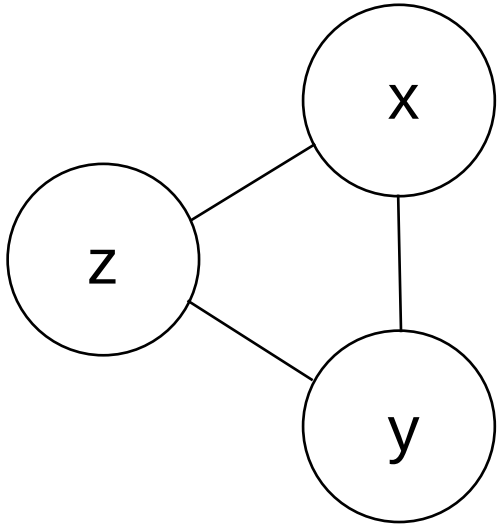
Graph colouring example

Fibonnaci program:

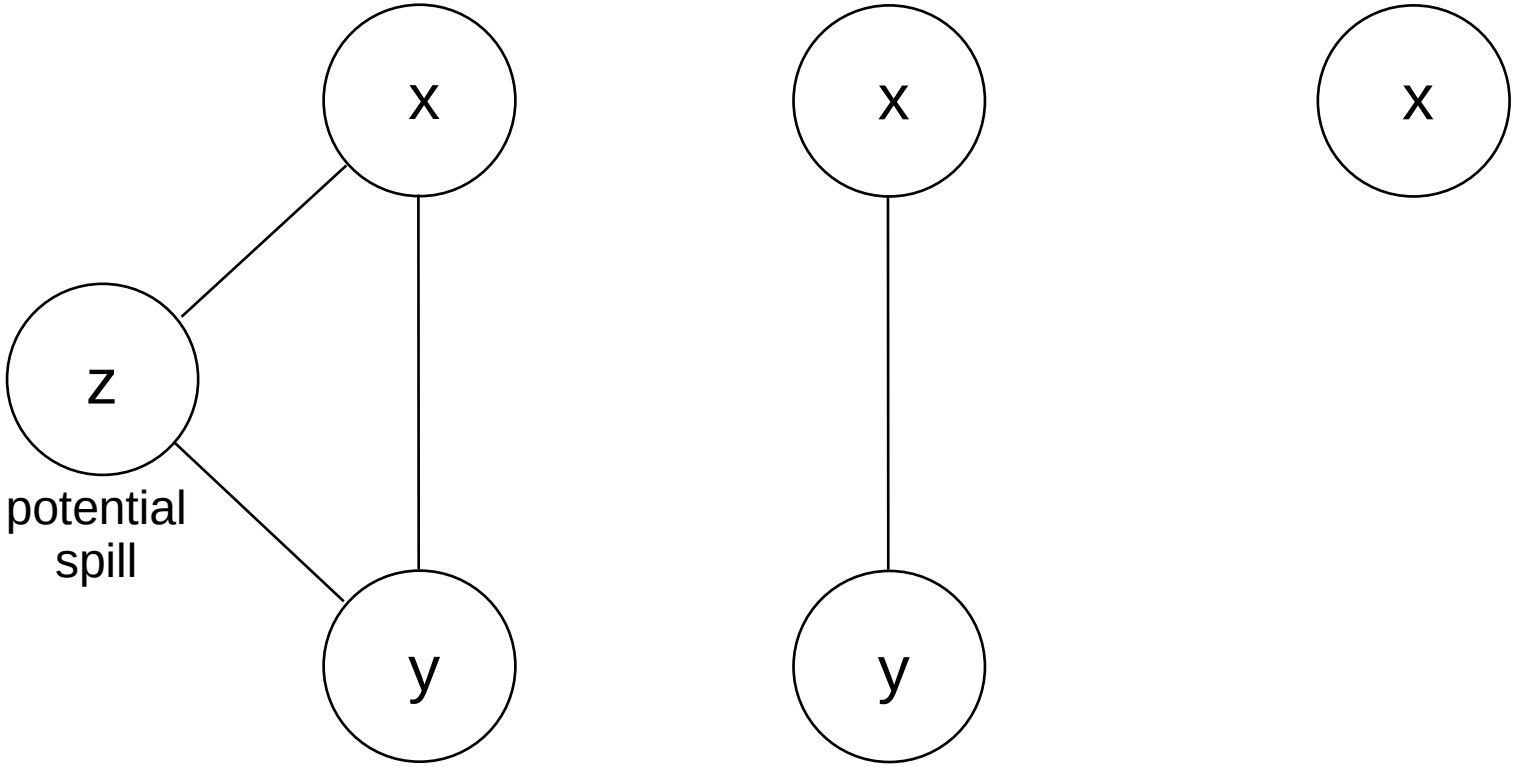
```
1:  x = 0                    {x}
2:  y = 1                    {x, y}     edge xy
3:  if (y > 1000) goto 8     {x, y}
4:  z = x + y                {y, z}     edge yz
5:  x = y                    {x, z}     edge xz
6:  y = z                    {x, y}
7:  goto 3
8:  write(y)
```

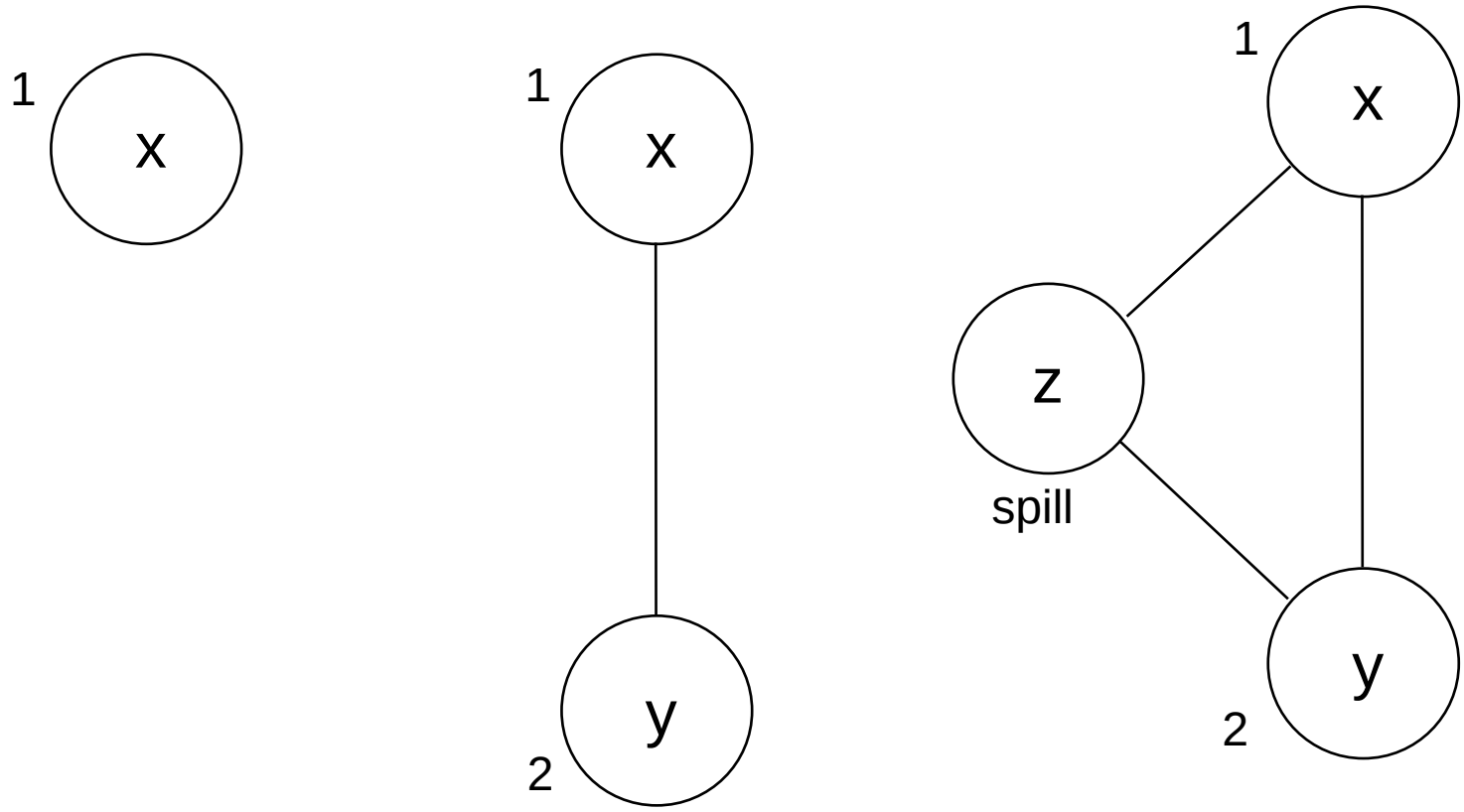
Can’t fit x, y, z into 2 registers.



Need to spill some temporary(s) to memory.

Try to colour graph with at most 2 colours:





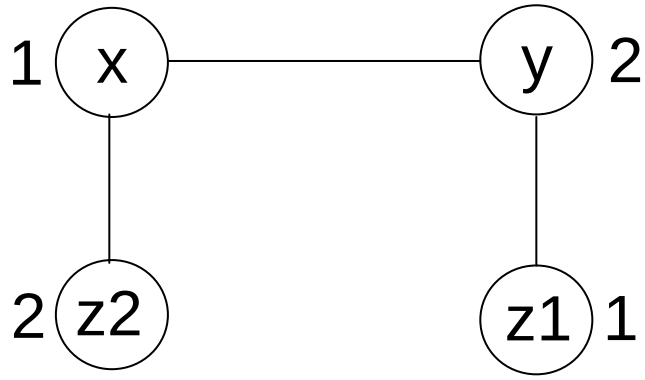
Can’t colour z: must rewrite program to spill z.

Spilling keeps variable in memory and creates new temporaries.

Rewritten program:

```
1:  x = 0                      {x}
2:  y = 1                      {x, y}
3:  if (y > 1000) goto 10      {x, y}
4:  z1 = x + y                 {y, z1}
5:  M[0] = z1                  {y}
6:  x = y                      {x}
7:  z2 = M[0]                  {x, z2}
8:  y = z2                     {x, y}
9:  goto 3
10: write(y)
```

Register interference graph for rewritten program:



Avoiding moves between registers

Coalescing nodes

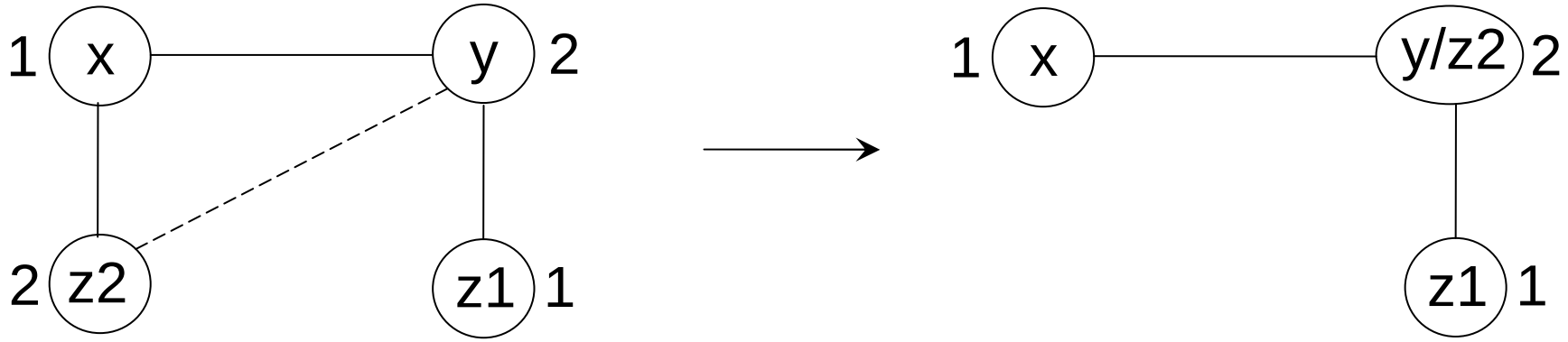
Can eliminate redundant copying:

e.g., `a = b` — use same register for `a` and `b`

- If there is a statement `a = b` such that there is no interference edge between `a` and `b`, coalesce `a` and `b` nodes into one.

- New node has union of old nodes' neighbours.

E.g., Fibonacci example: can coalesce y and z2:



Resulting program with MOVE deleted:

Before

```
1:  x = 0
2:  y = 1
3:  if (y > 1000) goto 10
4:  z1 = x + y
5:  M[0] = z1
6:  x = y
7:  z2 = M[0]
8:  y = z2
9:  goto 3
10: write(y)
```

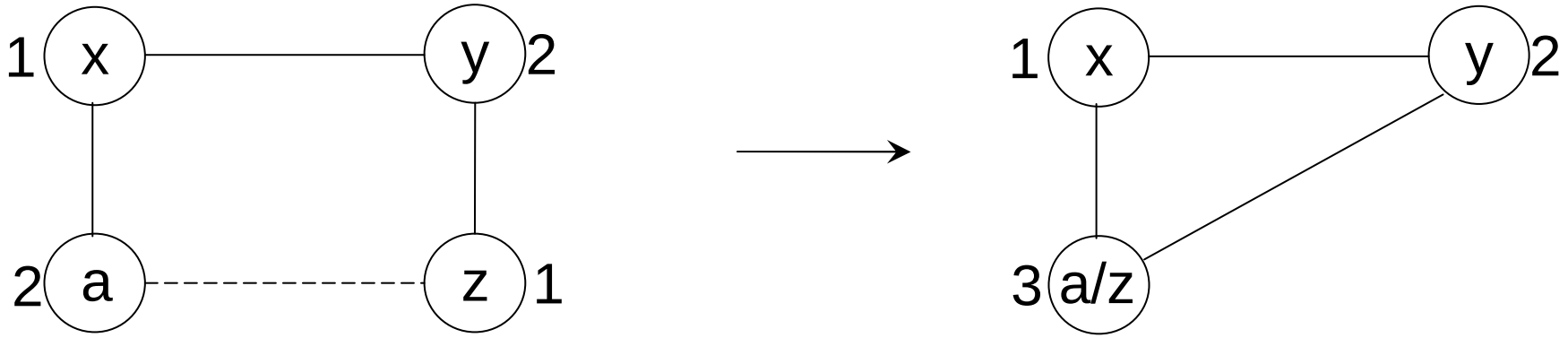
After

```
1:  x = 0
2:  y = 1
3:  if (y > 1000) goto 9
4:  z1 = x + y
5:  M[0] = z1
6:  x = y
7:  y = M[0]
8:  goto 3
9:  write(y)
```

Problem: may increase number of registers needed.

E.g., previous example program: can coalesce a and z:

```
x = 2       {x}
a = 1       {x, a}
y = x + a   {x, y}
z = y + x   {y, z}
z = y + z   {z}
a = z       {a}
x = a + a   {x}
```



Solution: coalesce only if the modified graph is still colourable

- Coalesce `a` and `b` if new node `a/b` would have $< k$ neighbours of degree $\geq k$

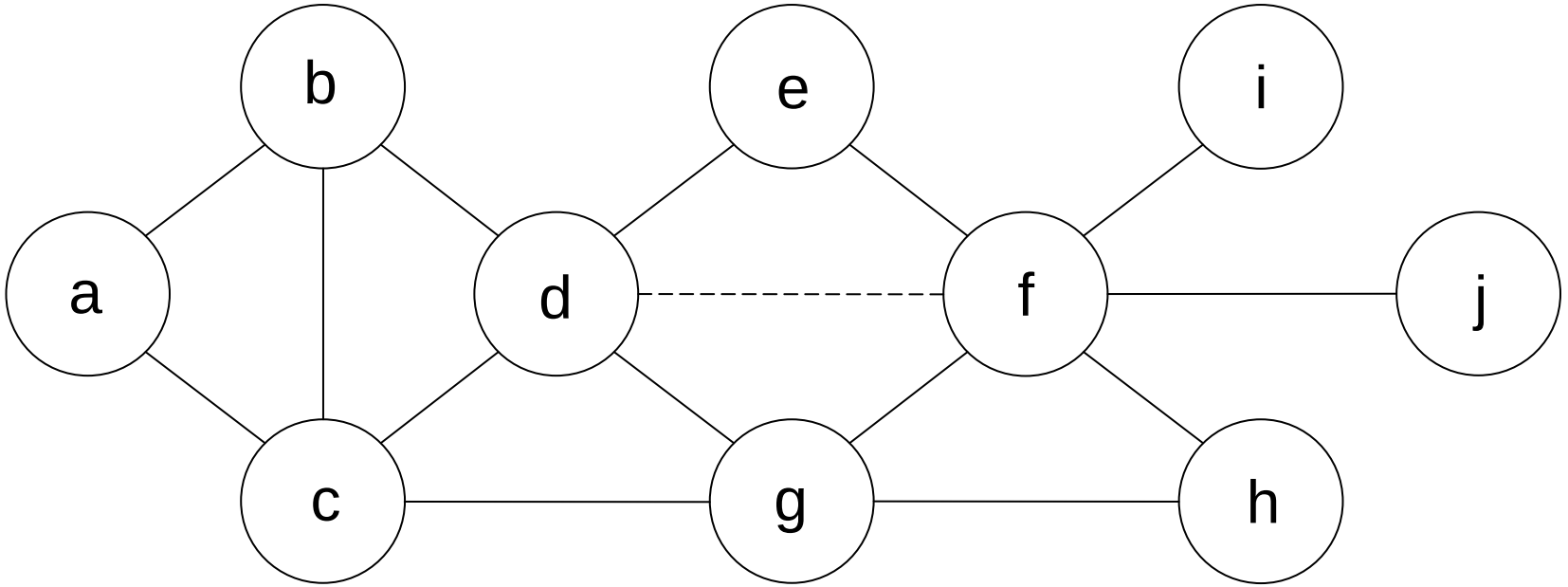
- Coalescing can be done during algorithm while simplifying graph

`colour_graph(k)`:

```
if (graph not empty) {
  if (graph contains non-move-related node u with degree < k)
    v = u;
  else if (graph contains 2 coalescable move-related nodes u1 and u2)
    v = coalesce(u1, u2);
  else if (graph contains a move-related node u with degree < k) {
    make u non-move-related;
    v = u;
  }
  else v = any node;
  remove_node(v);
  colour_graph(k);
  add_node(v);
  if (v's degree < k)
    colour[v] = any colour from 1..k not used by neighbours;
  else /* potential spill */
  if (some colour c from 1..k is not used by neighbours)
    colour[v] = c;
  else /* actual spill */
    colour[v] = NONE;
}
```

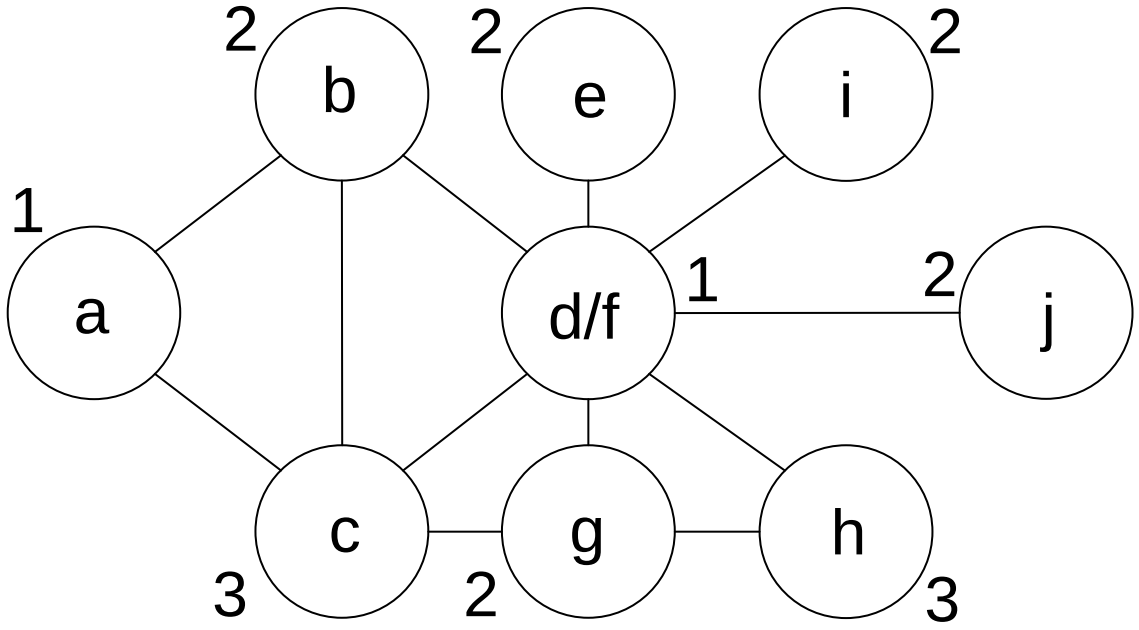
Example

Colour graph with 3 colours:



Can’t coalesce `d` and `f` because `d/f` would have 3 neighbours with degree = 3.

```
Remove non-move-related node a, b, c, e, h, g, i, j
Coalesce d and f
Colour d/f 1
Colour j 2
Colour i 2
Colour g 2
Colour h 3
Colour e 2
Colour c 3
Colour b 2
Colour a 1
```



Register allocation: other issues

Precoloured nodes:

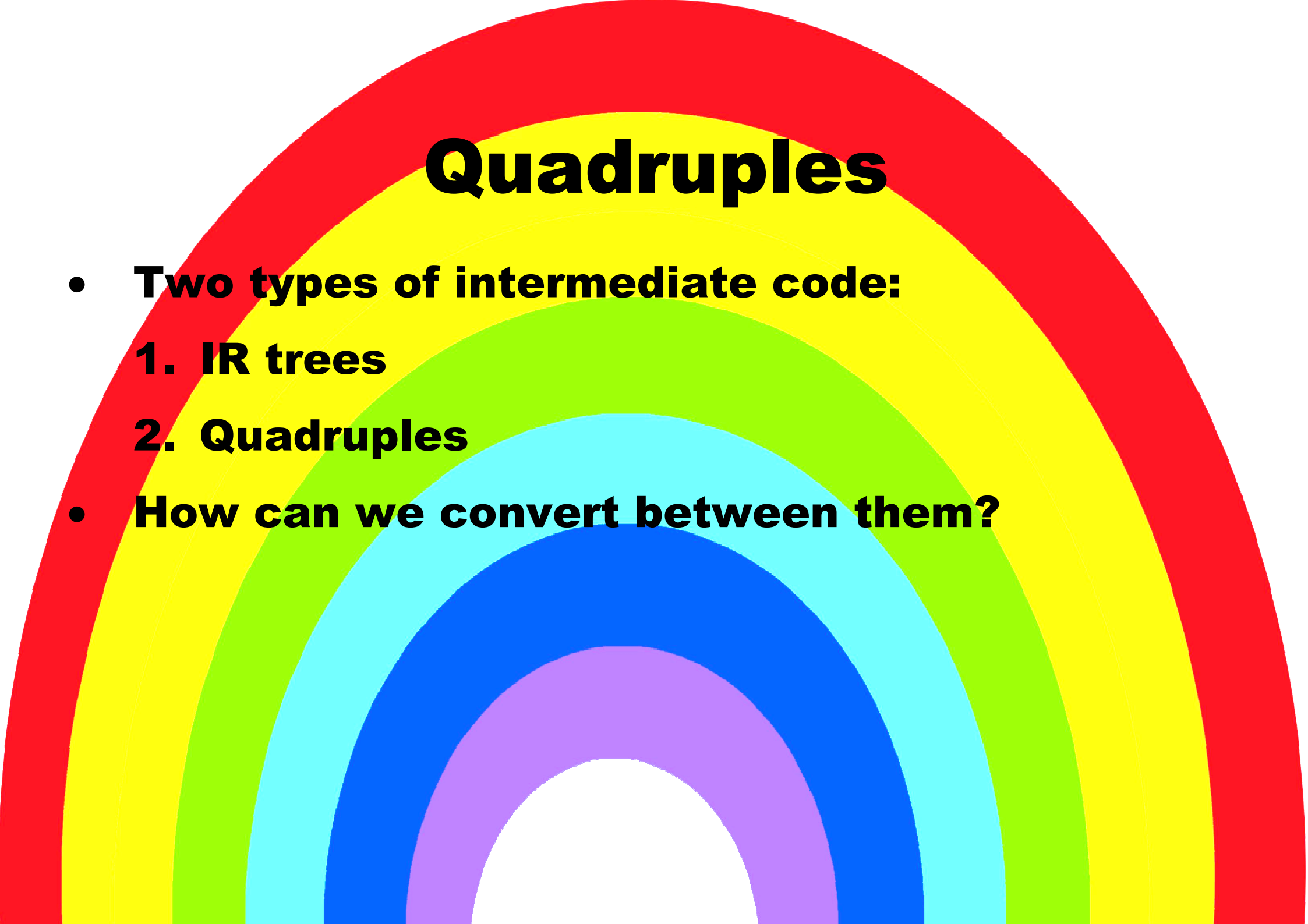
- Some nodes represent specific registers

- Give these nodes a colour before colouring graph

Callee-save registers:

- Values must be preserved across function calls

- In function body, callee-save register is implicitly *defined* at beginning and *used* at end



Quadruples

- Two types of intermediate code:
  1. IR trees
  2. Quadruples
- How can we convert between them?

Quadruple representation

Quadruples:

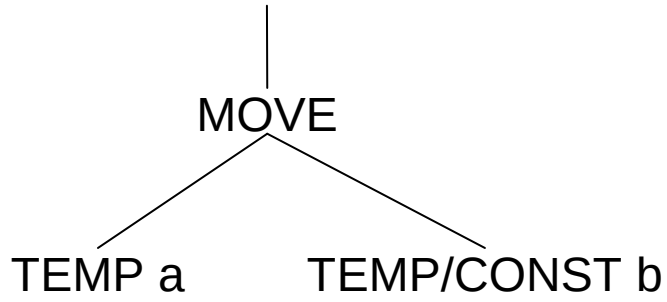
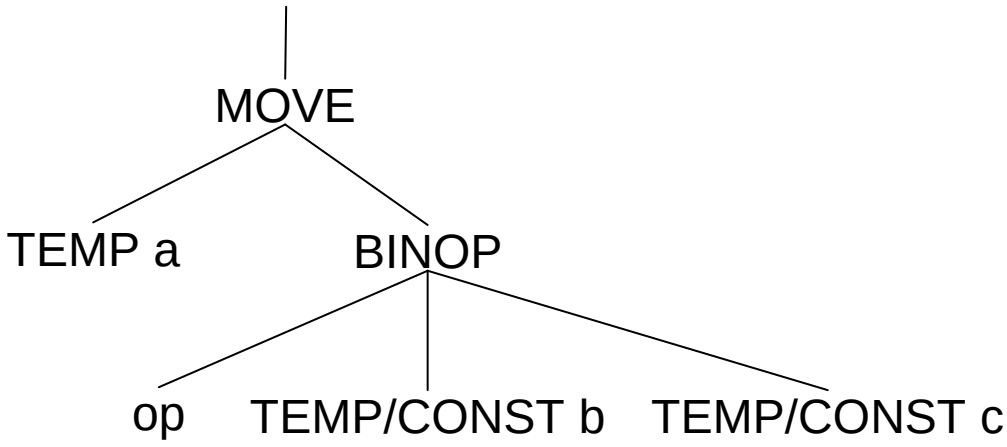
type of intermediate representation.

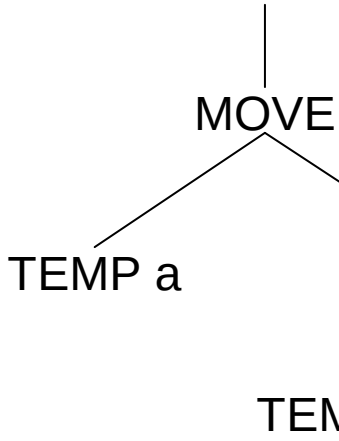
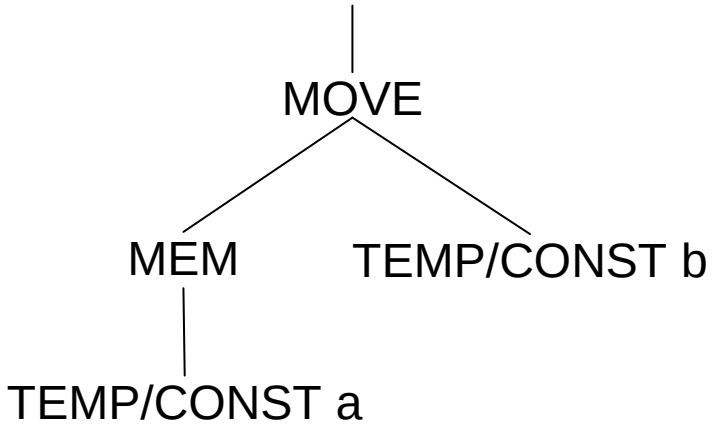
Quadruple = operator + destination + source + source.

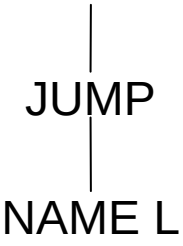
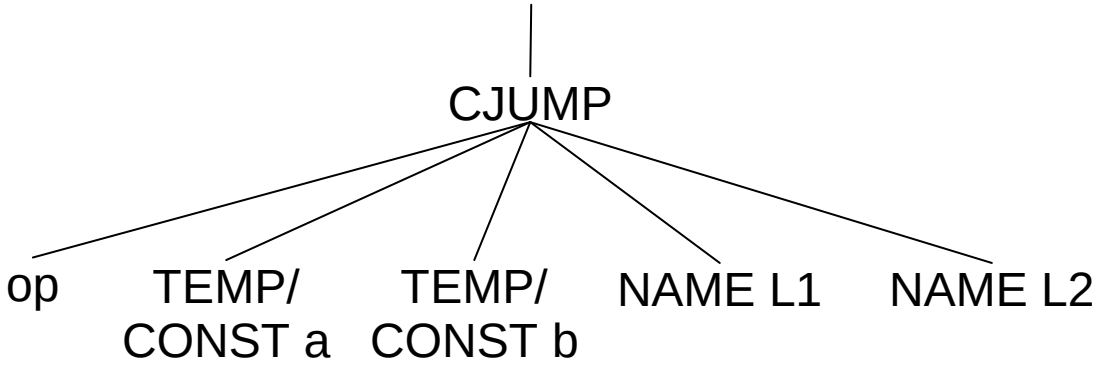
Like tree representation except:

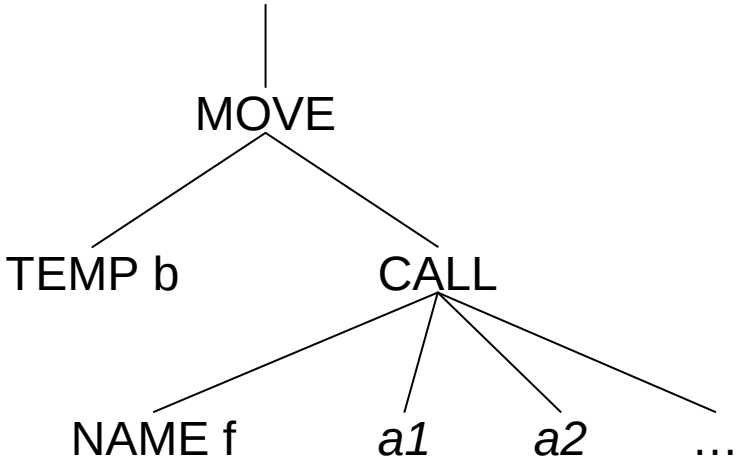
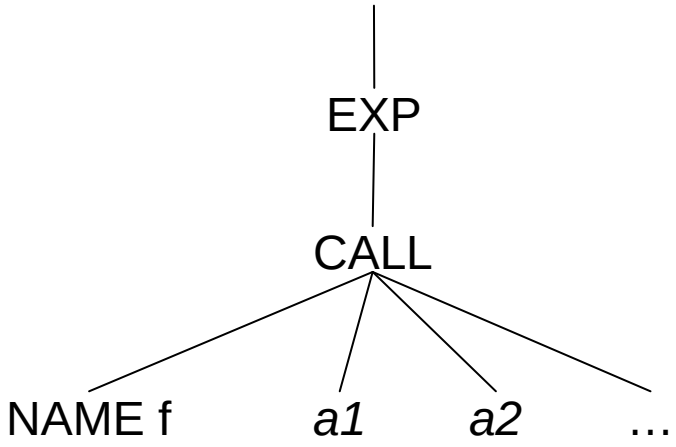
- no nested expressions
- all intermediate values are in explicit temporaries
- order of evaluation is explicit

Types of quadruple

Quadruple	Tree pattern
`a = b`	
`a = b op c`	

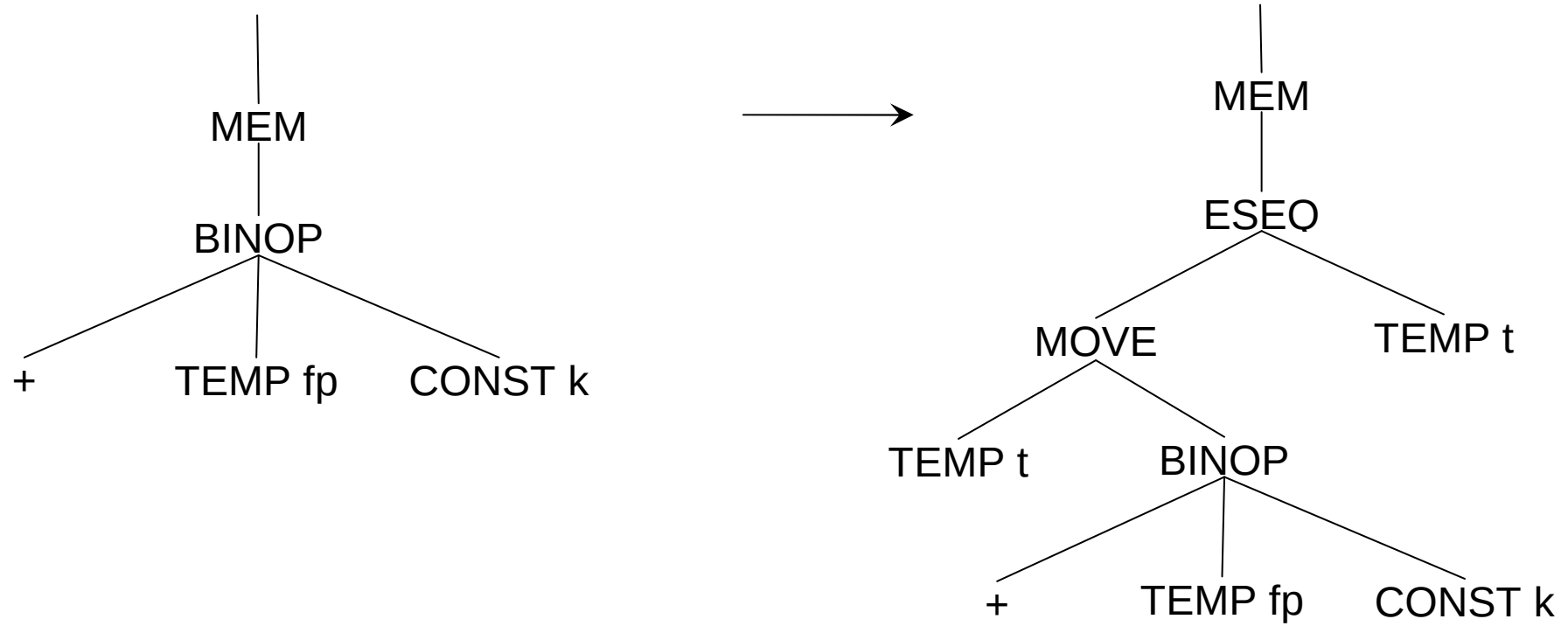
Quadruple	Tree pattern
`a = M[b]`	
`M[a] = b`	
`L:`	

Quadruple	Tree pattern
`goto L`	
`if (a op b) goto L1` `else goto L2`	

Quadruple	Tree pattern
`b = f(a1, …, an)`	
`f(a1, …, an)`	

Converting trees to quadruples:

Flatten expressions: introduce new temporaries and sequencing.



Converting quadruples to trees:

Search tree for every TEMP node that appears once (and on left of MOVE). Replace it by right subtree of corresponding MOVE tree.