Compilers I - Chapter 6: Optimisation and data-flow analysis

- Lecturers:
  - Part I: Paul Kelly (phjk@doc.ic.ac.uk)
    - Office: room 304, William Penney Building
  - Part II: Naranker Dulay (nd@doc.ic.ac.uk)
    - Office: room 562
- Materials:
  - Textbook
  - Course web pages (http://www.doc.ic.ac.uk/~phjk/Compilers)
  - Piazza (http://piazza.com/imperial.ac.uk/fall2016/221)

Overview

- This introductory course has focussed so far on fast, simple techniques which generated code that works reasonably well
- We now briefly look at what *optimising* compilers do, and how they do it
- Compare "gcc file.c" versus "gcc –O file.c"
- According to the gcc manual page ("man gcc"):
  - Without `-O', the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.
  - Without `-O', only variables declared "register" are allocated in registers

The plan

- To optimise or not to optimise?
- High-level vs low-level; role of analysis
- Peephole optimisation
- Local, global, interprocedural
  - Loop optimisations
  - Where optimisation fits in the compiler
    - Example: **live ranges**
      - Live ranges as a data flow problem
      - Solving the data-flow equations
      - Deriving the interference graph
    - Other data-flow analyses
    - **Loop-invariant code** and **code motion optimisations**
  - More sophisticated optimisations

Optimisation: example

- Consider the loop from tutorial exercise 4:

```
void P(int i, int j)
{
  int k, tmp;

  for (k=0; k<100; k++) {
    tmp = A[i+k];
    A[i+k] = A[j+k];
    A[j+k] = tmp;
  }
}
```

- What can optimisation do here?

Without optimisation….

```
_P:
        subl $36,%esp
        pushl %ebp
        pushl %ebx
        nop
        movl $0,28(%esp)
        .align 4
L3:
        cmpl $99,28(%esp)
        jle L6
        jmp L4
        .align 4
L6:
        movl 48(%esp),%eax
        movl 28(%esp),%edx
        addl %edx,%eax
        leal 0(,%eax,4),%edx
        movl $_A,%eax
        movl (%edx,%eax),%edx
        movl %edx,24(%esp)
        movl 48(%esp),%eax
        movl 28(%esp),%ecx
        leal (%ecx,%eax),%edx
        leal 0(,%edx,4),%eax
        movl $_A,%edx
        movl 52(%esp),%ecx
        movl 28(%esp),%ebx
        addl %ebx,%ecx
        leal 0(,%ecx,4),%ebx
        movl $_A,%ecx
        movl (%ebx,%ecx),%ebx
        movl %ebx,(%eax,%edx)
        movl 52(%esp),%eax
        movl 28(%esp),%ecx
        leal (%ecx,%eax),%edx
        leal 0(,%edx,4),%eax
        movl $_A,%edx
        movl 24(%esp),%ecx
        movl %ecx,(%eax,%edx)
L5:
        incl 28(%esp)
        jmp L3
        .align 4
L4:
L2:
        popl %ebx
        popl %ebp
        addl $36,%esp
        ret
```

Without optimisation, code is large, slow, but compiles quickly and works well with the debugger

31 instructions in loop

Performance:

- 27.5ns per iteration (gcc 2.95, 800MHz Pentium III)

Without optimisation….

```
_P:
        subl $36,%esp
        pushl %ebp
        pushl %ebx
        nop
        movl $0,28(%esp)
        .align 4
L3:
        cmpl $99,28(%esp)
        jle L6
        jmp L4
        .align 4
L6:
        movl 48(%esp),%eax
        movl 28(%esp),%edx
        addl %edx,%eax
        leal 0(,%eax,4),%edx
        movl $_A,%eax
        movl (%edx,%eax),%edx
        movl %edx,24(%esp)
        movl 48(%esp),%eax
        movl 28(%esp),%ecx
        leal (%ecx,%eax),%edx
        leal 0(,%edx,4),%eax
        movl $_A,%edx
        movl 52(%esp),%ecx
        movl 28(%esp),%ebx
        addl %ebx,%ecx
        leal 0(,%ecx,4),%ebx
        movl $_A,%ecx
        movl (%ebx,%ecx),%ebx
        movl %ebx,(%eax,%edx)
        movl 52(%esp),%eax
        movl 28(%esp),%ecx
        leal (%ecx,%eax),%edx
        leal 0(,%edx,4),%eax
        movl $_A,%edx
        movl 24(%esp),%ecx
        movl %ecx,(%eax,%edx)
L5:
        incl 28(%esp)
        jmp L3
        .align 4
L4:
L2:
        popl %ebx
        popl %ebp
        addl $36,%esp
        ret
```

Without optimisation, code is large, slow, but compiles quickly and works well with the debugger

31 instructions in loop

Performance:

- 8.2ns per iteration (gcc 3.2.2, 2GHz Pentium IV)

With optimisation:

- In this extreme example, optimised code is four times faster
  - Use registers not stack
  - One jump per iteration
  - Loop-invariant offset calculation moved out
  - Array pointers incremented instead of recalculated
  - Loop control variable replaced with down-counter
- Even faster code is possible by loop unrolling

```
_P: pushl %edi
    pushl %esi
    movl $99,%edi
    pushl %ebx
    movl $_A,%esi
    movl 20(%esp),%ebx
    movl 16(%esp),%ecx
    sall $2,%ebx
    sall $2,%ecx
    .align 4
L6:
    movl (%esi,%ecx),%edx
    movl (%esi,%ebx),%eax
    movl %eax,(%esi,%ecx)
    movl %edx,(%esi,%ebx)
    addl $4,%ecx
    addl $4,%ebx
    decl %edi
    jns L6
    popl %ebx
    popl %esi
    popl %edi
    ret
```

8 instructions in loop

Performance:

- 6.71ns per iteration (gcc 2.95, 800MHz Pentium III)

With optimisation:

- In this extreme example, optimised code is 2-4 times faster
  - Use registers not stack
  - One jump per iteration
  - Loop-invariant offset calculation moved out
  - Array pointers incremented instead of recalculated
  - Loop control variable replaced with down-counter

```
_P: pushl %edi
    pushl %esi
    movl $99,%edi
    pushl %ebx
    movl $_A,%esi
    movl 20(%esp),%ebx
    movl 16(%esp),%ecx
    sall $2,%ebx
    sall $2,%ecx
    .align 4
L6:
    movl (%esi,%ecx),%edx
    movl (%esi,%ebx),%eax
    movl %eax,(%esi,%ecx)
    movl %edx,(%esi,%ebx)
    addl $4,%ecx
    addl $4,%ebx
    decl %edi
    jns L6
    popl %ebx
    popl %esi
    popl %edi
    ret
```

8 instructions in loop

Performance:

- 3.4ns per iteration (gcc 3.2.2, 2GHz Pentium IV)

With optimisation:

- In this extreme example, optimised code is 2-4 times faster
  - Use registers not stack
  - One jump per iteration
  - Loop-invariant offset calculation moved out
  - Array pointers incremented instead of recalculated
  - Loop control variable replaced with down-counter

```
_P:  pushl  %esi
     pushl  %ebx
     movl   12(%esp), %edx
     movl   16(%esp), %ecx
     leal   0(,%edx,4), %ebx
     subl   %edx, %ecx
     movl   %ecx, %edx
     leal   _A(%ebx), %eax
     addl   $_A+400, %ebx
L2:  movl   (%eax), %ecx
     movl   (%eax,%edx,4), %esi
     movl   %esi, (%eax)
     movl   %ecx, (%eax,%edx,4)
     addl   $4, %eax
     cmpl   %ebx, %eax
     jne    L2
     popl   %ebx
     popl   %esi
     ret
```

7 instructions in loop
- 0.7ns per iteration (gcc 5.4 –O3, 3.2GHz Intel Skylake i76600U)

With optimisation:

- In this code, the compiler has used vector instructions that operate on four operands at a time
- The full code is rather complicated as care is needed to check whether the memory regions overlap

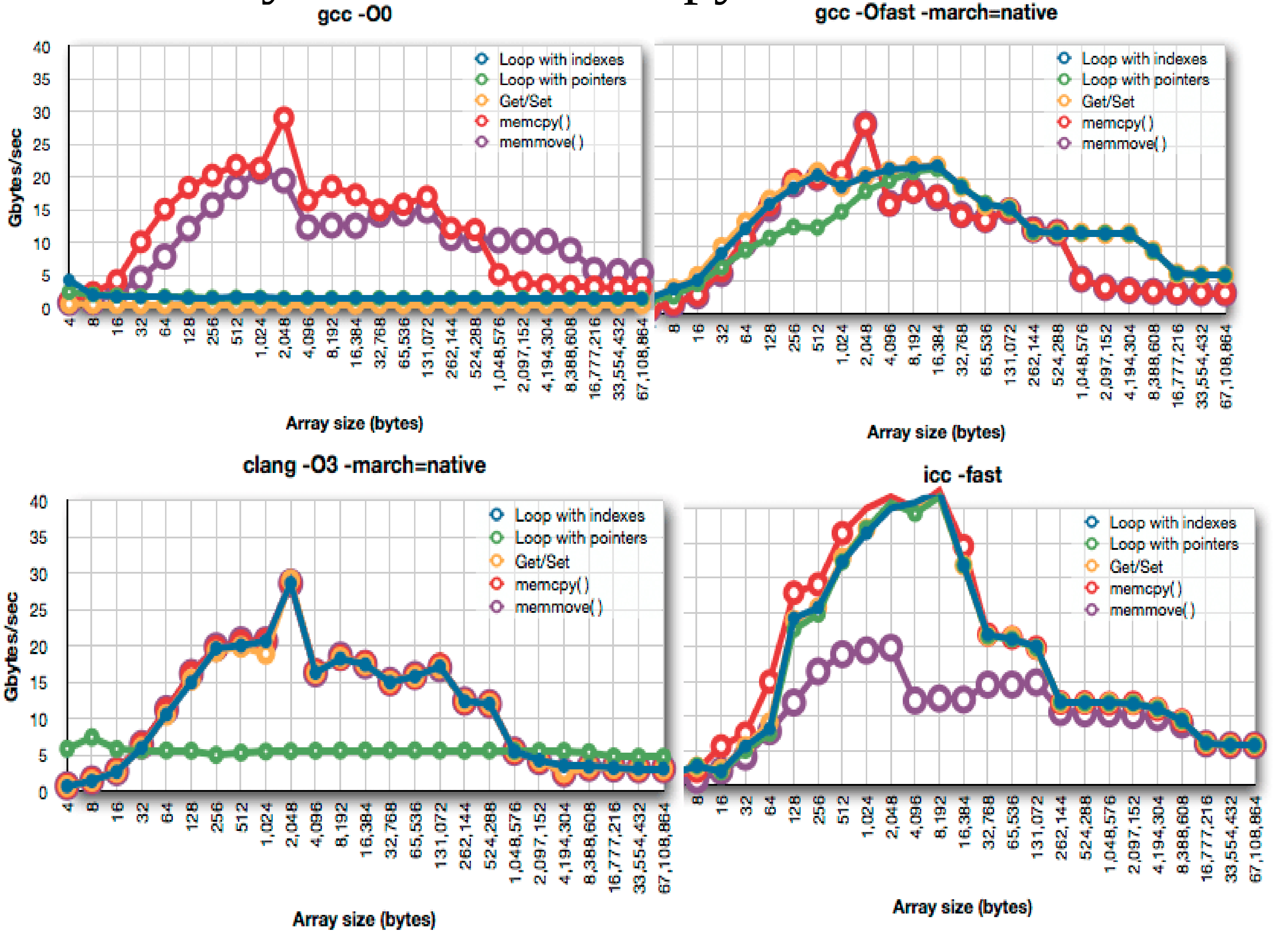
- (this example goes far beyond what we can hope to cover in this course)

```
_P:  ....
     ....
.L5: movdqu  (%rdx,%rax), %xmm0
     movdqu  (%rcx,%rax), %xmm1
     movdqu  %xmm1, (%rdx,%rax)
     movdqu  %xmm0, (%rcx,%rax)
     addq    $16, %rax
     cmpq    $400, %rax
     jne     .L5
     rep ret
```

7 instructions in loop

- 0.2ns per iteration (gcc 4.8.4 –O3, –march=native, 3.2GHz Intel Skylake i76600U)
- Vectorised

Never write your own memcopy



http://nadeausoftware.com/articles/2012/05/c_c_tip_how_copy_memory_quickly

Optimisation principles…

- To generate really good code, need to combine many techniques, including both high-level and low-level
- High-level example: **inlining**
  - replace a call “f(x)” with the function body itself
  - Avoids call/return overheads
  - Also creates further opportunities…
  - Can we inline virtual method calls “x.f(y)”?
  - Need *static analysis* of possible types of “x”
- Low-level example: **instruction scheduling**
  - Re-order instructions so processor executes them in parallel
  - To switch order of load A[i] and store A[j], need *dependence analysis*: could i and j refer to same location?

A simple local technique – peephole optimisation

- Scan assembly code, replacing obviously inane combinations of instructions (eg mov R0,a; mov a,R0)
- Easy to implement:

```
peep :: [Instruction] -> [Instruction]
peep (Store r1 dest : Load r2 src : rest)
 | src == dest
   = Store r1 dest : (peep (Load r2 r1 : rest))
 | otherwise
   = Store r1 dest : (peep (Load r2 src : rest))
```

- Endless possibilities…
- *Phase ordering problem*: in which sequence should optimisations be applied?

Spectrum…

- Peephole optimisation works at instruction level
- The Sethi-Ullman "weights" algorithm: expressions
- "**Local**" optimisation works at the level of *basic blocks* – a sequence of instructions which has a single point of entry and a single point of exit
- "**Global**" optimisation works on a whole procedure
- **Interprocedural** optimisation works on the whole program

- Local: generally runs quickly and easy to validate
- Global: may have worse-than-linear complexity, eg $O(N^2)$ where N is number of instructions, basic blocks, or local variables
- Interprocedural: rare – hard to avoid excessive compilation time

Some loop optimisations…

- Loop-invariant code motion
  - An instruction is **loop-invariant** if its operands can only arrive from outside the loop
  - move loop-invariant instructions into loop header
- Detection of induction variables
  - **Induction variable** is a variable which increases/decreases by a (loop-invariant) constant on each iteration
- **Strength reduction**: calculate induction variable by incrementing, instead of by multiplying other induction variables
- **Control variable selection**: replace loop control variable with one of the induction variables actually used in the loop

Loop optimisations - example

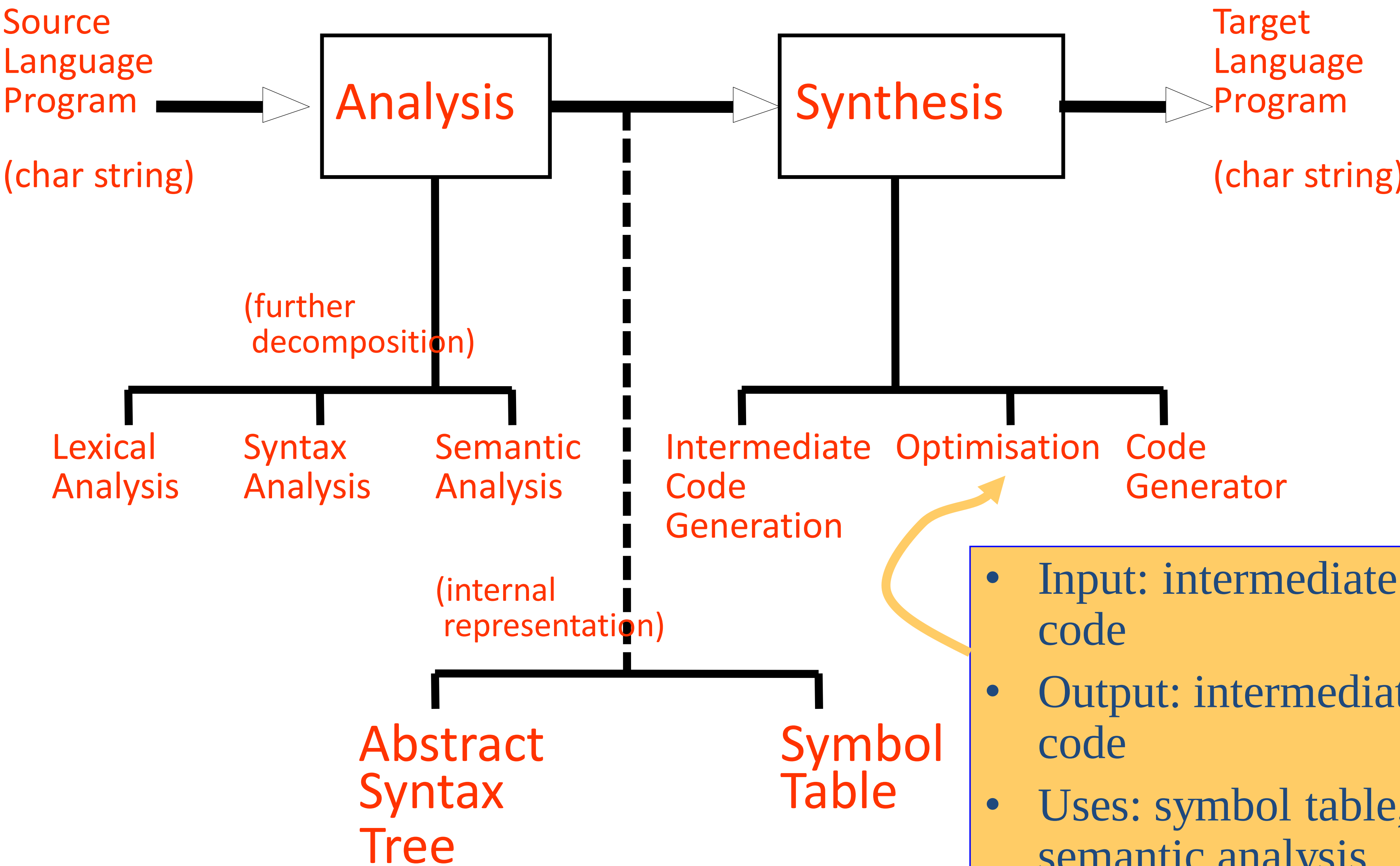
```
int P(int N, int M)
{
  int i, u, v, w, x, y;
  int z = 0;

  for (i=0; i<N; i++) {
    w = w+10;
    x = w*10;
    y = z*(w-x);
    u = w+x+y+N+M;
    v = v+u;
  }
  return v;
}
```

1. y is constant
2. w-x is dead code
3. y+N+M is loop-invariant
4. i, w and x are induction variables (so is w+x)
5. x increases by 100 each iteration
6. i is used only to control the loop, and can be omitted if convenient

1. (constant propagation Appel pg457)
2. (dead code elimination pg457,397)
3. (loop-invariant code motion pg422)
4. (induction variable recognition pg426)
5. (strength reduction ditto)
6. (rewriting comparisons, pg428)

Where does optimisation happen?



- Input: intermediate code
- Output: intermediate code
- Uses: symbol table, semantic analysis

Intermediate code

- In our simple compiler, translator traverses AST and produces assembler code directly
- In optimising compiler, translator traverses AST and produces "intermediate code"
- Intermediate code is designed to
  - Represent all primitive operations necessary to execute program
  - In a uniform way, easy to analyse and manipulate
  - Independently of target instruction set
- Compiler writers argue… Appel advocates two IRs:
    - Tree: before instruction selection
    - FlowGraph: after instruction selection
- IR uses "temporaries" T0, T1, T2… instead of real registers; after optimisation, use graph colouring to assign temporaries to real registers

Dataflow analysis (DFA)

- Optimisation consists of **analysis** and **transformation**
- **Analysis**: deduce program properties from IR
  - Analyse effect of each instruction
  - Compose these effects to derive information about the entire procedure
- Consider: Add (Reg T0) (Reg T1)
  - Uses temporaries T0 and T1
  - Kills old definition of T1
  - Generates new definition of T1
- We will see how to do "dataflow analysis" in order to use this local information to derive global properties

Example dataflow analysis: **live ranges**

- Recall graph colouring:

1. Generate code using temporaries T0… instead of registers
2. For each temporary T*i*, find T*i*'s "**live range**" – the set of instructions for which T*i* must reside in a register
3. LiveRange(T*i*) intersects LiveRange(T*j*) means they have to be allocated to different registers – they *interfere*
4. Assemble the register interference graph (RIG)
5. Colour the RIG by assigning real registers to temporaries avoiding interference
6. If successful, replace temporaries with registers and generate code
7. If graph cannot be coloured, find a temporary to *spill* to memory, then retry

Preliminary: build the control flow graph

- data CFG = ControlFlowGraph [CFGNode]
- data CFGNode = Node Id Instruction [Register] [Register] [Id]

  uses *defs* *succs*
- type Id = Int
- data Register = D Int | T Int *(temporaries before, real after)*
- buildCFG :: [Instruction] -> CFG

- Each node of the control flow graph contains an instruction, together with:
  - nodeDefs cfgnode = list of temporaries which this instruction updates
  - nodeUses cfgnode = list of temporaries which this instruction reads
  - nodeSuccs cfgnode = list of nodes which might be executed next

Source code

```
while (b<10)
{
  if (b<a)
    a = a*7;
    b = a+1;
  else
    a = b-1;
}
```

Intermediate code

```
    Bra L2
L1:
    cmp b a
    bge L3
    mul #7 a
    mov a b
    add #1 b
    bra L4
L3:
    mov b a
    sub #1 a
L4:
L2:
    Cmp b #10
    Blt L1
```

Control flow graph

1	Bra L2			10
2	cmp b a	a,b		3
3	bge L3			4,8
4	mul #7 a	a	a	5
5	mov a b	a	b	6
6	add #1 b	b	b	7
7	bra L4			10
8	mov b a	b	a	9
9	sub #1 a	a	a	10
10	Cmp b #10	b		11
11	Blt L1			12,2

Finding live ranges… example

Live variable analysis - definition

- **Point**: any location *between* adjacent nodes
- **Path**: a sequence of points $p_1..p_i\ p_{i+1}..p_n$ such that p_{i+1} is the immediate successor of p_i in the CFG

- "**x is live at p**": for some variable x and point p, the value of x could be used along some path starting at p.

1	Bra L2			10
2	cmp b a	a,b		3
3	bge L3			4,8
4	mul #7 a	a	a	5
5	mov a b	a	b	6
6	add #1 b	b	b	7
7	bra L4			10
8	mov b a	b	a	9
9	sub #1 a	a	a	10
10	Cmp b #10	b		11
11	Blt L1			12,2

"**x is live at p**": for some variable x and point p, the value of x could be used along some path starting at p.

1	Bra L2			10
2	cmp b a	a,b		3
3	bge L3			4,8
4	mul #7 a	a	a	5
5	mov a b	a	b	6
6	add #1 b	b	b	7
7	bra L4			10
8	mov b a	b	a	9
9	sub #1 a	a	a	10
10	Cmp b #10	b		11
11	Blt L1			12,2

"**x is live at p**": for some variable x and point p, the value of x could be used along some path starting at p.

Consider variable b after node 1

1	Bra L2			10
2	cmp b a	a,b		3
3	bge L3			4,8
4	mul #7 a	a	a	5
5	mov a b	a	b	6
6	add #1 b	b	b	7
7	bra L4			10
8	mov b a	b	a	9
9	sub #1 a	a	a	10
10	Cmp b #10	b		11
11	Blt L1			12,2

"**x is live at p**": for some variable x and point p, the value of x could be used along some path starting at p.

Consider variable b after node 2

1	Bra L2			10
2	cmp b a	a,b		3
3	bge L3			4,8
4	mul #7 a	a	a	5
5	mov a b	a	b	6
6	add #1 b	b	b	7
7	bra L4			10
8	mov b a	b	a	9
9	sub #1 a	a	a	10
10	Cmp b #10	b		11
11	Blt L1			12,2

"**x is live at p**": for some variable x and point p, the value of x could be used along some path starting at p.

Consider variable b after node 4

Dataflow equations for live variable analysis

Define:

- LiveIn(n): the set of temporaries live immediately **before** node n
- LiveOut(n): the set of temporaries live immediately **after** node n

- A variable is live immediately after node n if it is live before any of n's successors

- A variable is live immediately before node n if:
  - It is live after node n (ie some later instruction reads it)
  - Unless it is overwritten by node n

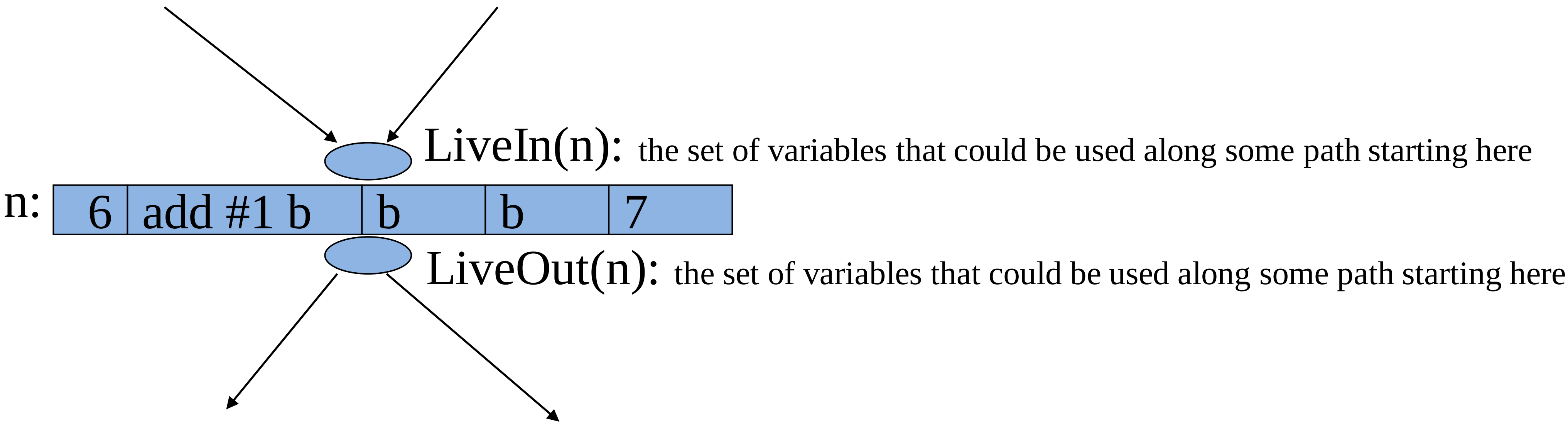
  OR
  - It is used by node n (ie the instruction reads it)

Dataflow equations for live variable analysis

- LiveIn(n): set of temporaries live immediately **before** node n
- LiveOut(n): set of temporaries live immediately **after** node n
- A variable is live immediately after node n if it is live before any of n's successors:
  - $\mathrm{LiveOut}(n) = \bigcup_{s \in \mathrm{succ}(n)} \mathrm{LiveIn}(s)$
- A variable is live immediately before node n if:
  - It is live after node n (ie some later instruction reads it)
  - Unless it is overwritten by node n

  OR
  - It is used by node n (ie the instruction reads it)
  - $\mathrm{LiveIn}(n) = \mathrm{uses}(n) \cup (\mathrm{LiveOut}(n) - \mathrm{defs}(n))$

- What’s the difference between LiveIn and LiveOut?



$\text{LiveIn}(1) = \text{uses}(1) \cup (\text{LiveOut}(1) - \text{defs}(1))$

$\text{LiveOut}(1) = \bigcup_{s \in \text{succ}(1)} \text{LiveIn}(s)$

$\text{LiveIn}(2) = \text{uses}(2) \cup (\text{LiveOut}(2) - \text{defs}(2))$

$\text{LiveOut}(2) = \bigcup_{s \in \text{succ}(2)} \text{LiveIn}(s)$

$\text{LiveIn}(3) = \text{uses}(3) \cup (\text{LiveOut}(3) - \text{defs}(3))$

$\text{LiveOut}(3) = \bigcup_{s \in \text{succ}(3)} \text{LiveIn}(s)$

$\text{LiveIn}(4) = \text{uses}(4) \cup (\text{LiveOut}(4) - \text{defs}(4))$

$\text{LiveOut}(4) = \bigcup_{s \in \text{succ}(4)} \text{LiveIn}(s)$

$\text{LiveIn}(5) = \text{uses}(5) \cup (\text{LiveOut}(5) - \text{defs}(5))$

$\text{LiveOut}(5) = \bigcup_{s \in \text{succ}(5)} \text{LiveIn}(s)$

$\text{LiveIn}(6) = \text{uses}(6) \cup (\text{LiveOut}(6) - \text{defs}(6))$

$\text{LiveOut}(6) = \bigcup_{s \in \text{succ}(6)} \text{LiveIn}(s)$

$\text{LiveIn}(7) = \text{uses}(7) \cup (\text{LiveOut}(7) - \text{defs}(7))$

$\text{LiveOut}(7) = \bigcup_{s \in \text{succ}(7)} \text{LiveIn}(s)$

$\text{LiveIn}(8) = \text{uses}(8) \cup (\text{LiveOut}(8) - \text{defs}(8))$

$\text{LiveOut}(8) = \bigcup_{s \in \text{succ}(8)} \text{LiveIn}(s)$

$\text{LiveIn}(9) = \text{uses}(9) \cup (\text{LiveOut}(9) - \text{defs}(9))$

$\text{LiveOut}(9) = \bigcup_{s \in \text{succ}(9)} \text{LiveIn}(s)$

$\text{LiveIn}(10) = \text{uses}(10) \cup (\text{LiveOut}(10) - \text{defs}(10))$

$\text{LiveOut}(10) = \bigcup_{s \in \text{succ}(10)} \text{LiveIn}(s)$

$\text{LiveIn}(11) = \text{uses}(11) \cup (\text{LiveOut}(11) - \text{defs}(11))$

$\text{LiveOut}(11) = \bigcup_{s \in \text{succ}(11)} \text{LiveIn}(s)$

Id		Uses	Defs	Ids of succs
1	Bra L2			10
2	cmp b a	a,b		3
3	bge L3			4,8
4	mul #7 a	a	a	5
5	mov a b	a	b	6
6	add #1 b	b	b	7
7	bra L4			10
8	mov b a	b	a	9
9	sub #1 a	a	a	10
10	Cmp b #	b		11
11	Blt L1			12,2

- 22 simultaneous equations

LiveIn(1)= uses(1) U (LiveOut(1) – defs(1))

LiveOut(1)=LiveIn(10)

LiveIn(2)= uses(2) U (LiveOut(2) – defs(2))

LiveOut(2)=LiveIn(3)

LiveIn(3)= uses(3) U (LiveOut(3) – defs(3))

LiveOut(3)=LiveIn(4) U LiveIn(8)

LiveIn(4)= uses(4) U (LiveOut(4) – defs(4))

LiveOut(4)=LiveIn(5)

LiveIn(5)= uses(5) U (LiveOut(5) – defs(5))

LiveOut(5)=LiveIn(6)

LiveIn(6)= uses(6) U (LiveOut(6) – defs(6))

LiveOut(6)=LiveIn(7)

LiveIn(7)= uses(7) U (LiveOut(7) – defs(7))

LiveOut(7)=LiveIn(10)

LiveIn(8)= uses(8) U (LiveOut(8) – defs(8))

LiveOut(8)=LiveIn(9)

LiveIn(9)= uses(9) U (LiveOut(9) – defs(9))

LiveOut(9)=LiveIn(10)

LiveIn(10)= uses(10) U (LiveOut(10) – defs(10))

LiveOut(10)=LiveIn(11)

LiveIn(11)= uses(11) U (LiveOut(11) – defs(11))

LiveOut(11)=LiveIn(12) U LiveIn(2)

Id		Uses	Defs	Ids of succs
1	Bra L2			10
2	cmp b a	a,b		3
3	bge L3			4,8
4	mul #7 a	a	a	5
5	mov a b	a	b	6
6	add #1 b	b	b	7
7	bra L4			10
8	mov b a	b	a	9
9	sub #1 a	a	a	10
10	Cmp b #	b		11
11	Blt L1			12,2

Clearer if we substitute in the successors:
succs(11) = {12,2}

- 22 simultaneous equations

Solving the dataflow equations

- We have a system of simultaneous equations for LiveIn(n) and LiveOut(n) for each node n
- How can we solve them?

Solving the dataflow equations

- Idea: Iterate!

```
for each n in CFG {
  LiveIn(n) := {}; LiveOut(n) := {};
}
repeat {
  for each n in CFG {
    LiveIn(n) = uses(n) U (LiveOut(n) – defs(n));
    LiveOut(n) = U s ∈ succ(n) LiveIn(s);
  }
} until LiveIn and LiveOut stop changing
```

Iteration… walkthrough

Node	uses	defs	succs	Step 0 in	Step 0 out
1			10	{ }	{ }
2	a,b		3	{ }	{ }
3			4,8	{ }	{ }
4	a	a	5	{ }	{ }
5	a	b	6	{ }	{ }
6	b	b	7	{ }	{ }
7			10	{ }	{ }
8	b	a	9	{ }	{ }
9	a	a	10	{ }	{ }
10	b		11	{ }	{ }
11			12,2	{ }	{ }

```
for each n in CFG {
  LiveIn(n) := {}; LiveOut(n) := {};
}
repeat {
  for each n in CFG {
    LiveIn(n) = uses(n) U (LiveOut(n) – defs(n));
    LiveOut(n) = U_{s ∈ succ(n)} LiveIn(s);
  }
} until LiveIn and LiveOut stop changing
```

Q: should I process the nodes in order?

- see Appel pg 226 for another example

Iteration… walkthrough

Node	uses	defs	succs	Step 1 in	Step 1 out
1			10	{ }	{b}
2	a,b		3	{a,b}	{ }
3			4,8	{ }	{a,b}
4	a	a	5	{a}	{a}
5	a	b	6	{a}	{b}
6	b	b	7	{b}	{ }
7			10	{ }	{b}
8	b	a	9	{b}	{a}
9	a	a	10	{a}	{b}
10	b		11	{b}	{ }
11			12,2	{ }	{ }

```
for each n in CFG {
  LiveIn(n) := {}; LiveOut(n) := {};
}
repeat {
  for each n in CFG {
    LiveIn(n) = uses(n) U (LiveOut(n) – defs(n));
```

$$\text{LiveOut}(n) = \bigcup_{s \in succ(n)} \text{LiveIn}(s);$$

```
  }
} until LiveIn and LiveOut stop changing
```

Q: should I process the nodes in order?

- see Appel pg 226 for another example

Node	uses	defs	succs	Step 1 in	Step 1 out	Step 2 in	Step 2 out	Step 3 in	Step 3 out	Step 4 in	Step 4 out	Step 5 in	Step 5 out	Step 5 in	Step 5 out
1			10	{ }	**{b}**	{ }	{b}	{ }	**{a,b}**	**{a,b}**	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}
2	a,b		3	**{a,b}**	{ }	{a,b}	**{a,b}**	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}
3			4,8	{ }	**{a,b}**	**{a,b}**	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}
4	a	a	5	**{a}**	**{a}**	{a}	{a}	{a}	{a}	{a}	{a}	{a}	{a}	{a}	{a}
5	a	b	6	**{a}**	**{b}**	{a}	{b}	**{a}**	{b}	{a}	{b}	{a}	{b}	{a}	**{a,b}**
6	b	b	7	**{b}**	{ }	{b}	**{b}**	{b}	{b}	{b}	{b}	{b}	**{a,b}**	**{a,b}**	{a,b}
7			10	{ }	**{b}**	**{b}**	{b}	{b}	{b}	{b}	**{a,b}**	**{a,b}**	{a,b}	{a,b}	{a,b}
8	b	a	9	**{b}**	**{a}**	{b}	{a}	**{a,b}**	**{a,b}**	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}
9	a	a	10	**{a}**	**{b}**	**{a,b}**	{b}	{a,b}	{b}	{a,b}	**{a,b}**	{a,b}	{a,b}	{a,b}	{a,b}
10	b		11	**{b}**	{ }	{b}	{ }	{b}	**{a,b}**	**{a,b}**	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}
11			12,2	{ }	{ }	{ }	**{a,b}**	**{a,b}**	{a.b}	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}	{a,b}

LiveIn(n) = uses(n) U (LiveOut(n) – defs(n));

LiveOut(n) = $\bigcup_{s \in succ(n)}$ LiveIn(s);

Real example: factorial loop

Concrete syntax

```
program
  declare x :
   Integer
  declare a :
   Integer
begin
  a := 1
  for x = 1 to 10
    a := a * x
  end
end
```

Abstract syntax

```
(Program [Decl "a" Integer]
 [(Assign (Var "a") (Const 1)),
  (For "x" (Const 1) (Const 10)
   [(Assign (Var "a")
     (Binop Times (Ref (Var "a")) (Ref (Var "x"))))]
)])
```

Real example: factorial loop

Concrete syntax

```
program
  declare x :
   Integer
  declare a :
   Integer
begin
  a := 1
  for x = 1 to 10
    a := a * x
  end
end
```

Code

```
.data
; Integer variable a has been allocated to T0
.text
move.l #1, T0
move.l #10, T1
move.l #1, T2
bra    L2
L1:
move.l T2, T3
move.l T0, T4
mul.l  T3, T4
move.l T4, T0
add.l  #1, T2
L2:
cmp.l  T1, T2
bgt    L3
bra    L1
L3:
move.l T2, x  (updates variable x on exit from loop – a bug! (?))
```

Real example: factorial loop

Concrete syntax

```
program
  declare x :
   Integer
  declare a :
   Integer
begin
  a := 1
  for x = 1 to 10
    a := a * x
  end
end
```

Code

```
Node 0 (Mov (ImmNum 1) (Reg T0)) [T0] [] [1] []
Node 1 (Mov (ImmNum 10) (Reg T1)) [T1] [] [2] [0]
Node 2 (Mov (ImmNum 1) (Reg T2)) [T2] [] [3] [1]
Node 3 (Bra "L2") [] [] [9] [2]
Node 4 (Mov (Reg T2) (Reg T3)) [T3] [T2] [5] [11]
Node 5 (Mov (Reg T0) (Reg T4)) [T4] [T0] [6] [4]
Node 6 (Mul (Reg T3) (Reg T4)) [T4] [T3,T4] [7] [5]
Node 7 (Mov (Reg T4) (Reg T0)) [T0] [T4] [8] [6]
Node 8 (Add (ImmNum 1) (Reg T2)) [T2] [T2] [9] [7]
Node 9 (Cmp (Reg T1) (Reg T2)) [] [T1,T2] [10] [3,8]
Node 10 (Bgt "L3") [] [] [11,12] [9]
Node 11 (Bra "L1") [] [] [4] [10]
Node 12 (Mov (Reg T2) (Abs "x")) [] [T2] [13] [10]
Node 13 Halt [] [] [] [12]
```

(Node id instrn defs uses succs preds)

Live range analysis for factorial example

	Step 0	Step 1	Step 2	Step 3	Step 4
LiveIns	([(0,[]),	([(0,[]),	([(0,[]),	([(0,[]),	([(0,[]),
	(1,[]),	(1,[]),	(1,[]),	(1,[]),	(1,[]),
	(2,[]),	(2,[]),	(2,[]),	(2,[T1]),	(2,[T1]),
	(3,[]),	(3,[]),	(3,[T1,T2]),	(3,[T1,T2]),	(3,[T1,T2]),
	(4,[]),	(4,[T2]),	(4,[T2,T0]),	(4,[T2,T0]),	(4,[T2,T0]),
	(5,[]),	(5,[T0]),	(5,[T0,T3]),	(5,[T0,T3]),	(5,[T0,T3,T2]),
	(6,[]),	(6,[T3,T4]),	(6,[T3,T4]),	(6,[T3,T4,T2]),	(6,[T3,T4,T2,T1]),
	(7,[]),	(7,[T4]),	(7,[T4,T2]),	(7,[T4,T2,T1]),	(7,[T4,T2,T1]),
	(8,[]),	(8,[T2]),	(8,[T2,T1]),	(8,[T2,T1]),	(8,[T2,T1]),
	(9,[]),	(9,[T1,T2]),	(9,[T1,T2]),	(9,[T1,T2]),	(9,[T1,T2]),
	(10,[]),	(10,[]),	(10,[T2]),	(10,[T2]),	(10,[T2]),
	(11,[]),	(11,[]),	(11,[]),	(11,[T2]),	(11,[T2,T0]),
	(12,[]),	(12,[T2]),	(12,[T2]),	(12,[T2]),	(12,[T2]),
	(13,[])],	(13,[])],	(13,[])],	(13,[])],	(13,[])],
LiveOuts	[(0,[]),	[(0,[]),	[(0,[]),	[(0,[]),	[(0,[]),
	(1,[]),	(1,[]),	(1,[]),	(1,[T1]),	(1,[T1]),
	(2,[]),	(2,[]),	(2,[T1,T2]),	(2,[T1,T2]),	(2,[T1,T2]),
	(3,[]),	(3,[T1,T2]),	(3,[T1,T2]),	(3,[T1,T2]),	(3,[T1,T2]),
	(4,[]),	(4,[T0]),	(4,[T0,T3]),	(4,[T0,T3]),	(4,[T0,T3,T2]),
	(5,[]),	(5,[T3,T4]),	(5,[T3,T4]),	(5,[T3,T4,T2]),	(5,[T3,T4,T2,T1]),
	(6,[]),	(6,[T4]),	(6,[T4,T2]),	(6,[T4,T2,T1]),	(6,[T4,T2,T1]),
	(7,[]),	(7,[T2]),	(7,[T2,T1]),	(7,[T2,T1]),	(7,[T2,T1]),
	(8,[]),	(8,[T1,T2]),	(8,[T1,T2]),	(8,[T1,T2]),	(8,[T1,T2]),
	(9,[]),	(9,[]),	(9,[T2]),	(9,[T2]),	(9,[T2]),
	(10,[]),	(10,[T2]),	(10,[T2]),	(10,[T2]),	(10,[T2,T0]),
	(11,[]),	(11,[]),	(11,[T2]),	(11,[T2,T0]),	(11,[T2,T0]),
	(12,[]),	(12,[]),	(12,[]),	(12,[]),	(12,[]),
	(13,[])])	(13,[])])	(13,[])])	(13,[])])	(13,[])])

Live range analysis for factorial example

	Step 5	Step 6	Step 7	Step 8	Step 9
LiveIns	([(0,[]),	([(0,[]),	([(0,[]),	([(0,[]),	([(0,[]),
	(1,[]),	(1,[]),	(1,[]),	(1,[]),	(1,[T0]),
	(2,[T1]),	(2,[T1]),	(2,[T1]),	(2,[T1,T0]),	(2,[T1,T0]),
	(3,[T1,T2]),	(3,[T1,T2]),	(3,[T1,T2,T0]),	(3,[T1,T2,T0]),	(3,[T1,T2,T0]),
	(4,[T2,T0]),	(4,[T2,T0,T1]),	(4,[T2,T0,T1]),	(4,[T2,T0,T1]),	(4,[T2,T0,T1]),
	(5,[T0,T3,T2,T1]),	(5,[T0,T3,T2,T1]),	(5,[T0,T3,T2,T1]),	(5,[T0,T3,T2,T1]),	(5,[T0,T3,T2,T1]),
	(6,[T3,T4,T2,T1]),	(6,[T3,T4,T2,T1]),	(6,[T3,T4,T2,T1]),	(6,[T3,T4,T2,T1]),	(6,[T3,T4,T2,T1]),
	(7,[T4,T2,T1]),	(7,[T4,T2,T1]),	(7,[T4,T2,T1]),	(7,[T4,T2,T1]),	(7,[T4,T2,T1]),
	(8,[T2,T1]),	(8,[T2,T1]),	(8,[T2,T1,T0]),	(8,[T2,T1,T0]),	(8,[T2,T1,T0]),
	(9,[T1,T2]),	(9,[T1,T2,T0]),	(9,[T1,T2,T0]),	(9,[T1,T2,T0]),	(9,[T1,T2,T0]),
	(10,[T2,T0]),	(10,[T2,T0]),	(10,[T2,T0]),	(10,[T2,T0]),	(10,[T2,T0,T1]),
	(11,[T2,T0]),	(11,[T2,T0]),	(11,[T2,T0]),	(11,[T2,T0,T1]),	(11,[T2,T0,T1]),
	(12,[T2]),	(12,[T2]),	(12,[T2]),	(12,[T2]),	(12,[T2]),
	(13,[])],	(13,[])],	(13,[])],	(13,[])],	(13,[])],
LiveOuts	[(0,[]),	[(0,[]),	[(0,[]),	[(0,[]),	[(0,[T0]),
	(1,[T1]),	(1,[T1]),	(1,[T1]),	(1,[T1,T0]),	(1,[T1,T0]),
	(2,[T1,T2]),	(2,[T1,T2]),	(2,[T1,T2,T0]),	(2,[T1,T2,T0]),	(2,[T1,T2,T0]),
	(3,[T1,T2]),	(3,[T1,T2,T0]),	(3,[T1,T2,T0]),	(3,[T1,T2,T0]),	(3,[T1,T2,T0]),
	(4,[T0,T3,T2,T1]),	(4,[T0,T3,T2,T1]),	(4,[T0,T3,T2,T1]),	(4,[T0,T3,T2,T1]),	(4,[T0,T3,T2,T1]),
	(5,[T3,T4,T2,T1]),	(5,[T3,T4,T2,T1]),	(5,[T3,T4,T2,T1]),	(5,[T3,T4,T2,T1]),	(5,[T3,T4,T2,T1]),
	(6,[T4,T2,T1]),	(6,[T4,T2,T1]),	(6,[T4,T2,T1]),	(6,[T4,T2,T1]),	(6,[T4,T2,T1]),
	(7,[T2,T1]),	(7,[T2,T1]),	(7,[T2,T1,T0]),	(7,[T2,T1,T0]),	(7,[T2,T1,T0]),
	(8,[T1,T2]),	(8,[T1,T2,T0]),	(8,[T1,T2,T0]),	(8,[T1,T2,T0]),	(8,[T1,T2,T0]),
	(9,[T2,T0]),	(9,[T2,T0]),	(9,[T2,T0]),	(9,[T2,T0]),	(9,[T2,T0,T1]),
	(10,[T2,T0]),	(10,[T2,T0]),	(10,[T2,T0]),	(10,[T2,T0,T1]),	(10,[T2,T0,T1]),
	(11,[T2,T0]),	(11,[T2,T0]),	(11,[T2,T0,T1]),	(11,[T2,T0,T1]),	(11,[T2,T0,T1]),
	(12,[]),	(12,[]),	(12,[]),	(12,[]),	(12,[]),
	(13,[])])	(13,[])])	(13,[])])	(13,[])])	(13,[])])

Derive interference graph from live ranges

Recall definition:

- "**x is live at p**": for some variable x and point p, the value of x could be used along some path starting at p.

- Eg: liveOut(7)= [T2,T1,T0]

  "The values of T2, T1 and T0 could be used along some path starting from 7"

- LiveOut:

  [(0,[T0]),
  (1,[T1,T0]),
  (2,[T1,T2,T0]),
  (3,[T1,T2,T0]),
  (4,[T0,T3,T2,T1]),
  (5,[T3,T4,T2,T1]),
  (6,[T4,T2,T1]),
  (7,[T2,T1,T0]),
  (8,[T1,T2,T0]),
  (9,[T2,T0,T1]),
  (10,[T2,T0,T1]),
  (11,[T2,T0,T1]),
  (12,[]),
  (13,[])])

Interference

Find overlapping live ranges

- For each temporary *t*
- For each node *id*
- If *t* is in liveOut(*id*)
- Then interferes(*t*) includes liveOut(*id*)

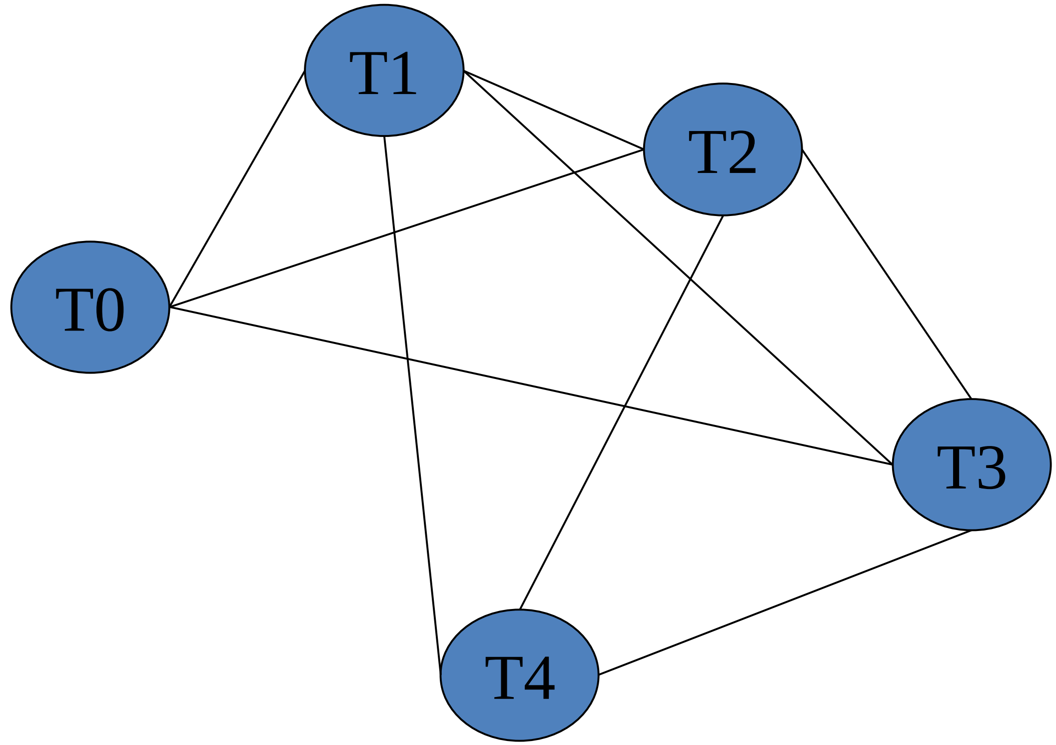
- Interference graph

interferes=

  [(T0,[T0,T1,T2,T3]),
  (T1,[T1,T0,T2,T3,T4]),
  (T2,[T1,T2,T0,T3,T4]),
  (T3,[T0,T3,T2,T1,T4]),
  (T4,[T3,T4,T2,T1])]

Derive interference graph from live ranges

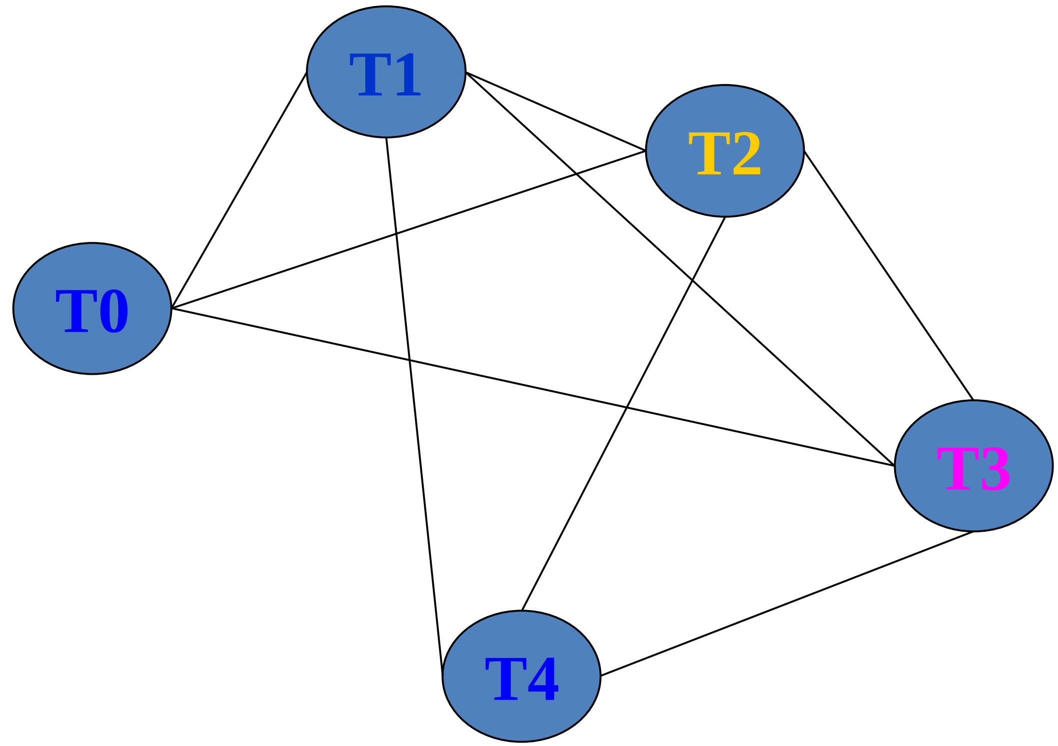
- Interference graph:
  [(T0,[T0,T1,T2,T3]),
  (T1,[T1,T0,T2,T3,T4]),
  (T2,[T1,T2,T0,T3,T4]),
  (T3,[T0,T3,T2,T1,T4]),
  (T4,[T3,T4,T2,T1])]



Use interference graph to assign temporaries

- Interference graph:

  [(T0,[T0,T1,T2,T3]),
  (T1,[T1,T0,T2,T3,T4]),
  (T2,[T1,T2,T0,T3,T4]),
  (T3,[T0,T3,T2,T1,T4]),
  (T4,[T3,T4,T2,T1])]



- Find colouring:

  [(T0,**D0**),(T1,**D1**),(T2,**D2**),(T3,**D3**),(T4,**D0**)]

Applying the colouring:

```
.data
; Integer variable a has been allocated to T0
.text
move.l #1, T0
move.l #10, T1
move.l #1, T2
bra    L2

L1:
move.l T2, T3
move.l T0, T4  (T0 & T4 assigned to D0)
mul.l  T3, T4
move.l T4, T0
add.l  #1, T2

L2:
cmp.l  T1, T2
bgt    L3
bra    L1

L3:
move.l T2, x
```

Before colouring

```
.data
; Integer variable a has been allocated to D0
.text
move.l #1, D0
move.l #10, D1
move.l #1, D2
bra    L2

L1:
move.l D2, D3
mul.l  D3, D0
add.l  #1, D2

L2:
cmp.l  D1, D2
bgt    L3
bra    L1

L3:
move.l D2, x
```

After colouring

Live variable analysis… summary

- We found we could find live ranges by constructing a system of dataflow equations and solving it by iteration
- The algorithm always terminates…
- The amount of work per iteration depends on program complexity - #instructions, #temporaries
- The number of iterations needed depends on the order in which the CFG is traversed…
  - See EaC pg445, Appel pg226, pg399
  - Live variable analysis is a *backwards* analysis – LiveIn(n) depends on its *successors*
  - Number of iterations depends on program's structural complexity – its "*loop interconnectiveness*"

APPENDIX: Liveness analysis, colouring in Haskell…

- Encode DFA equations:

```
newLiveIn liveIns liveOuts node
  = nodeUses node `union` ( (liveOutsOf node) \\ nodeDefs node )
     where
     liveOutsOf node = retrieve (nodeId node) liveOuts
newLiveOut liveIns liveOuts node
 = bigU [retrieve s liveIns | s <- nodeSuccs node]
    where bigU sets = nub (concat sets)
```

- Do one step: update LiveIn and LiveOut sets for each node:

```
updateLiveness [] (liveIns, liveOuts) = (liveIns, liveOuts)
updateLiveness (node:nodes) (liveIns, liveOuts)
 = updateLiveness nodes (newLiveIns, newLiveOuts)
   where
  newLiveIns  = subst (nodeId node) liveIns  (newLiveIn  liveIns liveOuts node)
  newLiveOuts = subst (nodeId node) liveOuts (newLiveOut newLiveIns liveOuts node)
```

Detailed code is shown in the hope that it will make the concepts clearer; please don't memorize it! Spend the time reading the textbook instead.

Solving DFAs in Haskell… (for completeness!)

- Iterate…

```
iterateUpdates nodes (liveIns, liveOuts)
 = let
    (newLiveIns, newLiveOuts) = updateLiveness nodes (liveIns, liveOuts)
   in
    if newLiveIns == liveIns && newLiveOuts == liveOuts
    then
      (newLiveIns, newLiveOuts)
    else
      iterateUpdates nodes (newLiveIns, newLiveOuts)
```

```
findLiveRanges :: CFG -> ([(Id,[Register])], [(Id,[Register])])

findLiveRanges (ControlFlowGraph cfgnodes)
  = iterateUpdates cfgnodes (initialLiveIns, initialLiveOuts)
    where
    initialLiveIns = initialLiveOuts
    initialLiveOuts = [(id,[]) | id <- map nodeId cfgnodes]
```

(live ranges liveIn & liveOut, each a mapping from node to list of temps)

(an empty list for each node)

- Now build the register interference graph (RIG):

```
buildInterferenceGraph cfg
 = [(t, nub (buildInterferenceList liveOuts t)) | t <- temporaries]
   where
   (liveIns, liveOuts) = findLiveRanges cfg
   temporaries = findTemporaries cfg

buildInterferenceList [] t = []
buildInterferenceList ( (id,livelist) : liveIns) t
  | t `elem` livelist       = livelist ++ buildInterferenceList liveIns t
  | otherwise               = buildInterferenceList liveIns t
```

(nub eliminates duplicates)

(findTemporaries lists temps used in code)

- If we assign T*i* to D*j*, will we have a conflict?

```
doesntInterfere :: (Register,Register) -> InterferenceGraph -> Bool

doesntInterfere (t,r) ifg
  = actualinterferences == []
    where
    actualinterferences = [ ai | ai <- potentialinterferences, ai == r ]
    potentialinterferences = retrieve t ifg \\ [t]
```

(retrieve finds the list corresponding to t)

(remove t itself, which also appears in list)

Solving DFAs in Haskell… (for completeness!)

- Colour the graph – find a conflict-free assignment

```
type Colouring = [(Register, Register)]    (temporary, real register)

findColouring cfg ifg
  = let temporaries = findTemporaries cfg
     in findColouring' temporaries ifg


findColouring' :: [Register] -> InterferenceGraph -> Colouring
findColouring' [] ifg  = []
findColouring' (t:ts) ifg
  = let
                                                  (theRealRegisters is [D0,D1..D31])
      possibleMappings = [(t,r) | r <- theRealRegisters]
      validMappings = [(t,r) | (t,r) <- possibleMappings, doesntInterfere (t,r) ifg]
    in
                                                  (updateIFG replaces temps with regs)
      head [ (t,r) : (findColouring' ts (updateIFG ifg (t,r))) | (t,r) <- validMappings ]
```

- If no colouring can be found, this function fails (the list above is empty). If this happens, we will have to “spill” one of the variables to memory and try again.
- This is a quick and dirty but dumb inefficient algorithm; see Appel pg239

Solving DFAs in Haskell… (for completeness!)

- Put it all together…

```
applyColouring :: [Instruction] -> [Instruction]

applyColouring code
  = let
      cfg = buildCFG code
      colouring = findColouring cfg (buildInterferenceGraph cfg)
    in
      map (replaceTemporaries colouring) code
```

(where "`replaceTemporaries colouring instruction`" updates the instruction to use the specified real registers instead of temporaries)