Code Generation

Mooly Sagiv

http://ellcc.org/demo/index.cgi
llvm.org
https://www.cis.upenn.edu/~stevez/ CS341

Outline

- Recap Register Allocation
- Generating LLVM for imperative programs
  - Local Variables
  - Expressions
  - Assignments
  - Boolean Expressions
  - Control Flow

Register Allocation

- Map symbolic registers into physical
- Chose between caller= and callee-save registers
- Reuse machine registers
- Avoid store/loads
- Sometimes eliminate mov
  - Allocate the same register to source and target

A Simple Example

```
L0:     a ← 0
L1:     b ← a + 1
        c ← c + b
        a ← b * 2
        if c < N goto L1
        return c
```

```
L0:     r1← 0
L1:     r1← r1 + 1
        r2 ← r2 + r1
        r1 ← r1 * 2
        if r2 < N goto L1
        return r2
```

Can this be implemented in a machine with two registers?

Live symbolic registers

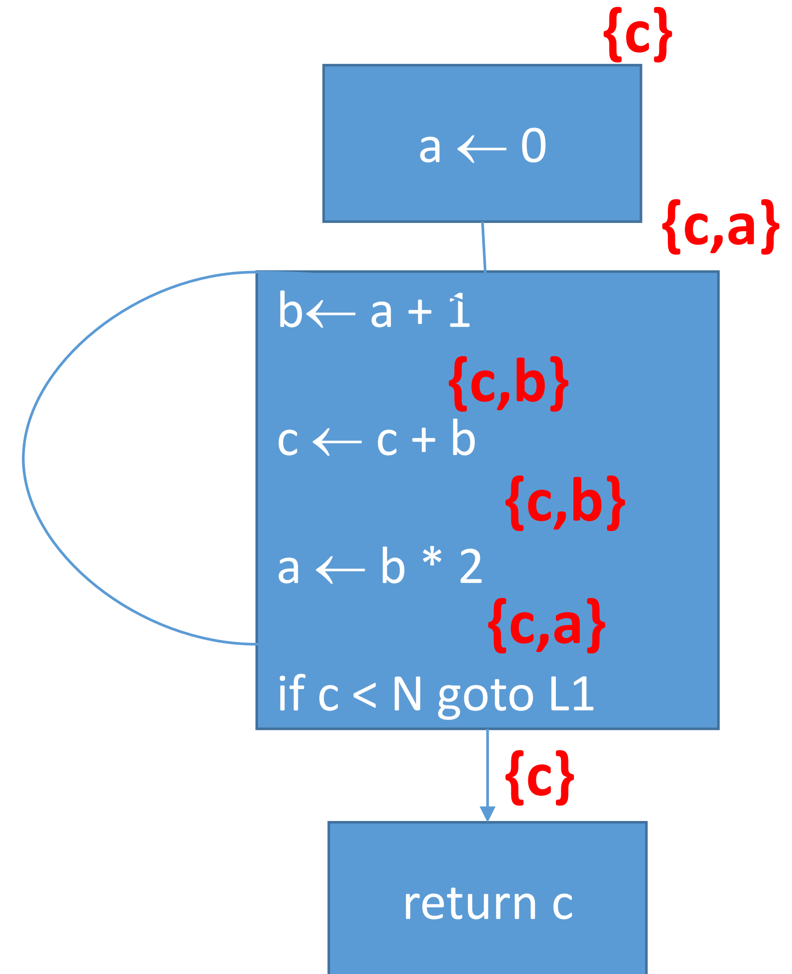
- A symbolic register is live at a program point if it may be used before set on some path from this point
- A symbolic register is not live (dead) at a program point if it is not used on all paths from this point

Using Liveness information

- Symbolic Registers which are not live together can share the same symbolic register

Liveness in the example

```
L0:     a ← 0

L1:     b ← a + 1

        c ← c + b

        a ← b * 2

        if c < N goto L1
        return c
```



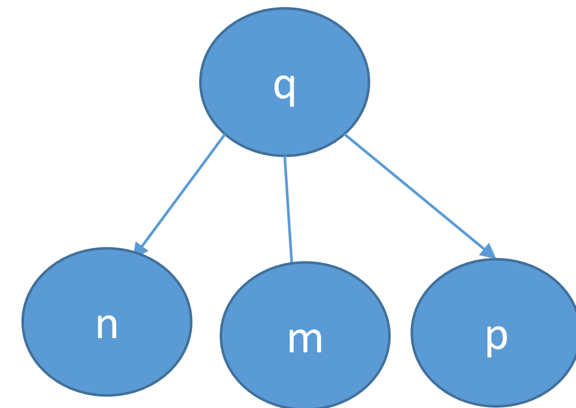
Iteratively Computing Liveness

- Start with dead variables at all program points
    - The return value is live
- Iteratively add live variables by backward propagation
- Terminate when no live variables are added

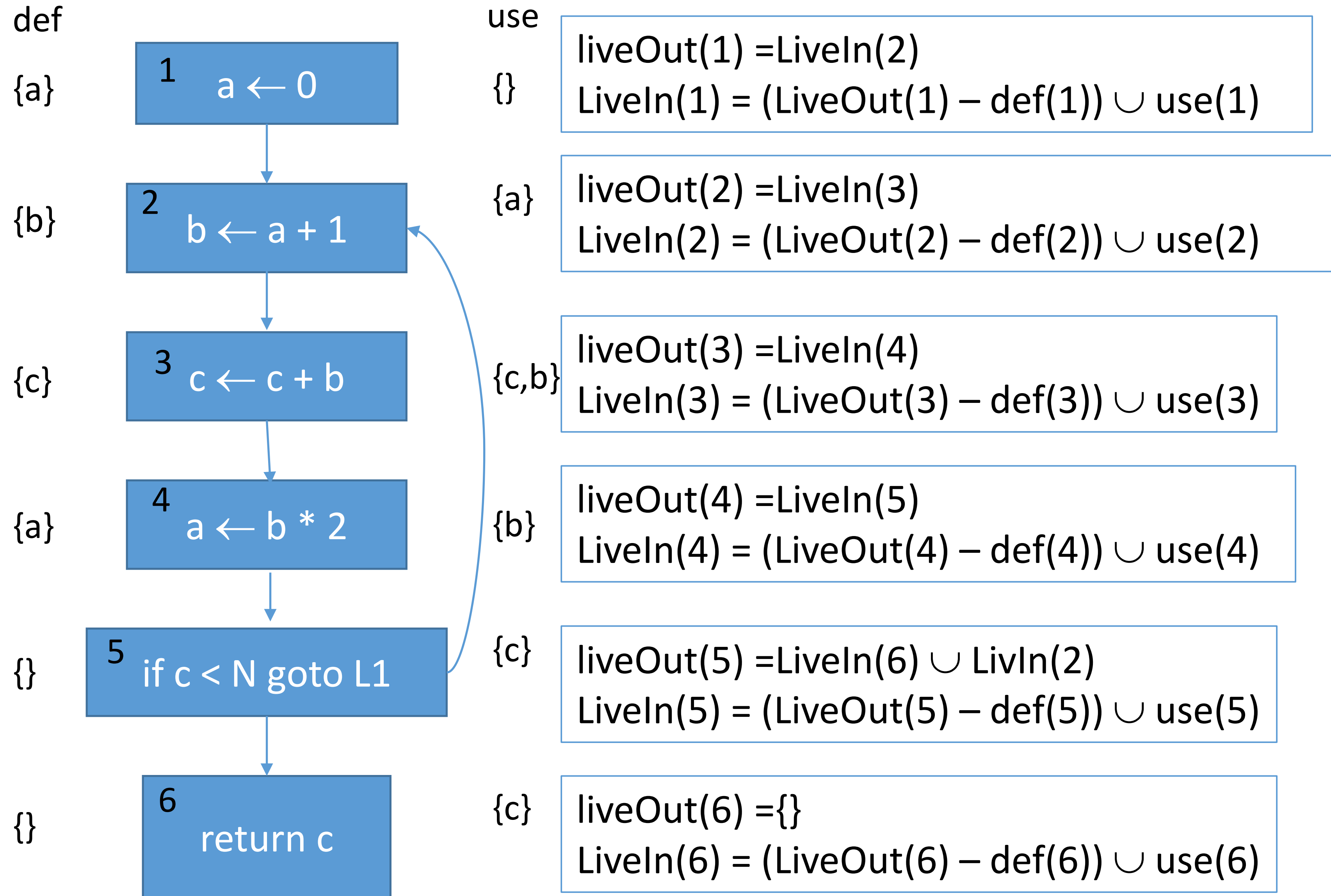
Iteratively Computing Liveness

- Construct a control flow graph of instructions
  - Every instruction uses a set of variables and defines a set of variables
    - example x = y+z
      - use({y, z})
      - def({x})
- Liveness Equations
  - liveOut(exit) = {}
  - liveIn(n) = (liveOut(n) – def(n)) $\cup$ use(n)
  - liveOut(n) = $\cup_{m:\ succ(n,\ m)}$ liveIn(m)
- Computed iteratively from the exit node

liveOut(q)=liveIn(n) $\cup$ liveIn(m)) $\cup$ liveIn(p)



Liveness Recursive Equations



liveOut(1) =LiveIn(2)
LiveIn(1) = (LiveOut(1) – def(1)) ∪ use(1)

liveOut(2) =LiveIn(3)
LiveIn(2) = (LiveOut(2) – def(2)) ∪ use(2)

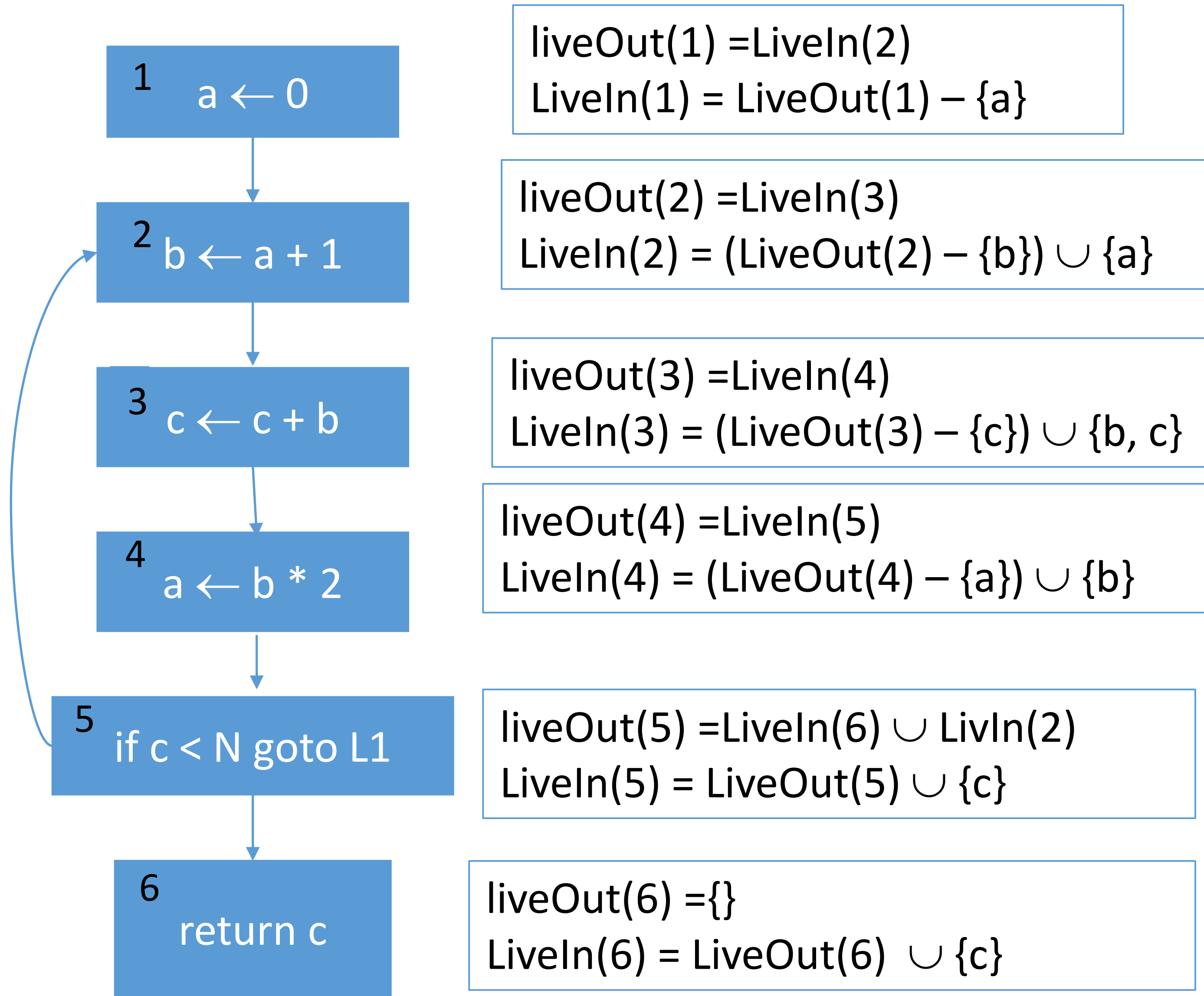
liveOut(3) =LiveIn(4)
LiveIn(3) = (LiveOut(3) – def(3)) ∪ use(3)

liveOut(4) =LiveIn(5)
LiveIn(4) = (LiveOut(4) – def(4)) ∪ use(4)

liveOut(5) =LiveIn(6) ∪ LivIn(2)
LiveIn(5) = (LiveOut(5) – def(5)) ∪ use(5)

liveOut(6) ={}
LiveIn(6) = (LiveOut(6) – def(6)) ∪ use(6)

Iteratively Computing Liveness



Node	LiveIn(Node)
6	{c}
5	{c}
4	{b,c}
3	{b,c}
2	{c, a}
5	{c, a}
4	{b, c}
3	{b, c}
2	{c, a}
1	{c}

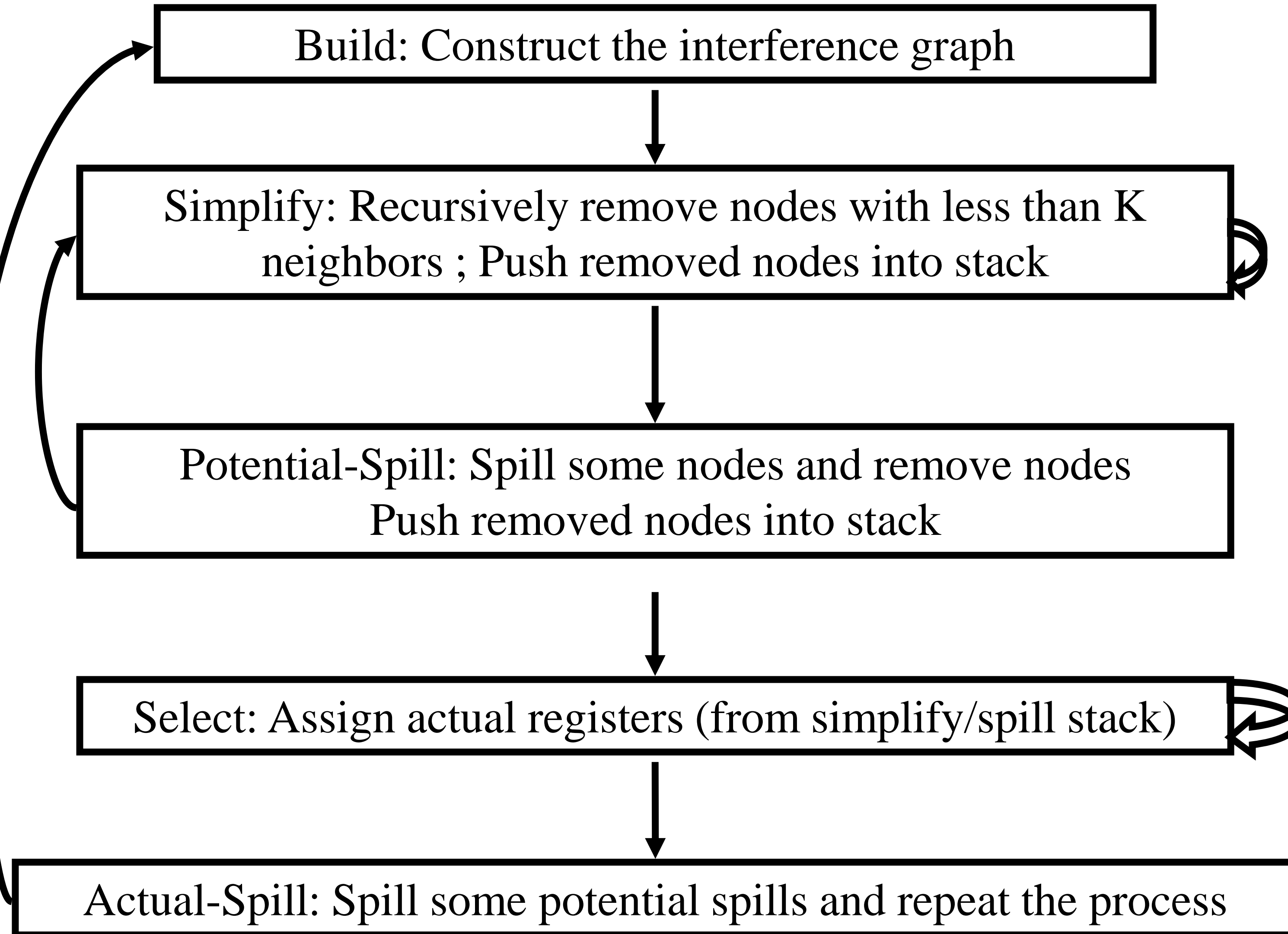
Constructing interference graphs

- Compute liveness information at every instruction
- Variables 'a' and 'b' interfere when there exists an instruction n: a $\leftarrow$ exp and 'b' $\in$ LiveOut[n]

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



Challenges

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

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)

Constructing interference graphs (take 2)

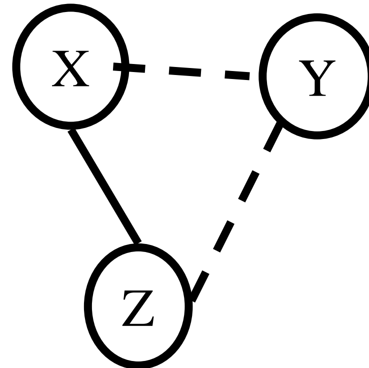
- Compute liveness information at every instruction
- Two types of edges: interfere and move
  - Variable ‘a’ and ‘b’ are mov if there exists an instruction n: a $\leftarrow$ b
  - Variables ‘a’ and ‘b’ interfere when there exists an instruction n: a $\leftarrow$ exp, exp $\neq$b and ‘b’ $\in$ LiveOut[n]

Constrained Moves

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

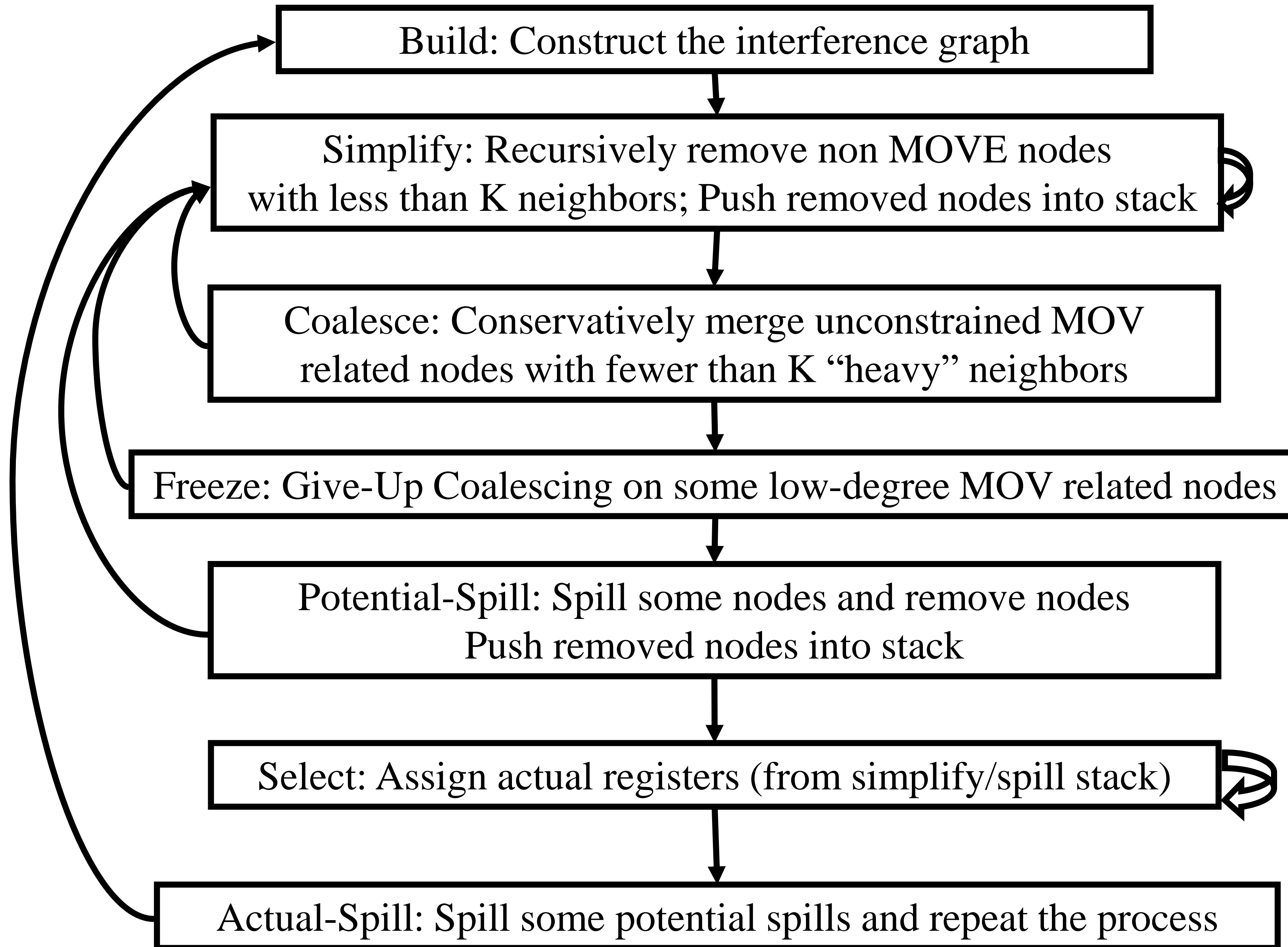
X ←Y   /* X, Y, Z */

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

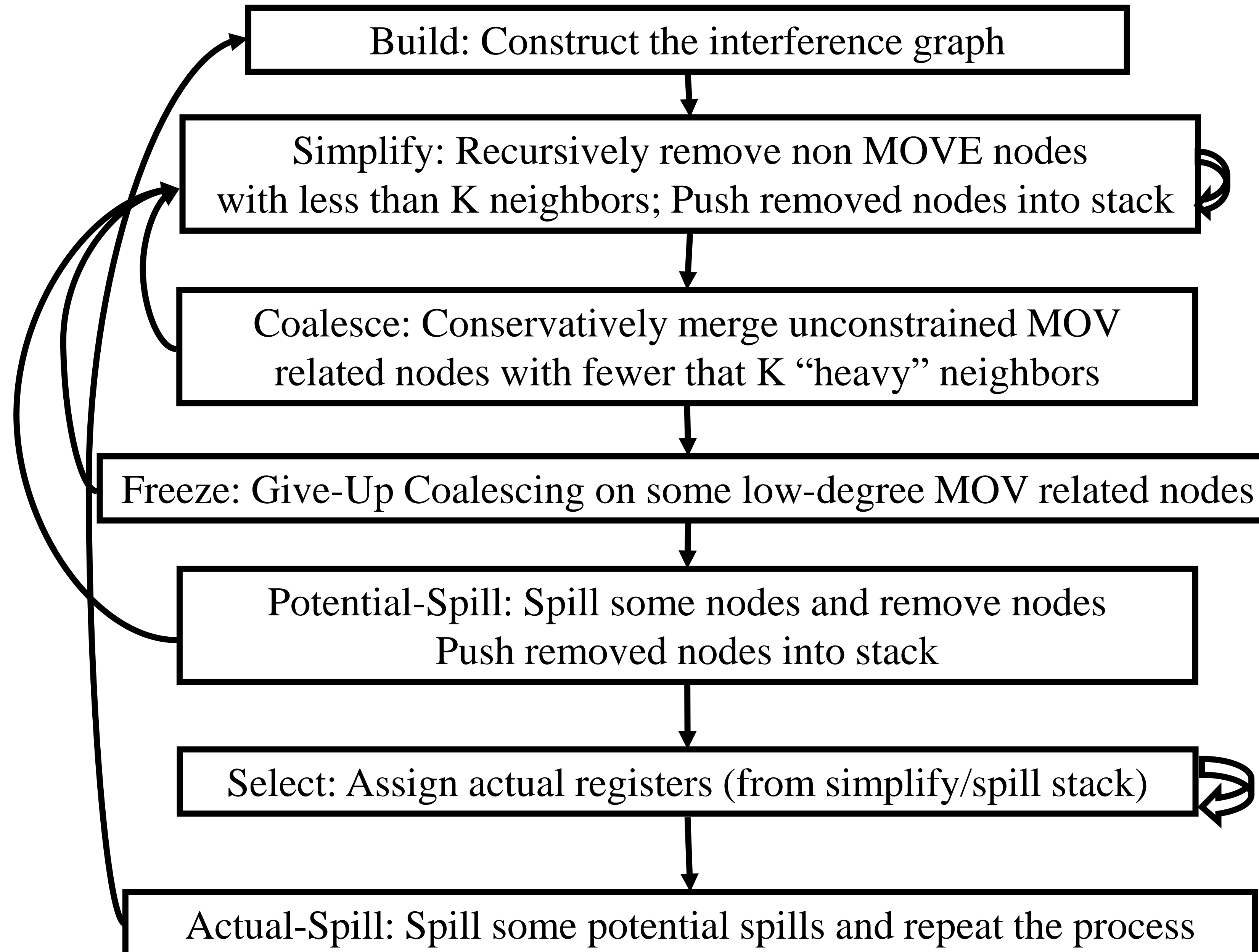
Pre-Colored Nodes

- 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

A Complete Example (Andrew Appel) https://www.cs.princeton.edu/~appel/

```
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

a := r1

b := r2

d := 0

e := a

loop:

d := d+b

e := e-1

if e>0 goto loop

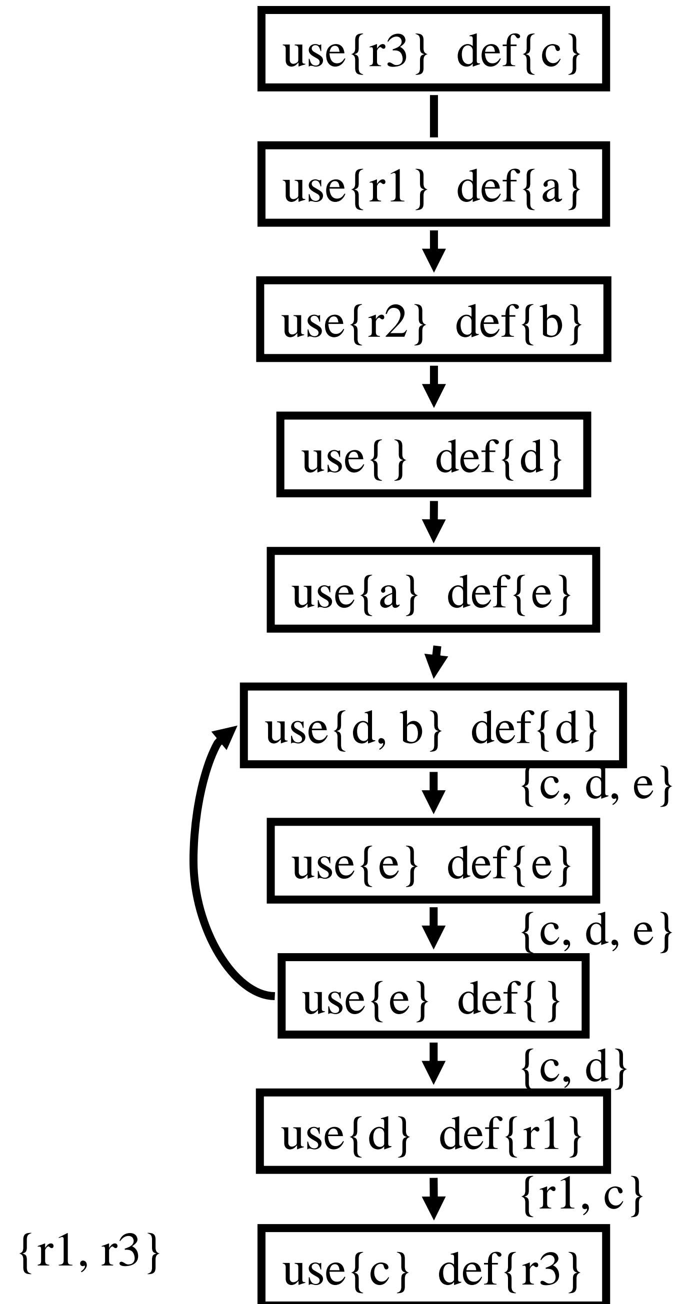
r1 := d

r3 := c

return /* r1,r3 */

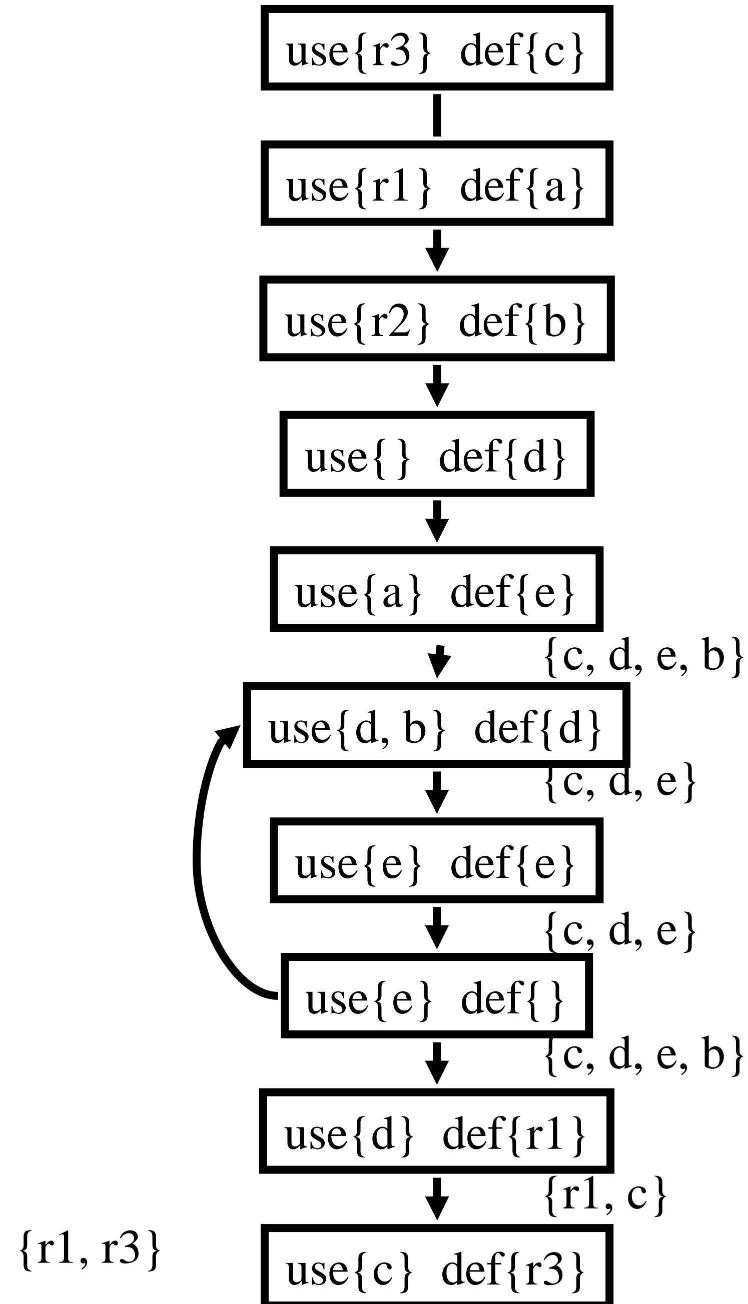
r1, r2 caller save

r3 callee-save



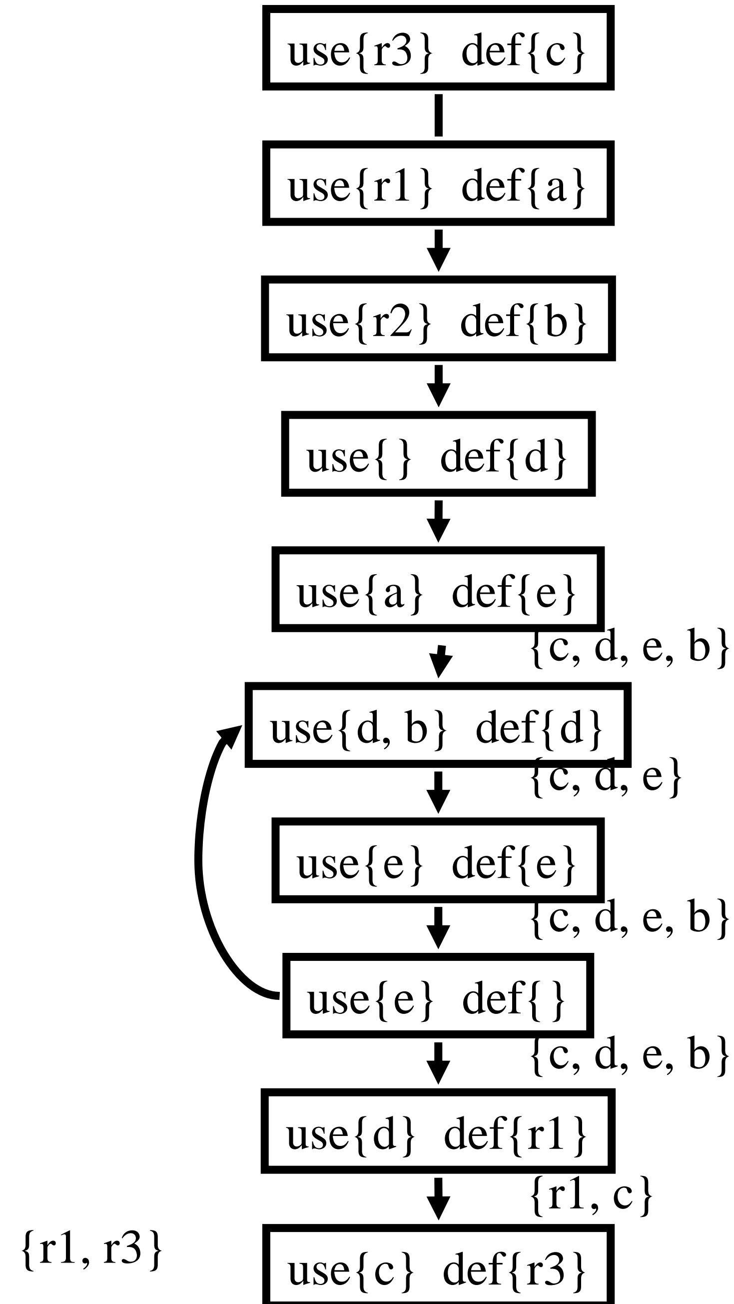
A Complete Example

```
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

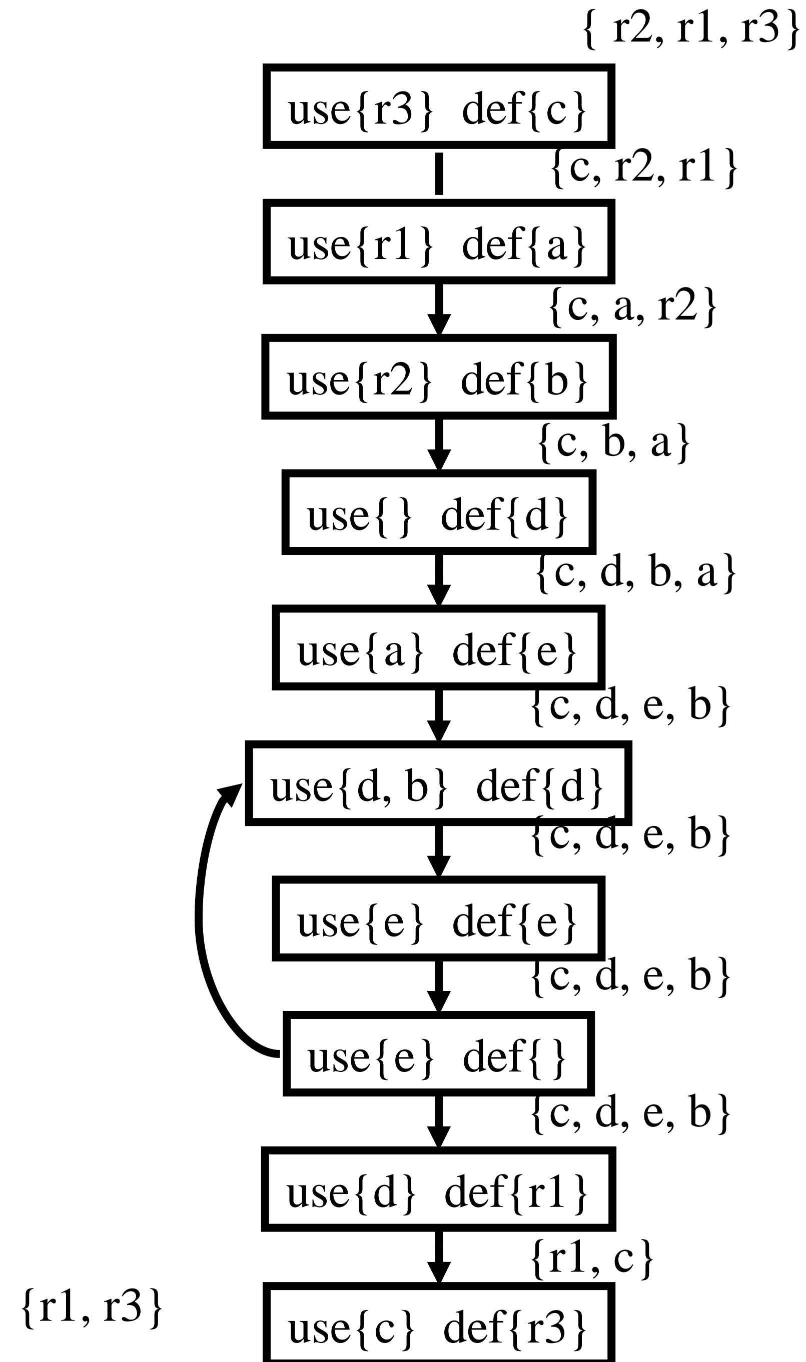
e := e-1

if e>0 goto loop

r1 := d

r3 := c

return /* r1,r3 */

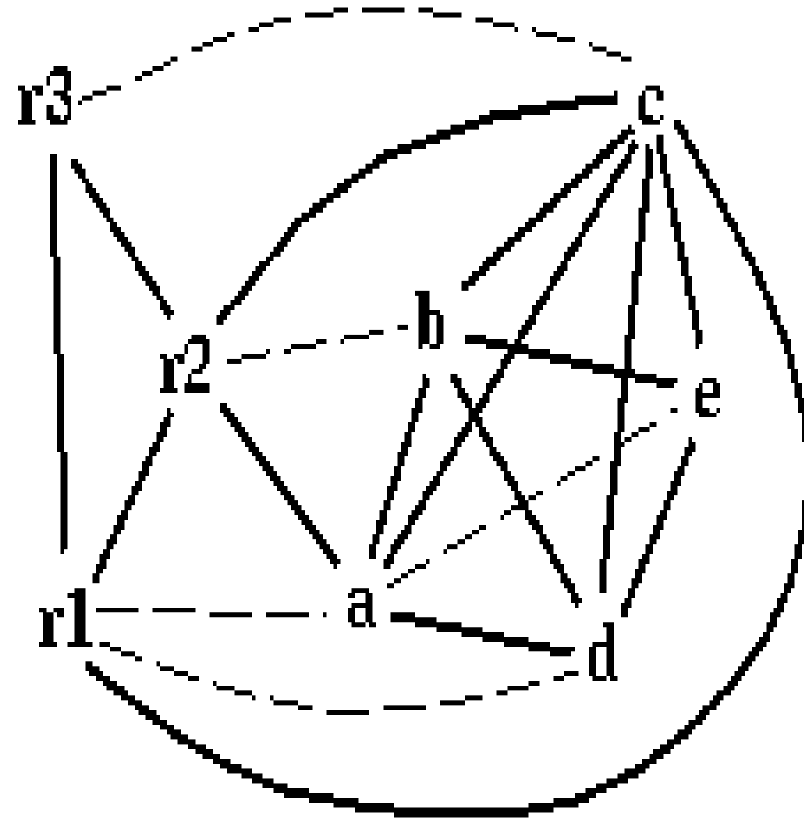


Live Variables Results

```
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 */
```

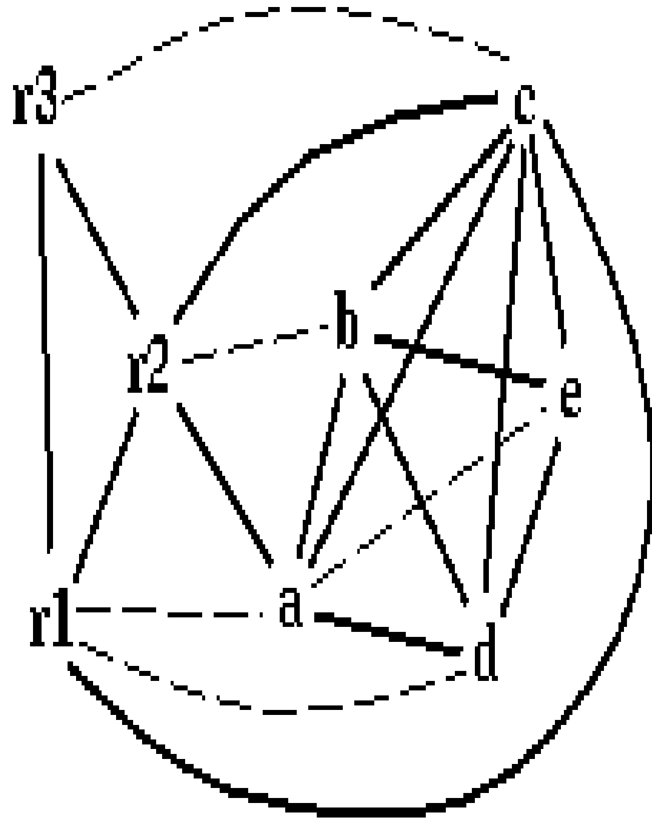


spill priority = (uo + 10 ui)/deg

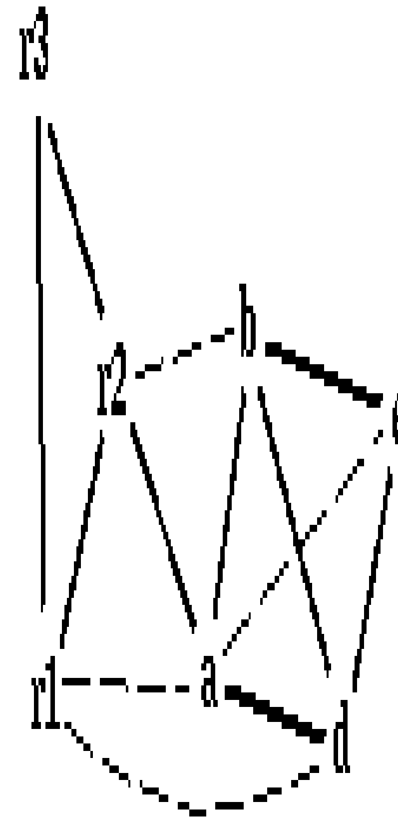
```
enter:              /* r2, r1, r3 */
        c := r3   /* c, r2, r1 */
        a := r1   /* a, c, r2 */
        b := r2   /* a, c, b */
        d := 0    /* a, c, b, d */
        e := a    / * e, c, b, d */
loop:
        d := d+b  /* e, c, b, d */
        e := e-1 /* e, c, b, d */
        if e>0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

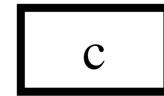
Spill C



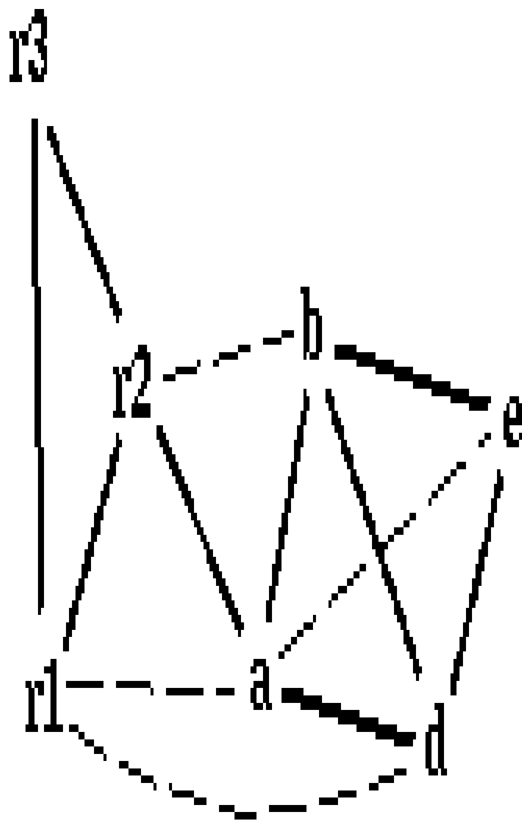
stack



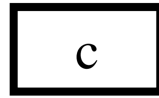
stack

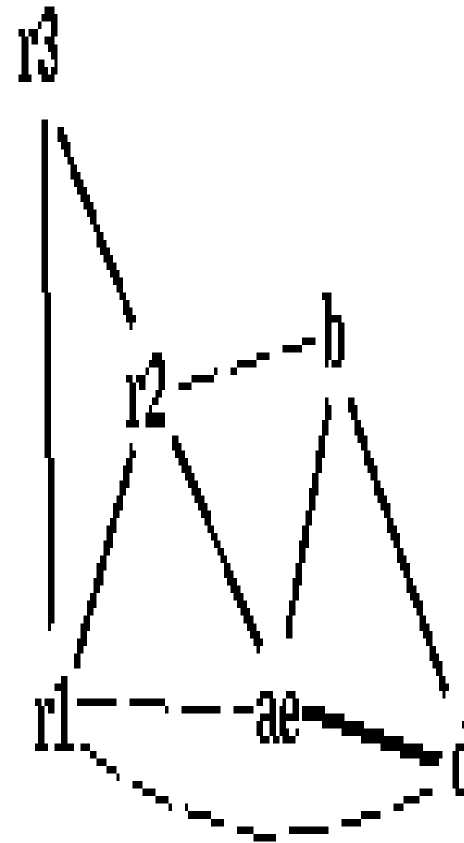


Coalescing *a+e*

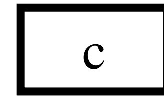


stack

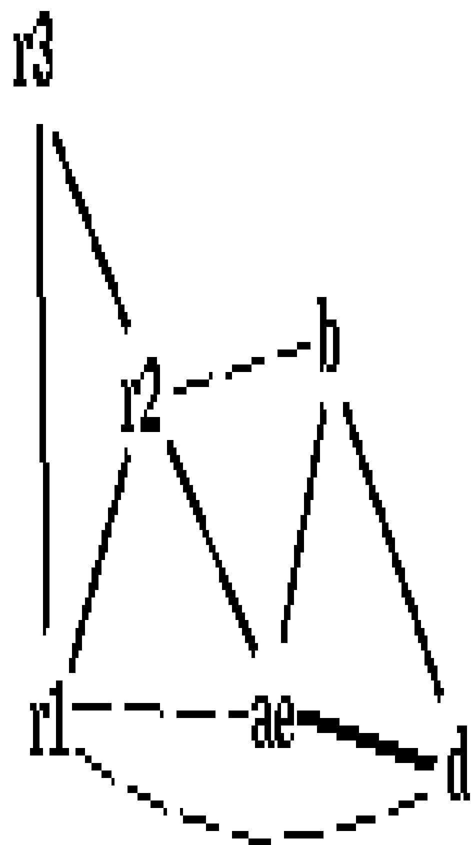




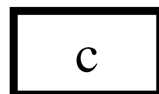
stack

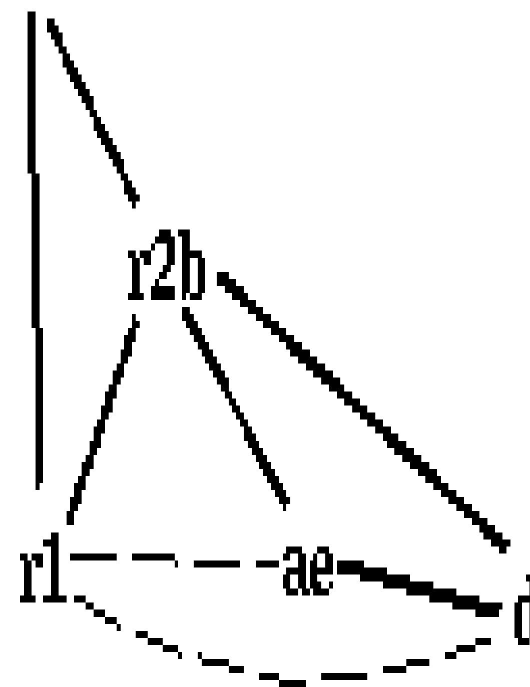


Coalescing *b+r2*



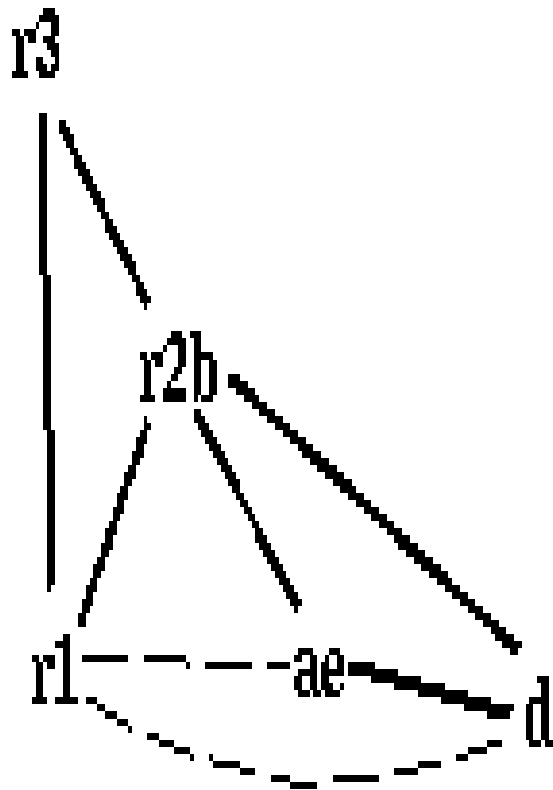
stack



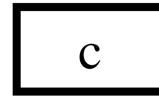


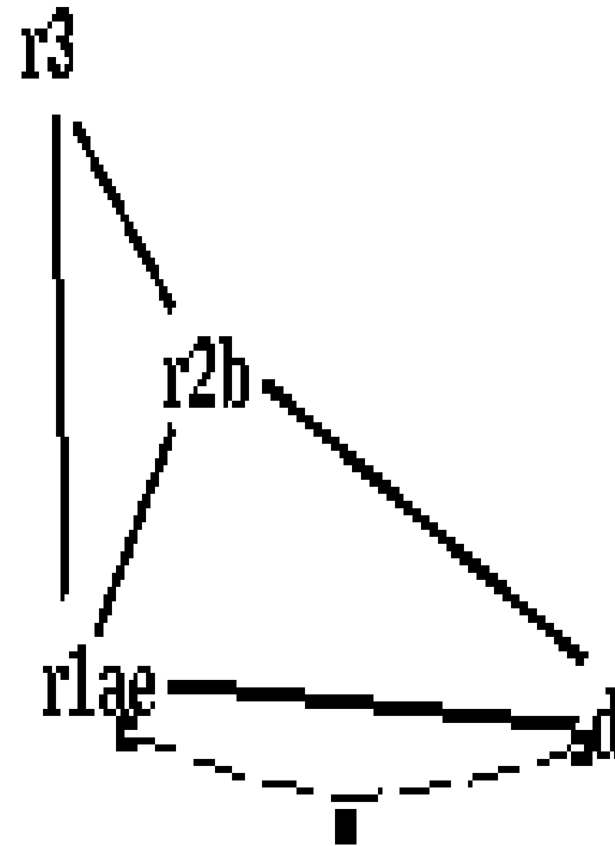
stack

Coalescing *ae+r1*

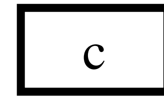


stack



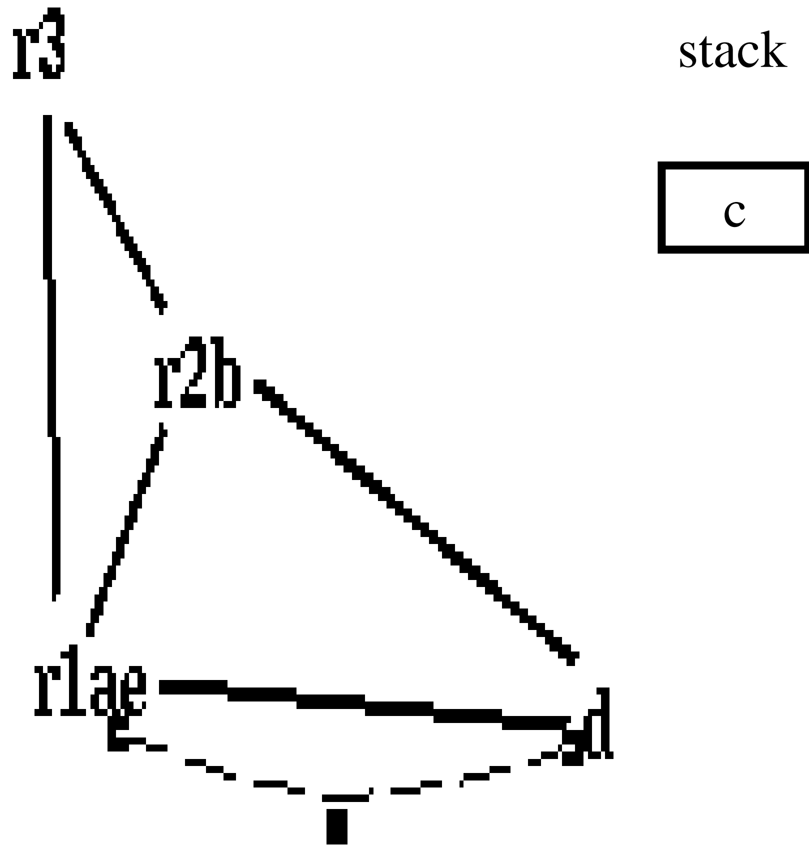


stack



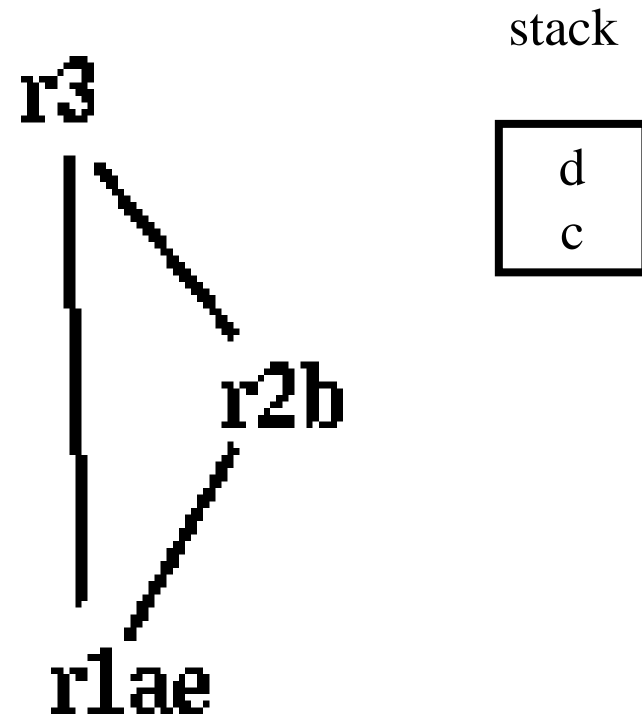
r1ae and d are constrained

Simplifying *d*



stack

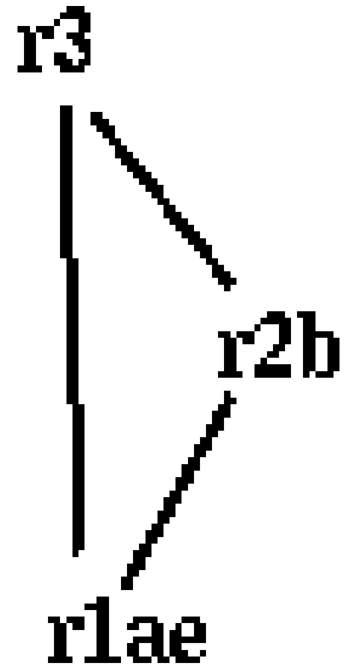
c



stack

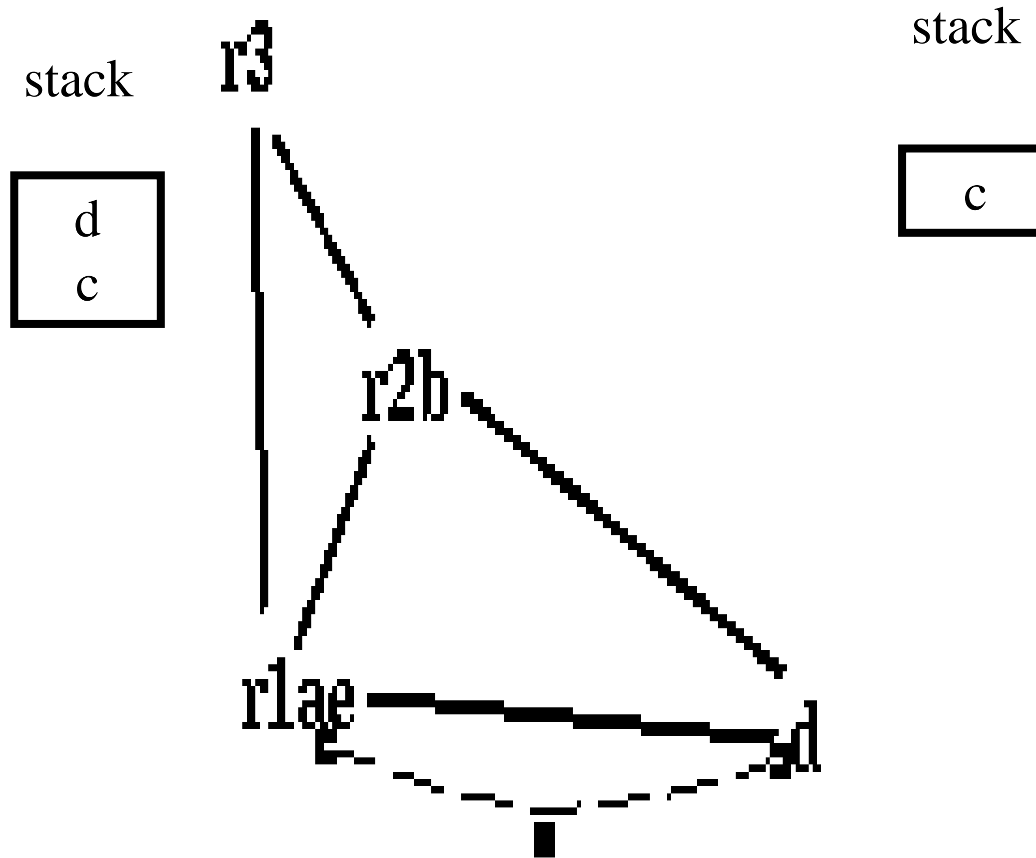
d
c

Pop *d*



stack

d
c

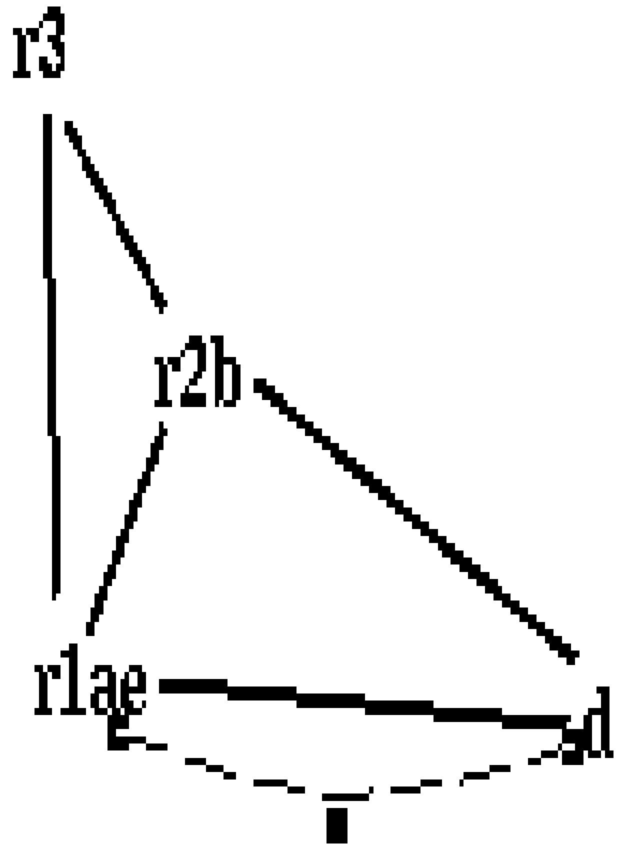


stack

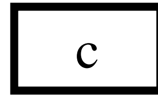
c

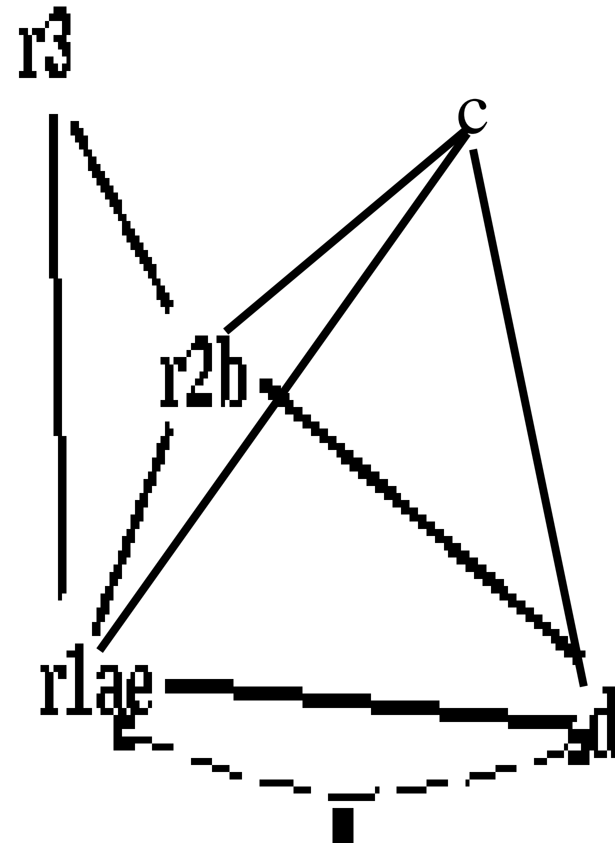
d is assigned to r3

Pop *c*



stack





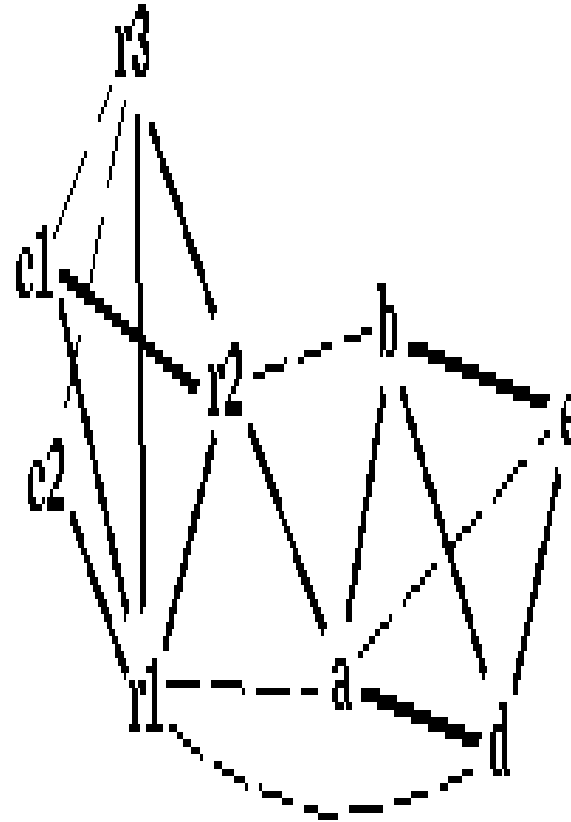
stack

actual spill!

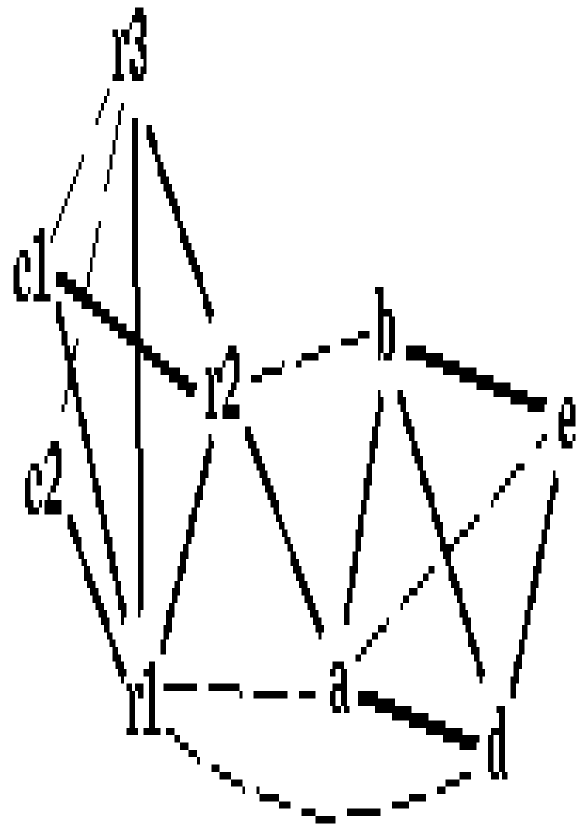
```
enter:            /* r2, r1, r3 */
      c := r3   /* c, r2, r1 */
      a := r1   /* a, c, r2 */
      b := r2   /* a, c, b */
      d := 0    /* a, c, b, d */
      e := a    / * e, c, b, d */
loop:
      d := d+b  /* e, c, b, d */
      e := e-1 /* e, c, b, d */
      if e>0 goto loop /* c, d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

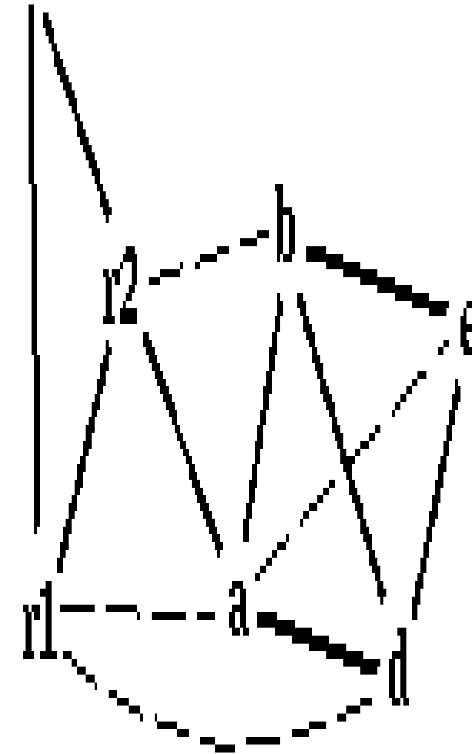


Coalescing c1+r3; c2+c1r3



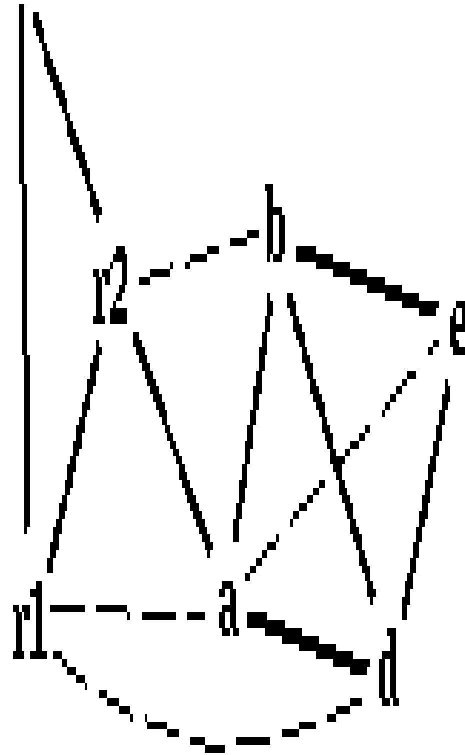
stack

r3c1c2

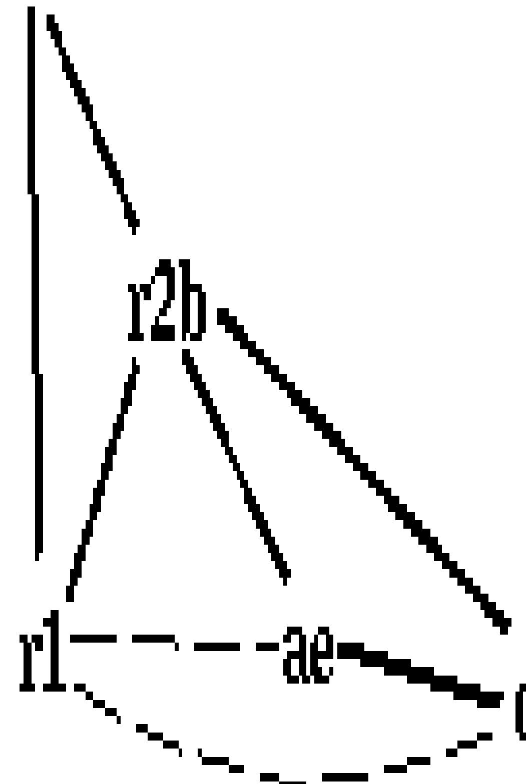


stack

Coalescing a+e; b+r2

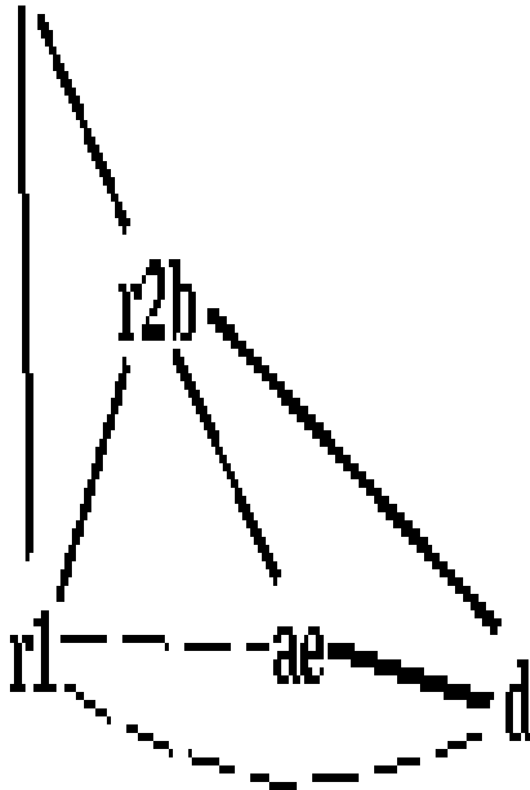


stack

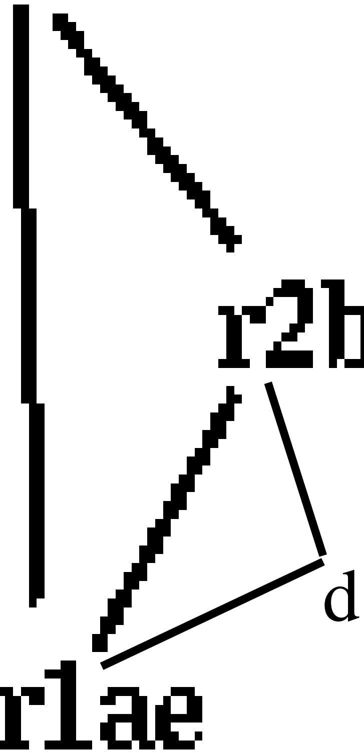


stack

Coalescing ae+r1



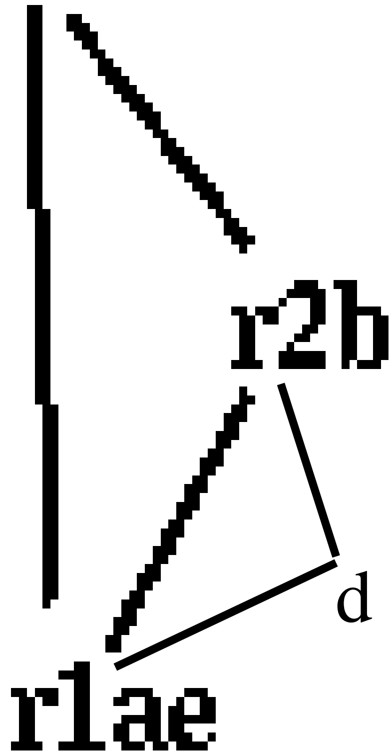
stack



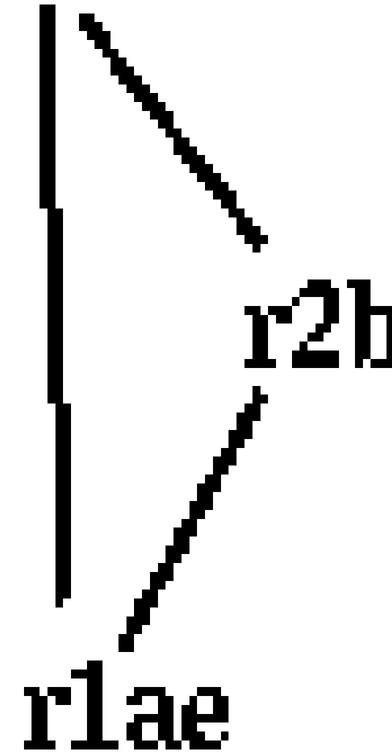
stack

r1ae and d are constrained

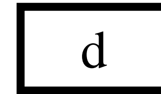
Simplify d



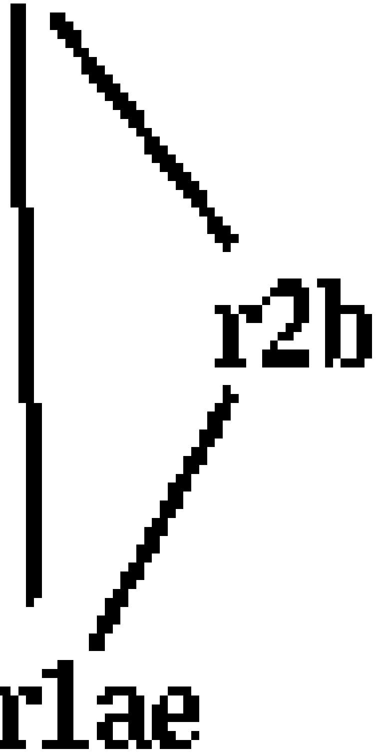
stack



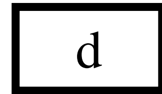
stack

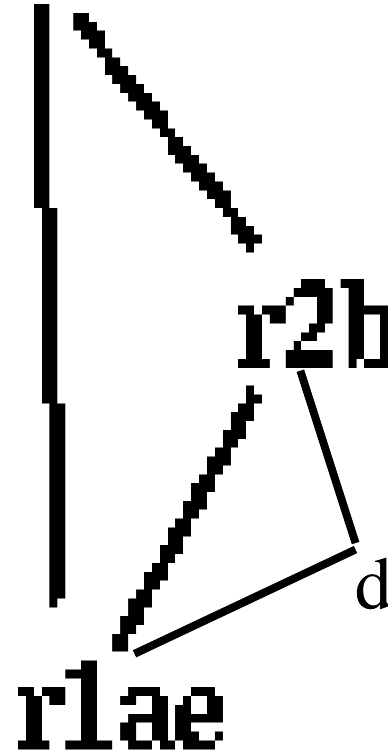


Pop d



stack





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 */
```

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 (Register Allocation)

- Two Register Allocation Methods
  - Local of every expression tree
    - Simultaneous instruction selection and register allocation
    - Reorder computations
    - Optimal (under certain conditions)
  - Global of every function
    - Applied after instruction selection
    - Performs well for machines with many registers
    - Can handle instruction level parallelism
    - More symbolic names help
      - Simplifies the coloring problem
- Missing
  - Interprocedural allocation

Generating LLVM Code

Variable Declarations

- Allocate space in the stack or data
- Use symbolic registers

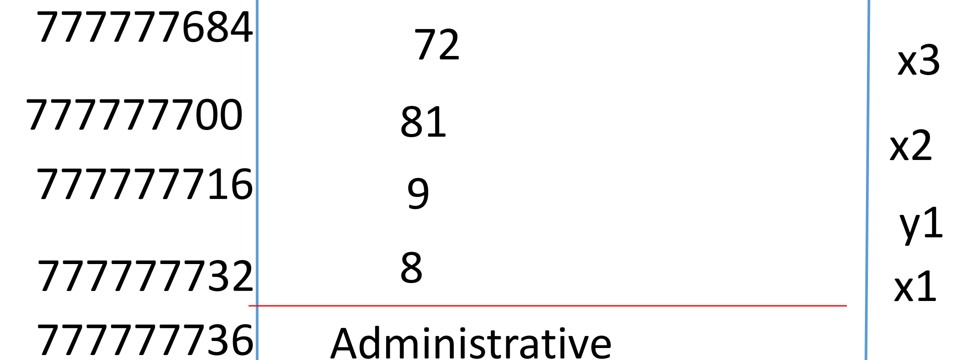
```
int foo()
{
int x;
static int y=7;
int z =5;
x = y + z;
return x;
}
```

```
@foo.y = interal global i32 7, align 4
define i32 @foo() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 5, i32* %2, align 4
  %3 = load i32, i32* @foo.y, align 4
  %4 = load i32, i32* %2, align 4
  %5 = add nsw i32 %3, %4
  store i32 %5, i32* %1, align 4
  %6 = load i32, i32* %1, align 4
  ret i32 %6
}
```

Code Blocks

- Programming languages provide code blocks

```
void foo()
{
 int x = 8 ; y=9;//1
  { int x = y * y ;//2 }
  { int x = y * 7 ;//3}
     x = y + 1;
 }
```



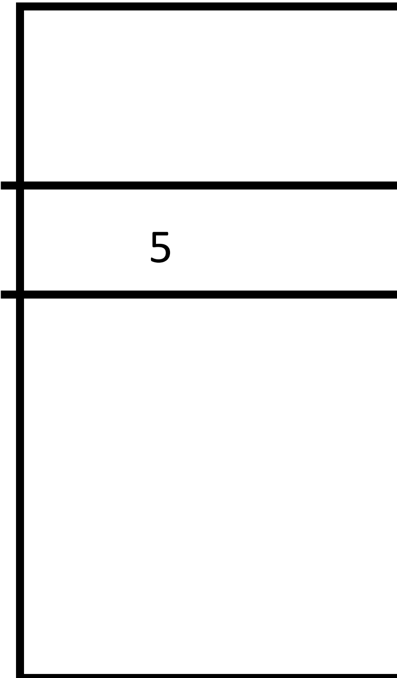
L-values vs. R-values

- Assignment x := exp is compiled into:
  - Compute the **address** of x
  - Compute the **value** of exp
  - Store the value of exp into the address of x
- Generalization
  - R-value
    - Maps program expressions into values
  - L-value
    - Maps program expressions into locations
    - Not always defined
  - Java has no small L-values

A Simple Example

```
int x = 5;

    x = x + 1;
```

Runtime memory



A Simple Example

int x = 5;

lvalue(x)=17, rvalue(x) =5

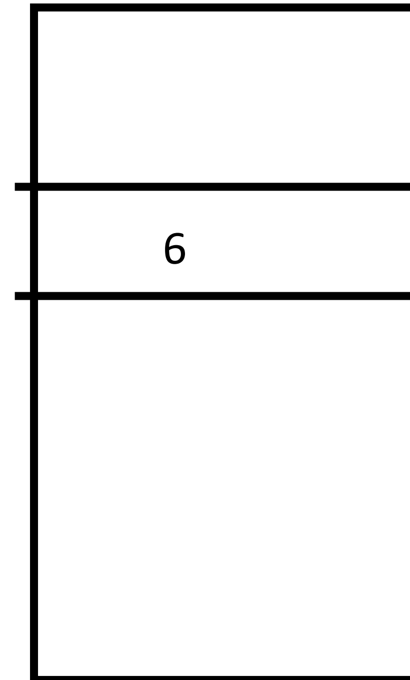
lvalue(5)=⊥, rvalue(5)=5

x = x + 1;

lvalue(x)=17, rvalue(x) =6

lvalue(5)=⊥, rvalue(5)=5

Runtime memory



Partial rules for Lvalue in C

- Type of e is pointer to T
- Type of e1 is integer
- lvalue(e2) ≠undefined

```
{ int a[100];
*(a + 5) = 8;
}
```

exp	**lvalue**	**rvalue**
id	location(id)	content(location(id))
const	undefined	value(const)
***e**	rvalue(e)	content(rvalue(e))
&e2	undefined	lvalue(e2)
e + e1	undefined	rvalue(e)+sizeof(T)*rvalue(e1)

Parameter passing

- Pass-by-reference
  - Place L-value (address) in activation record
  - Function can assign to variable that is passed
- Pass-by-value
  - Place R-value (contents) in activation record
  - Function cannot change value of caller’s variable
  - Reduces aliasing (alias: two names refer to same loc)

Prolog Code Generation

- Store L-values of automatic variables in symbolic registers
- Initialize automatic variables

```
int foo()
{
int x;
static int y=7;
int z =5;
x = y + z;
return x;
}
```

```
@foo.y = interal global i32 7, align 4
define i32 @foo() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 5, i32* %2, align 4
  %3 = load i32, i32* @foo.y, align 4
  %4 = load i32, i32* %2, align 4
  %5 = add nsw i32 %3, %4
  store i32 %5, i32* %1, align 4
  %6 = load i32, i32* %1, align 4
  ret i32 %6
}
```

Generating Code to Compute R-values

- Reclusively traverse the tree
- Load R-values of variables and constants into new symbolic register
- Store each subtree into a new symbolic registers

Pseudocode R -Value

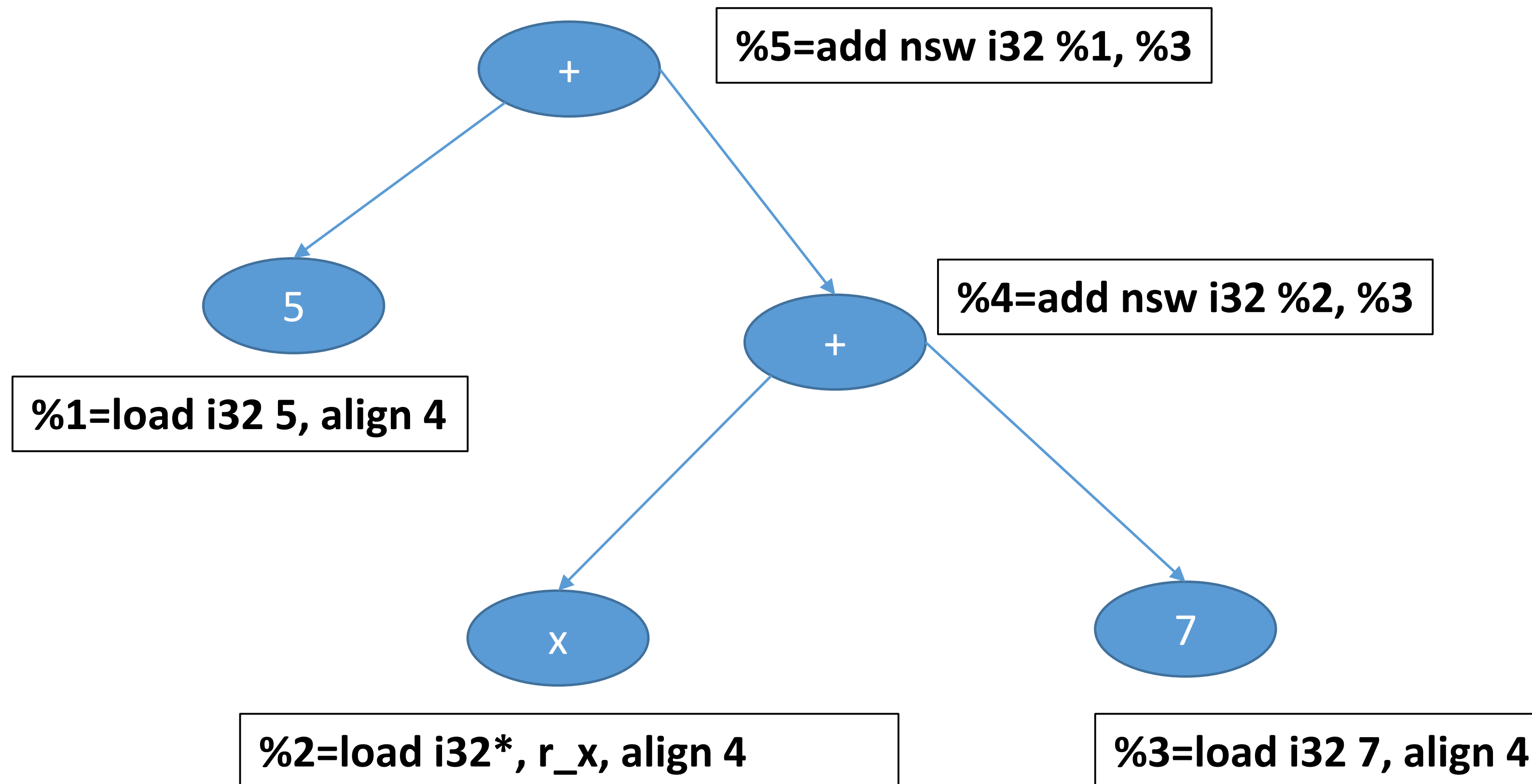
```
register rvalue(e: expression) {
new: register = newRegister()
switch e {
   case number(n: integer):
         {  emit(%new = load i32 n, align 4)      }
   case localVariable(v: symbol): {
       r: register = registerOf(v)
       emit(%new= load i32* r, align 4)       }
   case e1: expression PLUS e2: expression: {
       l: register = rvalue(e1)   // Generate code for lhs into l
       r: register  = rvalue(e2)  // Generate code for rhs into r
       emit(%new = add nsw i32 l, r)
      }
return new;
}
```

Simple Example

```
int foo()
{
int x;
static int y=7;
int z =5;
x = y + z;
return x;
}
```

```
@foo.y = interal global i32 7, align 4
define i32 @foo() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 5, i32* %2, align 4
  %3 = load i32, i32* @foo.y, align 4
  %4 = load i32, i32* %2, align 4
  %5 = add nsw i32 %3, %4
  store i32 %5, i32* %1, align 4
  %6 = load i32, i32* %1, align 4
  ret i32 %6
}
```

Example Compilation



Generating Code to Compute L-values

- Reclusively traverse the tree
- Load L-values into new symbolic register
- Store each subtree into a new symbolic registers

Pseudocode L-Value (partial)

```
register lvalue(e: expression) {
switch e: {
   case number(n: integer):
          {  exit("internal error no L-value"); }
   case localVariable(v: symbol): {
       r: register = registerOf(v)
       return r ;   }
   case deref(de: expression): {
      return ?
}
```

Pseudocode L-Value (partial)

```
register lvalue(e: expression) {
switch e: {
   case number(n: integer):
         {  exit("internal error no L-value"); }
   case localVariable(v: symbol): {
       r: register = registerOf(v)
       return r ;   }
   case deref(de: expression): {
      return rvalue(de)
    }
  ....
}
```

Assignments

- Computer L value of LHS into a register
- Compute R value of RHS into a register
- Store result

```
assign(le: expression, re: expression) {
l: register = lvalue(le);
r: register = rvalue(re);
emit(store i32 r, i32* l, align 4);
  }
```

Simple Example

```
int foo()
{
int x;
static int y=7;
int z =5;
x = y + z;
return x;
}
```

rvalue(x+y)

```
@foo.y = interal global i32 7, align 4
define i32 @foo() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 5, i32* %2, align 4
  %3 = load i32, i32* @foo.y, align 4
  %4 = load i32, i32* %2, align 4
  %5 = add nsw i32 %3, %4
  store i32 %5, i32* %1, align 4
  %6 = load i32, i32* %1, align 4
  ret i32 %6
}
```

Code Generation for Control Flow

Chapter 6.4

Motivating Example

```
void foo(int x) {
  if (!(!(x >7) || (x <=9))) {
        x = 5;
  }
}
```

```
define void @foo(i32) #0 {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  %3 = load i32, i32* %2, align 4
  %4 = icmp sgt i32 %3, 7
  br i1 %4, label %5, label %9
; <label>:5:                              ; preds = %1
  %6 = load i32, i32* %2, align 4
  %7 = icmp sle i32 %6, 9
  br i1 %7, label %9, label %8
; <label>:8:                              ; preds = %5
  store i32 5, i32* %2, align 4
  br label %9
; <label>:9:                              ; preds = %8, %5,
%1
  ret void
}
```

Boolean Expressions

- In principle behave like arithmetic expressions
- But are treated specially
  - Different machine instructions
  - Used in control flow instructions
  - Shortcut computations
  - Negations can be performed at compile-time

if (a < b) goto l

Code for a < b yielding a condition value

Conversion condition value into Boolean

Conversion from Boolean in condition value

Jump to l on condition value

Location vs. Value Computation

- Option 1: The value of expression is stored in a designated location or register
- Option 2: If e=N to v then the code will reach a location l
  - If the value of e is true then the program will reach lt
  - If the value of e is true then the program will reach lf
  - used for Booleans

Shortcut computations

- Languages such as C define shortcut computation rules for Boolean
- Incorrect translation of e1 && e2

Code to compute e1 in loc1
Code to compute e2 in loc2
Code for && operator on loc1 and loc2

Location Computation

- The result of a Boolean expression is pair of locations in the generated code
  - The true location corresponds to the target instruction when the condition holds
  - The false location corresponds to the target instruction when the condition does not hold

Code for e1 && e2

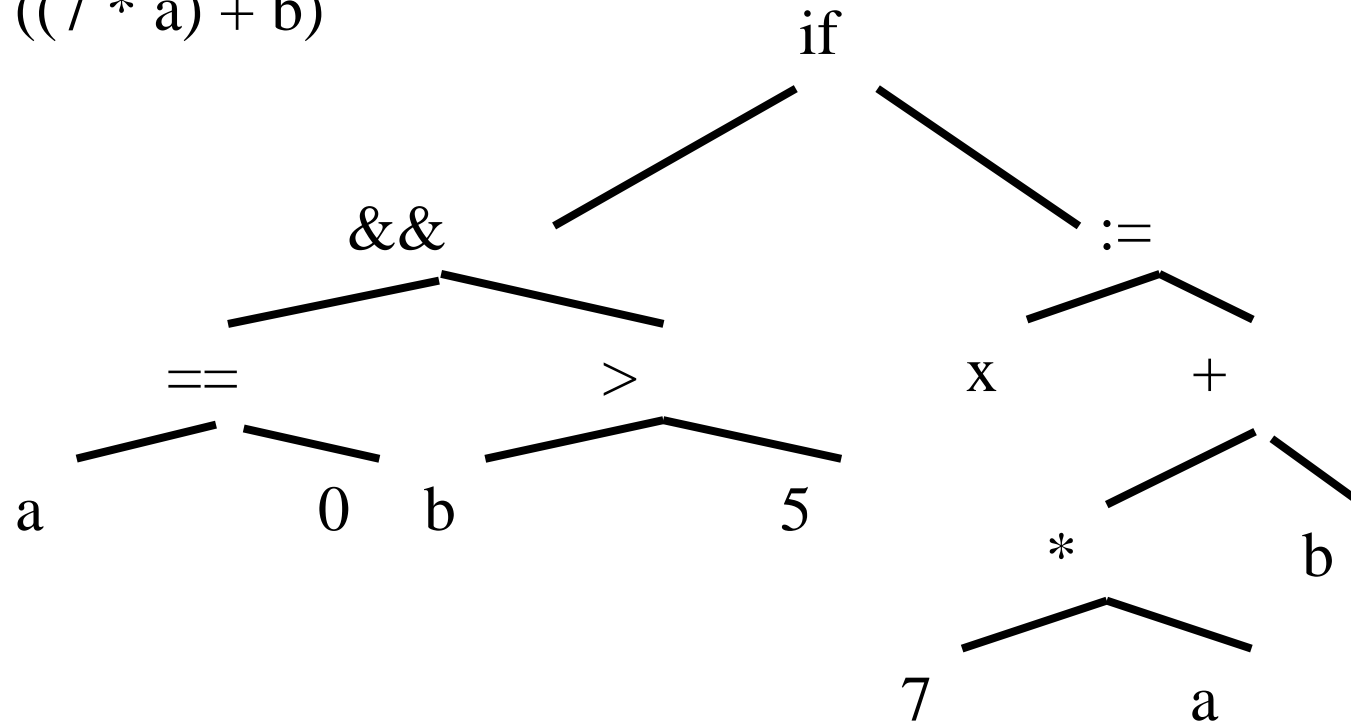
```
        Code for e1 into r1

        if r1 then goto L11        true-e1
        br L12                     false-e1
L11:
        Code for e2 into r2

        if r2 then goto L21        true-e2
        br L22                     false-e2
L21:                               true-e1&&e2

L12:
L22:                               false-e1&&e2
```

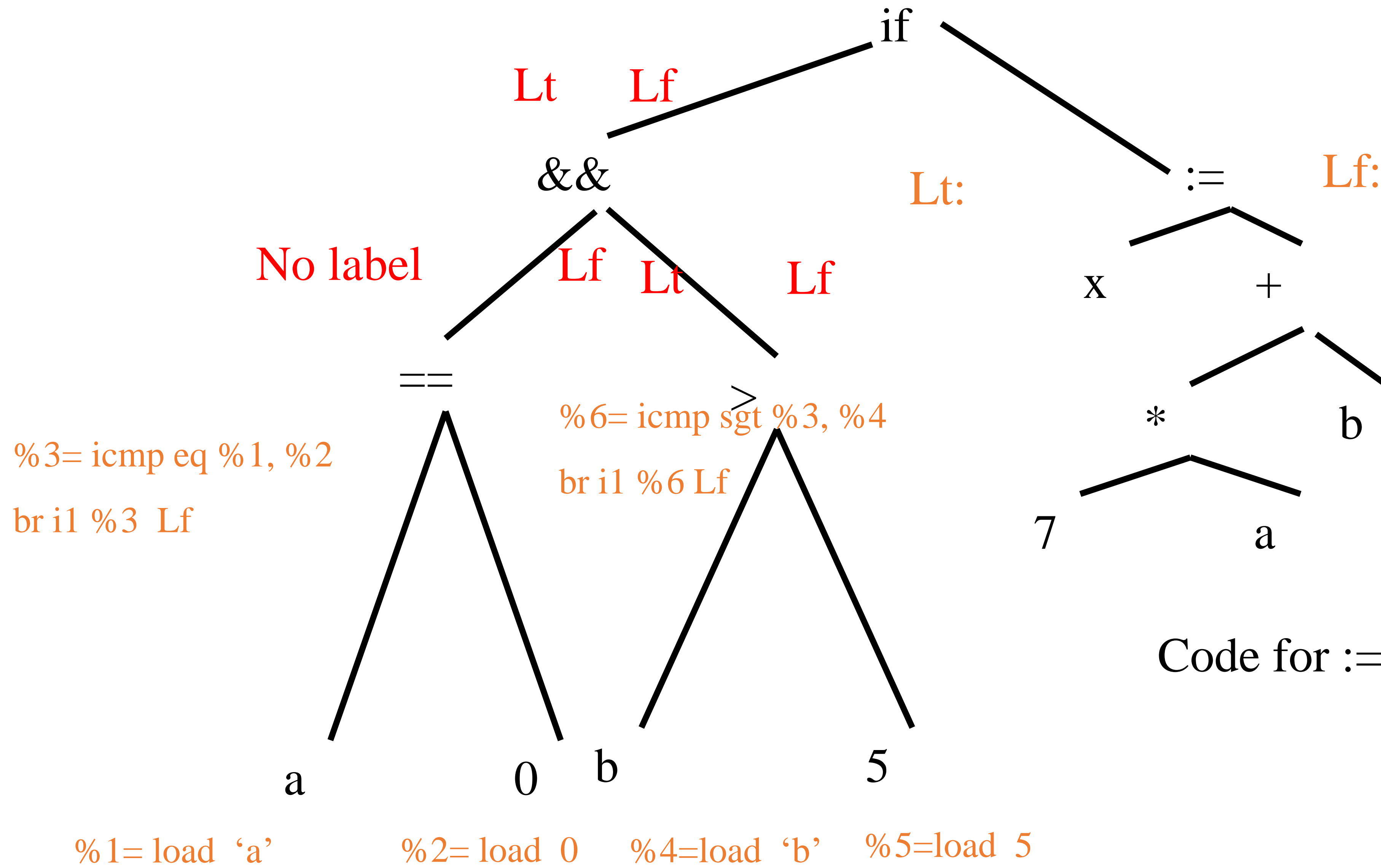
Code for Booleans (Location Computation)

- Top-Down tree traversal
- Generate code sequences instructions
- Jump to a designated ‘true’ label when the Boolean expression evaluates to 1
- Jump to a designated ‘false’ label when the Boolean expression evaluates to 0
- The true and the false labels are passed as parameters

Example

```
if ((a==0) && (b > 5))
    x = ((7 * a) + b)
```




if
Lt
Lf
&&
Lt:
:=
Lf:
No label
Lf
Lt
Lf
x
+
==
>
%6= icmp sgt %3, %4
br i1 %6 Lf
*
b
%3= icmp eq %1, %2
br i1 %3 Lf
7
a
Code for :=
a
0
b
5
%1= load 'a'
%2= load 0
%4=load 'b'
%5=load 5

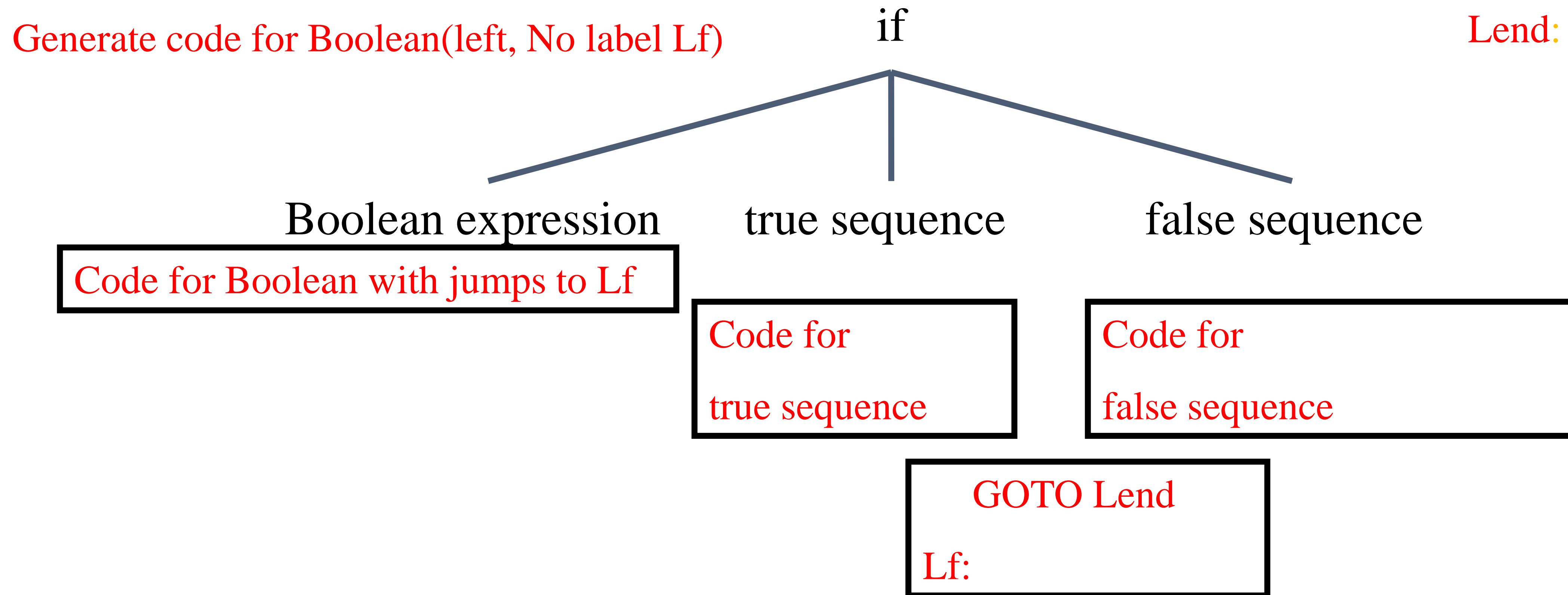
Location Computation for Booleans

```
void location_value(e: expression, t: label, f: label) {
switch e {
     case e1: expression GT e2: expression: {
      l: register = rvalue(e1)   // Generate code for lhs into l
      r: register  = rvalue(e2)  // Generate code for rhs into r
      if (t != no-label)  {
              n: register = newRegister()
              emit(n= icmp gt i32 l, r); emit(br   i1, n, t)
               if (f != no-label) {
                 emit(br f)
               }
         }
        else if (f != no-label) {
               emit(icmp le i32 l, r, t)
             }
     }  // similar for other relational operators
    case e1:expression LazyAnd e2:expression: { }
    case  e1:expression LazyOr e2: expression: {}
    case not e1: expression: {}
```

Location Computation for Booleans(2)

```
void location_value(e: expression, t: label, f: label) {
switch e {
   …
    case e1:expression LazyAnd e2:expression: {
            locationValue(e1, no-label, f)
            locationValue(e2, t, f)
             }
    case  e1:expression LazyOr e2: expression: {
             locationValue(e1, t, no-label)
             locationValue(e2, t, f)
              }
    case not e1: expression:
            locationValue(e1, f, t)
```

Code generation for IF



Code generation for IF (no-else)

Allocate new label Lend

Generate code for Boolean(left, no-label, Lend)

if

Boolean expression

true sequence

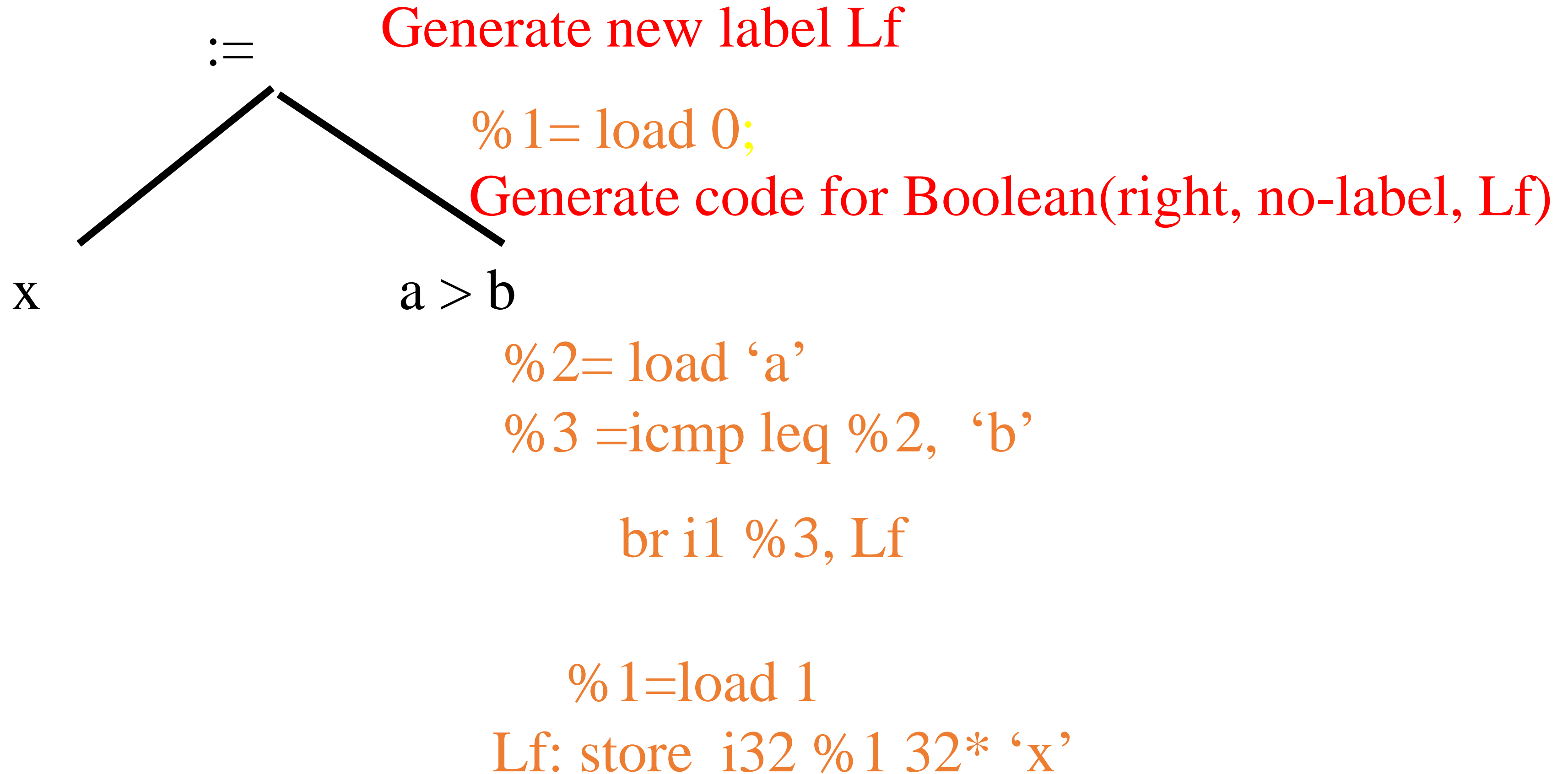
Lend:

Code for Boolean with jumps to Lend

Code for

true sequence

Coercions into value computations



Effects on performance

- Number of executed instructions
- Unconditional vs. conditional branches
- Instruction cache
- Branch prediction
- Target look-ahead

Code for case statements

- Three possibilities
  - Sequence of IFs
    - O(n) comparisons
  - Jump table
    - O(1) comparisons
  - Balanced binary tree
    - O(log n) comparisons
- Performance depends on n
- Need to handle runtime errors

Simple Translation

```
    tmp_case_value := case expression;
    IF tmp_case_value = l1 THEN GOTO label_1;
    IF tmp_case_value = l2 THEN GOTO label_2;
    …
    IF tmp_case_value = ln THEN GOTO label_n;
    GOTO label_else;  // or insert the code at label else
label 1:
  Code for statement sequence1
  GOTO label_next;
label 2:
  Code for statement sequence2
  GOTO label_next;
  …
label n:
  Code for statement sequencen
  GOTO label_next;
label else:
  Code for else-statement sequence
```

Balanced trees

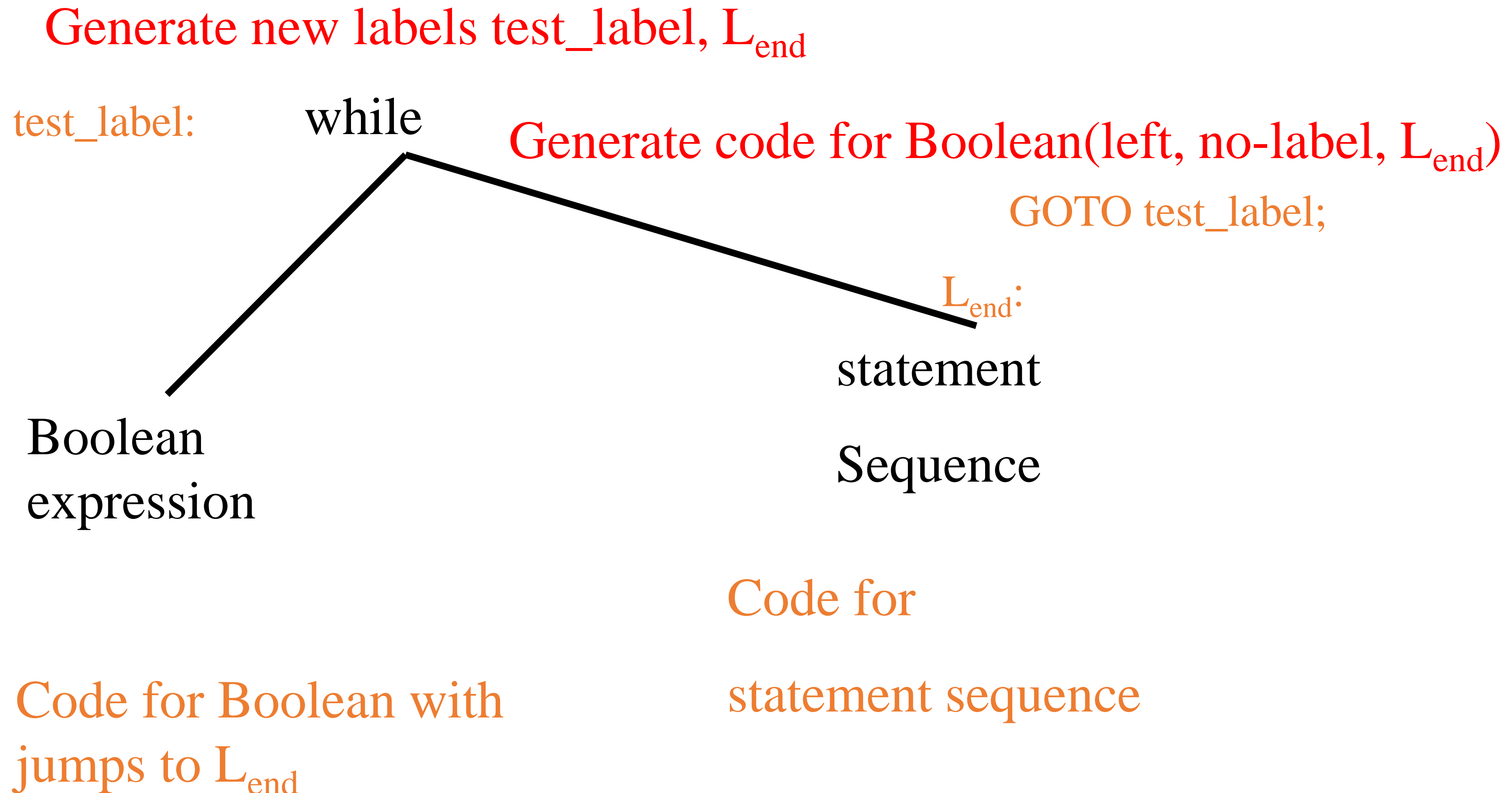
- The jump table may be inefficient
  - Space consumption
  - Cache performance
- Organize the case labels in a balanced tree
  - Left subtrees smaller labels
  - Right subtrees larger labels
- Code generated for node_k

```
label_k: IF tmp_case_value < l_k THEN
             GOTO label of left branch ;
         IF tmp_case_value >l_k THEN
             GOTO label of right branch;
         code for statement sequence;
         GOTO label_next;
```

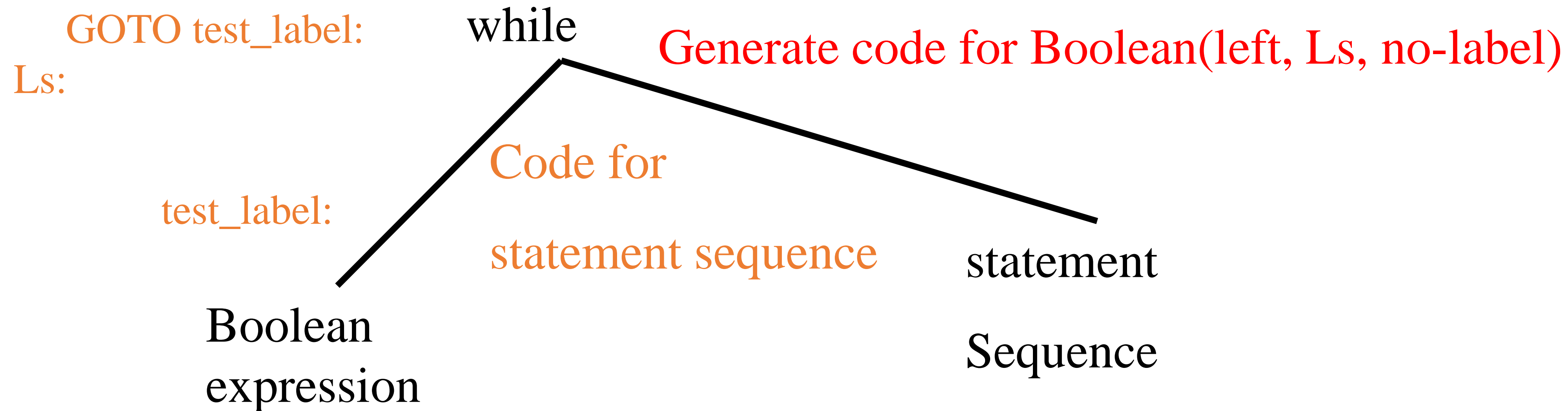
Repetition Statements (loops)

- Similar to IFs
- Preserve language semantics
- Performance can be affected by different instruction orderings
- Some work can be shifted to compile-time
  - Loop invariant
  - Strength reduction
  - Loop unrolling

while statements



while statements(2)



Simple-minded translation

```
FOR i in lower bound .. upper bound DO
    statement sequence

END for
```

```
i := lower_bound;

tmp_ub := upper_bound;

WHILE i <= tmp_ub DO
    code for statement sequence

    i := i + 1;
END WHILE
```

Correct Translation

```
FOR i in lower bound .. upper bound DO
    statement sequence

END for
```

```
        i := lower_bound;

        tmp_ub := upper_bound;
        IF i >tmp_ub THEN GOTO end_label;

loop_label:
    code for statement sequence
    if (i==tmp_ub) GOTO end_label;

    i := i + 1;

   GOTO loop_label;

end_label:
```

Tricky question

```
for (exp1; exp2; exp3) {
    body;
  }
```

```
exp1;
while (exp2) {
    body;
    exp3;
  }
```

Summary

- Handling control flow statements is usually simple
- Complicated aspects
  - Routine invocation
  - Non local gotos
- Runtime profiling can help

Summary Code Generation

- Preserve the semantics with local transformations into intermediate language
  - Perform computations at compile-time
  - Much more can be done
- Every subtree is converted into an equivalent instruction sequences
- Uses unbounded registers and labels to simplify matters