\sharp}(x, 0^{\sharp}), y, plus^{\sharp}(sub1^{\sharp}(x), add1^{\sharp}(y)))$$

Strictness analysis

$$plus^\sharp(x,y) = cond^\sharp(eq^\sharp(x,0^\sharp), y, plus^\sharp(sub1^\sharp(x), add1^\sharp(y)))$$

We already know all the other strictness functions:

$$\begin{aligned}
cond^\sharp(p,x,y) &= p \wedge (x \vee y) \\
eq^\sharp(x,y) &= x \wedge y \\
0^\sharp &= 1 \\
sub1^\sharp(x) &= x \\
add1^\sharp(x) &= x
\end{aligned}$$

So we can use these to simplify the expression for *plus*#.

Strictness analysis

$$
\begin{aligned}
\mathit{plus}^\sharp(x,y) &= \mathit{cond}^\sharp(\mathit{eq}^\sharp(x,0^\sharp), y, \mathit{plus}^\sharp(\mathit{sub1}^\sharp(x), \mathit{add1}^\sharp(y))) \\
&= \mathit{eq}^\sharp(x,0^\sharp) \wedge (y \vee \mathit{plus}^\sharp(\mathit{sub1}^\sharp(x), \mathit{add1}^\sharp(y))) \\
&= \mathit{eq}^\sharp(x,1) \wedge (y \vee \mathit{plus}^\sharp(x,y)) \\
&= x \wedge 1 \wedge (y \vee \mathit{plus}^\sharp(x,y)) \\
&= x \wedge (y \vee \mathit{plus}^\sharp(x,y))
\end{aligned}
$$

Strictness analysis

$$plus^\sharp(x,y) = x \wedge (y \vee plus^\sharp(x,y))$$

This is a recursive definition, and so unfortunately doesn't provide us with the strictness function directly.

We want a definition of *plus*$^\#$ which satisfies this equation — actually we want the *least fixed point* of this equation, which (as ever!) we can compute iteratively.

Algorithm

```
for i = 1 to n do f#[i] := λx⃗.0
while (f#[] changes) do
   for i = 1 to n do
      f#[i] := λx⃗.e_i^#
```

$e_i^\#$ means "e_i (from the recursion equations) with each A_j replaced with $a_j^\#$ and each F_j replaced with `f#[j]`".

Algorithm

We have only one user-defined function, *plus*, and so only one recursion equation:

$$plus(x, y) = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$$

We initialise the corresponding element of our `f#[ ]` array to contain the always-0 strictness function of the appropriate arity:

`f#[1] :=` $\lambda x, y.\ 0$

Algorithm

On the first iteration, we calculate $e_1^\#$:

- The recursion equations say
  $e_1 = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$
- The current contents of `f#[ ]` say $f_1^\#$ is $\lambda x,y.\ 0$
- So:

$$e_1^\# = cond^\#(eq^\#(x, 0^\#), y, (\lambda x,y.\ 0)\ (sub1^\#(x), add1^\#(y)))$$

Algorithm

$$e_1^\# = cond^\#(eq^\#(x, 0^\#), y, (\lambda x,y.\ 0)\ (sub1^\#(x), add1^\#(y)))$$

Simplifying:

$$e_1^\# = cond^\#(eq^\#(x, 0^\#), y, 0)$$

Using definitions of *cond*#, *eq*# and 0#:

$$e_1^\# = (x \wedge 1) \wedge (y \vee 0)$$

Simplifying again:

$$e_1^\# = x \wedge y$$

Algorithm

So, at the end of the first iteration,

`f#[1]` := $\lambda x,y.\ x \wedge y$

Algorithm

On the second iteration, we recalculate $e_1^\#$:

- The recursion equations still say
  $e_1 = cond(eq(x, 0), y, plus(sub1(x), add1(y)))$
- The current contents of `f#[ ]` say $f_1^\#$ is $\lambda x,y.\ x \wedge y$
- So:

$$e_1^\# = cond^\#(eq^\#(x, 0^\#), y, (\lambda x,y.\ x \wedge y)\ (sub1^\#(x), add1^\#(y)))$$

Algorithm

$$e_1^{\#} = cond^{\#}(eq^{\#}(x, 0^{\#}), y, (\lambda x,y.\, x \wedge y)\ (sub1^{\#}(x), add1^{\#}(y)))$$

Simplifying:

$$e_1^{\#} = cond^{\#}(eq^{\#}(x, 0^{\#}), y, sub1^{\#}(x) \wedge add1^{\#}(y))$$

Using definitions of $cond^{\#}$, $eq^{\#}$, $0^{\#}$, $sub1^{\#}$ and $add1^{\#}$:

$$e_1^{\#} = (x \wedge 1) \wedge (y \vee (x \wedge y))$$

Simplifying again:

$$e_1^{\#} = x \wedge y$$

Algorithm

So, at the end of the second iteration,

$$\texttt{f\#[1]} \;\; := \;\; \lambda x,y.\ x \wedge y$$

This is the same result as last time, so we stop.

Algorithm

$$plus^{\#}(x, y) = x \wedge y$$

Optimisation

So now, finally, we can see that

$$plus^{\#}(1, 0) = 1 \wedge 0 = 0$$

and

$$plus^{\#}(0, 1) = 0 \wedge 1 = 0$$

which means our concrete *plus* function is strict in its first argument and strict in its second argument: we may always safely use CBV when passing either.

Summary

- Functional languages can use CBV or CBN evaluation
- CBV is more efficient but can only be used in place of CBN if termination behaviour is unaffected
- Strictness shows dependencies of termination
- Abstract interpretation may be used to perform strictness analysis of user-defined functions
- The resulting strictness functions tell us when it is safe to use CBV in place of CBN

Lecture 11
Constraint-based analysis

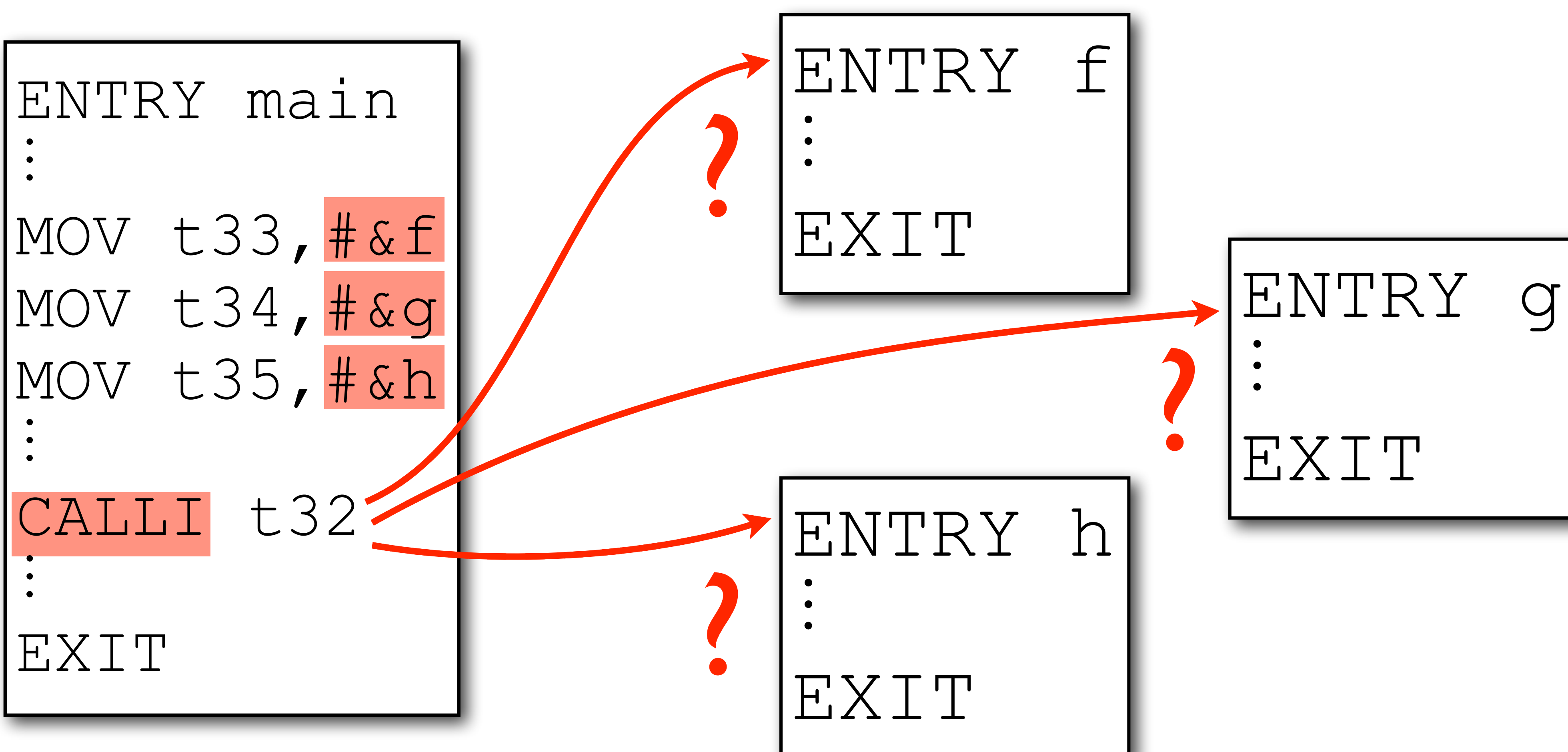
Motivation

Intra-procedural analysis depends upon accurate control-flow information.

In the presence of certain language features (e.g. indirect calls) it is nontrivial to predict accurately how control may flow at execution time — the naïve strategy is very imprecise.

A *constraint-based* analysis called 0CFA can compute a more precise estimate of this information.

Imprecise control flow



Constraint-based analysis

Many of the analyses in this course can be thought of in terms of *solving systems of constraints*.

For example, in LVA, we generate *equality constraints* from each instruction in the program:

$$\begin{aligned}
\textit{in-live}(m) &= (\textit{out-live}(m) \setminus \textit{def}(m)) \cup \textit{ref}(m) \\
\textit{out-live}(m) &= \textit{in-live}(n) \cup \textit{in-live}(o) \\
\textit{in-live}(n) &= (\textit{out-live}(n) \setminus \textit{def}(n)) \cup \textit{ref}(n) \\
&\vdots
\end{aligned}$$

and then iteratively compute their minimal solution.

0CFA

0CFA — "zeroth-order control-flow analysis" — is a constraint-based analysis for discovering which values may reach different places in a program.

When functions (or pointers to functions) are present, this provides information about which functions may potentially be called at each call site.

We can then build a more precise call graph.

Specimen language

Functional languages are a good candidate for this kind of analysis; they have functions as first-class values, so control flow may be complex.

We will use a minimal syntax for expressions:

$$e ::= x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \mathrm{let}\ x = e_1\ \mathrm{in}\ e_2$$

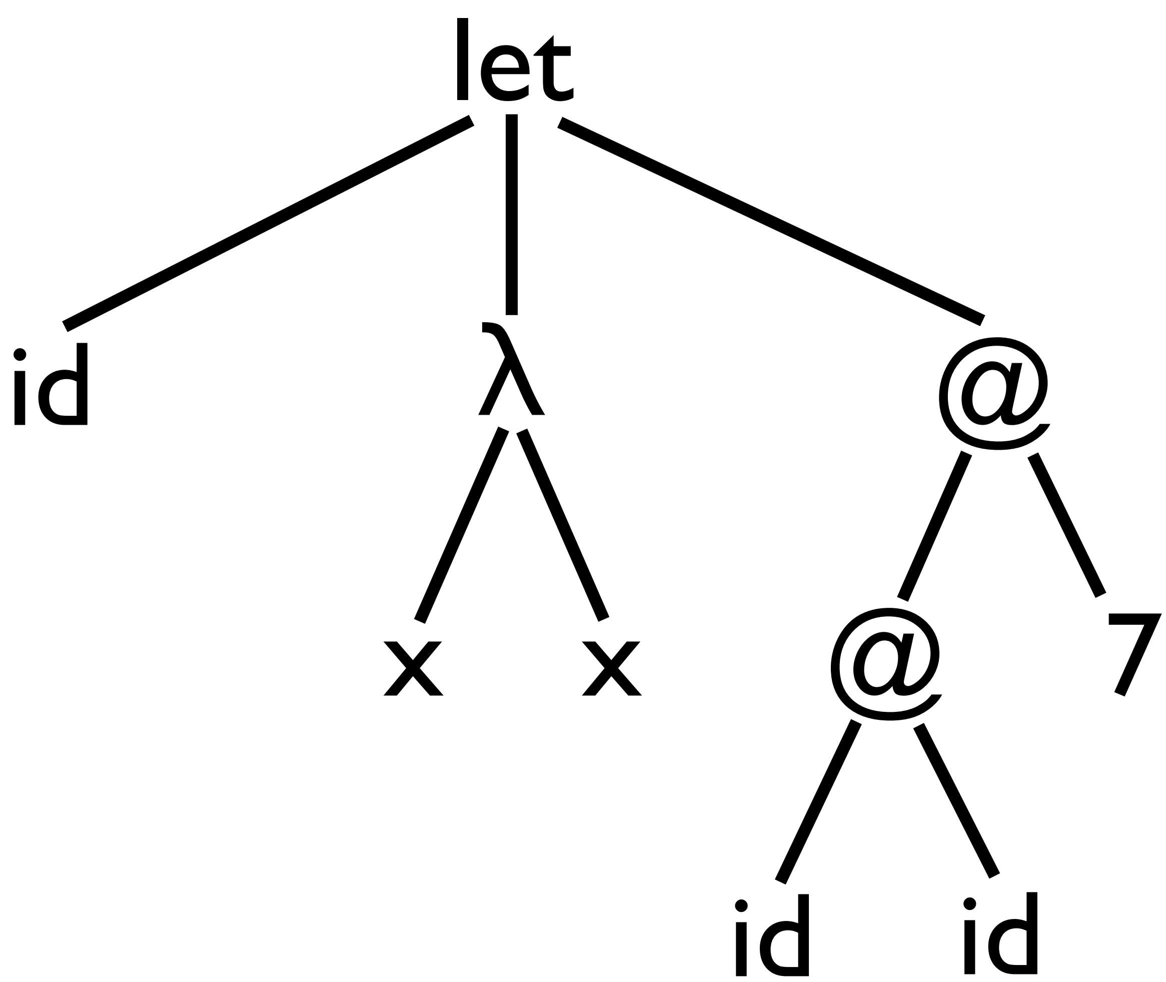
A program in this language is a *closed* expression.

Specimen program

let id = λx. x in id id 7

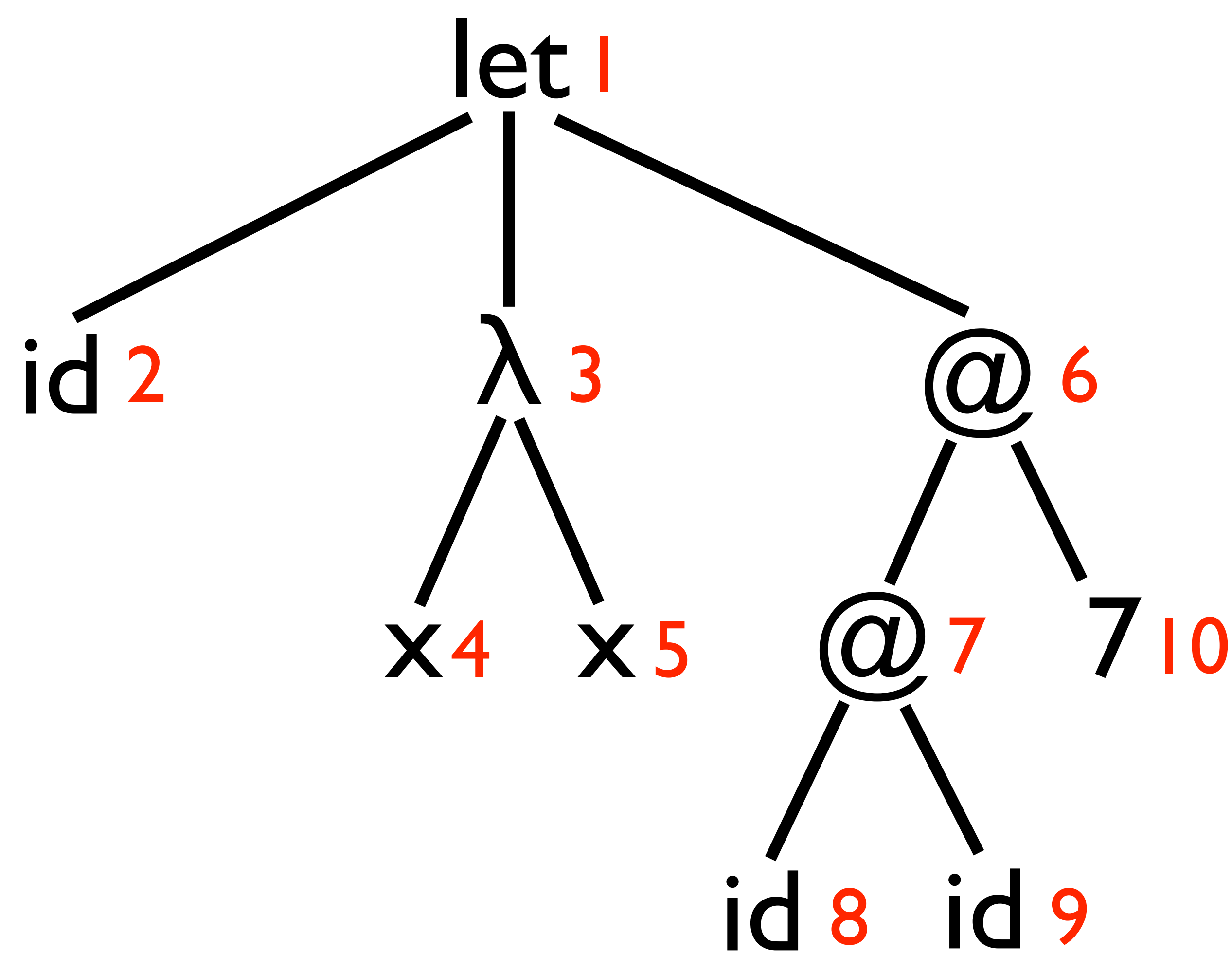
Program points

let id = λx. x in id id 7



Program points

$(\text{let id}^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((\text{id}^{8}\ \text{id}^{9})^{7}\ 7^{10})^{6})^{1}$



Program points

$$(\mathrm{let}\ \mathrm{id}^{2} = (\lambda x^{4}.\, x^{5})^{3}\ \mathrm{in}\ ((\mathrm{id}^{8}\ \mathrm{id}^{9})^{7}\ 7^{10})^{6})^{1}$$

Each program point i has an associated *flow variable* α_i.

Each α_i represents the set of *flow values* which may be yielded at program point i during execution.

For this language the flow values are integers and function closures; in this particular program, the only values available are 7^{10} and $(\lambda x^{4}.\, x^{5})^{3}$.

Program points

$$(\text{let id}^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((\text{id}^{8}\ \text{id}^{9})^{7}\ 7^{10})^{6})^{1}$$

The precise value of each α_i is undecidable in general, so our analysis will compute a safe overapproximation.

From the structure of the program we can generate a set of constraints on the flow variables, which we can then treat as data-flow inequations and iteratively compute their least solution.

Generating constraints

$(\text{let id}^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((\text{id}^{8}\ \text{id}^{9})^{7}\ 7^{10})^{6})^{1}$

Generating constraints

$$(\text{let } id^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((id^{8}\ id^{9})^{7}\ 7^{10})^{6})^{1}$$

$$c^{a} \qquad \alpha_{a} \supseteq \{\ c^{a}\ \}$$

Generating constraints

$(\text{let id}^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((\text{id}^{8}\ \text{id}^{9})^{7}\ 7^{10})^{6})^{1}$

$$7^{10} \longrightarrow \alpha_{10} \supseteq \{\ 7^{10}\ \}$$

$\alpha_{10} \supseteq \{\ 7^{10}\ \}$

Generating constraints

$(\text{let id}^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((\text{id}^{8}\ \text{id}^{9})^{7}\ 7^{10})^{6})^{1}$

$$(\lambda x^{a}.\, e^{b})^{c} \qquad \alpha_{c} \supseteq \{\, (\lambda x^{a}.\, e^{b})^{c} \,\}$$

$\alpha_{10} \supseteq \{\, 7^{10} \,\}$

Generating constraints

$(\text{let } id^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((id^{8}\ id^{9})^{7}\ 7^{10})^{6})^{1}$

$$(\lambda x^{4}.\, x^{5})^{3} \longrightarrow \alpha_{3} \supseteq \{\, (\lambda x^{4}.\, x^{5})^{3} \,\}$$

$\alpha_{10} \supseteq \{\, 7^{10} \,\}$

$\alpha_{3} \supseteq \{\, (\lambda x^{4}.\, x^{5})^{3} \,\}$

Generating constraints

$(\text{let } id^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((id^{8}\ id^{9})^{7}\ 7^{10})^{6})^{1}$

$\lambda x^{b}.\ \dots\ x^{a}\ \dots$

$\text{let } x^{b} =\ \dots\ x^{a}\ \dots$

$\alpha_a \supseteq \alpha_b$

$\alpha_{10} \supseteq \{\ 7^{10}\ \}$

$\alpha_{3} \supseteq \{\ (\lambda x^{4}.\, x^{5})^{3}\ \}$

Generating constraints

$$(\text{let } id^2 = (\lambda x^4.\, x^5)^3 \text{ in } ((id^8\ id^9)^7\ 7^{10})^6)^1$$

$$\lambda x^4. \ldots x^5 \ldots \longrightarrow \alpha_5 \supseteq \alpha_4$$

$$\text{let } id^2 = \ldots id^8 \ldots \longrightarrow \alpha_8 \supseteq \alpha_2$$

$$\text{let } id^2 = \ldots id^9 \ldots \longrightarrow \alpha_9 \supseteq \alpha_2$$

$\alpha_{10} \supseteq \{ 7^{10} \}$

$\alpha_3 \supseteq \{ (\lambda x^4.\, x^5)^3 \}$

$\alpha_5 \supseteq \alpha_4$

$\alpha_8 \supseteq \alpha_2$

$\alpha_9 \supseteq \alpha_2$

Generating constraints

$(\text{let id}^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((\text{id}^{8}\ \text{id}^{9})^{7}\ 7^{10})^{6})^{1}$

$(\text{let } \_^{a} = \_^{b} \text{ in } \_^{c})^{d}$

$\alpha_d \supseteq \alpha_c$

$\alpha_a \supseteq \alpha_b$

$\alpha_{10} \supseteq \{\ 7^{10}\ \}$

$\alpha_3 \supseteq \{\ (\lambda x^{4}.\, x^{5})^{3}\ \}$

$\alpha_5 \supseteq \alpha_4$

$\alpha_8 \supseteq \alpha_2$

$\alpha_9 \supseteq \alpha_2$

Generating constraints

$(\text{let id}^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((\text{id}^{8}\ \text{id}^{9})^{7}\ 7^{10})^{6})^{1}$

$(\text{let } \_^{2} = \_^{3} \text{ in } \_^{6})^{1}$

$\alpha_1 \supseteq \alpha_6$

$\alpha_2 \supseteq \alpha_3$

$\alpha_{10} \supseteq \{ 7^{10} \}$	$\alpha_8 \supseteq \alpha_2$	$\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^{4}.\, x^{5})^{3} \}$	$\alpha_9 \supseteq \alpha_2$	
$\alpha_5 \supseteq \alpha_4$	$\alpha_1 \supseteq \alpha_6$	

Generating constraints

$(\text{let id}^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((\text{id}^{8}\ \text{id}^{9})^{7}\ 7^{10})^{6})^{1}$

$(\_^{a}\ \_^{b})^{c} \qquad (\alpha_b \mapsto \alpha_c) \supseteq \alpha_a$

$\alpha_{10} \supseteq \{\ 7^{10}\ \}$ $\quad$ $\alpha_8 \supseteq \alpha_2$ $\quad$ $\alpha_2 \supseteq \alpha_3$

$\alpha_3 \supseteq \{\ (\lambda x^{4}.\, x^{5})^{3}\ \}$ $\quad$ $\alpha_9 \supseteq \alpha_2$

$\alpha_5 \supseteq \alpha_4$ $\quad$ $\alpha_1 \supseteq \alpha_6$

Generating constraints

$$(\text{let id}^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((\text{id}^{8}\ \text{id}^{9})^{7}\ 7^{10})^{6})^{1}$$

$$(\_^{8}\ \_^{9})^{7} \longrightarrow (\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$$

$$(\_^{7}\ \_^{10})^{6} \longrightarrow (\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$$

$\alpha_{10} \supseteq \{\ 7^{10}\ \}$	$\alpha_8 \supseteq \alpha_2$	$\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{\ (\lambda x^{4}.\, x^{5})^{3}\ \}$	$\alpha_9 \supseteq \alpha_2$	$(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$	$\alpha_1 \supseteq \alpha_6$	$(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

Generating constraints

$$(\text{let id}^{2} = (\lambda x^{4}.\, x^{5})^{3} \text{ in } ((\text{id}^{8}\ \text{id}^{9})^{7}\ 7^{10})^{6})^{1}$$

$\alpha_{10} \supseteq \{\, 7^{10} \,\}$

$\alpha_{3} \supseteq \{\, (\lambda x^{4}.\, x^{5})^{3} \,\}$

$\alpha_{5} \supseteq \alpha_{4}$

$\alpha_{8} \supseteq \alpha_{2}$

$\alpha_{9} \supseteq \alpha_{2}$

$\alpha_{1} \supseteq \alpha_{6}$

$\alpha_{2} \supseteq \alpha_{3}$

$(\alpha_{9} \mapsto \alpha_{7}) \supseteq \alpha_{8}$

$(\alpha_{10} \mapsto \alpha_{6}) \supseteq \alpha_{7}$

Solving constraints

$\alpha_{10} \supseteq \{\ 7^{10}\ \}$

$\alpha_3 \supseteq \{\ (\lambda x^4.\, x^5)^3\ \}$

$\alpha_5 \supseteq \alpha_4$

$\alpha_8 \supseteq \alpha_2$

$\alpha_9 \supseteq \alpha_2$

$\alpha_1 \supseteq \alpha_6$

$\alpha_2 \supseteq \alpha_3$

$(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{\ \}$

$\alpha_2 = \{\ \}$

$\alpha_3 = \{\ \}$

$\alpha_4 = \{\ \}$

$\alpha_5 = \{\ \}$

$\alpha_6 = \{\ \}$

$\alpha_7 = \{\ \}$

$\alpha_8 = \{\ \}$

$\alpha_9 = \{\ \}$

$\alpha_{10} = \{\ \}$

Solving constraints

$\boxed{\alpha_{10} \supseteq \{ 7^{10} \}}$

$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$

$\alpha_5 \supseteq \alpha_4$

$\alpha_8 \supseteq \alpha_2$

$\alpha_9 \supseteq \alpha_2$

$\alpha_1 \supseteq \alpha_6$

$\alpha_2 \supseteq \alpha_3$

$(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \}$

$\alpha_2 = \{ \}$

$\alpha_3 = \{ \}$

$\alpha_4 = \{ \}$

$\alpha_5 = \{ \}$

$\alpha_6 = \{ \}$

$\alpha_7 = \{ \}$

$\alpha_8 = \{ \}$

$\alpha_9 = \{ \}$

$\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\quad$ $\alpha_8 \supseteq \alpha_2$ $\quad$ $\alpha_2 \supseteq \alpha_3$

$\boxed{\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}}$ $\quad$ $\alpha_9 \supseteq \alpha_2$ $\quad$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$\alpha_5 \supseteq \alpha_4$ $\quad$ $\alpha_1 \supseteq \alpha_6$ $\quad$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$$\alpha_1 = \{ \} \qquad \alpha_6 = \{ \}$$

$$\alpha_2 = \{ \} \qquad \alpha_7 = \{ \}$$

$$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \} \qquad \alpha_8 = \{ \}$$

$$\alpha_4 = \{ \} \qquad \alpha_9 = \{ \}$$

$$\alpha_5 = \{ \} \qquad \alpha_{10} = \{ 7^{10} \}$$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$	$\alpha_8 \supseteq \alpha_2$	$\boxed{\alpha_2 \supseteq \alpha_3}$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$	$\alpha_9 \supseteq \alpha_2$	$(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$	$\alpha_1 \supseteq \alpha_6$	$(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \}$	$\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4 . x^5)^3 \}$	$\alpha_7 = \{ \}$
$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \}$	$\alpha_8 = \{ \}$
$\alpha_4 = \{ \}$	$\alpha_9 = \{ \}$
$\alpha_5 = \{ \}$	$\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$	$\boxed{\alpha_8 \supseteq \alpha_2}$	$\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \}$	$\alpha_9 \supseteq \alpha_2$	$(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$	$\alpha_1 \supseteq \alpha_6$	$(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \}$ $\qquad$ $\alpha_6 = \{ \}$

$\alpha_2 = \{ (\lambda x^4. x^5)^3 \}$ $\qquad$ $\alpha_7 = \{ \}$

$\alpha_3 = \{ (\lambda x^4. x^5)^3 \}$ $\qquad$ $\alpha_8 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_4 = \{ \}$ $\qquad$ $\alpha_9 = \{ \}$

$\alpha_5 = \{ \}$ $\qquad$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\quad$ $\alpha_8 \supseteq \alpha_2$ $\quad$ $\alpha_2 \supseteq \alpha_3$

$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$ $\quad$ $\boxed{\alpha_9 \supseteq \alpha_2}$ $\quad$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$\alpha_5 \supseteq \alpha_4$ $\quad$ $\alpha_1 \supseteq \alpha_6$ $\quad$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \}$ $\quad$ $\alpha_6 = \{ \}$

$\alpha_2 = \{ (\lambda x^4 . x^5)^3 \}$ $\quad$ $\alpha_7 = \{ \}$

$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \}$ $\quad$ $\alpha_8 = \{ (\lambda x^4 . x^5)^3 \}$

$\alpha_4 = \{ \}$ $\quad$ $\alpha_9 = \{ (\lambda x^4 . x^5)^3 \}$

$\alpha_5 = \{ \}$ $\quad$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$	$\alpha_8 \supseteq \alpha_2$	$\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \}$	$\alpha_9 \supseteq \alpha_2$	$\boxed{(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8}$
$\alpha_5 \supseteq \alpha_4$	$\alpha_1 \supseteq \alpha_6$	$(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_1 = \{ \}$	$\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4. x^5)^3 \}$	$\alpha_7 = \{ \}$
$\alpha_3 = \{ (\lambda x^4. x^5)^3 \}$	$\alpha_8 = \{ \boxed{(\lambda x^4. x^5)^3} \}$
$\alpha_4 = \{ \}$	$\alpha_9 = \{ (\lambda x^4. x^5)^3 \}$
$\alpha_5 = \{ \}$	$\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{\, 7^{10} \,\}$ | $\alpha_8 \supseteq \alpha_2$ | $\alpha_2 \supseteq \alpha_3$

$\alpha_3 \supseteq \{\, (\lambda x^4 . x^5)^3 \,\}$ | $\alpha_9 \supseteq \alpha_2$ | $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$\alpha_5 \supseteq \alpha_4$ | $\alpha_1 \supseteq \alpha_6$ | $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\boxed{\alpha_4 \supseteq \alpha_9}$ $\alpha_7 \supseteq \alpha_5$

$$\alpha_1 = \{\,\} \qquad \alpha_6 = \{\,\}$$

$$\alpha_2 = \{\, (\lambda x^4 . x^5)^3 \,\} \qquad \alpha_7 = \{\,\}$$

$$\alpha_3 = \{\, (\lambda x^4 . x^5)^3 \,\} \qquad \alpha_8 = \{\, (\lambda x^4 . x^5)^3 \,\}$$

$$\alpha_4 = \{\, (\lambda x^4 . x^5)^3 \,\} \qquad \alpha_9 = \{\, (\lambda x^4 . x^5)^3 \,\}$$

$$\alpha_5 = \{\,\} \qquad \alpha_{10} = \{\, 7^{10} \,\}$$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$	$\alpha_8 \supseteq \alpha_2$	$\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4 . x^5)^3 \}$	$\alpha_9 \supseteq \alpha_2$	$(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\boxed{\alpha_5 \supseteq \alpha_4}$	$\alpha_1 \supseteq \alpha_6$	$(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$
$\alpha_4 \supseteq \alpha_9$	$\alpha_7 \supseteq \alpha_5$	

$\alpha_1 = \{ \}$	$\alpha_6 = \{ \}$
$\alpha_2 = \{ (\lambda x^4 . x^5)^3 \}$	$\alpha_7 = \{ \}$
$\alpha_3 = \{ (\lambda x^4 . x^5)^3 \}$	$\alpha_8 = \{ (\lambda x^4 . x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4 . x^5)^3 \}$	$\alpha_9 = \{ (\lambda x^4 . x^5)^3 \}$
$\alpha_5 = \{ (\lambda x^4 . x^5)^3 \}$	$\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$

$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \}$

$\alpha_5 \supseteq \alpha_4$

$\alpha_4 \supseteq \alpha_9$

$\boxed{\alpha_7 \supseteq \alpha_5}$

$\alpha_8 \supseteq \alpha_2$

$\alpha_9 \supseteq \alpha_2$

$\alpha_1 \supseteq \alpha_6$

$\alpha_2 \supseteq \alpha_3$

$(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$$\alpha_1 = \{ \}$$

$$\alpha_2 = \{ (\lambda x^4. x^5)^3 \}$$

$$\alpha_3 = \{ (\lambda x^4. x^5)^3 \}$$

$$\alpha_4 = \{ (\lambda x^4. x^5)^3 \}$$

$$\alpha_5 = \{ (\lambda x^4. x^5)^3 \}$$

$$\alpha_6 = \{ \}$$

$$\alpha_7 = \{ (\lambda x^4. x^5)^3 \}$$

$$\alpha_8 = \{ (\lambda x^4. x^5)^3 \}$$

$$\alpha_9 = \{ (\lambda x^4. x^5)^3 \}$$

$$\alpha_{10} = \{ 7^{10} \}$$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$

$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \}$

$\alpha_5 \supseteq \alpha_4$

$\alpha_4 \supseteq \alpha_9$

$\alpha_7 \supseteq \alpha_5$

$\alpha_8 \supseteq \alpha_2$

$\alpha_9 \supseteq \alpha_2$

$\alpha_1 \supseteq \alpha_6$

$\alpha_2 \supseteq \alpha_3$

$(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$\boxed{(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7}$

$\alpha_1 = \{ \}$

$\alpha_2 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_3 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_4 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_5 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_6 = \{ \}$

$\alpha_7 = \{ \boxed{(\lambda x^4. x^5)^3} \}$

$\alpha_8 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_9 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{\ 7^{10}\ \}$ $\quad$ $\alpha_8 \supseteq \alpha_2$ $\quad$ $\alpha_2 \supseteq \alpha_3$

$\alpha_3 \supseteq \{\ (\lambda x^4. x^5)^3\ \}$ $\quad$ $\alpha_9 \supseteq \alpha_2$ $\quad$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$\alpha_5 \supseteq \alpha_4$ $\quad$ $\alpha_1 \supseteq \alpha_6$ $\quad$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_4 \supseteq \alpha_9$ $\quad$ $\alpha_7 \supseteq \alpha_5$ $\quad$ $\boxed{\alpha_4 \supseteq \alpha_{10}}$ $\quad$ $\alpha_6 \supseteq \alpha_5$

$$\alpha_1 = \{\ \} \qquad \alpha_6 = \{\ \}$$

$$\alpha_2 = \{\ (\lambda x^4. x^5)^3\ \} \qquad \alpha_7 = \{\ (\lambda x^4. x^5)^3\ \}$$

$$\alpha_3 = \{\ (\lambda x^4. x^5)^3\ \} \qquad \alpha_8 = \{\ (\lambda x^4. x^5)^3\ \}$$

$$\alpha_4 = \{\ (\lambda x^4. x^5)^3, 7^{10}\ \} \qquad \alpha_9 = \{\ (\lambda x^4. x^5)^3\ \}$$

$$\alpha_5 = \{\ (\lambda x^4. x^5)^3\ \} \qquad \alpha_{10} = \{\ 7^{10}\ \}$$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\quad$ $\alpha_8 \supseteq \alpha_2$ $\quad$ $\alpha_2 \supseteq \alpha_3$

$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \}$ $\quad$ $\alpha_9 \supseteq \alpha_2$ $\quad$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$\alpha_5 \supseteq \alpha_4$ $\quad$ $\alpha_1 \supseteq \alpha_6$ $\quad$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_4 \supseteq \alpha_9$ $\quad$ $\alpha_7 \supseteq \alpha_5$ $\quad$ $\alpha_4 \supseteq \alpha_{10}$ $\quad$ $\boxed{\alpha_6 \supseteq \alpha_5}$

$\alpha_1 = \{ \}$ $\quad$ $\alpha_6 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_2 = \{ (\lambda x^4. x^5)^3 \}$ $\quad$ $\alpha_7 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_3 = \{ (\lambda x^4. x^5)^3 \}$ $\quad$ $\alpha_8 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_4 = \{ (\lambda x^4. x^5)^3, 7^{10} \}$ $\quad$ $\alpha_9 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_5 = \{ (\lambda x^4. x^5)^3 \}$ $\quad$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{\, 7^{10} \,\}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$

$\alpha_3 \supseteq \{\, (\lambda x^4.\, x^5)^3 \,\}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$\boxed{\alpha_5 \supseteq \alpha_4}$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$ $\alpha_4 \supseteq \alpha_{10}$ $\alpha_6 \supseteq \alpha_5$

$\alpha_1 = \{\,\}$ $\alpha_6 = \{\, (\lambda x^4.\, x^5)^3 \,\}$

$\alpha_2 = \{\, (\lambda x^4.\, x^5)^3 \,\}$ $\alpha_7 = \{\, (\lambda x^4.\, x^5)^3 \,\}$

$\alpha_3 = \{\, (\lambda x^4.\, x^5)^3 \,\}$ $\alpha_8 = \{\, (\lambda x^4.\, x^5)^3 \,\}$

$\alpha_4 = \{\, (\lambda x^4.\, x^5)^3, 7^{10} \,\}$ $\alpha_9 = \{\, (\lambda x^4.\, x^5)^3 \,\}$

$\alpha_5 = \{\, (\lambda x^4.\, x^5)^3, 7^{10} \,\}$ $\alpha_{10} = \{\, 7^{10} \,\}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\quad$ $\alpha_8 \supseteq \alpha_2$ $\quad$ $\alpha_2 \supseteq \alpha_3$

$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \}$ $\quad$ $\alpha_9 \supseteq \alpha_2$ $\quad$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$\alpha_5 \supseteq \alpha_4$ $\quad$ $\alpha_1 \supseteq \alpha_6$ $\quad$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_4 \supseteq \alpha_9$ $\quad$ $\boxed{\alpha_7 \supseteq \alpha_5}$ $\quad$ $\alpha_4 \supseteq \alpha_{10}$ $\quad$ $\alpha_6 \supseteq \alpha_5$

$\alpha_1 = \{ \}$ $\qquad$ $\alpha_6 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_2 = \{ (\lambda x^4. x^5)^3 \}$ $\qquad$ $\alpha_7 = \{ (\lambda x^4. x^5)^3, 7^{10} \}$

$\alpha_3 = \{ (\lambda x^4. x^5)^3 \}$ $\qquad$ $\alpha_8 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_4 = \{ (\lambda x^4. x^5)^3, 7^{10} \}$ $\qquad$ $\alpha_9 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_5 = \{ (\lambda x^4. x^5)^3, 7^{10} \}$ $\qquad$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$	$\alpha_8 \supseteq \alpha_2$	$\alpha_2 \supseteq \alpha_3$
$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \}$	$\alpha_9 \supseteq \alpha_2$	$(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$
$\alpha_5 \supseteq \alpha_4$	$\boxed{\alpha_1 \supseteq \alpha_6}$	$(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_4 \supseteq \alpha_9$ $\quad$ $\alpha_7 \supseteq \alpha_5$ $\quad$ $\alpha_4 \supseteq \alpha_{10}$ $\quad$ $\alpha_6 \supseteq \alpha_5$

$\alpha_1 = \{ (\lambda x^4. x^5)^3 \}$	$\alpha_6 = \{ (\lambda x^4. x^5)^3 \}$
$\alpha_2 = \{ (\lambda x^4. x^5)^3 \}$	$\alpha_7 = \{ (\lambda x^4. x^5)^3, 7^{10} \}$
$\alpha_3 = \{ (\lambda x^4. x^5)^3 \}$	$\alpha_8 = \{ (\lambda x^4. x^5)^3 \}$
$\alpha_4 = \{ (\lambda x^4. x^5)^3, 7^{10} \}$	$\alpha_9 = \{ (\lambda x^4. x^5)^3 \}$
$\alpha_5 = \{ (\lambda x^4. x^5)^3, 7^{10} \}$	$\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{ 7^{10} \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$

$\alpha_3 \supseteq \{ (\lambda x^4. x^5)^3 \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$\alpha_5 \supseteq \alpha_4$ $\alpha_1 \supseteq \alpha_6$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$ $\alpha_4 \supseteq \alpha_{10}$ $\boxed{\alpha_6 \supseteq \alpha_5}$

$\alpha_1 = \{ (\lambda x^4. x^5)^3 \}$ $\alpha_6 = \{ (\lambda x^4. x^5)^3, 7^{10} \}$

$\alpha_2 = \{ (\lambda x^4. x^5)^3 \}$ $\alpha_7 = \{ (\lambda x^4. x^5)^3, 7^{10} \}$

$\alpha_3 = \{ (\lambda x^4. x^5)^3 \}$ $\alpha_8 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_4 = \{ (\lambda x^4. x^5)^3, 7^{10} \}$ $\alpha_9 = \{ (\lambda x^4. x^5)^3 \}$

$\alpha_5 = \{ (\lambda x^4. x^5)^3, 7^{10} \}$ $\alpha_{10} = \{ 7^{10} \}$

Solving constraints

$\alpha_{10} \supseteq \{\ 7^{10}\ \}$ $\alpha_8 \supseteq \alpha_2$ $\alpha_2 \supseteq \alpha_3$

$\alpha_3 \supseteq \{\ (\lambda x^4.\, x^5)^3\ \}$ $\alpha_9 \supseteq \alpha_2$ $(\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8$

$\alpha_5 \supseteq \alpha_4$ $\boxed{\alpha_1 \supseteq \alpha_6}$ $(\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7$

$\alpha_4 \supseteq \alpha_9$ $\alpha_7 \supseteq \alpha_5$ $\alpha_4 \supseteq \alpha_{10}$ $\alpha_6 \supseteq \alpha_5$

$\alpha_1 = \{\ (\lambda x^4.\, x^5)^3, 7^{10}\ \}$ $\alpha_6 = \{\ (\lambda x^4.\, x^5)^3, 7^{10}\ \}$

$\alpha_2 = \{\ (\lambda x^4.\, x^5)^3\ \}$ $\alpha_7 = \{\ (\lambda x^4.\, x^5)^3, 7^{10}\ \}$

$\alpha_3 = \{\ (\lambda x^4.\, x^5)^3\ \}$ $\alpha_8 = \{\ (\lambda x^4.\, x^5)^3\ \}$

$\alpha_4 = \{\ (\lambda x^4.\, x^5)^3, 7^{10}\ \}$ $\alpha_9 = \{\ (\lambda x^4.\, x^5)^3\ \}$

$\alpha_5 = \{\ (\lambda x^4.\, x^5)^3, 7^{10}\ \}$ $\alpha_{10} = \{\ 7^{10}\ \}$

Using solutions

$$\alpha_{10} = \{\, 7^{10} \,\}$$

$$\alpha_2, \alpha_3, \alpha_8, \alpha_9 = \{\, (\lambda x^4 .\, x^5)^3 \,\}$$

$$\alpha_1, \alpha_4, \alpha_5, \alpha_6, \alpha_7 = \{\, (\lambda x^4 .\, x^5)^3, 7^{10} \,\}$$

$$(\text{let } id^2 = (\lambda x^4 .\, x^5)^3 \text{ in } ((id^8\ id^9)^7\ 7^{10})^6)^1$$

Limitations

0CFA is still imprecise because it is *monovariant*: each expression has only one flow variable associated with it, so multiple calls to the same function allow multiple values into the single flow variable for the function body, and these values “leak out” at all potential call sites.

Limitations

$\alpha_{10} \supseteq \{\, 7^{10} \,\}$

$\alpha_3 \supseteq \{\, (\lambda x^4.\, x^5)^3 \,\}$

$\alpha_5 \supseteq \alpha_4$

$\boxed{\alpha_4 \supseteq \alpha_9}$

$\alpha_7 \supseteq \alpha_5$

$\alpha_8 \supseteq \alpha_2$

$\alpha_9 \supseteq \alpha_2$

$\alpha_1 \supseteq \alpha_6$

$\boxed{\alpha_4 \supseteq \alpha_{10}}$

$\alpha_2 \supseteq \alpha_3$

$\boxed{\begin{array}{l} (\alpha_9 \mapsto \alpha_7) \supseteq \alpha_8 \\ (\alpha_{10} \mapsto \alpha_6) \supseteq \alpha_7 \end{array}}$

$\alpha_6 \supseteq \alpha_5$

$$\alpha_7 = \{\boxed{(\lambda x^4.\, x^5)^3}\} \qquad \alpha_8 = \{\boxed{(\lambda x^4.\, x^5)^3}\}$$

$$\alpha_9 = \{\, (\lambda x^4.\, x^5)^3 \,\} \qquad \alpha_{10} = \{\, 7^{10} \,\}$$

$$(\text{let id}^2 = (\lambda x^4.\, x^5)^3 \text{ in } ((\text{id}^8\ \text{id}^9)^7\ 7^{10})^6)^1$$

1CFA

0CFA is still imprecise because it is *monovariant*: each expression has only one flow variable associated with it, so multiple calls to the same function allow multiple values into the single flow variable for the function body, and these values “leak out” at all potential call sites.

A better approximation is given by 1CFA (“first-order...”), in which a function has a separate flow variable for each call site in the program; this isolates separate calls to the same function, and so produces a more precise result.

1CFA

1CFA is a *polyvariant* approach.

Another alternative is to use a *polymorphic* approach, in which the values themselves are enriched to support specialisation at different call sites (cf. ML polymorphic types).

It's unclear which approach is "best".

Summary

- Many analyses can be formulated using constraints
- 0CFA is a constraint-based analysis
- Inequality constraints are generated from the syntax of a program
- A minimal solution to the constraints provides a safe approximation to dynamic control-flow behaviour
- Polyvariant (as in 1CFA) and polymorphic approaches may improve precision

Lecture 12
Inference-based analysis

Motivation

In this part of the course we're examining several methods of higher-level program analysis.

We have so far seen *abstract interpretation* and *constraint-based analysis*, two general frameworks for formally specifying (and performing) analyses of programs.

Another alternative framework is *inference-based analysis*.

Inference-based analysis

Inference systems consist of *sets of rules* for determining *program properties*.

Typically such a property of an entire program depends recursively upon the properties of the program's subexpressions; inference systems can directly express this relationship, and show how to recursively compute the property.

Inference-based analysis

An inference system specifies judgements:

$$\Gamma \vdash e : \phi$$

- e is an expression (e.g. a complete program)
- Γ is a set of assumptions about free variables of e
- ϕ is a program property

Type systems

Consider the ML type system, for example.

This particular inference system specifies judgements about a *well-typedness* property:

$$\Gamma \vdash e : t$$

means "under the assumptions in Γ, the expression e has type *t*".

Type systems

We will avoid the more complicated ML typing issues (see Types course for details) and just consider the expressions in the lambda calculus:

$$e ::= x \mid \lambda x.\, e \mid e_1\ e_2$$

Our program properties are types *t*:

$$t ::= \alpha \mid int \mid t_1 \rightarrow t_2$$

Type systems

Γ is a set of *type assumptions* of the form

$$\{ x_1 : t_1, \ldots, x_n : t_n \}$$

where each identifier x_i is assumed to have type t_i.

We write

$$\Gamma[x : t]$$

to mean Γ with the additional assumption that x has type t (overriding any other assumption about x).

Type systems

In all inference systems, we use a set of *rules* to inductively define which judgements are valid.

In a type system, these are the *typing rules*.

Type systems

$$\frac{}{\Gamma[x:t] \vdash x:t} \quad (\textsc{Var})$$

$$\frac{\Gamma[x:t] \vdash e:t'}{\Gamma \vdash \lambda x.e : t \rightarrow t'} \quad (\textsc{Lam})$$

$$\frac{\Gamma \vdash e_1 : t \rightarrow t' \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 e_2 : t'} \quad (\textsc{App})$$

Type systems

$$\Gamma = \{\, 2 : int,\ add : int \to int \to int,\ multiply : int \to int \to int \,\}$$

$$e = \lambda x.\, \lambda y.\, add\ (multiply\ 2\ x)\ y$$

$$t = {\color{red} int \to int \to int}$$

$$\dfrac{\dfrac{\dfrac{\Gamma[x : int][y : int] \vdash add : int \to int \to int \qquad \dfrac{\vdots}{\Gamma[x : int][y : int] \vdash multiply\ 2\ x : int}}{\Gamma[x : int][y : int] \vdash add\ (multiply\ 2\ x) : int \to int} \qquad \dfrac{}{\Gamma[x : int][y : int] \vdash y : int}}{\dfrac{\Gamma[x : int][y : int] \vdash add\ (multiply\ 2\ x)\ y : int}{\Gamma[x : int] \vdash \lambda y.\, add\ (multiply\ 2\ x)\ y : int \to int}}}{\Gamma \vdash \lambda x.\, \lambda y.\, add\ (multiply\ 2\ x)\ y : int \to int \to int}$$

Optimisation

In the absence of a compile-time type checker, all values must be tagged with their types and run-time checks must be performed to ensure types match appropriately.

If a type system has shown that the program is well-typed, execution can proceed safely without these tags and checks; if necessary, the final result of evaluation can be tagged with its inferred type.

Hence the final result of evaluation is identical, but less run-time computation is required to produce it.

Safety

The safety condition for this inference system is

$$(\{\} \vdash e : t) \Rightarrow (\llbracket e \rrbracket \in \llbracket t \rrbracket)$$

where $\llbracket e \rrbracket$ and $\llbracket t \rrbracket$ are the *denotations* of e and *t* respectively: $\llbracket e \rrbracket$ is the value obtained by evaluating e, and $\llbracket t \rrbracket$ is the set of all values of type *t*.

This condition asserts that the run-time behaviour of the program will agree with the type system's prediction.

Odds and evens

Type-checking is just one application of inference-based program analysis.

The properties do not have to be types; in particular, they can carry more (or completely different!) information than traditional types do.

We'll consider a more program-analysis–related example: detecting odd and even numbers.

Odds and evens

This time, the program property ϕ has the form

$$\phi ::= \mathit{odd} \mid \mathit{even} \mid \phi_1 \to \phi_2$$

Odds and evens

$$\frac{}{\Gamma[x:\phi] \vdash x:\phi} \quad (\textsc{Var})$$

$$\frac{\Gamma[x:\phi] \vdash e:\phi'}{\Gamma \vdash \lambda x.e : \phi \to \phi'} \quad (\textsc{Lam})$$

$$\frac{\Gamma \vdash e_1 : \phi \to \phi' \quad \Gamma \vdash e_2 : \phi}{\Gamma \vdash e_1 e_2 : \phi'} \quad (\textsc{App})$$

Odds and evens

$\Gamma = \{\, 2 : \mathit{even},\ \mathit{add} : \mathit{even} \rightarrow \mathit{even} \rightarrow \mathit{even},$
$\qquad \mathit{multiply} : \mathit{even} \rightarrow \mathit{odd} \rightarrow \mathit{even} \,\}$

$e = \lambda x.\, \lambda y.\, \mathit{add}\ (\mathit{multiply}\ 2\ x)\ y$

$\phi = \mathit{odd} \rightarrow \mathit{even} \rightarrow \mathit{even}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma[x : \mathit{odd}][y : \mathit{even}] \vdash \mathit{add} : \mathit{even} \rightarrow \mathit{even} \rightarrow \mathit{even}} \quad \dfrac{\vdots}{\Gamma[x : \mathit{odd}][y : \mathit{even}] \vdash \mathit{multiply}\ 2\ x : \mathit{even}}}{\Gamma[x : \mathit{odd}][y : \mathit{even}] \vdash \mathit{add}\ (\mathit{multiply}\ 2\ x) : \mathit{even} \rightarrow \mathit{even}} \quad \dfrac{}{\Gamma[x : \mathit{odd}][y : \mathit{even}] \vdash y : \mathit{even}}}{\Gamma[x : \mathit{odd}][y : \mathit{even}] \vdash \mathit{add}\ (\mathit{multiply}\ 2\ x)\ y : \mathit{even}}}{\Gamma[x : \mathit{odd}] \vdash \lambda y.\, \mathit{add}\ (\mathit{multiply}\ 2\ x)\ y : \mathit{even} \rightarrow \mathit{even}}}{\Gamma \vdash \lambda x.\, \lambda y.\, \mathit{add}\ (\mathit{multiply}\ 2\ x)\ y : \mathit{odd} \rightarrow \mathit{even} \rightarrow \mathit{even}}$$

Safety

The safety condition for this inference system is

$$(\{\} \vdash e : \phi) \Rightarrow ([\![e]\!] \in [\![\phi]\!])$$

where $[\![\phi]\!]$ is the denotation of ϕ:

$$[\![\mathit{odd}]\!] = \{ z \in \mathbb{Z} \mid z \text{ is odd} \},$$

$$[\![\mathit{even}]\!] = \{ z \in \mathbb{Z} \mid z \text{ is even} \},$$

$$[\![\phi_1 \to \phi_2]\!] = [\![\phi_1]\!] \to [\![\phi_2]\!]$$

Richer properties

Note that if we want to show a judgement like

$$\Gamma \vdash \lambda x.\, \lambda y.\, add\ (multiply\ 2\ x)\ (multiply\ 3\ y) : even \to even \to even$$

we need more than one assumption about *multiply*:

$$\begin{aligned} \Gamma = \{ \ldots,\ & multiply : even \to even \to even, \\ & multiply : odd \to even \to even, \ldots \} \end{aligned}$$

Richer properties

This might be undesirable, and one alternative is to enrich our properties instead; in this case we could allow *conjunction* inside properties, so that our single assumption about *multiply* looks like:

$$\begin{aligned} \mathit{multiply} : \; & \mathit{even} \to \mathit{even} \to \mathit{even} \; \wedge \\ & \mathit{even} \to \mathit{odd} \to \mathit{even} \; \wedge \\ & \mathit{odd} \to \mathit{even} \to \mathit{even} \; \wedge \\ & \mathit{odd} \to \mathit{odd} \to \mathit{odd} \end{aligned}$$

We would need to modify the inference system to handle these richer properties.

Summary

- Inference-based analysis is another useful framework
- Inference rules are used to produce judgements about programs and their properties
- Type systems are the best-known example
- Richer properties give more detailed information
- An inference system used for analysis has an associated safety condition

Lecture 13
Effect systems

Motivation

We have so far seen many analyses which deal with control- and data-flow properties of pure languages.

However, many languages contain operations with *side-effects*, so we must also be able to analyse and safely transform these impure programs.

Effect systems, a form of inference-based analysis, are often used for this purpose.

Side-effects

A side-effect is some event — typically a *change of state* — which occurs as a result of evaluating an expression.

- "`x++`" changes the value of variable `x`.
- "`malloc(42)`" allocates some memory.
- "`print 42`" outputs a value to a stream.

Side-effects

As an example language, we will use the lambda calculus extended with *read* and *write* operations on "channels".

$$e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid \xi?x.e \mid \xi!e_1.e_2$$

- ξ represents some channel name.
- $\xi?x.e$ reads an integer from the channel named ξ, binds it to x, and returns the result of evaluating e.
- $\xi!e_1.e_2$ evaluates e_1, writes the resulting integer to channel ξ, and returns the result of evaluating e_2.

Side-effects

Some example expressions:

$$\xi?x.\ x$$

read an integer from channel ξ and return it

$$\xi!x.\ y$$

write the (integer) value of *x* to channel ξ and return the value of *y*

$$\xi?x.\ \zeta!x.\ x$$

read an integer from channel ξ, write it to channel ζ and return it

Side-effects

Ignoring their side-effects, the typing rules for these new operations are straightforward.

Side-effects

$$\frac{\Gamma[x : int] \vdash e : t}{\Gamma \vdash \xi?x.e : t} \quad (\textsc{Read})$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \xi!e_1.e_2 : t} \quad (\textsc{Write})$$

Effect systems

However, in order to perform any transformations on a program in this language it would be necessary to pay attention to its potential side-effects.

For example, we might need to devise an analysis to tell us which channels may be read or written during evaluation of an expression.

We can do this by modifying our existing type system to create an *effect system* (or "type and effect system").

Effect systems

First we must formally define our *effects*:

An expression has effects F.
F is a set containing elements of the form

R_ξ read from channel ξ

W_ξ write to channel ξ

Effect systems

For example:

$$\xi?x.\,x \qquad F = \{\, R_\xi \,\}$$

$$\xi!x.\,y \qquad F = \{\, W_\xi \,\}$$

$$\xi?x.\,\zeta!x.\,x \qquad F = \{\, R_\xi, W_\zeta \,\}$$

Effect systems

But we also need to be able to handle expressions like

$$\lambda x.\, \xi !x.\, x$$

whose evaluation doesn't have any *immediate* effects.

In this case, the effect W_ξ may occur *later*, whenever this newly-created function is applied.

Effect systems

To handle these *latent effects* we extend the syntax of types so that function types are annotated with the effects that may occur when a function is applied:

$$t ::= int \mid t_1 \xrightarrow{F} t_2$$

Effect systems

So, although it has no immediate effects, the type of

$$\lambda x.\ \xi!x.\ x$$

is

$$\text{int} \xrightarrow{\{\ W_\xi\ \}} \text{int}$$

Effect systems

We can now modify the existing type system to make an effect system — an inference system which produces judgements about the type *and effects* of an expression:

$$\Gamma \vdash e : t, F$$

Effect systems

$$\frac{\Gamma[x : int] \vdash e : t}{\Gamma \vdash \xi?x.e : t} \quad (\textsc{Read})$$

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : t}{\Gamma \vdash \xi!e_1.e_2 : t} \quad (\textsc{Write})$$

Effect systems

$$\frac{\Gamma[x : int] \vdash e : t, F}{\Gamma \vdash \xi?x.e : t, \{R_\xi\} \cup F} \quad (\textsc{Read})$$

$$\frac{\Gamma \vdash e_1 : int, F \qquad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash \xi!e_1.e_2 : t, F \cup \{W_\xi\} \cup F'} \quad (\textsc{Write})$$

Effect systems

$$\frac{}{\Gamma[x:t] \vdash x : t, \{\}} \quad (\textsc{Var})$$

$$\frac{\Gamma[x:t] \vdash e : t', F}{\Gamma \vdash \lambda x.e : t \xrightarrow{F} t', \{\}} \quad (\textsc{Lam})$$

$$\frac{\Gamma \vdash e_1 : t \xrightarrow{F''} t', F \qquad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash e_1 e_2 : t', F \cup F' \cup F''} \quad (\textsc{App})$$

Effect systems

$$\cfrac{\cfrac{\cfrac{}{\{x : int, y : int\} \vdash x : int, \{\}}}{\{x : int, y : int\} \vdash \xi!x.\ x : int, \{W_\xi\}}}{\{y : int\} \vdash \lambda x.\xi!x.\ x : int \xrightarrow{\{W_\xi\}} int, \{\}} \qquad \cfrac{}{\{y : int\} \vdash y : int, \{\}}$$

$$\overline{\{y : int\} \vdash (\lambda x.\xi!x.\ x)\ y : int, \{W_\xi\}}$$

Effect subtyping

We would probably want more expressive control structure in a real programming language.

For example, we could add *if-then-else*:

$$e ::= x \mid \lambda x.e \mid e_1\ e_2 \mid \xi?x.e \mid \xi!e_1.e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$$

Effect subtyping

$$\frac{\Gamma \vdash e_1 : int, F \quad \Gamma \vdash e_2 : t, F' \quad \Gamma \vdash e_3 : t, F''}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : t, F \cup F' \cup F''} \quad (\textsc{Cond})$$

Effect subtyping

However, there are some valid uses of *if-then-else* which this rule cannot handle by itself.

Effect subtyping

$$\overset{int \xrightarrow{\{W_\xi\}} int}{\boxed{\lambda x.\ \xi!3.\ x + 1}}$$

if x then $\lambda x.\ \xi!3.\ x + 1$ else $\lambda x.\ x + 2$

$$\overset{int \xrightarrow{\{\}} int}{\boxed{\lambda x.\ x + 2}}$$

$$\frac{\Gamma \vdash e_1 : int, F \quad \Gamma \vdash e_2 : t, F' \quad \Gamma \vdash e_3 : t, F''}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : t, F \cup F' \cup F''}\ (\textsc{Cond})$$

Effect subtyping

✗

$$\overbrace{\lambda x.\ \xi!3.\ x + 1}^{\mathit{int} \xrightarrow{\{W_\xi\}} \mathit{int}} \qquad \overbrace{\lambda x.\ x + 2}^{\mathit{int} \xrightarrow{\{\ \}} \mathit{int}}$$

if x then $\lambda x.\ \xi!3.\ x + 1$ else $\lambda x.\ x + 2$

Effect subtyping

We can solve this problem by adding a new rule to handle *subtyping*.

Effect subtyping

$$\frac{\Gamma \vdash e : t \xrightarrow{F'} t', F \qquad F' \subseteq F''}{\Gamma \vdash e : t \xrightarrow{F''} t', F} \quad (\textsc{Sub})$$

Effect subtyping

$$\mathit{int} \xrightarrow{\{\,W_\xi\,\}} \mathit{int}$$

(Sub)

$$\mathit{int} \xrightarrow{\{\,W_\xi\,\}} \mathit{int} \qquad\qquad \mathit{int} \xrightarrow{\{\ \}} \mathit{int}$$

if x then $\lambda x.\ \xi!3.\ x + 1$ else $\lambda x.\ x + 2$

Effect subtyping

$$\mathit{int} \xrightarrow{\{W_\xi\}} \mathit{int}$$

$$\mathit{int} \xrightarrow{\{W_\xi\}} \mathit{int} \qquad \mathit{int} \xrightarrow{\{\ \}} \mathit{int}$$

if x then $\lambda x.\ \xi!3.\ x + 1$ else $\lambda x.\ x + 2$

Optimisation

The information discovered by the effect system is useful when deciding whether particular transformations are safe.

An expression with no immediate side-effects is *referentially transparent*: it can safely be replaced with another expression (with the same value and type) with no change to the semantics of the program.

For example, referentially transparent expressions may safely be removed if LVA says they are dead.

Safety

$$(\{\} \vdash e : t, F) \Rightarrow$$

$$(v \in [\![t]\!] \wedge f \subseteq F \text{ where } (v, f) = [\![e]\!])$$

Extra structure

In this analysis we are using *sets* of effects.

As a result, we aren't collecting any information about how many times each effect may occur, or the order in which they may happen.

$$\xi?x.\ \zeta!x.\ x \qquad F = \{\ R_\xi,\ W_\zeta\ \}$$

$$\zeta!y.\ \xi?x.\ x \qquad F = \{\ R_\xi,\ W_\zeta\ \}$$

$$\zeta!y.\ \xi?x.\ \zeta!x.\ x \qquad F = \{\ R_\xi,\ W_\zeta\ \}$$

Extra structure

If we use a different representation of effects, and use different operations on them, we can keep track of more information.

One option is to use *sequences* of effects and use an append operation when combining them.

Extra structure

$$\frac{\Gamma[x : int] \vdash e : t, F}{\Gamma \vdash \xi?x.e : t, \langle R_\xi \rangle \mathbin{@} F} \quad (\textsc{Read})$$

$$\frac{\Gamma \vdash e_1 : int, F \qquad \Gamma \vdash e_2 : t, F'}{\Gamma \vdash \xi!e_1.e_2 : t, F \mathbin{@} \langle W_\xi \rangle \mathbin{@} F'} \quad (\textsc{Write})$$

Extra structure

In the new system, these expressions all have different effects:

$$\xi?x.\ \zeta!x.\ x \qquad F = \langle\ R_\xi ; W_\zeta\ \rangle$$

$$\zeta!y.\ \xi?x.\ x \qquad F = \langle\ W_\zeta ; R_\xi\ \rangle$$

$$\zeta!y.\ \xi?x.\ \zeta!x.\ x \qquad F = \langle\ W_\zeta ; R_\xi ; W_\zeta\ \rangle$$

Extra structure

Whether we use sequences instead of sets depends upon whether we care about the order and number of effects. In the channel example, we probably don't.

But if we were tracking file accesses, it would be important to ensure that no further read or write effects occurred after a file had been closed.

And if we were tracking memory allocation, we would want to ensure that no block of memory got deallocated twice.

Summary

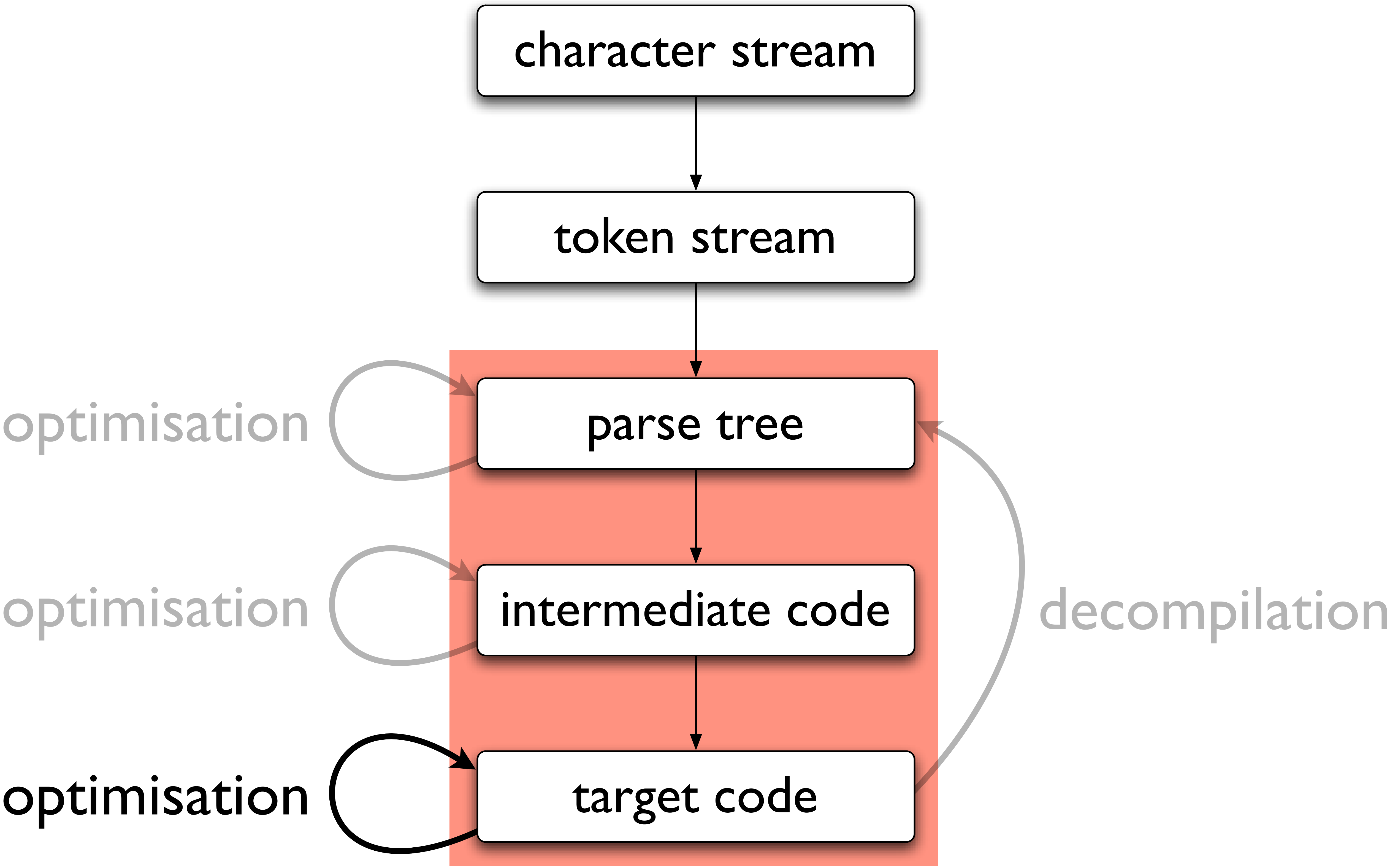
- Effect systems are a form of inference-based analysis
- Side-effects occur when expressions are evaluated
- Function types must be annotated to account for latent effects
- A type system can be modified to produce judgements about both types and effects
- Subtyping may be required to handle annotated types
- Different effect structures may give more information

Lecture 14
Instruction scheduling

Part C

Instruction scheduling

Instruction scheduling



Motivation

We have seen optimisation techniques which involve removing and reordering code at both the source- and intermediate-language levels in an attempt to achieve the smallest and fastest correct program.

These techniques are platform-independent, and pay little attention to the details of the target architecture.

We can improve target code if we consider the architectural characteristics of the target processor.

Single-cycle implementation

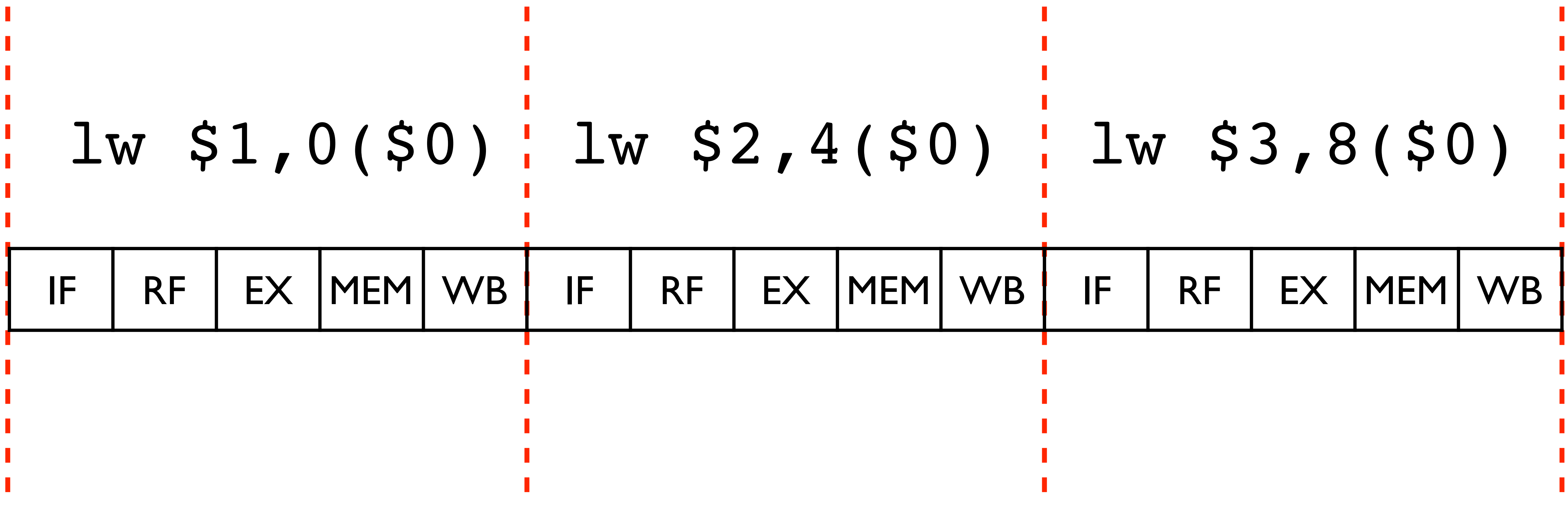
In *single-cycle* processor designs, an entire instruction is executed in a single clock cycle.

Each instruction will use some of the processor's processing stages:

Instruction fetch (IF)	Register fetch (RF)	Execute (EX)	Memory access (MEM)	Register write-back (WB)

For example, a load instruction uses all five.

Single-cycle implementation



Single-cycle implementation

On these processors, the order of instructions doesn't make any difference to execution time: each instruction takes one clock cycle, so *n* instructions will take *n* cycles and can be executed in any (correct) order.

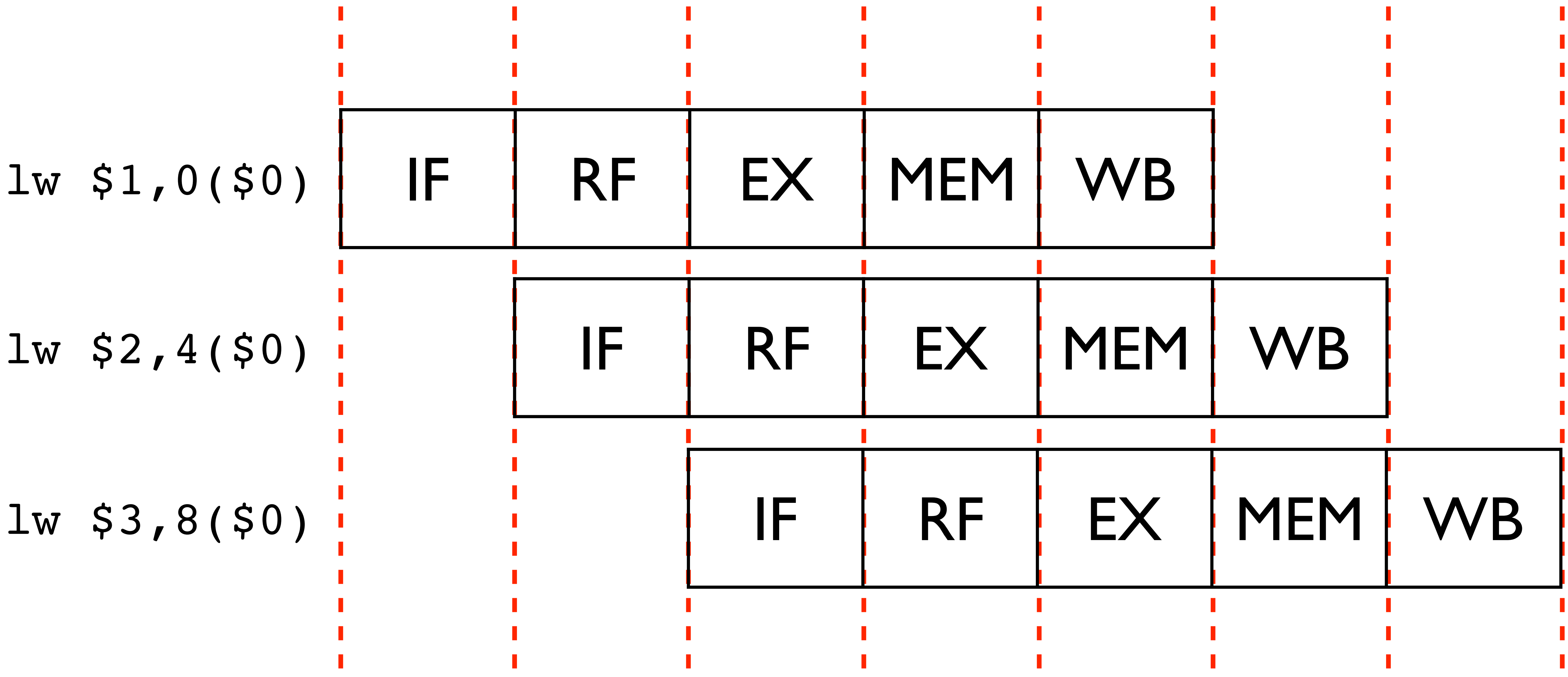
In this case we can naïvely translate our optimised 3-address code by expanding each intermediate instruction into the appropriate sequence of target instructions; clever reordering is unlikely to yield any benefits.

Pipelined implementation

In *pipelined* processor designs (e.g. MIPS R2000), each processing stage works independently and does its job in a single clock cycle, so different stages can be handling different instructions simultaneously.

These stages are arranged in a pipeline, and the result from each unit is passed to the next one via a pipeline register before the next clock cycle.

Pipelined implementation



Pipelined implementation

In this *multicycle* design the clock cycle is much shorter (one pipeline stage vs. one complete instruction) and ideally we can still execute one instruction per cycle when the pipeline is full.

Programs will therefore execute more quickly.

Pipeline hazards

However, it is not always possible to run the pipeline at full capacity.

Some situations prevent the next instruction from executing in the next clock cycle: this is a *pipeline hazard*.

On interlocked hardware (e.g. SPARC) a hazard will cause a *pipeline stall*; on non-interlocked hardware (e.g. MIPS) the compiler must generate explicit NOPs to avoid errors.

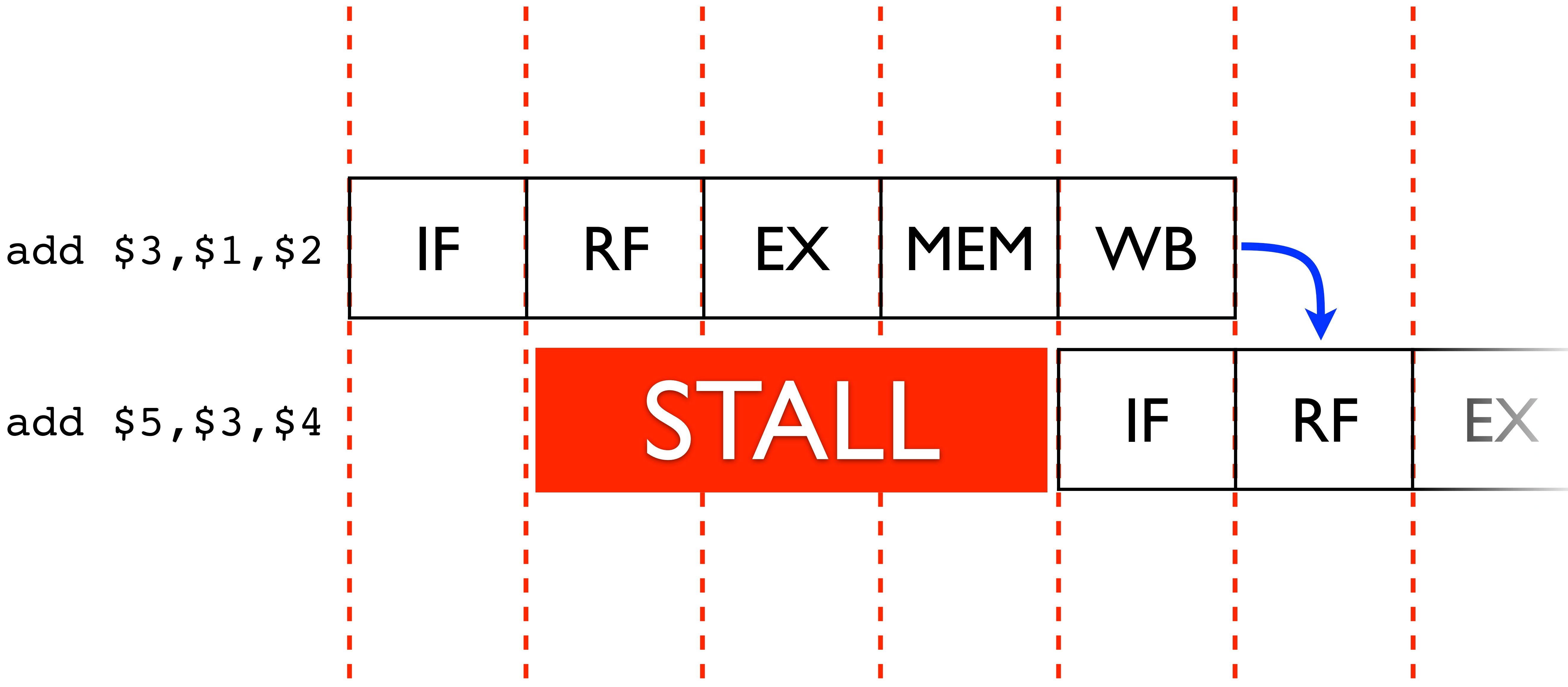
Pipeline hazards

Consider *data hazards*: these occur when an instruction depends upon the result of an earlier one.

```
add $3,$1,$2
add $5,$3,$4
```

The pipeline must stall until the result of the first `add` has been written back into register `$3`.

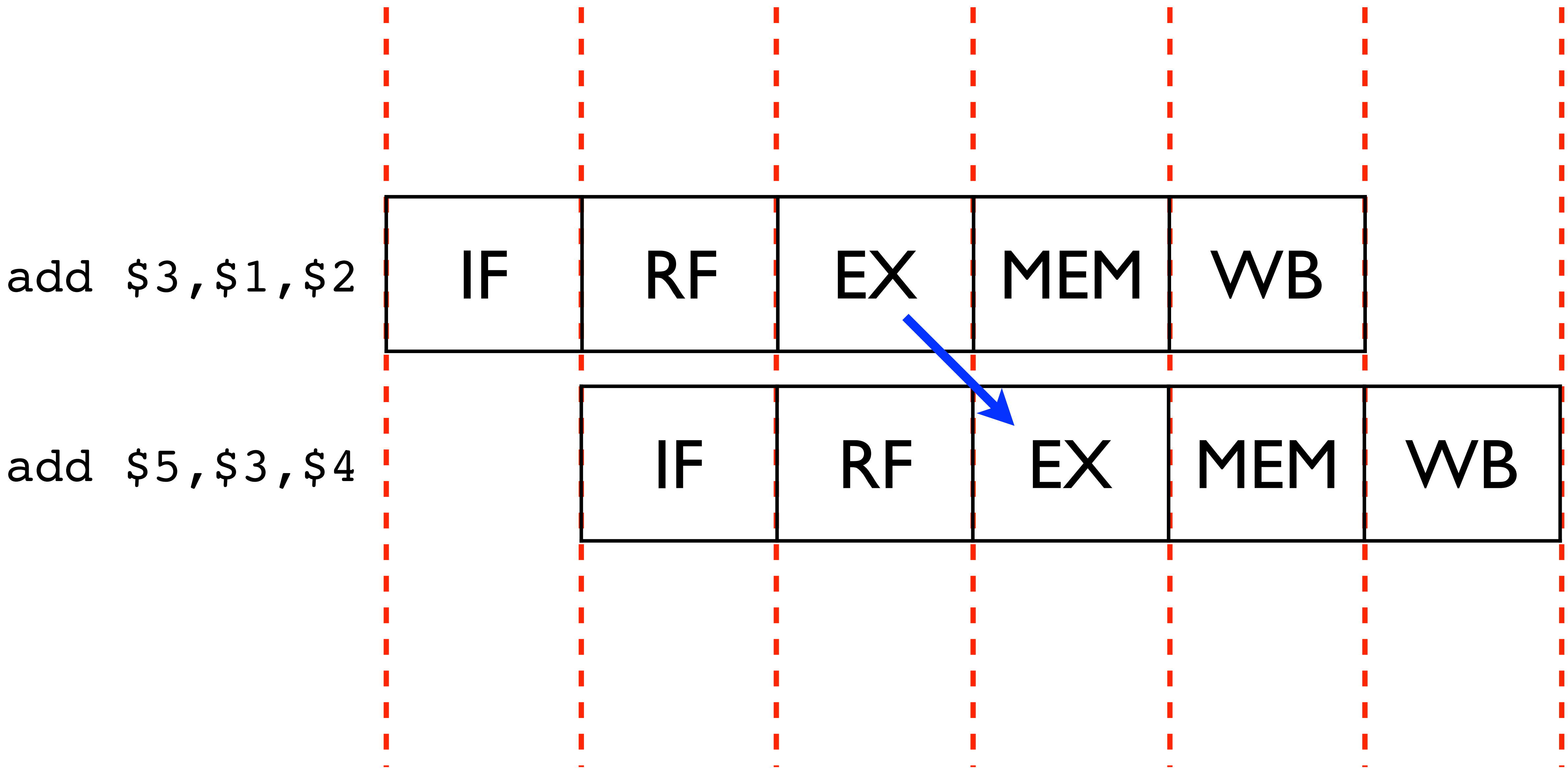
Pipeline hazards



Pipeline hazards

The severity of this effect can be reduced by using *bypassing*: extra paths are added between functional units, allowing data to be used before it has been written back into registers.

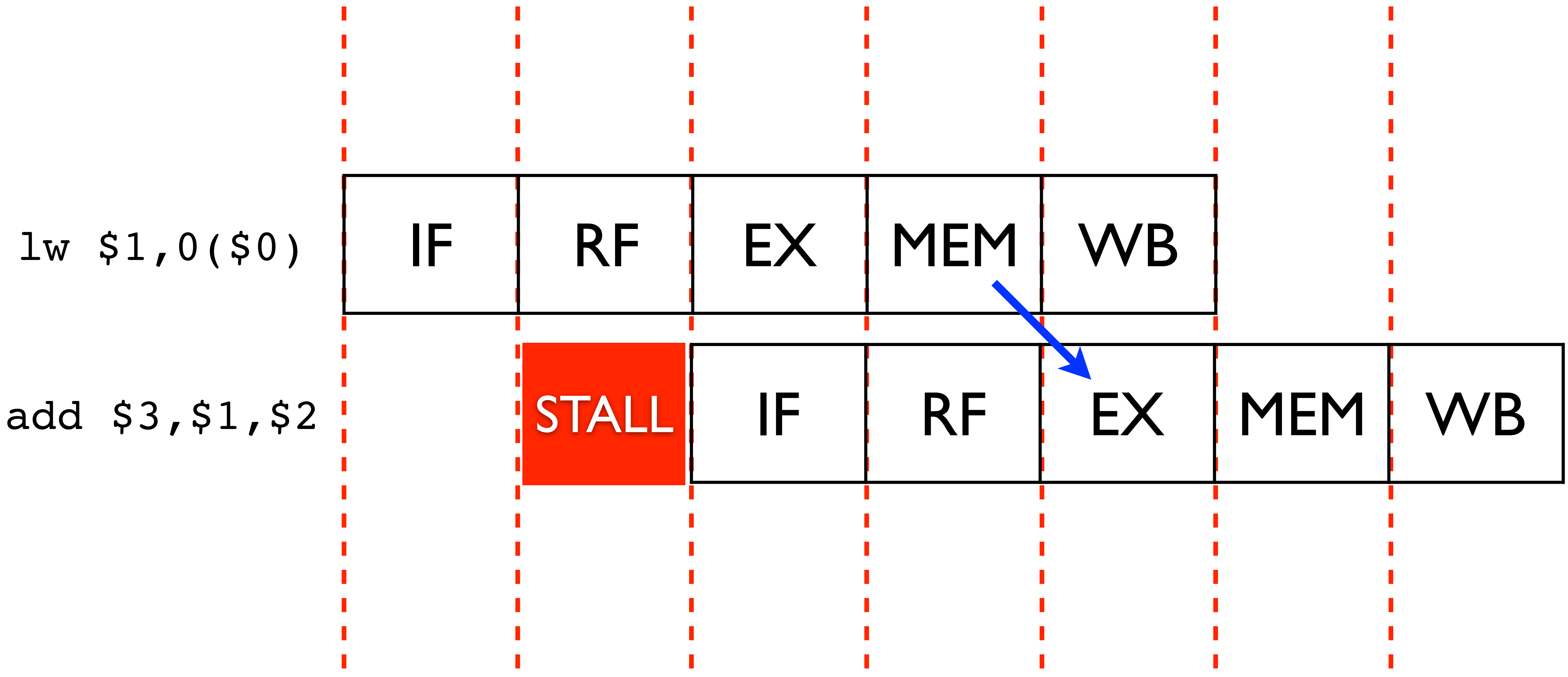
Pipeline hazards



Pipeline hazards

But even when bypassing is used, some combinations of instructions will always result in a stall.

Pipeline hazards

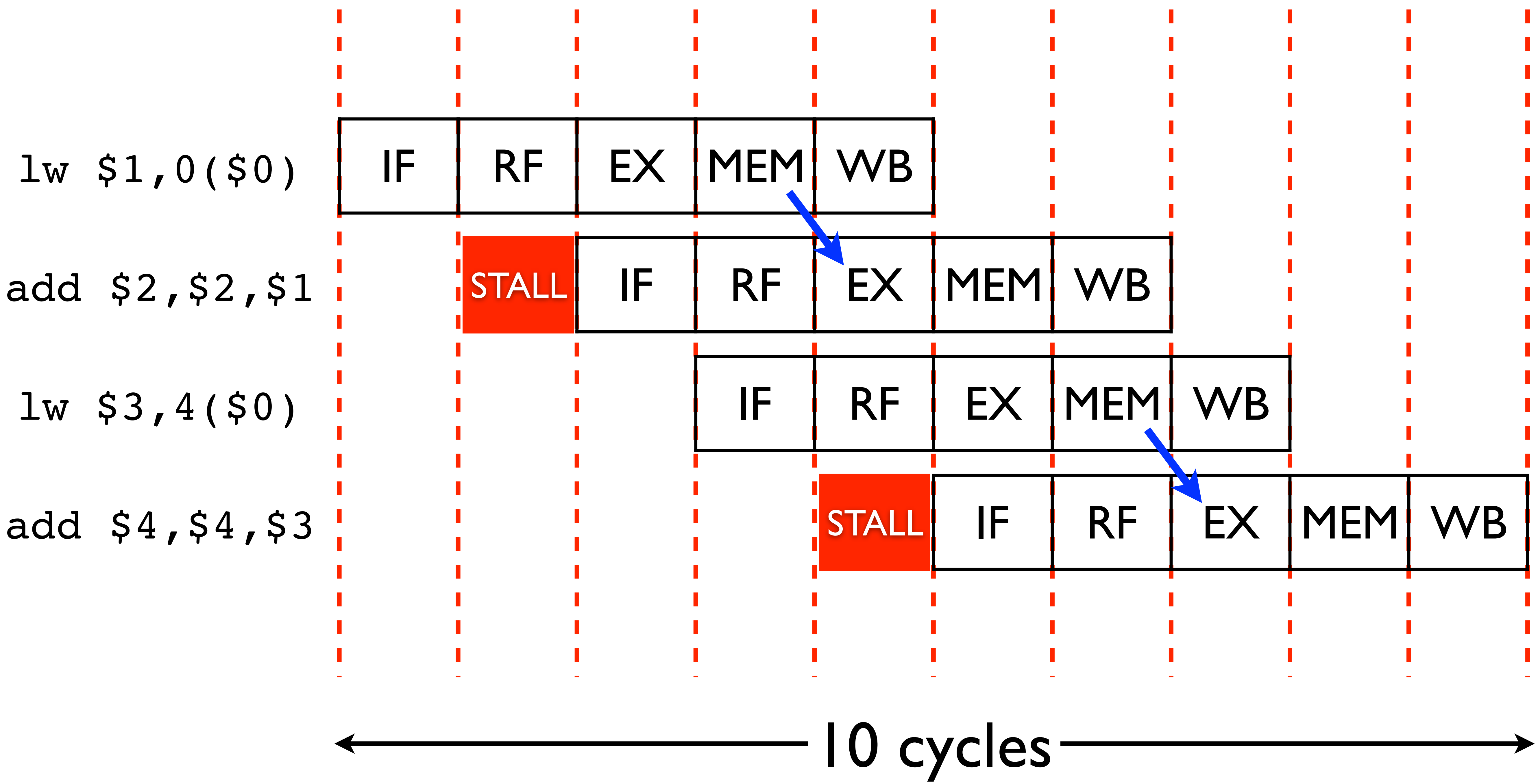


Instruction order

Since particular combinations of instructions cause this problem on pipelined architectures, we can achieve better performance by reordering instructions where possible.

```
lw $1,0($0)
add $2,$2,$1
lw $3,4($0)
add $4,$4,$3
```

Instruction order



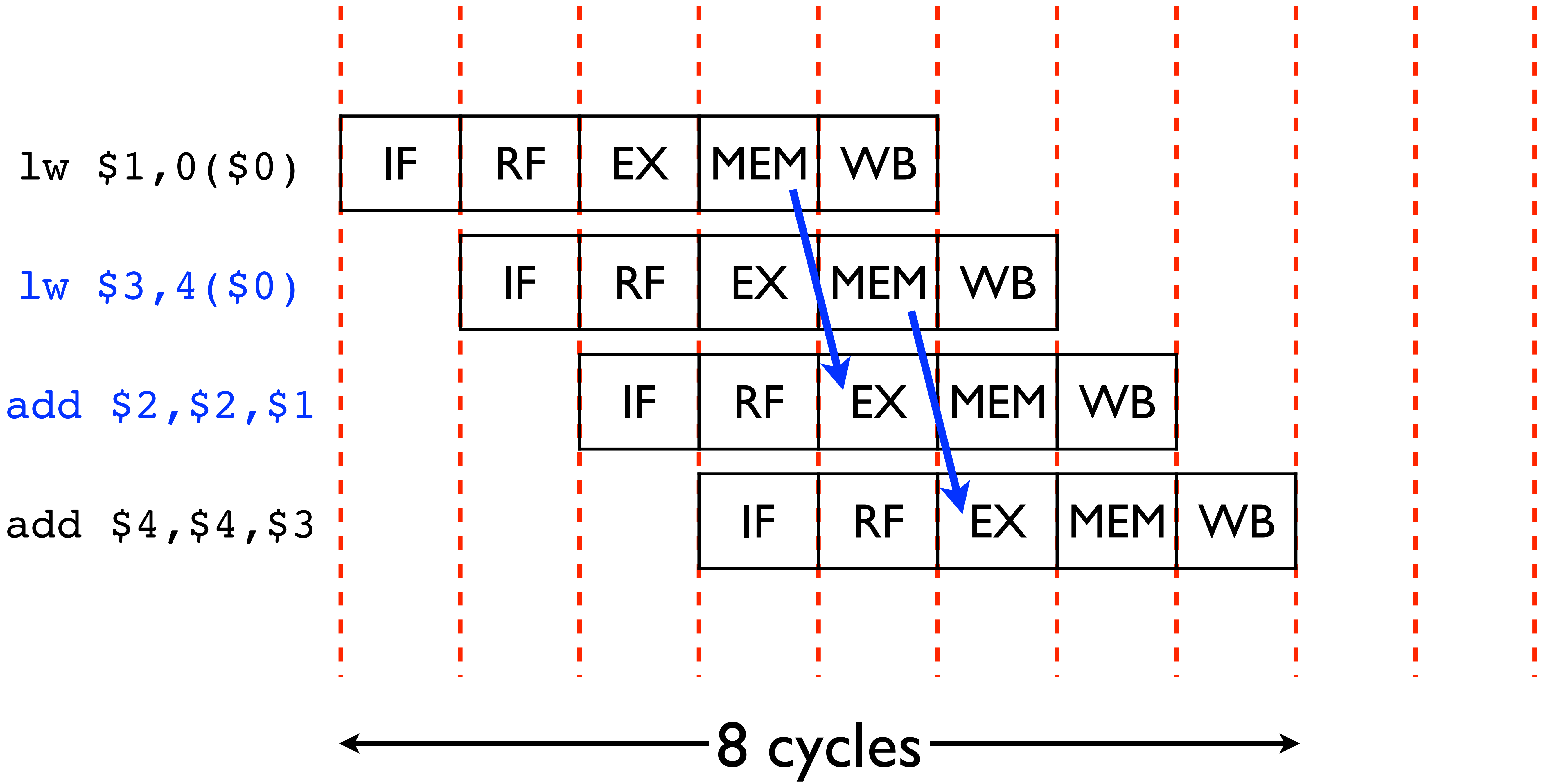
Instruction order

```
lw $1,0($0)
lw $3,4($0)
add $2,$2,$1
add $4,$4,$3
```

STALL FOR $1

STALL FOR $3

Instruction order



Instruction dependencies

We can only reorder target-code instructions if the meaning of the program is preserved.

We must therefore identify and respect the *data dependencies* which exist between instructions.

In particular, whenever an instruction is dependent upon an earlier one, the order of these two instructions must not be reversed.

Instruction dependencies

There are three kinds of data dependency:

- Read after write
- Write after read
- Write after write

Whenever one of these dependencies exists between two instructions, we cannot safely permute them.

Instruction dependencies

Read after write:
An instruction reads from a location
after an earlier instruction has written to it.

```
add $3,$1,$2
:
add $4,$4,$3
```

```
add $4,$4,$3
:
add $3,$1,$2
```

Reads old value

Instruction dependencies

Write after read:
An instruction writes to a location
after an earlier instruction has read from it.

```
add $4,$4,$3
⋮
add $3,$1,$2
```

```
add $3,$1,$2
⋮
add $4,$4,$3
```

Reads new value

Instruction dependencies

Write after write:
An instruction writes to a location
after an earlier instruction has written to it.

```
add $3,$1,$2          add $3,$4,$5
⋮                     ⋮
add $3,$4,$5          add $3,$1,$2
```

Writes old value

Instruction scheduling

We would like to reorder the instructions within each basic block in a way which

- preserves the dependencies between those instructions (and hence the correctness of the program), and
- achieves the minimum possible number of pipeline stalls.

We can address these two goals separately.

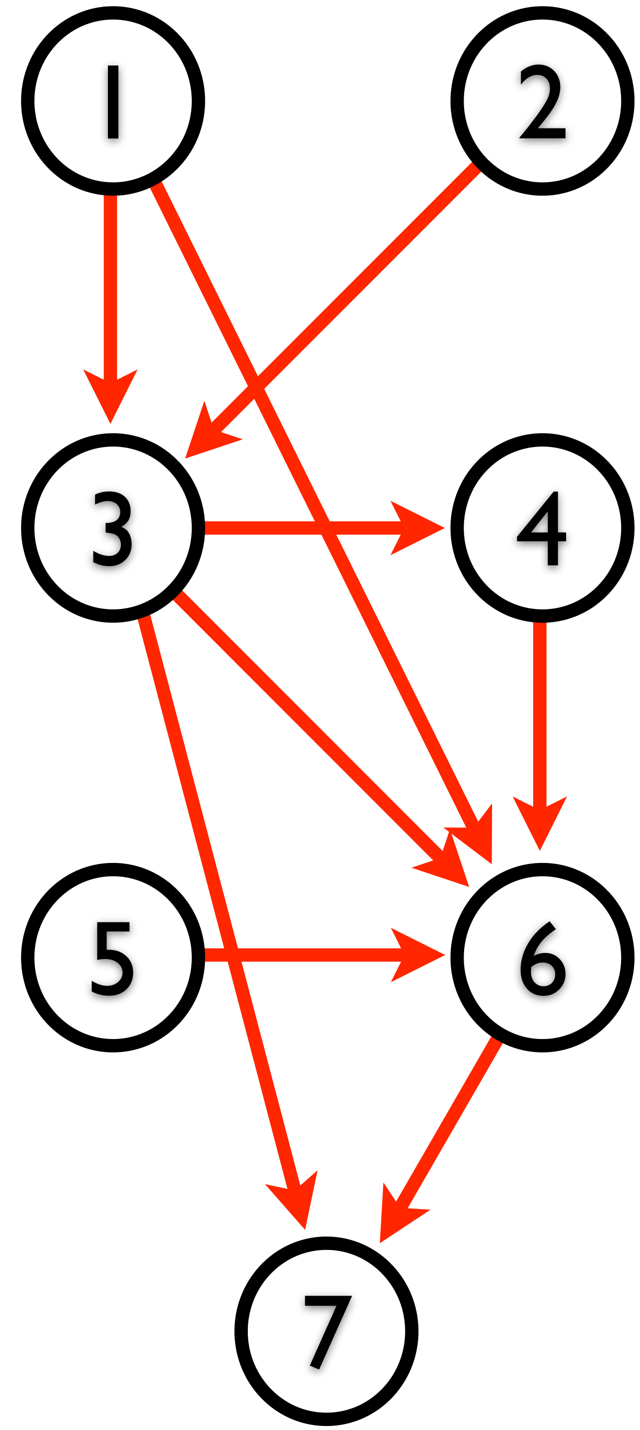
Preserving dependencies

Firstly, we can construct a *directed acyclic graph* (DAG) to represent the dependencies between instructions:

- For each instruction in the basic block, create a corresponding vertex in the graph.
- For each dependency between two instructions, create a corresponding edge in the graph.
  - This edge is *directed*: it goes from the earlier instruction to the later one.

Preserving dependencies

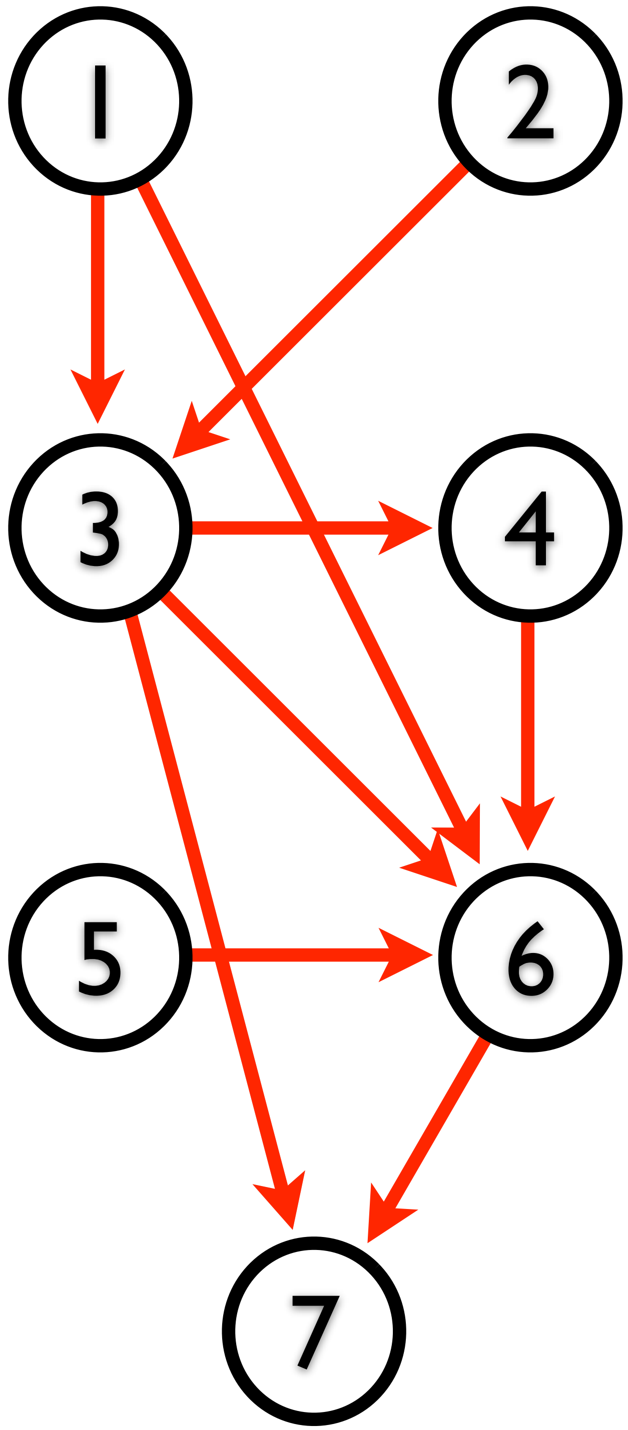
```
1 lw $1,0($0)
2 lw $2,4($0)
3 add $3,$1,$2
4 sw $3,12($0)
5 lw $4,8($0)
6 add $3,$1,$4
7 sw $3,16($0)
```



Preserving dependencies

Any topological sort of this DAG (i.e. any linear ordering of the vertices which keeps all the edges "pointing forwards") will maintain the dependencies and hence preserve the correctness of the program.

Preserving dependencies



1, 2, 3, 4, 5, 6, 7
2, 1, 3, 4, 5, 6, 7

1, 2, 3, 5, 4, 6, 7
1, 2, 5, 3, 4, 6, 7
1, 5, 2, 3, 4, 6, 7
5, 1, 2, 3, 4, 6, 7

2, 1, 3, 5, 4, 6, 7
2, 1, 5, 3, 4, 6, 7
2, 5, 1, 3, 4, 6, 7
5, 2, 1, 3, 4, 6, 7

Minimising stalls

Secondly, we want to choose an instruction order which causes the fewest possible pipeline stalls.

Unfortunately, this problem is (as usual) NP-complete and hence difficult to solve in a reasonable amount of time for realistic quantities of instructions.

However, we can devise some *static scheduling heuristics* to help guide us; we will hence choose a sensible and reasonably optimal instruction order, if not necessarily the absolute best one possible.

Minimising stalls

Each time we're emitting the next instruction,
we should try to choose one which:

- does not conflict with the previous emitted instruction
- is most likely to conflict if first of a pair (e.g. prefer `lw` to `add`)
- is as far away as possible (along paths in the DAG) from an instruction which can validly be scheduled last

Algorithm

Armed with the scheduling DAG and the static scheduling heuristics, we can now devise an algorithm to perform instruction scheduling.

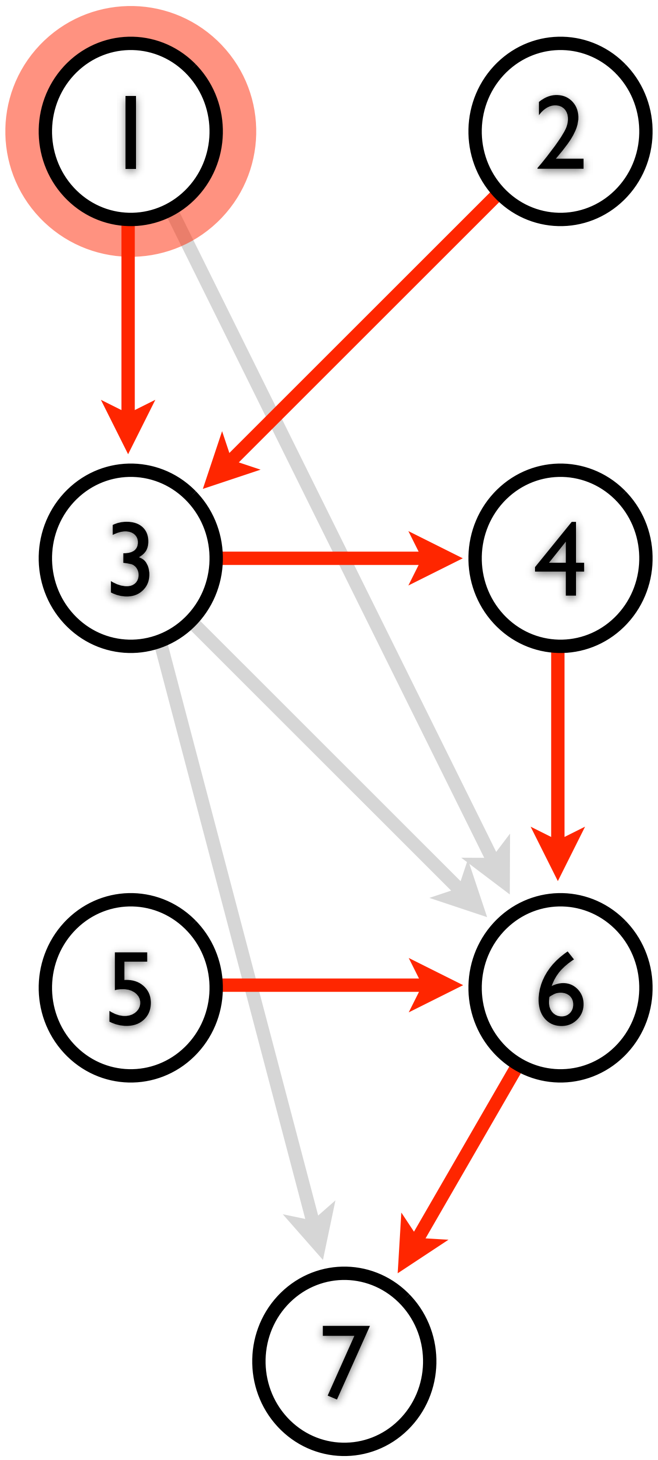
Algorithm

- Construct the scheduling DAG.
  - We can do this in $O(n^2)$ by scanning backwards through the basic block and adding edges as dependencies arise.
- Initialise the *candidate list* to contain the minimal elements of the DAG.

Algorithm

- While the candidate list is non-empty:
  - If possible, emit a candidate instruction satisfying all three of the static scheduling heuristics;
  - if no instruction satisfies all the heuristics, either emit NOP (on MIPS) or an instruction satisfying only the last two heuristics (on SPARC).
  - Remove the instruction from the DAG and insert the newly minimal elements into the candidate list.

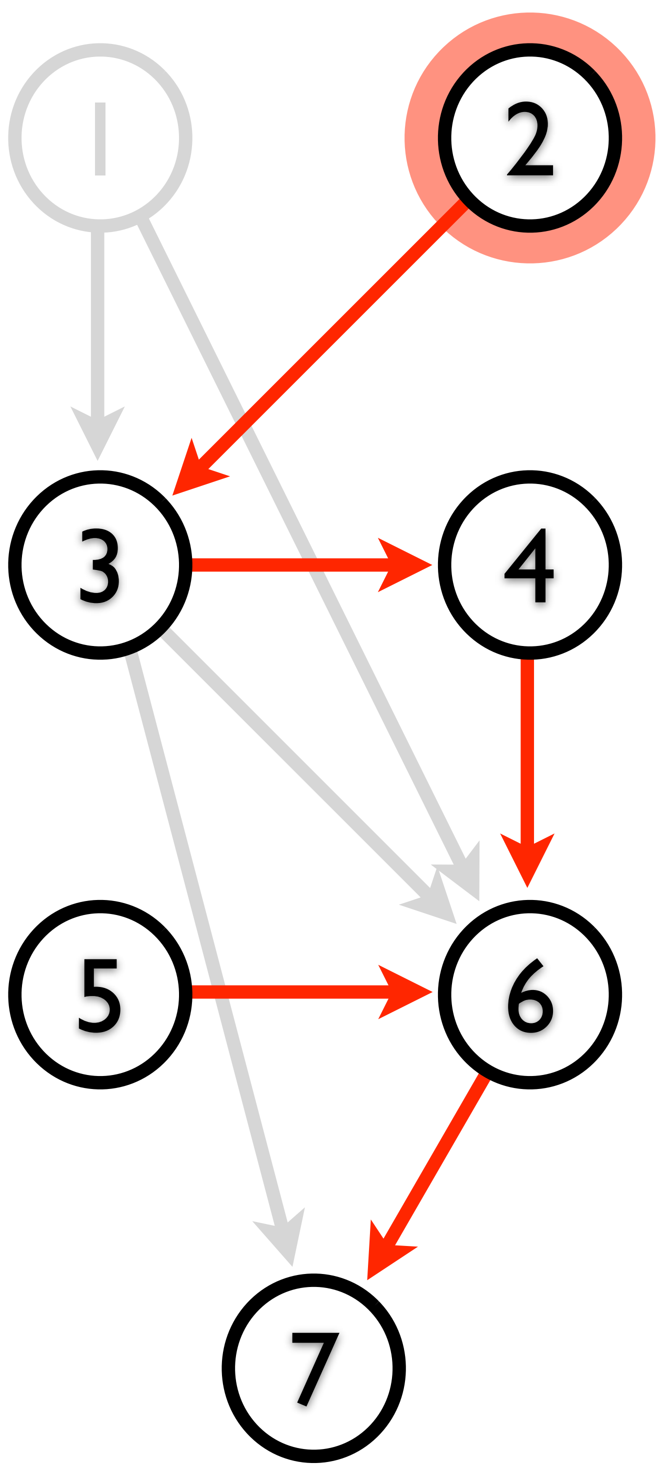
Algorithm



Candidates:
{ 1, 2, 5 }

1 `lw $1,0($0)`

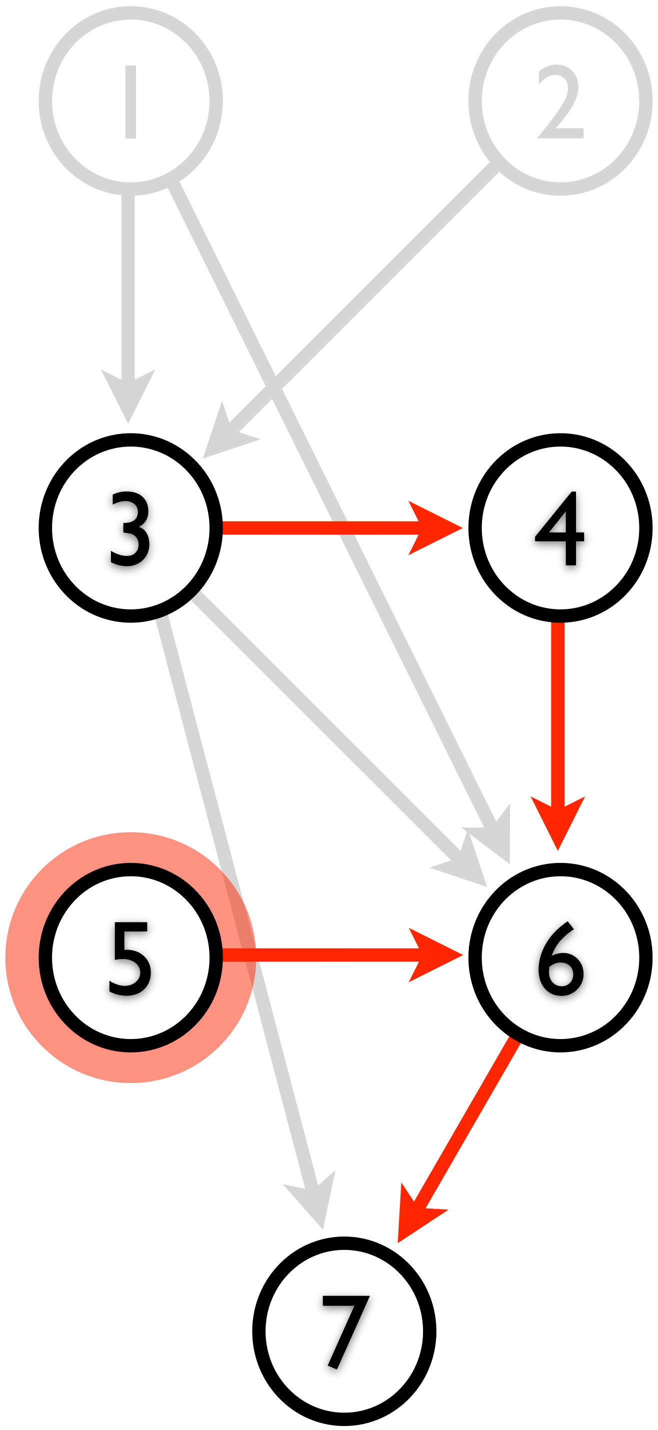
Algorithm



Candidates:
{ 2, 5 }

1 `lw $1,0($0)`
2 `lw $2,4($0)`

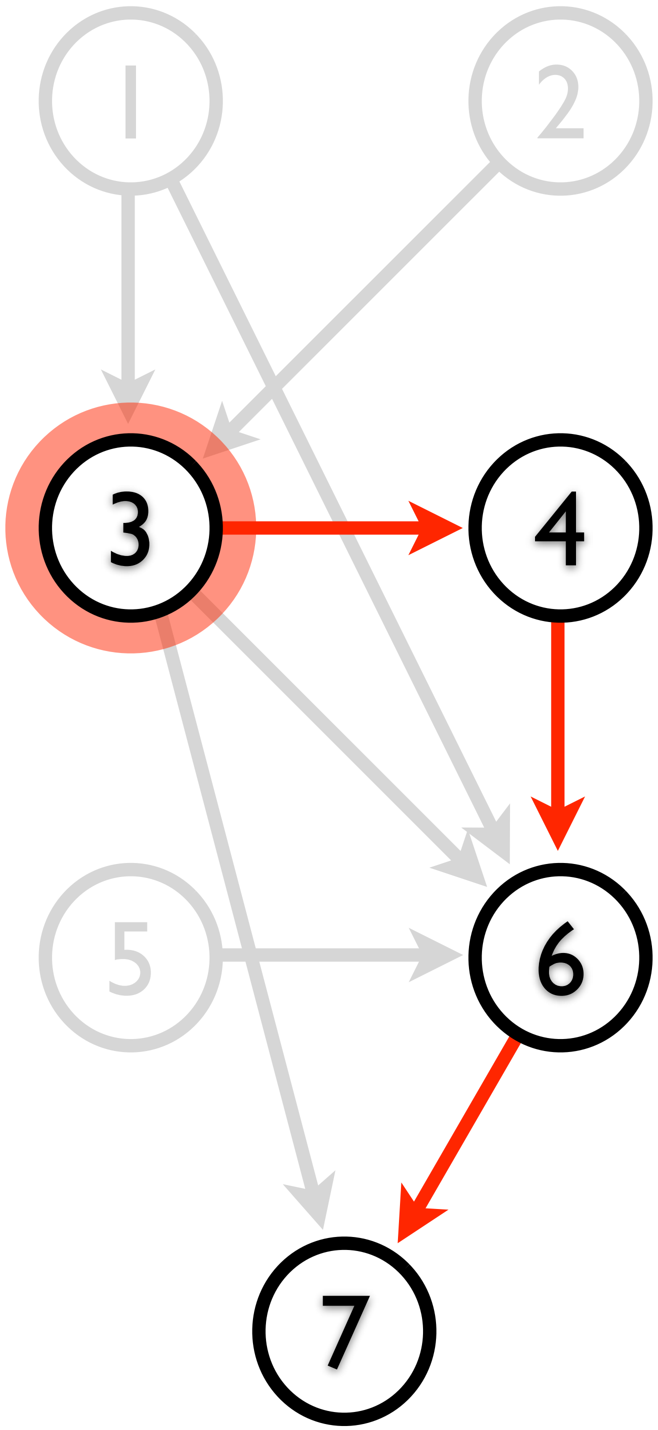
Algorithm



Candidates:
{ 3, 5 }

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
```

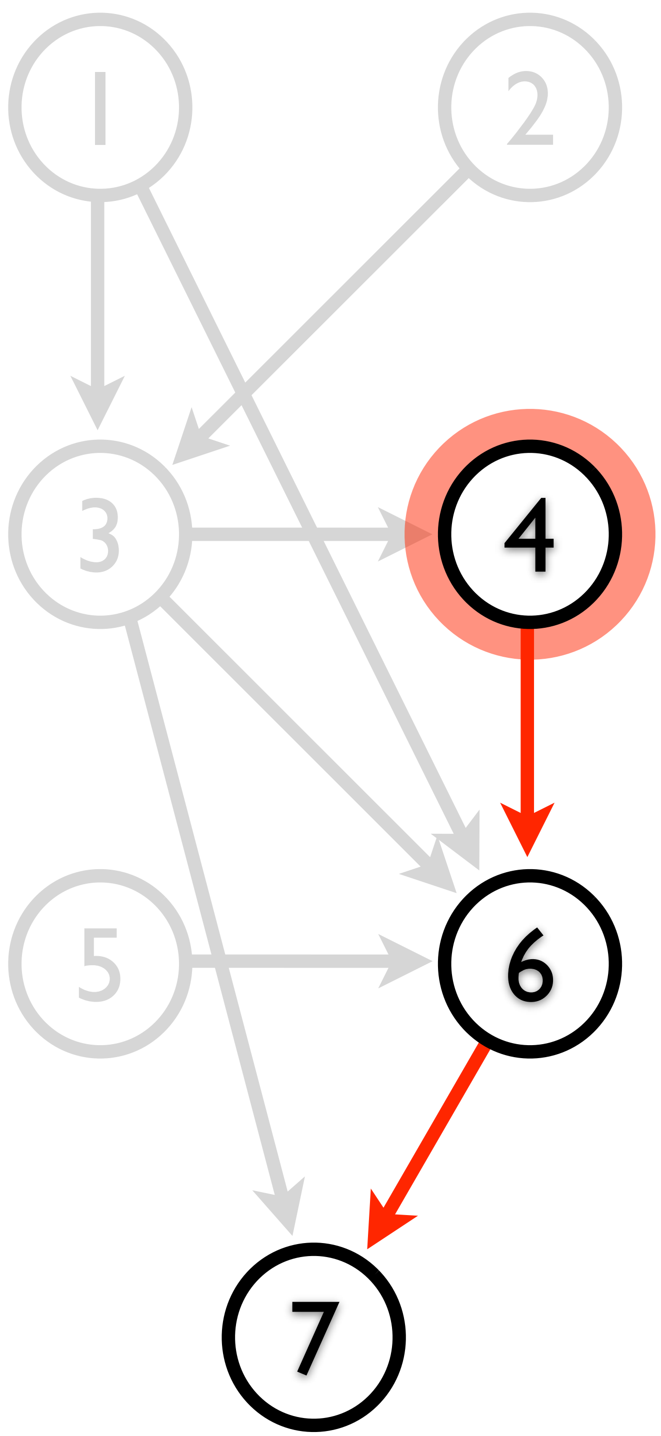
Algorithm



Candidates:
{ 3 }

1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
3 add $3,$1,$2

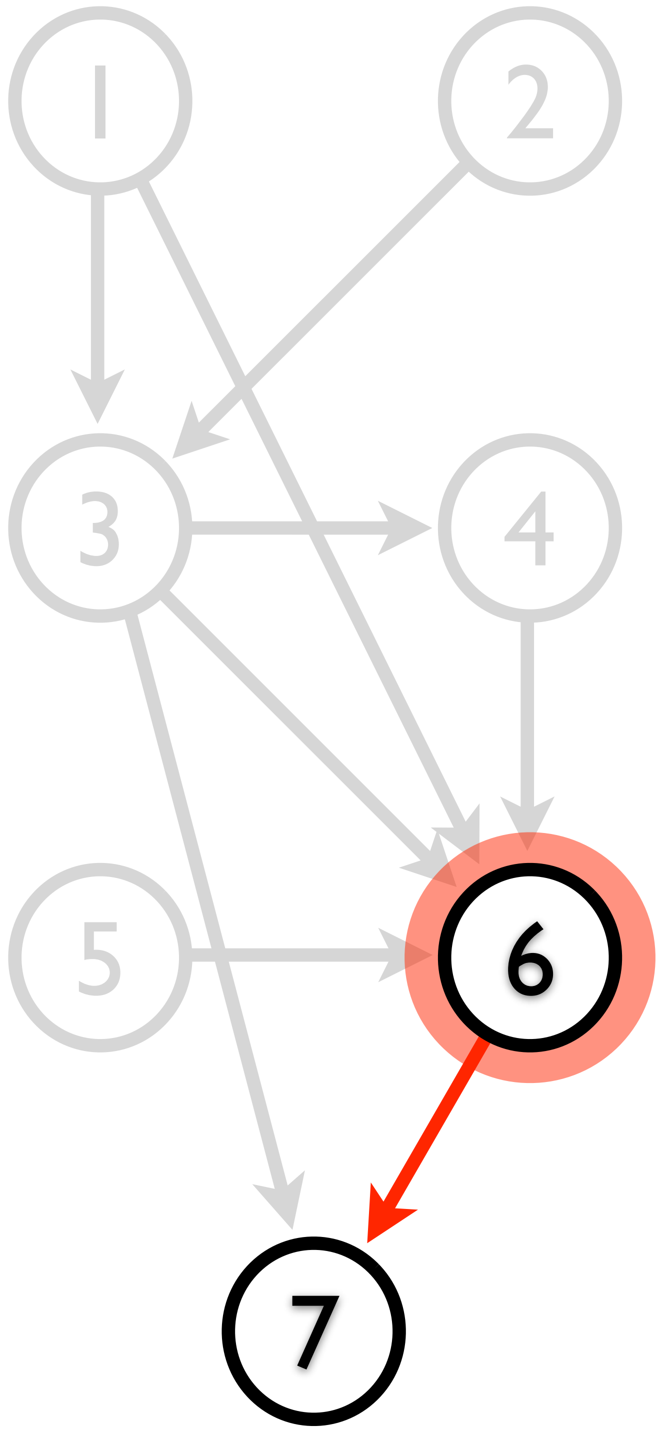
Algorithm



Candidates:
{4}

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
3 add $3,$1,$2
4 sw $3,12($0)
```

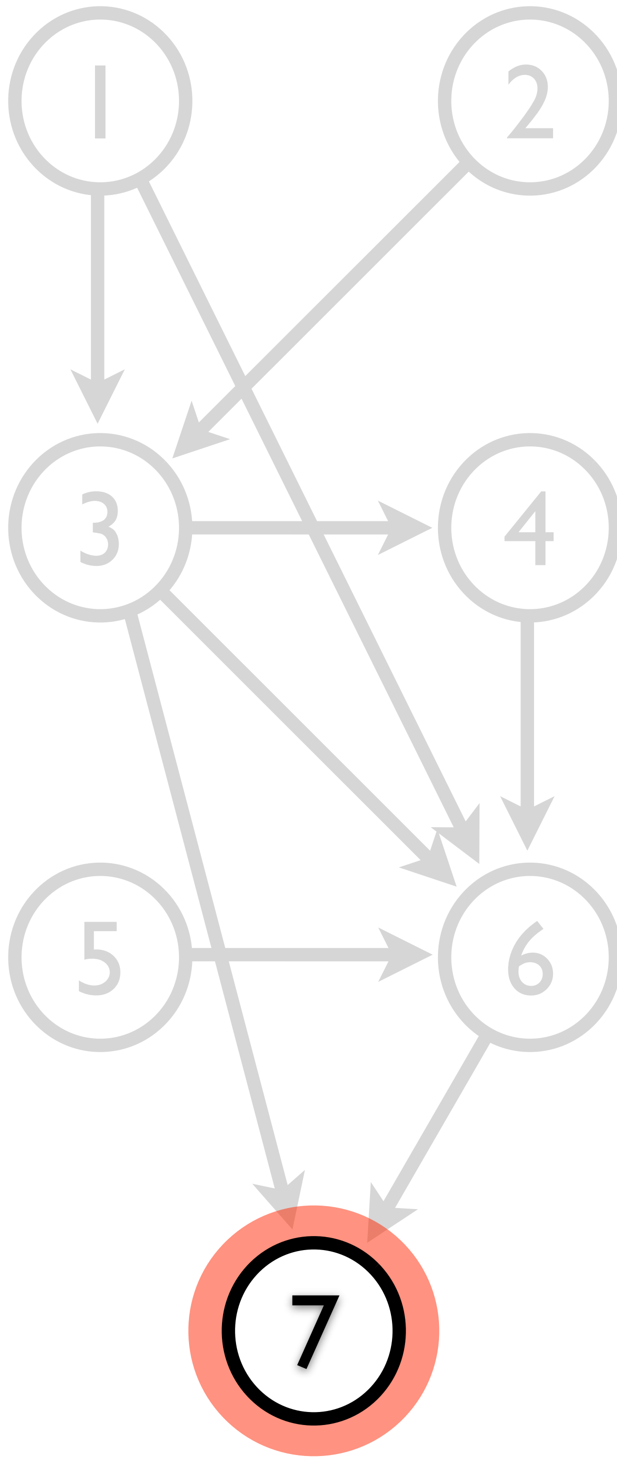
Algorithm



Candidates:
{ 6 }

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
3 add $3,$1,$2
4 sw $3,12($0)
6 add $3,$1,$4
```

Algorithm



Candidates:

{7}

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
3 add $3,$1,$2
4 sw $3,12($0)
6 add $3,$1,$4
7 sw $3,16($0)
```

Algorithm

Original code:

```
1 lw $1,0($0)
2 lw $2,4($0)
3 add $3,$1,$2
4 sw $3,12($0)
5 lw $4,8($0)
6 add $3,$1,$4
7 sw $3,16($0)
```

2 stalls

13 cycles

Scheduled code:

```
1 lw $1,0($0)
2 lw $2,4($0)
5 lw $4,8($0)
3 add $3,$1,$2
4 sw $3,12($0)
6 add $3,$1,$4
7 sw $3,16($0)
```

no stalls

11 cycles

Dynamic scheduling

Instruction scheduling is important for getting the best performance out of a processor; if the compiler does a bad job (or doesn't even try), performance will suffer.

As a result, modern processors have dedicated hardware for performing instruction scheduling dynamically as the code is executing.

This may appear to render compile-time scheduling rather redundant.

Dynamic scheduling

But:

- This is still compiler technology, just increasingly being implemented in hardware.
- Somebody — now hardware designers — must still understand the principles.
- Embedded processors may not do dynamic scheduling, or may have the option to turn the feature off completely to save power, so it's still worth doing at compile-time.

Summary

- Instruction pipelines allow a processor to work on executing several instructions at once
- Pipeline hazards cause stalls and impede optimal throughput, even when bypassing is used
- Instructions may be reordered to avoid stalls
- Dependencies between instructions limit reordering
- Static scheduling heuristics may be used to achieve near-optimal scheduling with an $O(n^2)$ algorithm

Lecture 15
Register allocation vs instruction scheduling, reverse engineering

Allocation vs. scheduling

We have seen why register allocation is a useful compilation phase: when done well, it can make the best use of available registers and hence reduce the number of spills to memory.

Unfortunately, by maximising the utilisation of architectural registers, register allocation makes instruction scheduling significantly more difficult.

Allocation vs. scheduling

```
*x := *a;
*y := *b;
```

lexing, parsing, translation

```
LDR v36,v32
STR v36,v33
LDR v37,v34
STR v37,v35
```

compilation

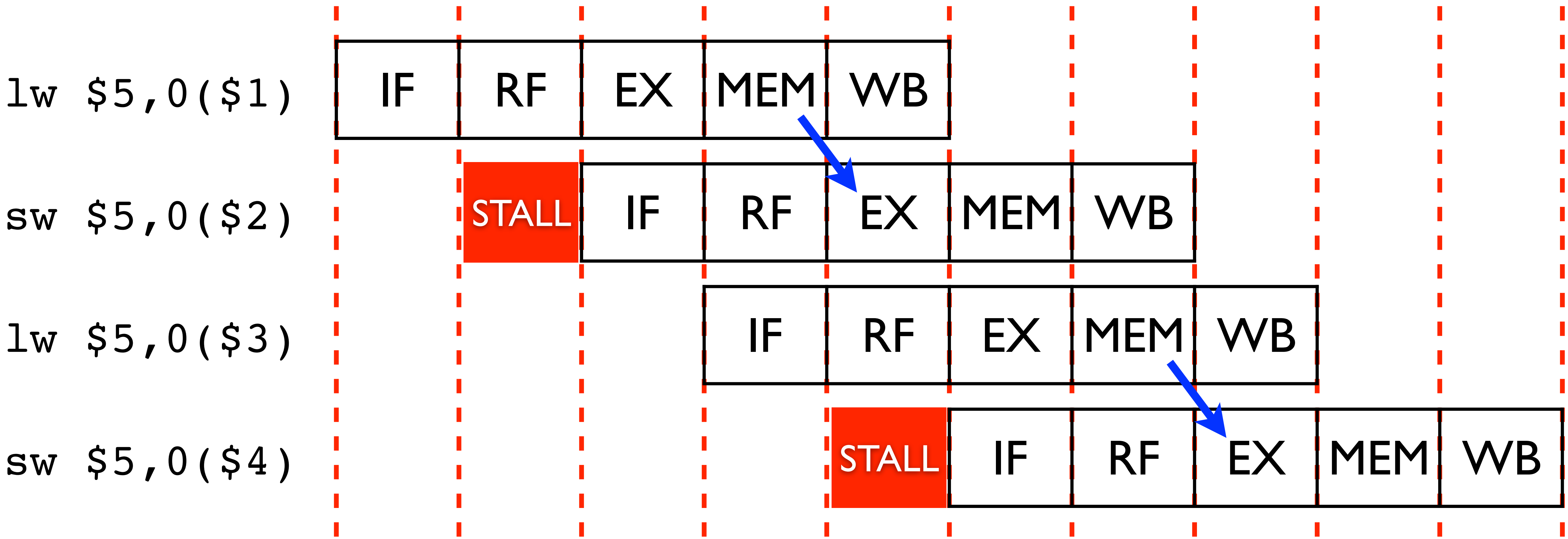
register allocation

```
LDR v5,v1
STR v5,v2
LDR v5,v3
STR v5,v4
```

code generation

```
lw $5,0($1)
sw $5,0($2)
lw $5,0($3)
sw $5,0($4)
```

Allocation vs. scheduling

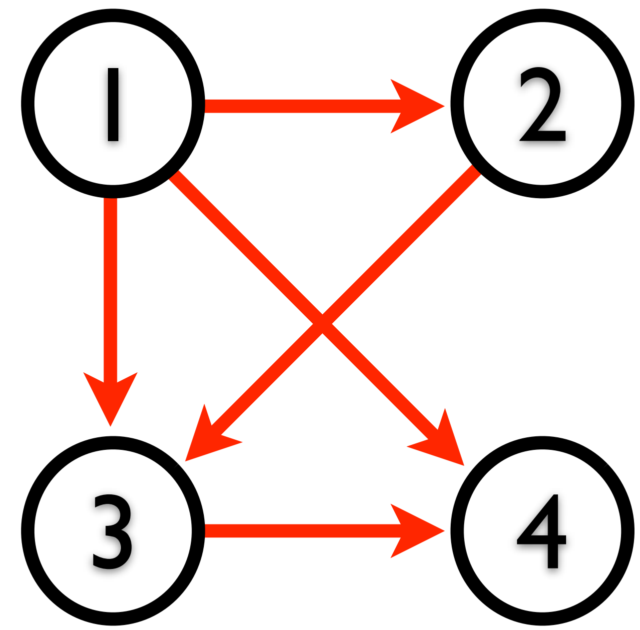



```
lw $5,0($1)
sw $5,0($2)
lw $5,0($3)
sw $5,0($4)
```

This schedule of instructions produces two pipeline stalls (or requires two NOPs).

Allocation vs. scheduling

Can we reorder them to avoid stalls?




```
1 lw $5,0($1)
2 sw $5,0($2)
3 lw $5,0($3)
4 sw $5,0($4)
```

1, 2, 3, 4

No: this is the *only* correct schedule for these instructions.

Allocation vs. scheduling

We might have done better if
register $5 wasn't so heavily used.

If only our register allocation had
been less aggressive!

Allocation vs. scheduling

```
*x := *a;
*y := *b;
```

lexing, parsing, translation →

```
LDR v36,v32
STR v36,v33
LDR v37,v34
STR v37,v35
```

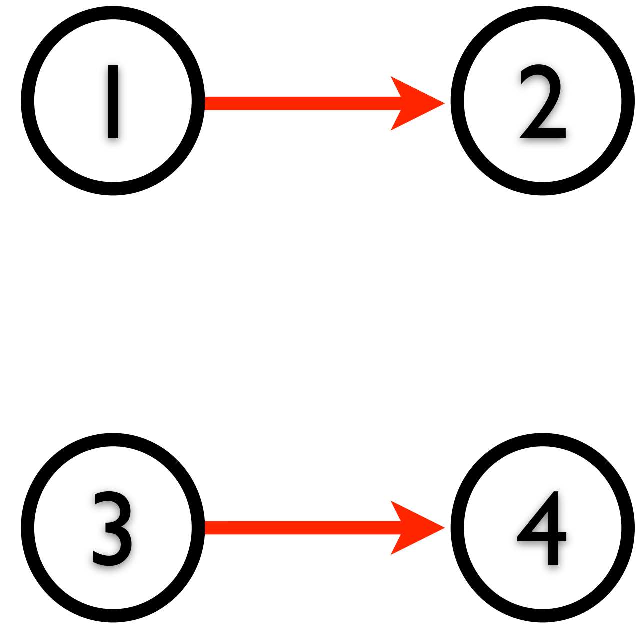
↓ register allocation

```
LDR v5,v1
STR v5,v2
LDR v6,v3
STR v6,v4
```

← code generation

```
lw $5,0($1)
sw $5,0($2)
lw $6,0($3)
sw $6,0($4)
```

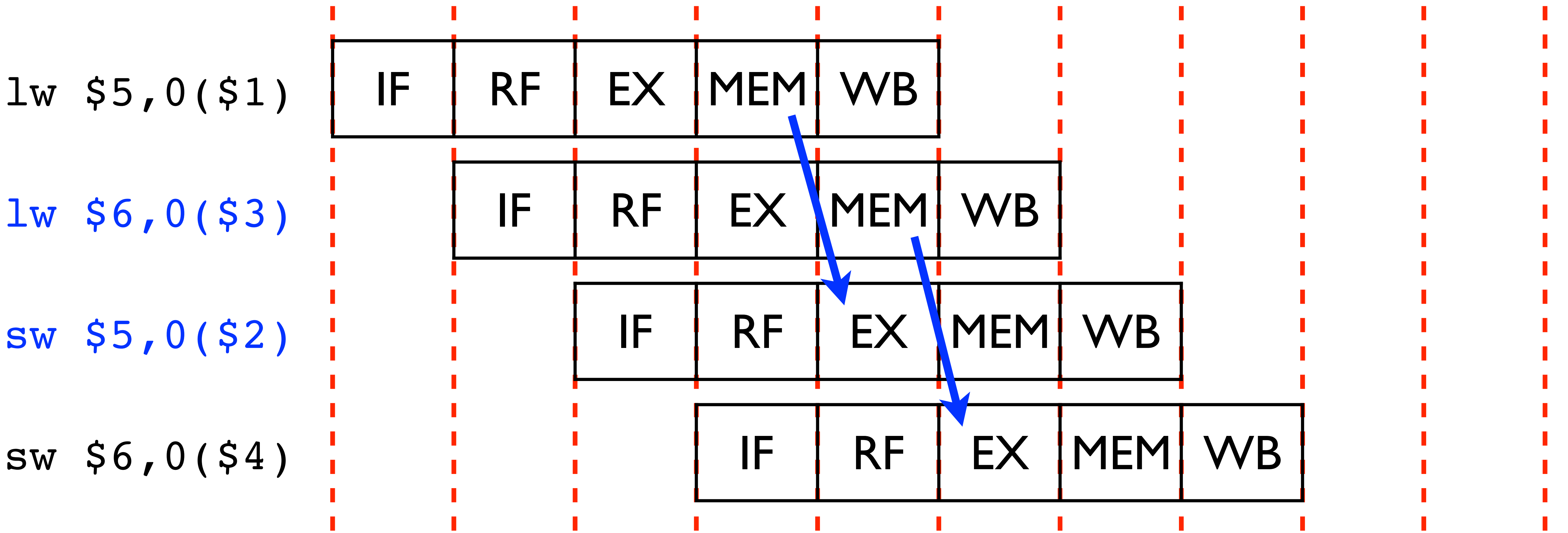
Allocation vs. scheduling




```
1 lw $5,0($1)
2 sw $5,0($2)
3 lw $6,0($3)
4 sw $6,0($4)
```

1, 2, 3, 4
1, 3, 2, 4
3, 1, 2, 4
1, 3, 4, 2
3, 1, 4, 2
3, 4, 1, 2

Allocation vs. scheduling



This schedule of the new instructions produces no stalls.

```
lw $5,0($1)
lw $6,0($3)
sw $5,0($2)
sw $6,0($4)
```

Allocation vs. scheduling

There is clearly antagonism between register allocation and instruction scheduling: one reduces spills by using *fewer* registers, but the other can better reduce stalls when *more* registers are used.

This is related to the *phase-order problem* discussed earlier in the course, in which we would like to defer optimisation decisions until we know how they will affect later phases in the compiler.

It’s not clear how best to resolve the problem.

Allocation vs. scheduling

One option is to try to allocate architectural registers cyclically rather than re-using them at the earliest opportunity.

It is this eager re-use of registers that causes stalls, so if we can avoid it — and still not spill any virtual registers to memory — we will have a better chance of producing an efficient program.

Allocation vs. scheduling

In practise this means that, when doing register allocation by colouring for a basic block, we should

- satisfy all of the important constraints as usual (i.e. clash graph, preference graph),
- see how many spare architectural registers we still have left over, and then
- for each unallocated virtual register, try to choose an architectural register distinct from all others allocated in the same basic block.

Allocation vs. scheduling

So, if we are less zealous about reusing registers, this should hopefully result in a better instruction schedule while not incurring any extra spills.

In general, however, it is rather difficult to predict exactly how our allocation and scheduling phases will interact, and this particular solution is quite ad hoc.

Some (fairly old) research (e.g. CRAIG system in 1995, Touati's PhD thesis in 2002) has improved the situation.

Allocation vs. scheduling

The same problem also shows up in dynamic scheduling done by hardware.

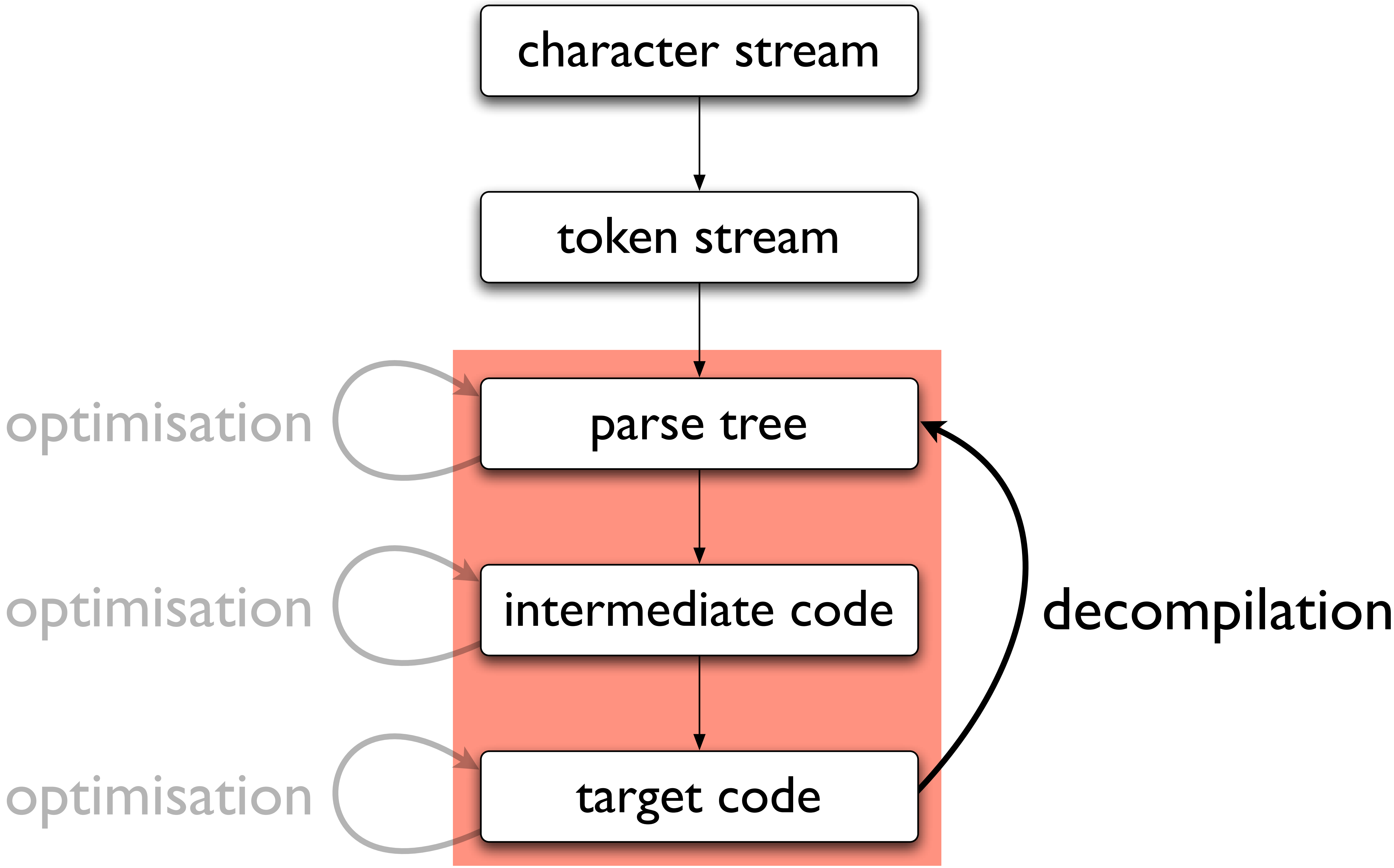
Executable x86 code, for example, has lots of register reuse because of the small number of architectural registers available.

Modern machines cope by actually having more registers than advertised; it does dynamic recolouring using this larger register set, which then enables more effective scheduling.

Part D

Decompilation and reverse engineering

Decompilation



Motivation

The job of an optimising compiler is to turn human-readable source code into efficient, executable target code.

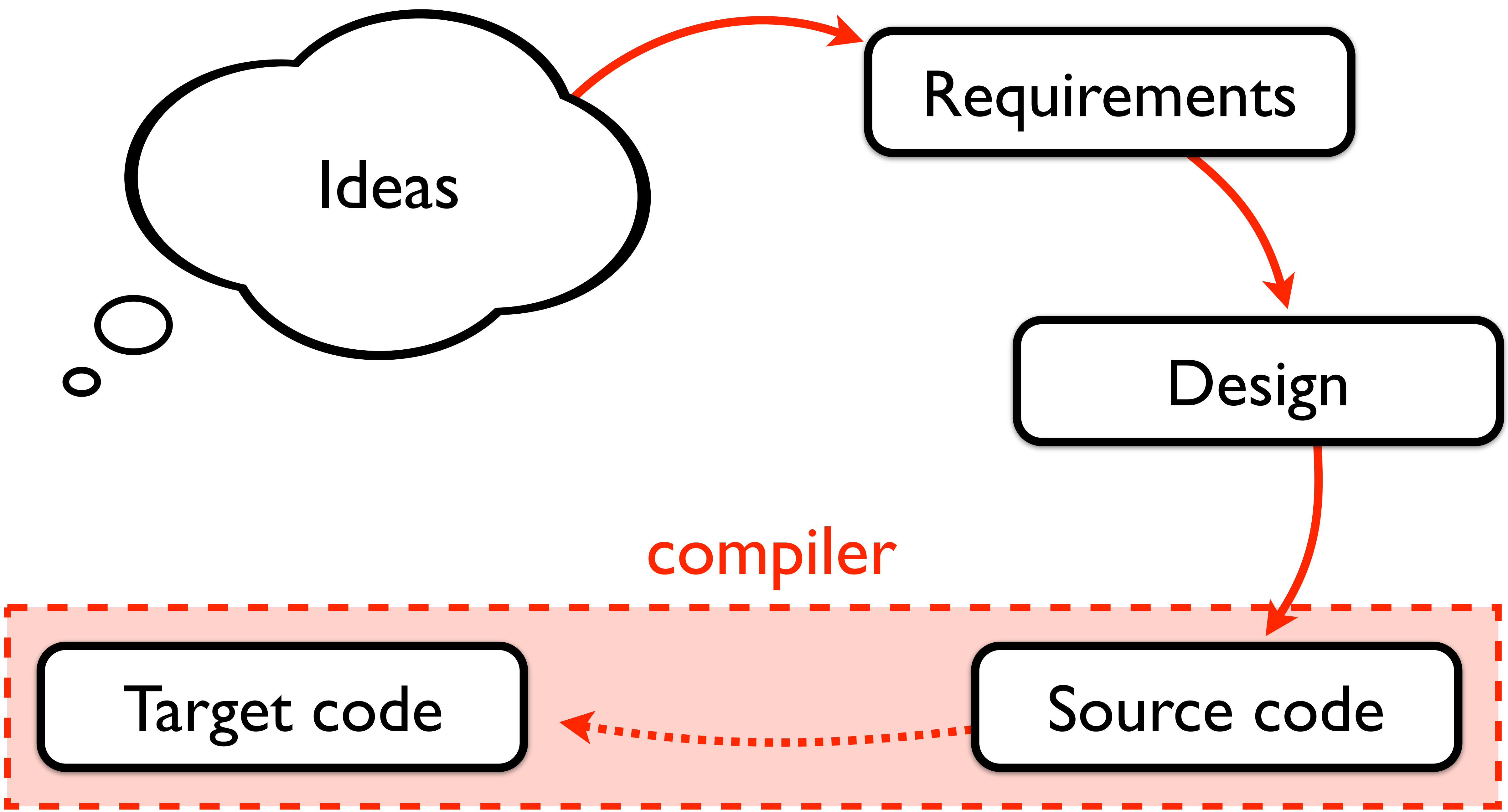
Although executable code is useful, software is most valuable in source code form, where it can be easily read and modified.

The source code corresponding to an executable is not always available — it may be lost, missing or secret — so we might want to use *decompilation* to recover it.

Reverse engineering

In general terms, engineering is a process which decreases the level of abstraction of some system.

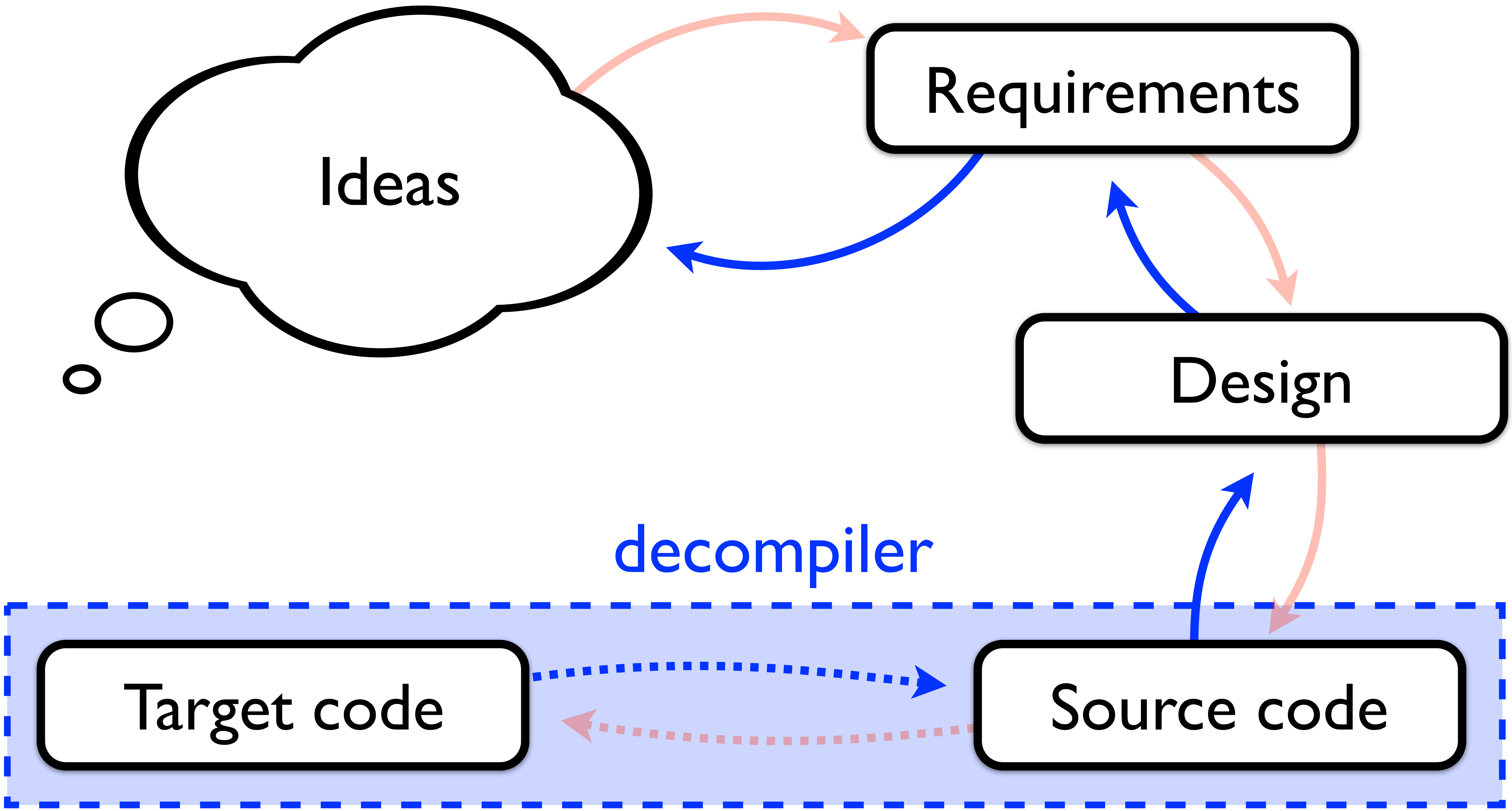
Reverse engineering



Reverse engineering

In contrast, *reverse engineering* is the process of *increasing* the level of abstraction of some system, making it less suitable for implementation but more suitable for comprehension and modification.

Reverse engineering



Legality and ethics

It is quite feasible to decompile and otherwise reverse-engineer most software.

So if reverse-engineering software is technologically possible, is there any *ethical* barrier to doing it?

In particular, when is it *legal* to do so?

Legality and ethics

Companies and individuals responsible for creating software generally consider source code to be their confidential intellectual property; they will not make it available, and they do not want you to reconstruct it.

(There are some well-known exceptions.)

Usually this desire is expressed via an end-user license agreement, either as part of a shrink-wrapped software package or as an agreement to be made at installation time ("click-wrap").

Legality and ethics

However, the European Union Software Directive of 1991 (91/250/EC) says:

Article 4 Restricted Acts

Subject to the provisions of Articles 5 and 6, the exclusive rights of the rightholder within the meaning of Article 2, shall include the right to do or to authorize:

(a) the permanent or temporary reproduction of a computer program by any means and in any form, in part or in whole. Insofar as loading, displaying, running, transmission or storage of the computer program necessitate such reproduction, such acts shall be subject to authorization by the rightholder;

(b) the translation, adaptation, arrangement and any other alteration of a computer program and the reproduction of the results thereof, without prejudice to the rights of the person who alters the program;

(c) any form of distribution to the public, including the rental, of the original computer program or of copies thereof. The first sale in the Community of a copy of a program by the rightholder or with his consent shall exhaust the distribution right within the Community of that copy, with the exception of the right to control further rental of the program or a copy thereof.

Article 5 Exceptions to the restricted acts

1. In the absence of specific contractual provisions, the acts referred to in Article 4 (a) and (b) shall not require authorization by the rightholder where they are necessary for the use of the computer program by the lawful acquirer in accordance with its intended purpose, including for error correction.

2. The making of a back-up copy by a person having a right to use the computer program may not be prevented by contract insofar as it is necessary for that use.

3. The person having a right to use a copy of a computer program shall be entitled, without the authorization of the rightholder, to observe, study or test the functioning of the program in order to determine the ideas and principles which underlie any element of the program if he does so while performing any of the acts of loading, displaying, running, transmitting or storing the program which he is entitled to do.

Article 6 Decompilation

1. The authorization of the rightholder shall not be required where reproduction of the code and translation of its form within the meaning of Article 4 (a) and (b) are indispensable to obtain the information necessary to achieve the interoperability of an independently created computer program with other programs, provided that the following conditions are met:

(a) these acts are performed by the licensee or by another person having a right to use a copy of a program, or on their behalf by a person authorized to to so;

(b) the information necessary to achieve interoperability has not previously been readily available to the persons referred to in subparagraph (a); and (c) these acts are confined to the parts of the original program which are necessary to achieve interoperability.

2. The provisions of paragraph 1 shall not permit the information obtained through its application:

(a) to be used for goals other than to achieve the interoperability of the independently created computer program;

(b) to be given to others, except when necessary for the interoperability of the independently created computer program; or (c) to be used for the development, production or marketing of a computer program substantially similar in its expression, or for any other act which infringes copyright.

Legality and ethics

"The authorization of the rightholder shall not be required where [...] translation [of a program is] necessary to achieve the interoperability of [that program] with other programs, provided [...] these acts are performed by [a] person having a right to use a copy of the program"

Legality and ethics

The more recent European Union Copyright Directive of 2001 (2001/29/EC, aka "EUCD") is the EU's implementation of the 1996 WIPO Copyright Treaty.

It is again concerned with the ownership rights of technological IP, but Recital 50 states that:
"[this] legal protection does not affect the specific provisions [of the EUSD]. In particular, it should not apply to [...] computer programs [and shouldn't] prevent [...] the use of any means of circumventing a technological measure [allowed by the EUSD]."

Legality and ethics

And the USA has its own implementation of the WIPO Copyright Treaty: the Digital Millennium Copyright Act of 1998 (DMCA), which contains a similar exception for reverse engineering:

"This exception permits circumvention [...] for the sole purpose of identifying and analyzing elements of the program necessary to achieve interoperability with other programs, to the extent that such acts are permitted under copyright law."

Legality and ethics

Predictably enough, the interaction between the EUSD, EUCD and DMCA is complex and unclear, particularly at the increasingly-blurred interfaces between geographical jurisdictions (cf. Dmitry Sklyarov), and between software and other forms of technology (cf. Jon Johansen).

Get a lawyer.

Clean room design

Despite the complexity of legislation, it is possible to do useful reverse-engineering without breaking the law.

In 1982, Compaq produced the first fully IBM-compatible personal computer by using *clean room design* (aka "Chinese wall technique") to reverse-engineer the proprietary IBM BIOS.

This technique is effective in legally circumventing copyrights and trade secrets, although not patents.

Summary

- Register allocation makes scheduling harder by creating extra dependencies between instructions
- Less aggressive register allocation may be desirable
- Some processors allocate and schedule dynamically
- Reverse engineering is used to extract source code and specifications from executable code
- Existing copyright legislation may permit limited reverse engineering for interoperability purposes

Lecture 16
Decompilation

Why decompilation?

This course is ostensibly about Optimising Compilers.

It is really about *program analysis and transformation*.

Decompilation is achieved through analysis and transformation of target code; the transformations just work in the opposite direction.

The decompilation problem

Even simple compilation discards a lot of information:

- Comments
- Function and variable names
- Structured control flow
- Type information

The decompilation problem

Optimising compilation is even worse:

- Dead code and common subexpressions are eliminated
- Algebraic expressions are rewritten
- Code and data are inlined; loops are unrolled
- Unrelated local variables are allocated to the same architectural register
- Instructions are reordered by code motion optimisations and instruction scheduling

The decompilation problem

Some of this information is never going to be automatically recoverable (e.g. comments, variable names); some of it we may be able to partially recover if our techniques are sophisticated enough.

Compilation is *not injective*. Many different source programs may result in the same compiled code, so the best we can do is to pick a reasonable *representative* source program.

Intermediate code

It is relatively straightforward to extract a flowgraph from an assembler program.

Basic blocks are located in the same way as during forward compilation; we must simply deal with the semantics of the target instructions rather than our intermediate 3-address code.

Intermediate code

For many purposes (e.g. simplicity, retargetability) it might be beneficial to convert the target instructions back into 3-address code when storing it into the flowgraph.

This presents its own problems: for example, many architectures include instructions which test or set condition flags in a status register, so it may be necessary to laboriously reconstruct this behaviour with extra virtual registers and then use dead-code elimination to remove all unnecessary instructions thus generated.

Control reconstruction

A compiler apparently destroys the high-level control structure which is evident in a program's source code.

After building a flowgraph during decompilation, we can recover some of this structure by attempting to match *intervals* of the flowgraph against some fixed set of familiar syntactic forms from our high-level language.

Finding loops

Any structured loops from the original program will have been compiled into tests and branches; they will look like arbitrary (“spaghetti”) control flow.

In order to recover the high-level structure of these loops, we must use *dominance*.

Dominance

In a flowgraph, we say a node *m dominates* another node *n* if control must go through *m* before it can reach *n*.

A node *m strictly dominates* another node *n* if *m* dominates *n* and $m \neq n$.

The *immediate dominator* of a node *n* is the unique node that strictly dominates *n* but doesn't dominate any other strict dominator of *n*.

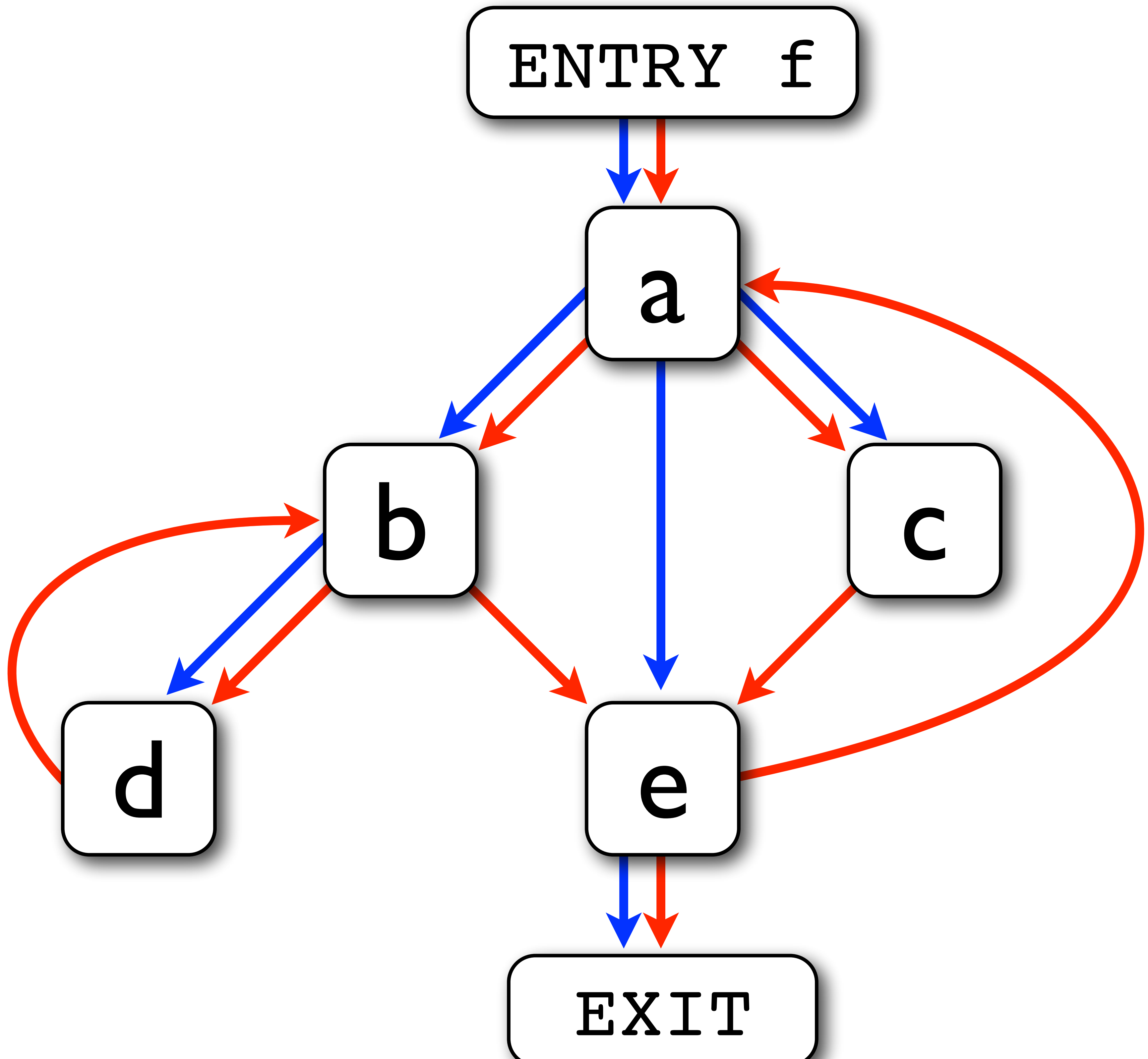
Dominance

A node *n* is in the *dominance frontier* of a node *m* if *m* does not strictly dominate *n* but does dominate an immediate predecessor of *n*.

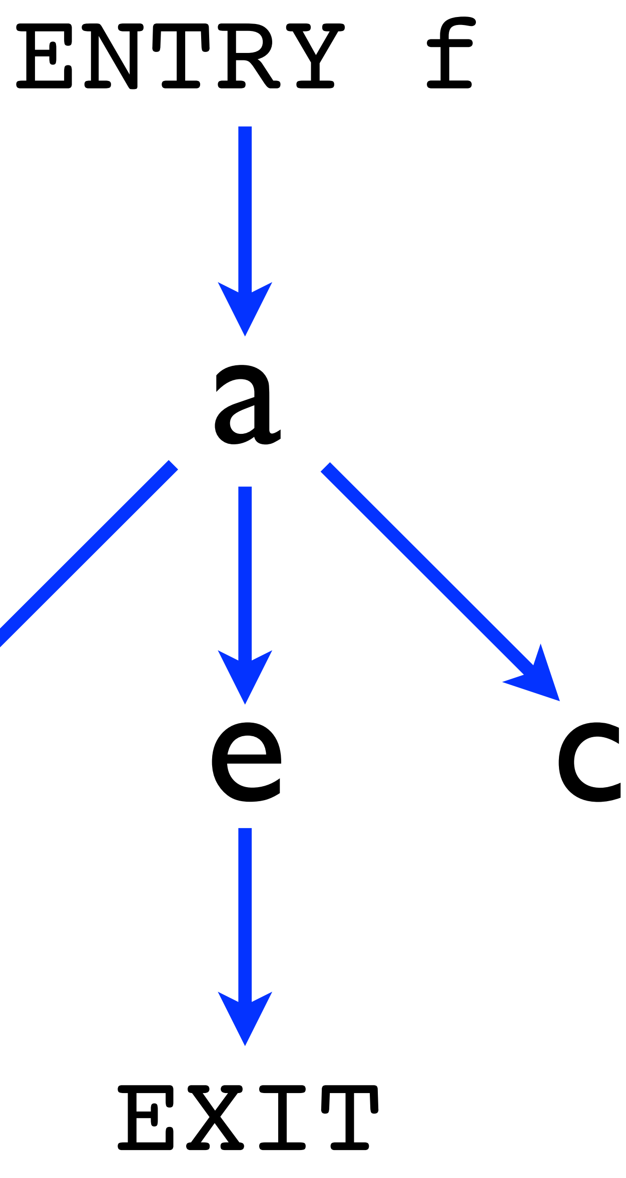
Intuitively this is the set of nodes where *m*'s dominance stops.

We can represent this dominance relation with a *dominance tree* in which each edge connects a node with its immediate dominator.

Dominance



Dominance

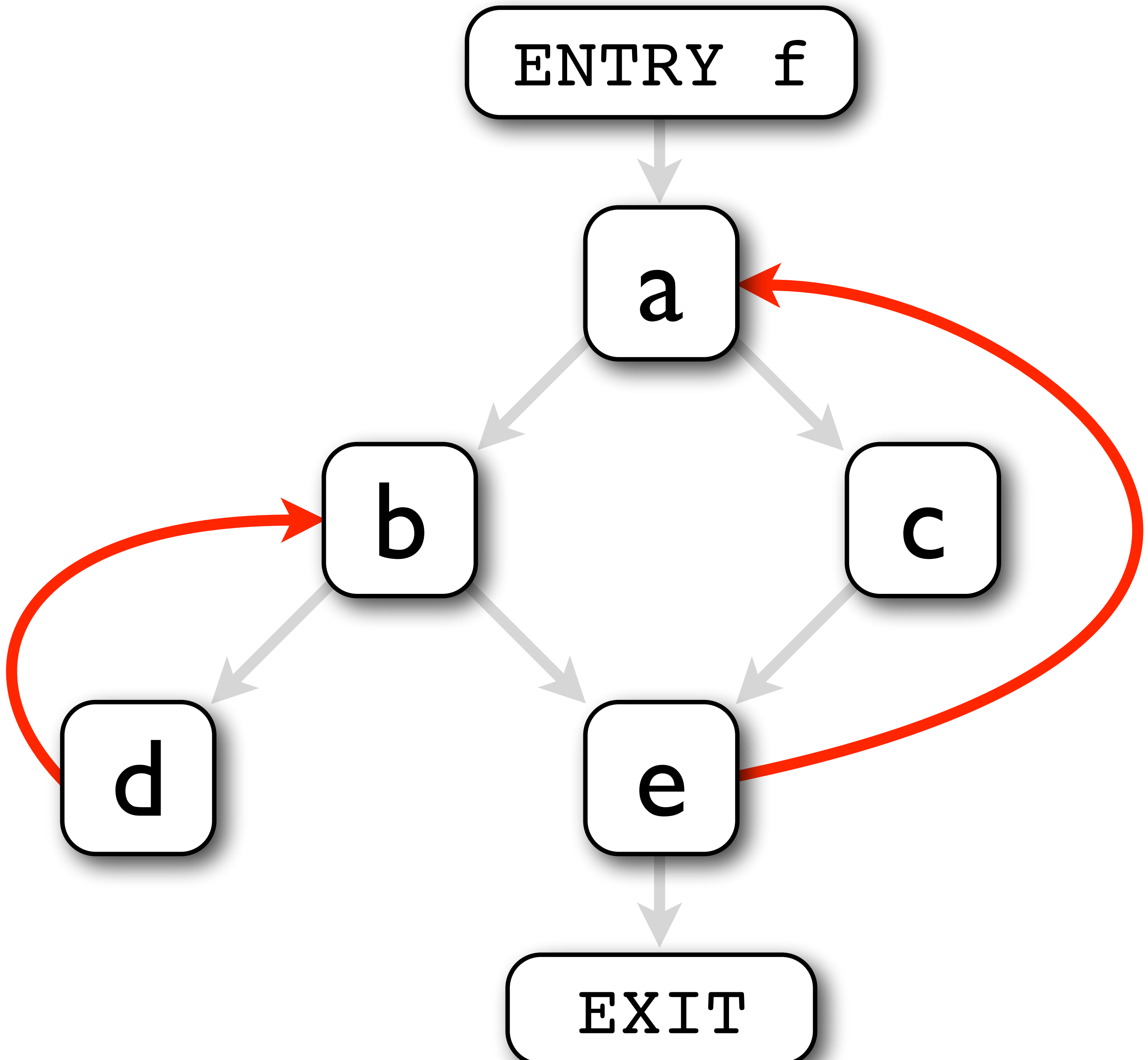


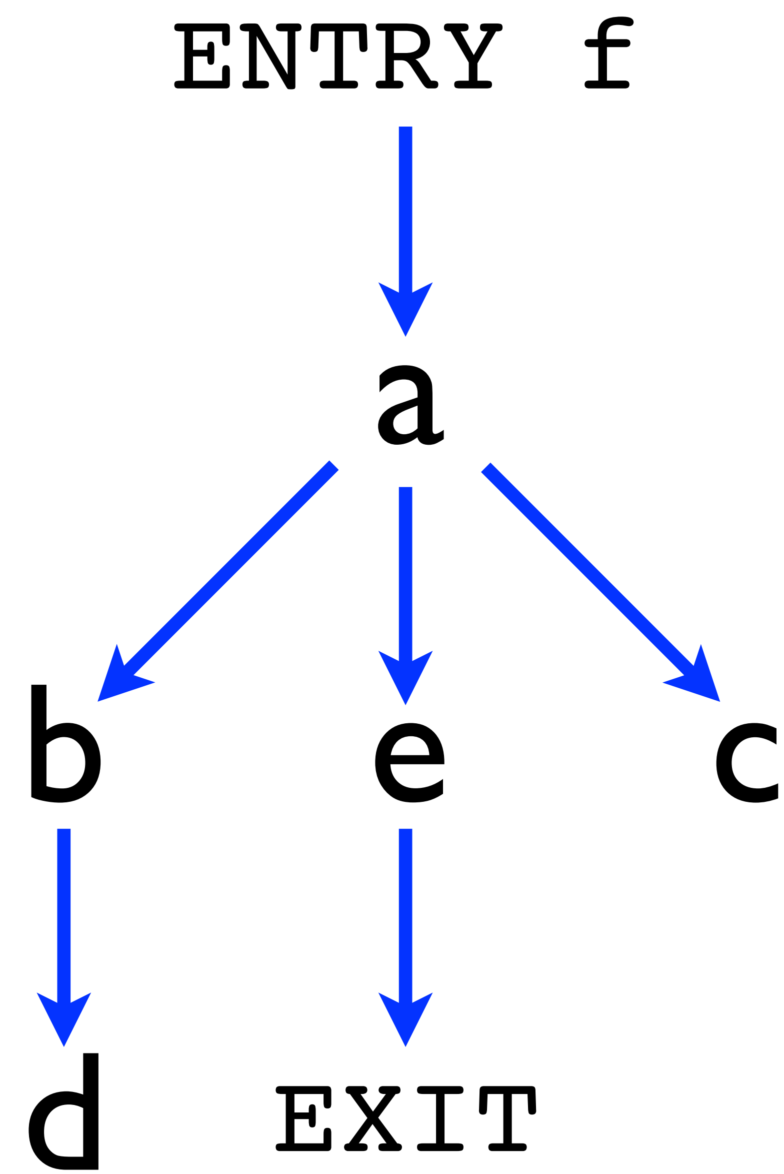
Back edges

We can now define the concept of a *back edge*.

In a flowgraph, a back edge is one whose head dominates its tail.

Back edges



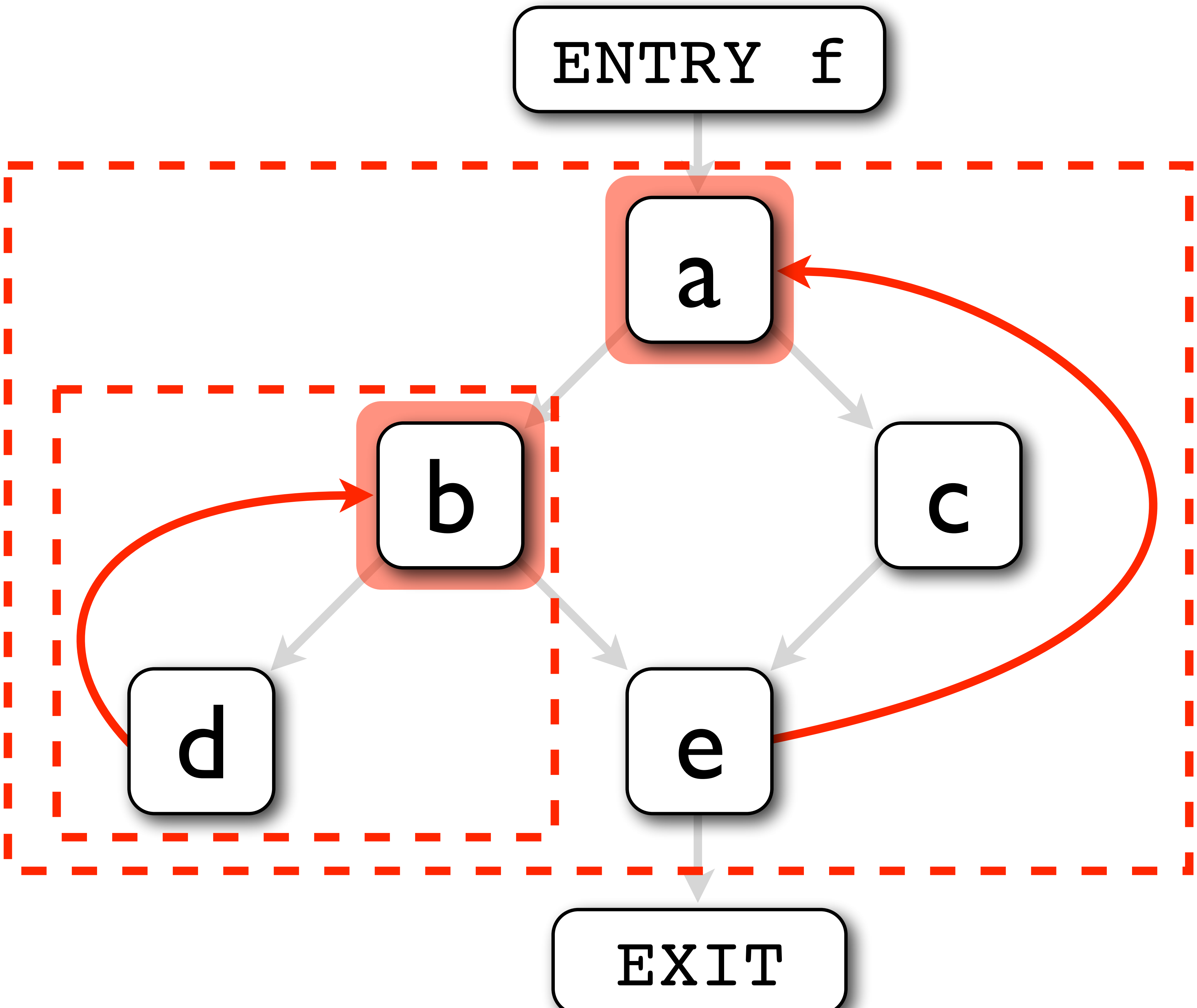


Finding loops

Each back edge has an associated loop.

The head of a back edge points to the *loop header*, and the *loop body* consists of all the nodes from which the tail of the back edge can be reached without passing through the loop header.

Finding loops

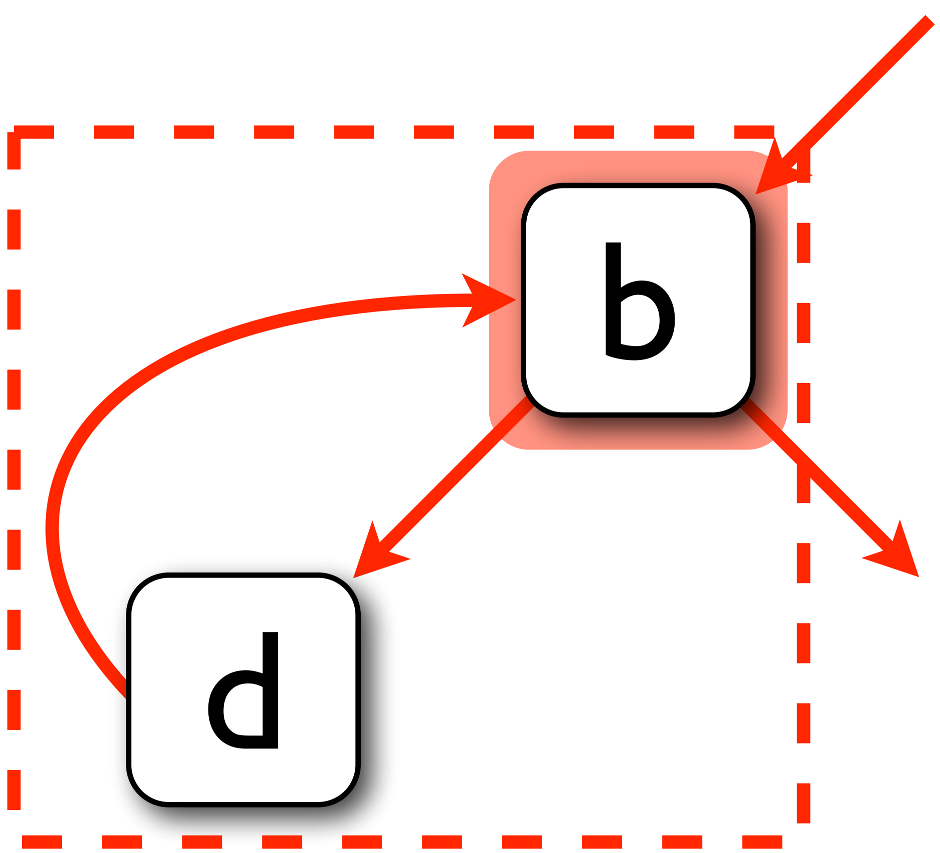


Finding loops

Once each loop has been identified, we can examine its structure to determine what kind of loop it is, and hence how best to represent it in source code.

Finding loops

Here, the loop header contains a conditional which determines whether the loop body is executed, and the last node of the body unconditionally transfers control back to the header.



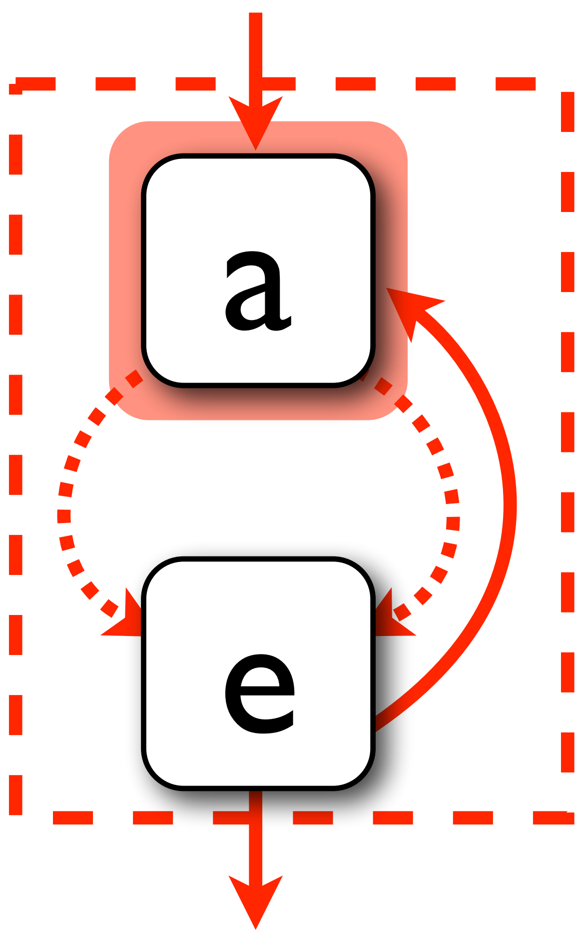
This structure corresponds to source-level

`while (...) {...}`

syntax.

Finding loops

Here, the loop header unconditionally allows the body to execute, and the last node of the body tests whether the loop should execute again.

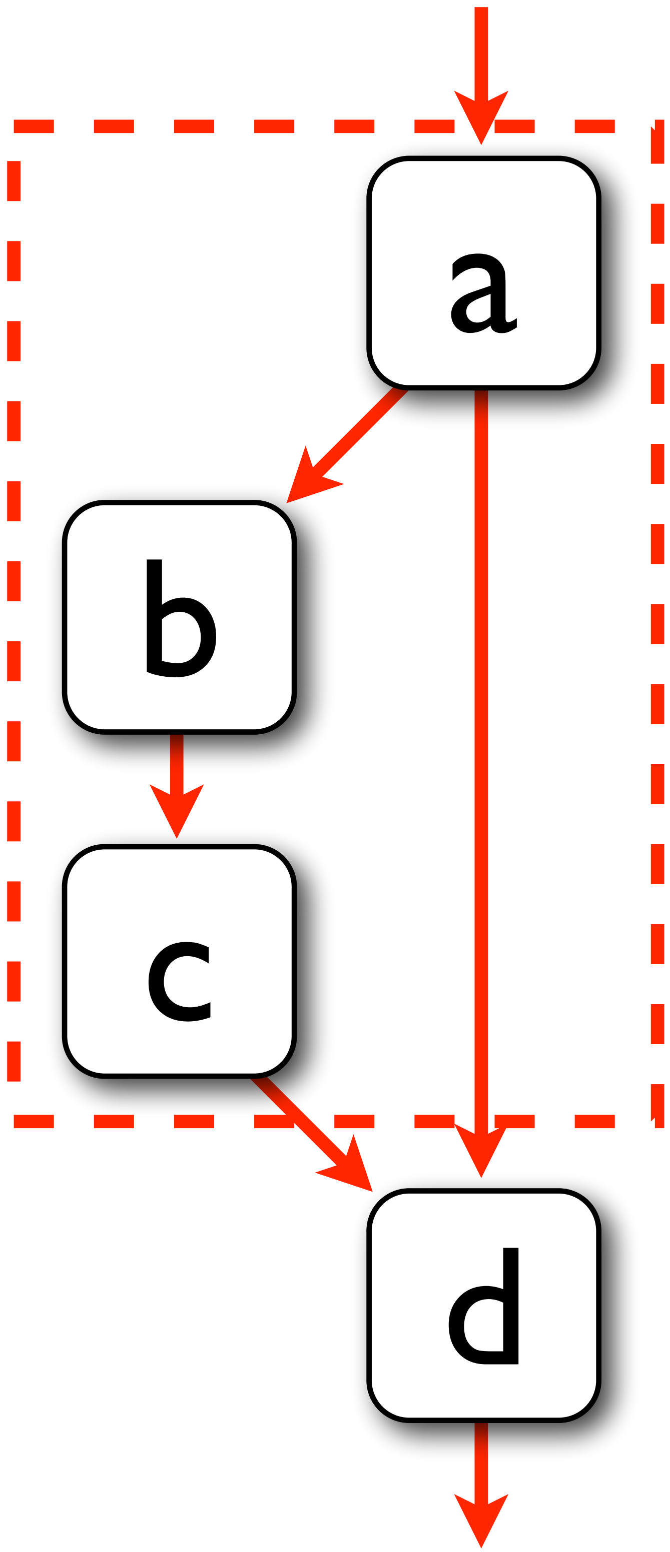


This structure corresponds to source-level `do {...} while (...)` syntax.

Finding conditionals

A similar principle applies when trying to reconstruct conditionals: we look for structures in the flowgraph which may be represented by particular forms of high-level language syntax.

Finding conditionals

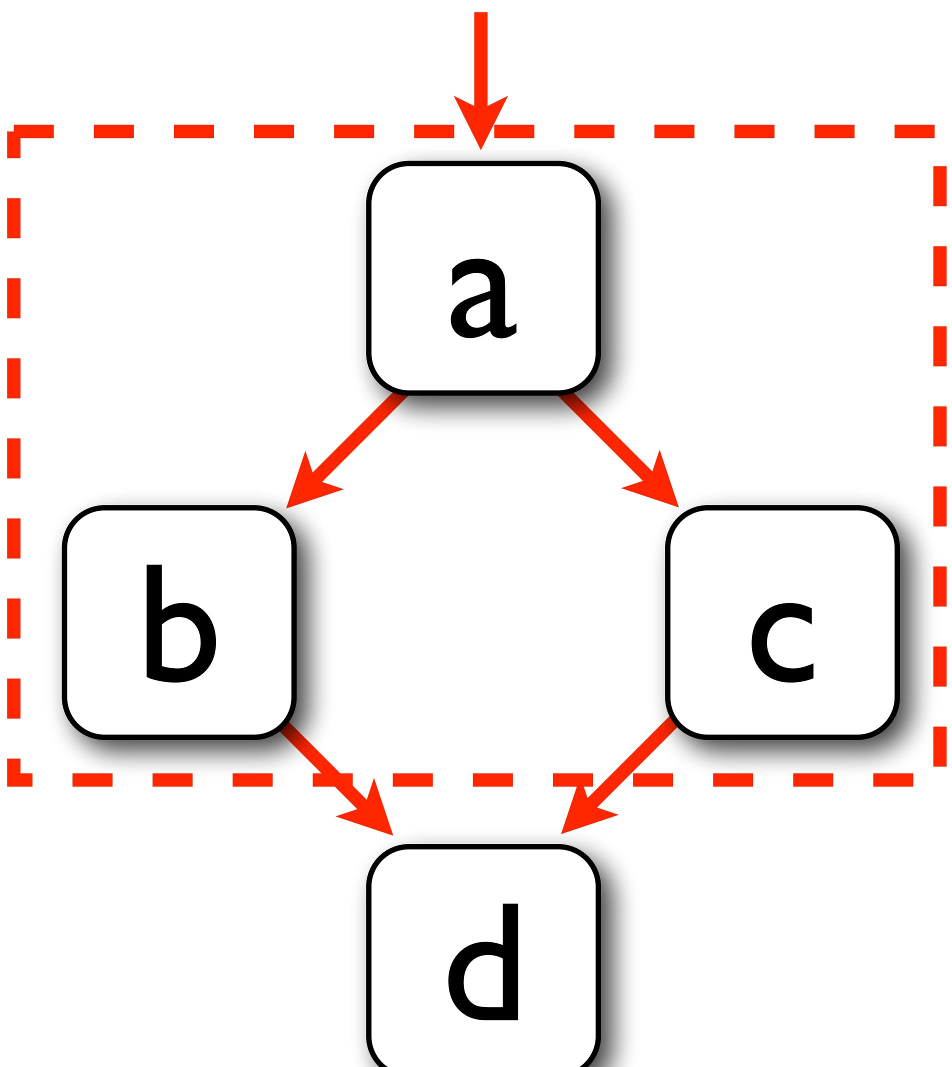


The first node in this interval transfers control to one node if some condition is true, otherwise it transfers control to another node (which control also eventually reaches along the first branch).

This structure corresponds to source-level `if (...) then {...}` syntax.

Finding conditionals

The first node in this interval transfers control to one node if some condition is true, and another node if the condition is false; control always reaches some later node.



This structure corresponds to source-level

```
if (...) then {...}
else {...}
```

syntax.

Control reconstruction

We can keep doing this for whatever other control-flow constructs are available in our source language.

Once an interval of the flowgraph has been matched against a higher-level control structure in this way, its entire subgraph can be replaced with a single node which represents that structure and contains all of the information necessary to generate the appropriate source code.

Type reconstruction

Many source languages also contain rich information about the *types* of variables: integers, booleans, arrays, pointers, and more elaborate data-structure types such as `unions` and `structs`.

At the target code level there are no variables, only registers and memory locations.

Types barely exist here: memory contains arbitrary bytes, and registers contain integers of various bit-widths (possibly floating-point values too).

Type reconstruction

Reconstruction of the types of source-level variables is made more difficult by the combination of SSA and register allocation performed by an optimising compiler.

SSA splits one user variable into many variables — one for each static assignment — and any of these variables with disjoint live ranges may be allocated to the same architectural register.

Type reconstruction

So each user variable may be spread between several registers — and each register may hold the value of different variables at different times.

It's therefore a bit hopeless to try to give a type to each architectural register; the notional type of the value held by any given register will change during execution.

```
int x = 42;            →   MOV r3,#42
⋮                          ⋮
char *y = "42";            MOV r3,#0xFF34
```

Type reconstruction

Happily, we can undo the damage by once again converting to SSA form: this will split a single register into many registers, each of which can be assigned a different type if necessary.

```
MOV r3,#42
⋮
MOV r3,#0xFF34
```

$\longrightarrow$

```
MOV r3a,#42
⋮
MOV r3b,#0xFF34
```

Type reconstruction

C

```
int foo (int *x) {
  return x[1] + 2;
}
```

compile

ARM

```
f: ldr r0,[r0,#4]
   add r0,r0,#2
   mov r15,r14
```

Type reconstruction

C

```
int f (int r0) {
  r0 = *(int *)(r0 + 4);
  r0 = r0 + 2;
  return r0;
}
```

decompile

ARM

```
f: ldr r0,[r0,#4]
   add r0,r0,#2
   mov r15,r14
```

Type reconstruction

```
int f (int r0) {
   r0 = *(int *)(r0 + 4);
   r0 = r0 + 2;
   return r0;
}
```

$\downarrow$ SSA

```
int f (int r0_a) {
   int r0_b = *(int *)(r0_a + 4);
   int r0_c = r0_b + 2;
   return r0_c;
}
```

Type reconstruction

```
int f (int *r0_a) {
  int r0_b = *(r0_a + 1);
  int r0_c = r0_b + 2;
  return r0_c;
}
```

reconstruct types

```
int f (int r0_a) {
  int r0_b = *(int *)(r0_a + 4);
  int r0_c = r0_b + 2;
  return r0_c;
}
```

Type reconstruction

```
int f (int *r0_a) {
   int r0_b = *(r0_a + 1);
   int r0_c = r0_b + 2;
   return r0_c;
}
```

reconstruct syntax

```
int f (int *r0_a) {
   int r0_b = r0_a[1];
   int r0_c = r0_b + 2;
   return r0_c;
}
```

Type reconstruction

```
int f (int *r0_a) {
  return r0_a[1] + 2;
}
```

propagate copies

```
int f (int *r0_a) {
  int r0_b = r0_a[1];
  int r0_c = r0_b + 2;
  return r0_c;
}
```

Type reconstruction

```
int f (int *r0a) {
  return r0a[1] + 2;
}
```

In fact, the return type could be anything, so more generally:

```
T f (T *r0a) {
  return r0a[1] + 2;
}
```

Type reconstruction

This is all achieved using constraint-based analysis: each target instruction generates constraints on the types of the registers, and we then solve these constraints in order to assign types at the source level.

Typing information is often incomplete intraprocedurally (as in the example); constraints generated at call sites help to fill in the gaps.

We can also infer `unions`, `structs`, etc.

Summary

- Decompilation is another application of program analysis and transformation
- Compilation discards lots of information about programs, some of which can be recovered
- Loops can be identified by using dominator trees
- Other control structure can also be recovered
- Types can be partially reconstructed with constraint-based analysis