

Compiler Course Summary

Mooly Sagiv

<http://www.cs.tau.ac.il/~msagiv/courses/wcc12-13.html>

Outlines

- Demonstrate compiler phase on a simple example
- Other topics

A Running Example

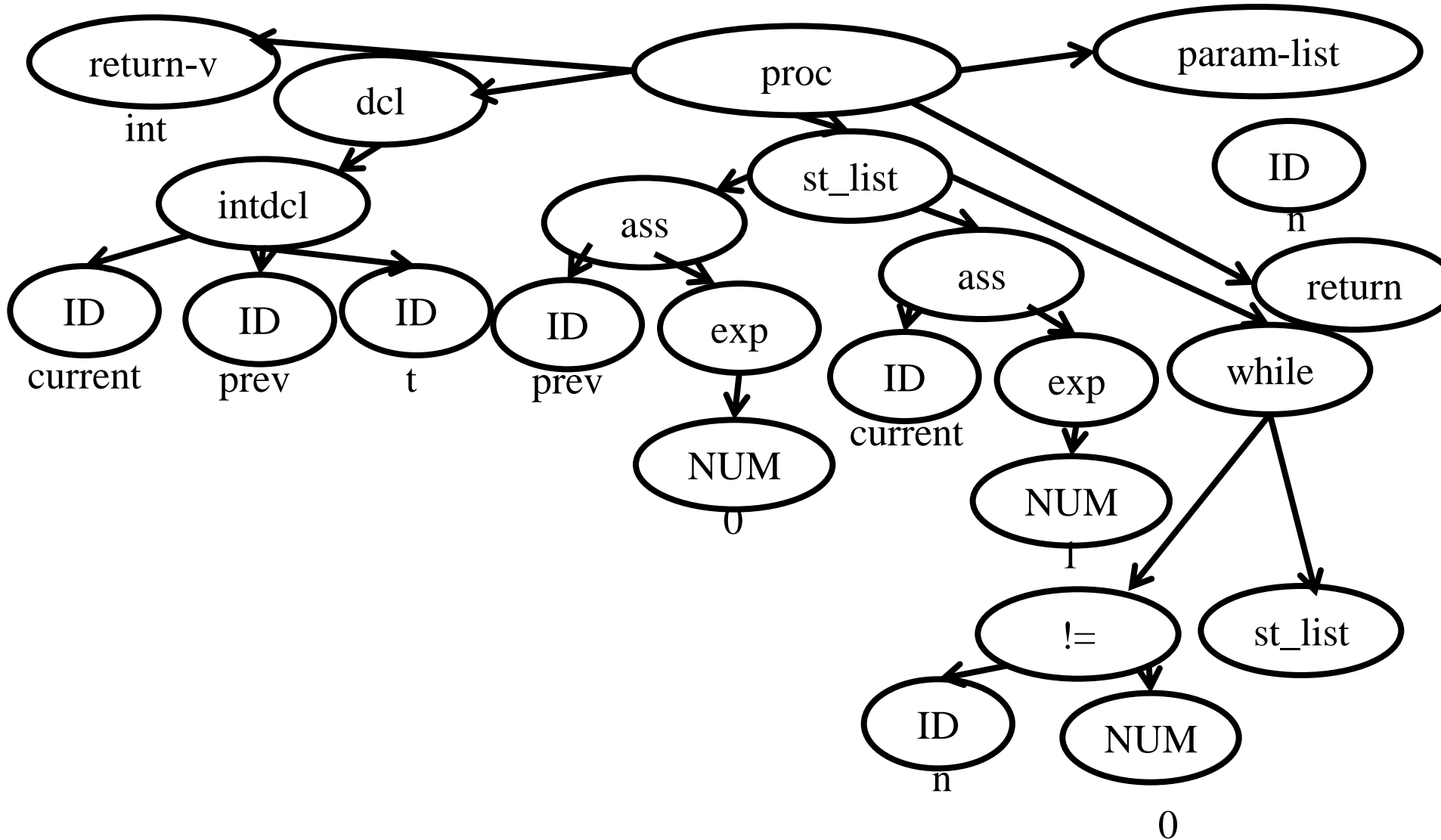
```
int fib(n) {  
    int current, prev, t;  
    prev = 0;  
    current = 1;  
    while (n != 0) {  
        n = n - 1 ;  
        t = prev + current ;  
        prev = current ;  
        current = t;  
    }  
    return current;  
}
```

Lexical Analysis

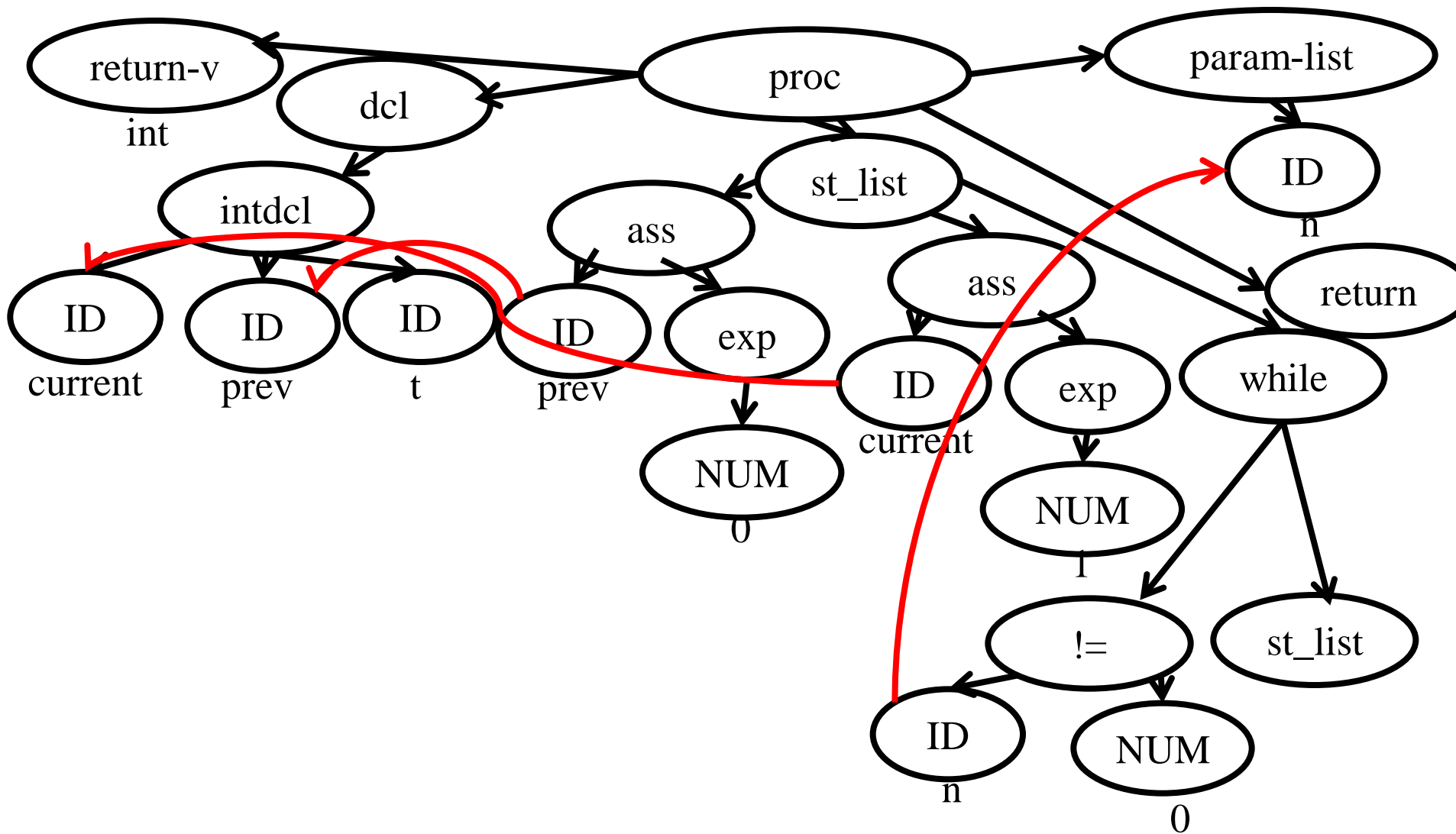
```
int fib(n) {  
    int current, prev, t;  
    prev = 0;  
    current = 1;  
    while (n != 0) {  
        n = n - 1 ;  
        t = prev + current ;  
        prev = current ;  
        current = t;  
    }  
    return current;  
}
```

Token	class	lexme
int	keyword	
fib	identifier	“fib”
((
n	identifier	“n”
))	
{	{	
int	keyword	
current	identifier	“current”
,	,	
prev	identifier	“prev”
,	,	
t	identier	“t”

Syntax Analysis



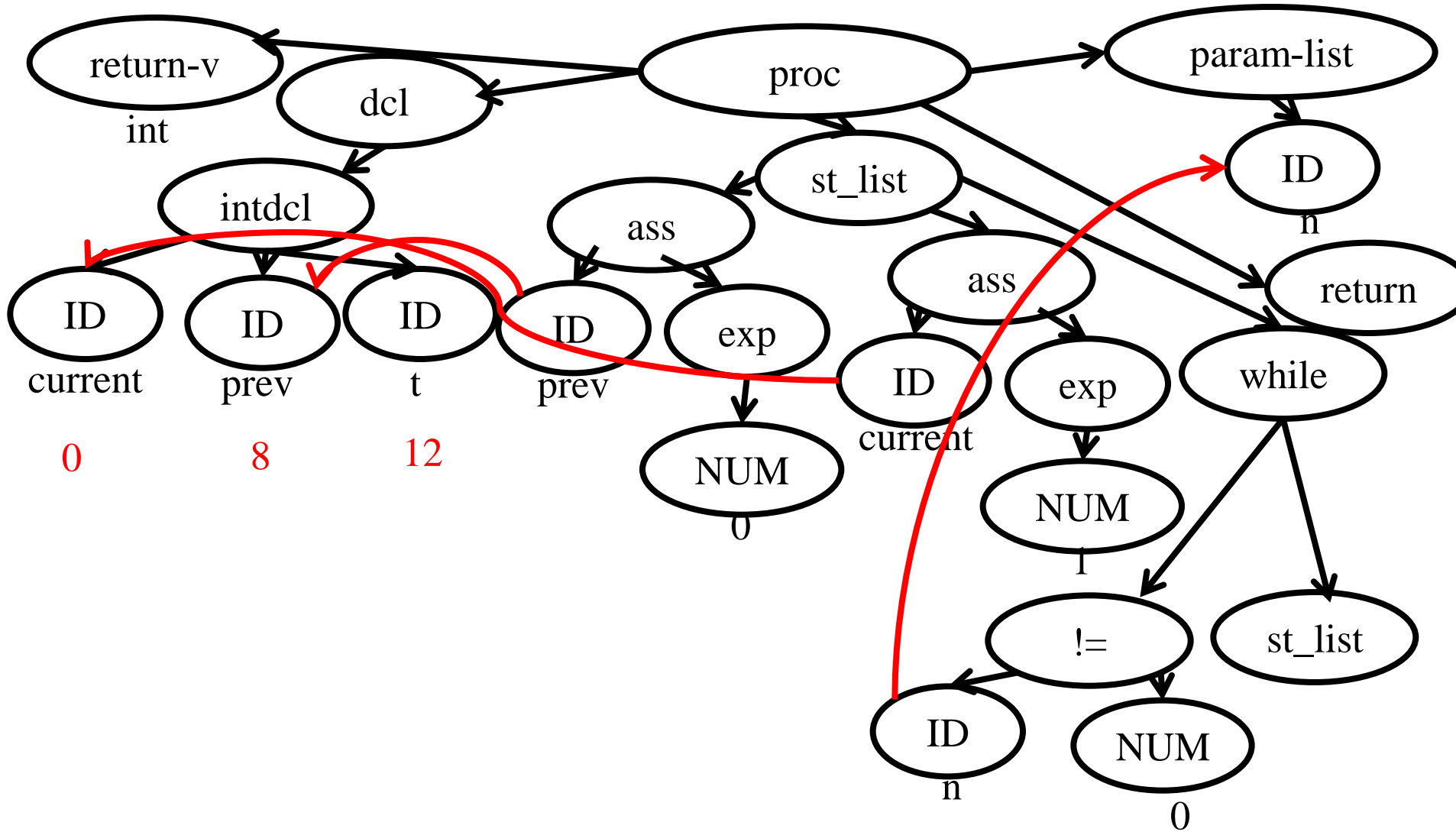
Context Analysis



Recursive Interpreter

```
interpret_st(node) {  
    switch(node)  
    case ASSIGN:  
        store(node.lhs, interpret_exp(node.rhs));  
    case WHILE:  
        if (interpret_exp(node.cond))  
            interpret_st(node.st)  
        ....
```

Context Analysis(2)



Simple generated code

```
int fib(n) {
    int current, prev, t;
    prev = 0;
    current = 1;
    while (n != 0) {
        n = n - 1 ;
        t = prev + current ;
        prev = current ;
        current = t;
    }
    return current;
}

fib: addiu $sp,$sp,-12
L1: sw $0, -4(fp) // prev = 0 ;
    addiu $5, $0, 1 ; sw $5, -0(fp) //current = 1
L2: beq $2, $0, L3
    addiu $2, $5, -1 // n = n -1 ;
    lw -4(fp), $5; lw -0(fp), $6;
    add $5, $6, $5; sw $5, -8(fp) // t= prev + current
    lw -0(fp), $5; sw $5, -4(fp) // prev = current
    lw -8(fp), $5, sw $5, -0(fp) // current = t
    b L2
L3: lw -0(fp), $4 // return current ;
    addiu, $sp, $sp, 12
```

Two Phase Solution

Dynamic Programming

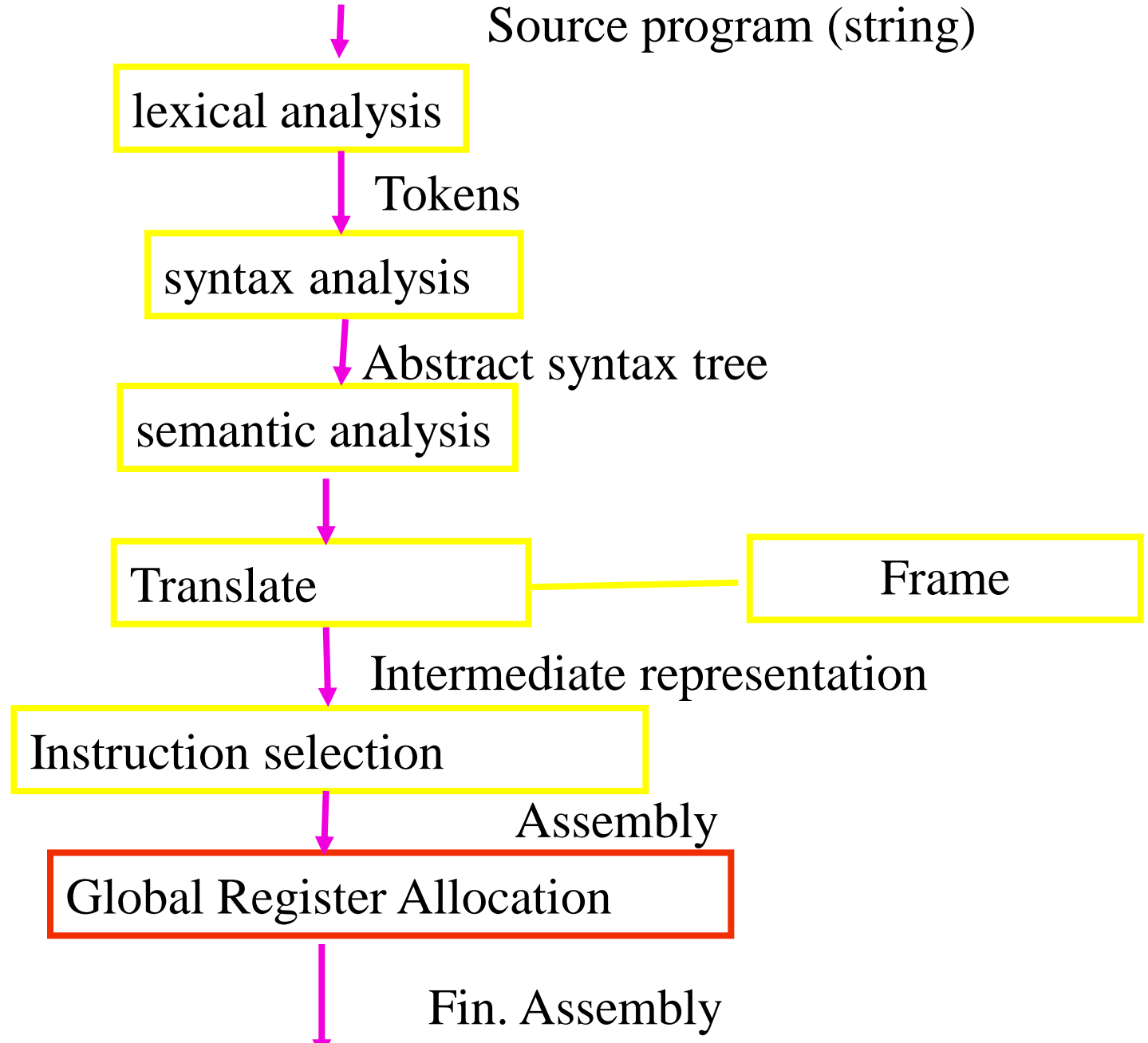
Sethi & Ullman

- 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

A Simple Example

```
int fib(n) {
    int current, prev, t;
    prev = 0;
    current = 1;
    while (n != 0) {
        n = n - 1 ;
        t = prev + current ;
        prev = current ;
        current = t;
    }
    return current;
}

fib:   addiu $sp,$sp,-K2
L1:    or r_n, $0, $2 // save n
        or  r_prev, $0, $0 // prev = 0 ;
        addiu r_current, $0, 1 // r_current = 1
L2:    beq r_n, $0, L3
        addiu r_n, r_n, -1 // n = n - 1 ;
        add  r_t, r_prev, r_current; // t= prev + current
        or r_prev, $0, r_current // prev = current
        or r_current, $0, r_t // current = t
        b L2
L3:    or $4, $0, r_current; // return current ;
        addiu, $sp, $sp, K2
```

Liveness Analysis

$\$0 \2

```
or r_n, $0, $2 // def {r_n} use {$2}
```

$\$0 r_n$

```
or r_prev, $0, $0 // def {r_prev} use {$0}
```

$\$0 r_prev r_n$

```
addiu r_current, $0, 1 // def {r_current} use {$0}
```

$\$0 r_current r_prev r_n$

```
L2: beq r_n, $0, L3 // def {} use {r_n, $0}
```

$\$0, r_current r_prev r_n$

```
addiu r_n, r_n, -1 // def {r_n} use {r_n}
```

$\$0 r_current r_prev$

r_n

```
addu r_t, r_prev, r_current // def {r_t} use {r_prev, r_current}
```

$\$0 r_current r_t$

r_n

```
or r_prev, $0, r_current // def {r_prev} use {$0, r_current}
```

$\$0 r_t$

$r_prev r_n$

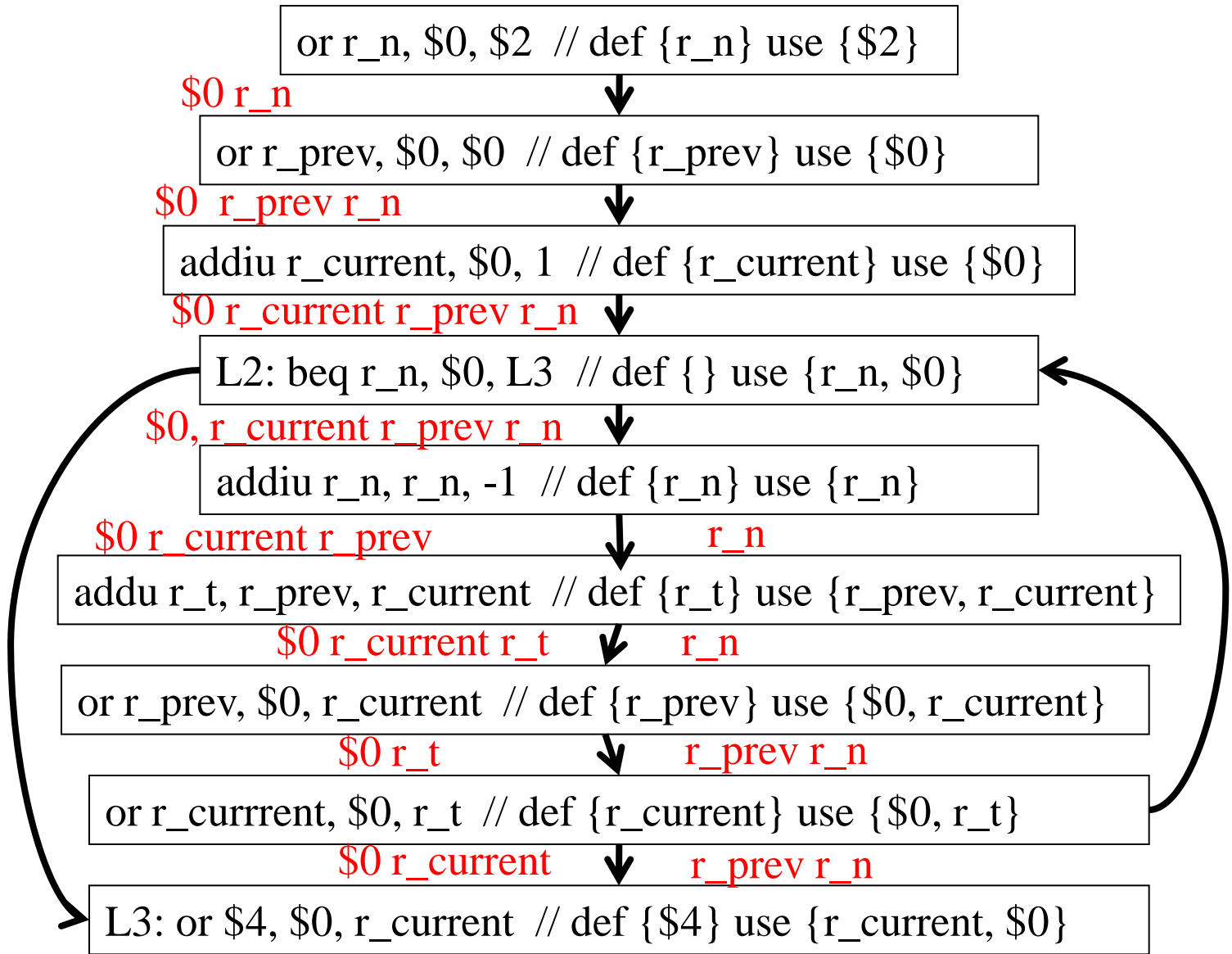
```
or r_current, $0, r_t // def {r_current} use {$0, r_t}
```

$\$0 r_current$

$r_prev r_n$

```
L3: or $4, $0, r_current // def {$4} use {r_current, $0}
```

$\$0 \4



Interference Graph Construction

$\$0 \2

or r_n, \$0, \$2 // def {r_n} use {\$2}

$\$0 r_n$

or r_prev, \$0, \$0 // def {r_prev} use {\$0}

$\$0 r_n r_prev$

addiu r_current, \$0, 1 // def {r_current} use {\$0}

$\$0 r_current r_prev r_n$

L2: beq r_n, \$0, L3 // def {} use {r_n, \$0}

$\$0, r_current r_prev r_n$

addiu r_n, r_n, -1 // def {r_n} use {r_n}

$\$0 r_current r_prev r_n$

addu r_t, r_prev, r_current // def {r_t} use {r_prev, r_current}

$\$0 r_current r_t$

r_n

or r_prev, \$0, r_current // def {r_prev} use {\$0, r_current}

$\$0 r_t$

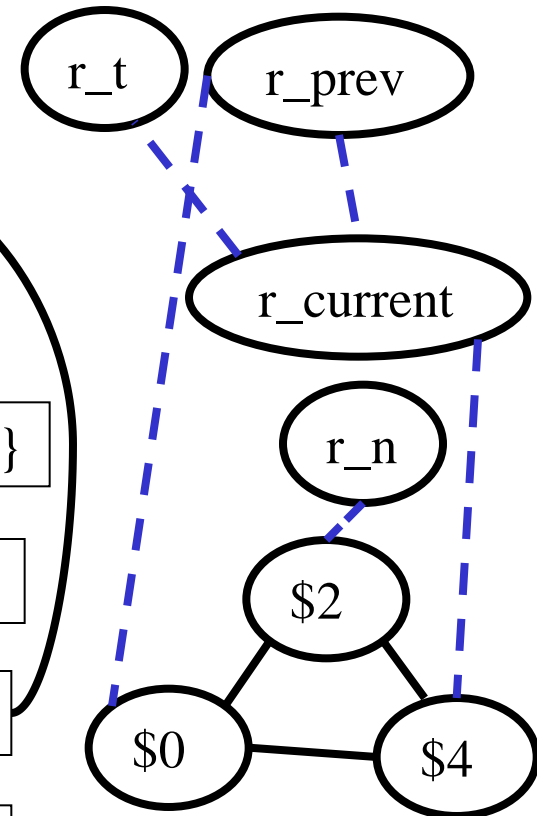
$r_prev r_n$

or r_current, \$0, r_t // def {r_current} use {\$0, r_t}

$\$0 r_current r_prev r_n$

L3: or \$4, \$0, r_current // def {\$4} use {r_current, \$0}

$\$0 \4



Interference Graph Construction(2)

$\$0$ $\$2$

or r_n, \$0, \$2 // def {r_n} use {\$2}

$\$0$ r_n

or r_prev, \$0, \$0 // def {r_prev} use {\$0}

$\$0$ r_n r_prev

addiu r_current, \$0, 1 // def {r_current} use {\$0}

$\$0$ r_current r_prev r_n

L2: beq r_n, \$0, L3 // def {} use {r_n, \$0}

$\$0$, r_current r_prev r_n

addiu r_n, r_n, -1 // def {r_n} use {r_n}

$\$0$ r_current r_prev r_n

addu r_t, r_prev, r_current // def {r_t} use {r_prev, r_current}

$\$0$ r_current r_t

r_n

or r_prev, \$0, r_current // def {r_prev} use {\$0, r_current}

$\$0$ r_t

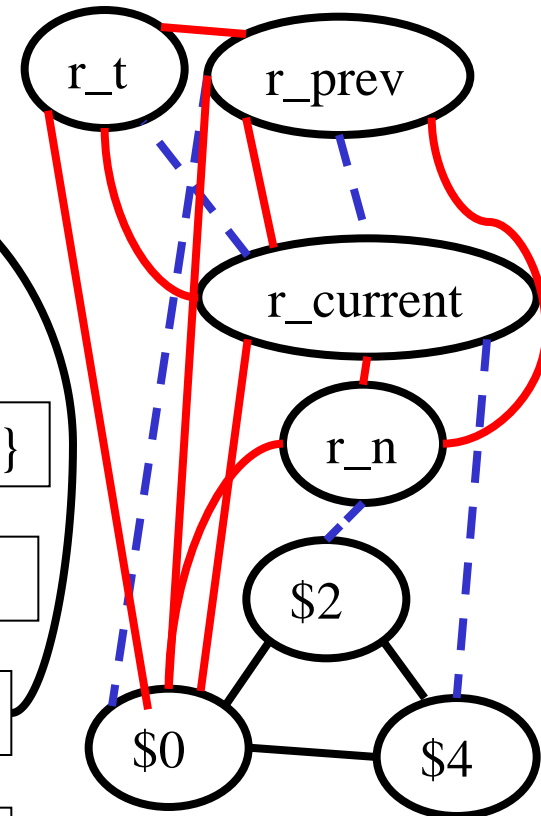
r_prev r_n

or r_current, \$0, r_t // def {r_current} use {\$0, r_t}

$\$0$ r_current r_prev r_n

L3: or \$4, \$0, r_current // def {\$4} use {r_current, \$0}

$\$0$ \$4



Coloring

$\$0 \2

```
or r_n, $0, $2 // def {r_n} use {$2}
```

$\$0 r_n$

```
or r_prev, $0, $0 // def {r_prev} use {$0}
```

$\$0 r_n r_prev$

```
addiu r_current, $0, 1 // def {r_current} use {$0}
```

$\$0 r_current r_prev r_n$

```
L2: beq r_n, $0, L3 // def {} use {r_n, $0}
```

$\$0, r_current r_prev r_n$

```
addiu r_n, r_n, -1 // def {r_n} use {r_n}
```

$\$0 r_current r_prev r_n$

```
addu r_t, r_prev, r_current // def {r_t} use {r_prev, r_current}
```

$\$0 r_current r_t$

r_n

```
or r_prev, $0, r_current // def {r_prev} use {$0, r_current}
```

$\$0 r_t$

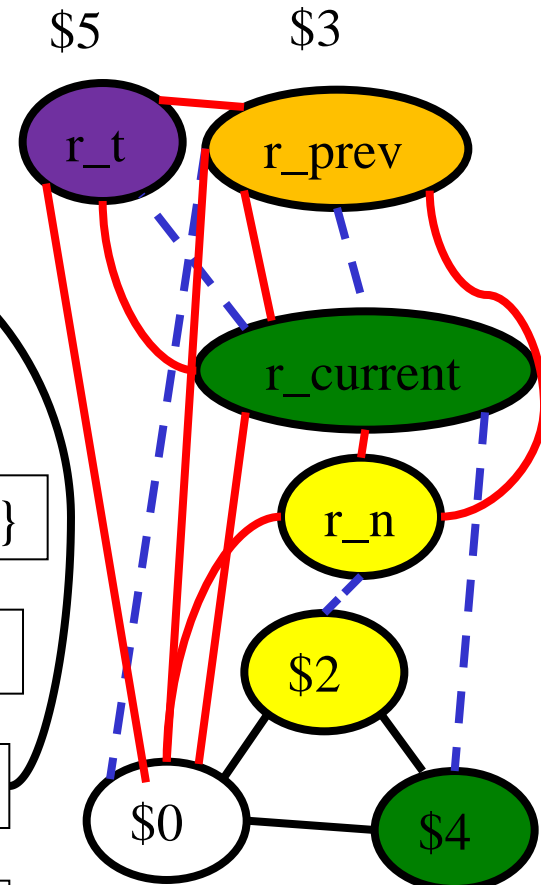
$r_prev r_n$

```
or r_current, $0, r_t // def {r_current} use {$0, r_t}
```

$\$0 r_current r_prev r_n$

```
L3: or $4, $0, r_current // def {$4} use {r_current, $0}
```

$\$0 \4



The generated code

```
int fib(n) {  
    int current, prev, t;  
    prev = 0;  
    current = 1;  
    while (n != 0) {  
        n = n - 1 ;  
        t = prev + current ;  
        prev = current ;  
        current = t;  
    }  
    return current;  
}
```

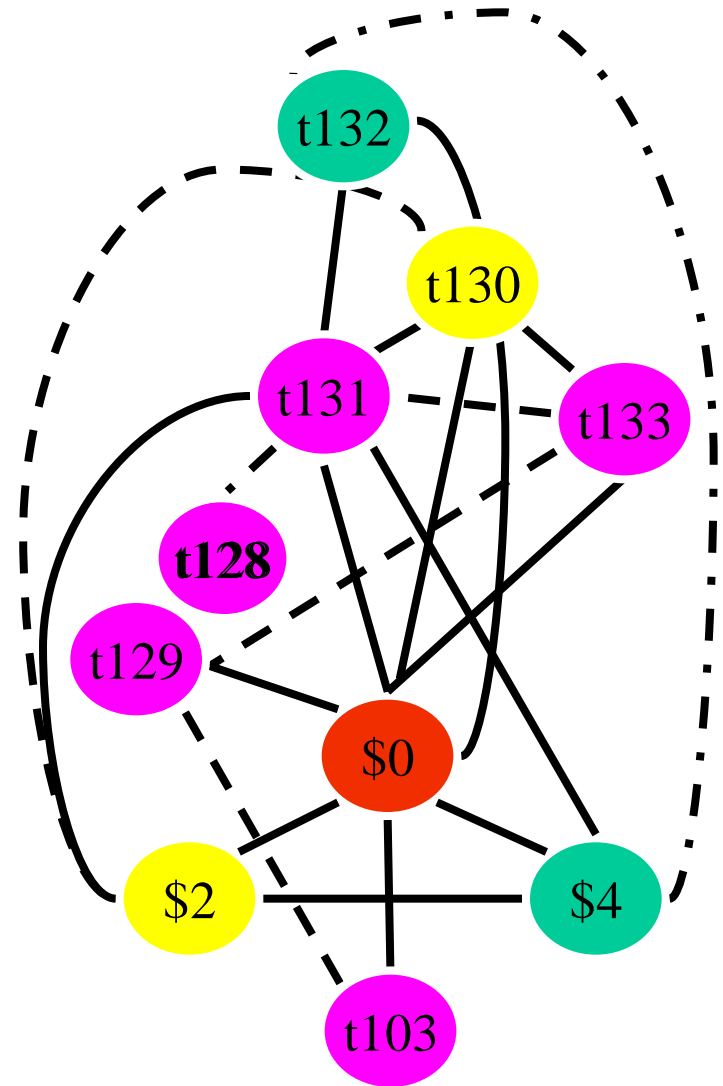
```
fib:   addiu $sp,$sp,-K2  
L1:   //or $2, $0, $2 // save n  
      or   $3, $0, $0 // prev = 0 ;  
      addiu $4, $0, 1 //current = 1  
L2:   beq  $2, $0, L3  
      addiu $2, $2, -1 // n = n -1 ;  
      add  $5, $3, $4; // t= prev + current  
      or  $3, $0, $4 // prev = current  
      or  $4, $0, $5 // current = t  
      b   L2  
L3:   // or $4, $0, $4 // return current ;  
      addiu $sp, $sp, K2
```

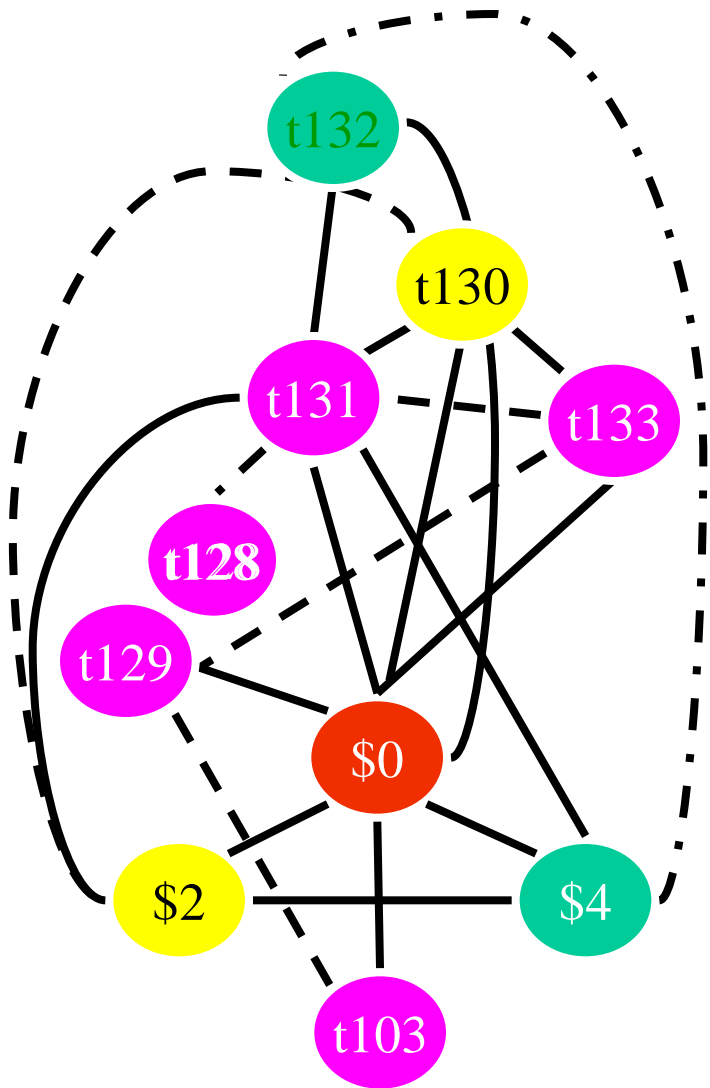


```

13:  beq t128, $0, 10 /* $0, t128 */
11:  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 */
12:  or t103, $0, t129 /* $0, t103 */
    b lend /* $0, t103 */
10:  addi t129, $0, 1 /* $0, t129 */
    b 12 /* $0, t129 */

```





```

13:    beq t128 $0 10
11:    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
12:    or t103 $0 t129
        b lend
10:    addi t129 $0, 1
        b 12

```

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]

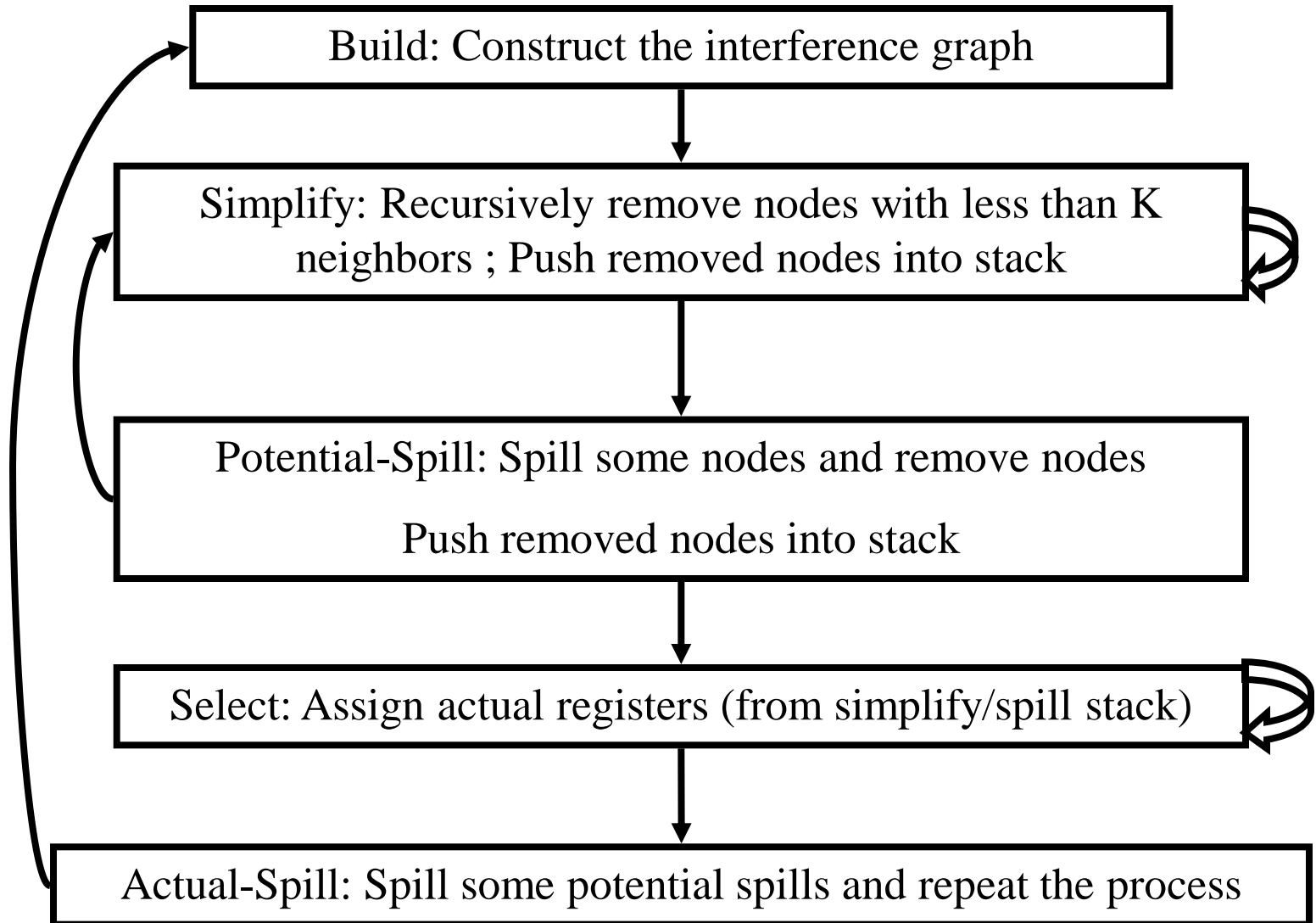
- Assume:
 - An undirected graph $G(V, E)$
 - A node $v \in V$ with less than K neighbors
 - $G - \{v\}$ is K colorable
- Then, G is K colorable

Coloring by Simplification

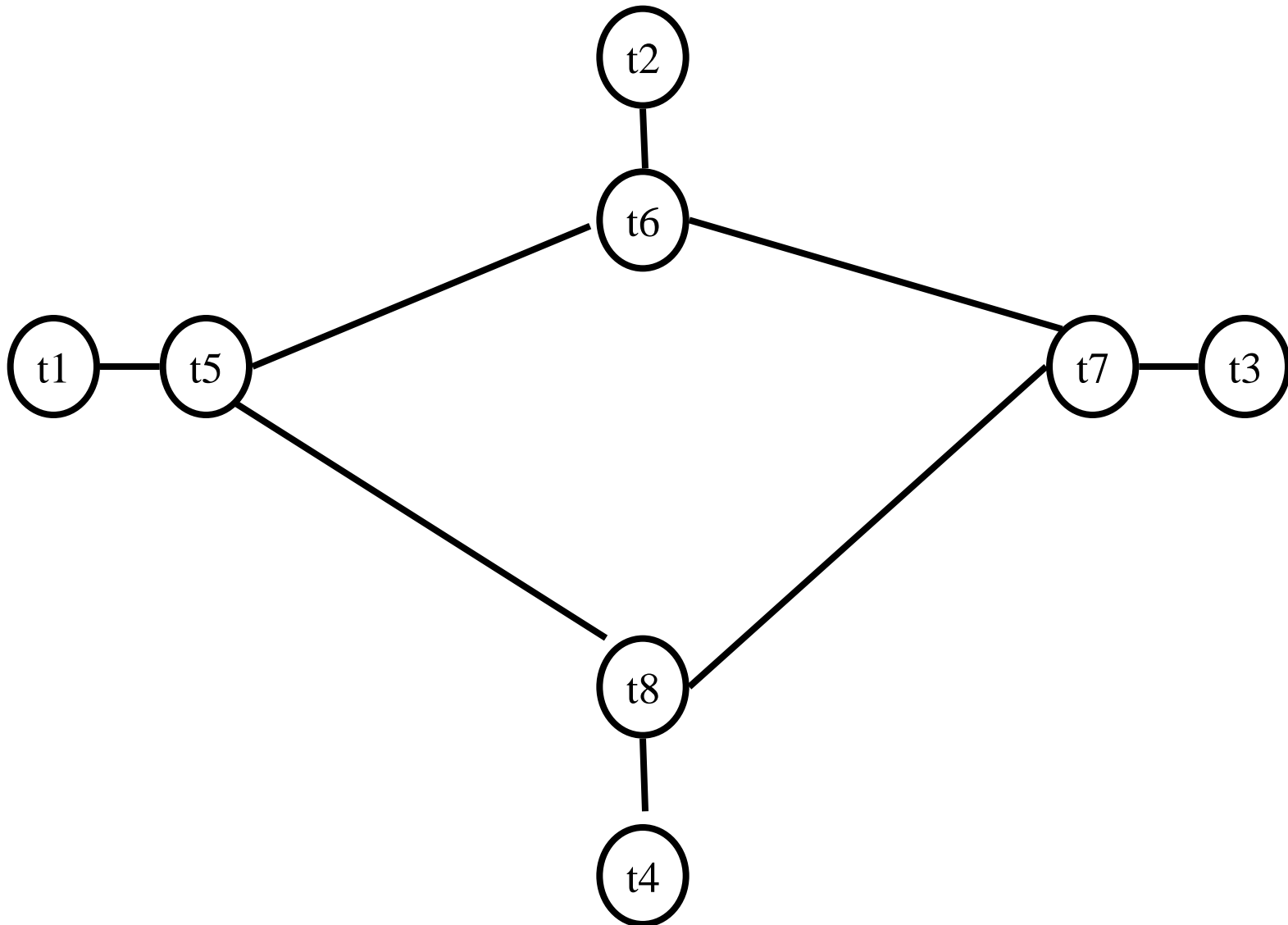
[Kempe 1879]

- 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

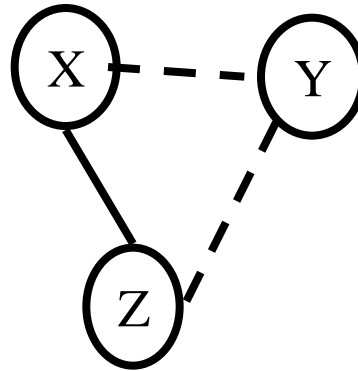
- MOVs can be removed if the source and the target share the same register
- The source and the target of the move can be merged into a single node (unifying the sets of neighbors)
- May require more registers
- **Conservative Coalescing**
 - Merge nodes only if the resulting node has fewer than K neighbors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

Constrained Moves

- A instruction $T \leftarrow S$ is **constrained**
 - if S and T interfere
- May happen after coalescing

$X \leftarrow Y$ */* X, Y, Z */*

$Y \leftarrow Z$



- Constrained MOVs are not coalesced

Constrained Moves

\$0 \$2

```
or r_n, $0, $2 // def {r_n} use {$2}
```

\$0 r_n

```
or r_prev, $0, $0 // def {r_prev} use {$0}
```

\$0 r_n r_prev

```
addiu r_current, $0, 1 // def {r_current} use {$0}
```

\$0 r_current r_prev r_n

```
L2: beq r_n, $0, L3 // def {} use {r_n, $0}
```

\$0, r_current r_prev r_n

```
addiu r_n, r_n, -1 // def {r_n} use {r_n}
```

\$0 r_current r_prev r_n

```
addu r_t, r_prev, r_current // def {r_t} use {r_prev, r_current}
```

\$0 r_current r_t

r_n

```
or r_prev, $0, r_current // def {r_prev} use {$0, r_current}
```

\$0 r_t

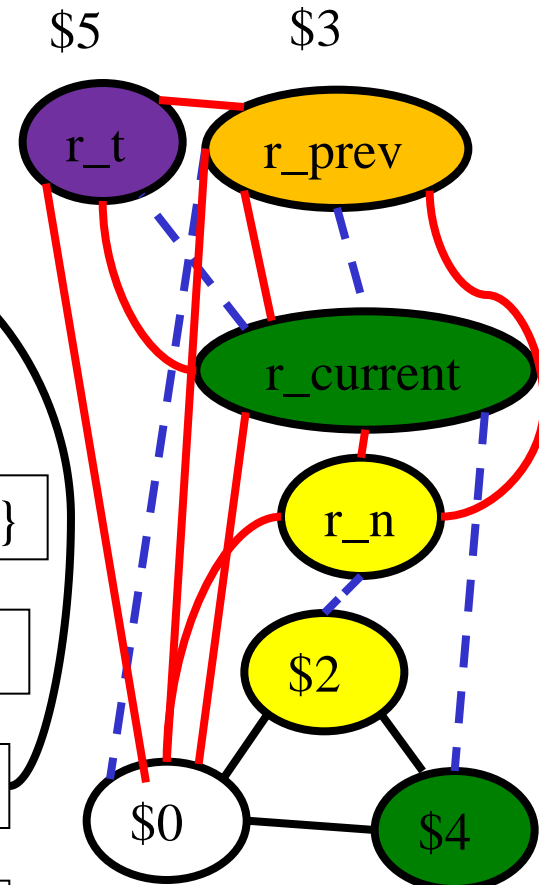
r_prev r_n

```
or r_current, $0, r_t // def {r_current} use {$0, r_t}
```

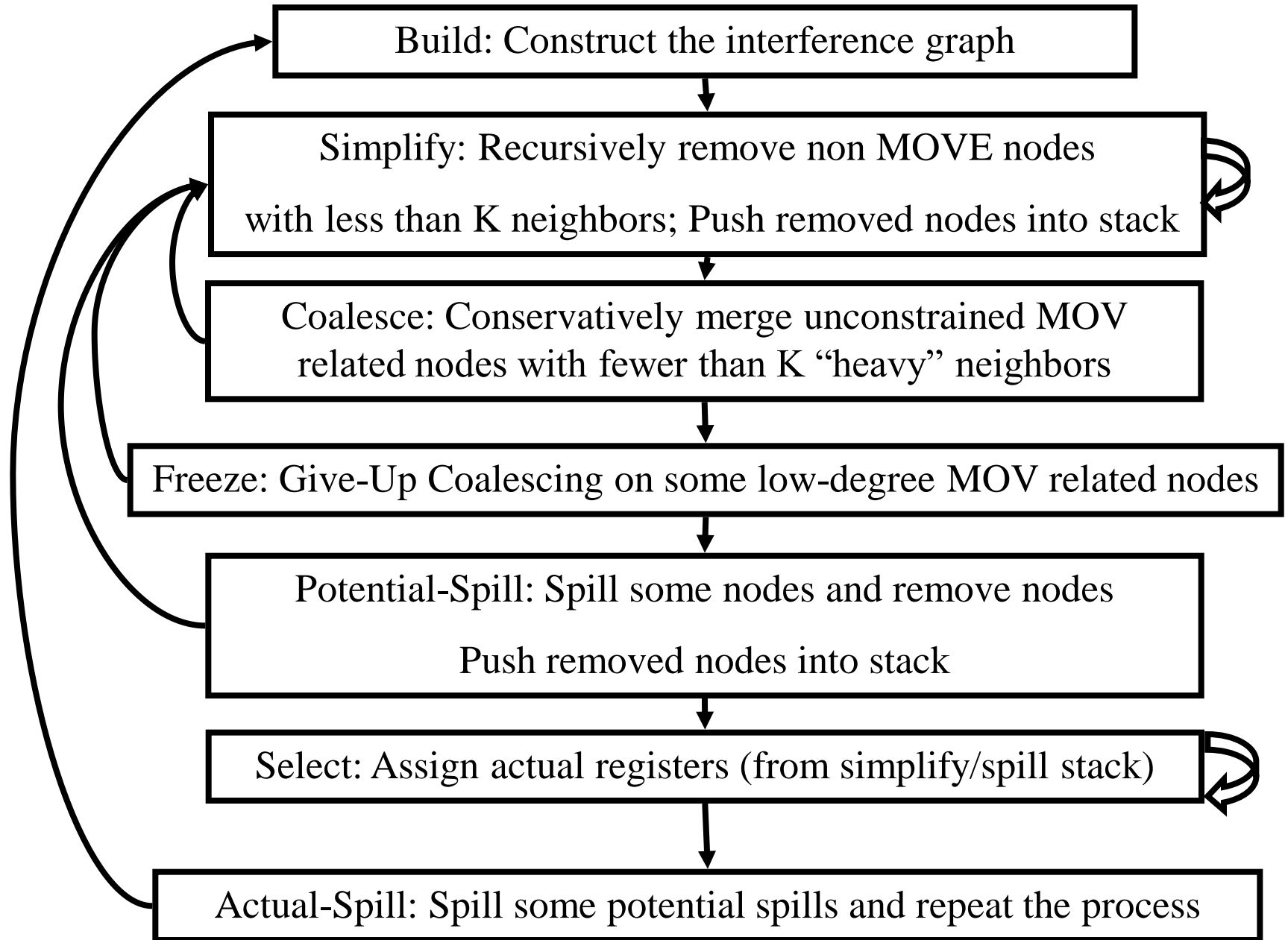
\$0 r_current r_prev r_n

```
L3: or $4, $0, r_current // def {$4} use {r_current, $0}
```

\$0 \$4



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

Coloring

\$0 \$2

```
or r_n, $0, $2 // def {r_n} use {$2}
```

\$0 r_n

```
or r_prev, $0, $0 // def {r_prev} use {$0}
```

\$0 r_n r_prev

```
addiu r_current, $0, 1 // def {r_current} use {$0}
```

\$0 r_current r_prev r_n

```
L2: beq r_n, $0, L3 // def {} use {r_n, $0}
```

\$0, r_current r_prev r_n

```
addiu r_n, r_n, -1 // def {r_n} use {r_n}
```

\$0 r_current r_prev r_n

```
addu r_t, r_prev, r_current // def {r_t} use {r_prev, r_current}
```

\$0 r_current r_t

r_n

```
or r_prev, $0, r_current // def {r_prev} use {$0, r_current}
```

\$0 r_t

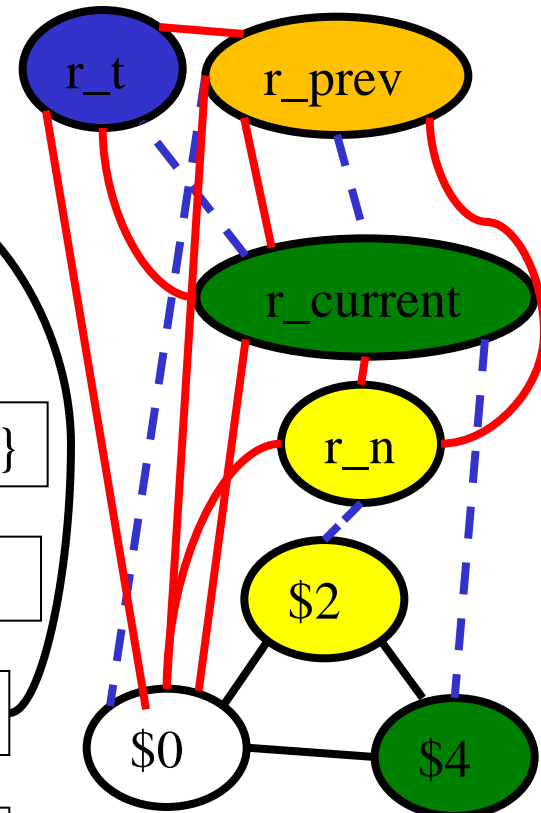
r_prev r_n

```
or r_current, $0, r_t // def {r_current} use {$0, r_t}
```

\$0 r_current r_prev r_n

```
L3: or $4, $0, r_current // def {$4} use {r_current, $0}
```

\$0 \$4



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

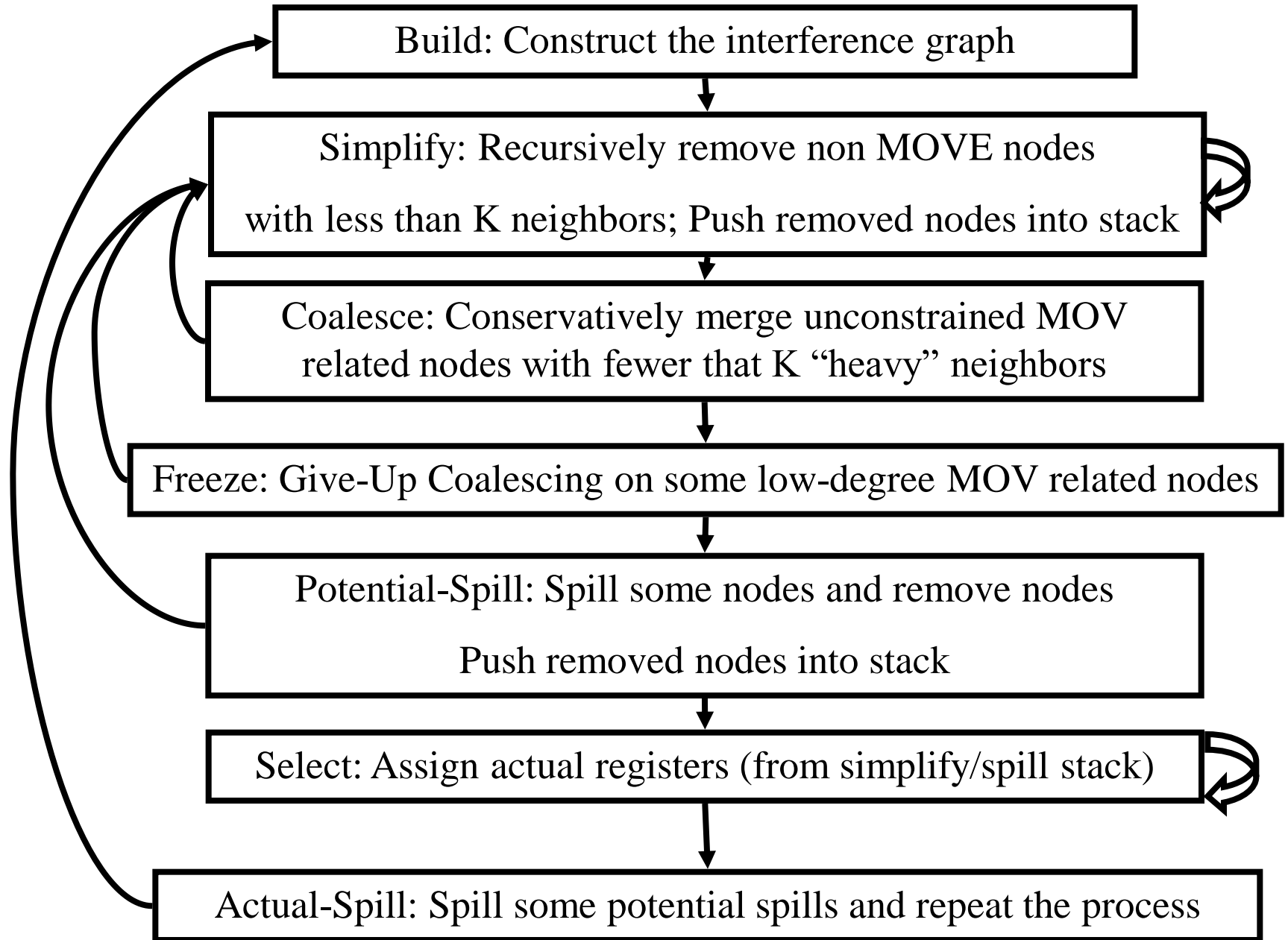
if e>0 goto loop

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

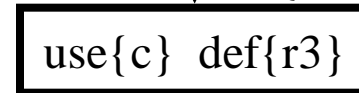
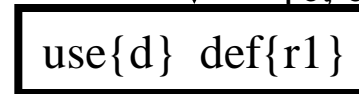
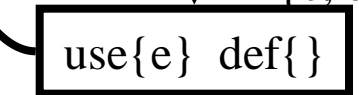
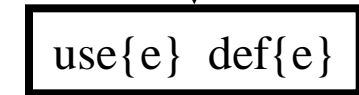
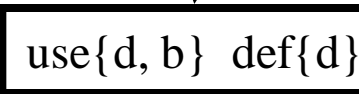
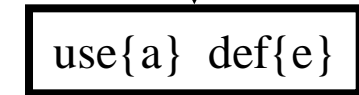
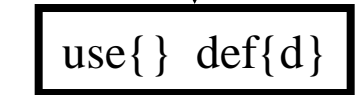
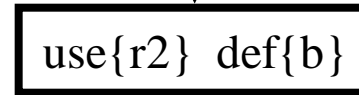
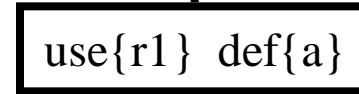
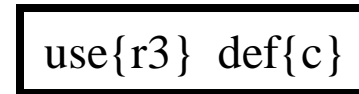
e := e-1

if e>0 goto loop

r1 := d

r3 := c

return /* r1,r3 */



{c, d, e}

{c, d, e}

{c, d}

{r1, c}

{r1, r3}



A Complete Example

enter:

c := r3

a := r1

b := r2

d := 0

e := a

loop:

d := d+b

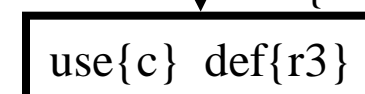
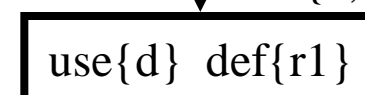
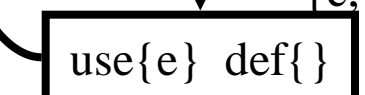
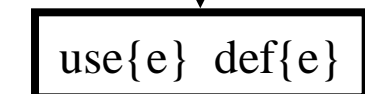
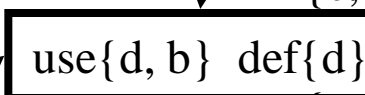
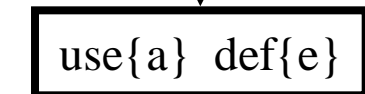
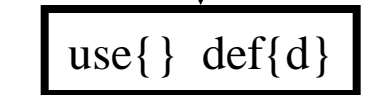
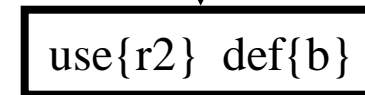
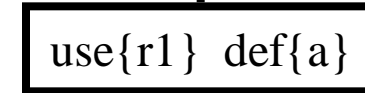
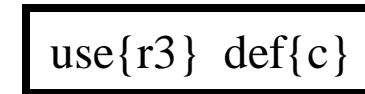
e := e-1

if e>0 goto loop

r1 := d

r3 := c

return /* r1,r3 */



{c, d, e, b}

{c, d, e}

{c, d, e}

{c, d, e, b}

{r1, c}

{r1, r3}

A Complete Example

enter:

c := r3

a := r1

b := r2

d := 0

e := a

loop:

d := d+b

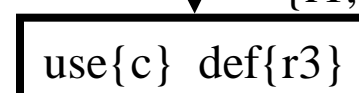
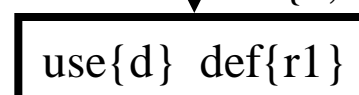
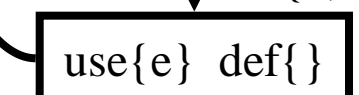
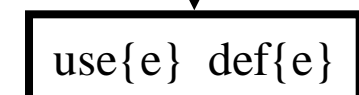
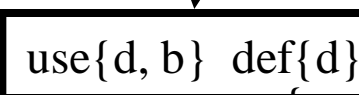
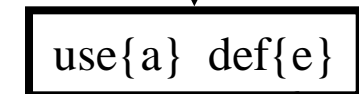
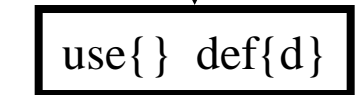
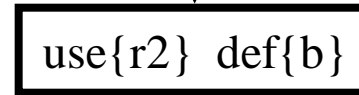
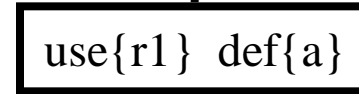
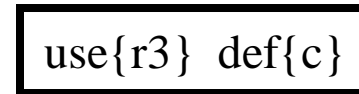
e := e-1

if e>0 goto loop

r1 := d

r3 := c

return /* r1,r3 */



{c, d, e, b}

{c, d, e}

{c, d, e, b}

{c, d, e, b}

{r1, c}

{r1, r3}

A Complete Example

enter:

c := r3

a := r1

b := r2

d := 0

e := a

loop:

d := d+b

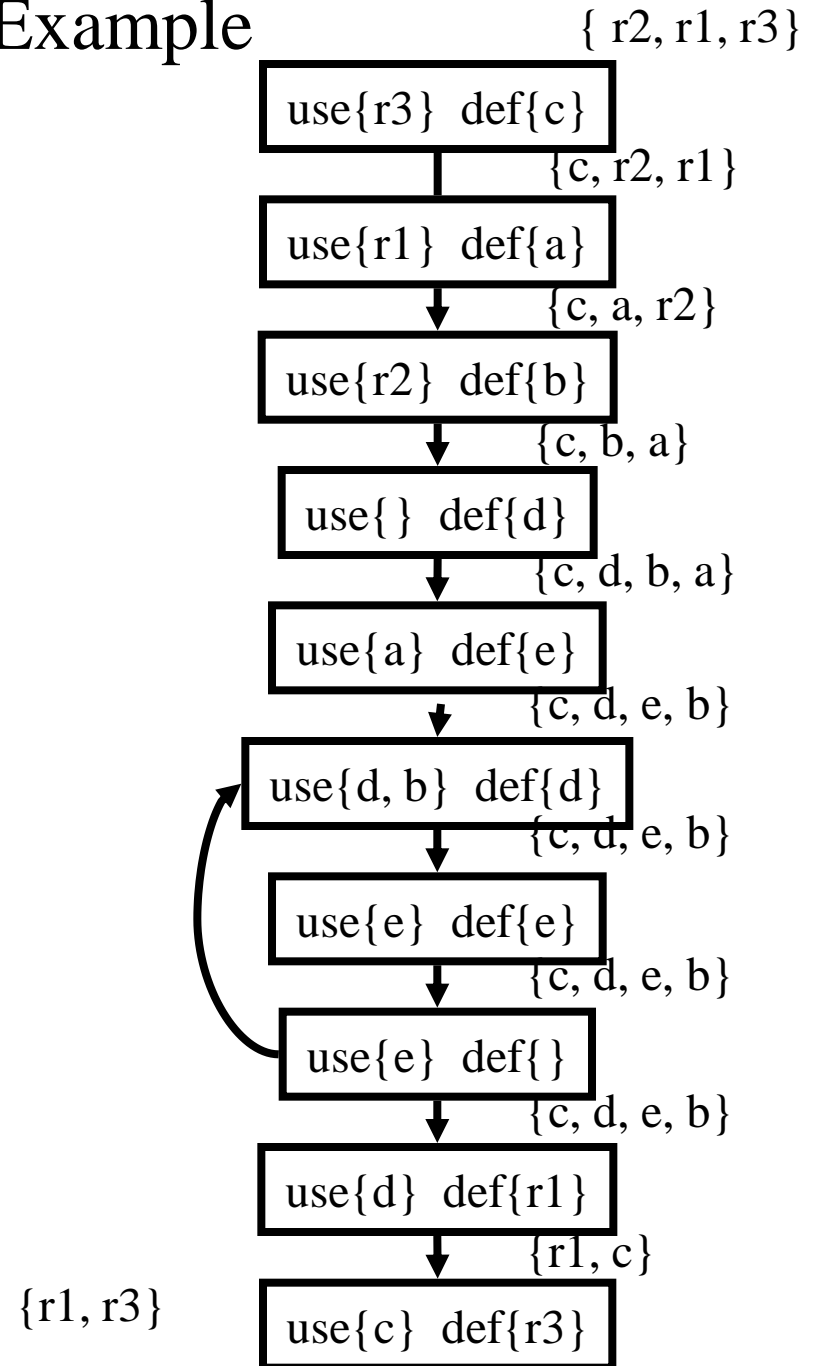
e := e-1

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 */

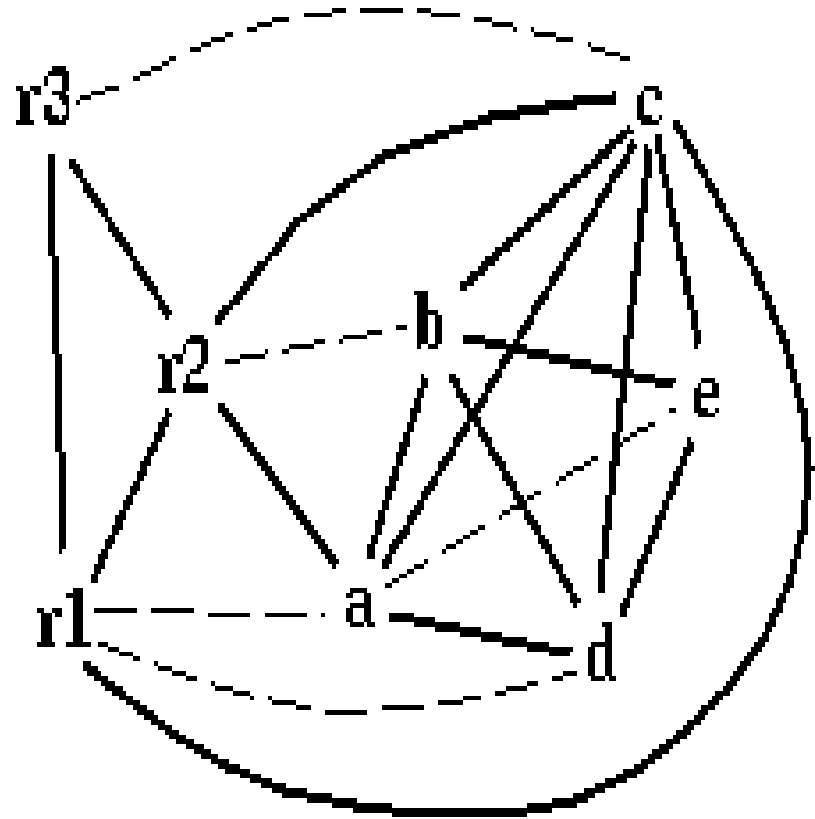
```

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 */

```



$$\text{spill priority} = (\text{uo} + 10 \text{ui})/\text{deg}$$

```

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 */

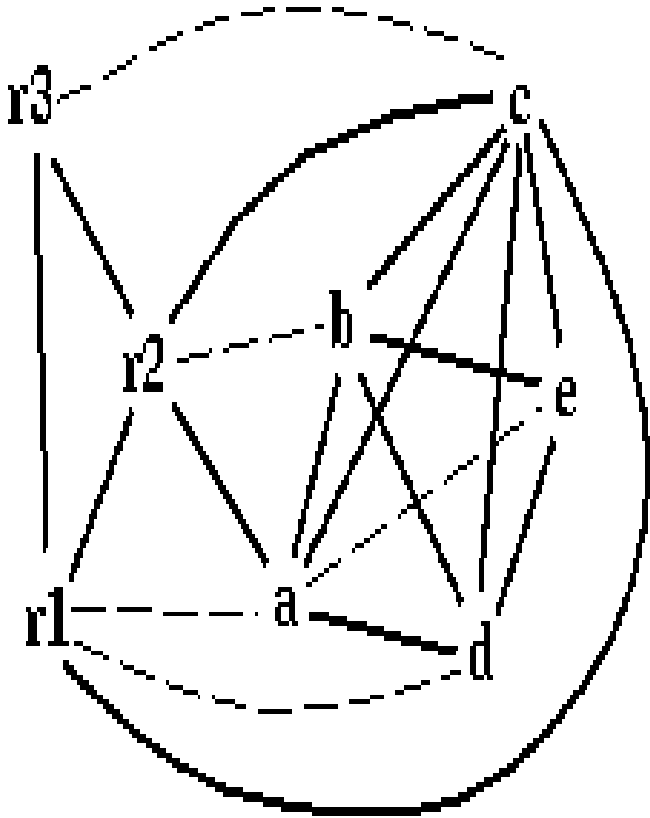
loop:
d := d+b /* e, c, b, d */
e := e-1 /* e, c, b, d */
if e>0 goto loop /* c, d */
r1 := d /* r1, c */
r3 := c /* r1, r3 */

return /* r1, r3 */

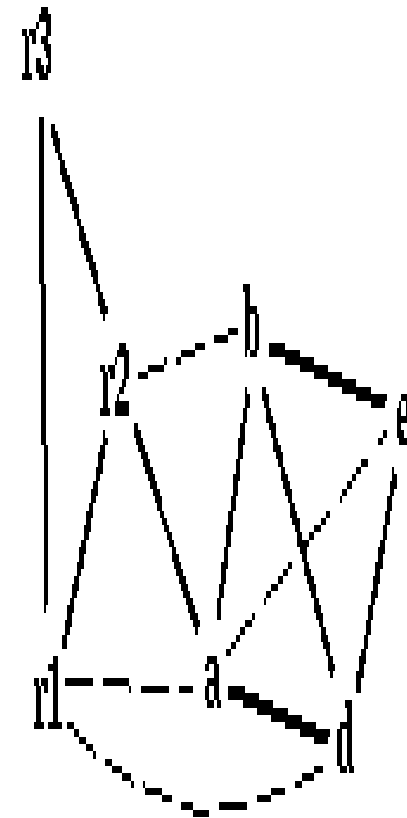
```

	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



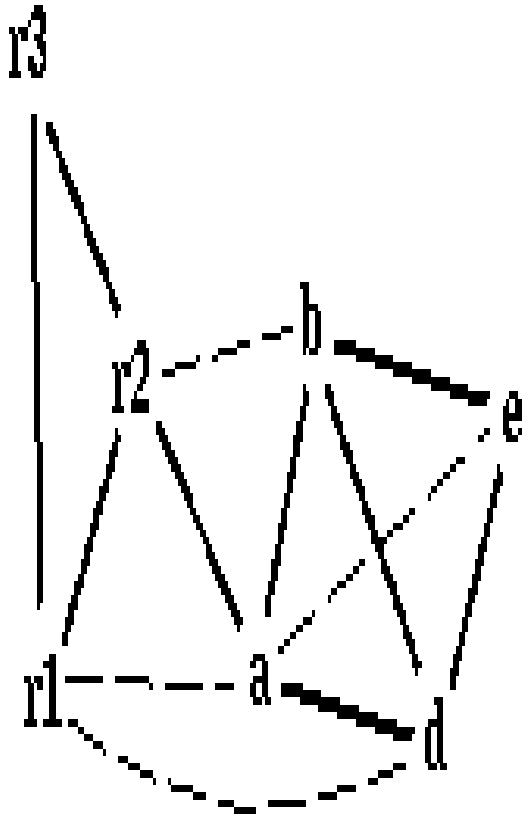
stack



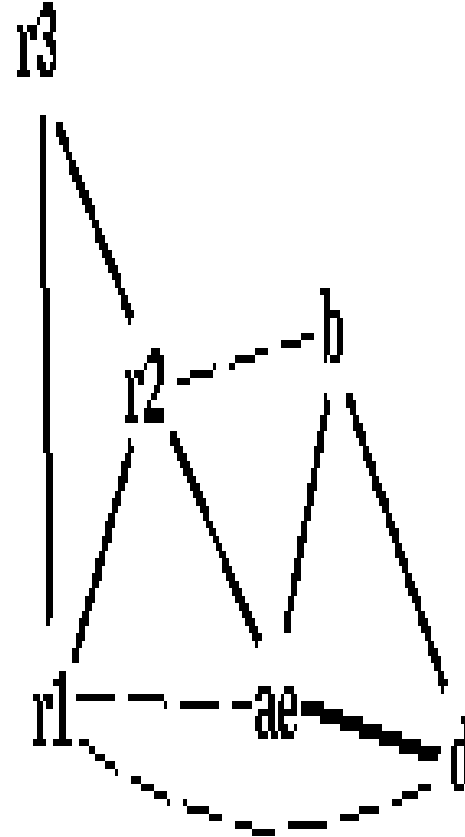
stack



Coalescing $a+e$



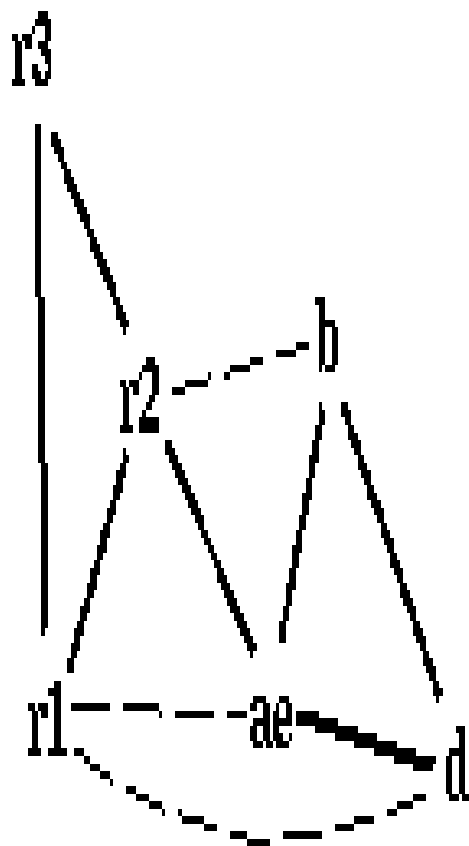
stack



stack



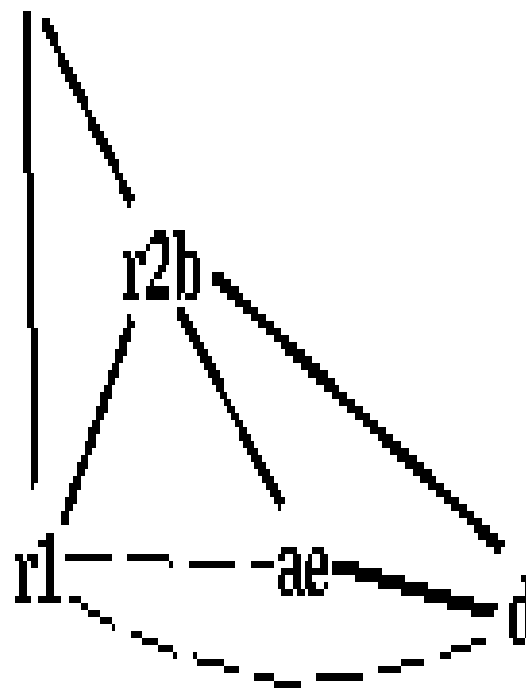
Coalescing $b+r2$



stack



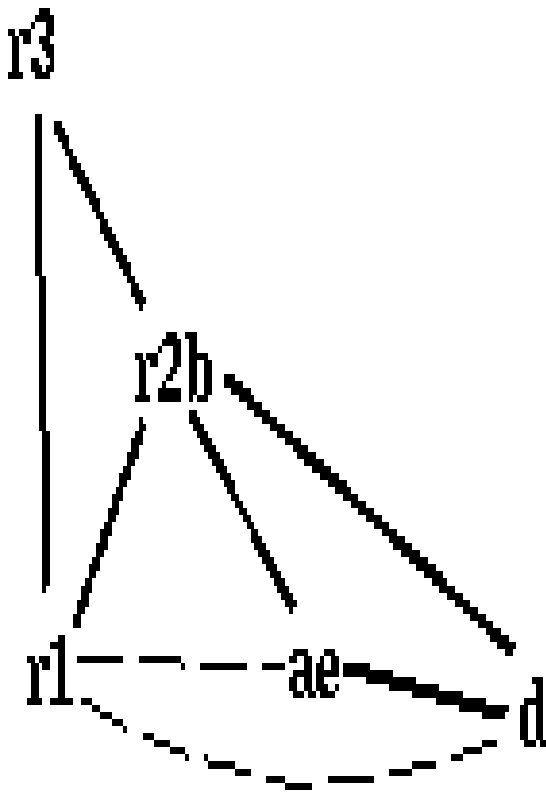
$r3$



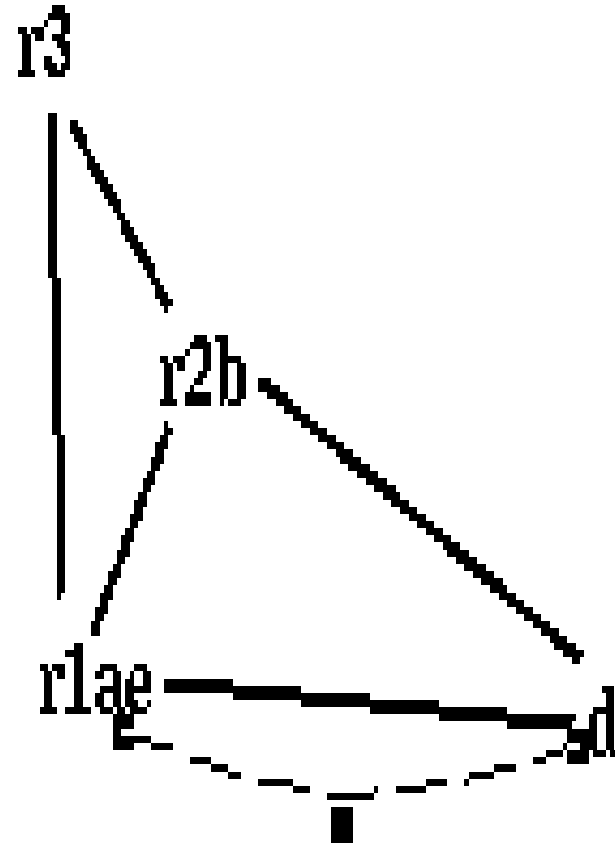
stack



Coalescing $ae+r1$



stack

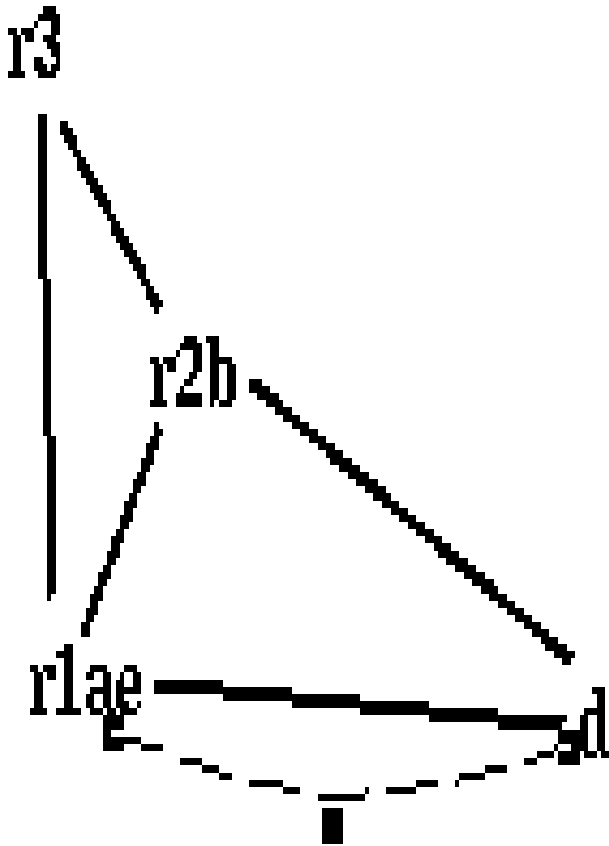


stack

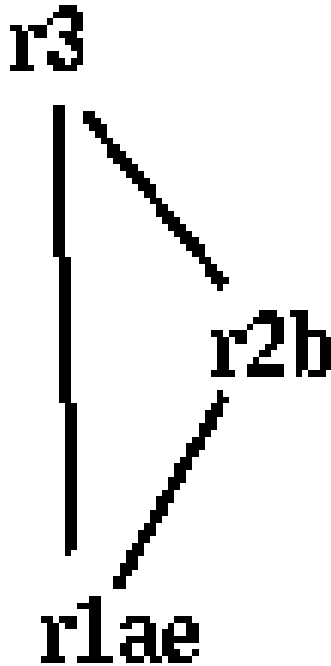


$r1ae$ and d are constrained

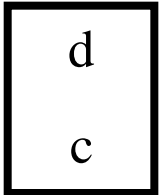
Simplifying d



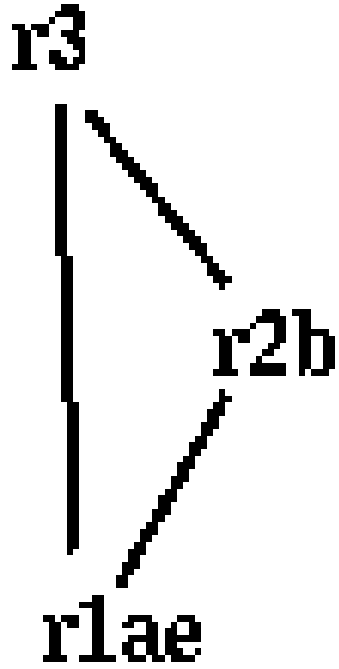
stack



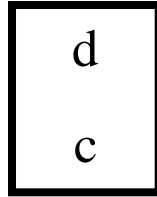
stack



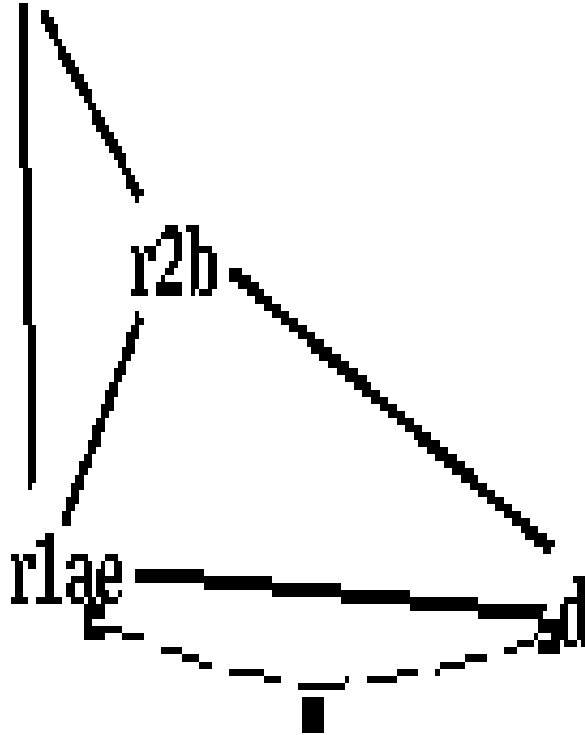
Pop d



stack



r3

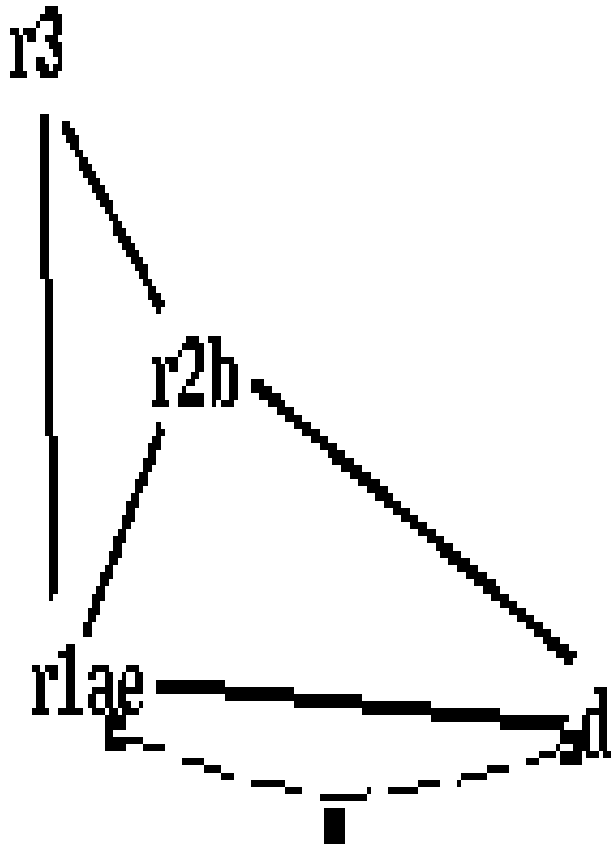


stack

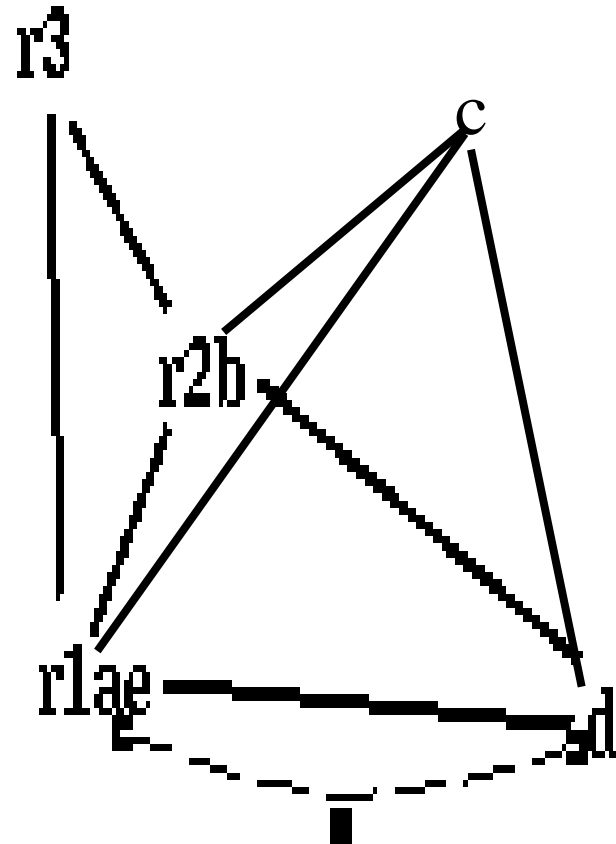
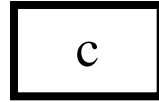


d is assigned to **r3**

Pop *c*



stack



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 */ loop:  
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 */
```

```
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 */
```

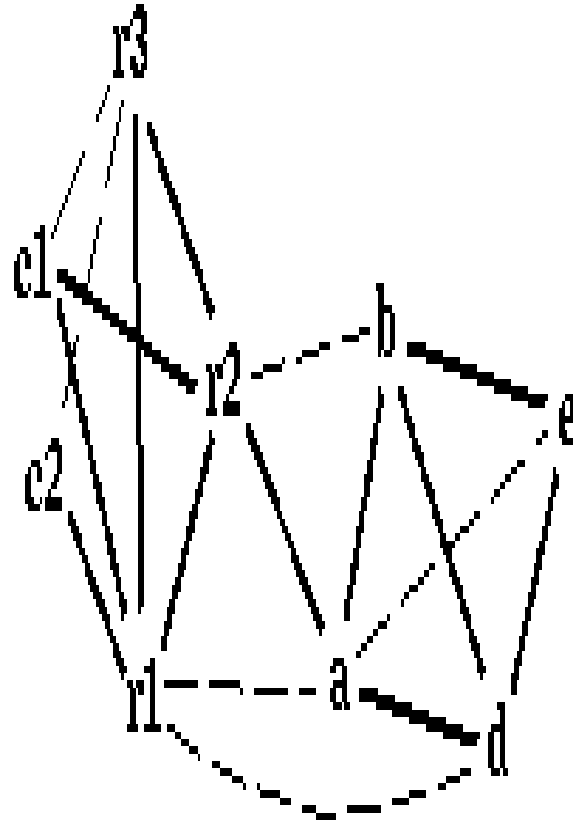
```

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 */

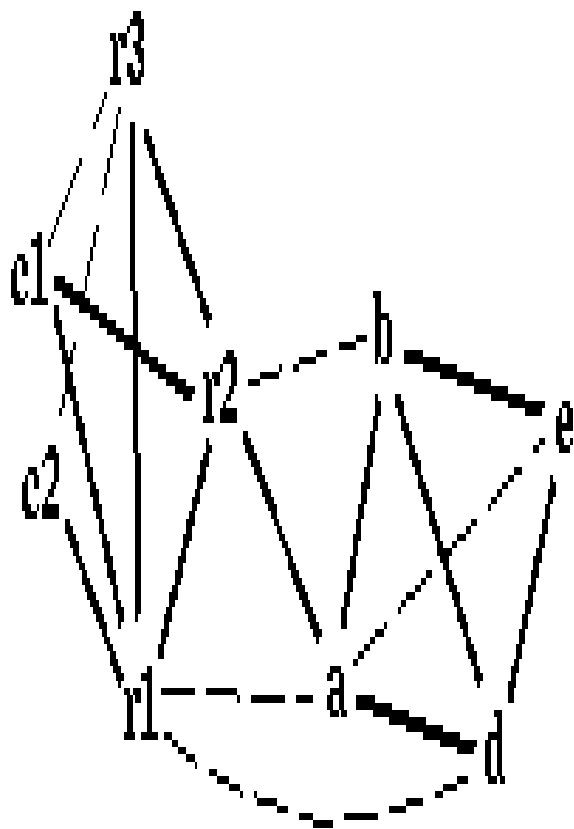
loop:
            d := d+b /* e, b, d */
            e := e-1 /* e, b, d */
            if e>0 goto loop /* d */
            r1 := d /* r1 */
            c2 := M[c_loc] /* r1, c2 */
            r3 := c2 /* r1, r3 */

return /* r1, r3 */

```



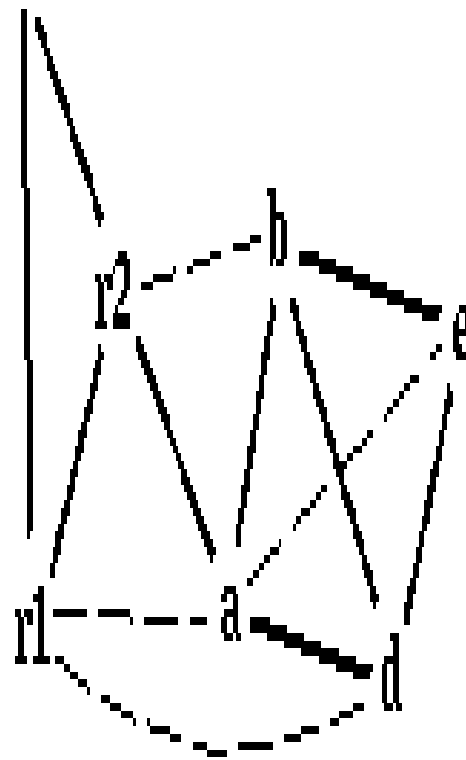
Coalescing c1+r3; c2+c1r3



stack



r3c1c2

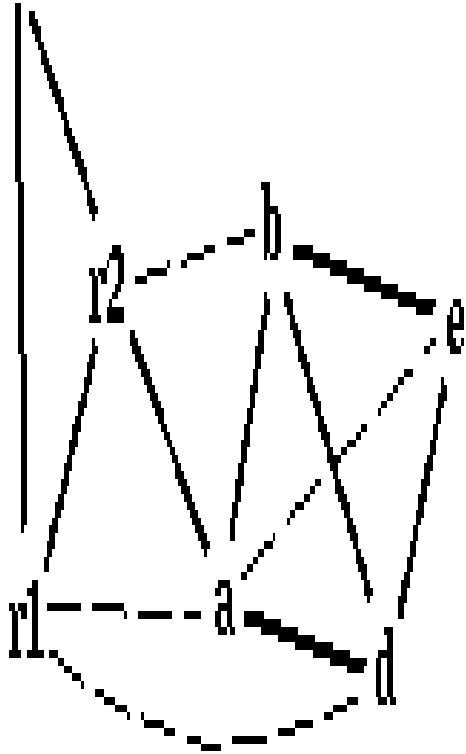


stack



Coalescing a+e; b+r2

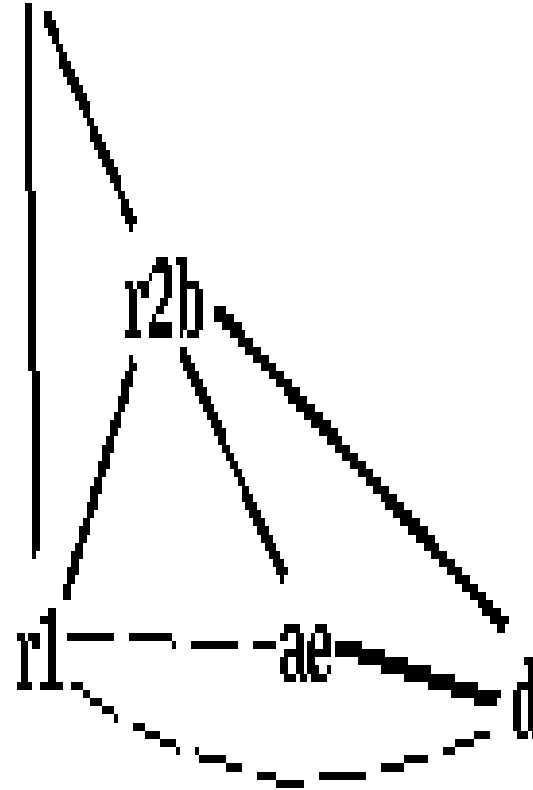
r3c1c2



stack



r3c1c2

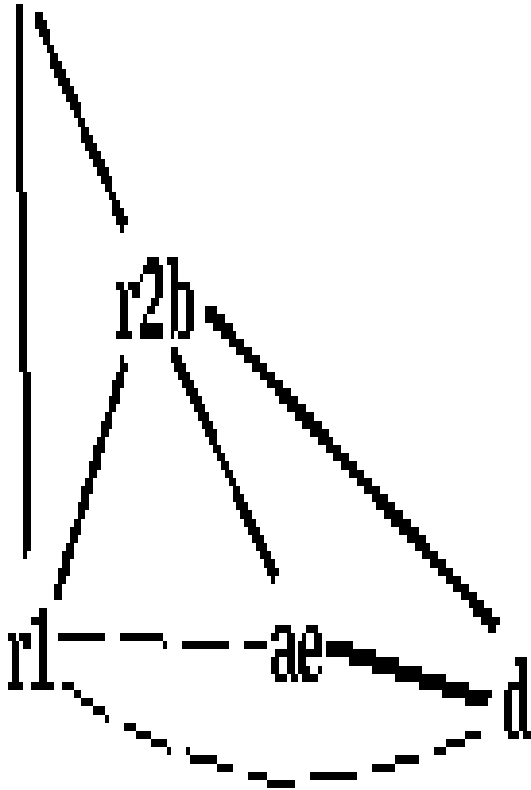


stack



Coalescing ae+r1

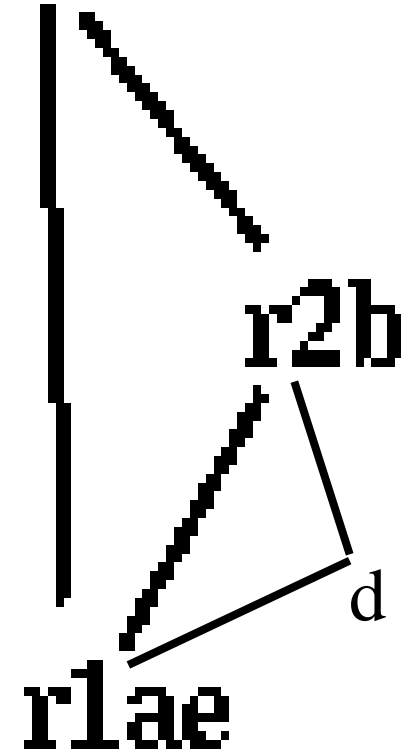
r3c1c2



stack



r3c1c2



stack

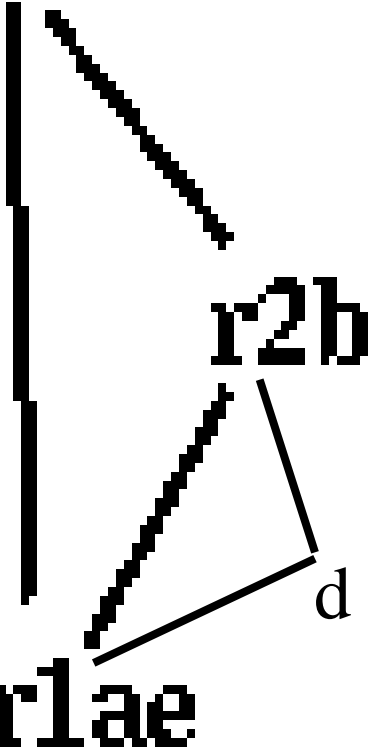


r1ae and **d** are constrained

Simplify d

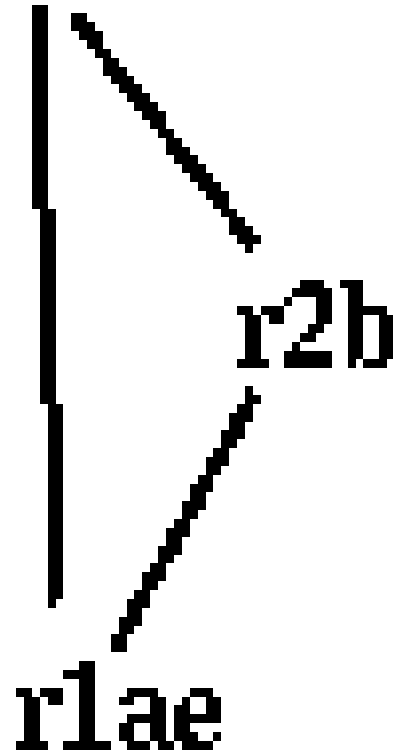
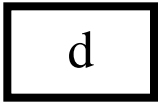
r3c1c2

stack



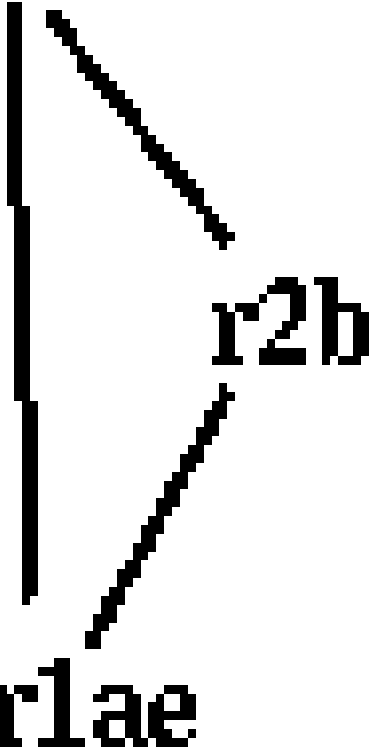
r3c1c2

stack

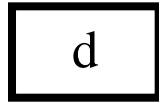


Pop d

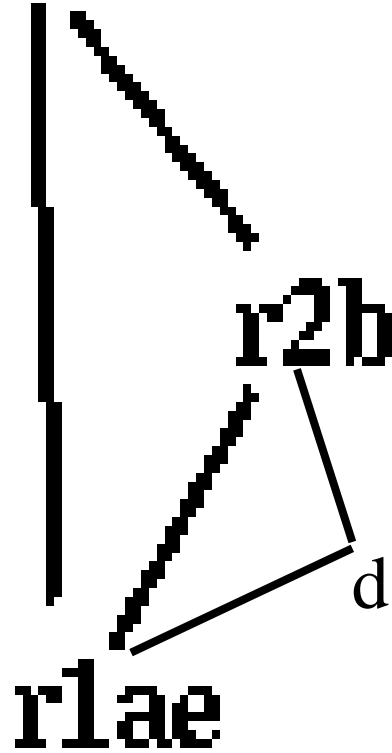
r3c1c2



stack



r3c1c2



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

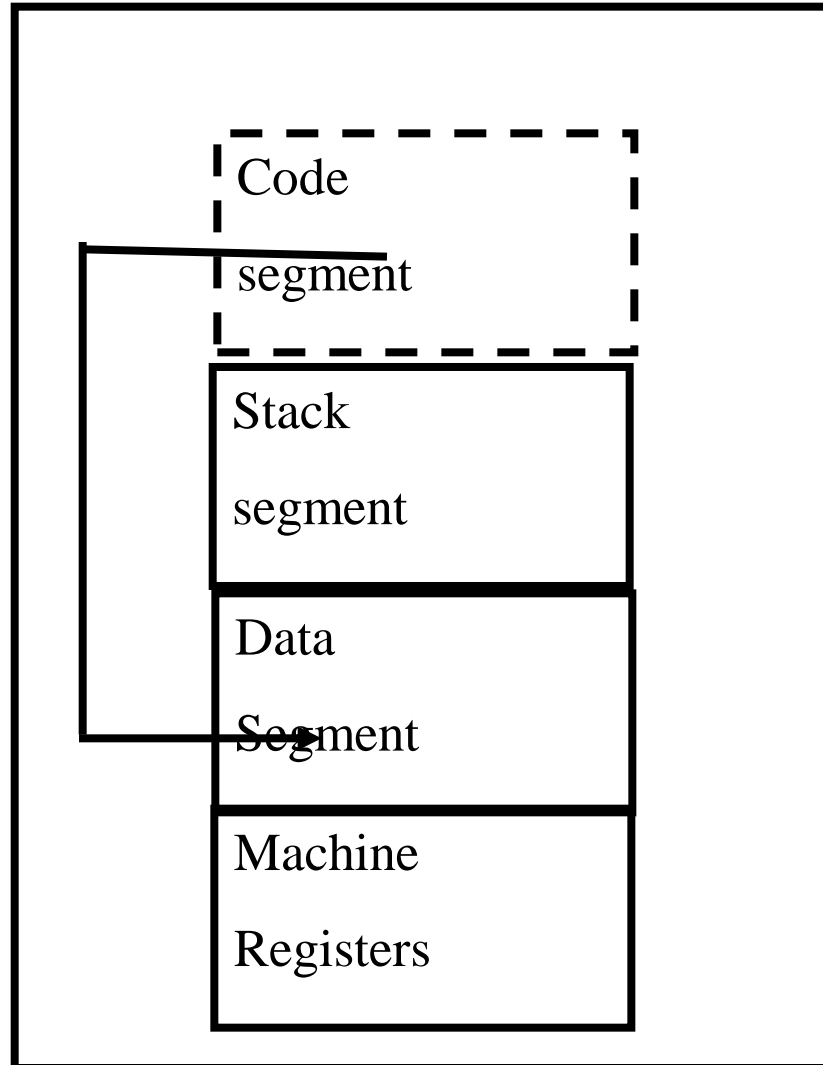
r1 := r3

r3 := M[c_loc]

return /* r1,r3 */

main: addiu \$sp,\$sp, -K1	nfactor: addiu \$sp,\$sp,-K2	or \$25,\$0,\$2
L4: sw \$2,0+K1(\$sp)	L6: sw \$2,0+K2(\$sp)	mult \$30,\$25
or \$25,\$0,\$31	or \$25,\$0,\$4	mflo \$30
sw \$25,-4+K1(\$sp)	or \$24,\$0,\$31	L2: or \$2,\$0,\$30
addiu \$25,\$sp,0+K1	sw \$24,-4+K2(\$sp)	lw \$30,-4+K2(\$sp)
or \$2,\$0,\$25	sw \$30,-8+K2(\$sp)	or \$31,\$0,\$30
addi \$25,\$0,10	beq \$25,\$0,L0	lw \$30,-8+K2(\$sp)
or \$4,\$0,\$25	L1: or \$30,\$0,\$25	b L5
jal nfactor	lw \$24,0+K2	L0: addi \$30,\$0,1
lw \$25,-4+K1	or \$2,\$0,\$24	b L2
or \$31,\$0,\$25	addi \$25,\$25,-1	L5: addiu \$sp,\$sp,K2
b L3	or \$4,\$0,\$25	j \$31
L3: addiu \$sp,\$sp,K1	jal nfactor	
j \$31		

Program Runtime State



Assembler/Linker/Loader

- Code generation yields code which is still far from executable
 - Delegate to existing assembler
- Assembler translates symbolic instructions into binary and creates relocation bits
- Linker creates executable from several files produced by the assembly
 - Dynamic linker of libraries
- Loader creates an image from executable

Dynamic Allocation

- Long lived objects
- Allocated by the programmer
 - malloc(), new
- Deallocated by explicit commands or garbage collection
 - free
- Resides in the data area

Garbage Collection Techniques

- Reference Counts
- Tracing
 - Mark and Sweep
 - Copying
- Generational
- Incremental