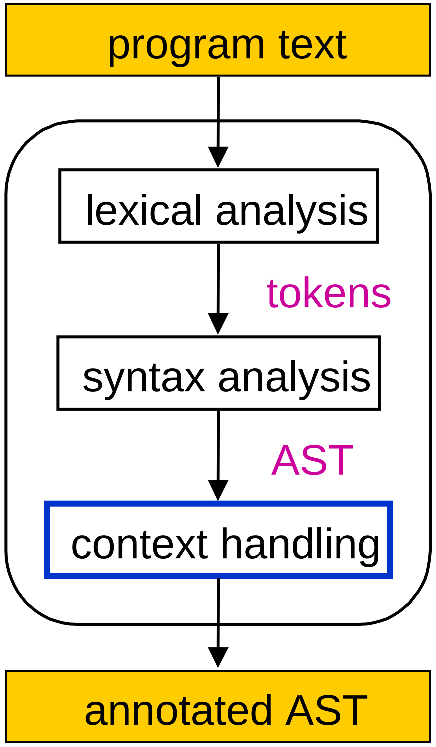
Compiler construction
in4303 – lecture 7

AST processing
manual methods

Chapter 3.2

Overview

- context handling
- annotating the AST
  - attribute grammars
  - manual methods
- symbolic interpretation
- data-flow equations



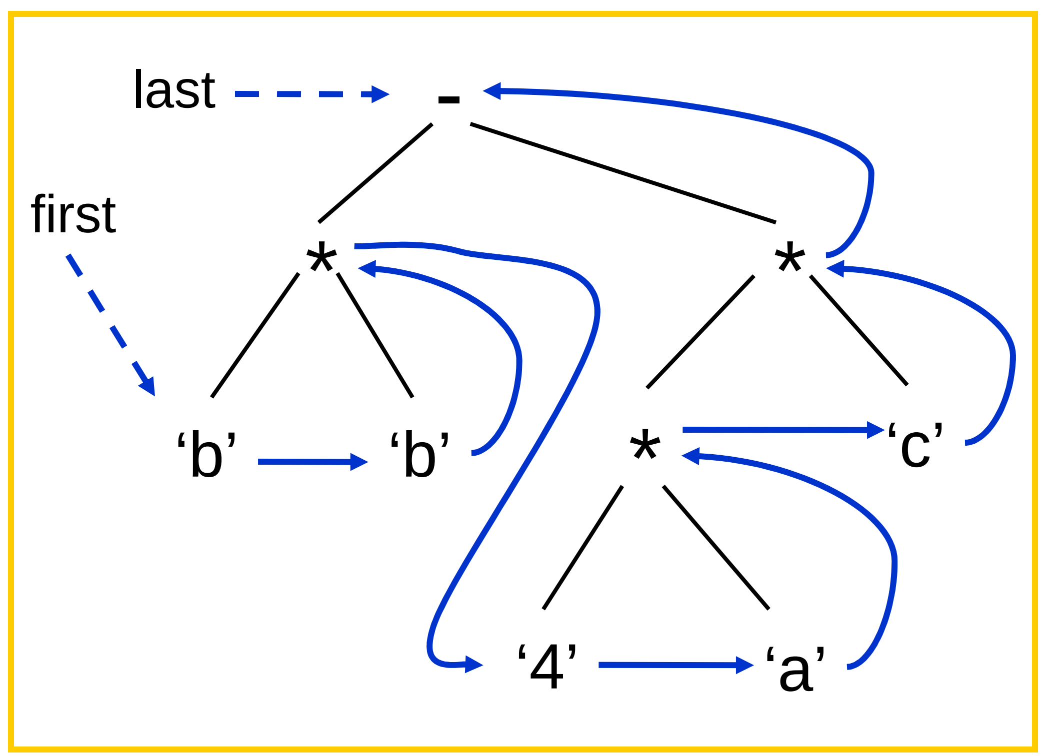
Manual methods for analyzing the AST

- preparing the grounds for code generation
  - constant propagation
  - last-def analysis (reaching definitions)
  - live analysis
  - common subexpression elimination
  - dead-code elimination
  - ...
- we need flow-of-control information

Threading the AST

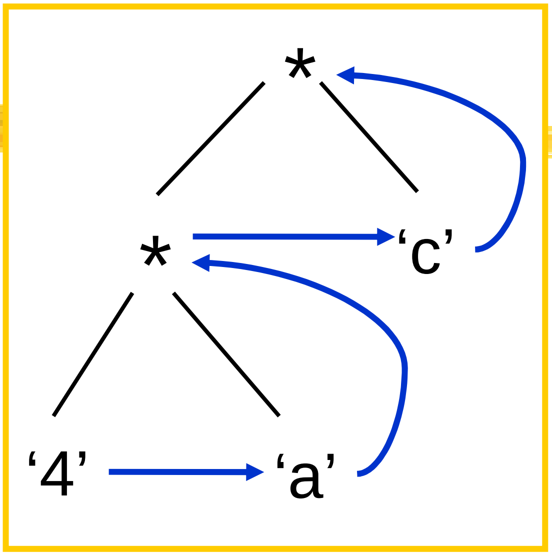
- determine the control flow graph that records the successor(s) of AST nodes
- intermediate code

  PUSH b
  PUSH b
  MUL
  PUSH 4
  PUSH a
  MUL
  PUSH c
  MUL
  SUB



Threading the AST

- result is a post-order traversal of the AST
- global variable: Last node pointer

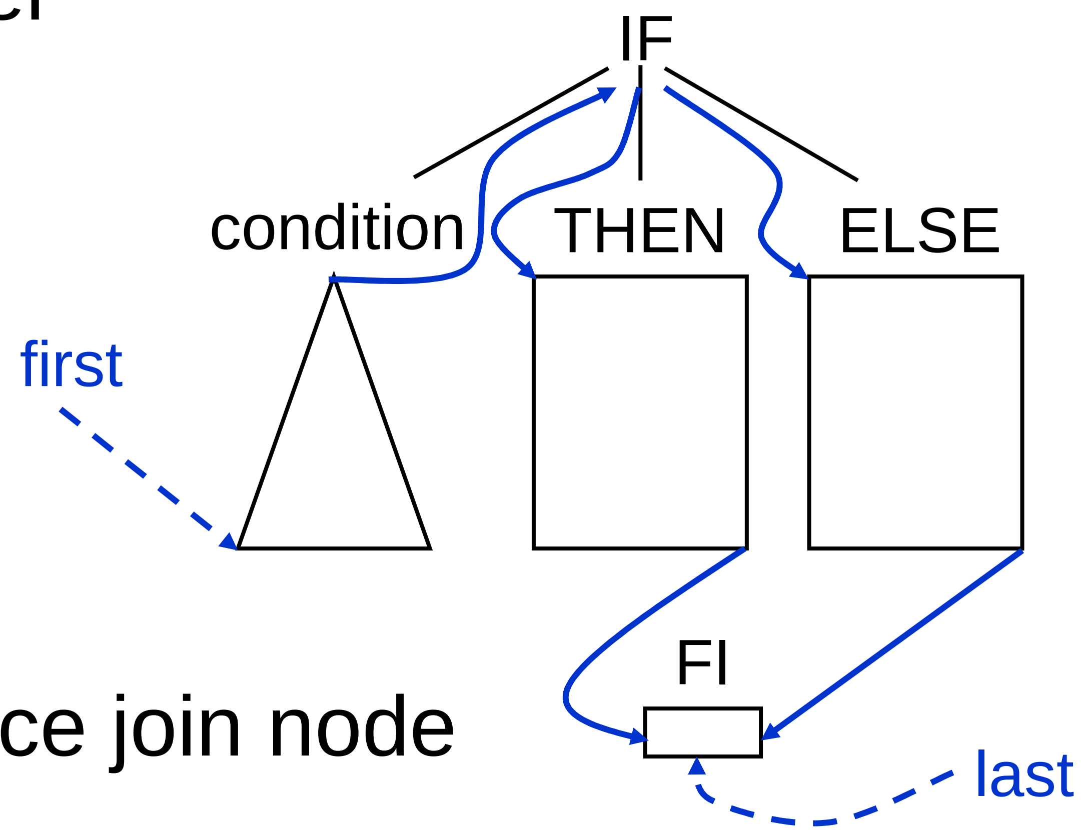



```
PROCEDURE Thread binary expression (Expr node pointer):
    Thread expression (Expr node pointer .left operand);
    Thread expression (Expr node pointer .right operand);

    // link this node to the dynamically last node
    SET Last node pointer .successor TO Expr node pointer;
    // make this node the new dynamically last node
    SET Last node pointer TO Expr node pointer;
```

Multiple successors

- problem: threading is built around single Last node pointer

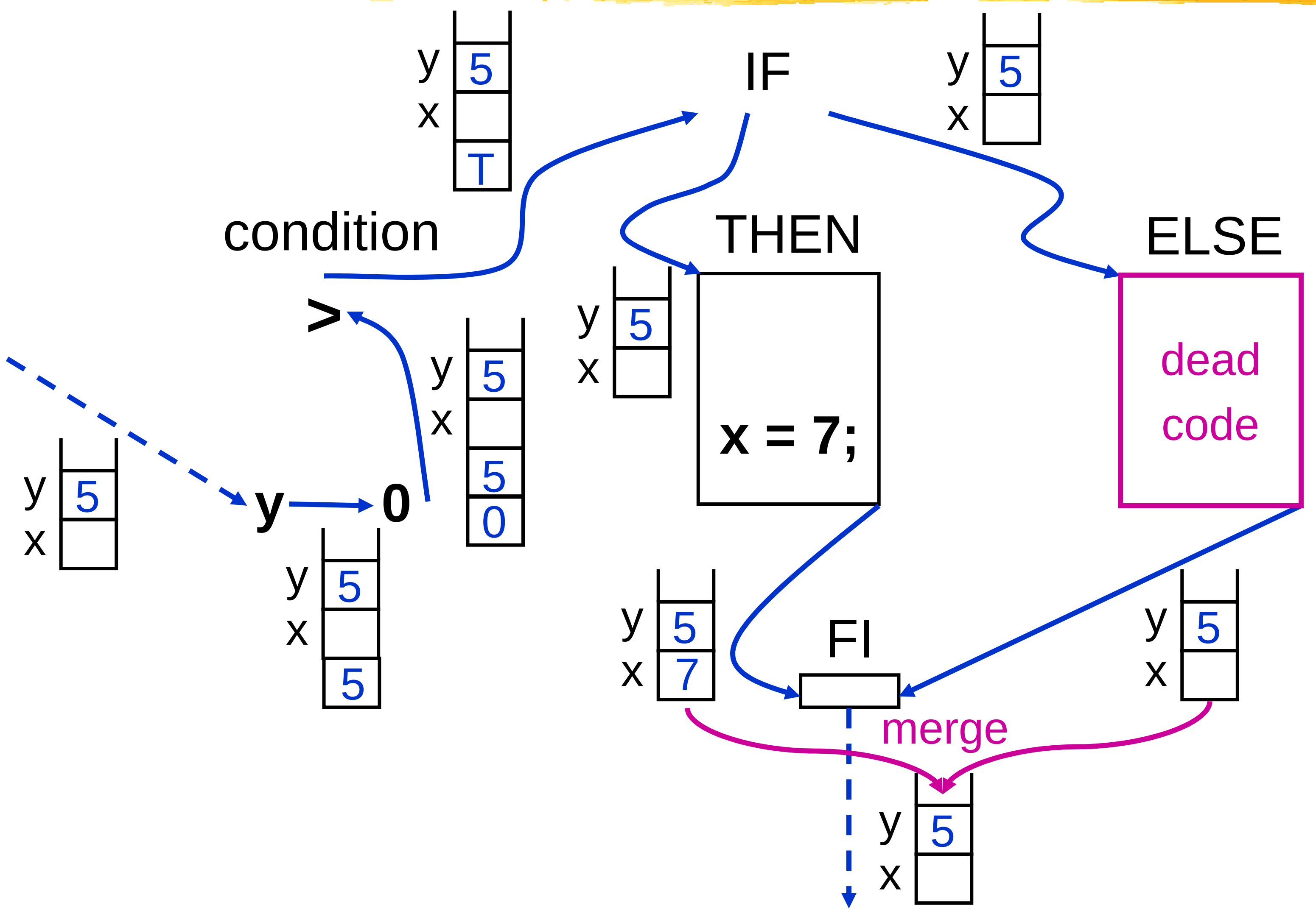


- solution: introduce join node

Symbolic interpretation

- behaviour of code is determined by (the values of) variables
- simulate run-time execution at compile time
- attach a stack representation to each arrow in the control flow graph
- an entry summarizes all compile-time knowledge about the variable/constant

Symbolic interpretation

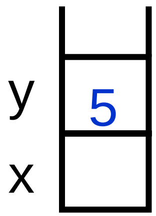


Exercise (5 min.)

- draw the control flow graph for

  while C **do** S **od**

- propagate initial stack



  when C represents y>x

  and S stands for x=7

Answers

Simple symbolic interpretation

- used in narrow compiler
- simple properties + simple control-flow
- example: detecting the use of uninitialized variables

```
int foo(int n)
{  int first;

   while (n-- > 0) {
      if (glob[n] == KEY)
         first = n;
   }
   return first;
}
```

- maintain property list
- advance list through control flow graph

Tracking uninitialized variables

- variable declaration: add (ID:Uninitialized) tuple
- parameter declaration: add (ID:Initialized) tuple
- expression: check status of used variables
- assignment: set tuple to (ID:Initialized)

- control statement
  - fork nodes: copy list
  - join nodes: merge lists

merge(Init, Init) = Init
merge(Uninit, Uninit) = Uninit
merge(x, y) = Maybe

```
int foo(int n)
{  int first;

   while (n-- > 0) {
      if (glob[n] == KEY)
         first = n;
   }
   return first;
}
```

Simple symbolic interpretation

- flow-of-control structures with one entry and one exit point (no GOTOs)
- the values of the property are severely constrained (see book)
- counter example: constant propagation

Constant propagation

- record the exact value iso Initialized

```
int i = 0;
// i == 0
while (condition) {
    // i == 0
    if (i>0) printf("loop reentered\n");
    …
    i++;
    // i == 1
}
// i == ANY
```

- simple symbolic interpretation fails

Full symbolic interpretation

- maintain a property list for each entry point (label), initially set to empty
- traverse flow of control graph

L:	merge current-list into list-L
	continue with list-L
jump L;	merge current-list into list-L

- repeat until nothing changed

guarantee termination – select suitable property

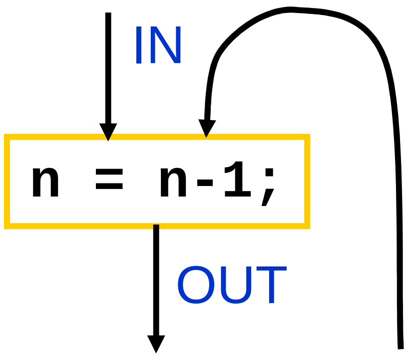
Constant propagation

- full symbolic interpretation
- property: unknown < value < any

```
int i = 0;  // i == 0
L: // i == ANY
if (condition) { // i == ANY
   if (i>0) printf("loop reentered\n");
   …
   i++; // i == ANY
   goto L;
}
 // i == ANY
```

Data flow equations

- “automated” full symbolic interpretation
- stack replaced by collection of sets
  - IN(N)
  - OUT(N)
- semantics are expressed by constant sets
  - KILL(N)
  - GEN(N)
- equations
  - $IN(N) = \cup_{M \in predecessor[N]} OUT(M)$
  - $OUT(N) = IN(N) \setminus KILL(N) \cup GEN(N)$



Data flow equations

- solved through iteration (closure algorithm)
- data-flow nodes may not change the stack
  - individual statements
  - basic blocks
- example: tracking uninitialized variables
  - properties: I x, M x, U x
  - GEN(*x* = *expr* ;) = {I *x*}
  - KILL(*x* = *expr* ;) = {U *x*, M *x*}

```
int foo(int n)
{  int first;

L: n = n-1;
   if (n >= 0) {
      if (glob[n] == KEY)
         first = n;
      goto L;
   }
   return first;
}
```

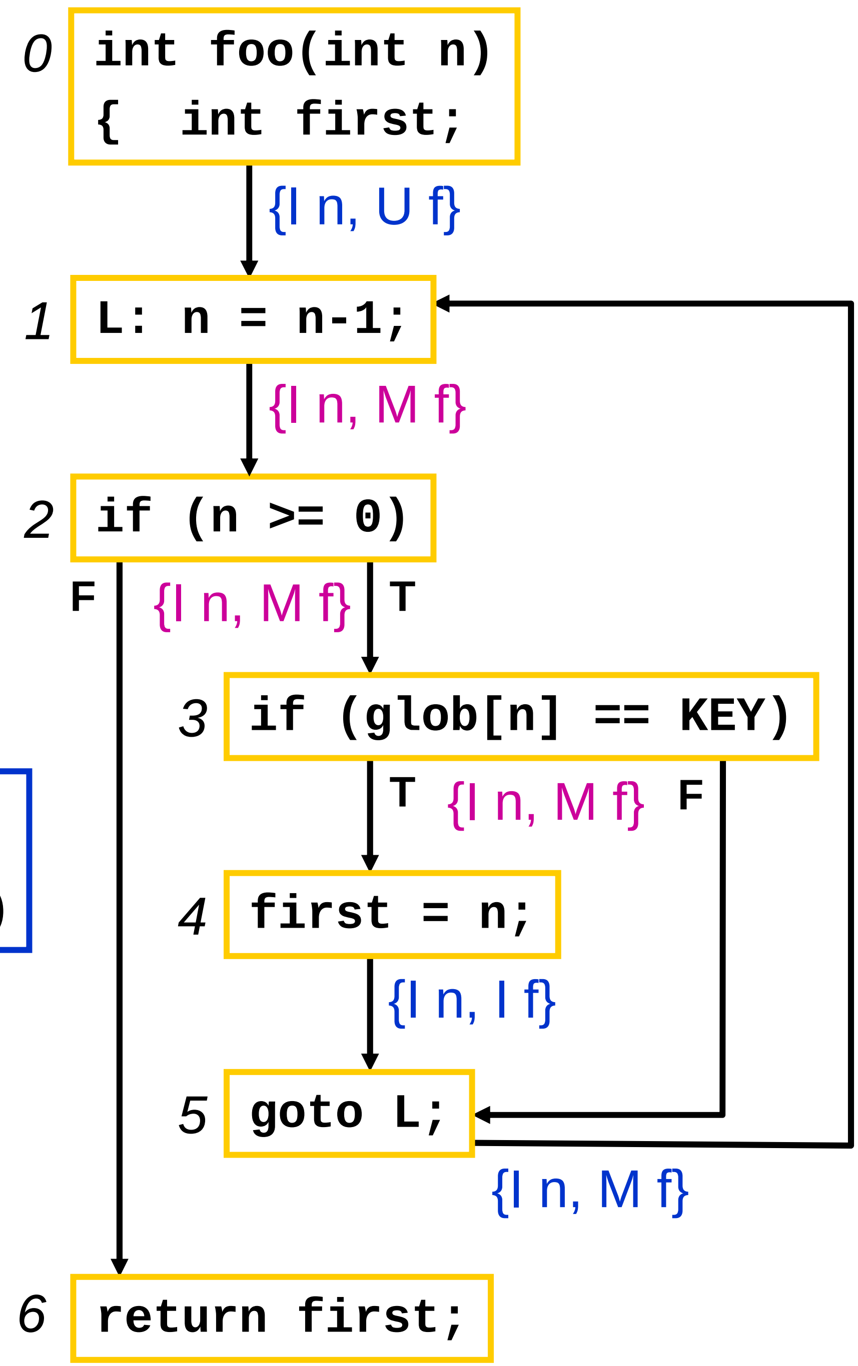
Iterative solution

- initialization: set all sets to empty
- iterate from top to bottom

$IN(N) = \cup_{M \in predecessor[N]} OUT(M)$

$OUT(N) = IN(N) \setminus KILL(N) \cup GEN(N)$

statement	KILL	GEN
0		$I_n \ U_f$
1	$U_n \ M_n$	I_n
4	$U_f \ M_f$	I_f



Iterative solution

statement			iteration 1		iteration 2	
	KILL	GEN	IN	OUT	IN	OUT
0		I_n U_f		I_n U_f		I_n U_f
1	U_n M_n	I_n	I_n U_f	I_n U_f	I_n M_f	I_n M_f
2			I_n U_f	I_n U_f	I_n M_f	I_n M_f
3			I_n U_f	I_n U_f	I_n M_f	I_n M_f
4	U_f M_f	I_f	I_n U_f	I_n I_f	I_n M_f	I_n I_f
5			I_n M_f	I_n M_f	I_n M_f	I_n M_f
6			I_n U_f	I_n U_f	I_n M_f	I_n M_f

Efficient data-flow equations

- limit to (set of) on/off properties
  - set union = bitwise OR
  - set difference = bitwise AND NOT
- combine statements into basic blocks
  - $KILL[S_1;S_2] = KILL[S_1] \cup KILL[S_2]$
  - $GEN[S_1;S_2] = GEN[S_2] \cup (GEN[S_1] \setminus KILL[S_2])$
- sort data-flow graph
  - depth-first traversal (break cycles!)

Live analysis

- a variable is live at node N if the value it holds is used on some path further down the control-flow graph; otherwise it is dead
- useful information for register allocation

- information must flow “backwards” (up) through the control-flow graph
  - difficult for symbolic interpretation
  - easy for data flow equations

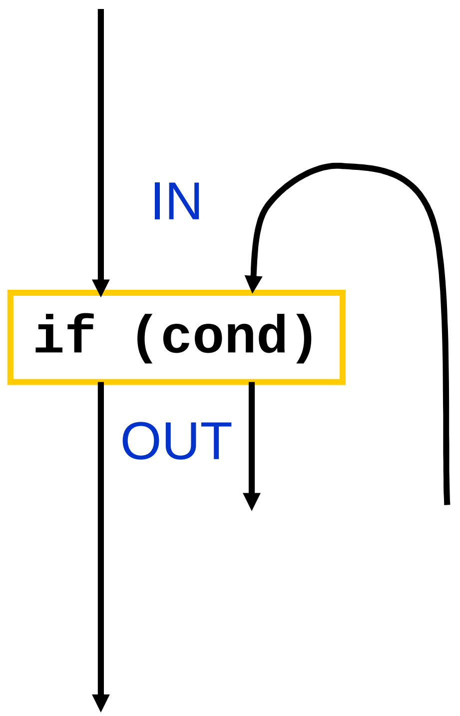
Solving data-flow equations

- forwards

$$IN(N) = \cup_{M \in predecessor[N]} OUT(M)$$
$$OUT(N) = IN(N) \setminus KILL(N) \cup GEN(N)$$

- backwards

$$OUT(N) = \cup_{M \in successor[N]} IN(M)$$
$$IN(N) = OUT(N) \setminus KILL(N) \cup GEN(N)$$



Exercise (5 min.)

- determine the KILL and GEN sets for the property "V is live here" for the following statements

statement S	KILL	GEN
$v = a \oplus b$		
v = M[i]		
M[i] = v		
$f(a_1, \ldots, a_n)$		
$v = f(a_1, \ldots, a_n)$		
if a>b **then** goto L_1 **else** goto L_2		
L:		
goto L		

Exercise (7 min.)

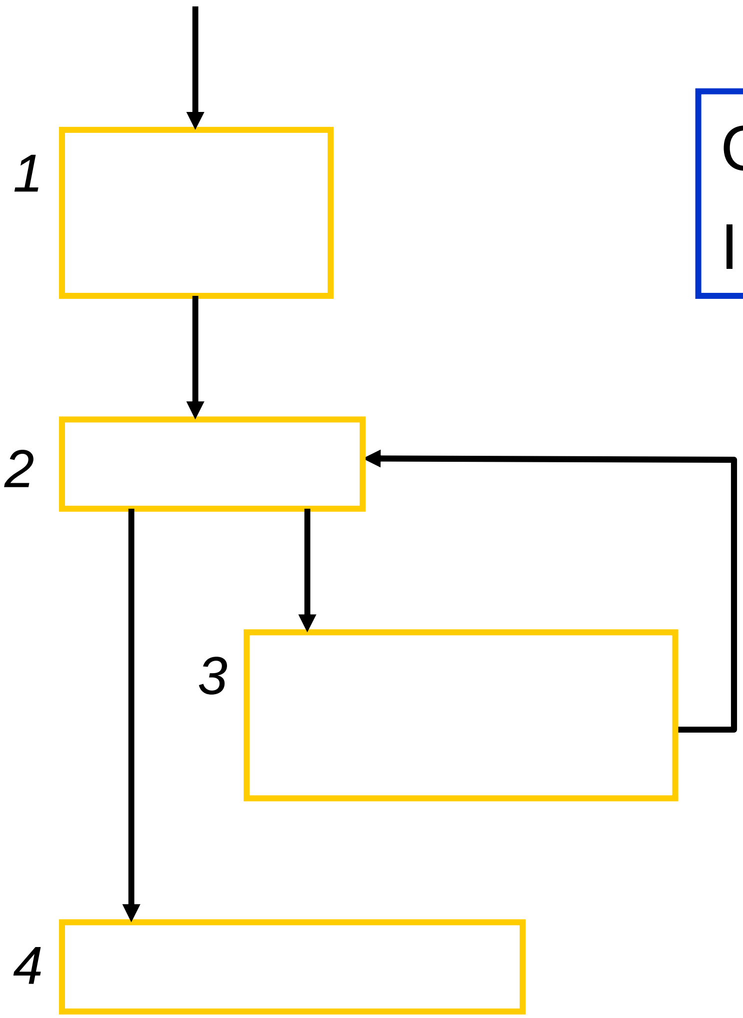
- draw the control-flow graph for the following code fragment

```
double average(int n, double v[])
{  int i;
   double sum = 0.0;

   for (i=0; i<n; i++) {
      sum += v[i];
   }
   return sum/n;
}
```

- perform backwards live analysis using the KILL and GEN sets from the previous exercise

Answers



$$OUT(N) = \cup_{M \in successor[N]} IN(M)$$

$$IN(N) = OUT(N) \setminus KILL(N) \cup GEN(N)$$

statement	KILL	GEN
1		
2		
3		
4		

Summary

manual methods for annotating the AST

- threading
- symbolic interpretation
- data-flow equations

	symbolic interpretation		data-flow equations
	simple	full	
method	stack-based simulation		IN, OUT, GEN, KILL sets
granularity	AST node		basic block
algorithm	one-pass	iterative	iterative
direction	forwards		forwards & backwards