Register Allocation

Mooly Sagiv

html://www.cs.tau.ac.il/~msagiv/courses/wcc08.html

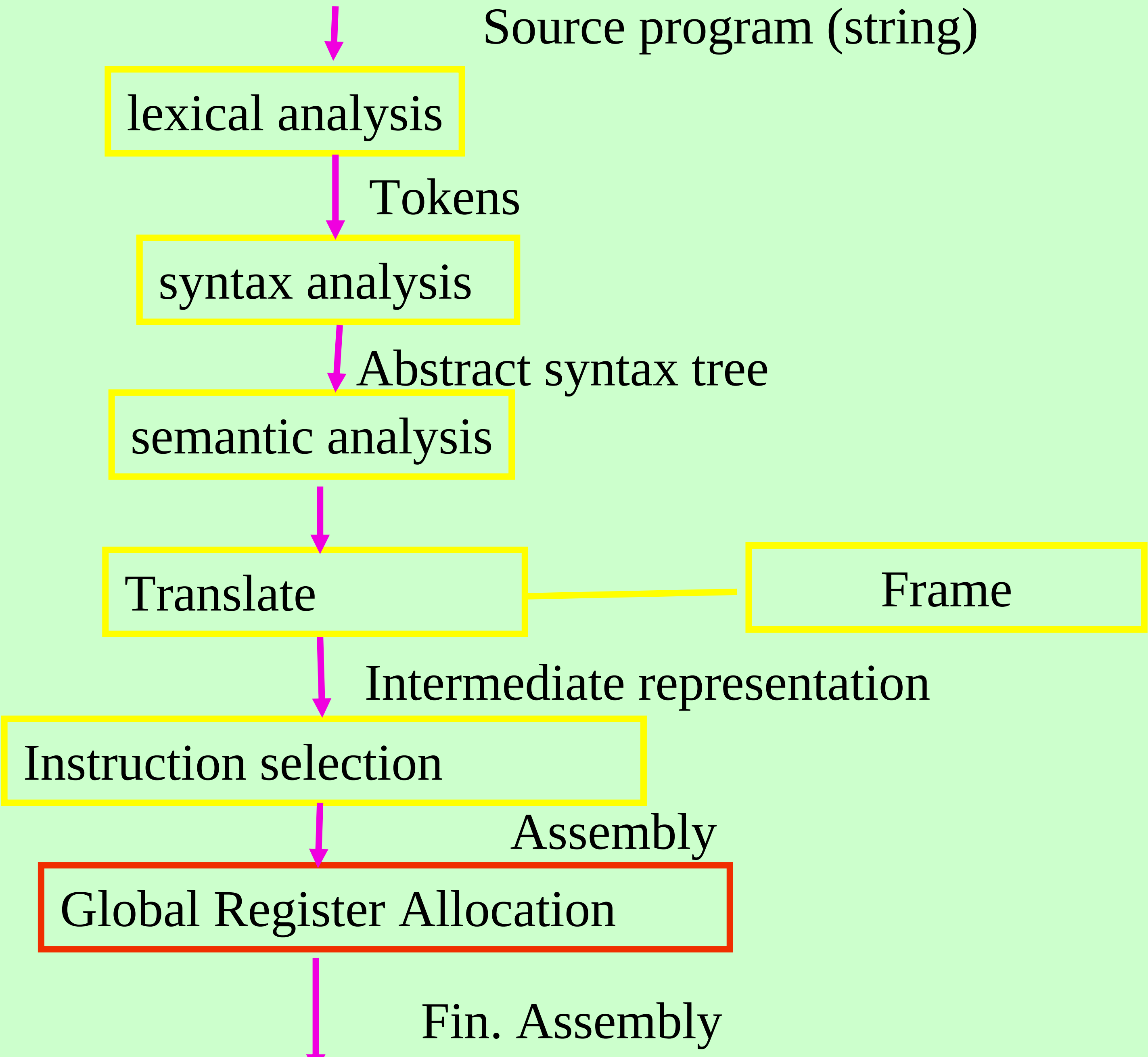
Two Phase Solution
Dynamic Programming
Sethi & Ullman

- Bottom-up (labeling)
  - Compute for every subtree
    - The minimal number of registers needed (weight)
- Top-Down
  - Generate the code using labeling by preferring “heavier” subtrees (larger labeling)

“Global” Register Allocation

- Input:
  - Sequence of machine code instructions (assembly)
    - Unbounded number of temporary registers
- Output
  - Sequence of machine code instructions (assembly)
  - Machine registers
  - Some MOVE instructions removed
  - Missing prologue and epilogue

Basic Compiler Phases



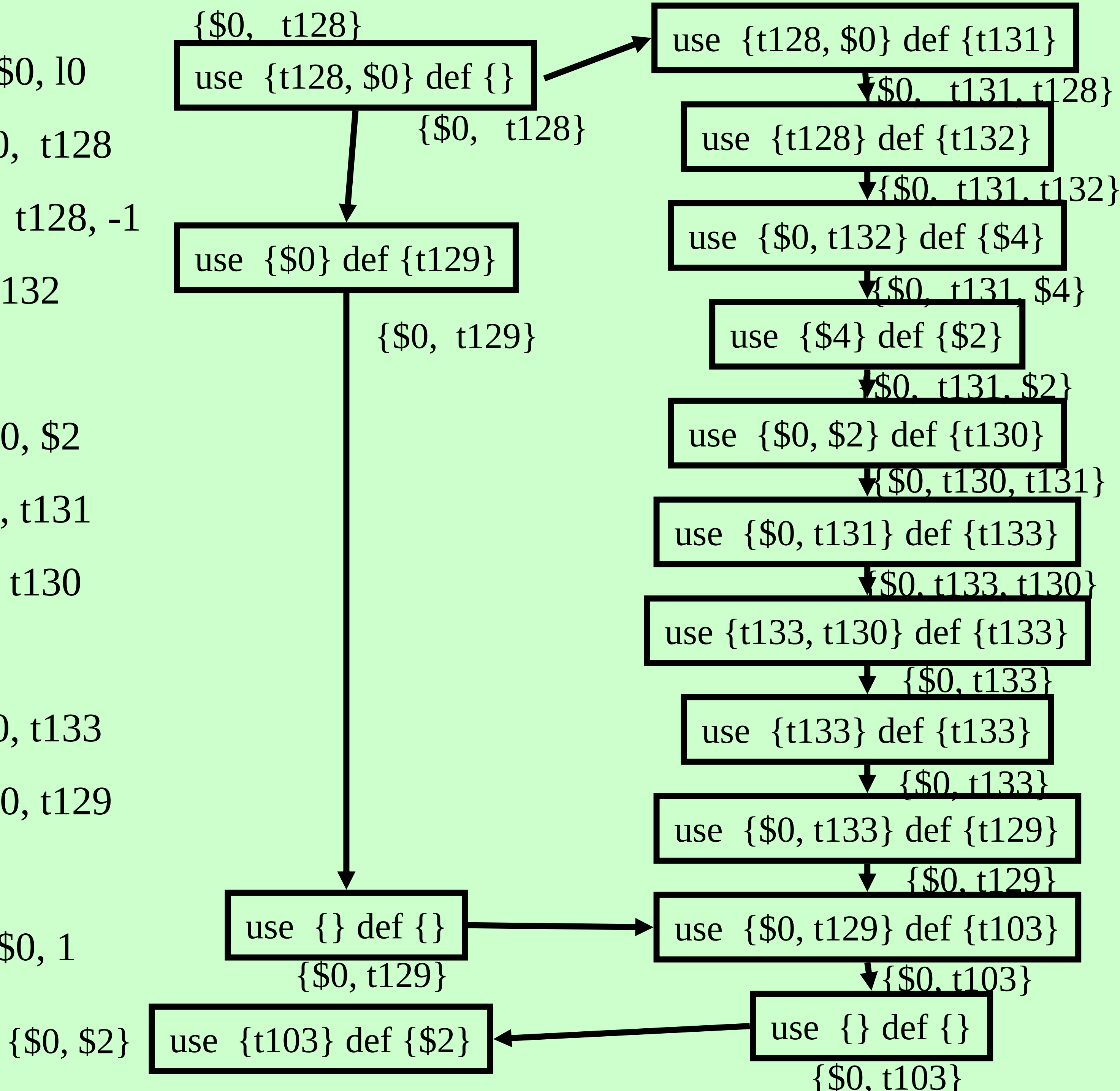
Global Register Allocation Process

Repeat

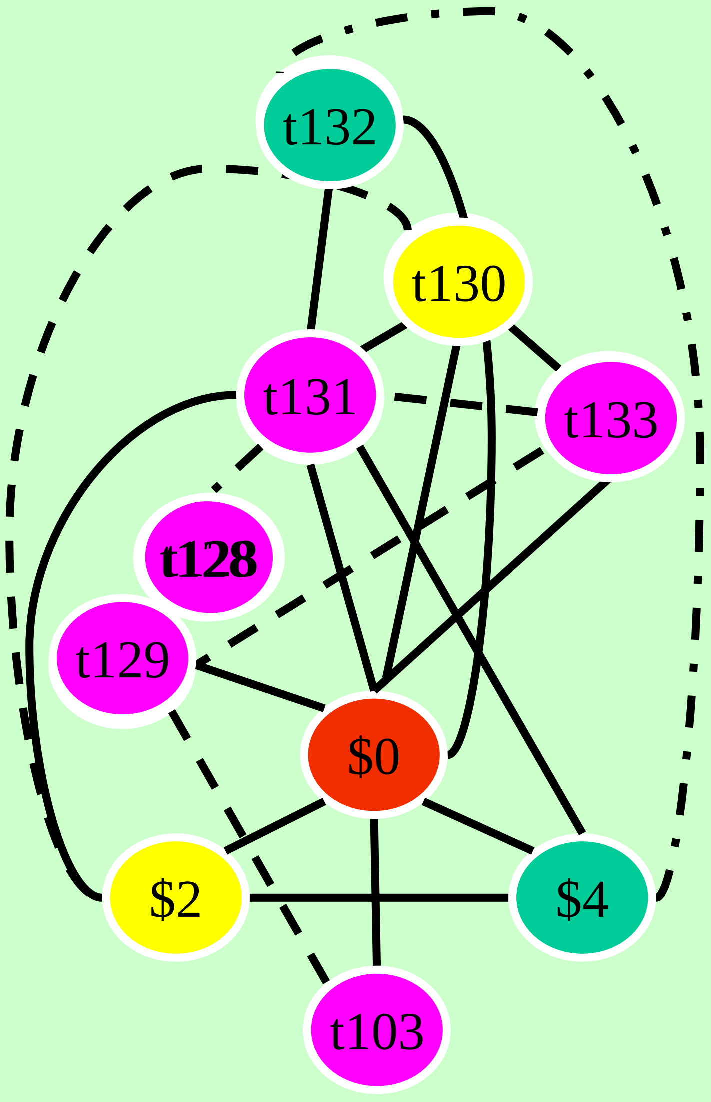
- Construct the interference graph
- Color graph nodes with machine registers
  - Adjacent nodes are not colored by the same register
- Spill a temporary into memory

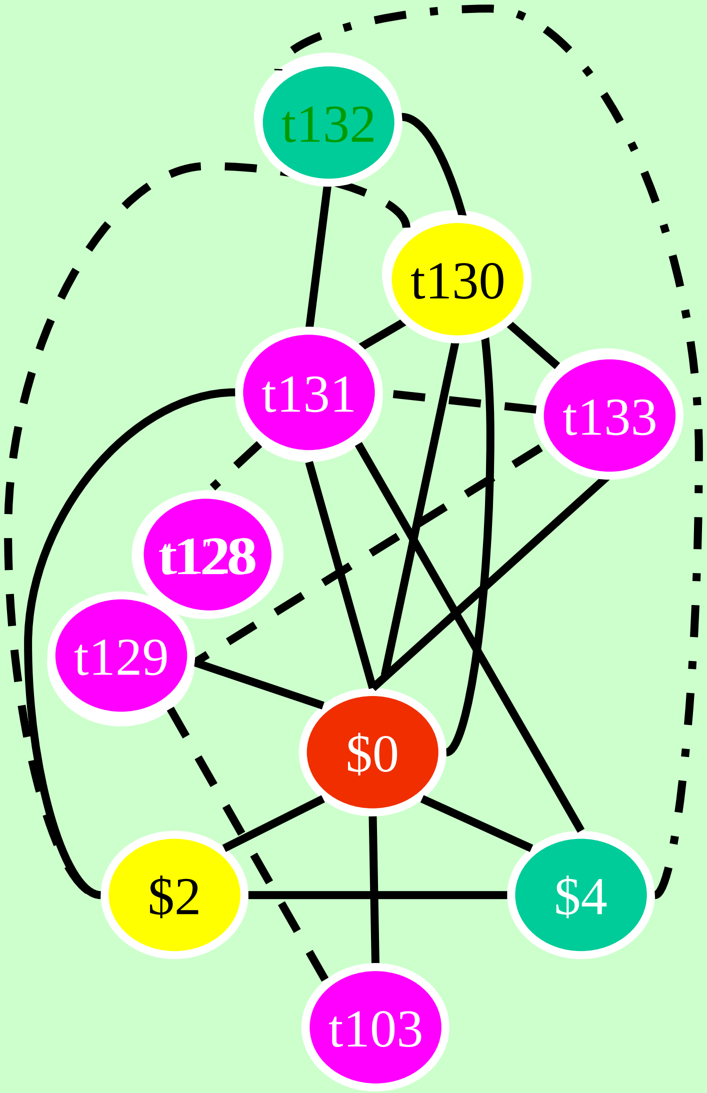
Until no more spill

```
l3:     beq  t128, $0, l0
l1:     or  t131, $0,  t128
        addi  t132,  t128, -1
        or $4, $0, t132
        jal  nfactor
        or  t130,  $0, $2
        or t133, $0, t131
        mult  t133, t130
        mflo t133
        or  t129, $0, t133
l2:     or  t103,  $0, t129
        b  lend
l0:     addi t129, $0, 1
        b l2
```



```
l3:     beq  t128, $0, l0 /* $0, t128 */
l1:     or  t131, $0,  t128 /* $0, t128, t131 */
        addi  t132,  t128, -1 /* $0, t131, t132 */
        or $4, $0, t132  /* $0, $4, t131 */
        jal  nfactor     /* $0, $2, t131 */
        or  t130,  $0, $2  /* $0, t130, t131 */
        or t133, $0, t131 /* $0, t130, t133 */
        mult  t133, t130 /* $0, t133 */
        mflo t133   /* $0, t133 */
        or  t129, $0, t133  /* $0, t129 */
l2:     or  t103,  $0, t129  /* $0, t103 */
        b  lend  /* $0, t103 */
l0:     addi t129, $0, 1    /* $0, t129 */
        b l2    /* $0, t129 */
```






```
l3:    beq  t128, $0, l0
l1:    or  t131, $0,  t128
       addi  t132,  t128, -1
       or $4, $0, t132
       jal  nfactor
       or  t130,  $0, $2
       or t133, $0, t131
       mult  t133, t130
       mflo t133
       or  t129, $0, t133
l2:    or  t103,  $0, t129
       b  lend
l0:    addi t129, $0, 1
       b l2
```

Challenges

- The Coloring problem is computationally hard
- The number of machine registers may be small
- Avoid too many MOVEs
- Handle “pre-colored” nodes

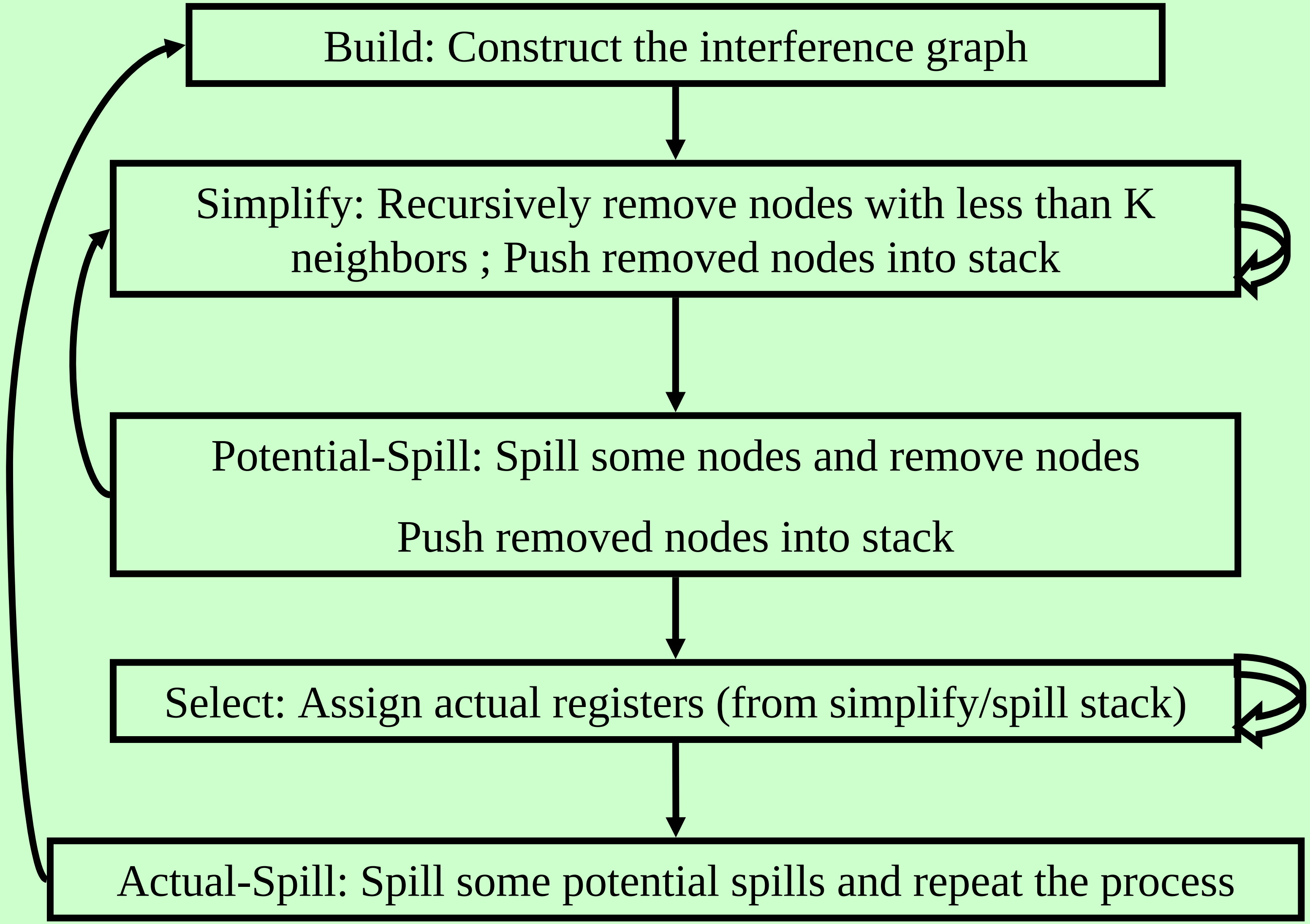
Theorem
[Kempe 1879]

- Assume:
  - An undirected graph G(V, E)
  - A node $v \in V$ with less than K neighbors
  - G – {v} is K colorable
- Then, G is K colorable

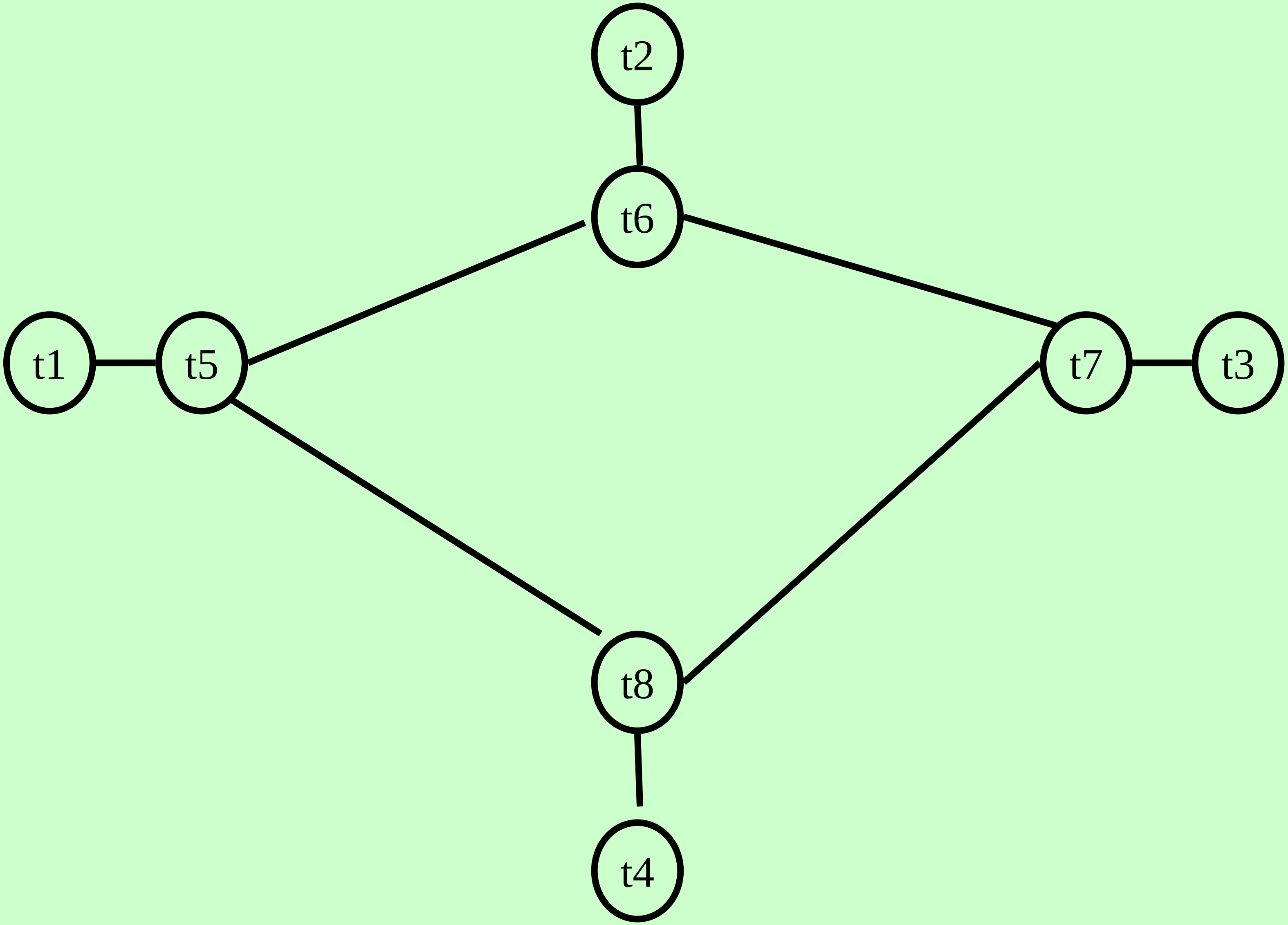
Coloring by Simplification [Kempe 1879]

- K
  - the number of machine registers
- G(V, E)
  - the interference graph
- Consider a node $v \in V$ with less than K neighbors:
  - Color G – v in K colors
  - Color v in a color different than its (colored) neighbors

Graph Coloring by Simplification



Artificial Example K=2



Coalescing

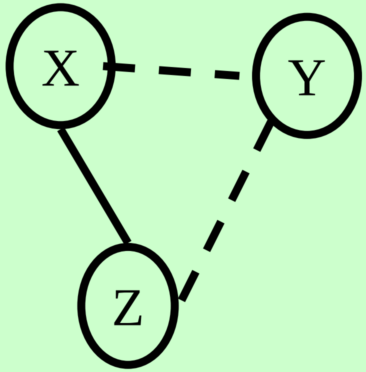
- MOVs can be removed if the source and the target share the same register
- The source and the target of the move can be merged into a single node (unifying the sets of neighbors)
- May require more registers
- Conservative Coalescing
  - Merge nodes only if the resulting node has fewer than K neighbors with degree $\geqslant$ K (in the resulting graph)

Constrained Moves

- A instruction T ← S is constrained
  - if S and T interfere
- May happen after coalescing

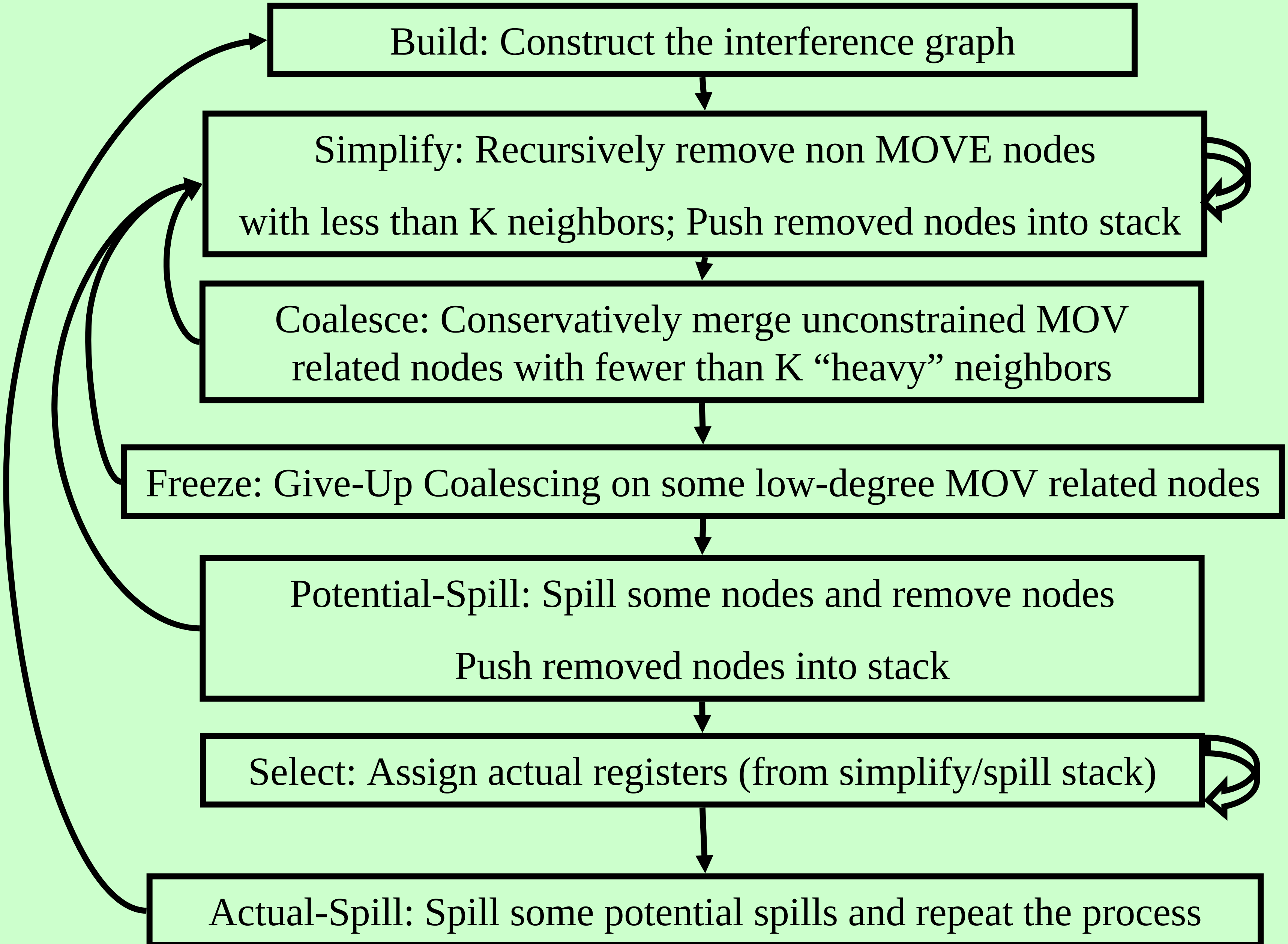
X ←Y   /* X, Y, Z */

Y←Z



- Constrained MOVs are not coalesced

Graph Coloring with Coalescing



Spilling

- Many heuristics exist
  - Maximal degree
  - Live-ranges
  - Number of uses in loops
- The whole process need to be repeated after an actual spill

Pre-Colored Nodes

- Some registers in the intermediate language are pre-colored:
  - correspond to real registers
    (stack-pointer, frame-pointer, parameters,)
- Cannot be Simplified, Coalesced, or Spilled (infinite degree)
- Interfered with each other
- But normal temporaries can be coalesced into pre-colored registers
- Register allocation is completed when all the nodes are pre-colored

Caller-Save and Callee-Save Registers

- callee-save-registers (MIPS 16-23)
  - Saved by the callee when modified
  - Values are automatically preserved across calls
- caller-save-registers
  - Saved by the caller when needed
  - Values are not automatically preserved
- Usually the architecture defines caller-save and callee-save registers
  - Separate compilation
  - Interoperability between code produced by different compilers/languages
- But compilers can decide when to use callller/callee registers

Caller-Save vs. Callee-Save Registers

```
int foo(int a)    {
    int b=a+1;
    f1();
    g1(b);
    return(b+2);
}
```

```
void bar (int y) {
    int x=y+1;
    f2(y);
    g2(2);
}
```

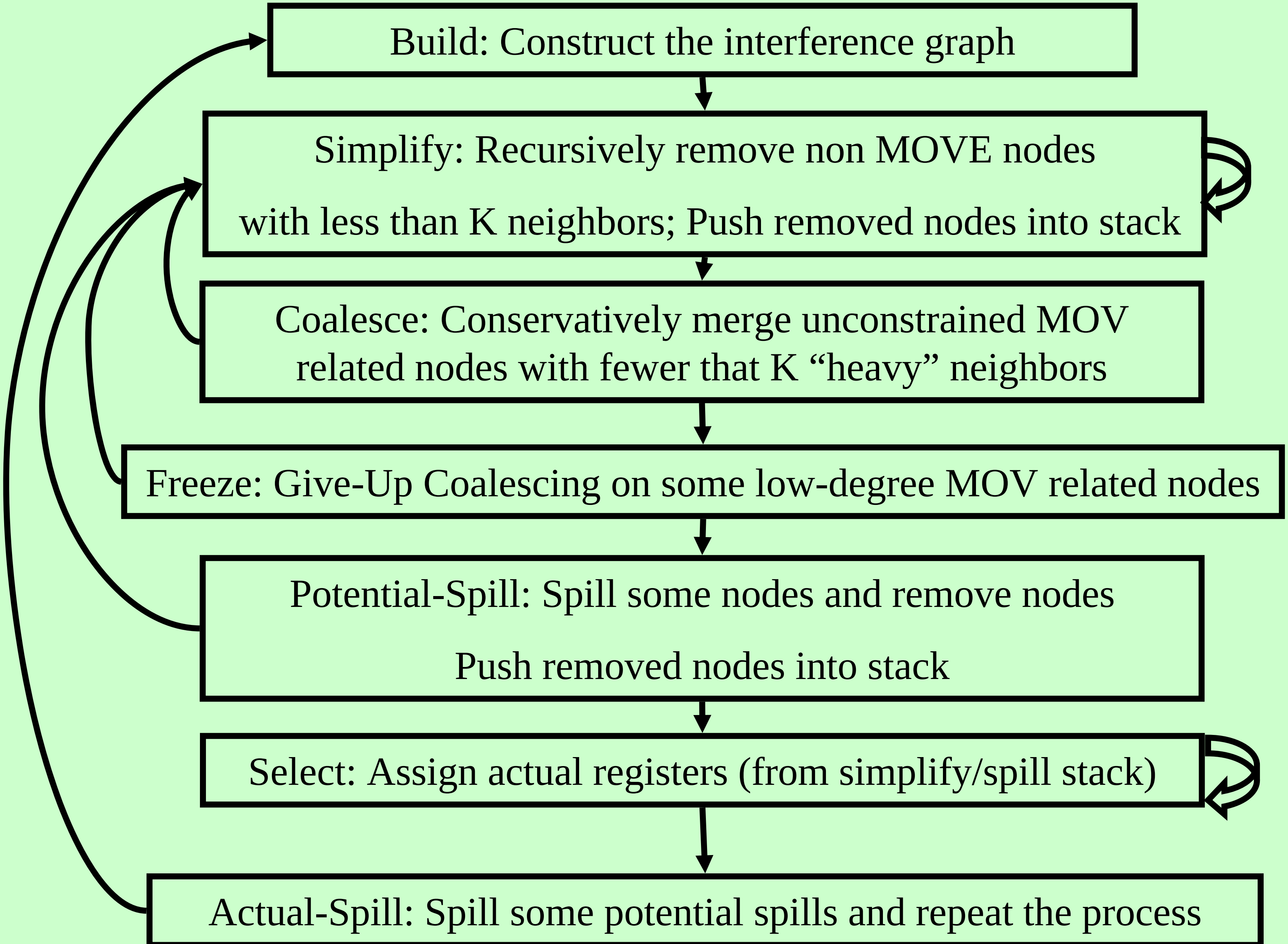
Saving Callee-Save Registers

enter: def(r_7)	enter: def(r_7)
	$t_{231} \leftarrow r_7$
…	…
	$r_7 \leftarrow t_{231}$
exit: use(r_7)	exit: use(r_7)

A Complete Example

```
enter:
        c := r3        r1, r2   caller save
        a := r1        r3        callee-save
        b := r2
        d := 0
        e := a
loop:
        d := d+b
        e := e-1
        if e>0 goto loop
        r1 := d
        r3 := c
  return  /* r1,r3 */
```

Graph Coloring with Coalescing



A Complete Example

enter:

c := r3   r1, r2   caller save

a := r1   r3   callee-save

b := r2

d := 0

e := a

loop:

d := d+b

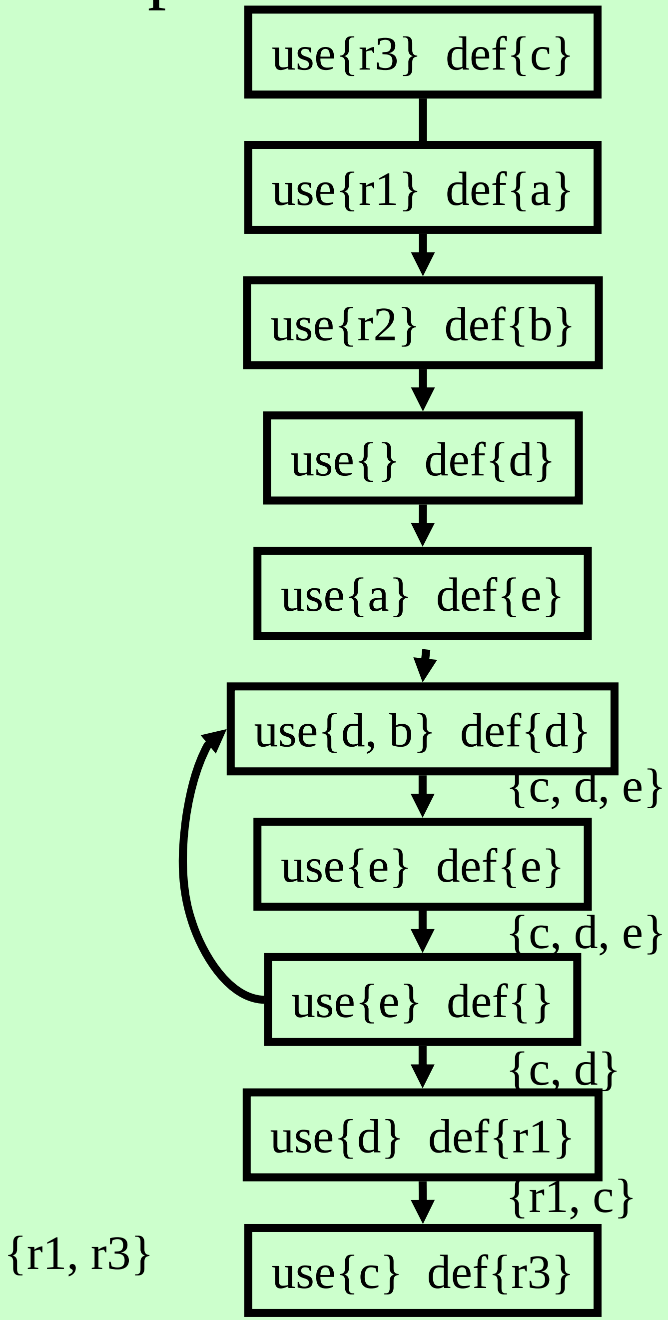
e := e-1

if e>0 goto loop

r1 := d

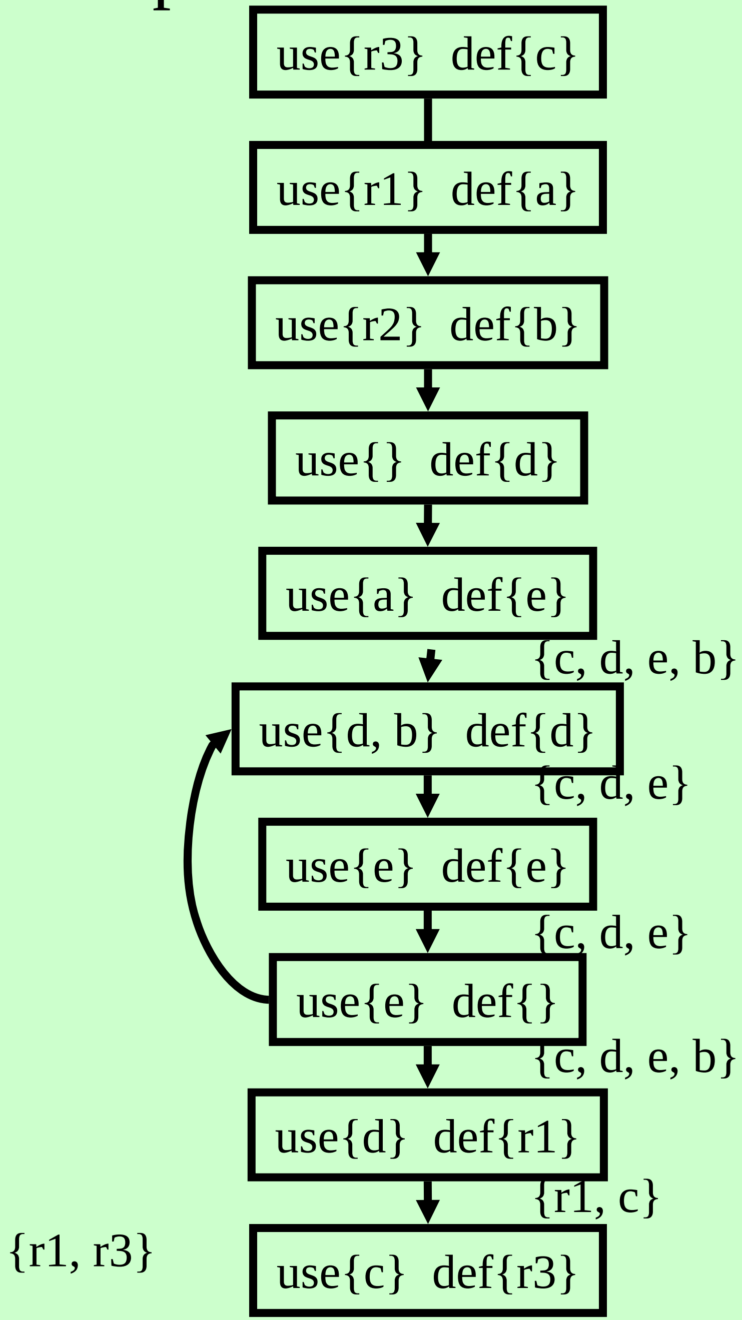
r3 := c

return /* r1,r3 */



A Complete Example

```
enter:
        c := r3
        a := r1
        b := r2
        d := 0
        e := a
loop:
        d := d+b
        e := e-1
        if e>0 goto loop
        r1 := d
        r3 := c
  return  /* r1,r3 */
```



A Complete Example

enter:

c := r3

a := r1

b := r2

d := 0

e := a

loop:

d := d+b

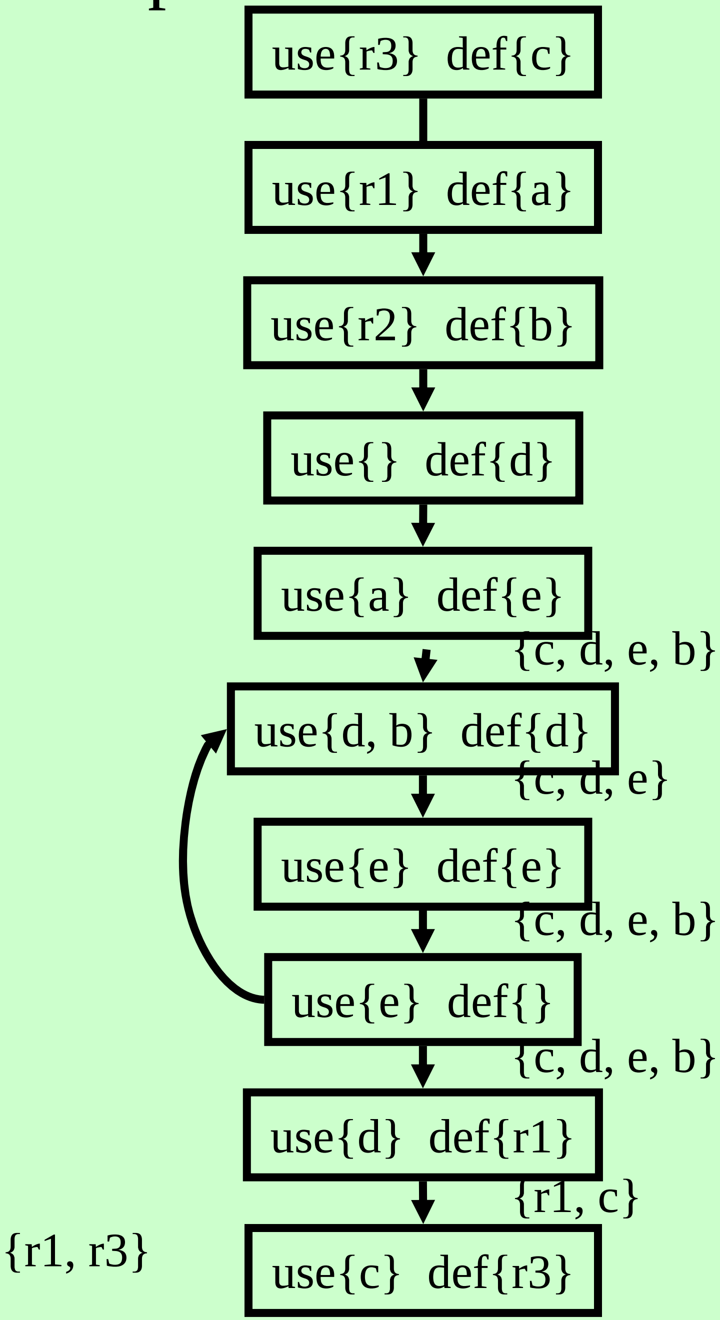
e := e-1

if e>0 goto loop

r1 := d

r3 := c

return /* r1,r3 */



A Complete Example

enter:

c := r3

a := r1

b := r2

d := 0

e := a

loop:

d := d+b

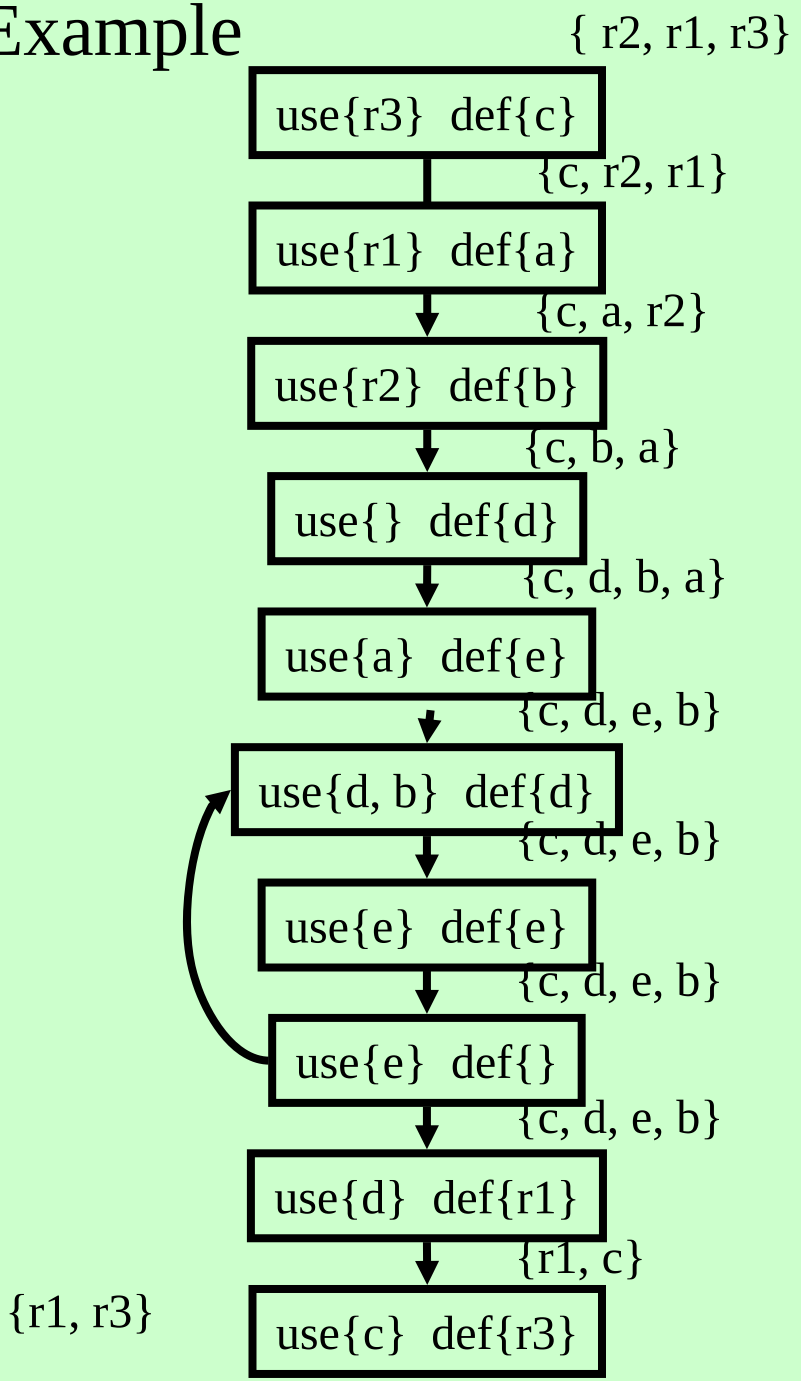
e := e-1

if e>0 goto loop

r1 := d

r3 := c

return /* r1,r3 */

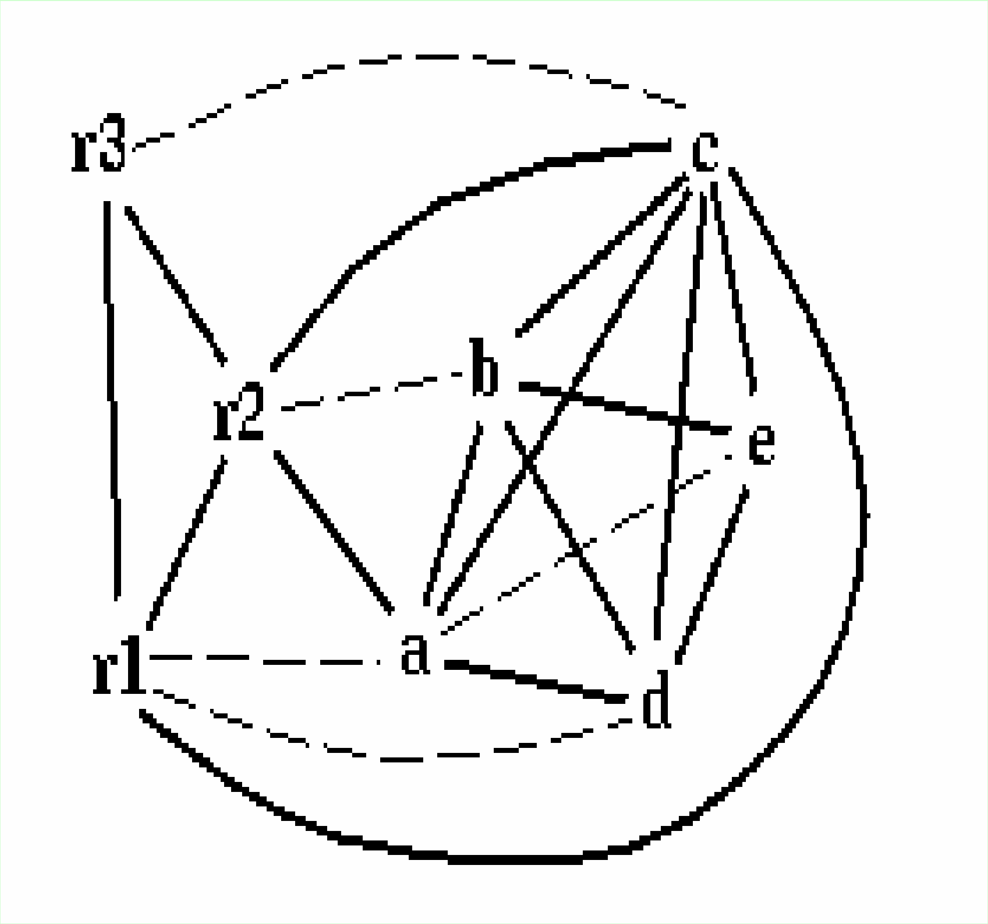


Live Variables Results

```
enter:
        c := r3
        a := r1
        b := r2
        d := 0
        e := a
loop:
        d := d+b
        e := e-1
        if e>0 goto loop
        r1 := d
        r3 := c
  return  /* r1,r3 */
```

```
enter:                 /* r2, r1, r3 */
        c := r3  /* c, r2, r1 */
        a := r1  /* a, c, r2 */
        b := r2  /* a, c, b */
        d := 0   /* a, c, b, d */
        e := a   /* e, c, b, d */
loop:
        d := d+b  /* e, c, b, d */
        e := e-1 /* e, c, b, d */
        if e>0 goto loop /* c, d */
        r1 := d  /* r1, c */
        r3 := c /* r1, r3 */
  return  /* r1, r3 */
```

```
enter:                /* r2, r1, r3 */
        c := r3  /* c, r2, r1 */
        a := r1  /* a, c, r2 */
        b := r2  /* a, c, b */
        d := 0   /* a, c, b, d */
        e := a   /* e, c, b, d */
loop:
        d := d+b  /* e, c, b, d */
        e := e-1 /* e, c, b, d */
        if e>0 goto loop /* c, d */
        r1 := d  /* r1, c */
        r3 := c /* r1, r3 */
  return  /* r1,r3 */
```

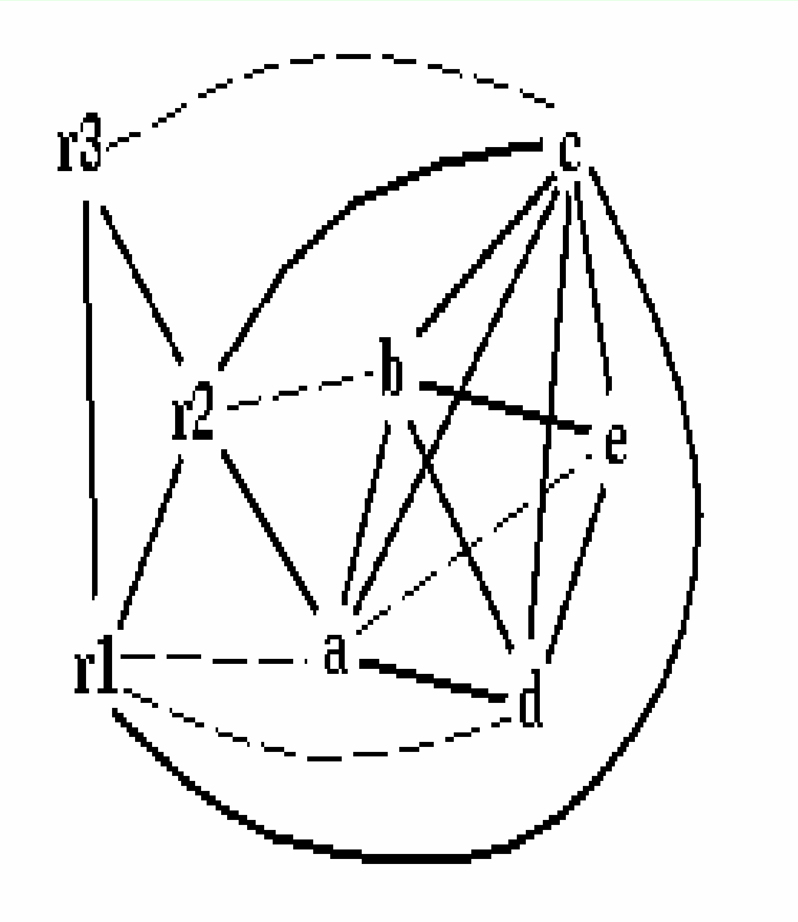


spill priority = (uo + 10 ui)/deg

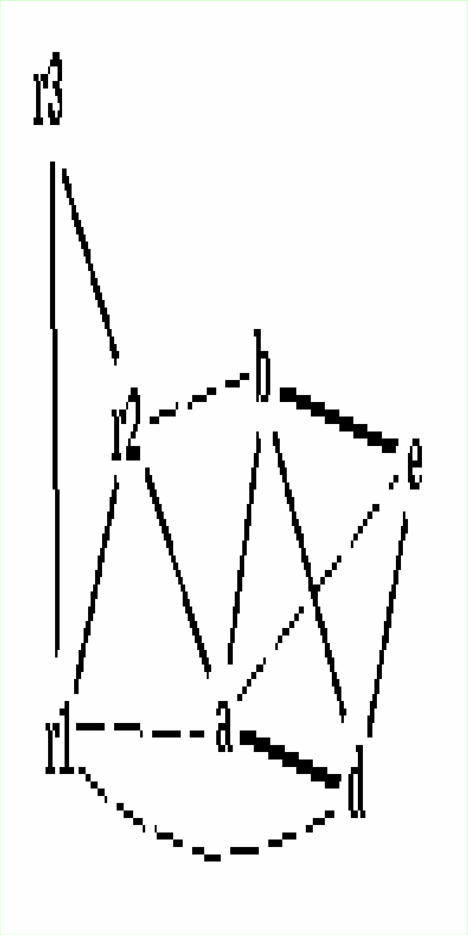
```
enter:                    /* r2, r1, r3 */
        c := r3  /* c, r2, r1 */
        a := r1  /* a, c, r2 */
        b := r2  /* a, c, b */
        d := 0   /* a, c, b, d */
        e := a   / * e, c, b, d */
loop:
        d := d+b  /* e, c, b, d */
        e := e-1 /* e, c, b, d */
        if e>0 goto loop /* c, d */
        r1 := d  /* r1, c */
        r3 := c /* r1, r3 */
 return  /* r1,r3 */
```

	use+ def outside loop	use+ def within loop	deg	spill priority
a	2	0	4	0.5
b	1	1	4	2.75
c	2	0	6	0.33
d	2	2	4	5.5
e	1	3	3	10.3

Spill C



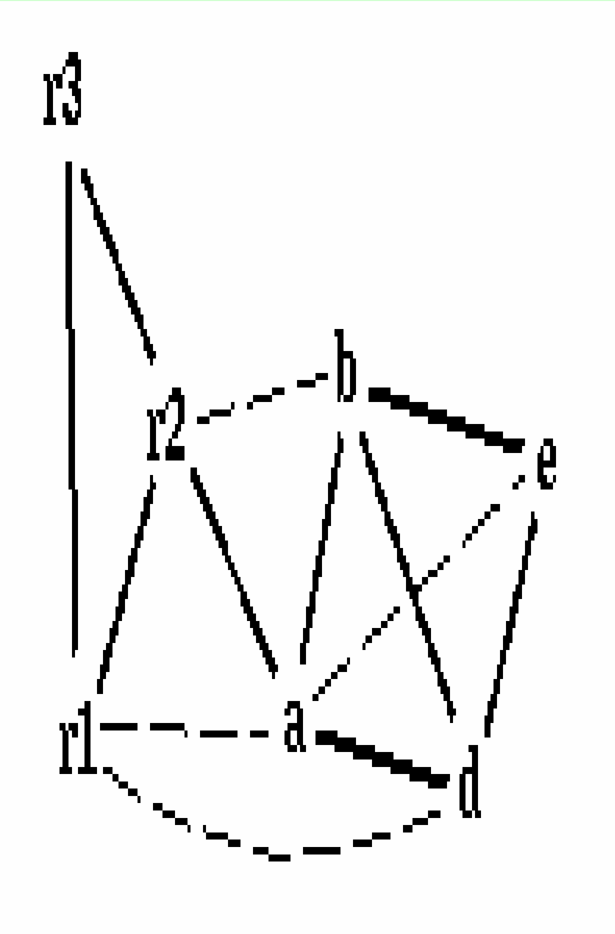
stack



stack

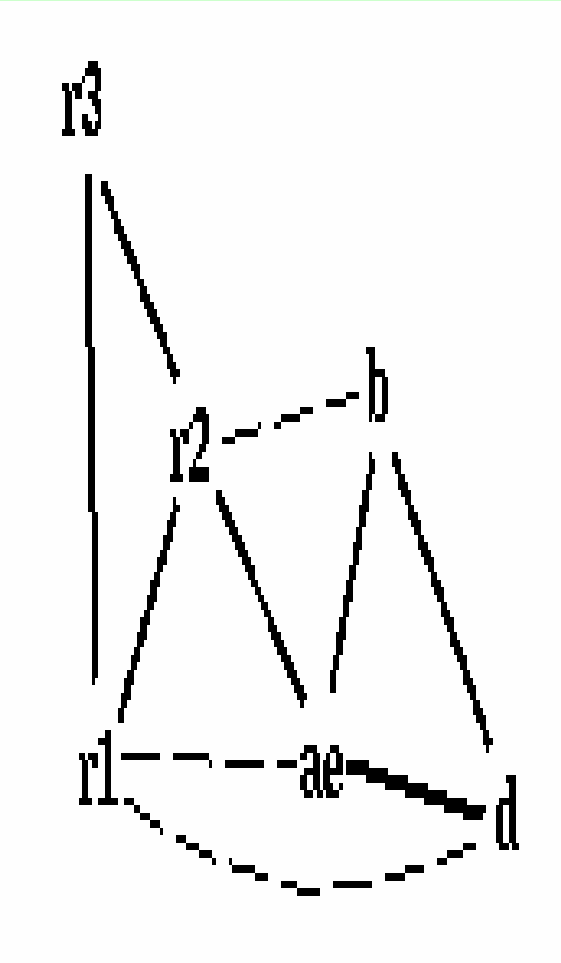
c

Coalescing *a*+*e*



stack

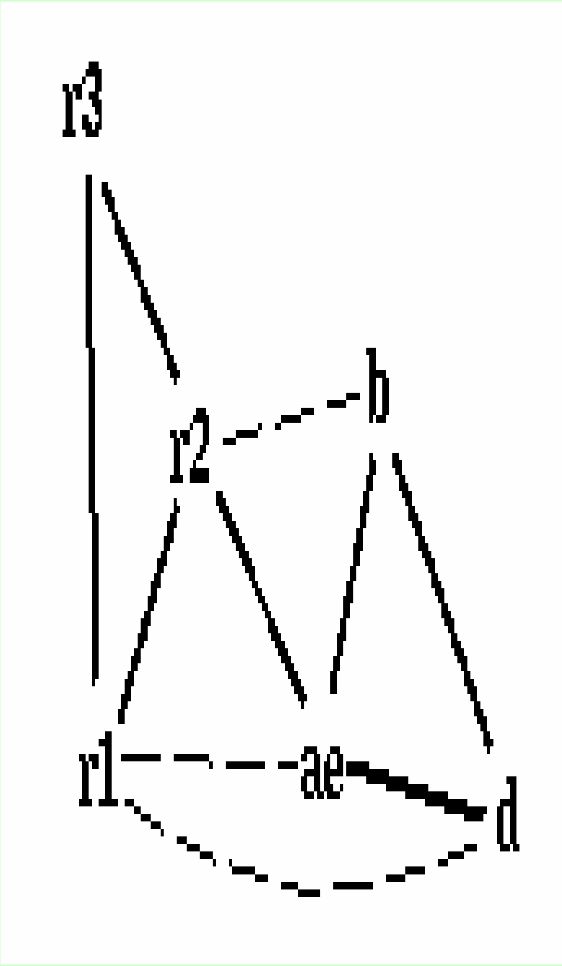
c



stack

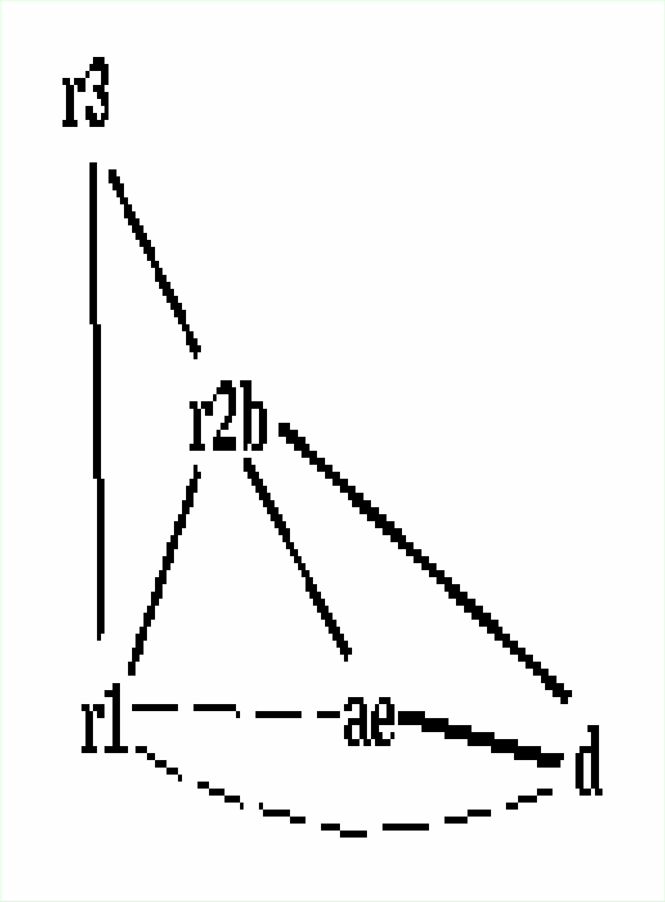
c

Coalescing *b+r2*



stack

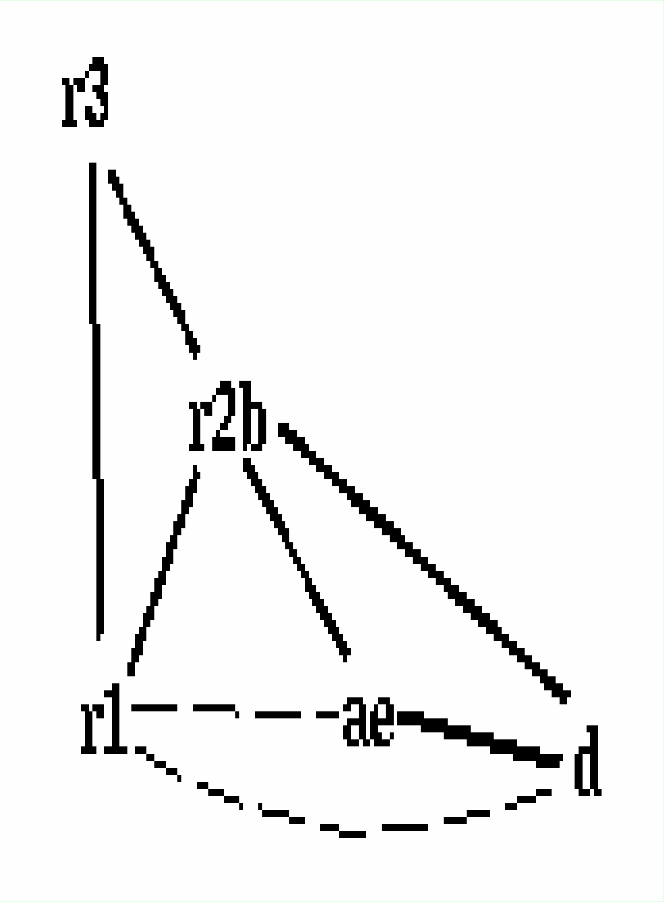
c



stack

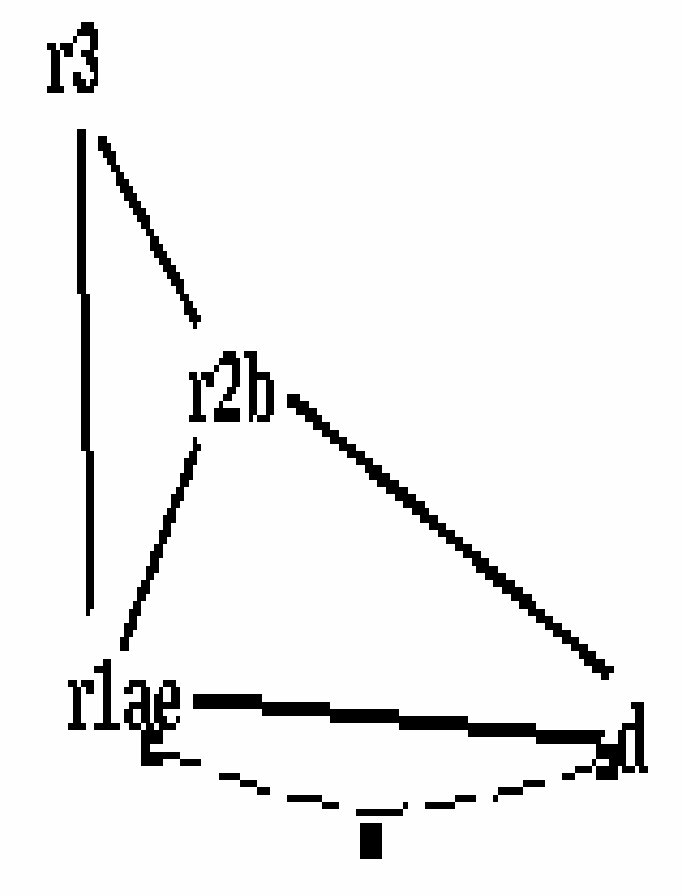
c

Coalescing *ae+r1*



stack

c

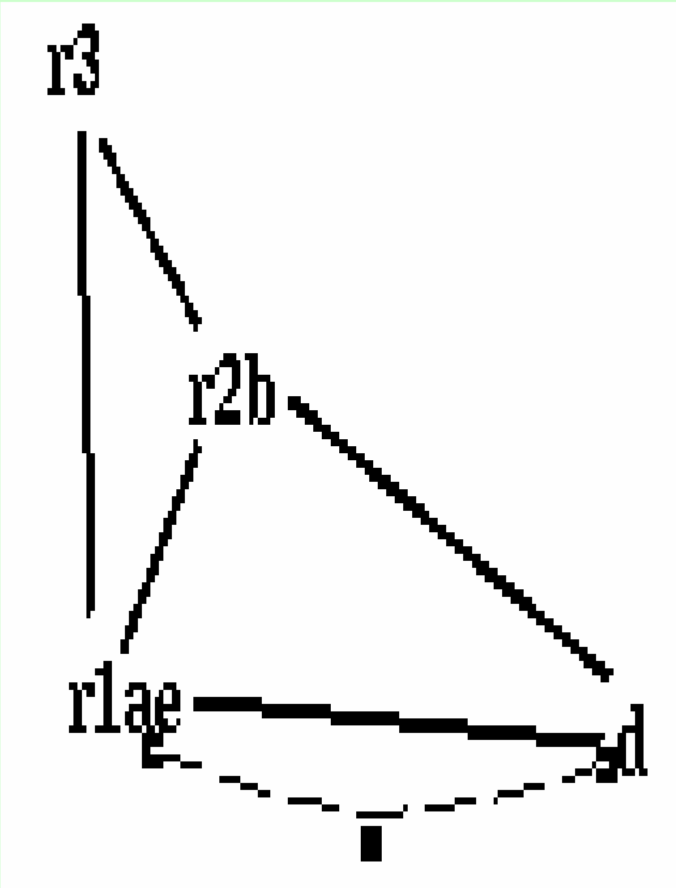


stack

c

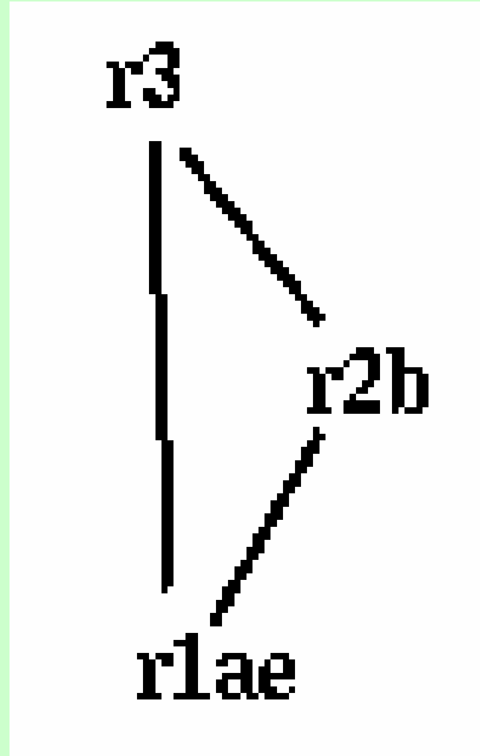
r1ae and d are constrained

Simplifying *d*



stack

c

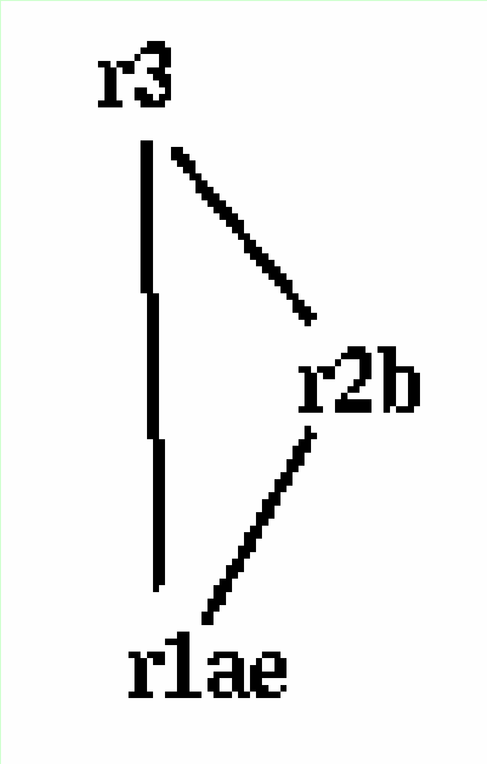


stack

d

c

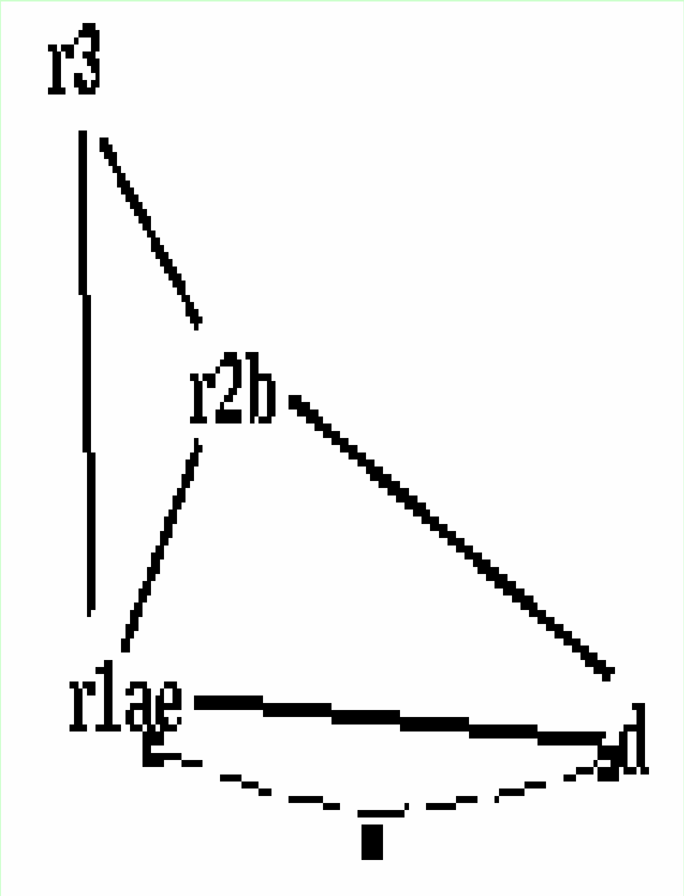
Pop *d*



stack

d

c

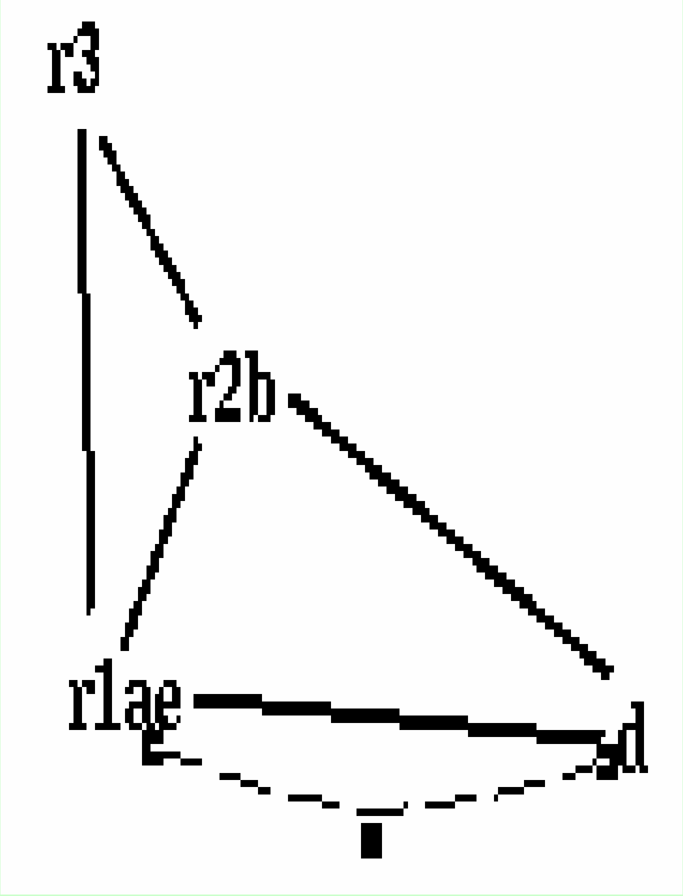


stack

c

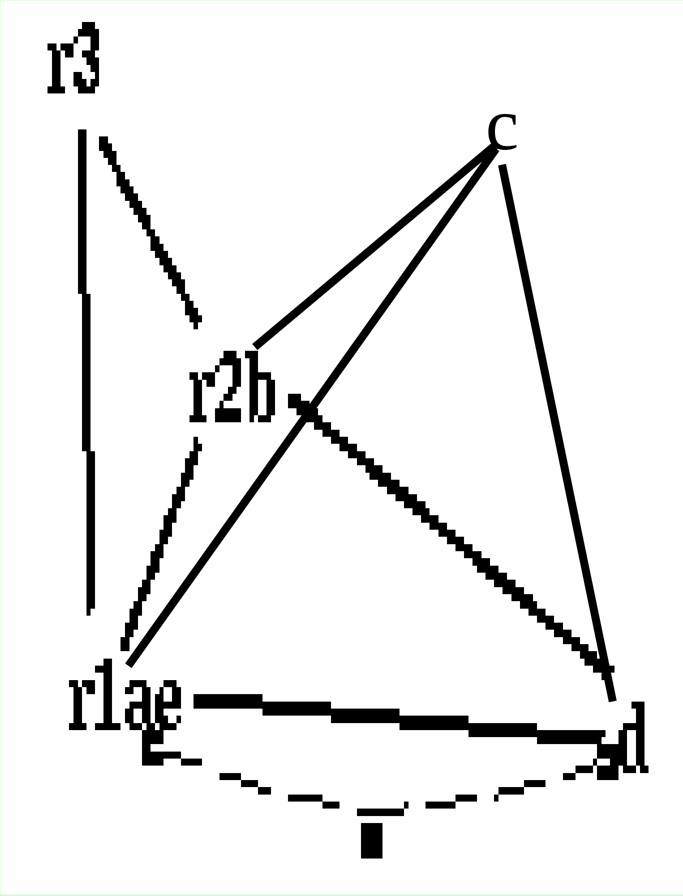
d is assigned to r3

Pop *c*



stack

c



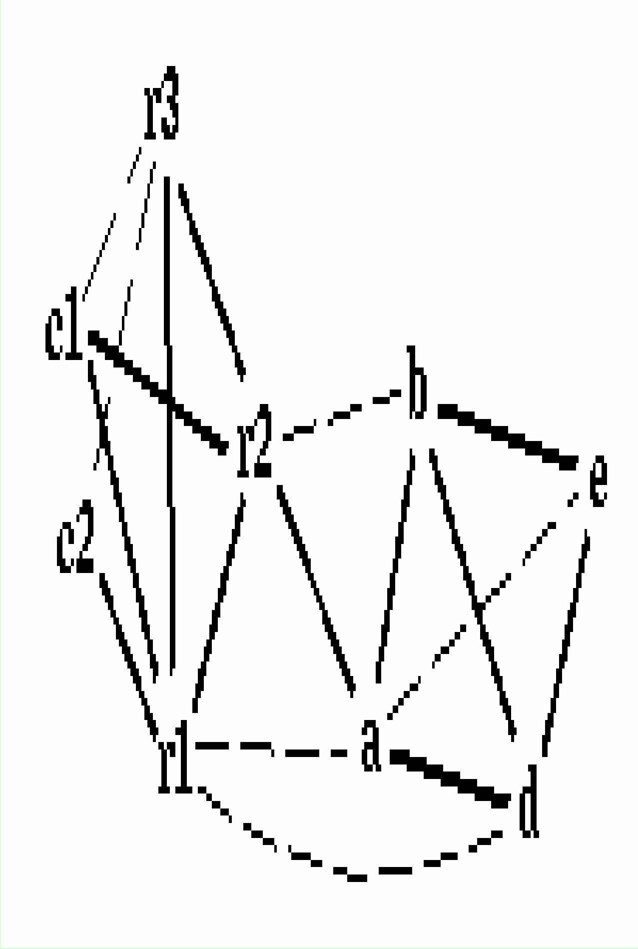
stack

actual spill!

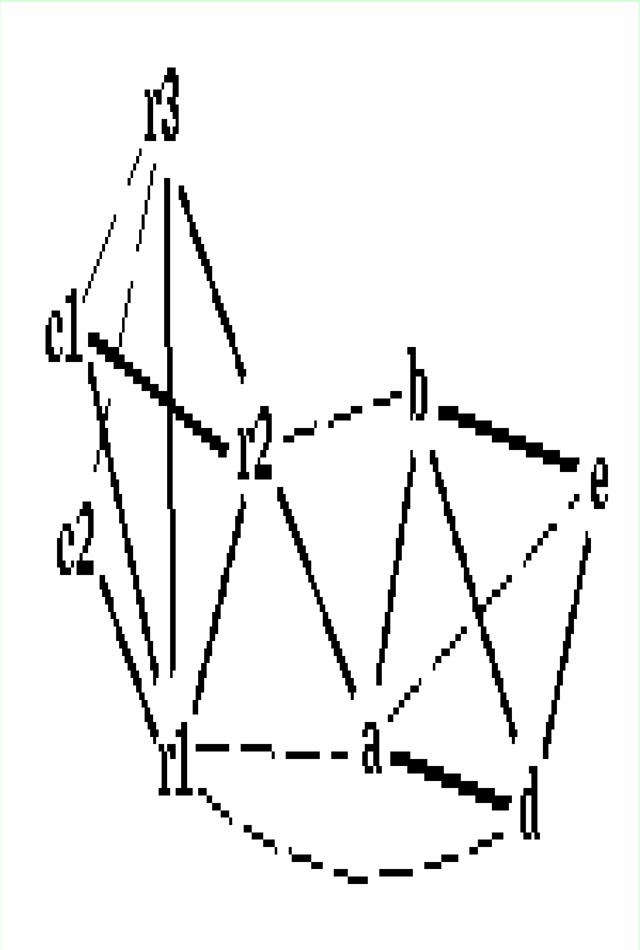
```
enter:                  /* r2, r1, r3 */
        c := r3   /* c, r2, r1 */
        a := r1   /* a, c, r2 */
        b := r2   /* a, c, b */
        d := 0    /* a, c, b, d */
        e := a    / * e, c, b, d */
loop:
        d := d+b  /* e, c, b, d */
        e := e-1 /* e, c, b, d */
        if e>0 goto loop /* c, d */
        r1 := d   /* r1, c */
        r3 := c /* r1, r3 */
  return  /* r1,r3 */
```

```
enter:                  /* r2, r1, r3 */
        c1 := r3  /* c1, r2, r1 */
        M[c_loc] := c1 /* r2 */
        a := r1   /* a, r2 */
        b := r2   /* a, b */
        d := 0    /* a, b, d */
        e := a    / * e, b, d */
loop:
        d := d+b  /* e, b, d */
        e := e-1 /* e, b, d */
        if e>0 goto loop /* d */
        r1 := d   /* r1 */
        c2 := M[c_loc] /* r1, c2 */
        r3 := c2 /* r1, r3 */
  return  /* r1,r3 */
```

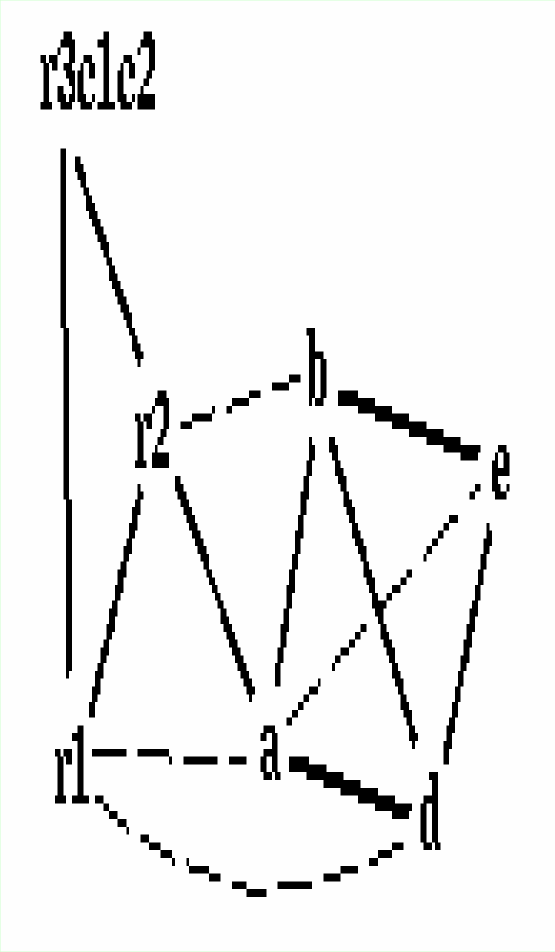
```
enter:                  /* r2, r1, r3 */
        c1 := r3  /* c1, r2, r1 */
        M[c_loc] := c1 /* r2 */
        a := r1  /* a, r2 */
        b := r2  /* a, b */
        d := 0   /* a, b, d */
        e := a   / * e, b, d */
loop:
        d := d+b  /* e, b, d */
        e := e-1 /* e, b, d */
        if e>0 goto loop /* d */
        r1 := d  /* r1 */
        c2 := M[c_loc] /* r1, c2 */
        r3 := c2 /* r1, r3 */
  return  /* r1,r3 */
```



Coalescing c1+r3; c2+c1r3

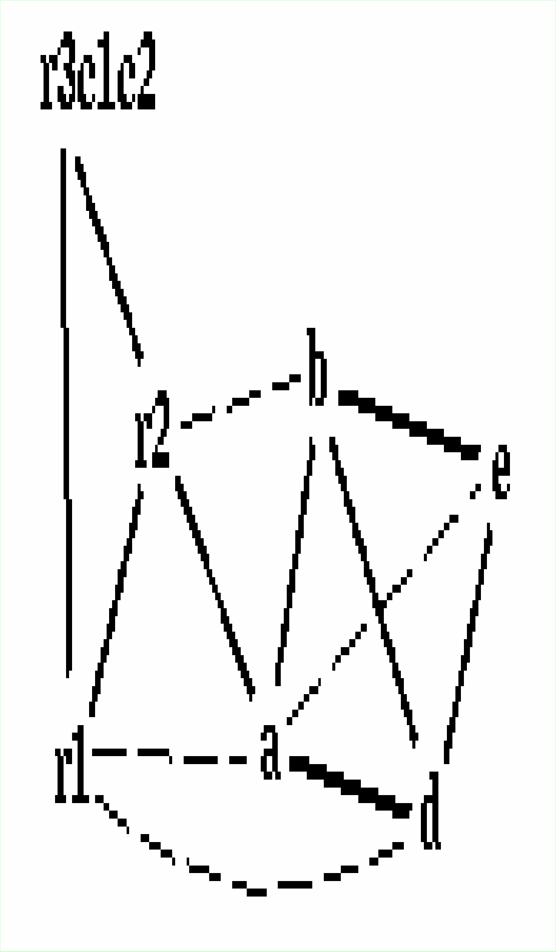


stack

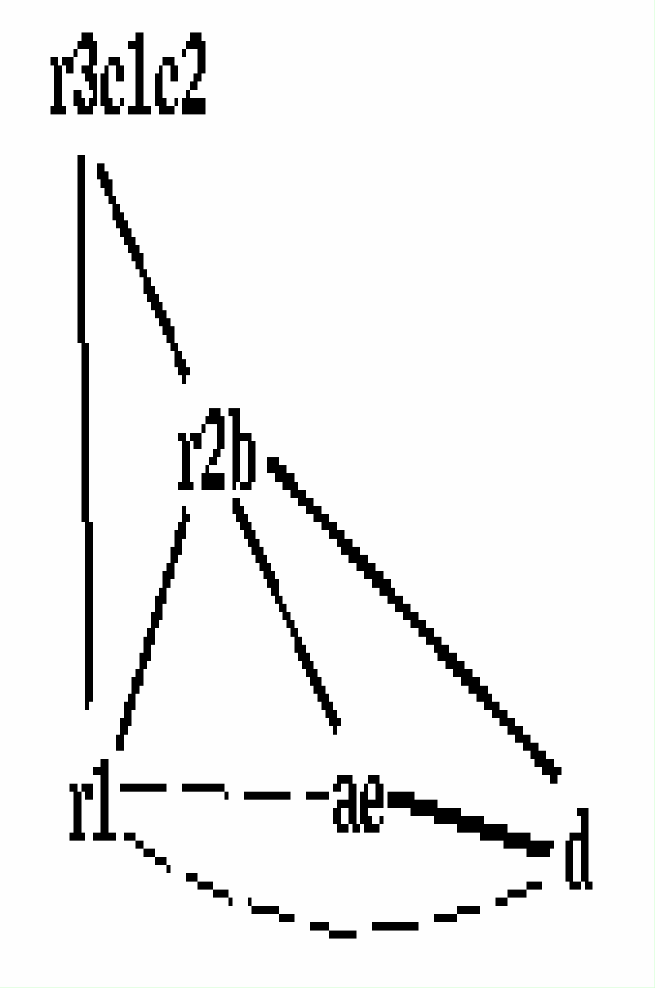


stack

Coalescing a+e; b+r2

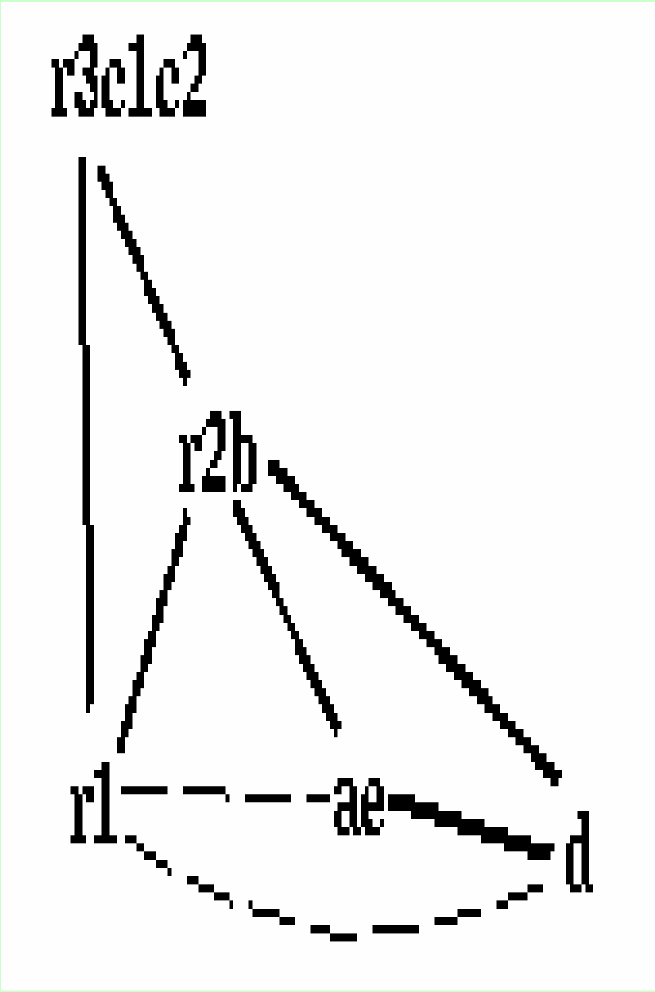


stack

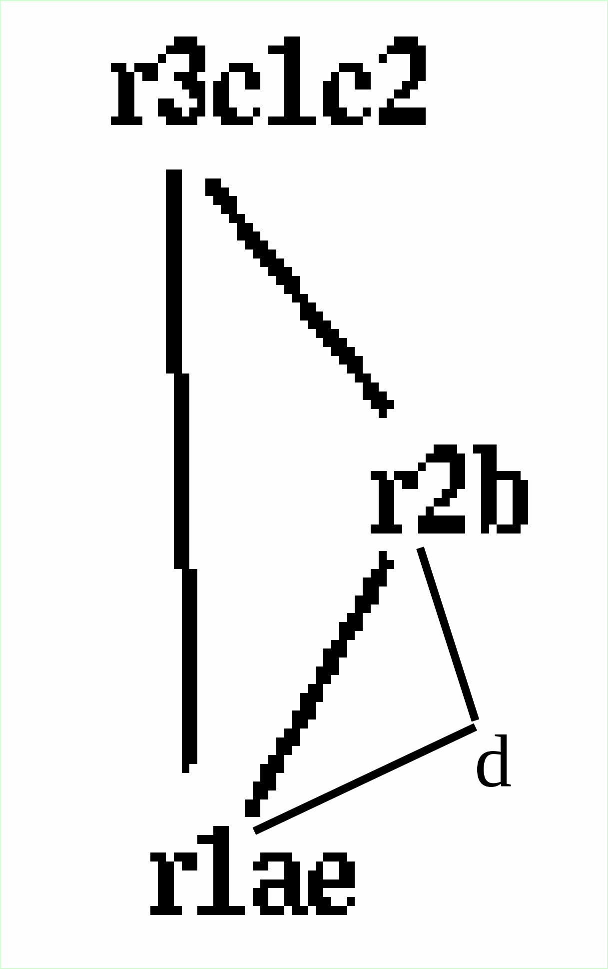


stack

Coalescing ae+r1



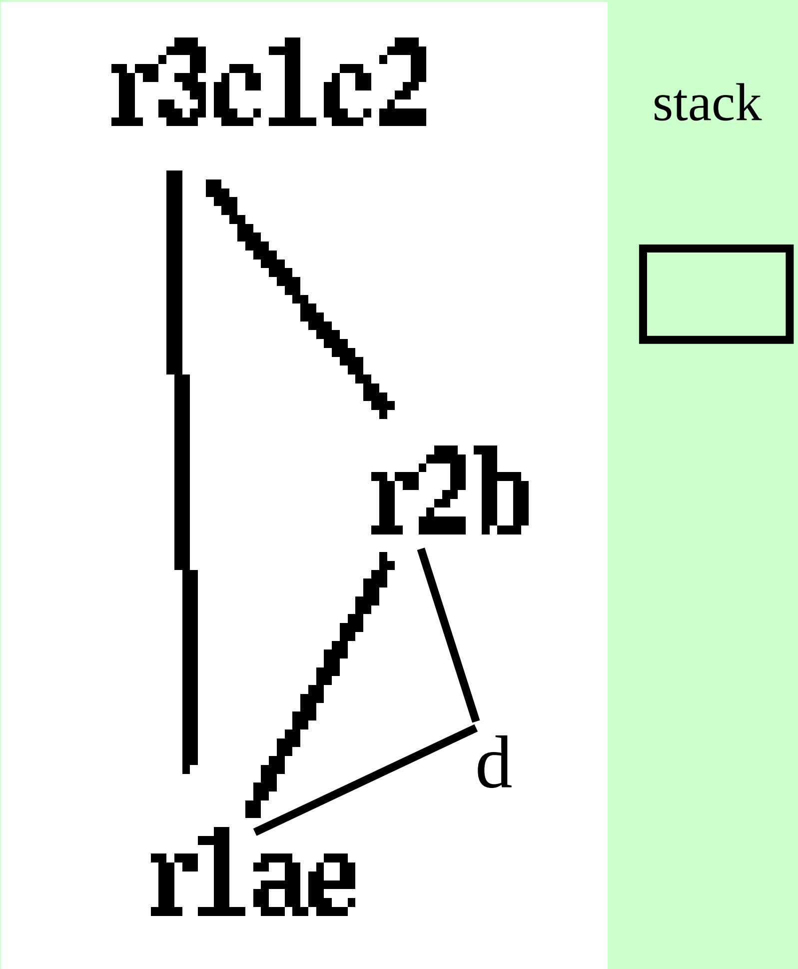
stack

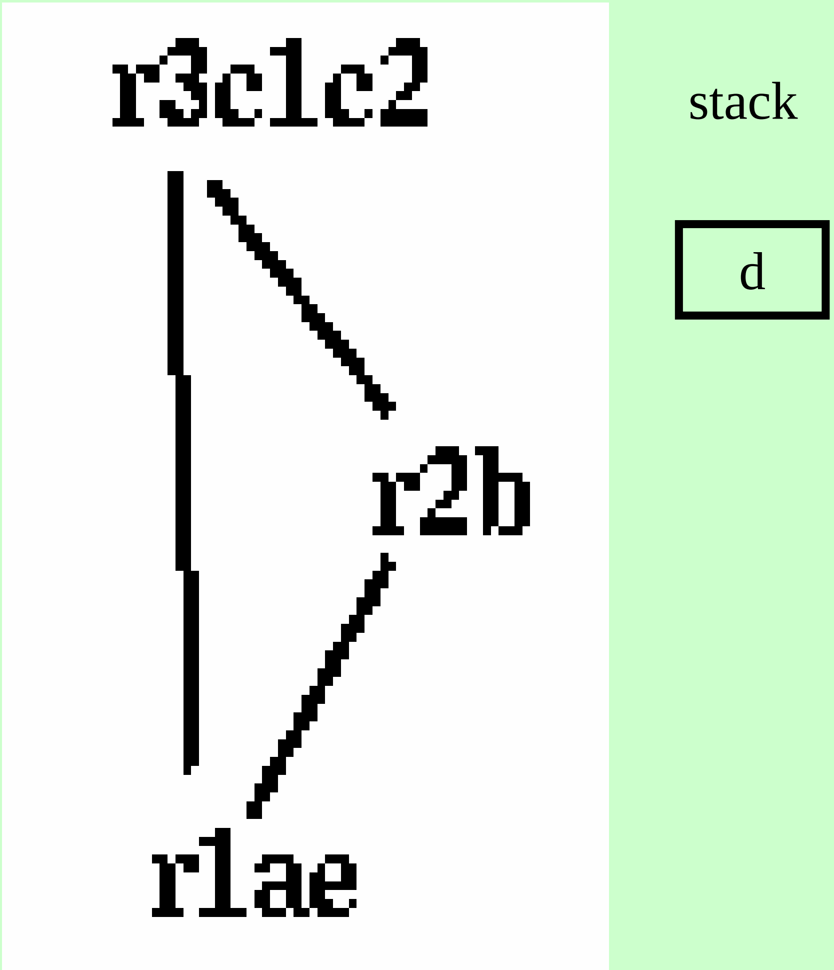


stack

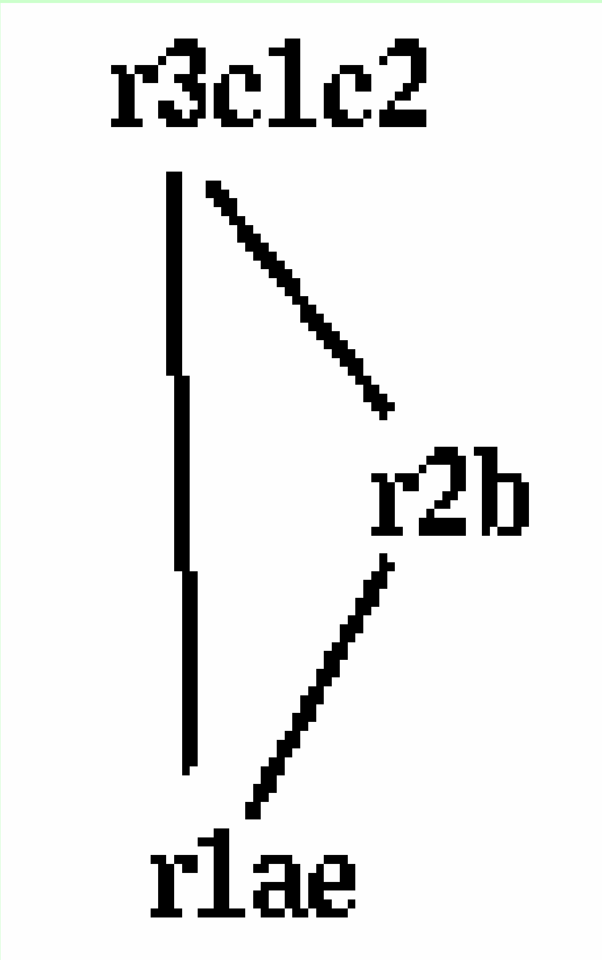
r1ae and d are constrained

Simplify d



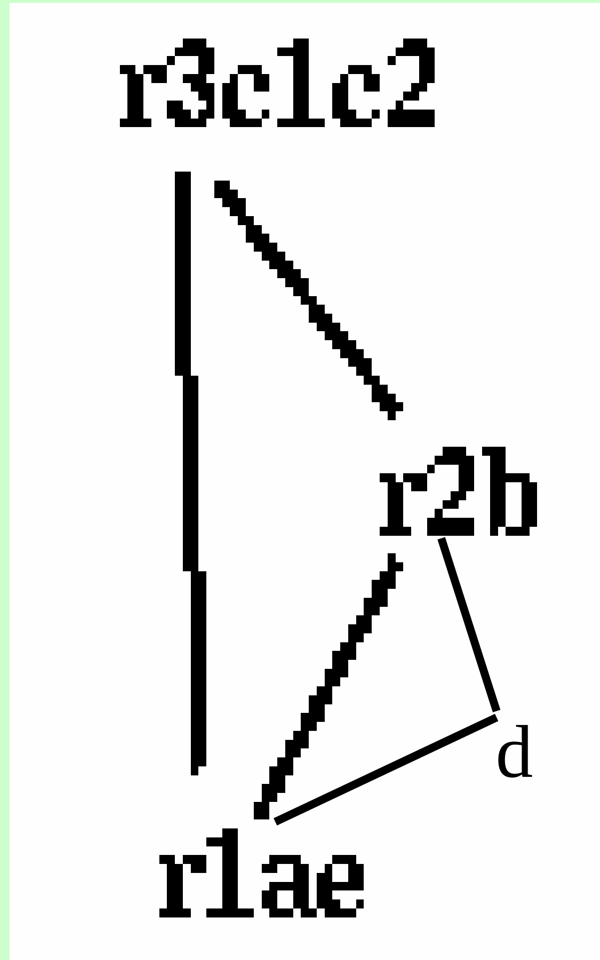


Pop d



stack

d



stack

a	r1
b	r2
c1	r3
c2	r3
d	r3
e	r1

```
enter:
        c1 := r3
        M[c_loc] := c1
        a := r1
        b := r2
        d := 0
        e := a
loop:
        d := d+b
        e := e-1
        if e>0 goto loop
        r1 := d
        c2 := M[c_loc]
        r3 := c2
  return  /* r1,r3 */
```

a	r1
b	r2
c1	r3
c2	r3
d	r3
e	r1

```
enter:
        r3 := r3
        M[c_loc] := r3
        r1 := r1
        r2 := r2
        r3 := 0
        r1 := r1
loop:
        r3 := r3+r2
        r1 := r1-1
        if r1>0 goto loop
        r1 := r3
        r3 := M[c_loc]
        r3 := r3
  return  /* r1,r3 */
```

```
enter:
        r3 := r3
        M[c_loc] := r3
        r1 := r1
        r2 := r2
        r3 := 0
        r1 := r1
loop:
        r3 := r3+r2
        r1 := r1-1
        if r1>0 goto loop
        r1 := r3
        r3 := M[c_loc]
        r3 := r3
  return  /* r1,r3 */
```

```
enter:
        M[c_loc] := r3
        r3 := 0
loop:
        r3 := r3+r2
        r1 := r1-1
        if r1>0 goto loop
        r1 := r3
        r3 := M[c_loc]
  return  /* r1,r3 */
```

```
main:   addiu  $sp,$sp, -K1
L4:  sw   $2,0+K1($sp)
     or   $25,$0,$31
     sw   $25,-4+K1($sp)
     addiu $25,$sp,0+K1
     or   $2,$0,$25
     addi $25,$0,10
     or   $4,$0,$25
     jal  nfactor
     lw   $25,-4+K1
     or   $31,$0,$25
     b    L3
L3: addiu $sp,$sp,K1
     j    $31

nfactor:  addiu   $sp,$sp,-K2
L6:   sw  $2,0+K2($sp)
      or  $25,$0,$4
      or  $24,$0,$31
      sw  $24,-4+K2($sp)
      sw  $30,-8+K2($sp)
      beq $25,$0,L0
L1:   or  $30,$0,$25
      lw  $24,0+K2
      or  $2,$0,$24
      addi $25,$25,-1
      or  $4,$0,$25
      jal nfactor

      or $25,$0,$2
      mult $30,$25
      mflo $30
L2:   or  $2,$0,$30
      lw  $30,-4+K2($sp)
      or  $31,$0,$30
      lw  $30,-8+K2($sp)
      b   L5
L0:   addi $30,$0,1
      b   L2
L5:   addiu  $sp,$sp,K2
      j   $31
```

Interprocedural Allocation

- Allocate registers to multiple procedures
- Potential saving
  - caller/callee save registers
  - Parameter passing
  - Return values
- But may increase compilation cost
- Function inline can help

Summary

- Two Register Allocation Methods
  - Local of every IR tree
    - Simultaneous instruction selection and register allocation
    - Optimal (under certain conditions)
  - Global of every function
    - Applied after instruction selection
    - Performs well for machines with many registers
    - Can handle instruction level parallelism
- Missing
  - Interprocedural allocation