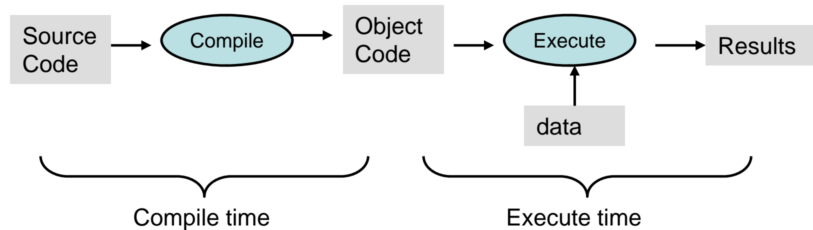
ELEC 876: Software Reengineering (Program Comprehension)

Dr. Ying Zou

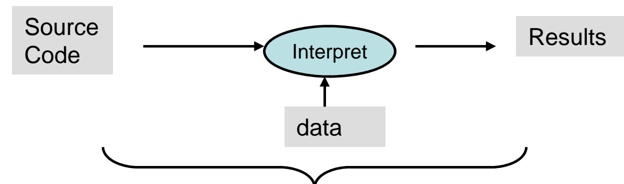
Department of Electrical & Computer Engineering
Queen's University

Compiler and Interpreter

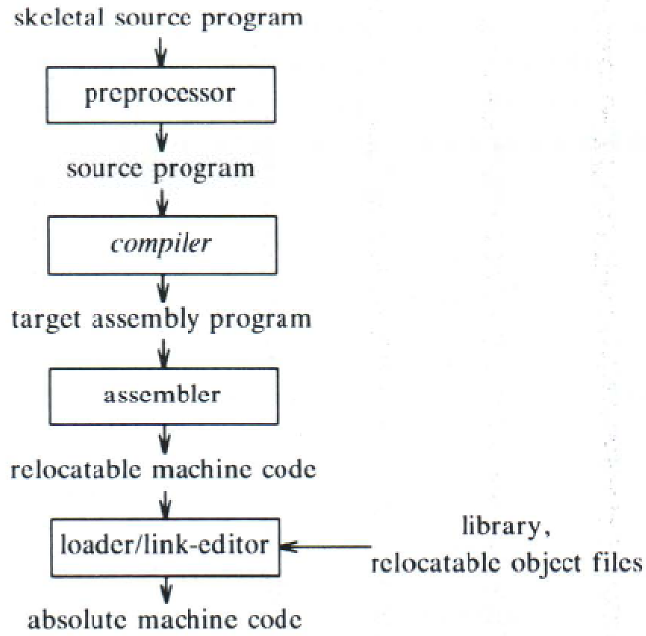
- Compiler



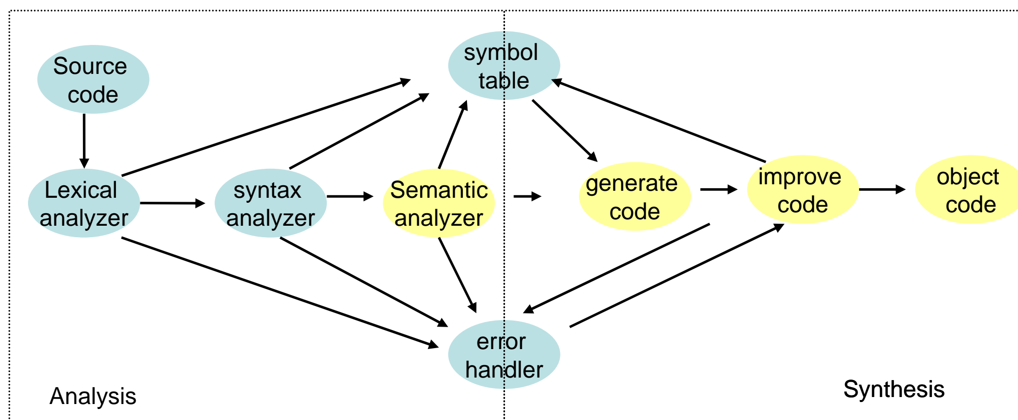
- Interpreter



Compile and run time



The structure (phases) of a compiler



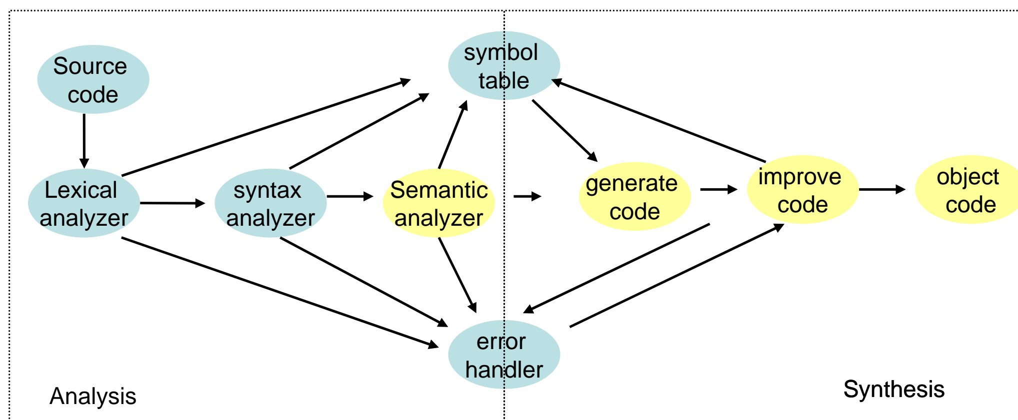
Lexical Analysis

```
position := initial + rate * 60
```

would be grouped into the following tokens:

1. The identifier `position`.
2. The assignment symbol `:=`.
3. The identifier `initial`.
4. The plus sign.
5. The identifier `rate`.
6. The multiplication sign.
7. The number `60`.

The structure (phases) of a compiler



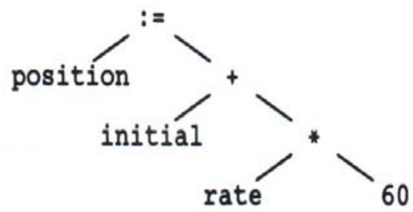


Fig. 1.2. Syntax tree for `position := initial + rate * 60`.

Syntax Analysis

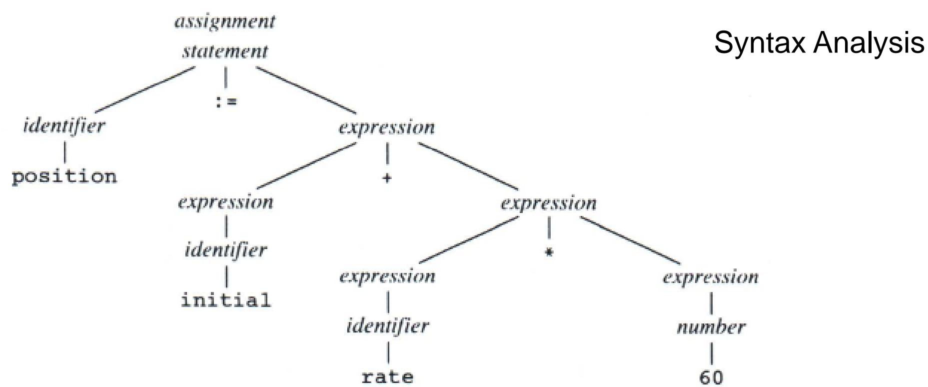
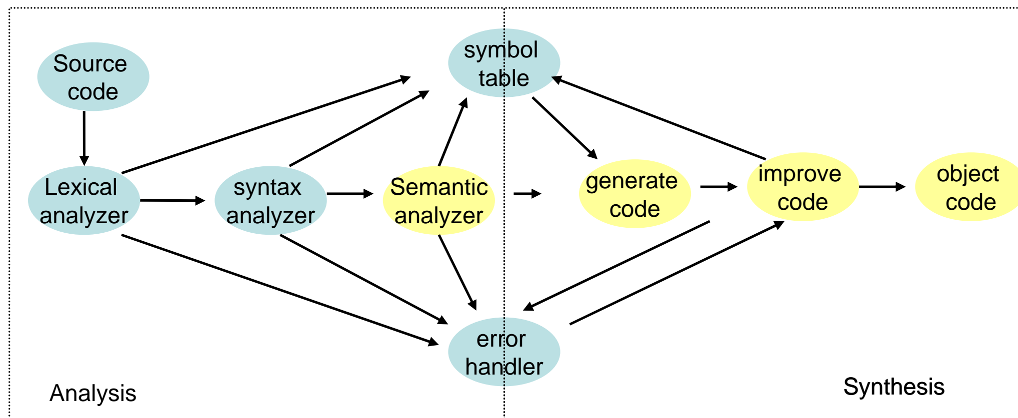


Fig. 1.4. Parse tree for `position := initial + rate * 60`.

1. Any identifier is an expression
2. Any number is an expression
3. If $expression_1$ and $expression_2$ are expressions, then so are
   $expression_1$ + $expression_2$
   $expression_1$ * $expression_2$
   ($expression_1$)

The structure (phases) of a compiler



Semantic Analysis

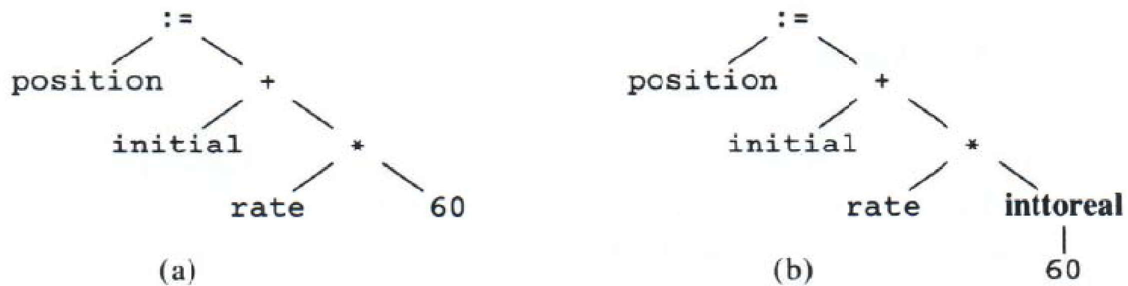
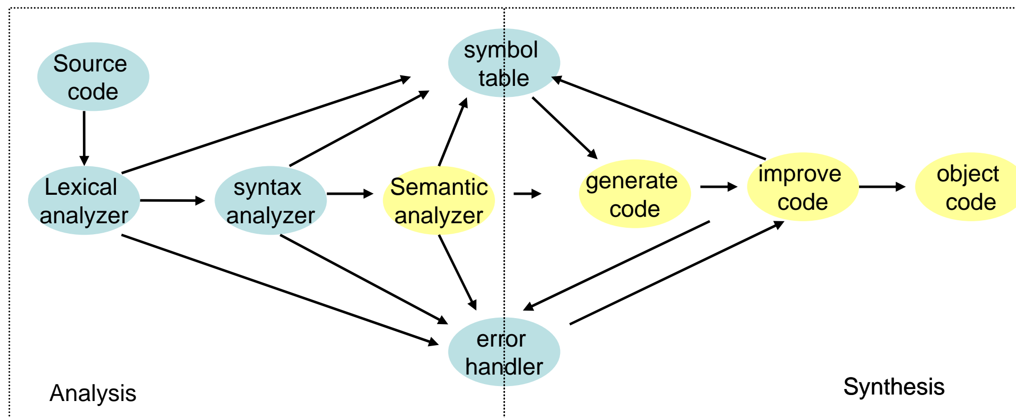


Fig. 1.5. Semantic analysis inserts a conversion from integer to real.

The structure (phases) of a compiler



Program Representation Schemes – Basic

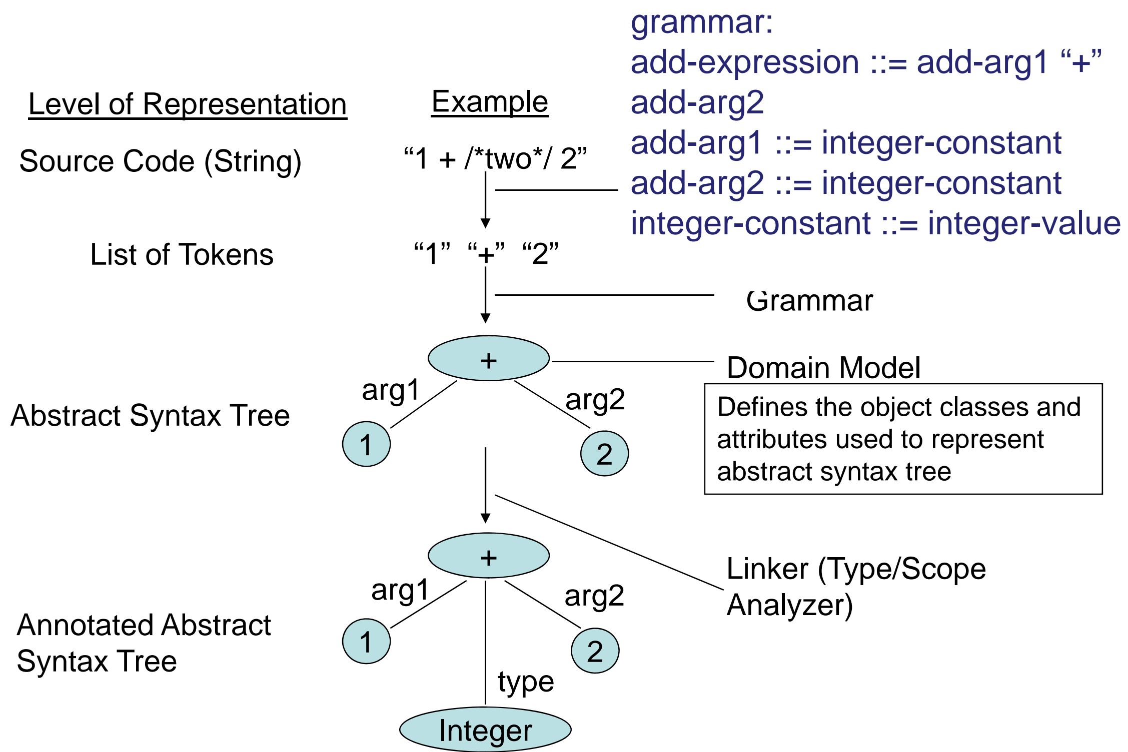
- Program representation schemes are chosen on the basis of objectives and tasks to be performed. Some popular program representation schemes are:
  - Abstract syntax trees
  - Control flow graphs
  - Data flow graphs
  - Structure charts
  - Module interconnection graphs
  - Informal information
  - Documentation

Abstract Syntax Trees

- Abstract Syntax Trees: offer a translation of an input string in terms of operands and operators, omitting superficial details such as which productions were used, syntactic properties of the input string etc
- To produce an AST:
  - Lexical Analyzer: turns input strings into tokens
  - Grammar: turns sequences of tokens into abstract syntax trees (AST)
  - Domain model: defines the nodes and the arcs that are allowable in the abstract syntax tree
  - Linker: annotations the AST nodes and arcs with global information (i.e., data types, static variables, scoping information etc.)

Abstract Syntax Trees (con't)

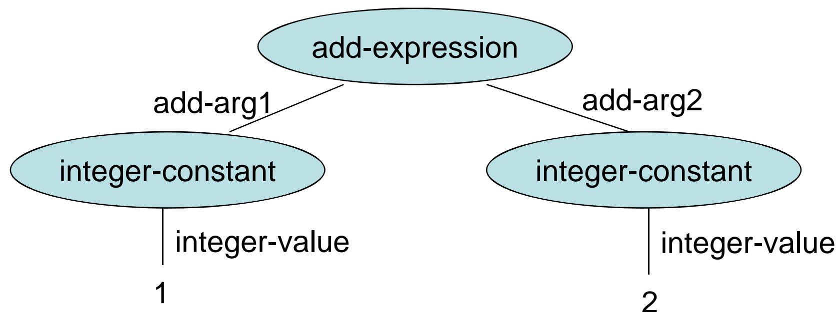
- Abstract Syntax Trees offer a good way to represent programs
  - do not emphasize on particular properties of the program
  - maintain all necessary information to generate further abstraction (i.e., Control Flow Graphs)



Example: define an example grammar for a subset of CALC, an imaginary language

grammar:
add-expression ::= add-arg1 “+” add-arg2
add-arg1 ::= integer-constant
add-arg2 ::= integer-constant
integer-constant ::= integer-value

We compile the grammar above, we can parse the text 1 + 2 to produce the abstract syntax tree:



Control Flow Graphs

- Control Flow Graphs: offer a way to eliminate variations in control statements by providing a normalized view of the possible flow of execution of a program.
  - Basic Blocks are nodes of a control flow graph
  - Arcs denote possible transfer of control from one basic block to another
- To provide a Control Flow Graph:
  - The AST of the program
  - A decomposition of a program into Basic Blocks combined with knowledge on the behavior of the control statements of the source language

Basic Block

- One of the first steps for analyzing a program is to subdivide the program into basic blocks
- A basic block is a sequence of consecutive instructions that are executed from start to finish without halt or the possibility of branching except at the end.
- A basic block can be entered at the first instruction and left at the last.
- The instruction that begins a basic block is called ***leader instruction***

Basic Block (con't)

- Once the program is subdivided into blocks, each block can be analyzed using local techniques
- Three address instructions are said to define and use (or reference) variables.
- For example, x=y+z
  – defines x and uses y and z

Three Address Instructions (example)

```
begin
   prod := 0;
   i := 1;
   do begin
       prod := prod + a[i] *b[i];
       i := i + 1;
   end
   while i<= 20
end
```

Program to compute dot product

(1) prod := 0
(2) i:= 1
(3) t_1 := 4 * i
(4) t_2 := a[t_1]
(5) t_3 := 4 * i
(6) t_4 := b[t_3]
(7) t_5 := t_2 * t_4
(8) t_6 := prod + t_5
(9) prod := t_6
(10)t_7 := i + 1
(11)i := t_7
(12)If i<= 20 goto (3)

Three-address code computing dot product

Algorithm for finding Basic Blocks

Algorithm Partition into basic blocks

Input. A sequence of three-address statements

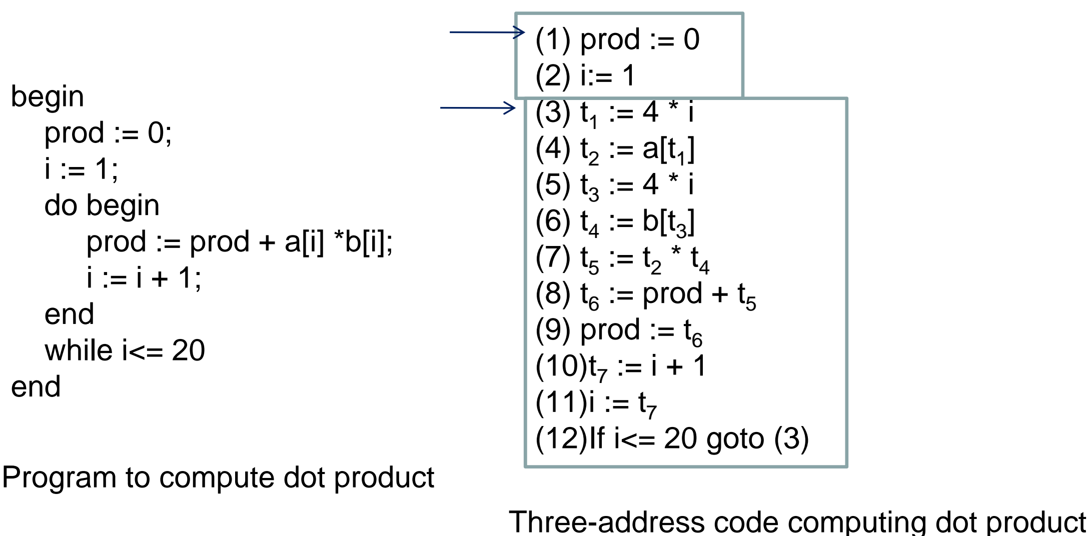
Output. A list of basic blocks with each three-address statement in exactly one block

Method.

1. We first determine the set of leaders, the first statements of basic blocks
   The rules we use are the following
   i) The ***first statement*** is a leader
   ii) Any statement that is the ***target*** of a conditional or unconditional goto is a leader
   iii) Any statement that ***immediately follows*** a goto or conditional goto statement is a leader

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program

Three Address Instructions (example)



Control Flow Graph

- A control flow graph is a directed graph that is constructed by adding flow-of-control information to the set of basic blocks making up a program.
  - The nodes of the flow graph are the basic blocks.
  - The node that contains a leader as the program's first statement is called ***initial node***

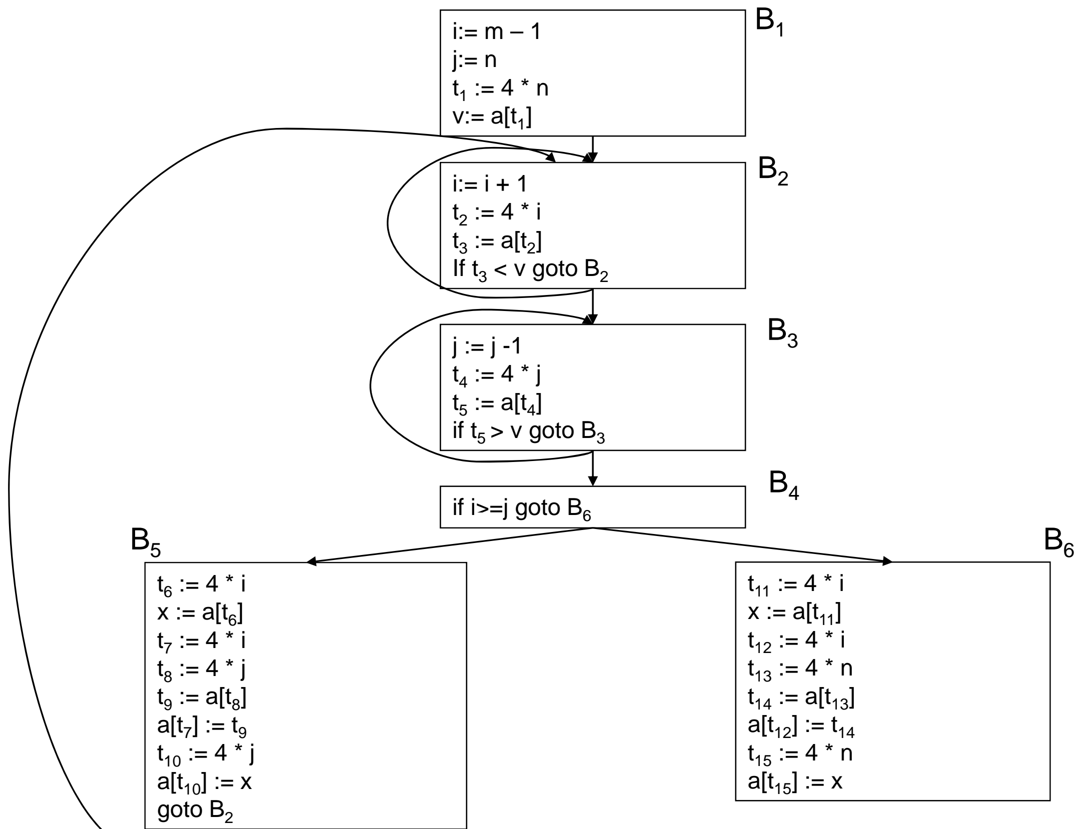
Control Flow Graph (con't)

- There is a directed edge from block B_1 to block B_2 if B_2 can immediately follow B_1 in some execution sequence; that is if:
  - There is a conditional or unconditional jump from the last statement of B_1 to the first statement of B_2 or
  - B_2 immediately follows B_1 in the order of the program and B_1 does not end in an unconditional jump.
  - We say that B_1 is a ***predecessor*** of B_2, and B_2 a ***successor*** of B_1

Example Control Flow Graph

```
void quicksort(m, n)
int m, n;
{
    int i, j, v, x;
    if ( n <= m) return;
    /* fragment begins here */
    i = m -1; j=n; v= a[n];
    while (1) {
            do i=i+1; while (a[i] < v);
            do j=j-1; while  (a[j] > v);
            if ( i>=j) break;
            x = a[i]; a[i] = a[j]; a [j] = x;
}
    x = a[i]; a[i] = a[n]; a[n] = x;
    /* fragment ends here */
    quicksort(m,j); quicksort(i+1, n);
}
```

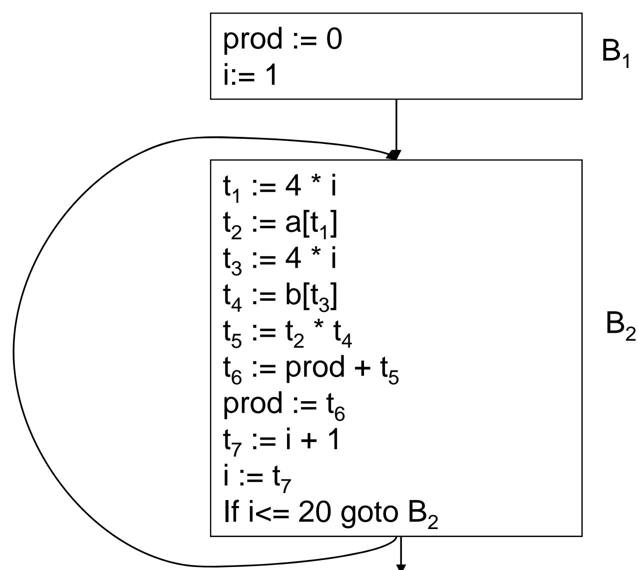
(1)	i:= m – 1	(16)	t_7 := 4 * i
(2)	j:= n	(17)	t_8 := 4 * j
(3)	t_1 := 4 * n	(18)	t_9 := a[t_8]
(4)	v:= a[t_1]	(19)	a[t_7] := t_9
(5)	i:= i + 1	(20)	t_{10} := 4 * j
(6)	t_2 := 4 * I	(21)	a[t_{10}] := x
(7)	t_3 := a[t_2]	(22)	goto (5)
(8)	If t_3 < v goto (5)	(23)	t_{11} := 4 * i
(9)	j := j -1	(24)	x := a[t_{11}]
(10)	t_4 := 4 * j	(25)	t_{12} := 4 * i
(11)	t_5 := a[t_4]	(26)	t_{13} := 4 * n
(12)	if t_5 > v goto (9)	(27)	t_{14} := a[t_{13}]
(13)	if i>=j goto (23)	(28)	a[t_{12}] := t_{14}
(14)	t_6 := 4 * i	(29)	t_{15} := 4 * n
(15)	x := a[t_6]	(30)	a[t_{15}] := x



Loops

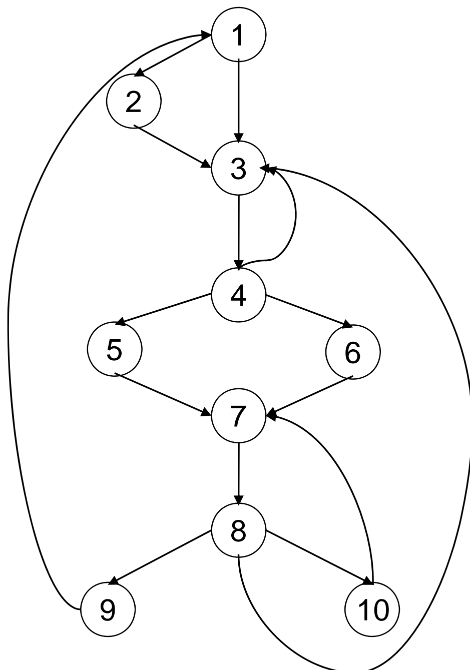
- In a flow graph we may have loops. In general a loop is defined as a collection of nodes in a flow graph such that:
  - All nodes in the collection are strongly connected; that is, from any node in the loop to any other, there is a path of length one or more, wholly within the loop
  - The collection of nodes has a unique entry, that is a node in the loop such that the only way to reach a node of the loop from a node outside the loop is first to through the entry

Loops (con't)



Dominators

- We say that a node d of a flow graph dominates node n
  - Written d dom n,
  - If every path from the initial node of the flow graph to n goes through d.
  - Under this definition every node dominates itself, and the entry of a loop dominates all nodes in the loop
- A useful way to represent dominator information is in a tree, called the dominator tree in which
  - The initial node is the root
  - Each node d dominates only its descendants in the tree.



Dominators (con't)

- The existence of dominator trees follows from the property of dominators:
  - Each node has a unique immediate dominator m that is the last dominator of n on any path from the initial node to n.
  - In terms of the dom relation, the immediate dominator m has the property that if $d \neq n$ and d *dom* n then d *dom* m

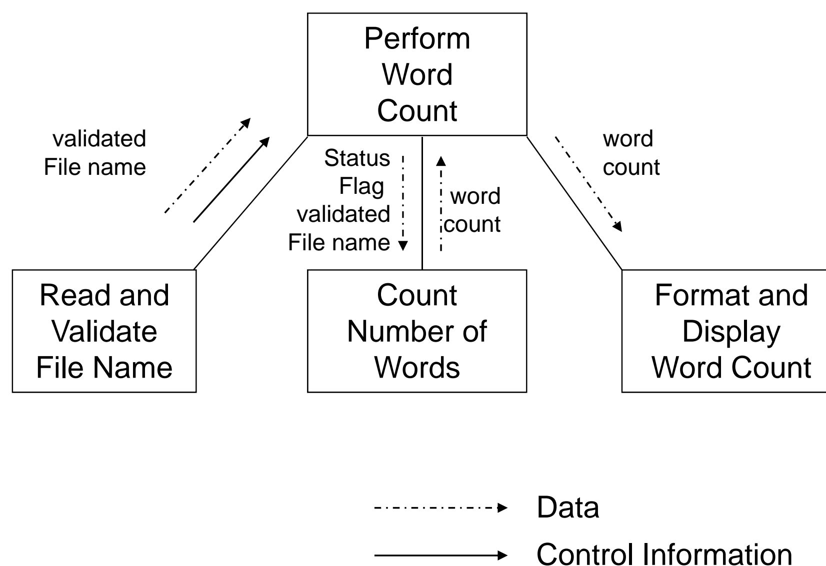
Data Flow Graphs

- Data Flow Graphs: offer a way to eliminate unnecessary control flow constraints, focusing mostly on the exchange of information between program components (i.e., basic blocks, functions, procedures, modules)
- To produce a Data Flow Graph:
  - The AST of the program
  - A decomposition of a program into Basic Blocks
  - Combined with annotations on uses and definitions of variables

Structure Charts

- Structure Charts: offer a way to represent data and control information in a concise and compact form.
  - Nodes represent program or system entities (i.e., basic blocks, procedures, modules)
  - Arcs are annotated with control and data flow information
- To produce a Structure Chart:
  - The Control Flow Graph of a program
  - The Data Flow Graph of the program
  - Representations from which you can compute the inter-procedural data flow

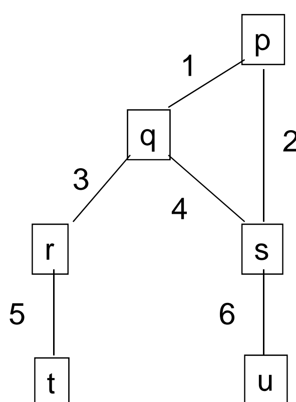
Structure Charts (con't)



Module Interconnection Graphs

- Module Interconnection Graphs: offer a way to represent data coupling and data dependencies between program and system entities
- To produce a Module Interconnection Graph:
  - The Structure Chart of a program
  - Some information of parameter passing between procedures or functions

Module Interconnection Graphs (con't)



Number	In	Out
1	Aircraft type	Status flag
2	--	List of aircraft parts
3	Function code	--
4	--	List of aircraft parts
5	Part number	Part manufacturer
6	Part number	Part name

Module Interconnection Diagram

Interface Description

p, t, u access the same database in update mode

Call Graphs

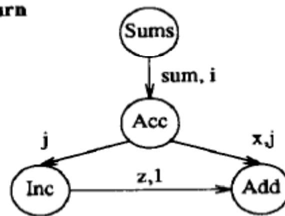
- A graph in which
  - Nodes represent individual procedures
  - Edges represent call sites
  - Each edge is labeled with the actual parameters associated with that call site

```
1. program Sums
2.   read(n);
3.   i := 1;
4.   while i ≤ n
5.     sum := 0;
6.     Acc(sum,i);
7.     write(sum,i);
8.     i := i + 1;
9.   endwhile
10. end.

11. procedure Acc(x,y)
      ref x, y
12.   j := 1;
13.   while j≤y
14.     Add(x,j);
15.     Inc(j);
16.   endwhile
17. return

18. procedure Inc(z)
      ref z
19.   Add(z,1);
20. return

21. procedure Add(a,b)
      ref a; value b
22.   a := a + b;
23. return
```



Call Graphs (con't)

- A call graph is a multi-graph with more one edge connecting two nodes in the case of one procedure calling another at many points
- A call graph represents the procedural structure of a grogram and illustrates the calling relationship among procedures, it is useful during maintenance for program understanding
- A call graph is also useful for interprocedural data flow analysis
  - Interprocedural data flow analysis algorithms that use a call graph are flow insensitive since they do not consider the control flow of individual procedures

Program Summary Graphs

- Program Summary Graphs (PSG) are modifications of the call graphs by taking into account of control and data flow in individual procedural call.
- A PSG contains information about actual reference parameters and global variables at call sites, and formal reference parameters and global variables at procedure entry and exit points
- There are two kinds of edges in the PSG:
  - ***Binding edges*** that relate actual and formal reference parameters and
  - ***Reaching edges*** that summarize the flow of data between procedural control points such as entry, exit, call and return for formal and actual parameters

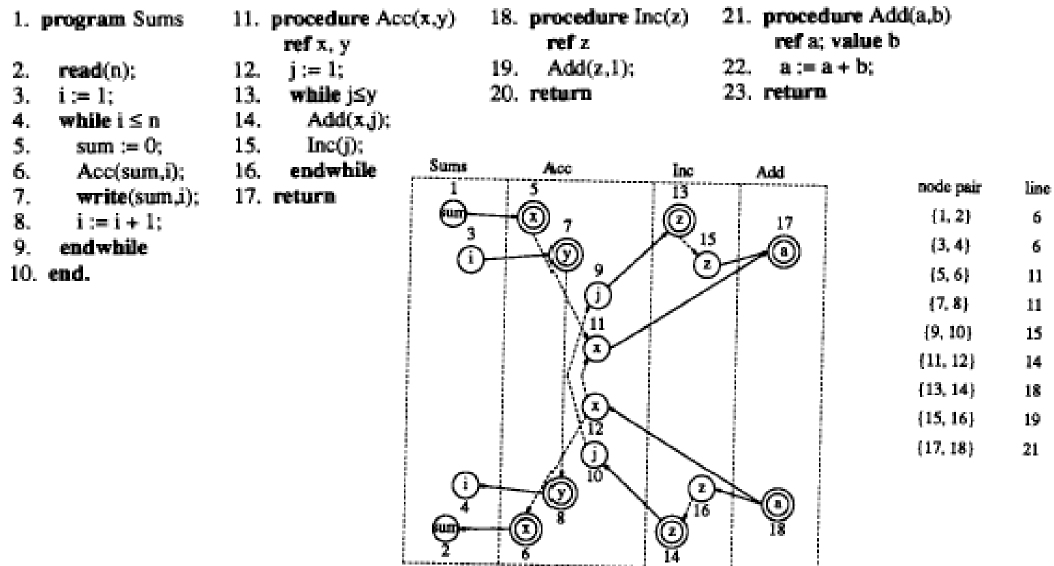


Fig. 3. The program summary graph for *Sums* in Fig. 2. Circles represent call/return nodes, double circles represent entry/exit nodes, solid lines represent call/return binding edges and dashed lines represent reaching edges. Node numbers facilitate presentation and do not correspond to the line numbers of Fig. 2. The correlation between node pairs and lines is shown in the table to the right of Fig. 3.

Data Flow Analysis

- Data flow analysis provides the framework for analyzing the behavior of programs in terms of their data flow properties
- Data flow analysis problems can be solved by setting and solving sets of equations involving data flow properties of a program at the control flow graph (CFG) basic block level
- We will be looking for the following data flow analysis problems
  – Reaching Definitions
  – Use-Def Chains
  – Available Expressions
  – Live Variable Analysis
  – Copy Propagation
  – Inter-procedural analysis of changed variables

Data Flow Analysis Equations

- Data Flow information can be collected by setting and solving systems of equations of the form:

$$Out(S) = gen(S) \cup (in(S) - kill(S))$$

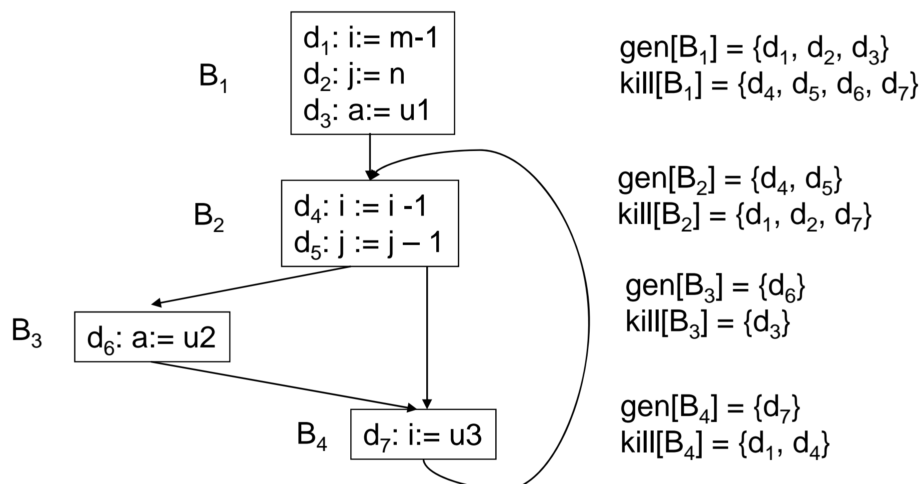
With the interpretation

– "The information at the end of a block is either generated within the block or enters at the beginning and is not killed as control flows through the block"

Data Flow Analysis Equations (con't)

- Consider the data flow equation:
  out(S) = gen(S) U (in(S) – kill(S))
- **Gen(S):** A definition *d* is in Gen(s) if *d* reaches the end of S via a path that does not go outside of S
- **Kill(S):** Is the set of definitions that never reach the end of S. In order for *d* to be in Kill(S), every path from the beginning to the end of S must have an unambiguous definition of the same variable defined by *d*, and if *d* appears in S, then following every occurrence of d along any path, must be another definition of the same variable
- **In(S):** The information flowing in the block
- **Out(S):** The information exiting the block

Example Data Flow Analysis Equations

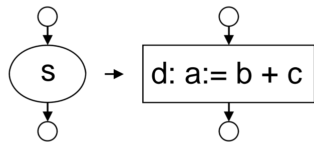


$gen[B_1] = \{d_1, d_2, d_3\}$
$kill[B_1] = \{d_4, d_5, d_6, d_7\}$

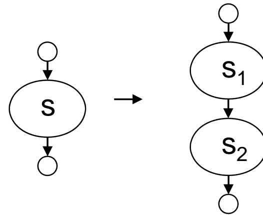
$gen[B_2] = \{d_4, d_5\}$
$kill[B_2] = \{d_1, d_2, d_7\}$

$gen[B_3] = \{d_6\}$
$kill[B_3] = \{d_3\}$

$gen[B_4] = \{d_7\}$
$kill[B_4] = \{d_1, d_4\}$

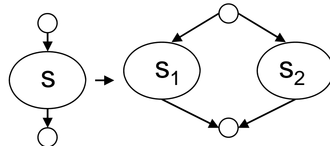


gen[S] = {d}
kill[S] = D_a *– {d}*
out[S] = gen[S] U (in[S] – kill[S])



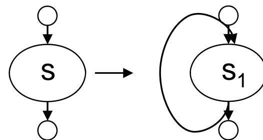
$gen[S] = gen[S_2]\ U\ (gen[S_1] - kill[S_2])$
$kill[S] = kill[S_2]\ U\ (kill[S_1] - gen[S_2])$

$in[S_1] = in[S]$
$in[S_2] = out[S_1]$
$out[S] = out[S_2]$



$gen[S] = gen[S_1]\ U\ gen[S_2]$
$kill[S] = kill[S_1] \cap kill[S_2]$

$in[S_1] = in[S]$
$in[S_2] = in[S]$
$out[S] = out[S_1]\ U\ out[S_2]$



$gen[S] = gen[S_1]$
$kill[S] = kill[S_1]$
$in[S_1] = in[S]\ U\ gen[S_1]$
$out[S] = out[S_1]$

Solution of Data Flow Equations

- The most common way to solve Data Flow Equations is via iterative solutions
- Iterative solutions to Data Flow equations require:
  - Build the Control Flow Graph
  - Compute, In, Out for each Basic Block
- There are two types of equations:
  - Forward equations: Out sets are computed in terms of In sets
  - Backward equations: In sets are computed in terms of Out sets

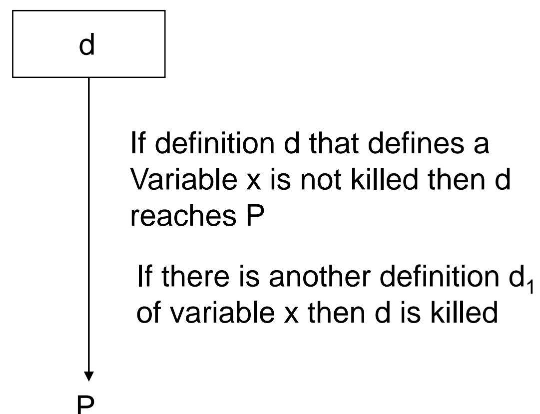
ELEC 876: Software Reengineering (Program Comprehension)

Dr. Ying Zou

Department of Electrical & Computer Engineering
Queen's University

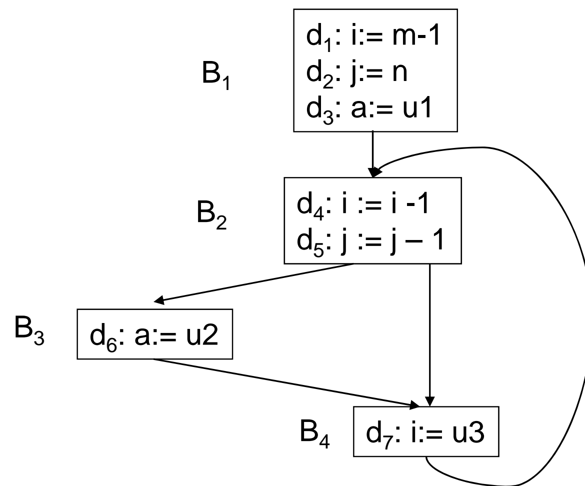
Reaching Definitions

- The idea behind ***reaching definitions*** is that given a program point p find all definitions d, (i.e., read, or assignment statements of variables) that reach p



```
int g(int m, int i);

int f(n)
	int n;
{
	int i = 0, j;
	if (n == 1) i = 2;
	while (n > 0) {
		j = i + 1;
		n = g(n, i);
	}
	return j;
}
```



Iterative Algorithm for Reaching Definitions

- For each basic Block b we can define In(B), Out(B), Gen(B), Kill(B) as follows:

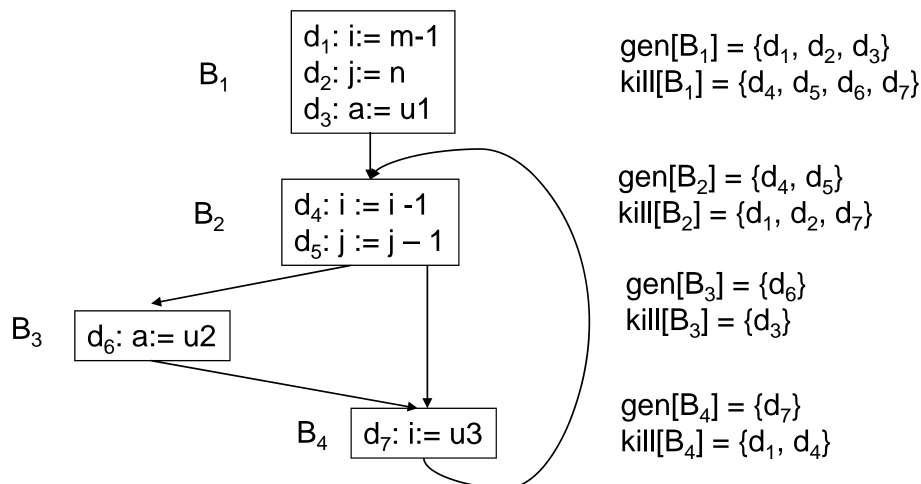
$$In(B) = U_p Out(P)$$

  - Where P is a predecessor of B
- If a flow graph has n nodes we get 2n equations
- The iterative algorithm for Reaching Definition is give in next slide

Input. A flow graph for which *kill*[B] and *gen*[B] have been computed for each block B.
Output. *In* [B] and *out*[B] for each block B
Method. We use an iterative approach, starting with the "estimate" *in*[B] = ∅ for all B and converging to the desired values of in and out. As we must iterate until the *in*'s (and hence the out's) converge, we use a boolean variable change to record on each pass through the blocks whether any *in* has changed.

```
/* initialize out on the assumption in[B] = ∅ for all B */
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] := U_{p is a predecessor of B} out[P];
           oldout := out[B];
         out[B] := gen[B] U (in[B] – kill[B])
         If out[B] ≠ oldout then change := true
      end
end
```

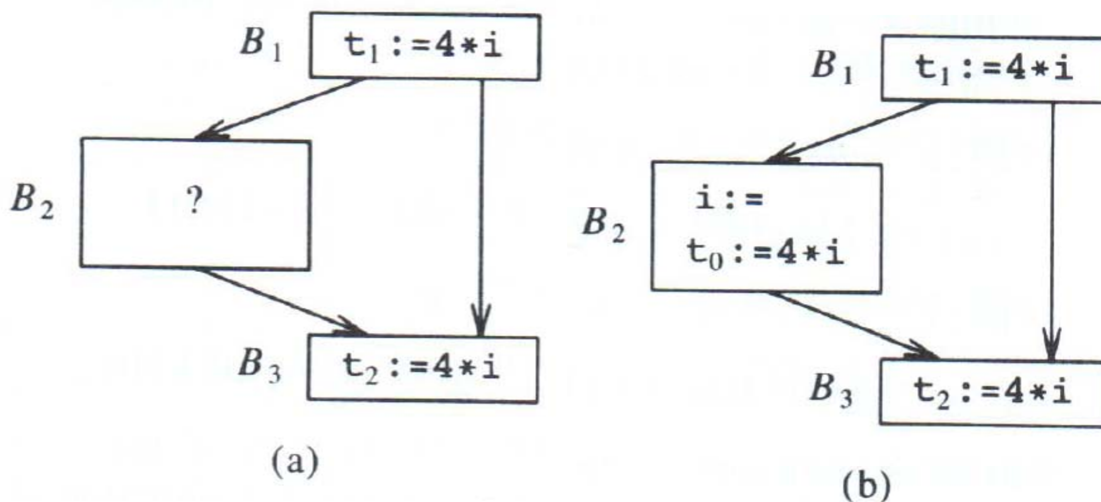
Algorithm to compute *in* and *out*



BLOCK B	Initial		Pass 1		Pass 2	
	in[B]	*out*[B]	*in*[B]	*out*[B]	*in*[B]	*out*[B]
B1	000 0000	111 0000	000 0000	111 0000	000 0000	111 0000
B2	000 0000	000 1100	111 0001	001 1100	111 0111	001 1110
B3	000 0000	000 0010	001 1100	000 1110	001 1110	000 1110
B4	000 0000	000 0001	001 1110	001 0111	001 1110	001 0111

Available Expressions

- An expression *E* (i.e, x+y) is ***available*** at a program point P, if every path from the initial node to point P, evaluates *E*, and after the last such evaluation prior to reaching P, there are no subsequent assignments to the components of *E* (i.e. x or y for x+y)
- For available expressions we say that a block B ***kills*** an expression *E* (i.e. x+y) if it assigns one or more of its components (i.e. x or y) and does not subsequently redefines *E*
- A block B ***generate*** expression *E* if it definitely evaluates E and does not subsequently redefines one or more of the components of *E* (i.e. x or y)
- For the calculation of the available expressions we assume *U* be the universal set of all expressions appearing in a program *A*



STATEMENTS	AVAILABLE EXPRESSIONS
....	none
a := b+c	
....	only b+c
b := a-d	
....	only a-d
c := b+c	
....	only a-d
d := a-d	
....	none

Available Expression Algorithm

Input. A flow graph G with *c_kill*[B] and *c_gen*[B] computed for each block B. The initial block is B_1.
Output. The set in[B] for each block B.
Method.

```
in[B1] := ∅
/* in and out never change for the initial node, B1 */
out[B1] := c_gen[B1];
/* initial estimate is too large */
for B ≠ B1 do out[B] := U – c_kill[B]
change := true;
while change do begin
        change := false
        for B ≠ B1 do begin
              in[B] := ∩ out[P]
              oldout := out[P]
                 out[B] := c_gen{B] ∪ (in[B] – c_kill[B]);
              if out[B] ≠ oldout then change := true
        end
end
```

Live Variable Analysis

- Live Variable analysis is a typical analysis that depends on information that is computed in the direction opposite to the flow of control in a program
- In Live Variable analysis we wish to know for variable x and point P, whether the value of x at P, could be used along some path in the flow graph starting at P. In other words is x used after P?

Live Variable Analysis (con't)

- We formulate this Data Flow analysis problem with the following set of equations:

$$In(B) = Use(B) \cup (Out(B) - Def(B))$$

$$Out(B) = \cup_s In(S)$$

Where:

- S: is a successor basic block of B
- In(B): is the set of live variables at the beginning of block B
- Out(B): is the set of live variables when existing the block B
- Def(B): is the set of variables, definitely assigned values in B prior to any use of that variable in B
- Use(B): is the set of variables whose values may be used in B prior to any definition of the variable

Live Variable Analysis Algorithm

Input. A flow graph with *def* and *use* computed for each block.
Output. *out*[B], the set of variables live on exit from each block B of the flow graph
Method.

for each block B do *in*[B] := ∅
while changes to any of the *in*'s occur do
    for each block B do begin
        out[B] = ∪ *in*[S]
        in[B] := *use*[B] ∪ (*out*[B] – *def*[B])
    end

Def-Use Chains

- The Du-Chaining problem is to compute for a point P the set of uses S, of a variable x, such that, there is a path from P to S, that does not redefine x
- *DU-Chain*: A list of possible uses of a definition
- A variable is used in a statement if its value is required
- The DU-chaining problem can be formulated as Data Flow equations as follows:

In(B)=UpwdExpposseduses(B) ∪ (out(B) – NewDefs(S,x))

$Out(B) = \cup_{S} In(S)$

where S is a successor of B

- *UpwdExposedUses(B):* The set of pairs (S, x), such that, S is a statement in B, which uses x, and such that no prior definition of x occurs in B
- *NewDefs(B):* The set of pairs (S, x), such that, S is a statement which uses x, S is not in B, and B has a definition of x

Copy Propagation

- It is sometimes possible to eliminate copy statements S (I.e. x := y) if we determine all places where this definition is used. We may then substitute y for x, provided the following conditions are met by every such use u of x
  1. Statement S must be the only definition of x reaching u (that is ud-chain for use u consists only of S)
  2. On every path from S to u, including paths that go through u several times (but ***do not go through S a second time***), there are ***no assignments to y***
- Condition (1) can be checked using ud-chaining information
- For Condition (2) we define a new Data Flow analysis problem

Copy Propagation (con't)

- Let In(B) be the set of copies (I.e. S: x:=y) such that every path from the initial node to the beginning of B, contains the statement S, and subsequent to the last occurrence of S, there are no assignments to y
- Let Out(B) be defined correspondingly but with respect to the end of B
- We say that the copy statement S x:=y, is ***generated*** in block B, if it occurs in B, and there is ***no subsequent assignment of y*** within B
- We say that the copy statement S x:=y, is killed in B, if x or y is assigned and S is not in B
- Notes:
  - That an assignment to x kills x:=y, is similar to the reaching definitions problem
  - That ***y*** to do so is special to this problem
  - In(B) can contain only one copy statement with x on the left
  - Different assignment x:=y kill each other

Copy Propagation (con't)

- Let U be the universal set of all copy statements in a program. It is important to note that different copy statement x:=y are different entries in U
- Lets define:
  - C_gen(B) to be all copies generated in block B
  - C_kill(B) to be all copies in U but killed in block B
- The Copy propagation equations are formulated as follows:
  - Out(B) = c_gen(B) U (in(B) – c_kill(B))
  - In(B) = $\cap_p$ out(P) for B not initial, where P is a predecessor block of B
  - $In(B_1) = \varnothing$

Example: Consider the flow graph of the figure.

Here, $c_gen[B_1] = \{x:=y\}$ and $c_gen[B_3] = \{x:=z\}$. Also, $c_kill\{B_2\} = \{x:= y\}$. since y is assigned in B_2, Finally, $c_kill[B_1] = \{x:=z\}$ since x is assigned in B1 and c_kill[B3]={x:=y} for the same reason.

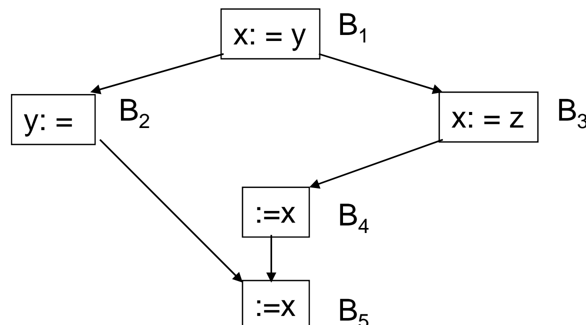
The other c_gen's and c_kill's are $\varnothing$. Also, $in[B_1]$ = by equation. Algorithm in one pass determines that

$$in[B_2] = in[B_3] = out[B_1] = \{x:=y\}$$

Likewise, $out[B_2] = \varnothing$ and

$$out[B_3] = in[B_4] = out[B_4] = \{x:=z\}$$

Finally, $in[B_5] = out[B_2] \cap out[B_4] = \varnothing$



Copy Propagation Example (con't)

- From the previous example we observe:
  - Neither copy x=y nor x=z reaches the use of x in B5
  - It is not possible to substitute y (respectively z) in all uses of x that definition x = y (respectively x = z) reaches
  - We could have substituted x for z in B4

Copy Propagation Algorithm

Input. A flow graph G with ud-chains giving the definitions reaching block B, and which c_in[B] representing the solution to, that is, the set of copies x:= y that reach block B along every path, with no assignment to x or y following the last occurrence of x := y on the path. We also need du-chains giving the uses of each definition.

Output. A revised flow graph

Method. for each copy s: x:= y do the following

1. determine those uses of x that are reached by this definition of x, namely, s: x:= y.
2. determine whether for every use of x found in (1) s in in c_in[B], where B is the block of this particular use,, and moreover, no definitions of x or y occur prior to this use of x within B. Recall that if s is in c_in[B], then s is the only definition of x that reaches B.
3. if s meets the conditions of (2), then remove s and replace all uses of x found in (1) by y.