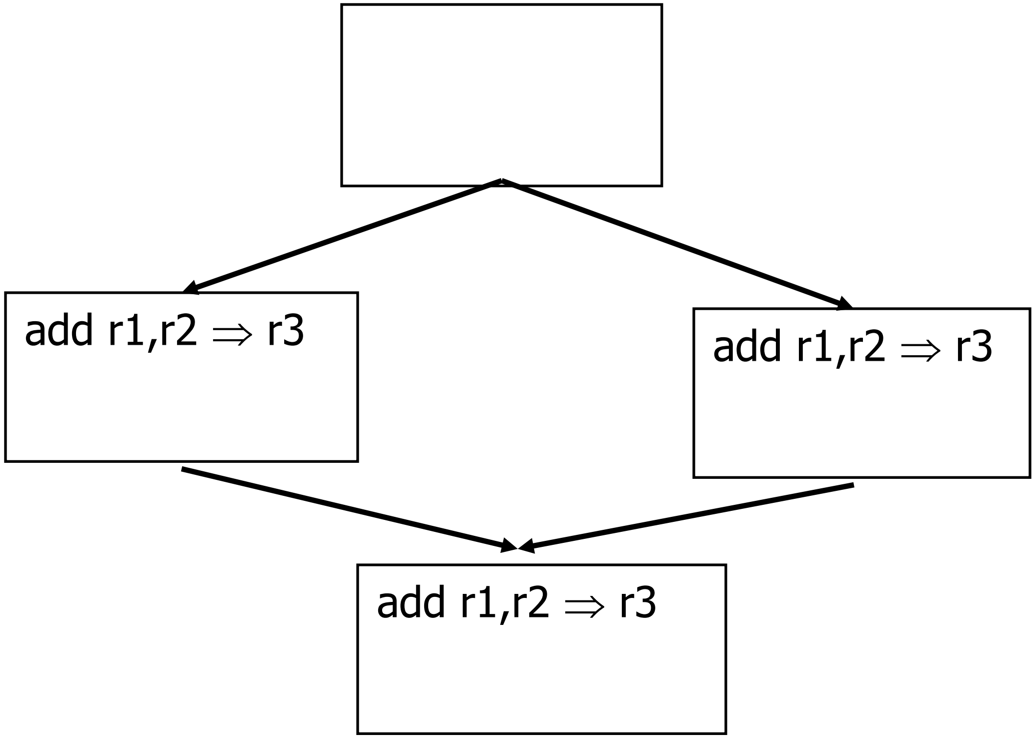
Global Data-flow Analysis and Optimization (Objectives)

- Given a function in intermediate form, the student will be able to perform available expression analysis and global redundancy elimination.
- Given a function in intermediate form, the student will be able to perform live-variable analysis and global dead code elimination.

Motivation

- Local techniques for removing redundancy do not consider an entire function as context



Method

1. Build the basic blocks and control flow graph (CFG)
2. Compute local availability of expressions information
3. propagate local information throughout the CFG
4. goto 2 until a fixed point is reached
5. Use information at each basic block to remove redundant expressions

Assumptions

- The intermediate code is generated such that all lexically identical instructions store into the same temporary register (the only instructions that store into this register)
- Therefore, expressions can be identified by the result register number only
- The set representation used will be bit vectors.

Local Information

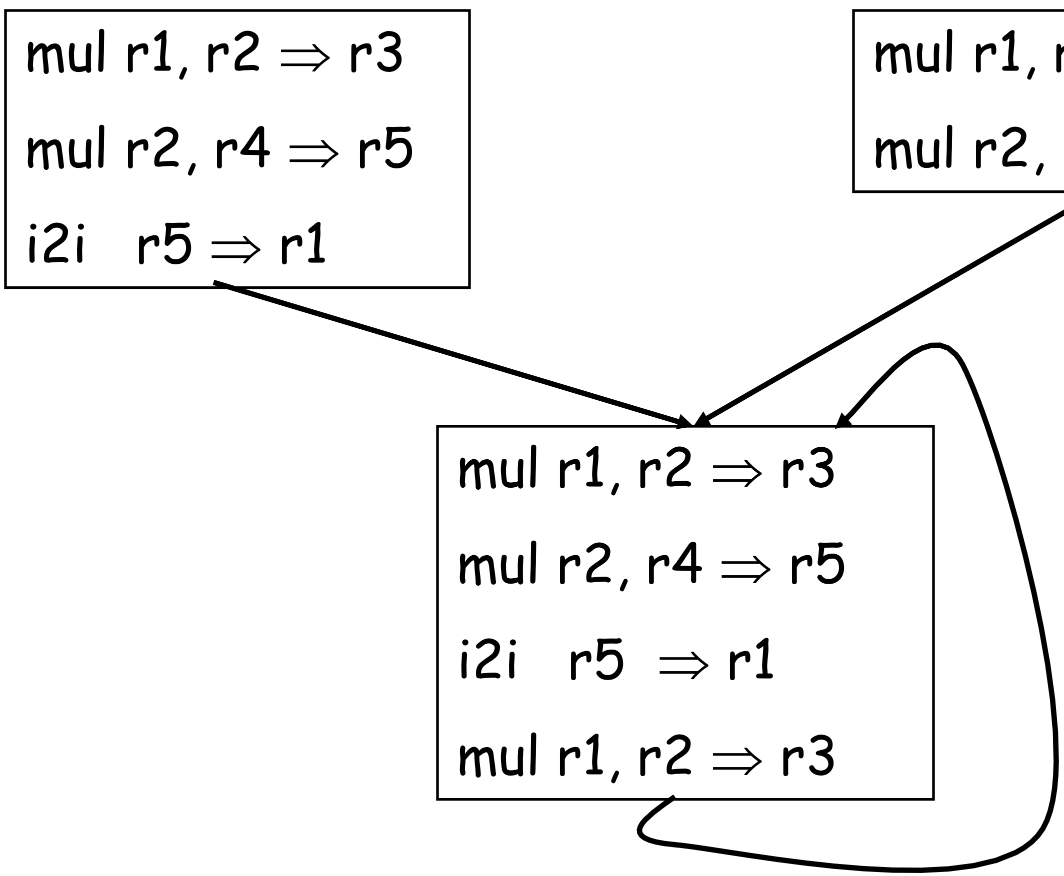
1. The set of expressions that have definitely been computed and have not had operands redefined before entering the current basic block – IN
2. The set of expressions that are computed in the basic block whose operands are not changed later in the same block – GEN
3. The set of expressions computed anywhere in the function that do not have their operands defined in the current block – PRSV
4. The set of expressions that are definitely computed before or in the block and do not have their operand redefined in the block - OUT

Computing Local Information

```
ComputeLocal(b) {
    IN(b) = U  (Entry node is ∅)
    GEN(b) = ∅
    PRSV(b) = U
    KILLED = ∅
    for each i∈b in reverse order {
        if the operands of i are not in KILLED
            GEN(b) ∪= {i.lval()}
        KILLED ∪= {i.lval()}
        for each e in which i.lval() is an operand
            PRSV(b) -= {e.lval()}
    }
    OUT(b) = GEN(b) ∪ (IN(b) ∩ PRSV(b))
}
```

Example

- Compute local information on the following CFG



Global Propagation

- The expressions available at the entry of a basic block, b, (IN(b)) are those available at the end of every predecessor block, p, of b (OUT(p))

$$IN(b) = \bigcap_{p \in pred(b)} OUT(p)$$

- The expressions available at the exit of the block, b,are OUT(b)

$$OUT(b) = GEN(b) \cup (IN(b) \cap PRSV(b))$$

Global Propagation

- Iteratively compute $\forall b$, IN(b) and OUT(b) until there is no change in any set (fixed point)
- The sets can be computed in any order and the answer will not change
- For efficiency compute the OUT of all predecessors before the IN of the current block

```
Propagate(b) {
    mark node as visited
    for each unvisited
            predecessor p of b
        Propagate(p)
    compute IN and OUT
}


while any IN or OUT changes
    Propagate(Exit)
```

Example

- Propagate information in the previous example

Global Redundancy Elimination

```
EliminateRedundacy(G) {
    for each b ∈ G {
      AVAIL = IN(b)
      for each i ∈ b in execution order {
        if i.lval() ∈ AVAIL
          remove i
        else {
          AVAIL ∪= {i.lval()}
          for each instruction j in i.lval() is an operand
             AVAIL -= {j.lval()}
        }
      }
}
```

Example

- Perform redundancy elimination on the example

Live-variable Analysis

1. IN(b) – all variables that have an upwards exposed use after the beginning of b.
2. GEN(b) – all variables used in B but not defined in earlier in b.
3. PRSV(b) – all variables not defined in b.
4. OUT(b) – all variables that have an upwards exposed use on some path exiting b.

Data-flow Equations

- Initialization

$$\forall b \in Blocks, OUT(b) = \varnothing$$

- Flow between basic blocks

$$OUT(b) = \bigcup_{s \in succs(b)} IN(s)$$

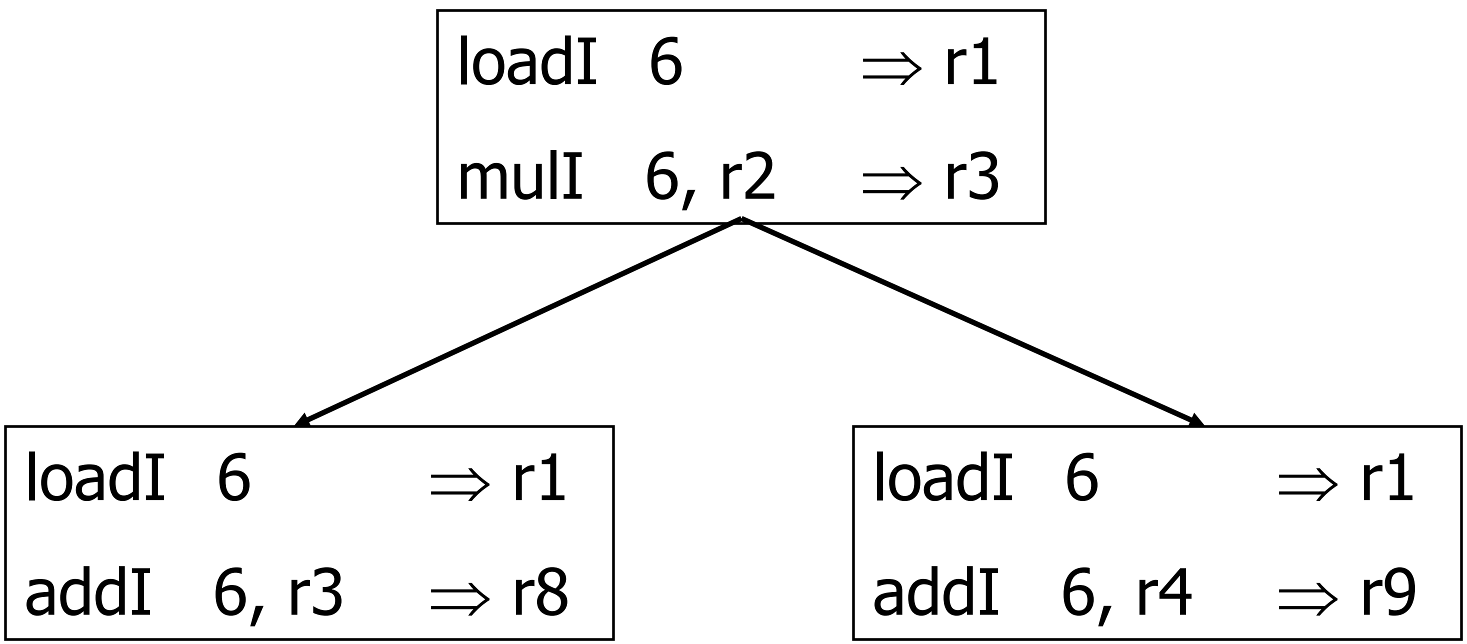
$$IN(b) = GEN(b) \cup (OUT(b) \cap PRSV(b))$$

Computing Local Information

```
ComputeLocal(b) {
   GEN(b) = ∅
   PRSV(b) = U
   for each instruction i ∈ b in execution order {
      for each rvalue, r, of i such that r ∈ PRSV(b) {
         GEN(b) ∪= {r}
      }
      PRSV(b) -= {i.lval}
   }
}
```

Example

- Perform live variable analysis assuming r8 and r9 are live on exit



Removing Dead Code

```
RemoveDeadCode(G) {
    for each b ∈ G {
      LIVE = OUT(b)
      for each i ∈ b in reverse order {
         if (i.lval ∉ LIVE)
            remove i from b
         else
            LIVE -= {i.lval}
            for each rvalue, r, such that r ∈ i
               LIVE ∪= {r}
      }
    }
}
```

Example

- Perform dead-code elimination: OUT(b) ={r4,r8}

```
add   r1,r2    ⇒ r3
i2i   r3       ⇒ r4
add   r5,r6    ⇒ r7
i2i   r7       ⇒ r8
add   r6,r7    ⇒ r9
i2i   r9       ⇒ r4
add   r5,r10   ⇒ r11
i2i   r11      ⇒ r8
```

Reaching Definitions

1. IN(b) – the set of definitions whose value can reach the beginning of b.
2. GEN(b) – the set of definitions in b that are not subsequently killed in b
3. PRSV(b) – the set of definitions that have no redefinition in b
4. OUT(b) – the set of definitions that reach beyond the end of b.

Data-flow Equations

- Initialization

$$\forall b \in G, IN(b) = \varnothing$$

- Propagation

$$IN(b) = \bigcup_{p \in pred(b)} OUT(p)$$

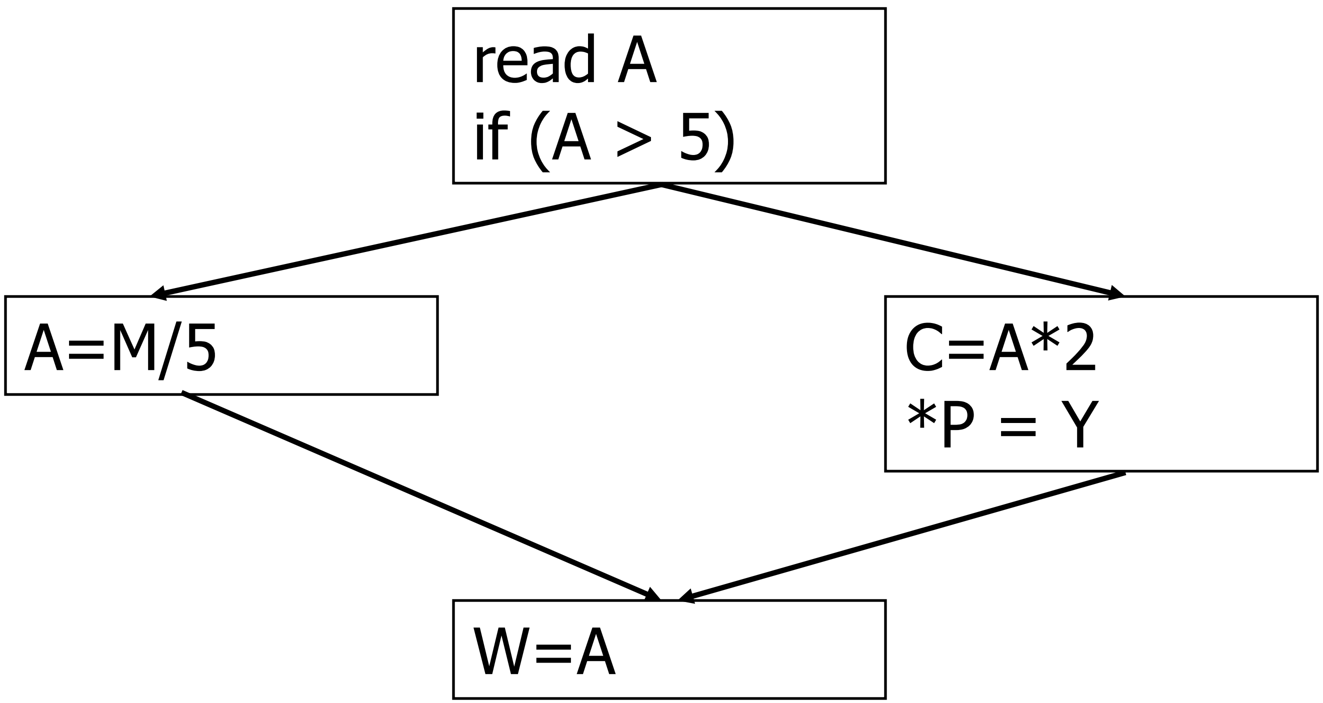
$$OUT(b) = GEN(b) \cup (IN(b) \cap PRSV(b))$$

Computing Local Information

```
ComputeLocal(G) {
   GEN(b) = ∅
   PRSV(b) = U
   for each instruction i ∈ b in reverse order {
      if (i.lval ∈ PRSV(b))
         GEN(b) ∪= {i.lval}
      remove all definitions of i.lval from PRSV(b)
   }
}
```

Example

- Compute reaching definitions

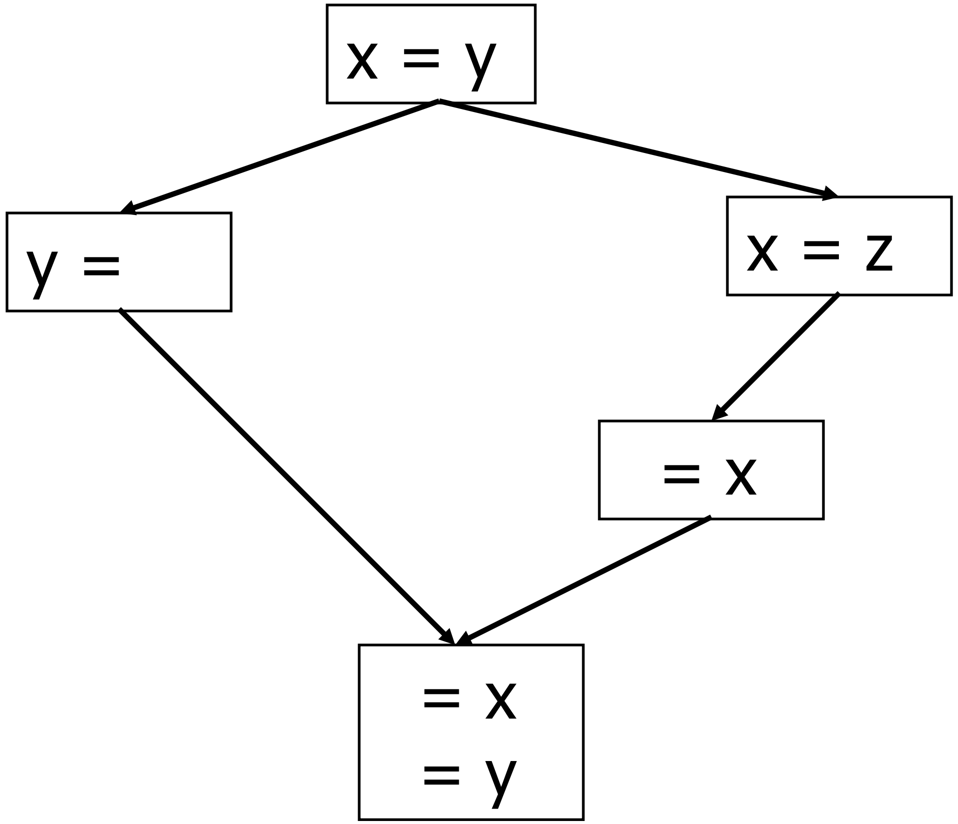


DU-UD Chains

- link together the definitions and uses of values in a program
- Perform reaching definitions analysis
- At each use create a bi-directional link from the use to each of its reaching definitions
- This can be used for
  - copy propagation
  - constant propagation

Example

- Compute the DU-UD chains for the following



Copy Propagation

- forward data-flow problem
- GEN(b) – the set of copy statements x:=y that occur in b for which neither x nor y is later redefined.
- PRSV(b) – the set of copy statements x:=y that occur anywhere in the program such that neither x nor y is defined in b

$$IN(b) = \bigcap_{p \in preds(b)} out(p)$$

$$OUT(b) = GEN(b) \cup (IN(b) \cap PRSV(b))$$

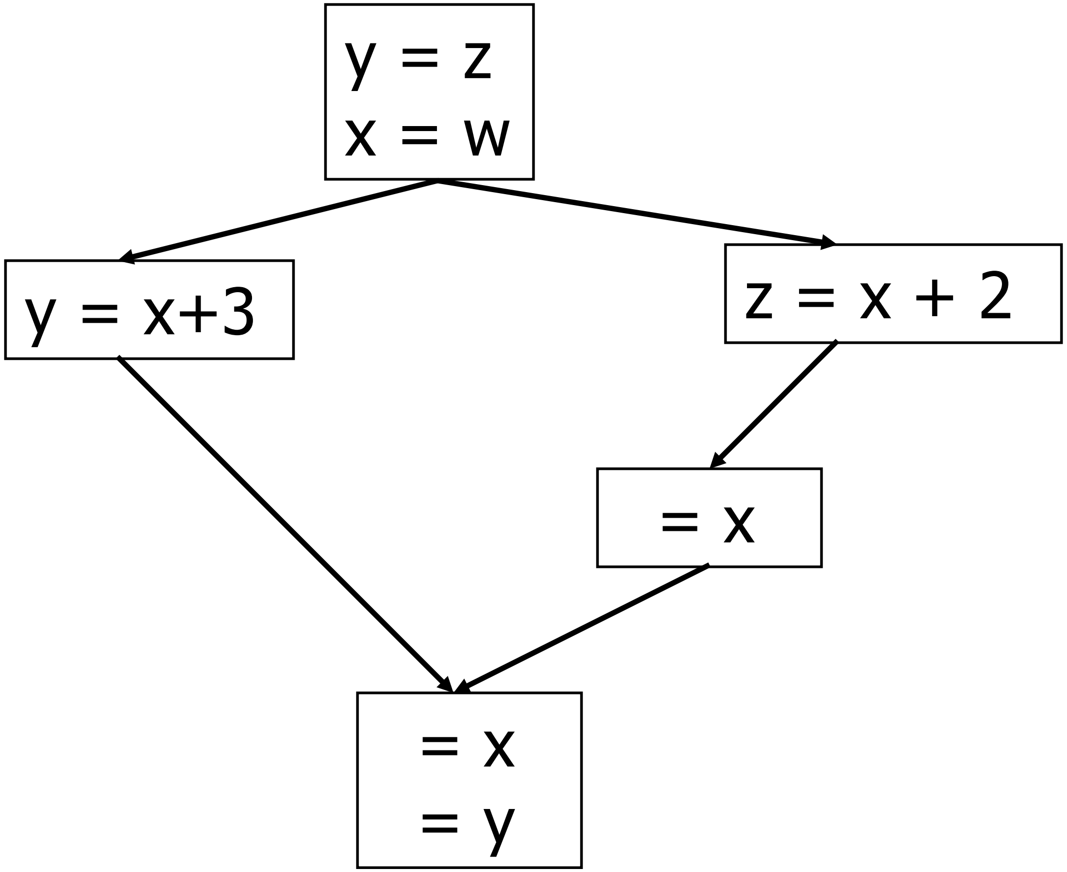
- Propagation is just like available expression analysis

Copy Propagation

```
for each copy s: x := y do {
   use du chains to find all uses of x
   if ∀u ∈ uses(x) , s ∈ IN(block(u)) and no definition
        of y or x occurs in block(u) before u
      remove s and replace the uses of x on the
         du chains with y
   }
}
```

Example

- Perform copy propagation on the following



Constant Propagation

- Use du-ud chains.
- Associate a value cell with each definition
- Three possible values
  1. unknown
  2. notconst
  3. constval
- Meet operation

$$a \wedge b = \begin{cases} a & a == b \\ notconst & a \neq b \end{cases}$$

$$a \wedge unknown = a$$

$$a \wedge notconst = notconst$$

Constant Propagation

```
CP(D) {
   set all value cells to unknown
   for each cell walking preds before each cell {
      for each rval, r, used to compute this cell {
         perform the meet of the cells r's reaching defs
      }
      if all rvals are constant
         compute result and store if new val
      else if any rval is notconst
         make r notconst if not already
   }
   if any cell changed call CP(D)
}
```

Example