Code Optimization

Optimization

- Process of transforming a piece of code to make more efficient without changing output or side effects
  - Only difference visible to the user should be that code runs faster and/or consumes less memory
- Name implies you are finding an "optimal solution"
  - Misnomer! optimization aims to improve, not perfect, the result

Optimization

- Field where most compiler research is done today
  - tasks of the front-end (scanning, parsing, semantic analysis) are well understood
  - unoptimized code generation is relatively straightforward
- High-quality optimization is more of an craft than a science
- Compilers for mature languages judged by the quality of the object code they produce

Optimization

- Many optimization problems are NP-complete
- Most optimization algorithms rely on heuristics and approximations
- However, optimization algorithms tend to do rather well overall

Efficient Code

- Efficient code starts with intelligent decisions by the programmer!
  - No one expects a compiler to replace BubbleSort with Quicksort
  - If a programmer uses a lousy algorithm, no amount of optimization can make it zippy
- In terms of *big-O*, a compiler can only make improvements to constant factors
  - all else being equal, you want an algorithm with low constant factors

Optimization

- You shouldn't try to optimize the way a compiler does
  - Consider code to walk through an array and set every element to one

```
int arr[10000];

void Binky() {
        int i;
        for (i=0; i < 10000; i++)
        arr[i] = 1;
}
```

```
int arr[10000];

void Winky() {
        register int *p;
        for (p = arr; p < arr + 10000;
        p++)
        *p = 1;
}
```

Which one is faster?

Optimization

- The second one of course!
  - But many modern compilers emit the same object code for both
  - Use of clever techniques
    - Here, loop-induction variable elimination

Moral

Write code that is easy to understand and let the compiler do the optimization

Stating the Obvious

- Optimization should not change the correctness of the generated code
  - Transforming the code to something that runs faster but incorrectly is of little value
  - Expected that the unoptimized and optimized variants give the same output for all inputs
- N.B.: May not hold for an incorrectly written program (e.g., one that uses an uninitialized variable)

When to Optimize

- Some techniques applied to intermediate code
  - Streamline, rearrange, compress, etc.
  - Try to reduce the size of the abstract syntax tree or shrink the number of instructions
- Others applied as part of final code generation
  - Choosing which instructions to emit, how to allocate registers etc.
- Still others occur after final code generation
  - Attempt to re-work code itself into something more efficient

Optimization

- Can be very complex and time-consuming
- Often involves multiple sub-phases, some applied more than once
- Most compilers allow optimization to be turned off to speed up compilation
  - gcc has specific flags to turn on and off individual optimizations

Basic Blocks

- Better code generation requires information about points of definition and points of use of variables
- Value of a variable can depend on multiple points in the program

```
y := 12;
x := y * 2;     -- here x = 24
Label1: ...
x := y * 2;     -- 24?   Can't tell, y may be different
```

Basic Blocks

- Segment of code that a program must enter at the beginning and exit only at the end
- Only the first statement can be reached from outside the block
  - no branches into the middle of the block
- All statements are executed consecutively after the first one
  - no branches or halts until the exit
- Basic block has exactly one entry point and one exit point

If a program executes the first instruction in a basic block, it must execute every instruction following it in the block sequentially

Basic Blocks

- Can begin with:
  - entry point into the function
  - target of a branch
  - instruction immediately following a branch or a return
- Can end with:
  - jump statement
  - conditional or unconditional branch
  - return statement

Example

Pascal

```
function fib(n:integer) : result integer;
        var j, k : integer;
begin
        if n = 1 then
                fib := 1
        else if n = 2 then
                fib := 1
        else
                fib := fib(n-1) + fib(n-2)
end;
```

TVI Code

```
param  @_0
PROCBEGIN       Fib
        alloc   2
        beq     ^%0, 1, 2
        beq     ^%0, 2, 2
        sub     ^%0, 1, %1
        param   @%1
        call    Fib, 1
        sub     ^%0, 2, %2
        param   @%2
        call    Fib, 1
        add     %1, %2, ^%0
        goto    3
2:      move    1, ^%0
3:      free    2
PROCEND
```

Basic Blocks in TVI Code

```
        PROCBEGIN       Fib
        alloc   2
        beq     ^%0, 1, 2
```

```
        beq     ^%0, 2, 2
```

```
        sub     ^%0, 1, %1
        param   @%1
        call    Fib, 1
```

```
        sub     ^%0, 2, %2
        param   @%2
        call    Fib, 1
```

```
        add     %1, %2, ^%0
        goto    3
```

```
2:      move    1, ^%0
```

```
3:      free    2
        PROCEND
```

Note: called with `param @_0`

Control Flow Analysis

- Up to now, compiler has done an enormous amount of analysis (lexical, syntactic, semantic), but still doesn't really know how the program does what it does
- Control-flow analysis
  - compiler figures out more information about how the program does its work
  - can assume that there are no syntactic or semantic errors in the code

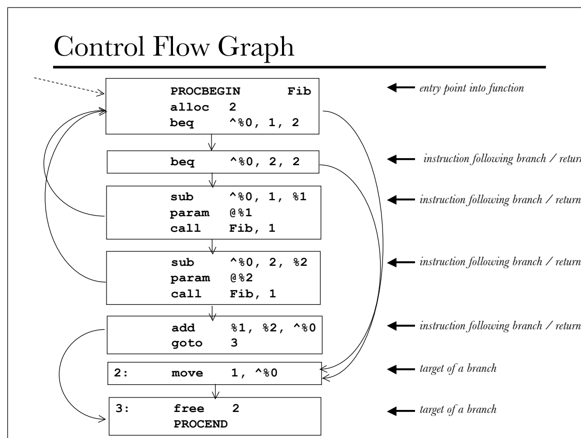
Control Flow Analysis

- Begins by constructing a control-flow graph
  - Graph of different possible paths program flow could take through a function
- First divide the code into basic blocks

Control Flow Graph

- Graph between basic blocks
  - Each basic block is a node in the graph
  - Possible different routes a program might take are edges
    - if a block ends with a branch, there is a path leading from that block to the branch target
- Blocks that can follow a block are called its successors
  - A block may have one or multiple successors
- Block may have many, one, or no predecessors

Control Flow Graph



Unreachable Code

- Look familiar?
  - gcc warning or javac error :

`Unreachable code at line XXX.`

How does the compiler know when code is unreachable?

Local Optimizations

- Optimizations performed exclusively *within a basic block*
  - typically easiest to perform since do not consider any control flow information
  - work only with statements within the block
- Many local optimizations have corresponding global optimizations
  - operate on the same principle, but require additional analysis to perform

Local Optimizations

- Evaluation at compile-time of expressions whose operands are known to be constant
  - Simplest form:
    1. determine that all operands in an expression are constant-valued
    2. perform the evaluation of the expression at compile-time
    3. replace the expression by its value

Examples

- Expression `10+2*3`
  - compiler can compute result at compile-time, emit code as if the input contained the result rather than the original expression
- Conditional branch `if a < b goto L1 else goto L2`
  - if *a* and *b* are constant, replace with `Goto L1` or `Goto L2` (depending on the truth of the expression evaluated at compile-time)

Constant Folding

- The constant expression has to be evaluated at least once
  - But if compiler does it, don't have to do it again (possibly repeatedly) at runtime
- Caution:
  - Must obey grammar and semantic rules from the source language that apply to expression evaluation
    - may not necessarily match the language compiler is written in
  - Should respect expected treatment of exceptional conditions (divide by zero, over/underflow)

Example

```
a = 10 * 5 + 6 - b;
```

unoptimized

```
tmp0 = 10
tmp1 = 5
tmp2 = tmp0 * tmp1
tmp3 = 6
tmp4 = tmp2 + tmp3
tmp5 = tmp4 - b
a    = tmp5
```

optimized

```
tmp0 = 56
tmp1 = tmp0 - b
a    = tmp1
```

Constant Folding

- Allows a language to accept constant expressions where a constant is required
- E.g., in C:
  - array size

    ```
    int arr[20 * 4 + 3];
    ```
  - case label

    ```
    switch (i) {  case 10 * 5: ...  }
    ```
- Expression can be resolved to an integer constant at compile time
- If expression involves a variable, **error!**

Constant propagation

- If a variable is assigned a constant value, subsequent uses can be replaced by the constant *as long as no intervening assignment has changed the value of the variable*
- Example

```
tmp4  = 0
f0    = tmp4
tmp5  = 1
f1    = tmp5
tmp6  = 2
i     = tmp6
```

➡

```
f0 = 0
f1 = 1
i  = 2
```

Algebraic Simplification and Reassociation

- Simplification
  - Use algebraic properties or particular operator-operand combinations to simplify expressions
- Reassociation
  - Use properties such as associativity, commutativity and distributivity to rearrange an expression to enable other optimizations (e.g. constant-folding)
- Most obvious are optimizations that can remove useless instructions entirely via algebraic identities
- Rules of arithmetic can come in handy when looking for redundant calculations to eliminate

Examples

Simplification using algebraic rules

```
x+0          = x
0+x          = x
x*1          = x
1*x          = x
0/x          = 0
x-0          = x
b && true    = b
b && false   = false
b || true    = true
b || false   = b
```

Algebraic rearrangement to restructure an expression to enable constant-folding, common sub-expression elimination, etc.

```
b = 5 + a + 10 ;
```

➡

```
tmp0 = 5
tmp1 = tmp0 + a
tmp2 = tmp1 + 10
b    = tmp2
```

➡

```
tmp0 = 15
tmp1 = a + tmp0
b    = tmp1
```

Operator strength reduction

- Replaces an operator by a "less expensive" one
- Often performed as part of loop-induction variable elimination

```
while (i < 100) {
   arr[i] = 0;
   i = i + 1;
}
```

Each time through the loop, multiply i by 4 (element size) and add to the array base

Unoptimized

```
L0: tmp2 = i < 100
    If not tmp2 Goto L1
    tmp4 = 4 * i
    tmp5 = arr + tmp4
    *(tmp5) =  0
    i = i + 1
L1:
```

Could instead maintain the address to the current element and add 4 each time

Optimized

```
     tmp4 = arr
L0:  tmp2 = i < 100
     If not tmp2 Goto L1
     *tmp4 =  0
     tmp4 = tmp4 + 4
     i = i + 1
L1:
```

Copy propagation

- Similar to constant propagation, but generalized to non-constant values
  - E.g., for assignment **`a = b`**, replace later occurrences of **`a`** with **`b`** (assuming there are no changes to either variable in between)
- Particularly valuable because eliminates a large number of instructions that only serve to copy values from one variable to another

```
tmp2 = tmp1
tmp3 = tmp2 * tmp1
tmp4 = tmp3
tmp5 = tmp3 * tmp2
c    = tmp5 + tmp4
```

```
tmp3 = tmp1 * tmp1
tmp5 = tmp3 * tmp1
c    = tmp5 + tmp3
```

Copy propagation

- Can also drive copy propagation "backwards"'
  - Enables recognizing that original assignment made to a temporary can be eliminated in favor of direct assignment to the final goal

```
tmp1 = Call Binky
a    = tmp1
tmp2 = Call Winky
b    = tmp2
tmp3 = a * b
c    = tmp3
```

```
a = Call Binky
b = Call Winky
c = a * b
```

Dead Code Elimination

- If an instruction's result is never used, the instruction is considered dead
  - can be removed from the instruction stream
- E.g., for **`tmp1 = tmp2 + tmp3`**
  - If **`tmp1`** never used again, eliminate this instruction altogether
  - Have to be careful about making assumptions, e.g., if **`tmp1`** holds the result of a function call
    - **`tmp1 = Call Binky`**
    - Even if **`tmp1`** never used again, cannot eliminate the instruction because can't be sure that the called function has no side-effects

Dead code can occur in the original source program but is more likely to have resulted from some of the optimization techniques run previously

Common sub-expression elimination

- Two operations are common if they produce the same result
  - more efficient to compute the result once and reference it the second time rather than re-evaluate
- An expression is alive if the operands used to compute the expression have not been changed
- An expression that is no longer alive is dead

Example

Source code

```
main() {
   int x, y, z;
   x = (1+20)* -x;
   y = x*x+(x/y);
   y = z = (x/y)/(x*x);
}
```

Straight translation

```
tmp1 = 1 + 20
tmp2 = -x
x    = tmp1 * tmp2
tmp3 = x * x
tmp4 = x / y
y    = tmp3 + tmp4
tmp5 = x / y
tmp6 = x * x
z    = tmp5 / tmp6
y    = z
```

Optimized version, after constant folding and propagation and elimination of common sub-expressions

```
tmp2 = -x
x    = 21 * tmp2
tmp3 = x * x
tmp4 = x / y
y    = tmp3 + tmp4
tmp5 = x / y
z    = tmp5 / tmp3
y    = z
```

Common Sub-Expressions

- Use directed acyclic graph (DAG) to recognize common sub-expressions and remove redundant quadruples
- Intermediate code optimization:
  - basic block => DAG => improved block => assembly
- Leaves are labeled with identifiers and constants
- Internal nodes are labeled with operators and identifiers

DAG construction

- Forward pass over basic block
- For x := y *op* z;
  - Find node labeled y, or create one
  - Find node labeled z, or create one
  - Create new node for *op*, or find an existing one with descendants y, z
  - Add x to list of labels for new node
  - Remove label x from node on which it appeared
- For x := y;
  - Add x to list of labels of node which currently holds y

Example

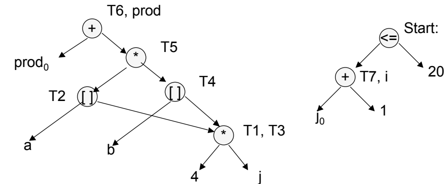
```
prod := 0;
for j in 1..20 loop
  prod := prod + a(j) * b(j);            assume 4-byte integer
end loop;
```

Quadruples:

```
             prod   := 0                   basic  block leader
             J      := 1
      start: T1     := 4 * j               basic block leader
             T2     := a (T1)
             T3     := 4 * j               redundant
             T4     := b (T3)
             T5     := T2 * T4
             T6     := prod + T5
             prod   := T6
             T7     := j + 1
             j      := T7
             If j <= 20 goto start:
```

DAG for body of loop

- Common sub-expression identified



From DAG to improved block

- A node without a label is a dead value
- Choose the label of a live variable over a temporary

```
start:   T1    := 4 * j;
         T2    := a [ T1]
         T4    := b [ T1]
         T5    := T2 * T4
         prod  := prod + T5
         J     := J =1
         If j <=20 goto start
```

Fewer quadruples, fewer temporaries

Global Optimizations

- So far considered making changes within one basic block
- With additional analysis, can apply similar optimizations across basic blocks, making them global optimizations
  - "global" in this case does not mean across the entire program
  - usually only optimize one function at a time
- Inter-procedural analysis is an even larger task, not even attempted by some compilers

Code Motion

- Also called **code hoisting**
- Unifies sequences of code common to one or more basic blocks to reduce code size and avoid expensive re-evaluation
- Most common form of code motion is loop-invariant code motion
  - Moves statements that evaluate to the same value every iteration of the loop to somewhere outside the loop

Loops

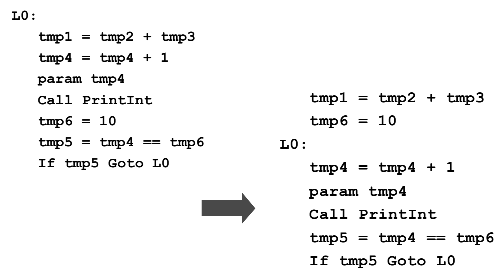
Loop:
A set of basic blocks that satisfies the following:

1. All are strongly connected
   - i.e. there is a path between any two blocks
2. The set has a unique entry point
   - i.e. every path from outside the loop that reaches any block inside the loop enters through a single node

- A block *n* dominates *m* if all paths from the starting block to *m* must travel through *n*. Every block dominates itself.
- For loop L, moving invariant statement s in block B which defines variable v outside the loop is a safe optimization if:
  1. B dominates all exits from L
  2. No other statement assigns a value to *v*
  3. All uses of *v* inside L are from the definition in *s*
- Loop invariant code can be moved to just above the entry point to the loop.

Example

```
L0:
   tmp1 = tmp2 + tmp3
   tmp4 = tmp4 + 1
   param tmp4
   Call PrintInt
   tmp6 = 10
   tmp5 = tmp4 == tmp6
   If tmp5 Goto L0
```



```
   tmp1 = tmp2 + tmp3
   tmp6 = 10
L0:
   tmp4 = tmp4 + 1
   param tmp4
   Call PrintInt
   tmp5 = tmp4 == tmp6
   If tmp5 Goto L0
```

Machine Optimizations

- Code generation provides an opportunity for cleverness in generating efficient target code
- Specific machines features (specialized instructions, hardware pipeline abilities, register details) are taken into account to produce code optimized for a particular architecture

Register allocation

- Perhaps the single most effective optimization for all architectures
- Registers
  - Fastest kind of memory available
  - But as a resource can be scarce
- **Problem**
  - How to minimize traffic between the registers and what lies beyond them in the memory hierarchy to eliminate time wasted sending data back and forth across the bus and the different levels of caches

Register Coloring

- Consider which variables are more heavily in demand
- Keep in registers and spill those that are no longer needed or won't be needed until much later
- **Register coloring**
  - A graph-coloring approach
  - E.g., If have 8 registers, try to color a graph with eight different colors
  - Graph's nodes are made of webs and the arcs are determined by calculating interference between the webs
    - web represents a variable's definitions--places where it is assigned a value (e.g., `x =` ....)--and the possible different uses of those definitions (e.g., `y = x + 2`)

Register Coloring

- Web can be represented as another graph
  - Definition and uses of a variable are nodes
  - If a definition reaches a use, there is an arc between the two nodes
  - If two portions of a variable's definition-use graph are unconnected, then there are two separate webs for a variable
- **Interference graph algorithm**
  - Each node is a web
  - Determine which webs don't interfere with one another
  - If not, can use the same register for those two variables.

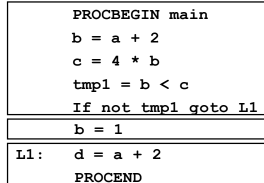
Example

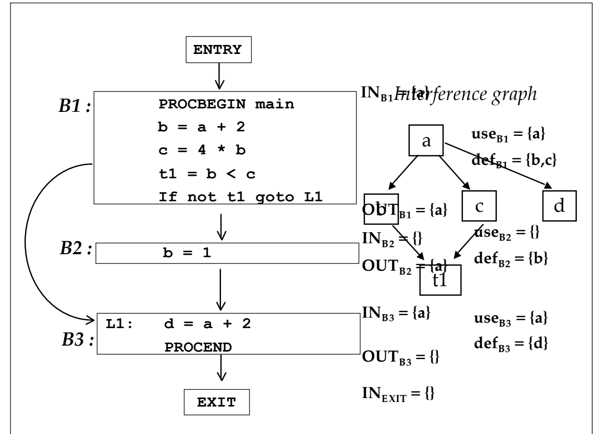
```
i = 10
j = 20
x = i + j
y = j + k
```

- *i* interferes with *j* because at least one pair of *i*'s definitions and uses is separated by a definition or use of *j*
- *i* and *j* are alive at the same time
  - A variable is alive between the time it has been defined and that definition's last use, after which the variable is dead
- If two variables interfere, cannot use the same register for each
- Otherwise, can use same register since there is no overlap in the liveness

Example

Basic Blocks





R-coloring

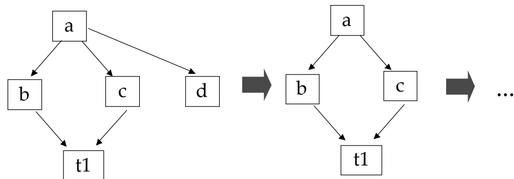
- Once interference graph constructed, r-color it so that no two adjacent nodes share the same color
  - *r* is the number of registers
  - each color represents a different register
- Graph-coloring is NP-complete
  - employ a heuristic rather than an optimal algorithm

Simplified Version of a Heuristic

1. Find the node with the least neighbors (Break ties arbitrarily)
2. Remove it from the interference graph and push it onto a stack
3. Repeat steps 1 and 2 until the graph is empty
4. Rebuild the graph as follows:
   a) Take the top node off the stack and reinsert it into the graph
   b) Choose a color for it based on the color of any of its neighbors presently in the graph, rotating colors in case there is more than one choice
   c) Repeat a and b until the graph is either completely rebuilt, or there is no color available to color the node

- If stuck, graph may not be r-colorable
- May have to spill a variable to memory

Coloring

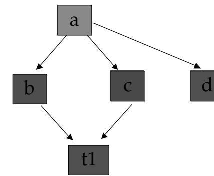


Order of removal : d, b, c, t1, a

Assume 4 colors are available: assign colors in reverse order, constrained by already colored nodes.

a (no constraint) t1 c (a, t1) b (a, t1) d (a)

Coloring



Variables can be assigned to registers corresponding to their colors

d and t1 can use the same register, the rest cannot

Peephole optimizations

- A pass that operates on the target assembly
  - Only considers a few instructions at a time (through a "peephole")
  - Attempts to do simple, machine-dependent code improvements
- Examples
  - Elimination of multiplication by 1
  - Elimination of load of a value into a register when the previous instruction stored that value from the register to a memory location
  - Replacing a sequence of instructions by a single instruction with the same effect

Because of its myopic view, a peephole optimizer does not have the potential payoff of a full-scale optimizer, but can significantly improve code at a very local level and can be useful for cleaning up the final code that resulted from more complex optimizations

Peephole Optimization

- Much of the work done in peephole optimization can be though of as find-replace activity
  - Look for idiomatic patterns in a single or sequence of two to three instructions that can be replaced by more efficient alternatives
- Example:
  - MIPS has instructions that can add a small integer constant to the value in a register without loading the constant into a register first

```
li $t0, 10
lw $t1, -8($fp)
add $t2, $t1, $t0
sw $t1, -8($fp)
```

➡

```
lw $t1, -8($fp)
addi $t2, $t1, 10
sw $t1, -8($fp)
```

Optimization soup

- You might wonder about the interactions between the various optimization techniques
  - Some transformations may expose possibilities for others, or one optimization may obscure or remove possibilities for others
    - Algebraic rearrangement may allow for common subexpression elimination or code motion
    - Constant folding usually paves the way for constant propagation and then it turns out to be useful to run another round constant-folding, etc.
- **How do you know you are done?**

You don't!

Moral

"Adding optimizations to a compiler is a lot like eating chicken soup when you have a cold. Having a bowl full never hurts, but who knows if it really helps. If the optimizations are structured modularly so that the addition of one does not increase compiler complexity, the temptation to fold in another is hard to resist. How well the techniques work together or against each other is hard to determine."

(Pyster)

Appendix

Beyond Basic Blocks: Data Flow Analysis

- Basic blocks are nodes in the flow graph
- Can compute global properties of program as iterative algorithms on graph:
  - Constant folding
  - Common subexpression elimination
  - Live-dead analysis
  - Loop invariant computations
- Requires complex data structures and algorithms

Data-flow Analysis

- Additional analysis the optimizer must do to perform optimizations across basic blocks
- More complicated than control-flow analysis
- Example: global common sub-expression elimination
  - Careful analysis across blocks can determine whether an expression is alive on entry to a block
    - Said to be **available** at that point
    - Once set of available expressions is known, common sub-expressions can be eliminated on a global basis

Data-flow Analysis

- Each block is a node in the flow graph of a program
- **Successor set** (*succ*(*x*))
  - set of all nodes that *x* directly flows into
- **Predecessor set** (*pred*(*x*))
  - set of all nodes that flow directly into *x*
- An expression is defined at the point where it is assigned a value
- An expression is killed when one of its operands is subsequently assigned a new value
- An expression is available at some point *p* in a flow graph if every path leading to *p* contains a prior definition of that expression which is not subsequently killed

Definitions

`avail[B]` = set of expressions available on entry to block B
`exit[B]` = set of expressions available on exit from B
`avail[B]` = ∩ `exit[x]`
(i.e. B has available the intersection of the exit of its predecessors, where `x ∈ pred[B]`)
`killed[B]` = set of expressions killed in B
`defined[B]` = set of expressions defined in B
`exit[B]` = `avail[B]- killed[B]+ defined[B]`
`avail[B]` = `∩ (avail[x]-killed[x]+ defined[x])`
where `x ∈ pred[B]`

Algorithm

1. Compute defined and killed sets for each basic block (does not involve any of its predecessors or successors)
2. Iteratively compute avail and exit sets for each block as follows, until a stable fixed point is hit:
   a) Identify each statement **`s`** of the form **`a = b op c`** in some block B such that **`b op c`** is available at the entry to B and neither **`b`** nor **`c`** is redefined in B prior to **`s`**
   b) Follow flow of control backward in the graph passing back to, but not through, each block that defines **`b op c`**. The last computation of **`b op c`** in such a block reaches **`s`**
   c) After each computation **`d = b op c`** identified in step 2a, add statement **`t = d`** to that block, where **`t`** is a new temp
   d) Replace **`s`** by **`a = t`**

Data Flow Analysis: Exercise

```
main:
   Procbegin
   b = a + 2
   c = 4 * b
   tmp1 = b < c
   if not tmp1 goto L1
   b = 1
L1:
   d = a + 2
   Procend
```

A:
```
main:
   Procbegin
   b = a + 2
   c = 4 * b
   tmp1 = b < c
   if tmp1 goto L1
```

B:
```
   b = 1
```

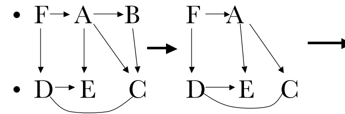
C:
```
L1:
   d = a + 2
   Procend
```

```
defined(A) =
killed(A)  =
avail(A)   =
exit (A)   =
```

```
defined(B) =
killed(B)  =
avail(B)   =
exit (B)   =
```

```
defined(C) =
killed(C)  =
avail(C)   =
exit(C)    =
```

Another R-Coloring Example



- Order of removal: B, C, A, E, F, D
- Assume 3 colors are available: assign colors in reverse order, constrained by already colored nodes.
- D (no constraint) F (D) E (D) A (F, E) C (D, A) B (A, C)

Constant Propagation in RISC

- Constant propagation is particularly important in RISC architectures
  - moves integer constants to the place they are used.
  - may reduce the number of registers needed and instructions executed

Example
  - MIPS has an addressing mode that uses the sum of a register and a constant offset, but not one that uses the sum of two registers
  - Propagating a constant to such an address construction can eliminate the extra add instruction as well as the additional registers needed for that computation

Unoptimized IC:

```
_tmp0 = 12
_tmp1 = arr + _tmp0
_tmp2 = *(_tmp1)
```

MIPS:

```
li $t0, 12
lw $t1, -8($fp)
add $t2, $t1, $t0
lw $t3, 0($t2)
```

Optimized IC:

```
_tmp0 = *(arr + 12)
```

MIPS:

```
lw $t0, -8($fp)
lw $t1, 12($t0)
```