CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107HS in the
Winter 2013/2014 term at the University of Toronto

Reading Assignment

Fischer, Cytron and LeBlanc

Sections 10.3, 14.7

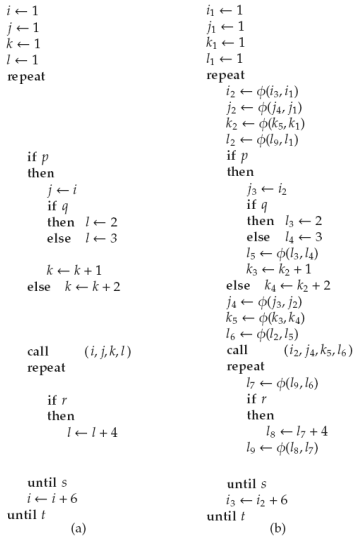
Sections 14.2 .. 14.2.3

Sections 14.3 .. 14.3.4

Static Single Assignment

- The Static Single Assignment (SSA) form is an IR used to facilitate certain kinds of optimization.
- Key features of SSA
  - Each assignment to a variable is a **definition** of the variable for the particular value being assigned.
  - *Rename* all variables in the program using some systematic scheme so that each variable is assigned a value exactly *once*.
  - If more than one definition for the same variable is available at some point in the program, add an *articulation point* (ϕ function) to identify the conflict.
  - Once the variables have been renamed, each uniique variable has fixed value from its point of definition to the end of the program
- The SSA form facilitates optimizations like constant propagation and constant folding (Slide 468) and variable folding (Slide 478)

SSA Example (Figure 10.5)



Data Flow Analysis

- Data Flow Analysis is a technique for discovering properties of the run-time behavior of programs during compilation.
- These properties are *universal*, i.e. they apply to all possible sequences of control and data flow.
- The information from data flow analysis is used to determine feasibility and safety of various optimizations.
- Control Flow Graphs ($\mathcal{G}_{cf}$) are used to represent the flow of control between statements in a single routine.
  Procedure Call Graphs ($\mathcal{G}_{pc}$) are used to represent the flow of control between functions and procedures.
- Data Flow Analysis and related optimizations are covered in much greater detail in the follow-on course ECE489H/ECE540H .

Analysis and Transformations

Data Flow Analysis	Transformation
Available expressions	Common subexpression elimination
Detection of loop invariants	Invariant code motion
Detection of induction variables	Strength reduction
Copy analysis	Copy propagation
Live variables	Dead code elimination
Reaching definitions	Identifying constants
. . .	. . .

Data Flow Analysis Overview

- Subdivide the program into Basic Blocks or finer grain (e.g. statements).
- Analyze the control structure of the program to determine the interrelationship of the blocks.
- Represent program control flow as a directed graph ($\mathcal{G}_{cf}$) .
  - Each node in $\mathcal{G}_{cf}$ is a basic block or an individual statement.
  - Each edge in $\mathcal{G}_{cf}$ represents a possible node to node transfer of control.
- Analyze each node to determine where some property of interest is defined, used and killed. e.g. where expressions are computed.
- Dataflow analysis can be applied

  Intraprocedurally within a single routine

  Interprocedurally within some set of routines

  Globally to an entire program
- Sets are represented using bit-vectors (one bit/element).
  Iterative solutions to data flow equations are typically $O(B^2 \times V)$ for a problem with B basic blocks and V variables or expressions.

Definitions for Data Flow Analysis

Basic Block A sequence of instructions that is always executed from start to finish. i.e. it contains no branches.

Definition Point Point where some interesting property is established. (i.e. a variable or an expression is given a value.)

Use Point Point where some interesting property is used.

Killed, Kill Point A point where in interesting property is rendered invalid.

- A variable is assigned a new value.
- *Any* of the variables used to compute the value of an expression is assigned a new value.

Forward Flow What can happen before a given point in a program.

Backward Flow What can happen after a given point in the program.

Any Path (May) analysis What property holds on some path leading to or from a given basic block. Examples: uninitialized variable, live variables.

All Path (Must) analysis What property holds on all paths leading to or from a basic block. Example: availability of an expression.

Predecessors(*b*) the set of all basic blocks that are *immediate* predecessors of block *b* in the control flow graph.

There is an edge in the control graph leading *from* each basic block in Predecessors(*b*) *to b*.

Successors(*b*) the set of all basic blocks that are *immediate* successors of block *b* in the control flow graph.

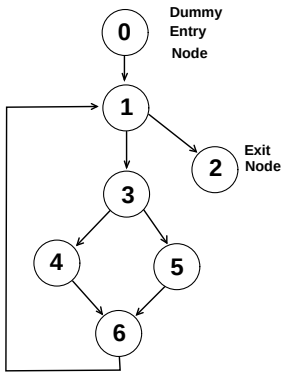
There is an edge in the control graph leading *from b to* each basic block in Successors(*b*).

Basic Block Example

Program and Basic Blocks

```
while ( 1 ) {
   scanf( " %f %f %f", &A, &B, &C ) ;                    1
   if( A == 0.0 )
      break ;                                            2
   DISC = B * B - 4.0 * A * C ;                          3
   if( DISC >= 0.0 ) {
      DROOT = sqrt( DISC ) ;
      R1 = ( - B + DROOT ) / ( 2.0 * A ) ;               4
      R2 = ( -B  - DROOT ) / ( 2.0 * A ) ;
   } else {
      DROOT = sqrt( - DISC ) ;
      R1 = - B / ( 2.0 * A ) ;                           5
      R2 = DROOT / ( 2.0 * A ) ;
   } ;
   printf( "%f %f %f ", DISC, R1, R2 ) ;                 6
} ;
```

Control Flow Graph



Predecessors

1	{ 0 , 6 }	4	{ 3 }
2	{ 1 }	5	{ 3 }
3	{ 1 }	6	{ 4 , 5 }

Successors

1	{ 2 , 3 }	4	{ 6 }
2	{ }	5	{ 6 }
3	{ 4 , 5 }	6	{ 1 }

Types of Data Flow Problems

Forward Flow problems

- Data flows from the first block to the last block.
- *out* sets are computed from *in* sets within basic blocks.
- *in* sets are computed from *out* sets across basic blocks.

Backward Flow problems

- Data flows from the last block to the first block.
- *in* sets are computed from *out* sets within basic blocks.
- *out* sets are computed from *in* sets across basic blocks.

Any Path problems

Values coming into a block through *any* path are valid.

Use $\bigcup$, start with minimum info[a]

All Path problems

Only values coming into a block through *every* possible path are valid.

Use $\bigcap$, start with maximum info[a]

[a] For first/last block other value may be necessary

Generic Data Flow Analysis

- Data flow analysis can be used to compute a wide variety of properties of basic blocks that are useful for optimization.
- Generic definitions

  $In(b)$ what properties hold on entry to basic block b.

  $Out(b)$ what properties hold on exit from basic block b.

  $Gen(b)$ the properties that are generated in basic block b.

  $Killed(b)$ the properties that are invalidated in basic block b.

- For each problem of interest, the precise rules for determining membership in the sets $In, Out, Gen, Killed$ have to be specified.
- Once the membership rules have been determined, one of the iterations over the control flow graph described on the following slide is used to determine In and Out for all basic blocks in the control flow graph.

Generic Forward Data Flow Equations

Any Path	$Out(b) = Gen(b) \bigcup (In(b) - Killed(b))$ $In(b) = \bigcup_{i \in Predecessors(b)} Out(i)$
All Paths	$Out(b) = Gen(b) \bigcup (In(b) - Killed(b))$ $In(b) = \bigcap_{i \in Predecessors(b)} Out(i)$

Generic Backward Data Flow Equations

Any Path	$In(b) = Gen(b) \bigcup (Out(b) - Killed(b))$ $Out(b) = \bigcup_{i \in Successors(b)} In(i)$
All Paths	$In(b) = Gen(b) \bigcup (Out(b) - Killed(b))$ $Out(b) = \bigcap_{i \in Successors(b)} In(i)$

Forward Data Flow Algorithm (All Paths)

for each block B **do**
    $out[B]$:= $gen[B] \cup (in[B] - kill[B])$
change := true
while *change* **do begin**
    change := false
    for each block B **do begin**
        $in[B]$:= $\bigcap_{P \in Pred(B)}$ $out[P]$
        oldout := $out[B]$
        $out[B]$:= $gen[B] \cup (in[B] - kill[B])$
        if $out[B] \neq$ *oldout* **then** *change* := true
    end
end

Available Expression Analysis
Forward All-Paths Analysis

- An expression is *available* at a point P in the program graph G if *every* path leading to P contains a definition of the expression which is not subsequently killed.
- Definitions

$In(B)$ the set of expressions available on entry to basic block B .

$Kill(B)$ the set of expressions that are killed in basic block B .

$Gen(B)$ the set of expressions that are defined in basic block B and are not subsequently killed within the block.

$Out(B)$ the set of expressions available at the exit of block B

$In(Init)$ is $\emptyset$

- Use Available Expression Analysis to implement global common sub-expression elimination.

An expression is *available* at the end of a basic block B if

it is defined in the block and not subsequently killed.

or it is available on entry to the block B and is not killed within the block.

Therefore

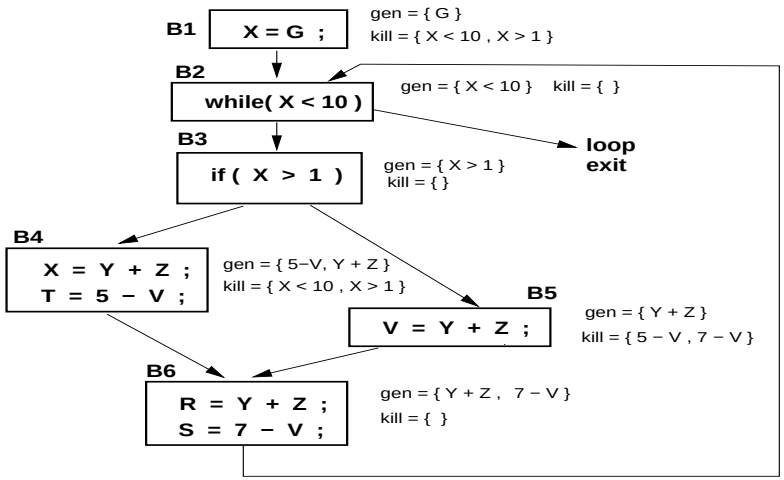
$$Out(B) = gen(B) \bigcup (In(B) - Kill(B))$$

An expression is available on entry to a basic block if it is available on *all* paths leading into the block.

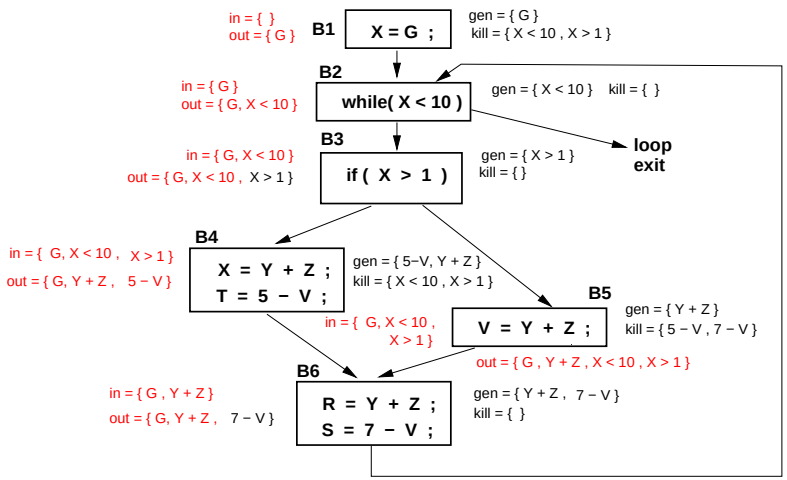
For all basic blocks B compute

$$In(B) = \bigcap_{p \, \varepsilon \, Predecessors(B)} Out(\, p \,)$$

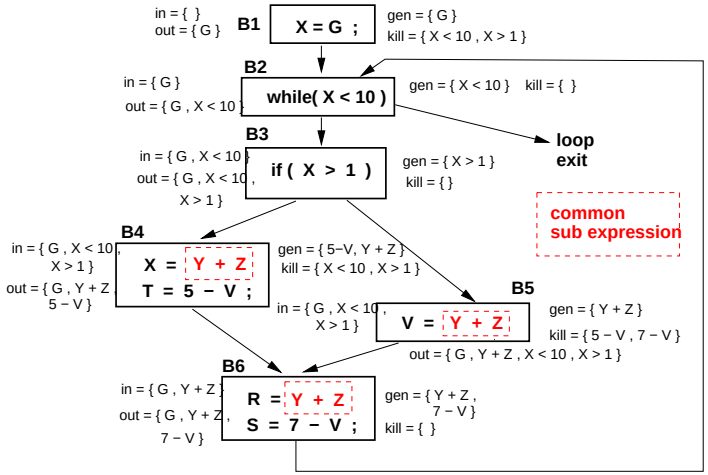
Available Expressions Example



Available Expressions Example



Common Sub−Expression Elimination Example



Forward Data Flow Algorithm (Any Path)

for each block B **do**
    $out[B] := gen[B]$
change := true
while *change* **do begin**
    change := false
    for each block B **do begin**
        $in[B] := \bigcup_{P \in Pred(B)} out[P]$
        oldout := $out[B]$
        $out[B] := gen[B] \cup (in[B] - kill[B])$
        if $out[B] \neq$ *oldout* **then** *change* := true
    end
end

Reaching Definitions
Forward Any-Path Analysis

- A *definition* of a variable x is a statement that assigns a value to x
- A definition d reaches a point p if there is a path from the point immediately following d to p, such that d is not "killed" along that path
- Definitions

$In(B)$ definitions available upon entry to B

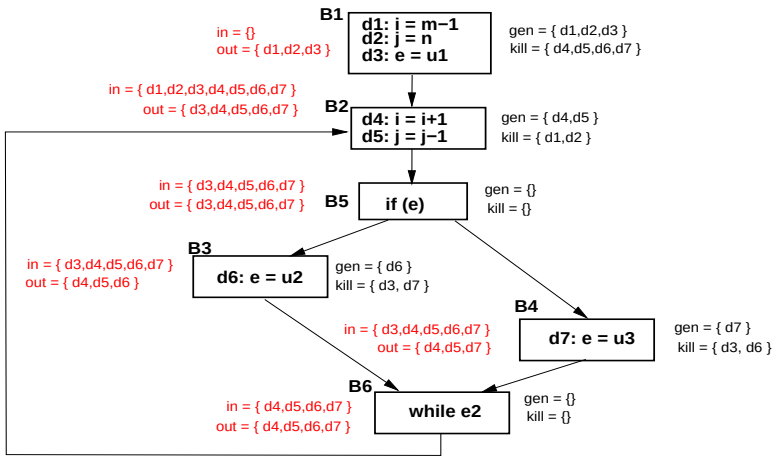
$Gen(B)$ definitions made in B

$Kill(B)$ definitions invalidated by B

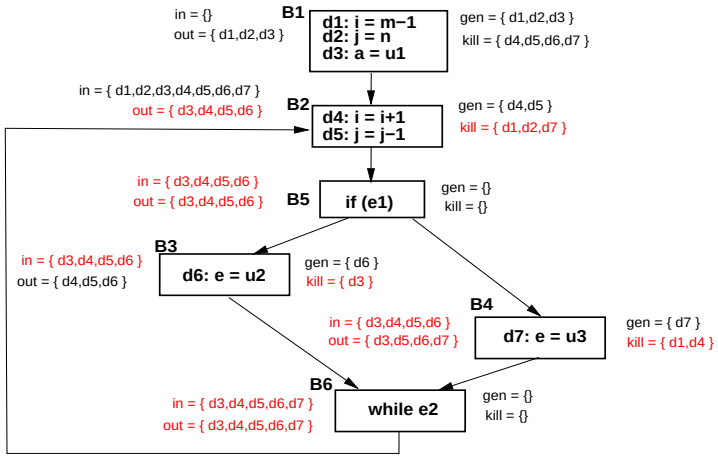
$Out(B)$ is $Gen(b) \bigcup (In(B) - Kill(B))$

$In(Init)$ is $\emptyset$

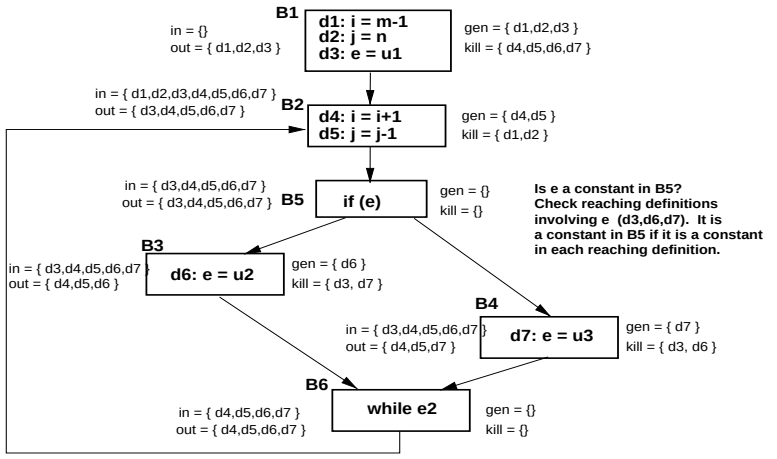
Reaching Definitions Example



Reaching Definitions Example (2nd Iteration)



Identifying Constants



Backward Data Flow Algorithm (Any Path)

for each block B **do**
    $in[B] := gen[B]$
change := true
while *change* **do begin**
    change := false
    for each block B **do begin**
        $out[B] := \bigcup_{P \in Succ(B)} in[P]$
        oldin := $in[B]$
        $in[B] := gen[B] \cup (out[B] - kill[B])$
        if $in[B] \neq$ *oldin* **then** *change* := true
    end
end

Live Variable Analysis
Backward Any Path Data Flow Example

- For each definition/use of a variable V, Live Variable Analysis answers the question:

  Could the value of V computed/used here be used *further on* in the program?

- Used to enable certain kinds of optimizations
  - If V is being stored in a register over some stretch of code it is not necessary to store the register back into V if V is not live.
    Assignments to non-live variables can be deleted.
  - Duration of liveness can be used to pick variables that should be loaded into registers.
  - Example

```
r23 = V        /* V stored in a register */
r23 = r23 + 1  /* V = V + 1              */
/* no uses of V below here */
/* no need to store r23 back into V */
```


Live Variable Analysis
Backward Any-Path Analysis

- A path in the graph G is $V - clear$ if it contains no assignments to the variable V.
  A variable V is *live* at a point P in the graph G if there is a $V - clear$ path from P to a use of V, i.e. there is a path from P to somewhere that V is used which does *not* contain a redefinition of V

- Definitions

$in(B)$ Variables live on entrance to block B

$out(B)$ Variables live on exit from block B.

$def(B)$ Variables assigned values (redefined) in block B *before* the variable is used. (same as $Kill(B)$)

$use(B)$ Variables whose values are used before being assigned to. (same as $Gen(B)$)

$Out(Final)$ is $\emptyset$

A variable V is live at the entrance to a block B if

it is used in B *before* it is defined in B

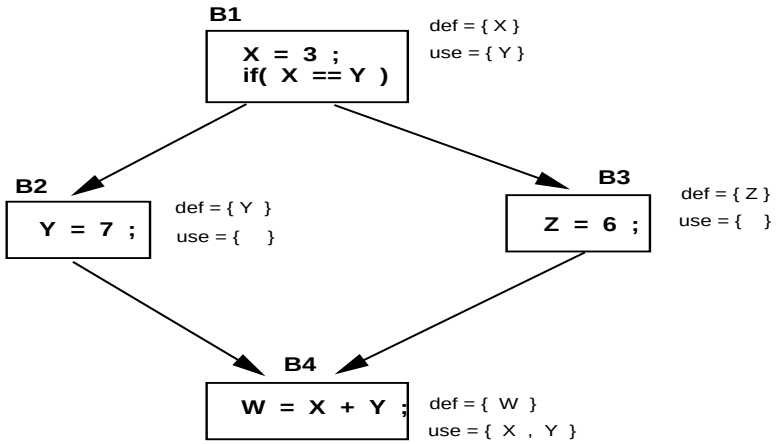
or it is live at the exit to block B and it is not defined within the block

$$in(B) = use(B) \bigcup (out(B) - def(B))$$

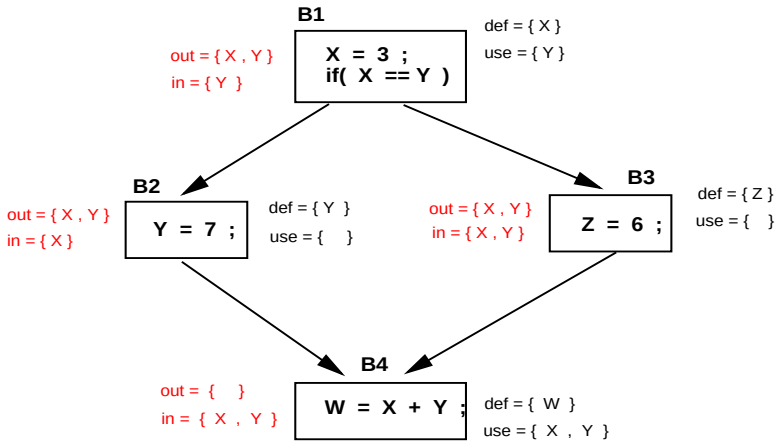
A variable V is live coming out of a block B if it is live going into any of B's successors.

$$out(B) = \bigcup_{S \in successors(B)} in(S)$$

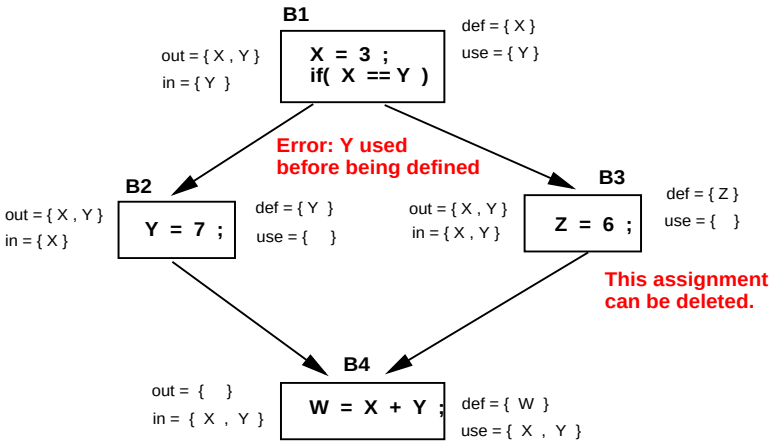
Live Variable Analysis Example



Live Variable Analysis Example



Live Variable Analysis Example



Backward Data Flow Algorithm (All Path)

for each block B **do**
    $in[B] := gen[B] \cup (U - kill[b])$
change := true
while *change* **do begin**
    change := false
    for each block B **do begin**
        $out[B] := \bigcap_{P \in Succ(B)} in[P]$
        oldin := $in[B]$
        $in[B] := gen[B] \cup (out[B] - kill[B])$
        if $in[B] \neq$ *oldin* **then** *change* := true
    end
end

Very Busy Expressions
Backward All-Paths Analysis

- An expression *e* is *very busy* at point p if not matter what path is taken from p, the expression *e* will be evaluated before any of its operands are redefined.
- Definitions

$Out(B)$ expressions very busy after B

$Gen(B)$ expressions constructed by B

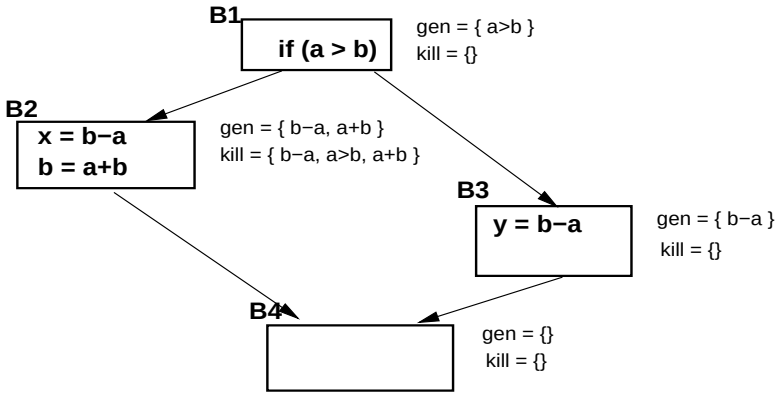
$Kill(B)$ expressions whose operands are redefined in B

$In(B)$ is $Gen(b) \bigcup (Out(B) - Kill(B))$

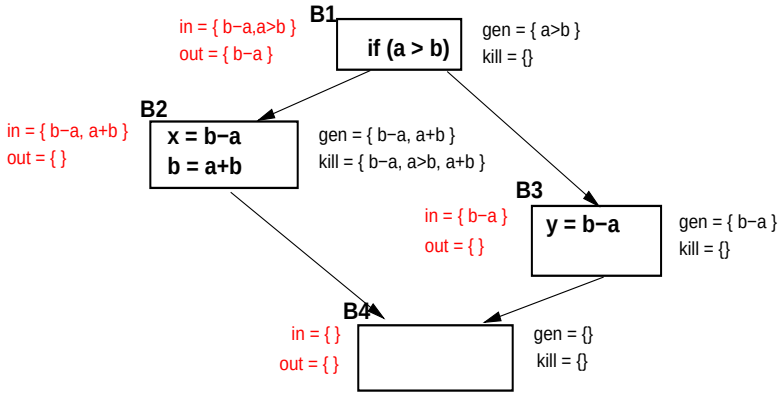
$Out(Final)$ is $\emptyset$

- Use Very Busy Expressions analysis to *hoist* expressions

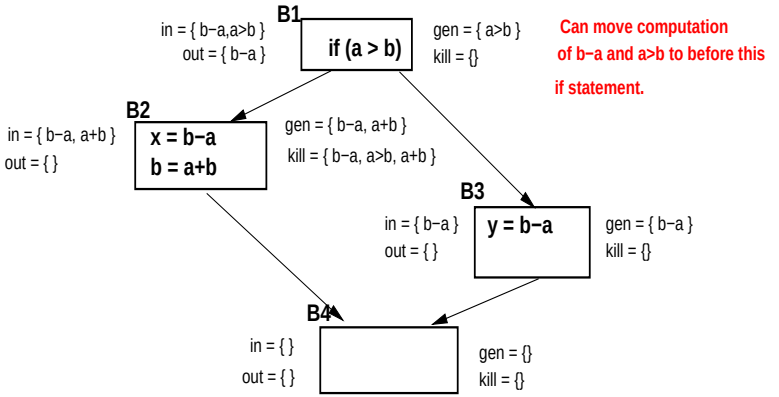
Very Busy Expression Example



Very Busy Expression Example



Code Hoisting

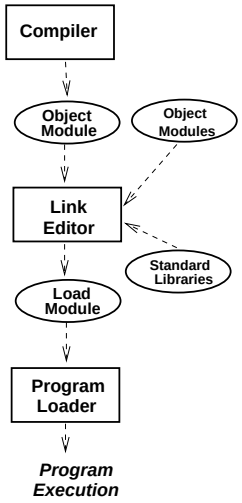


Linking and Loading[a]

An Object module produced by a compiler is linked together with other object modules and standard libraries to form a complete executable program.
Issues dealt with by the linking process

- Resolution of external symbol references
- Relocation of code addresses
- Support for separate compilation
- Support use of standard libraries
- Creation of loadable module

Loading reads a complete executable program (`a.out` or `a.exe`) from disk into memory and starts its execution..



[a]See: John R. Levine, *Linkers & Loaders*, Morgan Kaufmann, 2002

Object Module Information

A typical object module may contain

- Header information. e.g. code size, name of source file, creation date, etc.
- Object code. Binary machine instructions and data.
  Data may include initialized data (e.g. constants) and uninitialized space for static storage.
- Relocation information. Places in the object code that need to be modified when the linker places the object code.
- Symbols
  – **Exported names** defined in the object module.
  – **Imported names** used in the object module
- Debugging information. Information about the object module that is useful to a runtime debugger. e.g. source file coordinates, internal symbol names, data structure definitions.

Linking is a Two Pass Process

Input: a collection object modules and libraries

First Pass

- Scan input files to determine size of all code and data segments.
- Build a symbol table for all imported or exported symbols in all input files
- Resolve all connections between imported symbols and exported symbols
- Maps object code segments to image of output file
- Assign relative addresses in this output block to all symbols

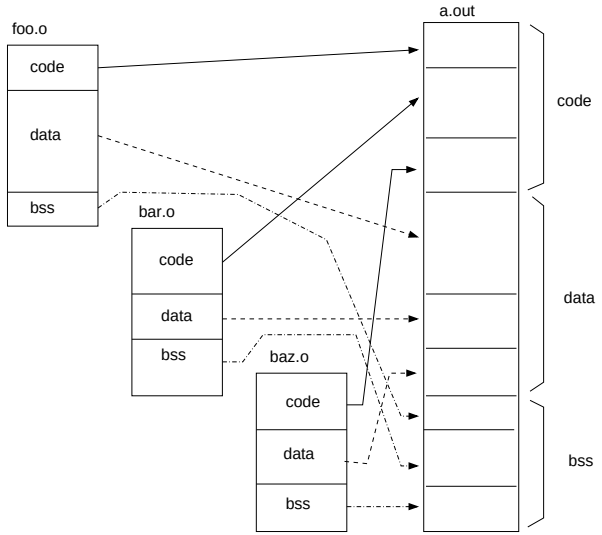
Resolving Symbols

- Every external symbol used in an object module should be resolved to a symbol exported from some other object module or library.
- Unresolved symbol are usually a link time error.
- If multiple instances of a external symbol are available there are several strategies:
  – emit a link time error message
  – pick one symbol with a warning message
  – silently pick one symbol

Second Pass

- Read and relocate object code and data to the output block.
- Replace symbolic addresses with block relative numeric addresses.
- Adjust memory addresses in the object code to reflect new location in the output block.
- Write output file
  – header information
  – relocated code and data segments
  – symbol table information
  – debugging information

Linking Example



Loading

- In most modern systems, the load module can simply be read into a block of memory and then executed. No load time relocation is required.
- To make this possible compilers must generate position independent code e.g. branches are relative to the program counter, data access is via a base register.
- Dynamic linking allows some binding of symbols to library modules to be deferred until runtime.

Dynamic Linking and Loading

- Defer much of the linking process until a program is loaded into memory and starts execution.
- Advantages
  - Dynamically linked shared libraries are easier to create.
  - Dynamic libraries are easier to update without required extensive relinking.
  - Dynamic linking allows runtime loading and unloading of modules.
- Disadvantages
  - Substantial runtime overhead for dynamic linking
  - Dynamic linking redone for every execution.
  - Dynamic link libraries are larger than static libraries due to symbol information.
  - Changes to/unavailability of dynamic libraries may make programs unexecutable.