Local and Regional Optimizations (Objectives)

- The student will be able to define and differentiate between local, global and whole-program optimization.
- The student will be able to state the benefits and potential negative effects of optimization (specifically redundancy elimination).
- Given a program, the student will be able to eliminate redundant expressions locally and regionally.

Motivation

- Code generated from the front end in a more general fashion than may be necessary.
- Consider a reference to A[i,j]
  - Code will look like

    $addr(A) + ((i - low_1(A)) * (high_2(A) - low_2(A) + 1) + (j - low_2(A))) * size(A)$
  - The bounds may be known constants allowing us to simplify the expression
    - constant propagation

Code Optimization

- The goal of code optimization is to discover, at compile time, information about the run-time behavior of the program and to use that information to improve the code generated by the compiler.
- Optimizations
  - improve execution efficiency
  - improve space efficiency

Example

- Consider matrix vector multiply

```
do j = 1, n
   do i = 1, n
      y(i) = y(i) + x(j) *m(i,j)
   enddo
enddo
```

- We can improve the code by "unrolling" the outer loop.

- After unrolling by 4

```
do j = 1, n, 4
   do i = 1, n
     y(i) = y(i) + x(j) *m(i,j)
     y(i) = y(i) + x(j+1) *
                   m(i,j+1)
     y(i) = y(i) + x(j+2) *
                   m(i,j+2)
     y(i) = y(i) + x(j+3) *
                   m(i,j +3)
   enddo
enddo
```

- What has improved?

Considerations for Optimization

- Safety – Does the optimization preserve the semantics of the original code?
  - order of definitions and uses
  - maintain control flow restrictions
- Profitability – Does the optimization improve the code in some way
  - execution speed
  - code size
  - power consumption
- Downside Risk – Does the optimization make it harder to generate good code in the back end
  - register pressure
  - scheduling
  - address calculation form

Opportunities for Optimization

- In general, compiler optimizations are for

1. Reducing the overhead of abstraction
2. Taking advantage of special cases
3. Matching processor resources

Local Optimization

- Optimization that occurs over a basic block – a sequence of instructions with one entry point and one exit point.
  - DAG
  - Local Value Numbering

Redundant Expressions

$m = 2 \times y \times z$
$n = 3 \times y \times z$
$o = 2 \times y - z$

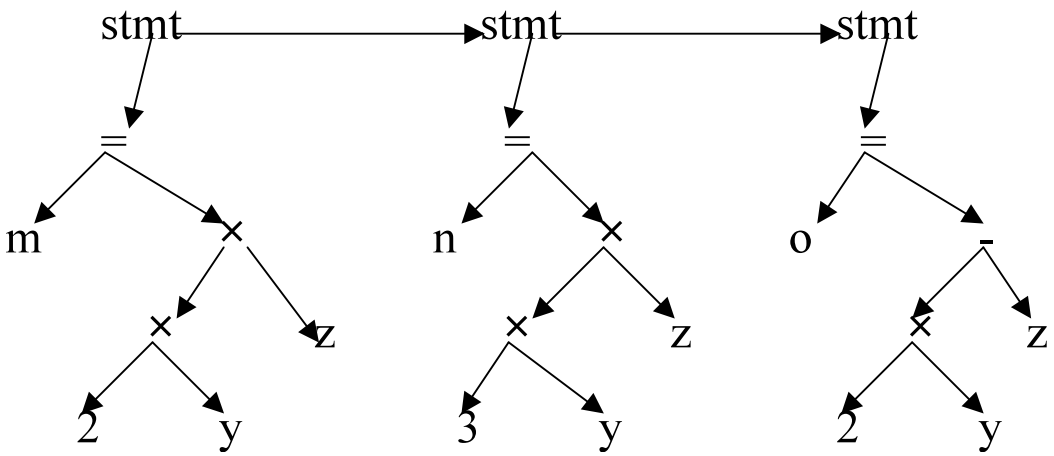
- Both $2 \times y$ and $y \times z$ are redundant

$t_0 = 2 \times y$
$m = t_0 \times z$
$n = 3 \times y \times z$
$o = t_0 - z$

- Code after eliminating $2 \times y$

Local Redundancy Elimination

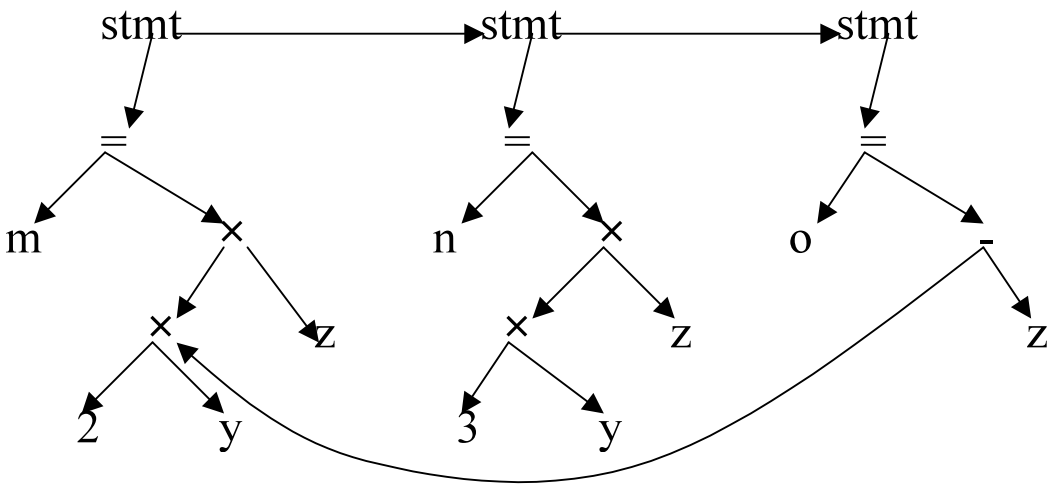
- The AST for the previous code is



- Does not easily reveal redundancy

Local Redundancy Elimination

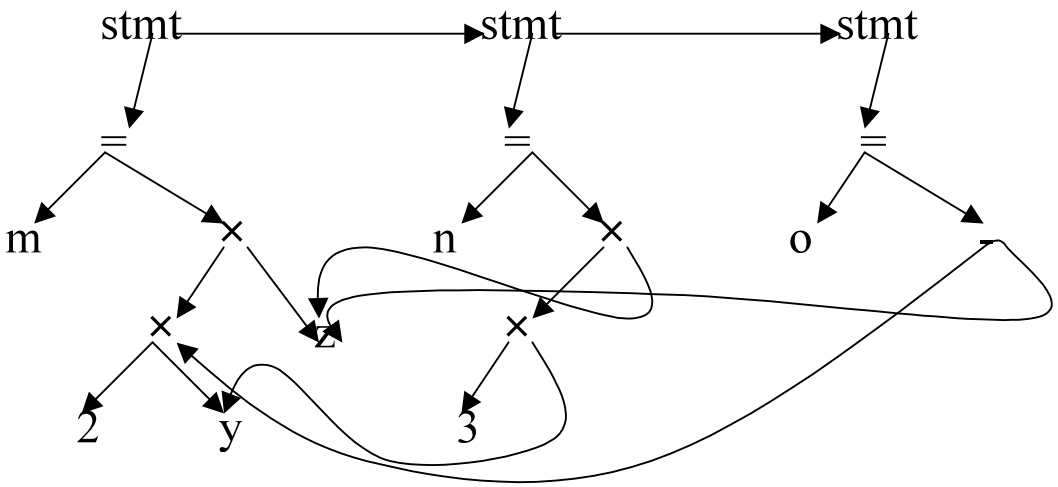
- To eliminate the redundancy we can build a DAG instead of an AST



- We can actually eliminate load redundancies too

Local Redundancy Elimination

- Complete DAG



Method

- Construct a hash table of expressions
- Associate a counter with each variable
  - increment the counter with each assignment
- For each expression do a lookup in the hash table of <(var,#),op,(var,#)> in the hash table
  - if it is there we have a redundant expression, return a pointer to the DAG
    - also handle lookup of variables too
  - if the expression is not there, add it to the table and create a new DAG

Pointers

- What do we do with a pointer assignment like

```
*p = 0;
```

  - increment the counter of all variables that p could point to -- safety

Example

- Generate a DAG for the following code

```
x = 3 * a
b = a
y = 3 * b
a = 5
z = 3 * a
*p = 7
m = 3 * a
```

Value Numbering

- Eliminates redundancy locally on a linear IR
- Assign a unique number to each value computed at run-time
- Use a hash table indexed by <left value #, op, right value #> to identify redundant computation.

Simple Value Numbering

For each operation i in a basic block of the form

$op_i\ x_1, x_2, \ldots, x_{k-1} \Rightarrow x_k$

1. get the value numbers for $x_1, x_2, \ldots, x_{k-1}$ (create a new unique number if none exists)
2. construct a hash key from op_i and the value numbers for $x_1, x_2, \ldots, x_{k-1}$
3. if the value already exists in the table
   replace op_i with a copy operation
   record the value number for x_k

   else
   insert the hash key into the table
   assign the hash key a new value number
   record the value number for x_k

Example

- Apply value numbering to the following

```
a = b + c
b = a - d
c = b + c
d = a - d
```

Example

- Apply value numbering to the following

```
a = b + c
b = a - d
e = b + c
d = a - d
f = d + c
```

Extending the Algorithm

- Perform local constant propagation
  - if all operands are constant, then compute the result at compile time
- Recognize algebraic identities
  - replace expressions like $a \times 1$ with a; $a \times 0$ with 0
- Handle commutativity
  - sort operands by value number to put in canonical form

New Algorithm

For each operation i in a basic block of the form

$op_i\ x_1, x_2, \ldots, x_{k-1} \Rightarrow x_k$

1. get the value numbers for $x_1, x_2, \ldots, x_{k-1}$ (create a new unique number if none exists)
2. if all operands are constant
   evaluate expression and replace with loadI
3. if operation is an identity replace with result of identity
4. if op_i is commutative sort operand by value number
5. construct a hash key from op_i and the value numbers for $x_1, x_2, \ldots, x_{k-1}$
6. if the value already exists in the table
   replace op_i with a copy operation
   record the value number for x_k

   else
   insert the hash key into the table
   assign the hash key a new value number
   record the value number for x_k

Example

loadI	5	$\Rightarrow r_1$
add	r_2,r_1	$\Rightarrow r_{11}$
move	r_{11}	$\Rightarrow r_3$
loadI	3	$\Rightarrow r_4$
add	r_4,r_1	$\Rightarrow r_{12}$
add	r_1,r_2	$\Rightarrow r_{13}$
loadI	5	$\Rightarrow r_1$

Naming

- Consider the following

  $a = x + y$
  $b = x + y$
  $a = 17$
  $c = x + y$

- Give names a unique id to fix this problem

  $a_0 = x_0 + y_0$
  $b_0 = x_0 + y_0$
  $a_1 = 17$
  $c_0 = x_0 + y_0$

  - must insert copy operation

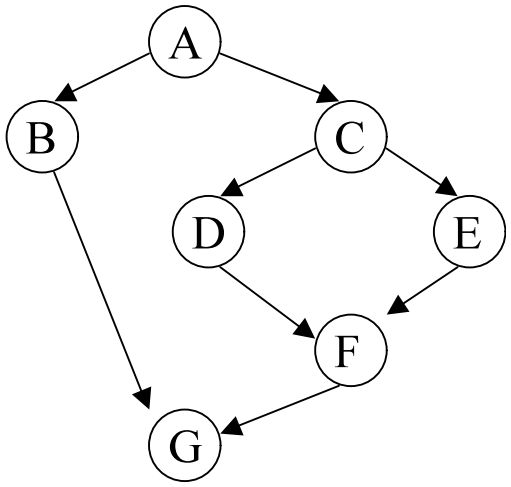
- Also can just keep track of all names with same value # and delete when new assignment occurs (in example use b at end)
- Can generate expressions to always store in same temporary

Superlocal Redundancy Elimination

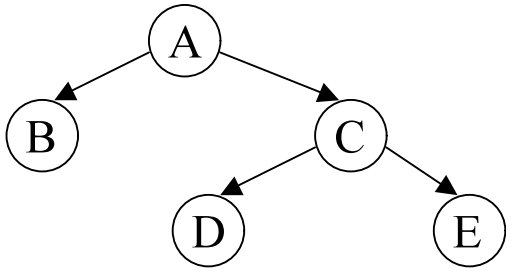
- Operate over extended basic blocks
  - a set of basic blocks B_1, B_2, ..., B_n such that B_1 may have multiple predecessors, each B_k for $2 \leq k \leq n$ has a single predecessor and there is a path from B_1 to B_k only going through other nodes in the extended basic block
- An extended basic block can have multiple exit points but only one entry point.

Example

- CFG



- An EBB rooted at A



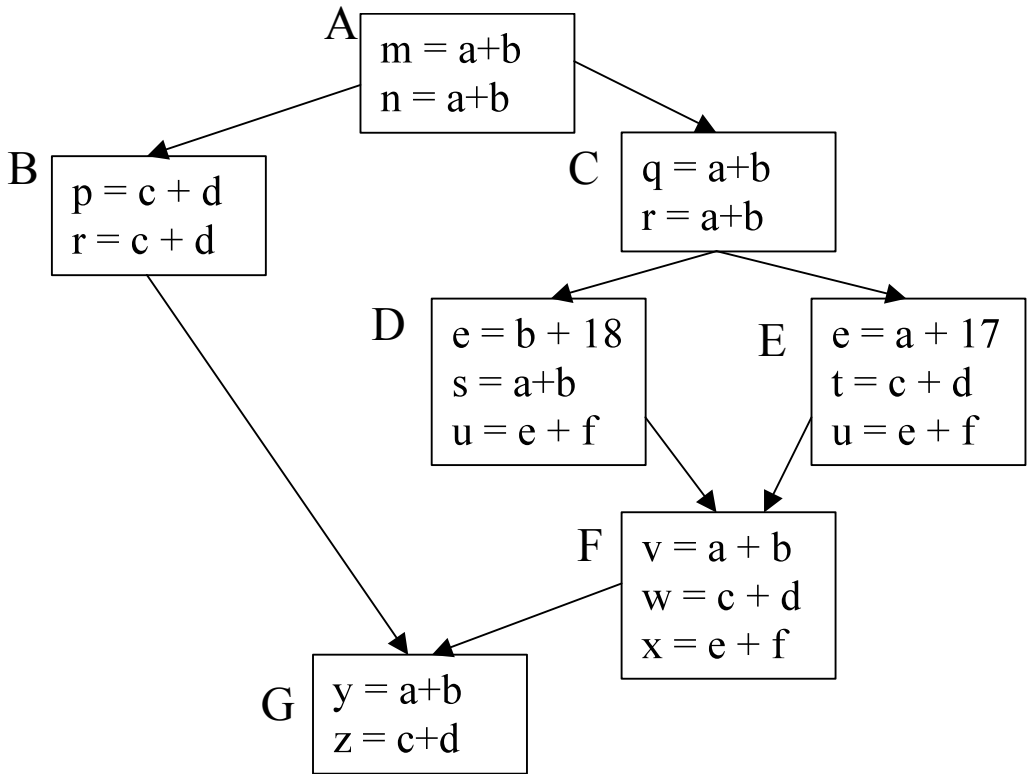
- The EBB is a tree
- Are there other EBBs?

Value Numbering on an EBB

- Compute maximal EBBs
- Consider the value flow down any path
- Propagate the hash tables from predecessor to successor
- Throw away information about the current block when moving back up the tree.
  - How can we organize the tables to accomplish this?

Example

- Perform value numbering on the following using EBBs



Regional Methods

- Consider regions larger than an extended basic block
  - loop structures
  - if then else
- Can use a hierarchy of regions
  - in the previous example one region could be {C,D,E,F}
  - the super region would be {A,B,{C,D,E,F},G}
- Regions have to deal with merge points
  - What do we do with value numbers that merge from different paths?
    - consider basic block F

Global Methods

- Consider an entire function as the scope
  - also called intraprocedural methods
- Local decisions can have bad consequences globally
- Functions are natural boundaries for optimization
  - separate compilation
  - insulate run-time environments
- Need global analysis
  - global data-flow analysis

Whole-program Methods

- Consider the entire program
  - also called interprocedural methods
- Have to deal with
  - parameter binding
  - aliasing
  - multiple invocations
- Difficult to determine if the optimization will be profitable
  - example: function inlining