COMPILERS
Register Allocation

hussein suleman
uct csc3003s 2009

Register Allocation

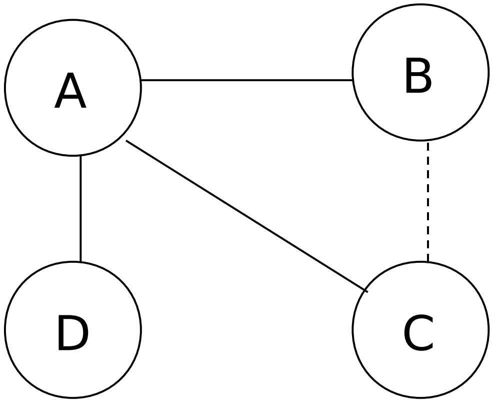
- Want to maximise use of registers for temporaries.
- Build interference graph for each program point.
  - Compute set of temporaries simultaneously live.
  - Add edge to graph for each pair in set.
- Then K-colour the graph by assigning a maximum of K colours such that interfering variables always have different colours.

Step 1: Build

- Start with final live sets from liveness analysis. Add an edge for each pair of simultaneously live variables to interference graph.
- Add a dotted line for any “MOVE a,b” instructions

	out	in
a←1	a	-
b←2	ab	a
c←a+b	ac	ab
b←c	ac	ac
d←a+c	ad	ac
d←a+d	-	ad

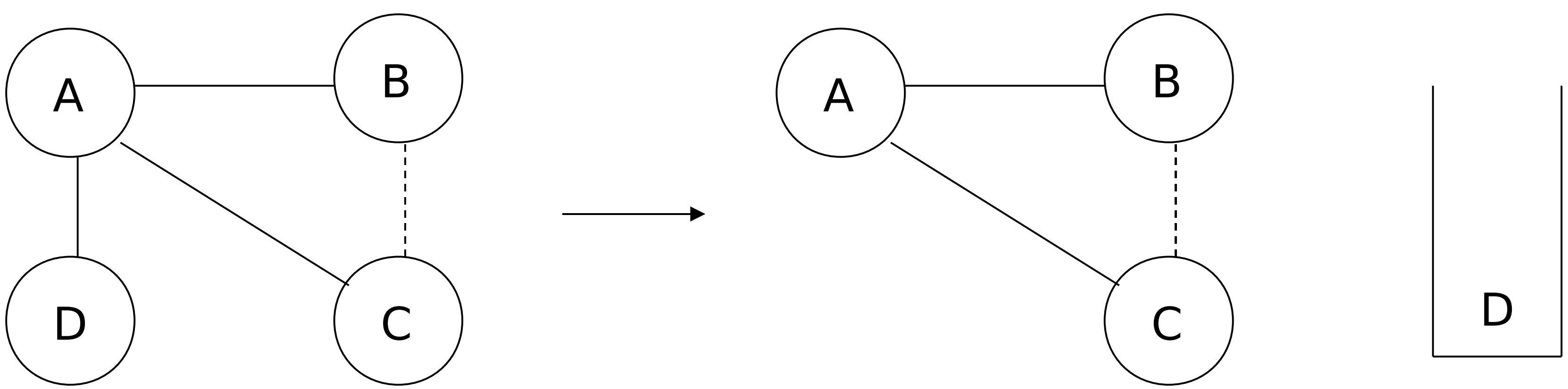
code and liveness analysis



interference graph (assume K=2)

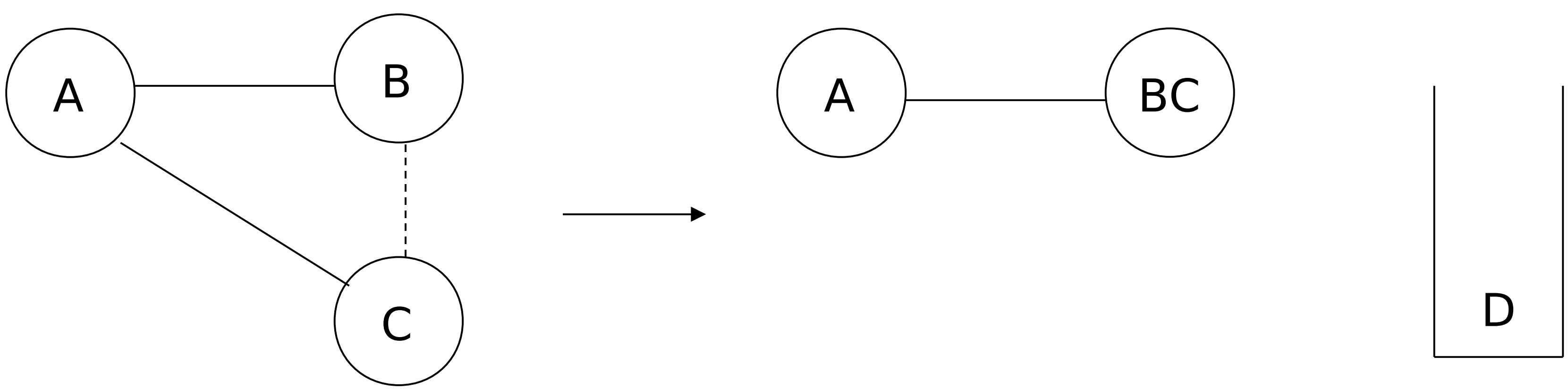
Step 2: Simplify

- Remove any non-MOVE-related nodes with degree<K and place on stack.
  - K-colourability is maintained in the new subgraph.
  - Every node removed is guaranteed a colour.
  - The removal of edges may create other <K degree nodes.



Step 3: Coalesce

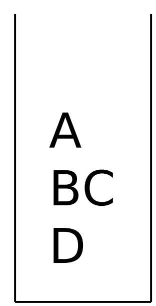
- Merge together MOVE-related nodes if it does not decrease colourability.
  - Briggs: only if merged node ab has <K significant degree neighbours
  - George: only if all significant degree neighbours of a interfere with b



And Repeat …

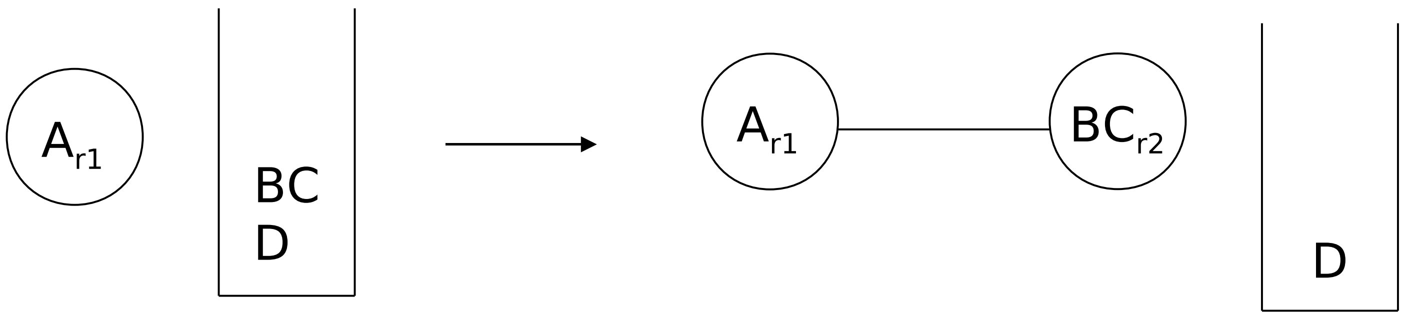
- Simplify again

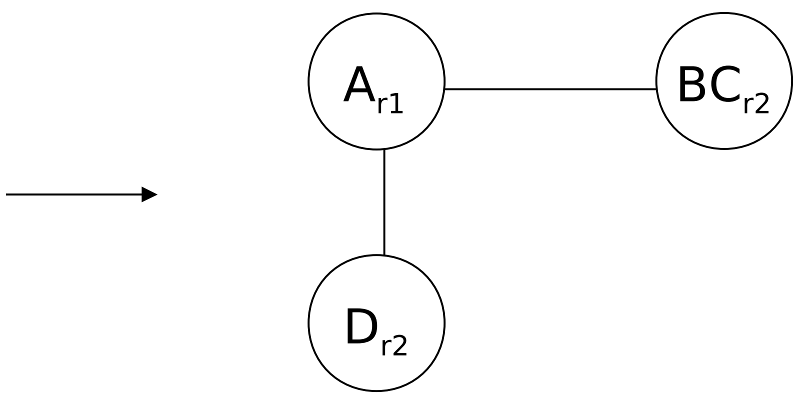
- And simplify again



Step 6: Select

- Pop nodes off stack and assign colours/registers.





R1 ← 1

R2 ← 2

R2← R1 + R2

~~R2 ← R2~~

R2 ← R1 + R2

R2 ← R1 + R2

Redundant MOVE (from coalesce) can be removed!

Step 4: Freeze

- If there are no nodes that can be simplified or coalesced AND there are MOVE-related nodes, freeze all MOVEs of a low-degree MOVE-related node.
  - Ignore the MOVE and treat the nodes like any others.

- Repeat steps 2-3-4

Step 5: Potential Spill

- If there are no nodes that can be simplified, coalesced or frozen, choose a node that isnt used much in the program and spill it.
  - Add it to the stack just like a simplify.
  - There may or may not be a colour to allocate to it during the Select step – if there isnt, the potential spill becomes an actual spill.

- Repeat steps 2-3-4-5

Actual Spills

- An actual spill is when there aren't enough registers to store the temporaries.
- Rewrite program to shorten live range of spilled variable.
  - Move variable to memory after define.
  - Move memory to variable before use.
- Then, repeat process from Step 1.

$c \leftarrow a$		$c_1 \leftarrow a$
.		$Mem_c \leftarrow c_1$
.	→	.
.		$c_2 \leftarrow Mem_c$
$b \leftarrow c+1$		$b \leftarrow c_2 + 1$

Spilled Temporaries

- Spilled temporaries can be graph-coloured to reuse activation record slots.
  - Coalescing can be aggressive, since (unlike registers) there is no limit on the number of stack-frame locations.
  - Aggressive coalescing: Any non-interfering nodes can be coalesced since there is no upper bound K.

Precoloured Nodes

- Precoloured nodes correspond to machine registers (e.g., stack pointer, arguments)
  - Select and Coalesce can give an ordinary temporary the same colour as a precoloured register, if they don't interfere
    - e.g., argument registers can be reused inside procedures for a temporary
  - Simplify, Freeze and Spill cannot be performed on them
- Precoloured nodes interfere with other precoloured nodes.

Temporary Copies

- Since precoloured nodes don’t spill, their live ranges must be kept short:
  - Use MOVE instructions.
  - Move callee-save registers to fresh temporaries on procedure entry, and back on exit, spilling between as necessary.
  - Register pressure will spill the fresh temporaries as necessary, otherwise they can be coalesced with their precoloured counterpart and the moves deleted.

Handling CALL instructions

- Variables whose live ranges span calls should go to callee-save registers, otherwise to caller-save.
- This is easy for graph coloring allocation with spilling
  - Calls define (interfere with) caller-save registers.
  - Calls use parameter registers.
  - A variable that is alive before and after a call interferes with all precoloured caller-save registers, as well as with the fresh temporaries created for callee-save copies, forcing a spill.
  - Choose nodes with high degree but few uses, to spill the fresh callee-save temporary instead of the cross-call variable. This makes the original callee-save register available for colouring the cross-call variable

Example

```
f:   c ← r3
     p ← r1
     if p=0 goto L1
     r1 ← M[p]
     call f
     s ← r1
     r1 ← M[p+4]
     call f
     t ← r1
     u ← s + t
     goto L2
L1:  u ← 1
L2:  r1 ← u
     r3 ← c
     return
```

- 3 Machine registers (K=3)
- Caller-save:
  - r1, r2
- Callee-save:
  - r3
- CALL parameters are set in r1 and results are returned in r1

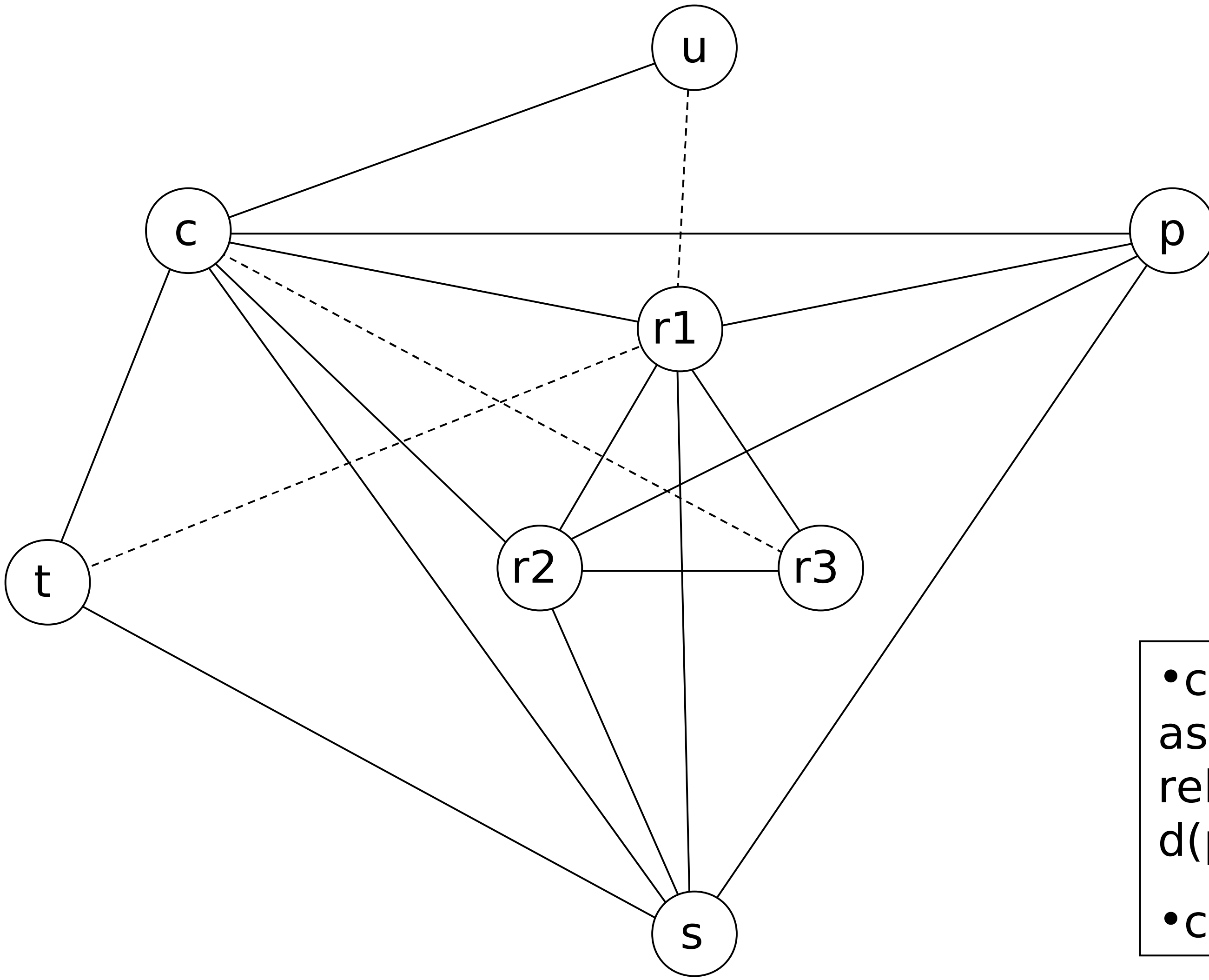
Example: Liveness Analysis

#		Statement	Succ	Use	Def	Out	In
1	f:	c ← r3	2	r3	c	cr1	r1r3
2		p ← r1	3	r1	p	cp	cr1
3		if p=0 goto L1	4,12	p		cp	cp
4		r1 ← M[p]	5	p	r1	cpr1	cp
5		call f	6	r1	r1r2	cpr1	cpr1
6		s ← r1	7	r1	s	cps	cpr1
7		r1 ← M[p+4]	8	p	r1	cpr1	cps
8		call f	9	r1	r1r2	csr1	csr1
9		t ← r1	10	r1	t	cst	csr1
10		u ← s + t	11	st	u	cu	cst
11		goto L2	13			cu	cu
12	L1:	u ← 1	13		u	cu	c
13	L2:	r1 ← u	14	u	r1	cr1	cu
14		r3 ← c	15	c	r3	r1r3	cr1
15		return		r1r3			r1r3

Example: Edge Determination

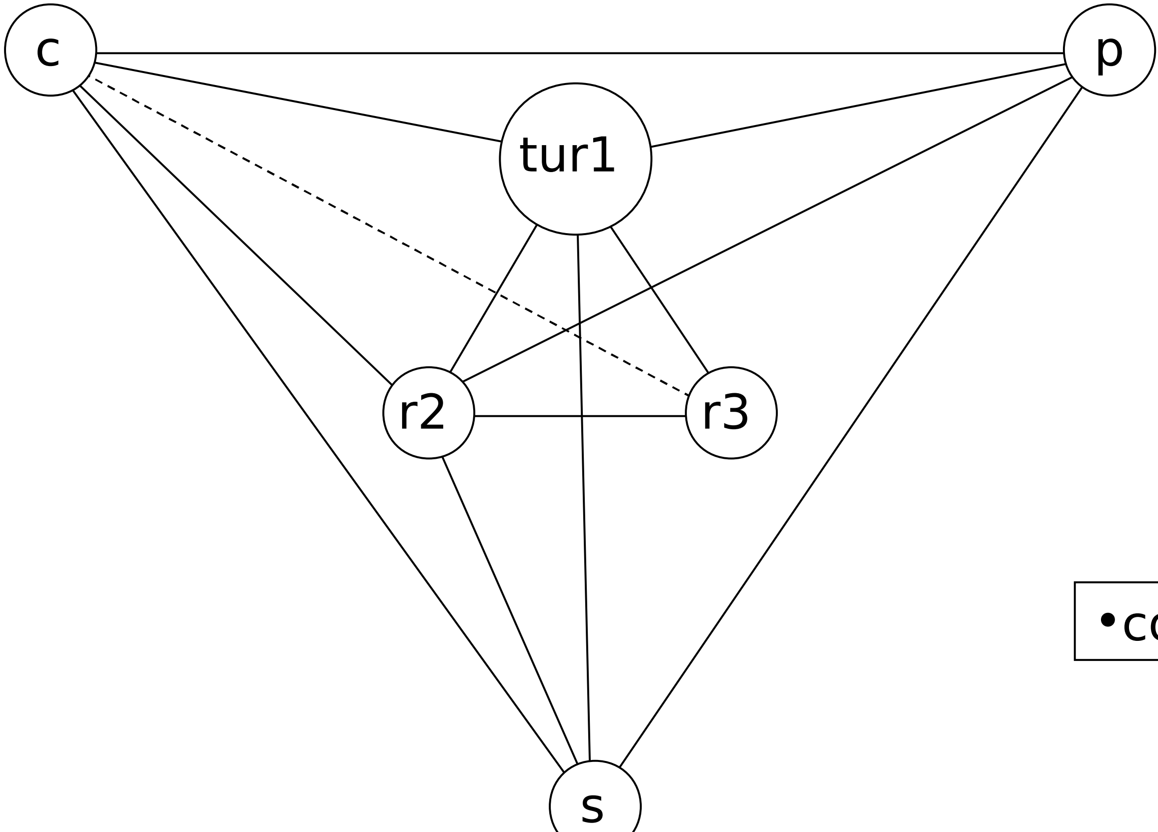
- Calculate live pairs based on liveness sets:
  - liveness sets: cpr1, cps, csr1, cst, cu
  - edges ⊇ {cp, cr1, cs, ct, cu, pr1, ps, sr1, st}
- For each CALL, the variables that are live-in and also live-out must interfere with all caller-save registers (r1r2).
  - cp is live-in and live-out in line 5, cs in line 8
  - edges ⊇ {cp, cs, cr1, cr2, pr1, pr2, sr1, sr2}
- Create pairs of precoloured nodes (e.g., machine registers).
  - edges ⊇ {r1r2, r2r3, r1r3}
- Determine move instructions that are not already constrained.
  - moves = {cr3, tr1, ur1} (constrained = {pr1, sr1})

Example 1



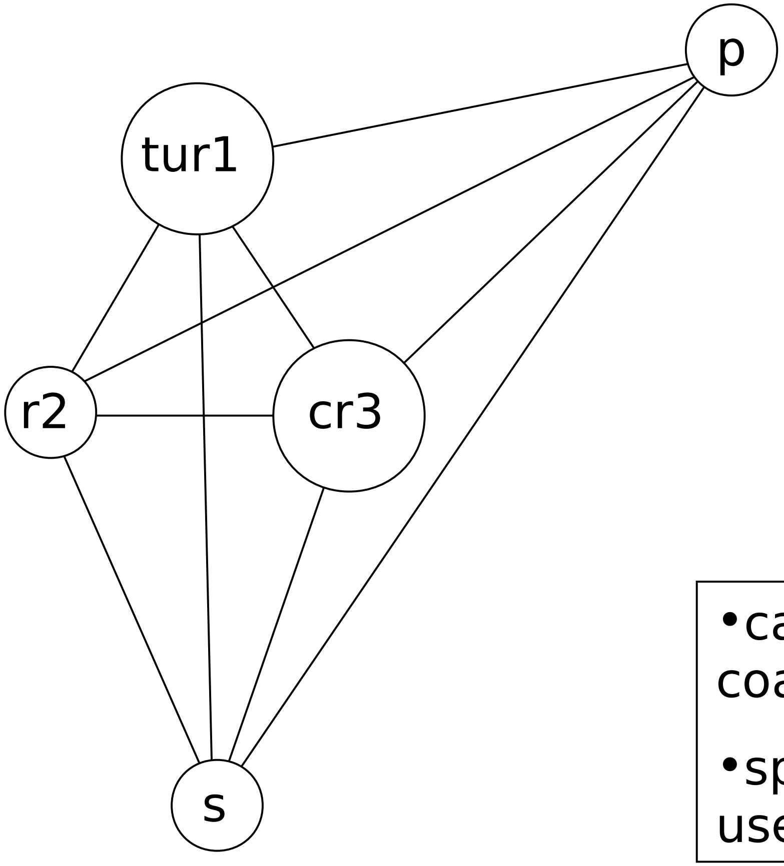
- cannot simplify as all non-MOVE-related nodes: d(ps) >= K
- coalesce tur1

Example 2



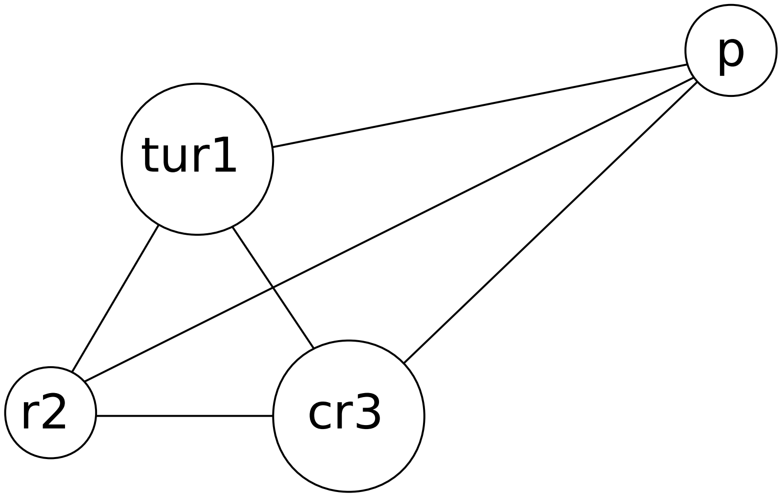
- coalesce cr3

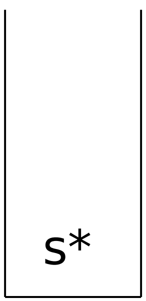
Example 3



- cannot simplify, coalesce, freeze
- spill s (as it is used less than p)

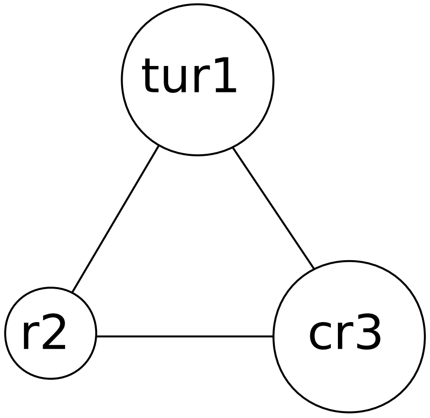
Example 4

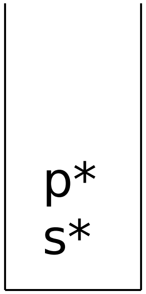




- cannot simplify, coalesce, freeze
- spill p

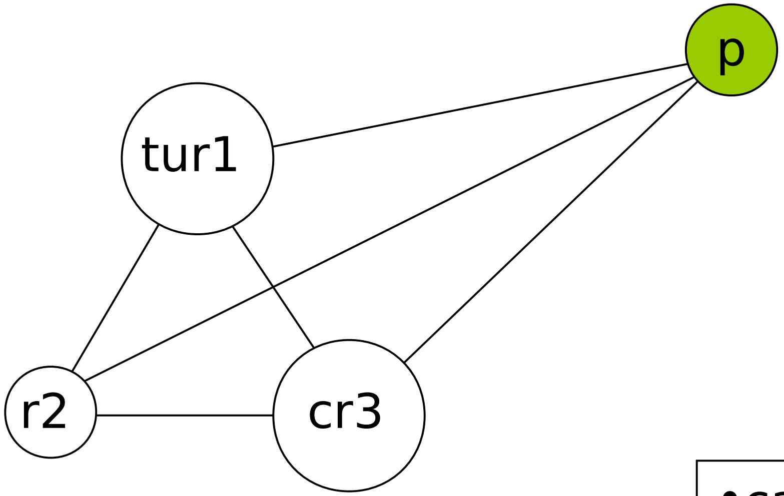
Example 5

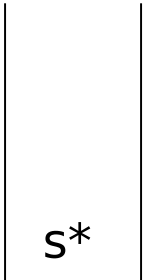




- cannot remove precoloured nodes
- select to put back nodes and colour
- add p

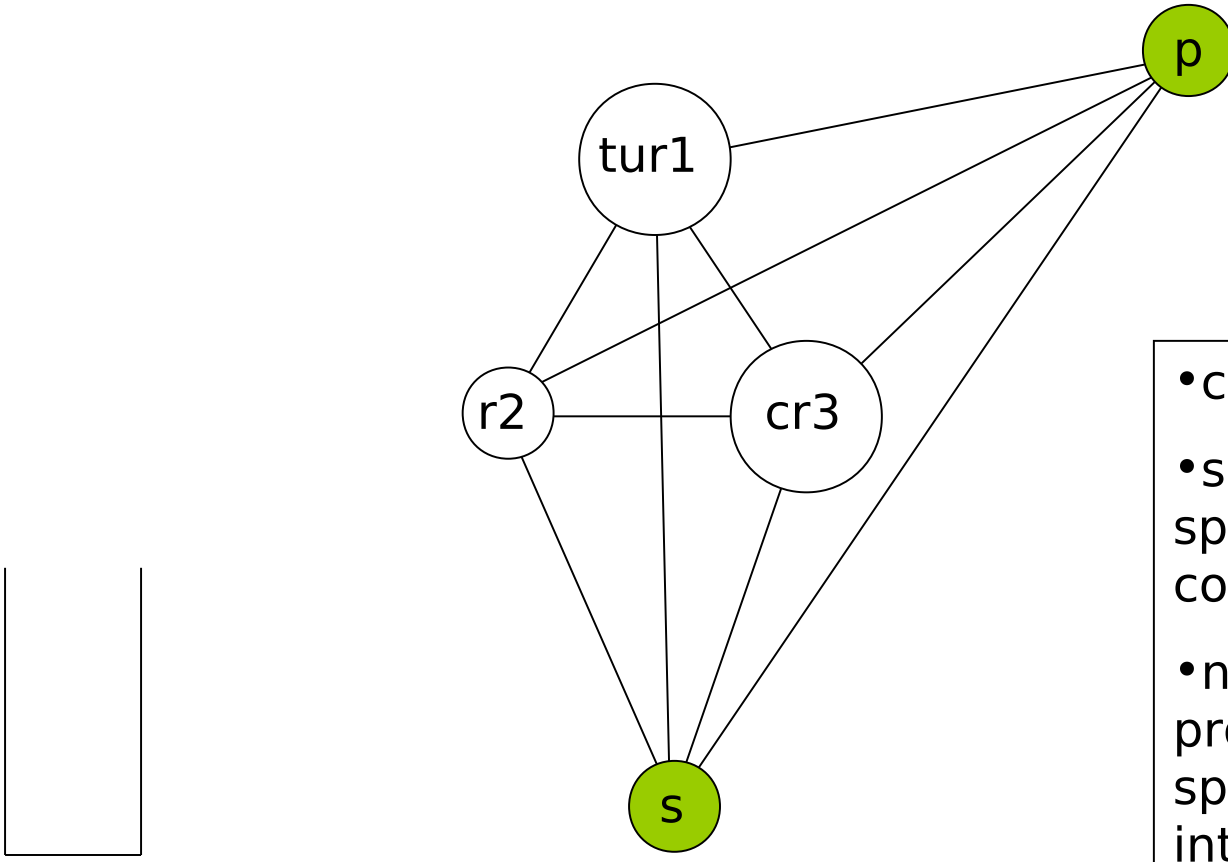
Example 6





- cannot colour p
- p is an actual spill – do not colour it
- add potential spill s

Example 7



- cannot colour s
- s in an actual spill – do not colour it
- now rewrite program to handle spills and build interference graph

Example Rewritten

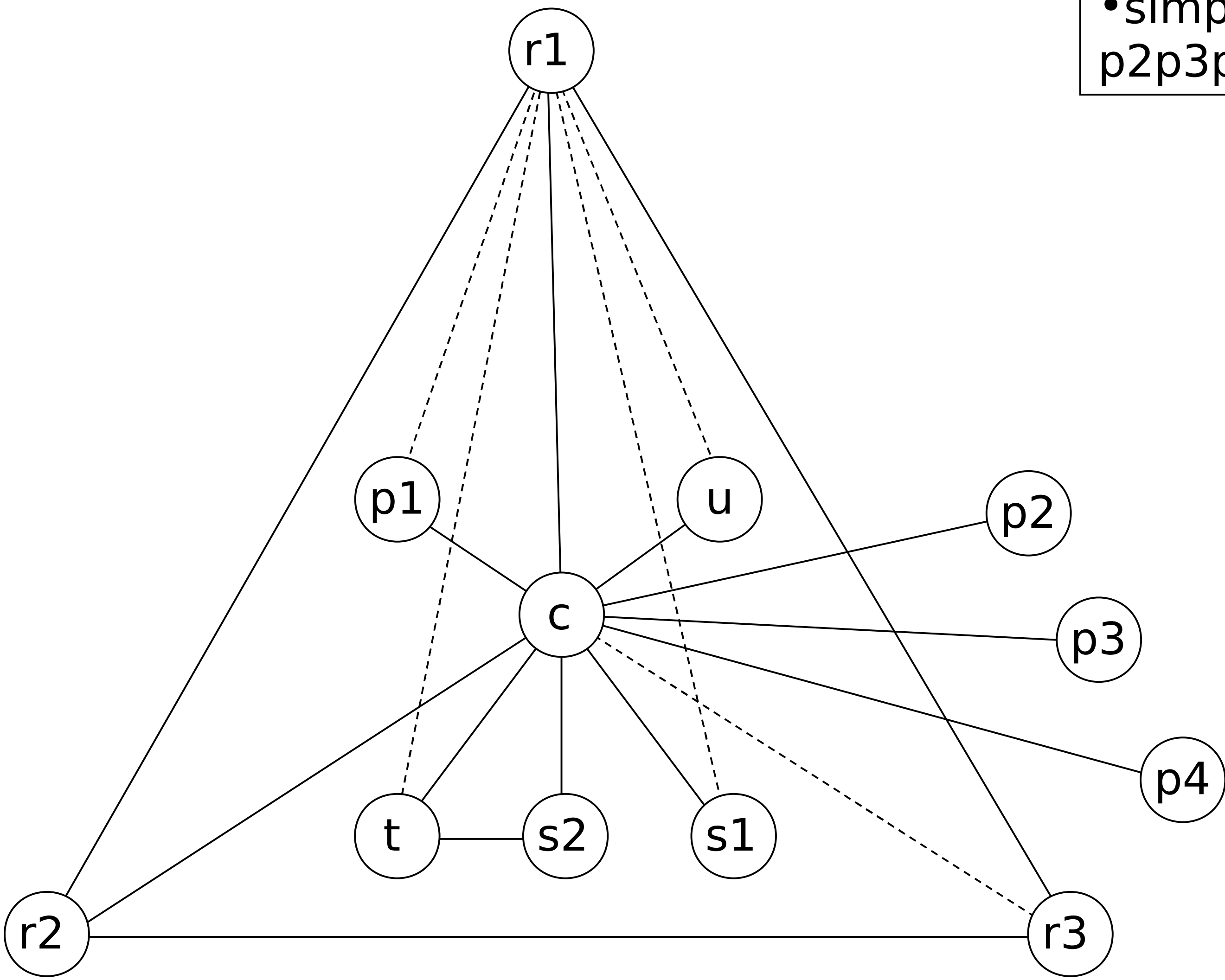
#		Statement	Succ	Use	Def	Out	In
1	f:	c ← r3	2	r3	c	cr1	r1r3
2		p1 ← r1	3	r1	p	cp1	cr1
3		Mp ← p1	4	p1		c	cp1
4		p2 ← Mp	5		p2	cp2	c
5		if p2=0 goto L1	6,18	p2		c	cp2
6		p3 ← Mp	7		p3	cp3	c
7		r1 ← M[p3]	8	p3	r1	cr1	cp3
8		call f	9	r1	r1r2	cr1	cr1
9		s1 ← r1	10	r1	s1	cs1	cr1
10		Ms ← s1	11	s1		c	cs1
11		p4 ← Mp	12		p4	cp4	c
12		r1 ← M[p4+4]	13	p4	r1	cr1	cp4
13		call f	14	r1	r1r2	cr1	cr1
14		t ← r1	15	r1	t	ct	cr1
15		s2 ← Ms	16		s2	cs2t	ct
16		u ← s2 + t	17	s2t	u	c	cs2t
17		goto L2	18			c	c
18	L1:	u ← 1	19		u	cu	c
19	L2:	r1 ← u	20	u	r1	cr1	cu
20		r3 ← c	21	c	r3	r1r3	cr1
21		return		r1r3			r1r3

Example: Edge Determination B

- Calculate live pairs based on liveness sets:
  - liveness sets: cr1, cp1, cp2, cp3, cs1, cp4, cs2t, cu
  - edges ⊇ {cr1, cp1, cp2, cp3, cs1, cp4, cs2, ct, s2t, cu}
- For each CALL, the variables that are live-in and also live-out must interfere with all caller-save registers (r1r2).
  - c is live-in and live-out in line 8 and in line 13
  - edges ⊇ {cr1, cr2}
- Create pairs of precoloured nodes (e.g., machine registers).
  - edges ⊇ {r1r2, r2r3, r1r3}
- Determine move instructions that are not already constrained.
  - moves = {p1r1, s1r1, tr1, ur1, cr3}

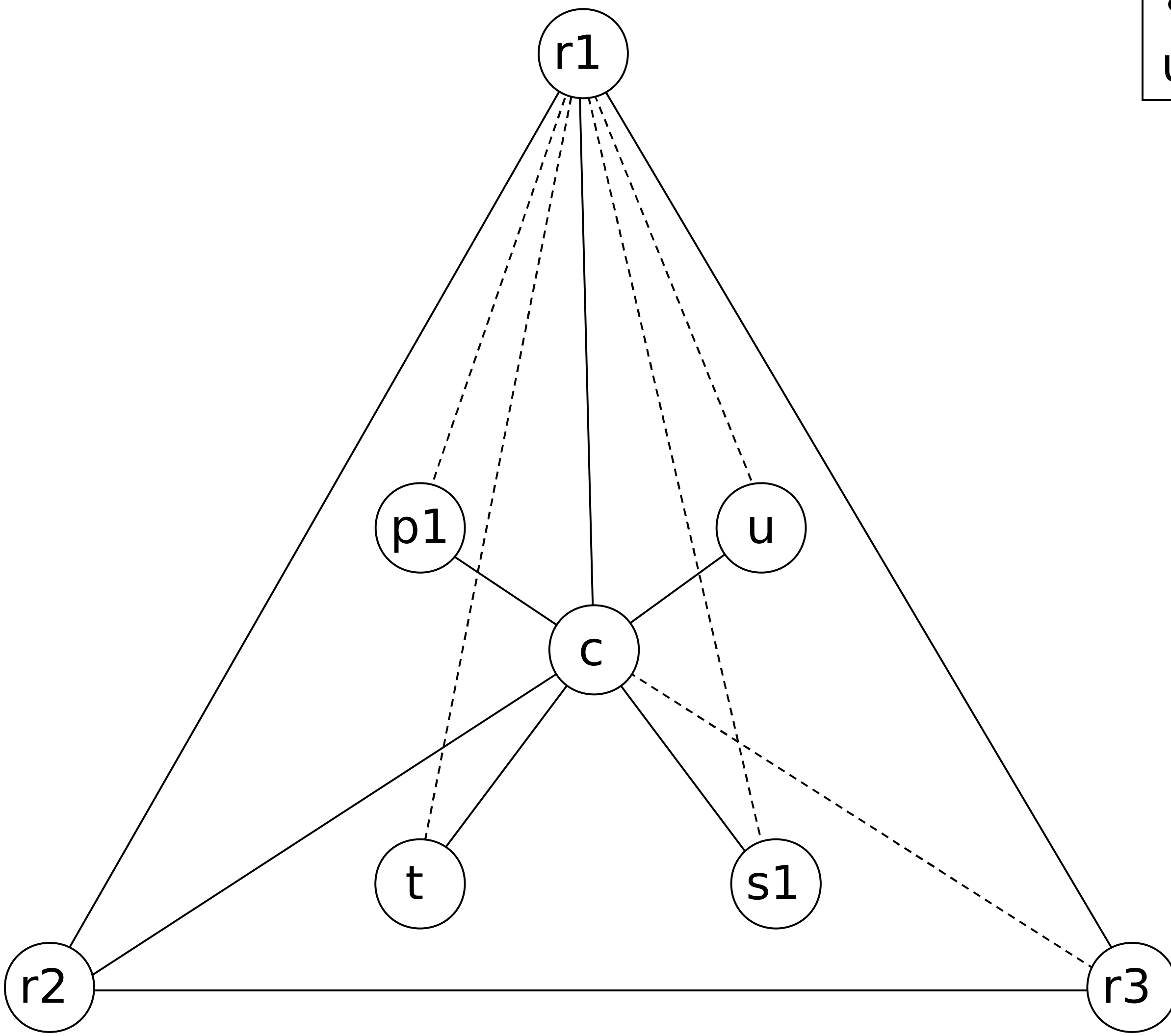
Example 1B

- simplify p2p3p4s2



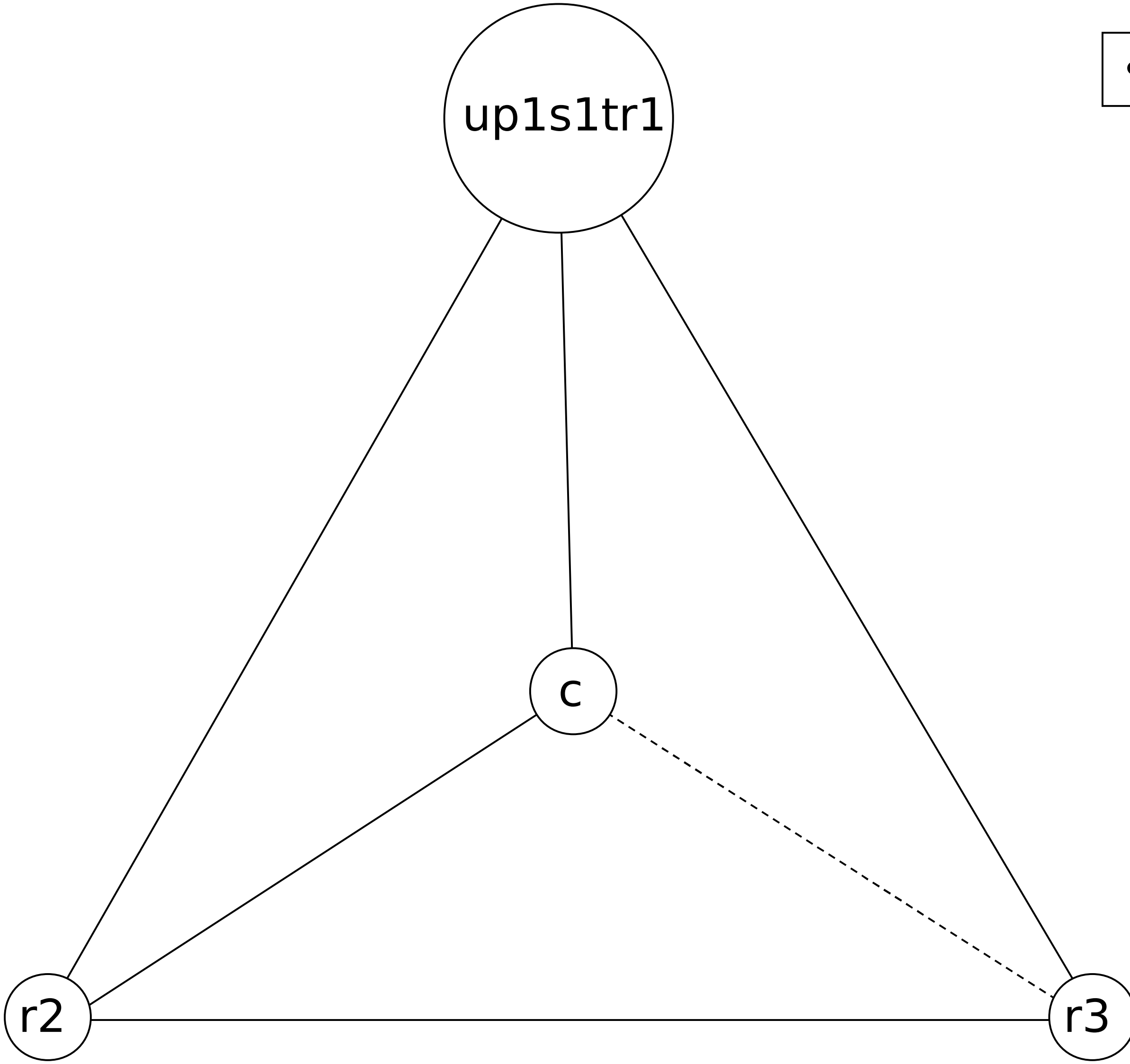
Example 2B

- coalesce up1s1tr1



p2
p3
p4
s2

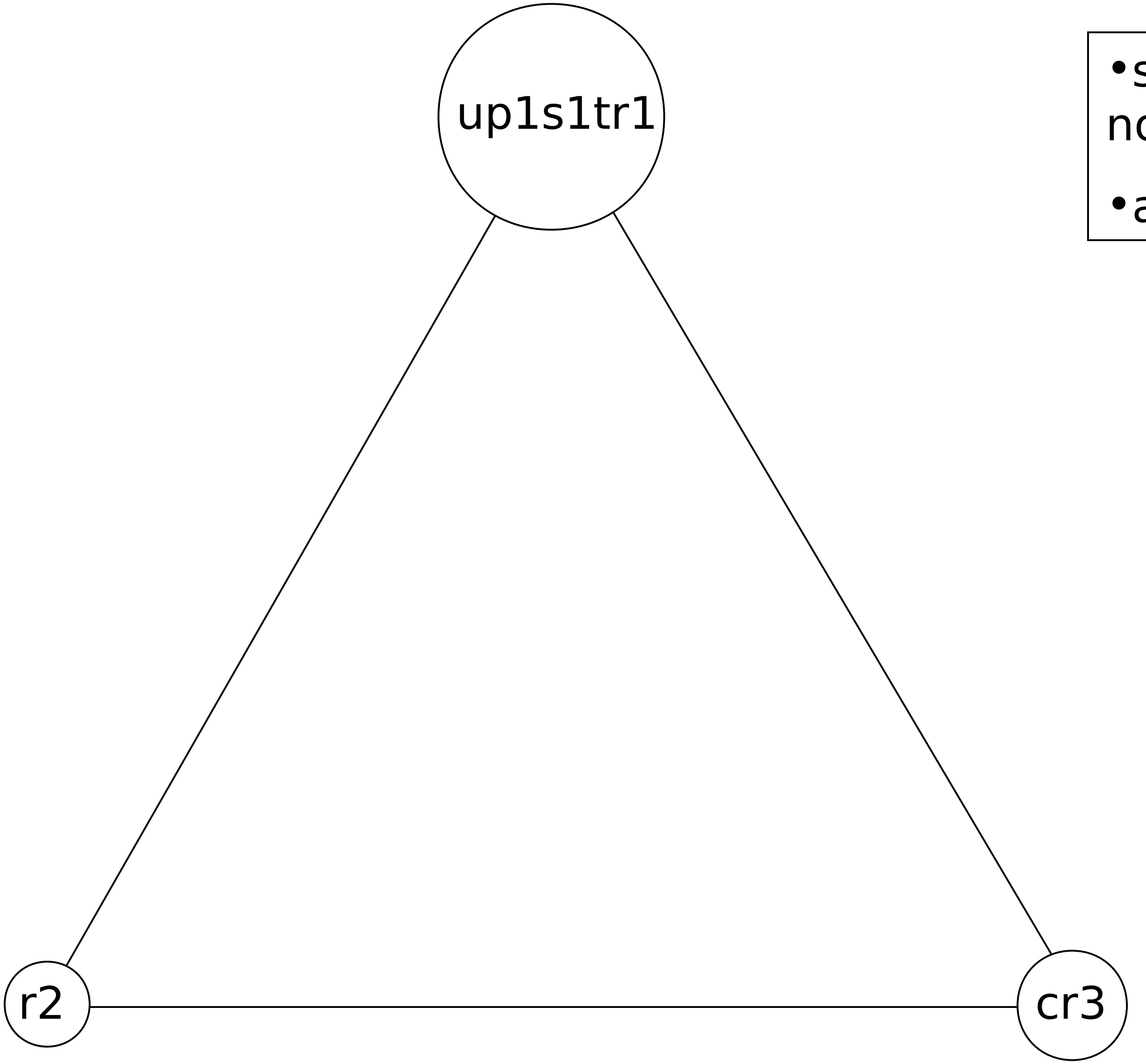
Example 3B



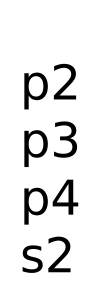
- coalesce cr3

p2
p3
p4
s2

Example 4B

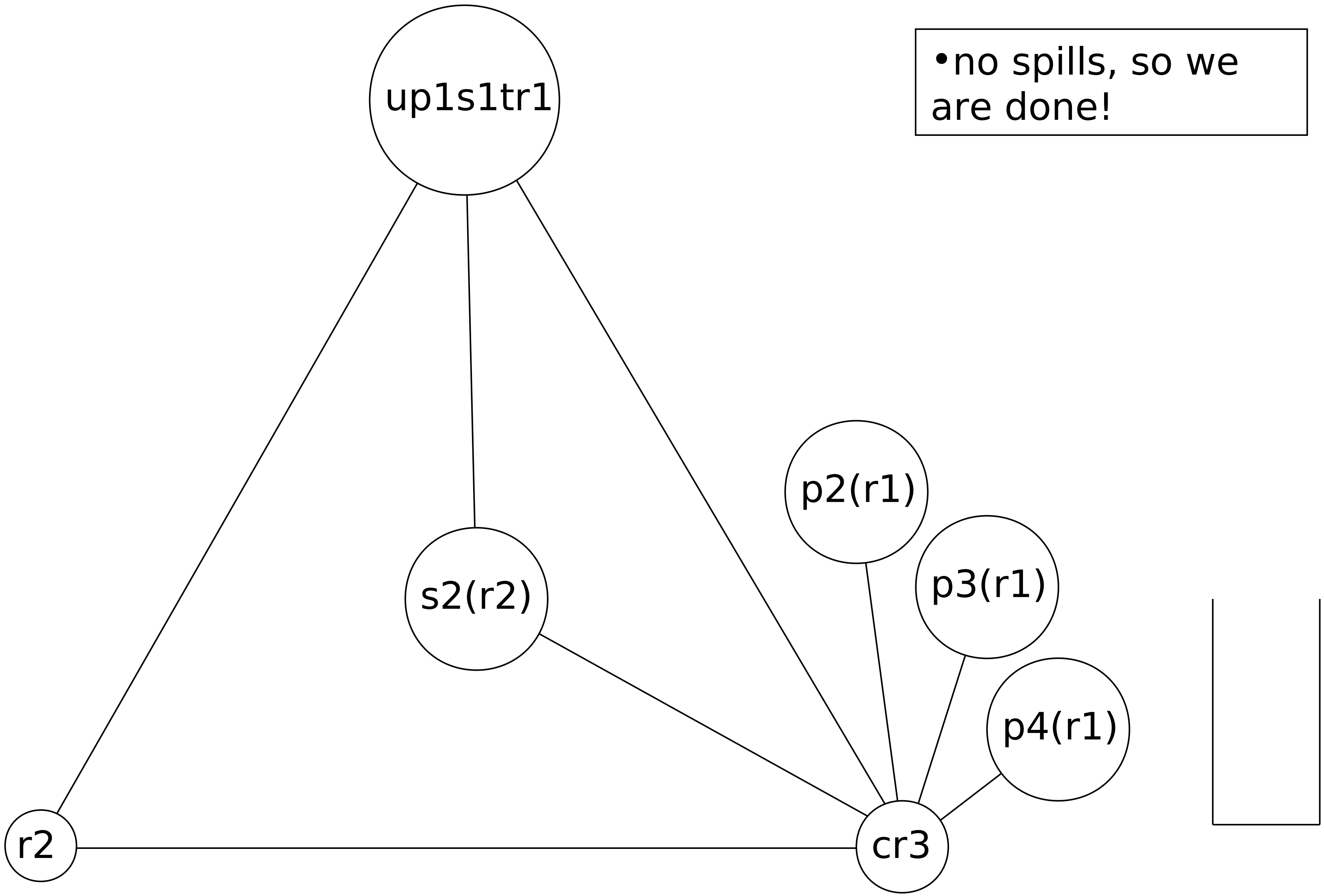


- select to put back nodes and colour
- add p2 p3 p4 s2



Example 5B

- no spills, so we are done!



Final Register Allocation

```
f:  r3 ← r3
    r1 ← r1
    Mp ← r1
    r1 ← Mp
    if r1=0 goto L1
    r1 ← Mp
    r1 ← M[r1]
    call f
    r1 ← r1
    Ms ← r1
    r1 ← Mp
    r1 ← M[r1+4]
    call f
    r1 ← r1
    r2 ← Ms
    r1 ← r2 + r1
    goto L2
L1: r1 ← 1
L2: r1 ← r1
    r3 ← r3
    return
```

eliminate MOVEs with the same source and destination →

```
f:  Mp ← r1
    r1 ← Mp
    if r1=0 goto L1
    r1 ← Mp
    r1 ← M[r1]
    call f
    Ms ← r1
    r1 ← Mp
    r1 ← M[r1+4]
    call f
    r2 ← Ms
    r1 ← r2 + r1
    goto L2
L1: r1 ← 1
L2: return
```