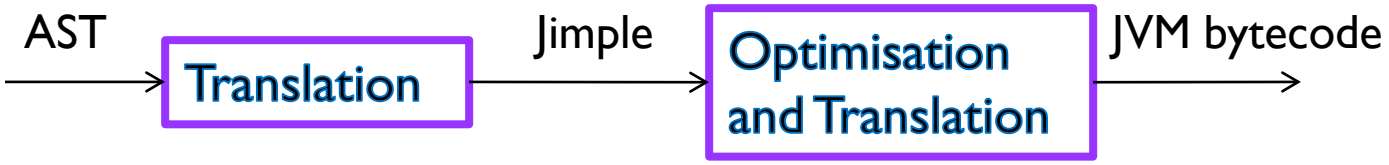
3. Code Generation using Soot

Compiling to Bytecode using Soot

- Our goal is to describe compilation from a high-level (Java-like) language to JVM bytecode
- However, we do not show how to generate bytecode directly; instead, we show how to generate Jimple, and then rely on Soot to generate bytecode for us
- There are three major advantages to this scheme:
  1. Jimple is (slightly) more high-level than raw bytecode, and hence is easier to generate
  2. It is easier to write optimisations at the Jimple level
  3. We can reuse optimisations already available in Soot



Compiling to Jimple

- For code generation, there are two problems to solve:
  1. How to map the data types and data abstractions of the source language to the target language
  2. How to map the code abstractions of the source language to the target language
- Soot provides the same high-level data abstractions (classes, objects and arrays) as the JVM, so the mapping is straightforward; this is more complicated for native code
- But we still need to map code abstractions:
  1. Need to compile nested expressions to Jimple's 3-address representation
  2. Need to compile structured control flow to Jimple's conditional jumps

Compiling Expressions

- In all modern high-level programming languages, expressions can be nested:

```
b*b-4*a*c
```

- In native code and JVM bytecode, however, every instruction only performs a single operation, and instructions do not nest
- Thus we need to introduce temporary variables to hold intermediate results
- To do this in a systematic fashion, expressions are translated to Jimple `Values` recursively
  - Some examples of Jimple `Values` are `Constants`, or subclasses of `Expr` such as `AddExpr`, which represents the addition of two `Values`.

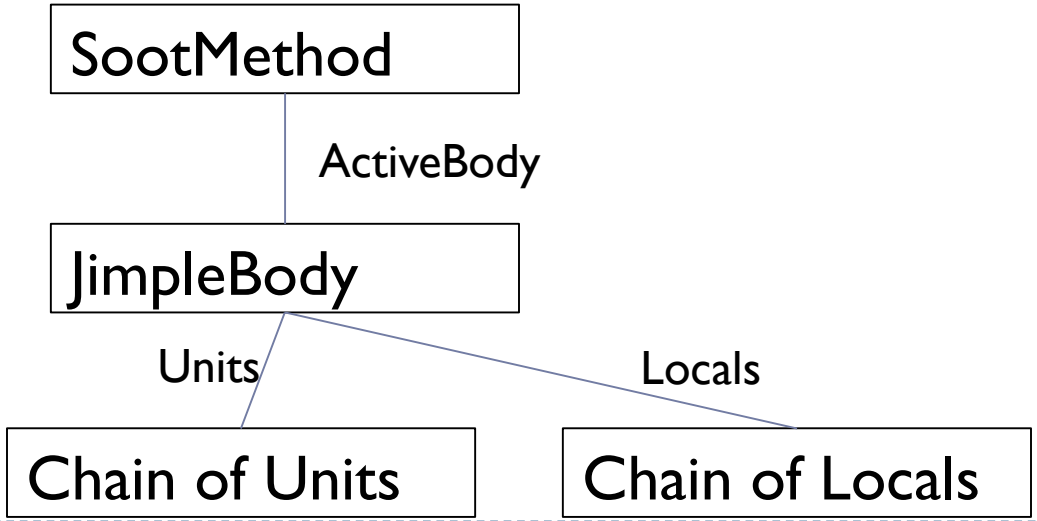
Generating Code for Literals

- Literals map directly to class `soot.jimple.Constant` and its subclasses:
  - `NullConstant`
  - `IntConstant`, `LongConstant`
  - `FloatConstant`, `DoubleConstant`
  - `StringConstant`
- As usual with Soot, these classes cannot be instantiated directly; instead, they have a factory method `v` that is used to obtain new instances:

```
IntConstant.v(23)
```

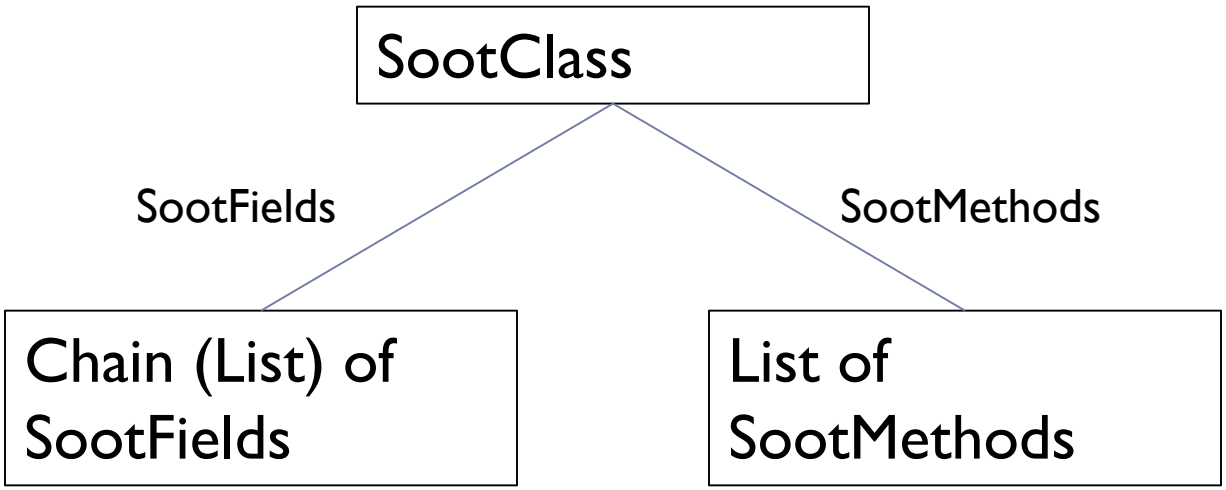
Generating Code for Variable Names

- For a given name x, we first use name analysis to determine whether it refers to a local variable/parameter or to a field
  - The code generator maintains a mapping from local variables and parameters to Soot locals (class `soot.Local`): when a variable declaration is encountered, a new Soot local is created for it and added to the enclosing method body; when the variable is used, the code generator looks up the associated Soot local and returns a reference to it



Generating Code for Variable Names

- If the given name x refers to a field, the code generator emits a field reference (class `soot.jimple.FieldRef`), which specifies the declaring class, the field name, and its type

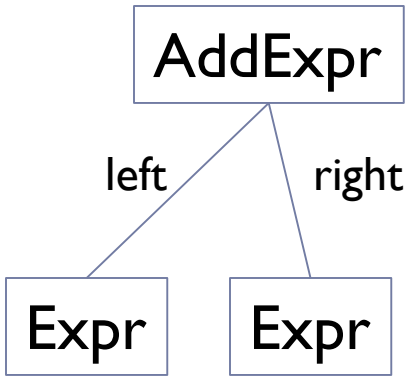


Generating Code for Binary Operators

- For a binary operator such as "+", we first recursively generate code for the left and the right operand, resulting in two Jimple values *left* and *right*
- Jimple provides the same expression types as Java, e.g. `AddExpr` for "+", with factory method `Jimple.newAddExpr`
- Note: Operands of a BinopExpr such as `AddExpr` can only be constants or local variables (3-address form!), so if left and right are complex expressions, we first generate the following two assignment instructions:

```
tleft = left expr
tright = right expr
```

(tleft and tright are temporary variables)



- Now the whole expression can be compiled to

```
tleft + tright
```

Generating Code for Binary Operators (2)

- Unlike bytecode, Jimple allows operands of arithmetic operators to have different types; Soot will generate the appropriate conversion operations when producing bytecode
- Code generation for most other arithmetic and logical operators (both unary and binary) works the same:
  1. Recursively generate code for operands
  2. If an operand is a complex expression, introduce a fresh temporary variable and generate an assignment to store the operand into the temporary
  3. Construct appropriate Jimple expression
- Short-circuiting operators are an exception: they are really just a shorthand for an **if** statement, so they must be compiled using conditional jumps (see below), e.g. if (i < n && a[i] != 0)

Code Generation for Assignments

- Assignment expressions correspond quite straightforwardly to Jimple's assignment instruction
- Again, both left and right hand side are recursively compiled into a Jimple expression
- The left hand side should either be a local variable or a reference to a field or array element:
  - In the former case, the right hand side can be a Jimple expression, so does not need to be stored in a temporary
  - In the latter case, however, a temporary needs to be introduced as before if the right hand side is an expression:

```
tright = right expr
a[i] = tright
```

Code Generation for Method Calls

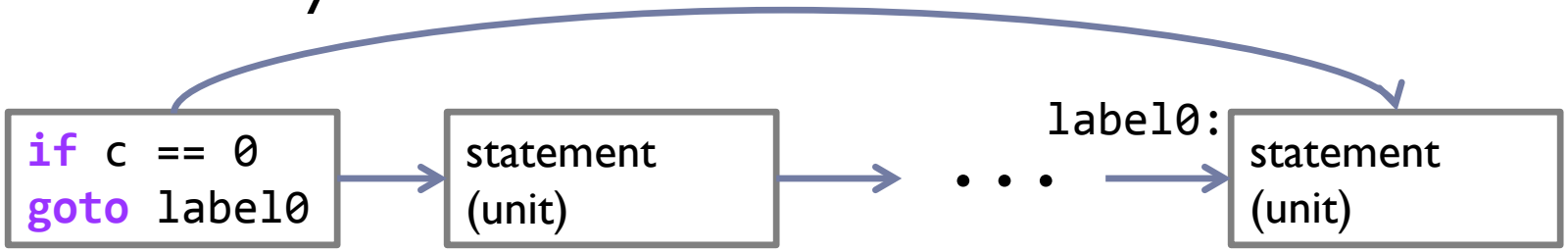
- To generate code for a method call, we use either `Jimple.newStaticInvokeExpr` or `Jimple.newVirtualInvokeExpr`, depending on whether the callee is static or not
- Both expect a list of argument values (which can be generated recursively), and a `SootMethodRef` identifying the callee
- To build the `SootMethodRef`, use name analysis to find the callee's declaration, then use `Scene.makeMethodRef`
- If the callee has return type `void`, the call must be wrapped into an invocation statement (`soot.jimple.InvokeStmt`)
- Otherwise, we introduce a fresh temporary variable into which to store the result

Code Generation for Statements

- Code for statements is again generated recursively, by first generating code for child expressions (or statements), e.g. for an **if** statement, we generate code for the condition, then the **then** part of the statement, then the **else** part of the statement
- Unlike for expressions, code generation for statements does not return any value, it simply adds the generated code to the list of statements of the enclosing `JimpleBody`
- A statement is represented by a unit in Jimple. Suppose we are generating a new statement and units is the list of statements generated so far:
  - Unit stmt = *statement* ; // generate code for current statement
  - units.add(stmt); // add this statement to list

Code Generation for Jump Statements

- Jump statements need to refer to other statements; in bytecode this is done by specifying the offset of the other statement; in Jimple, we can simply use a reference to the Java object representing the statement which is the target of the jump
- When Soot pretty-prints Jimple code, it introduces labels to refer to statements, but these labels do not really exist: they are simply a way to print the statement references in a human-readable way



Code Generation for If Statements

- When generating code for an **if** statement, we generate the code for the conditional expression and assign the result to an object c of type Value
- If the condition is false, we should skip the **then** part, so we need to generate the target statement for the branching

Basic idea:

```
if(cond)
        stmt1
else
        stmt2
```

```
        // compute value of cond
        c = …
        if c == 0 goto label0
        // code for stmt1
        goto label1
label0:
        // code for stmt2
label1:
```

Code Generation for If Statements (2)

- However, when generating the code for the jump, we do not know which statement should be our target statement (i.e. which unit is the statement after the **then** part)

Problem:

When we generate this statement, we do not yet know what unit is at `label0`!

(Same problem for `label1`)

```
    // compute value of cond
    c = …
    if c == 0 goto label0
    // code for stmt1
    goto label1
label0:
    // code for stmt2
label1:
```

Code Generation for If Statements (3)

Possible Solutions:

1. **Backpatching**: Put in **null** references for the jump targets; overwrite once we know the right values
2. **NOP Padding**: Generate two NOP instructions as jump targets, and insert them in the right positions

```
    // compute value of cond
    c = …
    if c == 0 goto label0
    // code for stmt1
    goto label1
label0:
    // code for stmt2
label1:
```

With backpatching, if we have many statements in the **then** part, we would need to overwrite the null reference in a jump that was generated a long time ago

Code Generation for If Statements (4)

Possible Solutions:

1. **Backpatching**: Put in **null** references for the jump targets; overwrite once we know the right values
2. **NOP Padding**: Generate two NOP instructions as jump targets, and insert them in the right positions (Soot eliminates NOP when generating bytecode)

```
    // compute value of cond
    c = …
    if c == 0 goto label0
    // code for stmt1
    goto label1
label0:
    // code for stmt2
label1:
```

```
NopStmt label0 =
    j.newNopStmt();
Unit stmt =
    j.newIfStmt(..., label0);
units.add(stmt);
```

Code Generation for If Statements (5)

INPUT: if statement

`label0` = new **NOP** statement
`label1` = new **NOP** statement
`c` = generate code for if condition (and store in temporary variable if condition is a complex expression)
emit statement `if c == 0 goto label0`
generate code for then branch
emit statement `goto label1`
emit statement `label0`
generate code for else branch
emit statement `label1`

Example: If Statement – Lab 4

```
public Void visitIfStmt(IfStmt nd) {
        Value cond = ExprCodeGenerator.generate(nd.getExpr(), fcg);
       NopStmt join = j.newNopStmt();
       units.add(j.newIfStmt(j.newEqExpr(cond, IntConstant.v(0)), join));
       nd.getThen().accept(this);
       if(nd.hasElse()) {
              NopStmt els = join;
              join = j.newNopStmt();
              units.add(j.newGotoStmt(join));
              units.add(els);
              nd.getElse().accept(this);
       }
       units.add(join);
}
```

Code Generation for While Statements

Basic idea:

```
while(cond)
        body
```

Same problem as before with `label1`; can be solved in the same two ways

```
label0:
   // compute value of cond
   c = …
   if c == 0 goto label1
   // code for body
   goto label0
label1:
```

Code Generation for While Statements (2)

The condition may be a complex expression which is translated recursively

It may involve assignments to temporary variables at different levels of recursion

It is not so easy to determine the unit that is the start of the condition

It is therefore convenient to generate a NOP instruction representing `label0`

```
label0:
   // compute value of cond
   c = …
   if c == 0 goto label1
   // code for body
   goto label0
label1:
```

```
NopStmt label0 =
    j.newNopStmt();
units.add(label0);
```

Code Generation for While Statements (3)

The jump targets `label0` and `label1` are also important for another reason:

A break statement within the body should be translated into a jump to `label1`, while a continue statement becomes a jump to `label0`

As while loops may be nested, we keep a mapping from while loops to their corresponding break targets (and similarly for continue targets)

```
label0:
    // compute value of cond
    c = …
    if c == 0 goto label1
    // code for body
    goto label0
label1:
```

Code Generation for Other Statements

- Expression statements: simply generate code for the expression
- Block statements: generate code for each statement in the block in sequence
- For loops and do-while loops are translated similarly to while statements
- Switch statements can be flattened into a nested if statement; alternatively, Jimple provides special instructions for implementing **switch**, which we do not discuss here
- Return statements are translated in one of two ways: for `return expr`, there is `Jimple.newReturnStmt`; for returning without a value: `Jimple.newReturnVoidStmt`
- There are also instructions for generating code for exception throwing and handling, which we do not discuss here

From Jimple to JVM Bytecode

- Although we can rely on Soot to generate JVM bytecode for us, it is useful to have an overview of how this is done
- There are four steps necessary for this conversion:

1. Perform a tree traversal of the Jimple code, directly translating it to naïve Baf stack machine code
2. Optimise the naïve Baf by eliminating redundant store/load instructions
3. Pack local variables to minimize the number of slots used
4. Translate the Baf code directly to JVM bytecode, calculating the maximum height of the operand stack for each method

Direct Translation to Naïve Baf

- We will use an idealized stack machine code to illustrate the method
- Direct translation

```
x = y op z;
```

3-address code

=>

```
load y
load z
op
store x
```

- Example

```
x = (y * z)/2;
```

Java

=>

```
$i0 = y * z;
x = $i0 / 2;
```

3-address code

=>

```
load y
load z
mul
store $i0
```

```
load $i0
load 2
div
store x
```

Optimising Naïve Stack Code

- Eliminate redundant store/load instructions – there are two patterns we can eliminate:
  - store/load: a store instruction followed by a load instruction referring to the same local variable *with no other uses* – both the store and load instructions can be eliminated, and the value will simply remain on the stack
  - store/load/load: a store instruction followed by two load instructions, all referring to the same local variable *with no other uses* – the three instructions can be eliminated and a dup instruction introduced to replace the second load by duplicating the value left on the stack after eliminating the store and the first load

Optimising Naïve Stack Code (2)

- Eliminating redundant patterns is trivial when the relevant instructions directly follow each other
- If there are intermediate instructions then some care must be taken to eliminate the pair safely
  - We compute the net stack height variation (nshv) and the minimum stack height variation (mshv) for these intermediate instructions to see if they form a level sequence
  - Note that in calculating mshv a binary operator such as mul should be regarded as first reducing the stack height by 2 before increasing it by 1 (so the net effect is -1)
  - Only if nshv and mshv are both equal to zero can we eliminate the pair (or triple)
  - If they are not zero, it may be possible to reorder the stack machine instructions to make them zero

Optimising Naïve Stack Code (3)

- Patterns might be interleaved with each other, so we need to iterate until no more can be eliminated

- Example

```
r = b*b - 4*a*c;
```

Java

=>

```
$i0 = b * b;
$i1 = 4 * a;
$i2 = $i1 * c;
r = $i0 - $i2;
```

3-address code

=>

Instr	shv
load b load b mul store $i0	
load 4 load a; mul store $i1	+1 +2 +1 0
load $i1 load c mul store $i2	+1 +2 +1 0
load $i0 load $i2 sub store r	

(shv = stack height variation)

Optimising Naïve Stack Code (4)

- The instructions between store `$i0` and load `$i0` have nshv = 0 and mshv = 0 so this store/load pair can be eliminated

Instr	shv

`load b`
`load b`
`mul`
~~`store $i0`~~	

`load 4`
`load a;`
`mul`
`store $i1`
	+1
	+2
	+1
	0

`load $i1`
`load c`
`mul`
`store $i2`
	+1
	+2
	+1
	0

~~`load $i0`~~
`load $i2`
`sub`
`store r`	

(shv = stack height variation)

Optimising Naïve Stack Code (4)

- The instructions between store `$i0` and load `$i0` have nshv = 0 and mshv = 0 so this store/load pair can be eliminated
- load `$i1` directly follows store `$i1`, so this pair can be eliminated

Instr	shv

`load b`
`load b`
`mul`	

`load 4`
`load a;`
`mul`
~~`store $i1`~~
	+1
	+2
	+1
	0

~~`load $i1`~~
`load c`
`mul`
`store $i2`
	+1
	+2
	+1
	0

`load $i2`
`sub`
`store r`	

(shv = stack height variation)

Optimising Naïve Stack Code (4)

- The instructions between store `$i0` and load `$i0` have nshv = 0 and mshv = 0 so this store/load pair can be eliminated
- load `$i1` directly follows store `$i1`, so this pair can be eliminated
- After eliminating store `$i0` and load `$i0`, load `$i2` directly follows store `$i2`, so this pair can be eliminated

Instr	shv

`load b`
`load b`
`mul`	

`load 4`
`load a;`
`mul`
	`+1`
	`+2`
	`+1`

`load c`
`mul`
~~`store $i2`~~
	`+2`
	`+1`
	`0`

~~`load $i2`~~
`sub`
`store r`	

(shv = stack height variation)

Optimising Naïve Stack Code (4)

- The instructions between store `$i0` and load `$i0` have nshv = 0 and mshv = 0 so this store/load pair can be eliminated
- load `$i1` directly follows store `$i1`, so this pair can be eliminated
- After eliminating store `$i0` and load `$i0`, load `$i2` directly follows store `$i2`, so this pair can be eliminated
- This gives the final optimised stack machine code as shown

Instr	shv

`load b`
`load b`
`mul`	

`load 4`
`load a;`
`mul`
	`+1`
	`+2`
	`+1`

`load c` `mul`	`+2` `+1`
`sub` `store r`	

(shv = stack height variation)

Packing Local Variables and Translation

- Pack local variables to minimize the number of local slots used
  - Local variables whose lifespans do not overlap may share the same slot
  - This is similar to register allocation and uses an interference graph and graph colouring algorithm (see later)
  - Two packings are performed, one for 32-bit variable slots and another for 64-bit variable slots
- Translate the Baf code directly to JVM bytecode
  - JVM requires that the maximum operand stack height be specified for each method
  - This can be computed by performing a simple traversal of the stack machine code and recording the effect that each instruction has on the stack height

4. Code Generation for Physical Architectures

Native Code Generation

- Compilation is not over after the class file has been built.
- "Hotspots" in the code are compiled into native code by the JVM.
- Compiling JVM code to native code involves:
  - Register allocation: native instructions often require that operands be in registers
  - Optimisation: to make native code run even faster
  - Instruction selection: to choose and generate the appropriate native code instructions
  - Runtime support: for example, support for creating new objects on the heap

Challenges

- Native instruction sets are more low-level than bytecode instruction sets; for instance, there is no built-in parameter passing and return mechanism
- There is no a priori distinction between constant pool, stack and heap, and no typing: the memory is simply a sequence of bytes
- No automated garbage collection or range checks for array accesses: this has to be implemented by the runtime library
- There is a (small) number of *registers*, which are fixed-size storage units that allow very fast access
- Whenever possible, we want to make instructions work on registers (as opposed to main memory)

Register Allocation

- Ideally, we would like to keep as many local variables as possible in registers (as opposed to main memory)
- But of course there can be an arbitrary number of local variables, while the number of registers is typically small
- Sometimes the value of a local variable is no longer needed, then its register can be reused
- If the value in a register has to be saved to memory (in order to be reloaded later), this is known as *register spilling*
- The goal of register allocation is to allocate registers to local variables in such a way that the need for register spilling is minimised

Register Allocation Example

- Consider the following code snippet, where x, y and z are locals:

```
x = 23
y = 42
z = x + y
y = y + z
```

- Assume we have only two registers r and s
- What happens if we allocate register r to both x and y?
  How about x and z?
  And y and z?

Liveness

- We call a local variable *live* at a program point if its value may be read before it is reassigned; otherwise we call it *dead*
- Live variables in our example:

```
// x, y, z all dead
x = 23
// x live; y, z dead
y = 42
// x, y live; z dead
z = x + y
// y, z live; x dead
y = y + z
// x, y, z all dead
```

Liveness and Conditionals

- More elaborate example; comments indicate variables that are live *before* the *following* statement

```
    x = 0                    // x
    y = 1                    // x, y
l0: z = 2                    // x, y, z
    a = x                    // a, x, y, z
l1: if z == y goto l3        // a, x, y, z
    b = z * 2                // a, b, x, y, z
    if b > y goto l2         // a, x, y, z
    a = a * a                // a, x, y, z
    z = z * 2                // a, x, y, z
    goto l1
l2: a = y * x                // a, x, y, z
    z = z + 1                // a, x, y, z
    goto l1
l3:
```

Liveness and Register Allocation

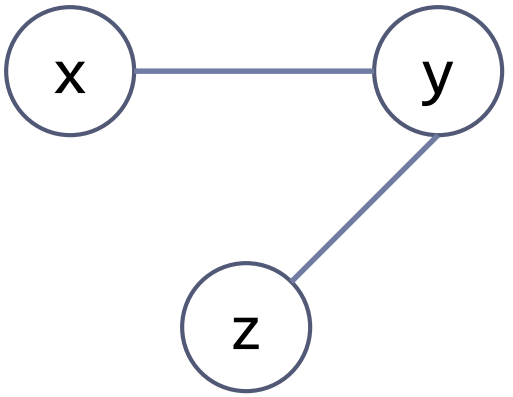
- We cannot allocate the same register to two variables that are live at the same point in the program
- Conversely: variables x and y can share the same register if there is no point in the program where both x and y are live
- This information can be captured in an *interference graph*:
  - One node for each local variable
  - Undirected edge between nodes for x and y if they are both live at some point in the program

Interference Graph Examples

Program:

```
x = 23
y = 42
z = x + y
y = y + z
```

Interference graph:



So x and z can be allocated the same register, but not x and y, or y and z

Graph Colouring Register Allocation

- A *graph colouring* is an assignment of colours to the nodes of a graph such that there is no edge between nodes with the same colour
- A *k-colouring* is a graph colouring with k different colours, and a graph is called *k-colourable* if it has a k-colouring
- Basic insight: If the interference graph is k-colourable, then there is a register assignment that uses no more than k registers such that no register spilling is necessary
- Unfortunately, graph colouring is a hard problem

Chaitin's Algorithm

- Chaitin's algorithm is a good heuristic algorithm for finding k-colourings (and hence for register allocation)
- It is based on two basic insights:
  1. If we have k colours, and if there is some node n in the graph with less than k neighbours, then the whole graph is k-colourable if the graph without n is k-colourable – so we can simplify the problem by disregarding n
  2. If every node in the graph has at least k neighbours, we need to select a spill candidate, i.e. a node that will not be assigned a register – it is generally a good idea to choose a spill candidate that has the maximum number of neighbours: throwing it out will do the most to simplify the remaining colouring problem

Chaitin's Algorithm: Pseudocode

INPUT: Interference graph IG, number k of registers

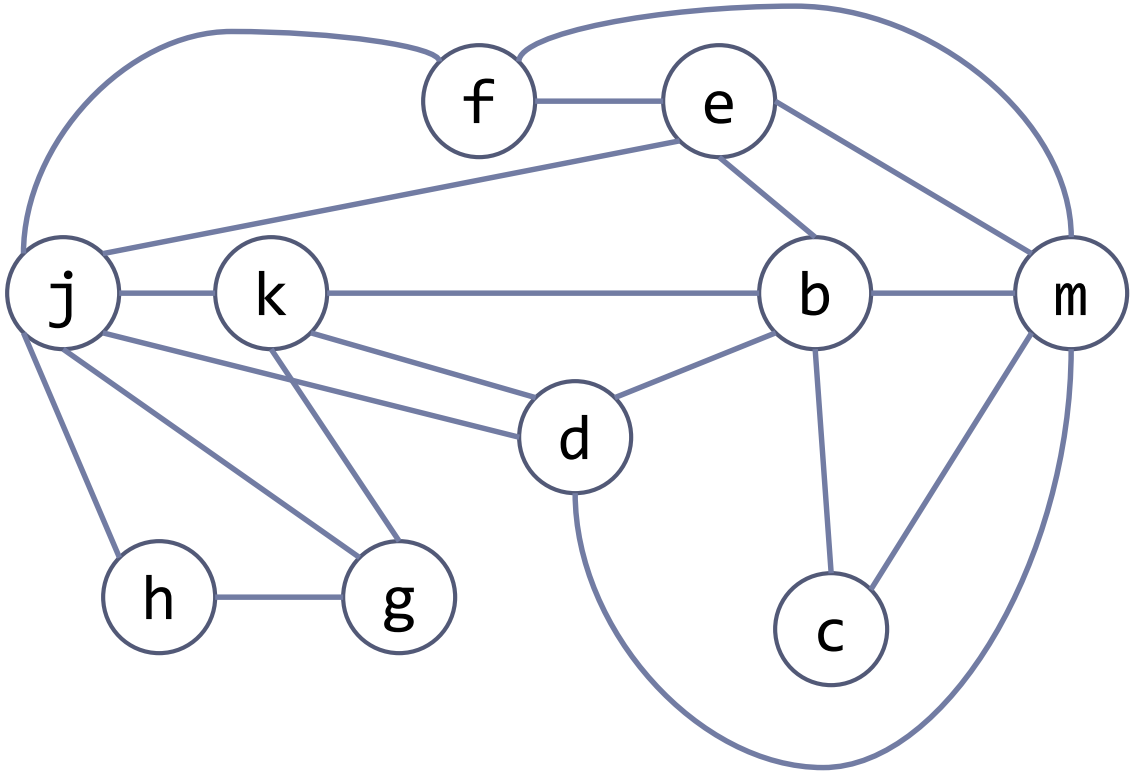
s = empty stack

while IG not empty **do**
  if there is node n with neighbours(n) < k **then**
    remove n from IG and push it onto s
  else
    let d be node with maximum number of neighbours
    remove d from IG, mark as spill candidate and push it onto s

while s not empty do
  pop node n off s
  if n marked and neighbours(n) have k different colours **then**
    do not allocate a register for n
  else
    allocate n a register not allocated to any of neighbours(n)

Example (from Appel's textbook)

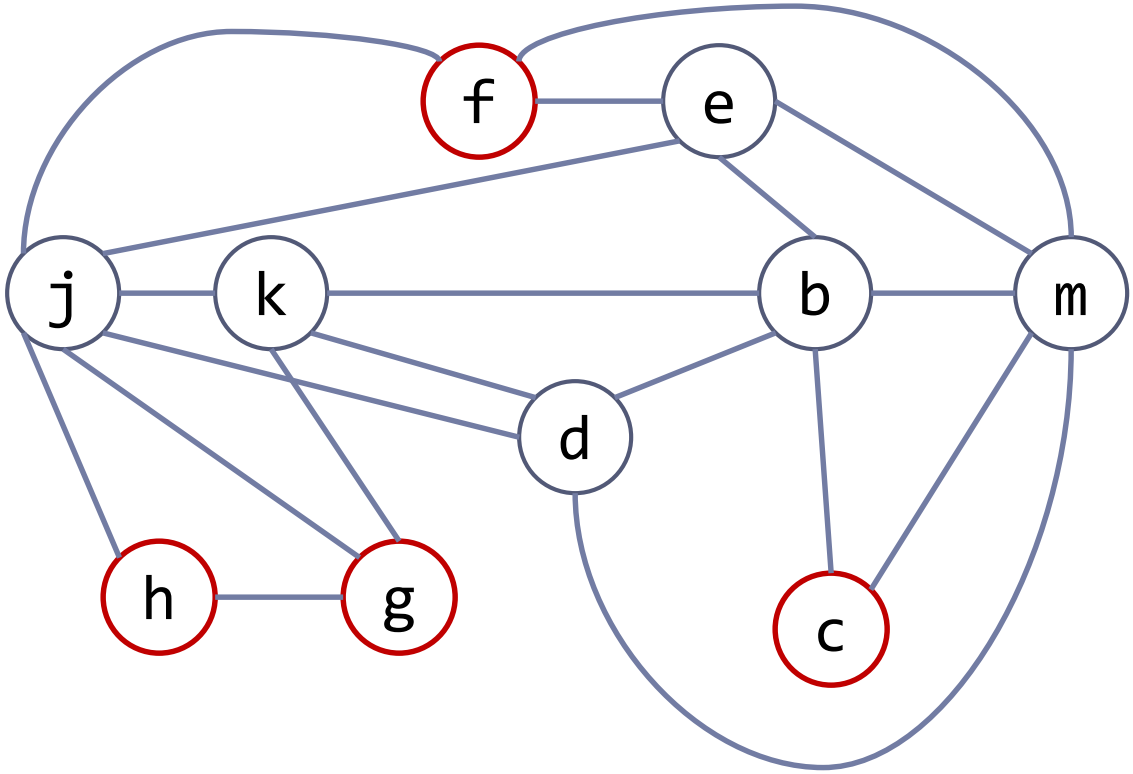
Assume we have the following interference graph:



Can we allocate registers to the variables using four registers r, s, t, u?

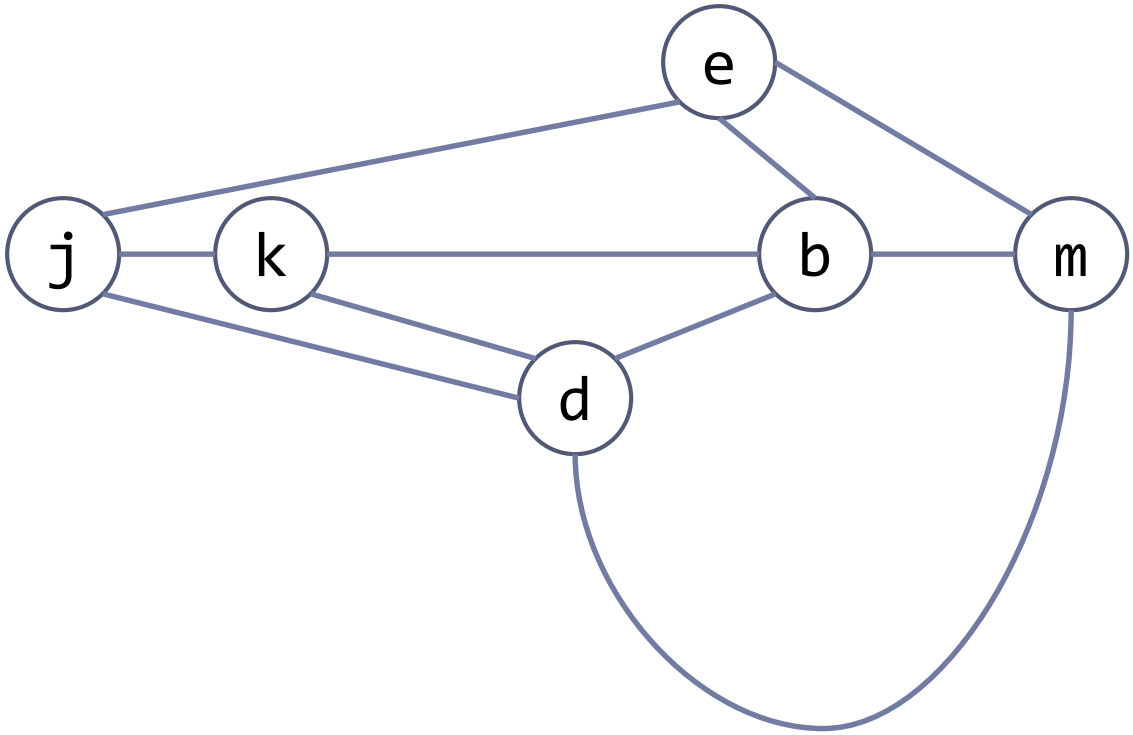
Example (from Appel's textbook)

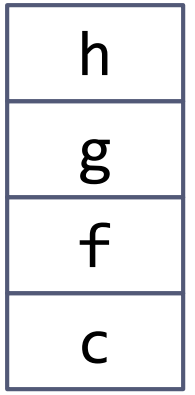
First step: find nodes with less than four neighbours



Example (from Appel's textbook)

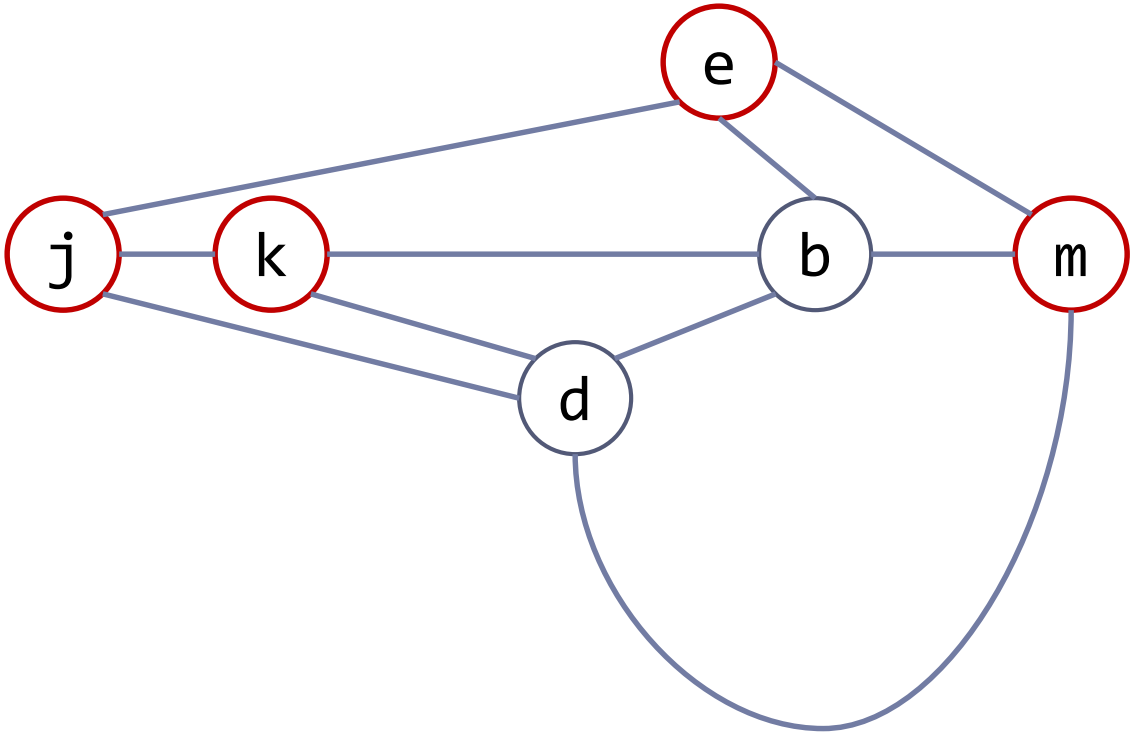
Remove these nodes and push them onto stack





Example (from Appel's textbook)

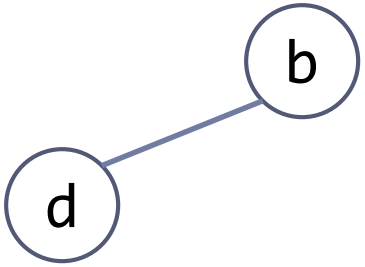
Find nodes with less than four neighbours

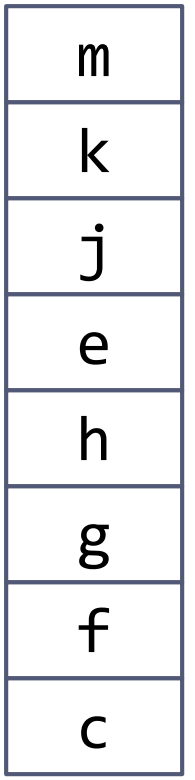


h
g
f
c

Example (from Appel's textbook)

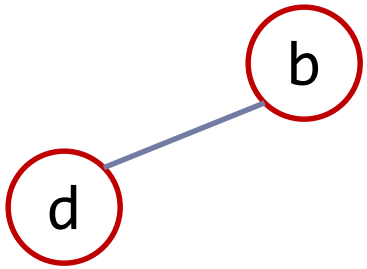
Remove these nodes and push them onto stack





Example (from Appel's textbook)

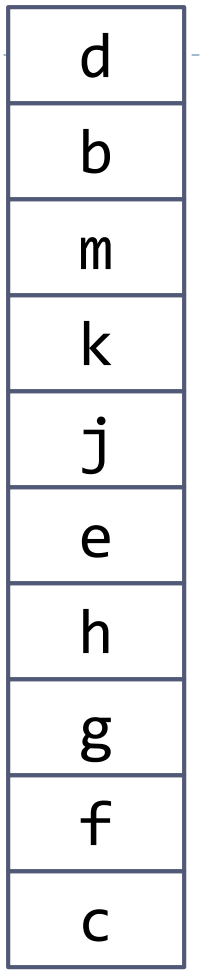
Find nodes with less than four neighbours



m
k
j
e
h
g
f
c

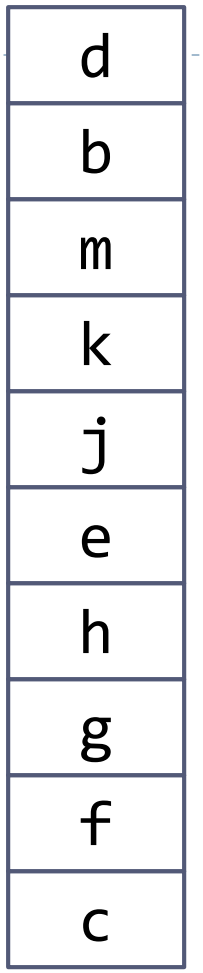
Example (from Appel's textbook)

Remove these nodes and push them onto stack



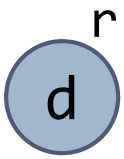
Example (from Appel's textbook)

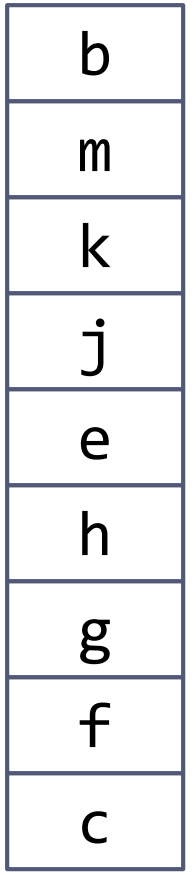
No nodes left, so pop nodes off the stack one by one, and assign a "colour" (register)



Example (from Appel's textbook)

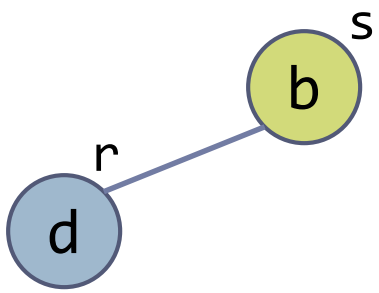
Pop off d, assign arbitrary register, e.g. r





Example (from Appel's textbook)

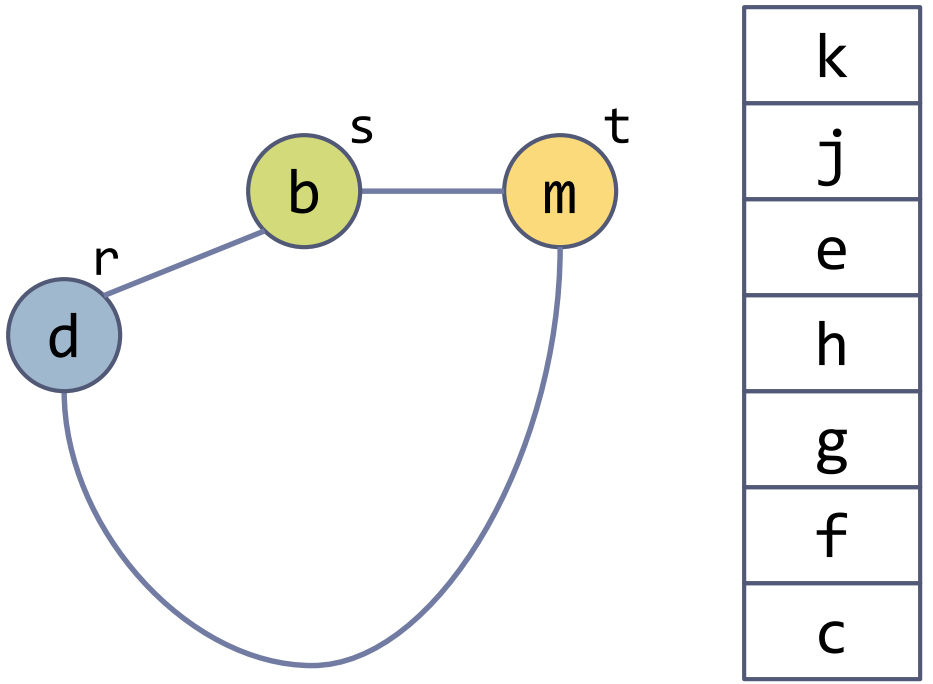
Pop off b, assign register other than r, e.g. s



m
k
j
e
h
g
f
c

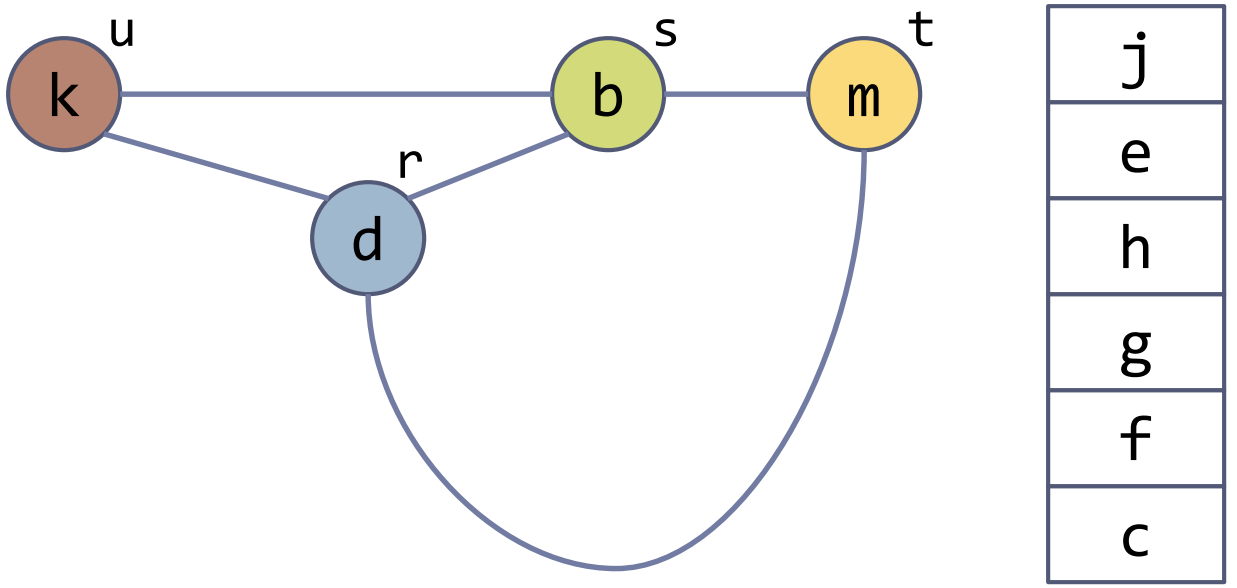
Example (from Appel's textbook)

Pop off m, assign register other than r and s, e.g. t



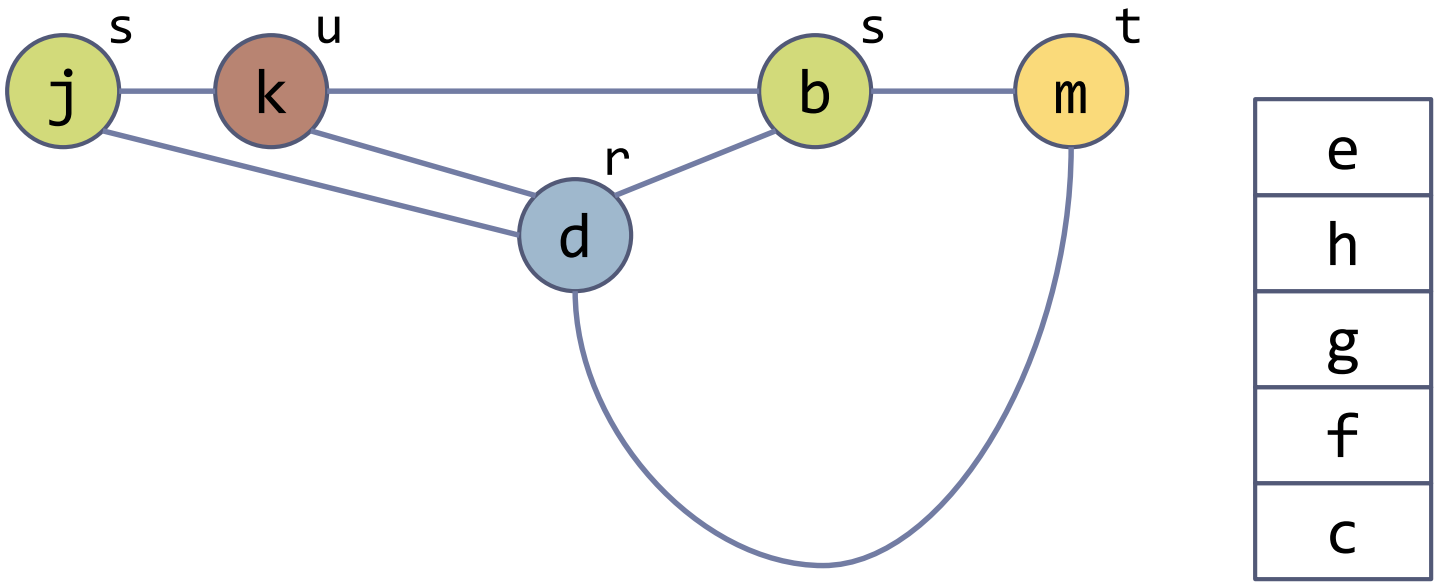
Example (from Appel's textbook)

Pop off k, assign register other than r, s, e.g. u



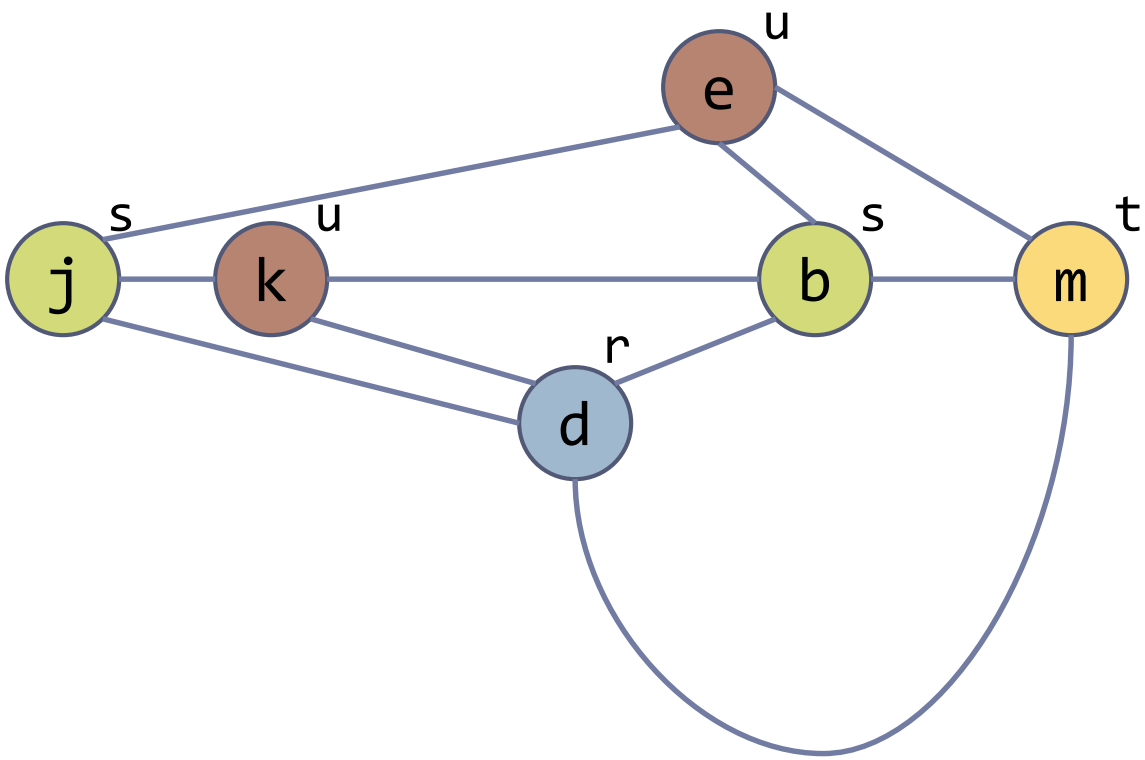
Example (from Appel's textbook)

Pop off j, assign register other than u and r, e.g. s



Example (from Appel's textbook)

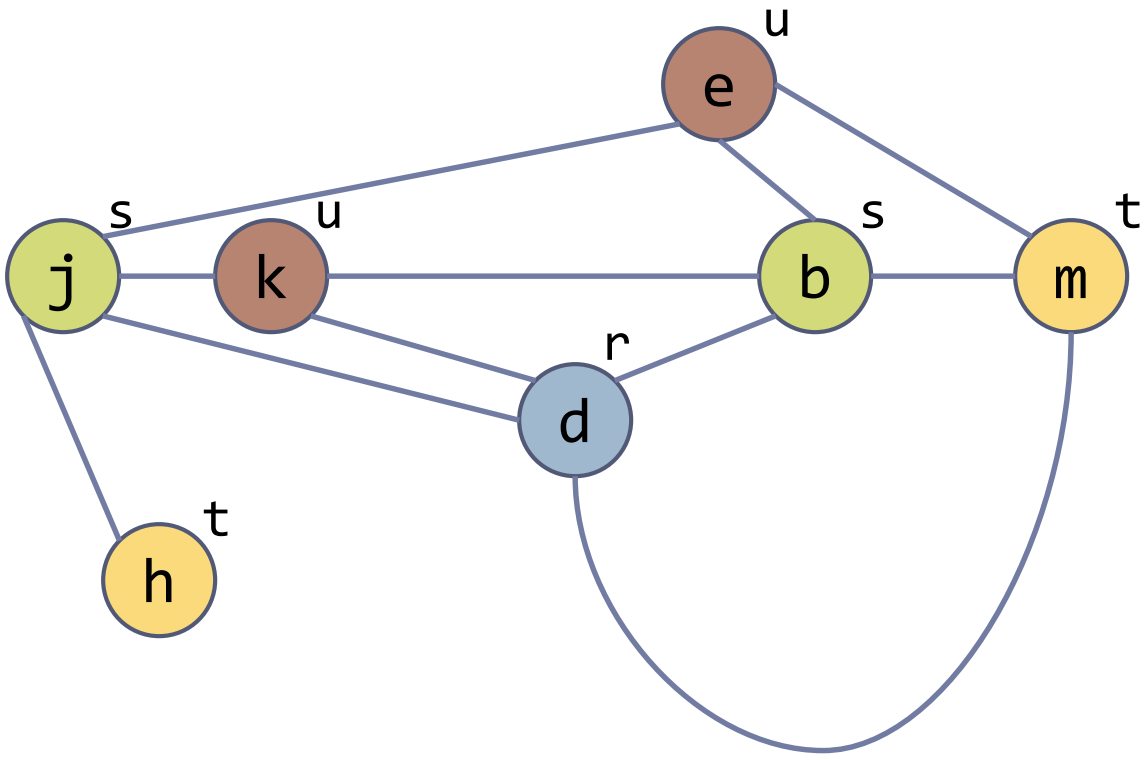
Pop off e, assign register other than s and t, e.g. u



h
g
f
c

Example (from Appel's textbook)

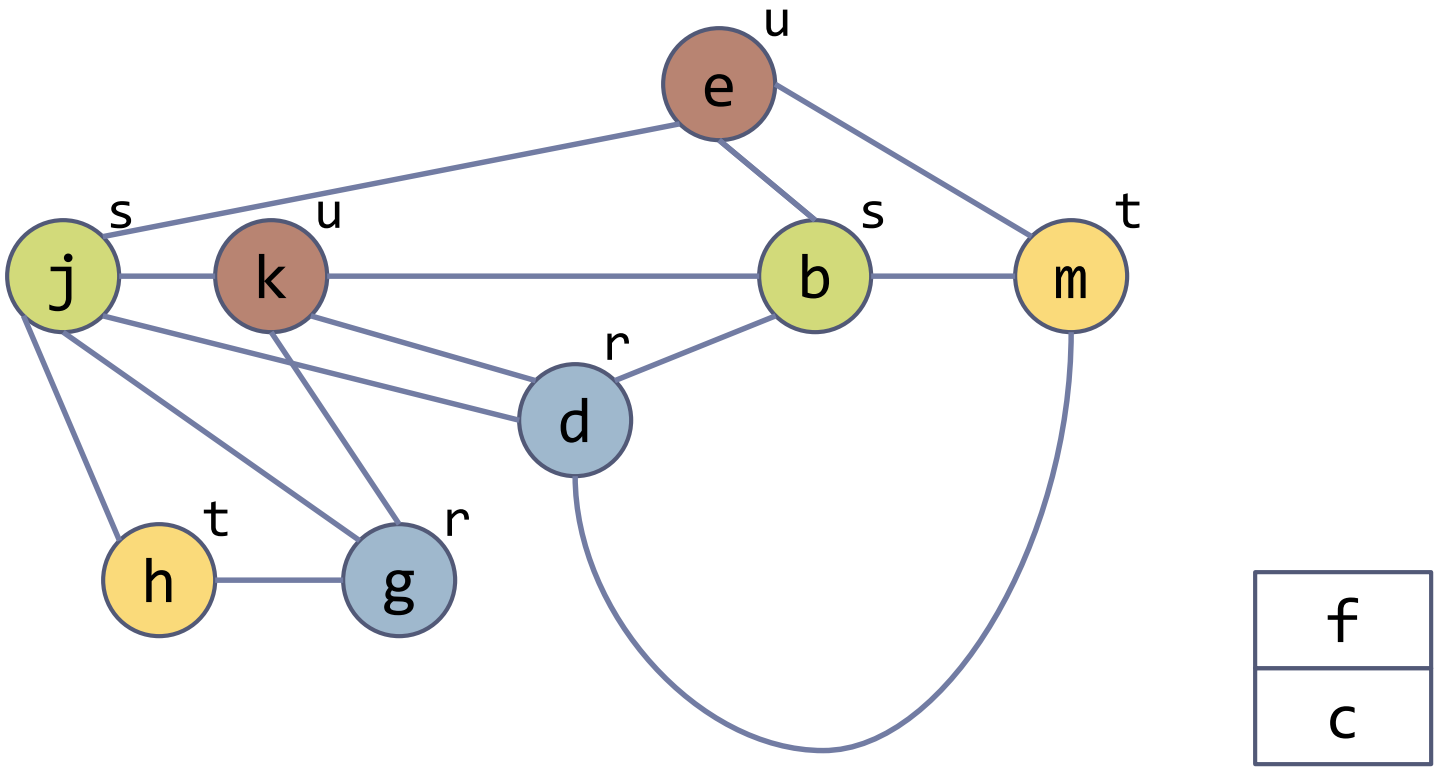
Pop off h, assign register other than s, e.g. t



g
f
c

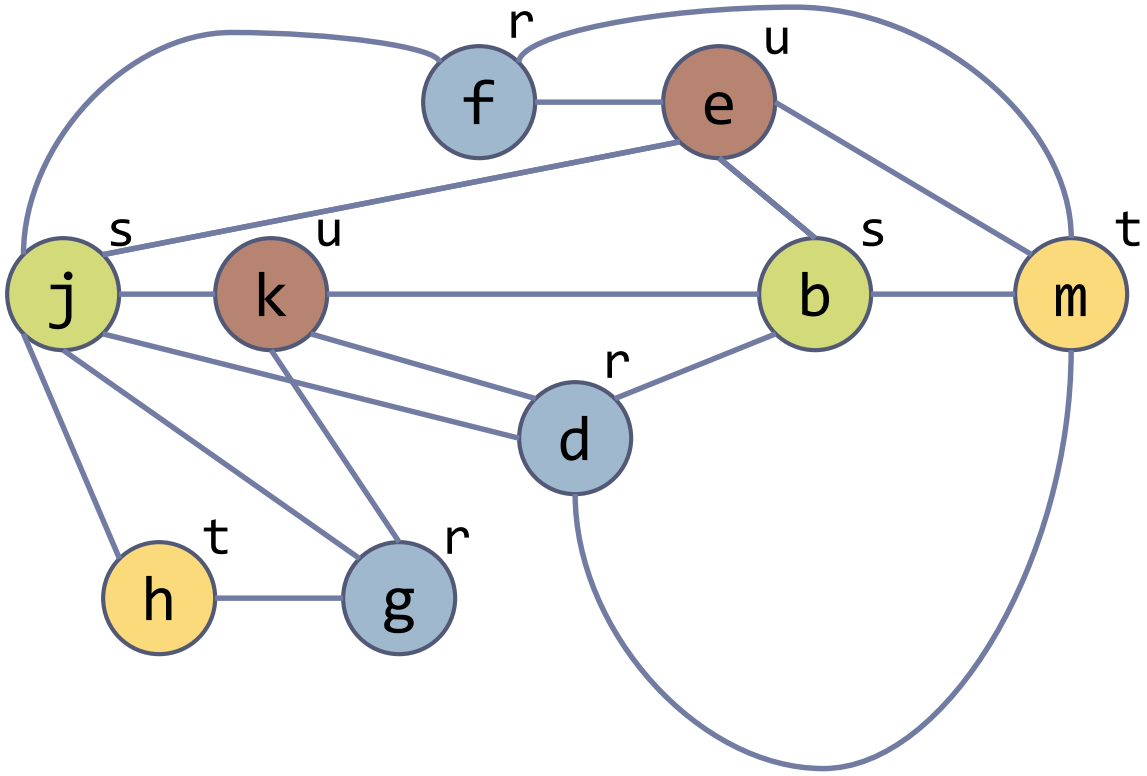
Example (from Appel's textbook)

Pop off g, assign register other than s, t, and u; i.e. r



Example (from Appel's textbook)

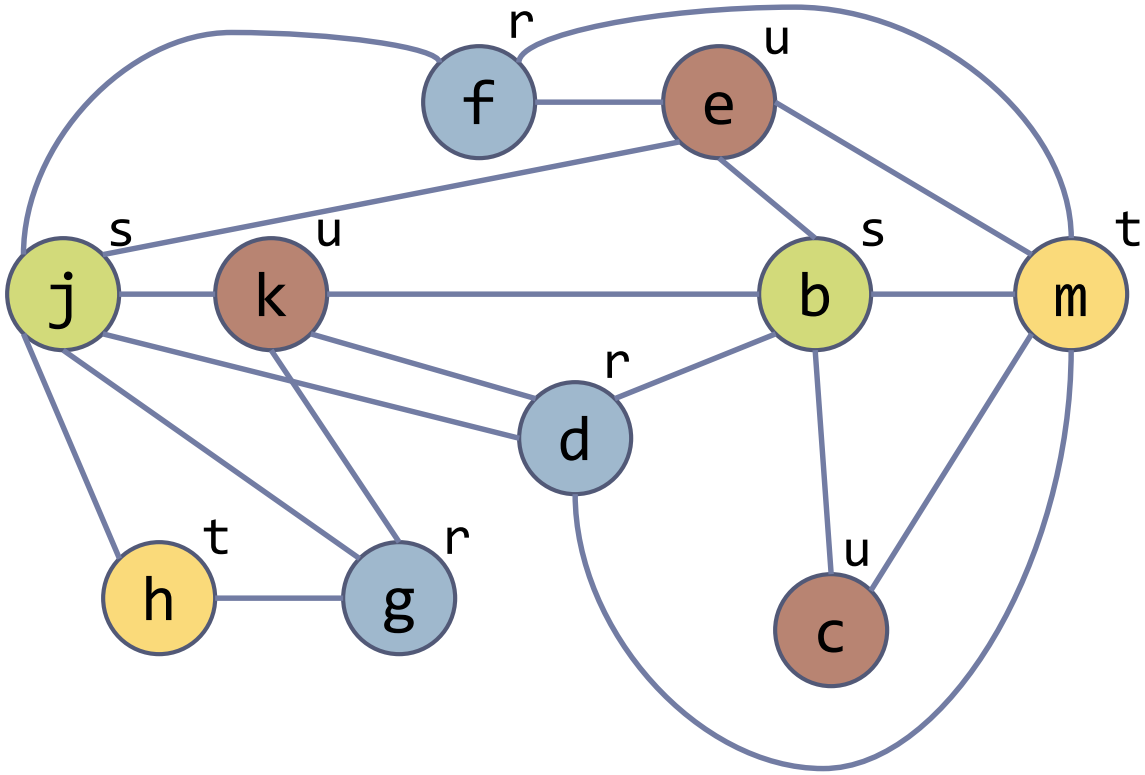
Pop off f, assign register other than s, t, and u; i.e. r



c

Example (from Appel's textbook)

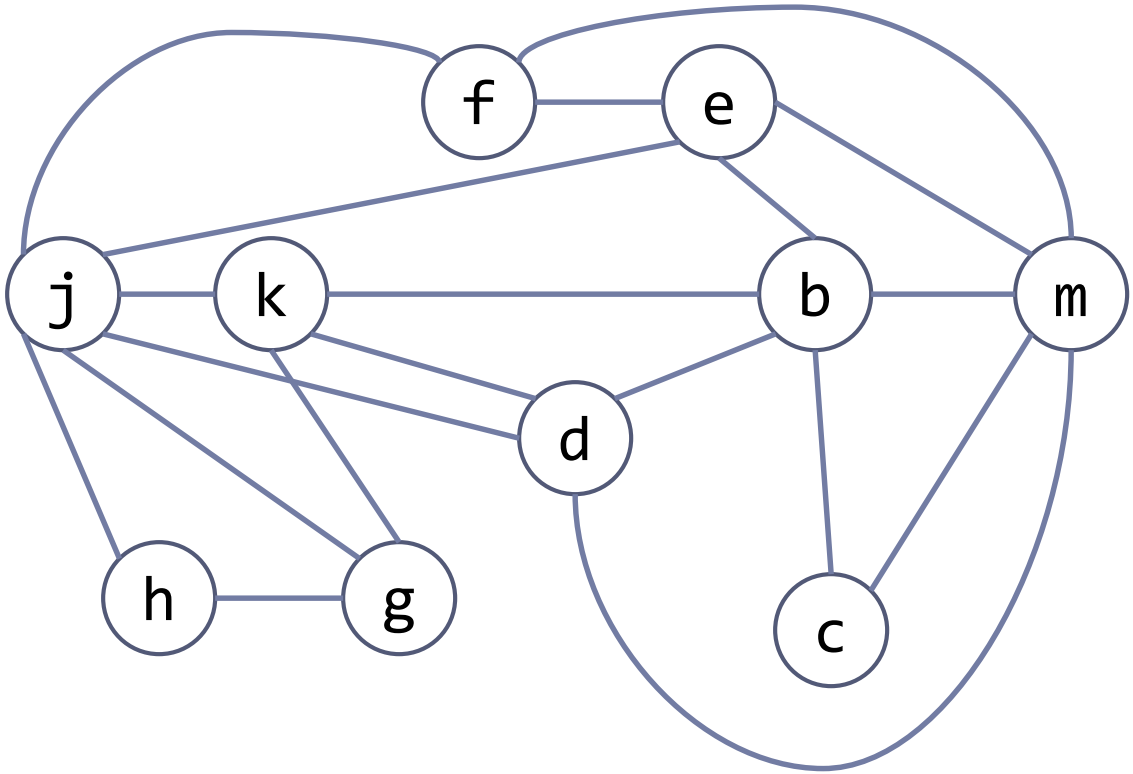
Pop off c, assign register other than s and t; e.g. u



We have found a 4-colouring of the interference graph

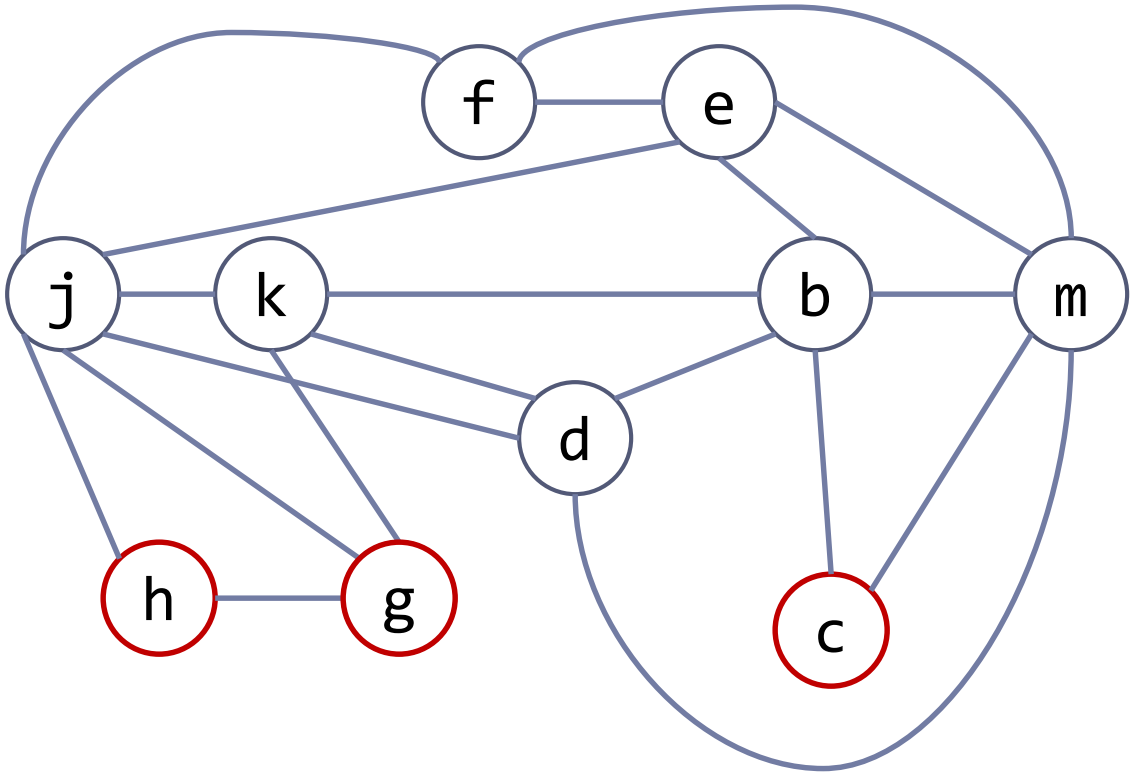
Example (from Appel's textbook)

What if we only have three registers r, s, t?



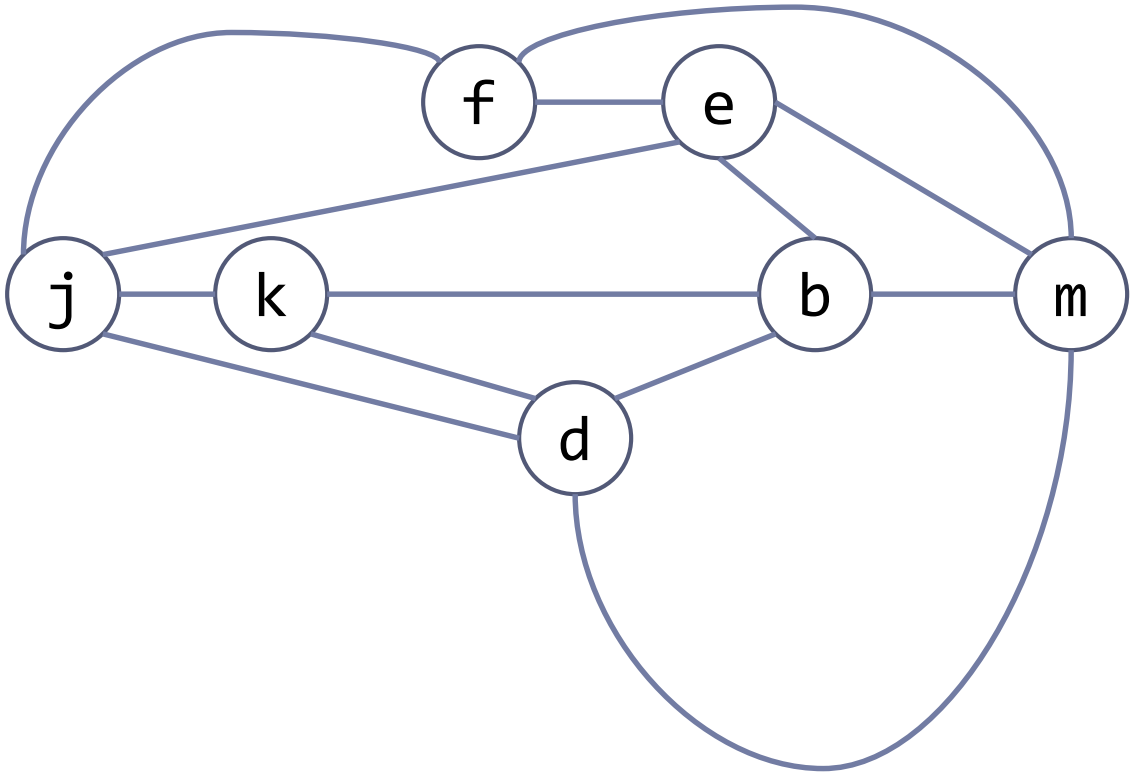
Example (from Appel's textbook)

We proceed as before: first find nodes with less than three neighbours



Example (from Appel's textbook)

Remove these nodes and push them onto stack

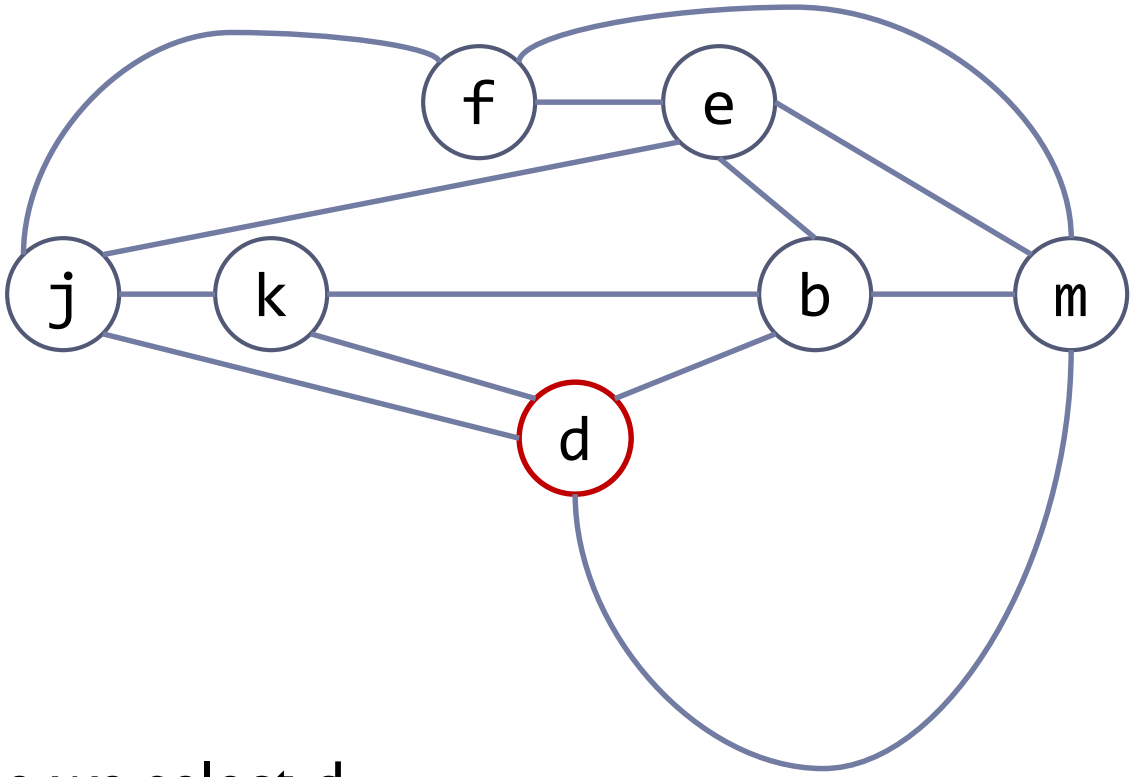


g
h
c

All remaining nodes have at least three neighbours!

Example (from Appel's textbook)

We select a spill candidate: a node with the *greatest* number of neighbours

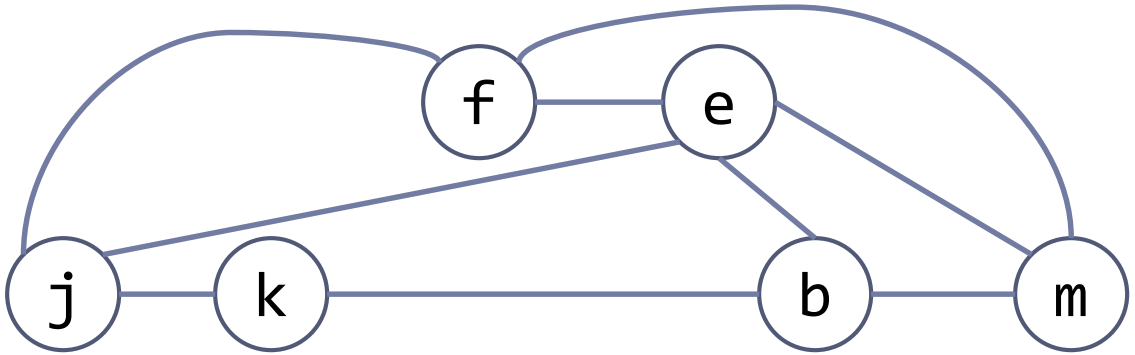


g
h
c

Suppose we select d

Example (from Appel's textbook)

We remove the spill candidate and push it onto the stack, marking it with an asterisk

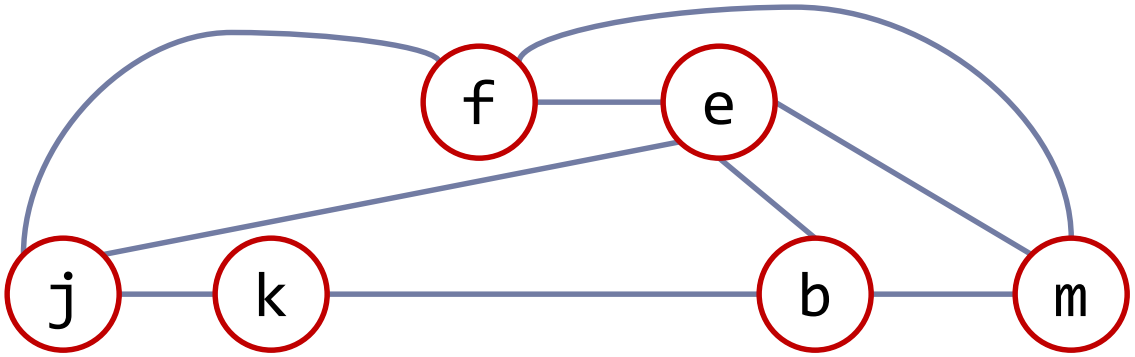


d*
g
h
c

Then we proceed as before

Example (from Appel's textbook)

Find nodes with less than three neighbours

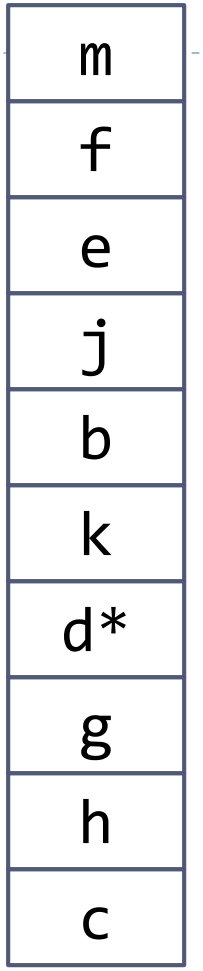


d*
g
h
c

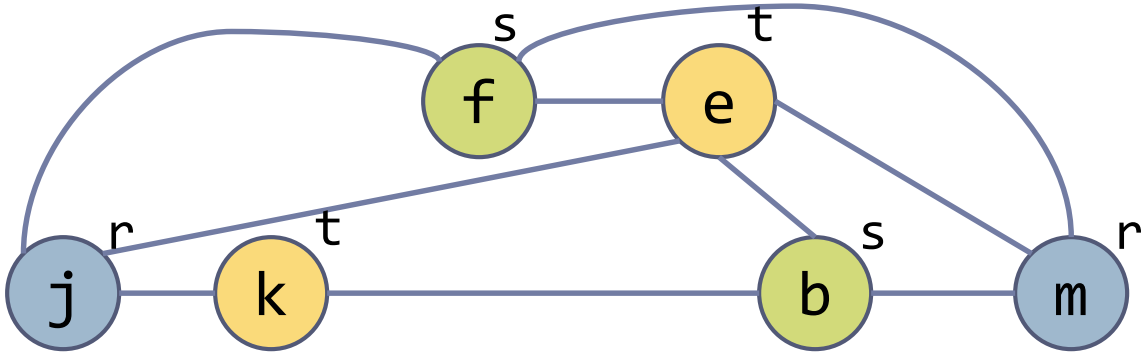
Example (from Appel's textbook)

All other nodes can be removed without incident

We start putting them back in and assigning colors as before



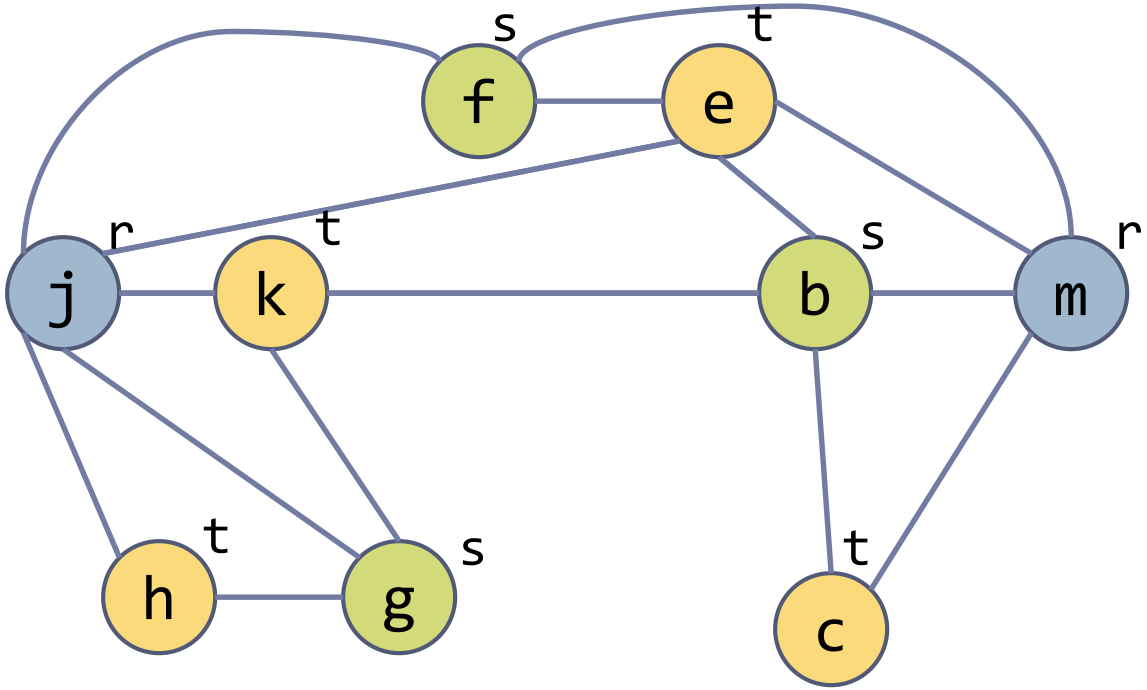
Example (from Appel's textbook)



d*
g
h
c

Note that d cannot be coloured, so it cannot be allocated a register – we simply skip it

Example (from Appel's textbook)



The remaining graph can be 3-coloured