Politecnico
di Milano

Analysis

Analysis (vs. testing)

- Does not involve the actual execution of software
- Two main approaches
  - Manual inspection/walkthrough
  - Automated static analysis
- Can be applied at any stage of the development process
- Particularly well-suited at the early stages of specification and design
  - Lack of executability makes testing impossible

Inspections - definitions

ANSI/IEEE Standard 729-1983 IEEE Standard Glossary of SE Terminology defines inspection as

"... a formal evaluation technique in which software requirements, design, or code are examined in detail by a person or group other than the author to detect faults, violations of development standards, and other problems...."

Reviews, walkthrough, inspections

- Two kinds of reviews
  - Walkthroughs
  - Inspections
- Latter more formal than first.

- Both:
  - Review is an in-depth examination of some work product by a team of reviewers.
  - Product is anything produced for the lifecycle, i.e., requirements, plans, design, code, test cases, documentation, manuals, everything!
  - No meeting is more than 2 hours.

Reviews, walkthrough, inspections

- The focus is on finding errors in the product,
  - not on correcting them, and
  - not in finding fault in producer of product.

- It is essential that it not be used for employee performance evaluation, either of producer or reviewers.

Differences

- Atmosphere:
  - Walkthroughs: informal
  - Inspections: formal
- Reviewers:
  - Walkthroughs: experts in the domain
  - Inspections: trained, professional inspectors
- Subject of review:
  - Walkthroughs: correctness of product, as seen by experts
  - Inspections: correctness of product, according to checklist of items to be examined
- Leader of discussion and session controller:
  - Walkthroughs: producer of product
  - Inspections: official moderator of review team

Walkthroughs

- The producer presents product (if code, then also its documentation) and the reviewers comment on the correctness of the product (if code, also consistency with documentation).

Software inspection: history

- Inspection technique was developed by Michael E. Fagan at IBM Kingston.
- Fagan was a certified quality engineer and studied the methods of Deming and Juran.
- He used inspection on a SW project he was managing in 72–74, in effect applying industrial hardware quality methods to SW
- It was very successful!
- He reported results in a now famous 1976 paper.
- The method became very popular in IBM, although there was some resistance.

Software inspection: history

- AT&T Bell Labs started using technique in 1977.
- In 1986, a major Bell Labs SW development organization with 200 people reported its experience with inspections:
  - 14% productivity increase for a single release
  - better tracking and phasing
  - early defect density data improved 10-fold
  - staff credited inspection as an ‘‘important influence on quality and productivity’’

Inspections

- Fagan's Code Inspections

- Roles
- Checklists
- Steps
- Users at Inspections

Software Inspection Roles

- Inspection team members have specific roles:
  - Moderator:
    - Typically borrowed from another project. Chairs meeting, chooses participants, controls process
  - Readers, Testers (inspectors):
    - Read code to group, look for flaws
  - Author:
    - Passive participant; answer questions when asked
  - Scribe (recorder)

Software Inspection Process

- Planning
  - Moderator checks entry criteria, choose participants, schedule meeting
- Overview
  - Provide background education, assign roles
- Preparation
- Inspection (see next slide)
- Rework
  - The producer fixes the product according to list of faults in report
- Follow-up (& possible re-inspection)
  - The moderator makes sure that all faults have been fixed and calls another inspection if more than 5% of the product has been modified

In the Meeting

- Goal: Find as many faults as possible
  - max 2 x 2 hour sessions per day
  - approx. 150 source lines/hour
- Approach: Line-by-line paraphrasing
  - Reconstruct intent of code from source
  - May also "hand test"
- Find and log defects, but don't fix them
  - Moderator responsible for staying on track

Checklists — NASA example

From “Software Formal Inspections Guidebook,” Office of Safety and Mission Assurance, NASA-GB-A302 approved August 1993

- About 2.5 pages for C code, 4 for FORTRAN
  - Divided into: Functionality, Data Usage, Control, Linkage, Computation, Maintenance, Clarity
- Examples:
  - Does each module have a single function?
  - Does the code match the Detailed Design?
  - Are all constant names upper case?
  - Are pointers not typecast (except assignment of NULL)?
  - Are pointers immediately set to NULL (or 0) following the deallocation of memory?
  - Are nested “INCLUDE” files avoided?
  - Are non-standard usages isolated in subroutines and well documented?
  - Are there sufficient comments to understand the code?

Example:
Do you see the problem here?

- Code from Apache web server, version 2.0.48
- Response to normal page request on secure (https) port

```
static void ssl_io_filter_disable(ap_filter_t *f)
{   bio_filter_in_ctx_t *inctx = f->ctx;

    inctx->ssl = NULL;
    inctx->filter_ctx->pssl = NULL;
}
```

Are pointers immediately set to NULL (or 0) following the deallocation of memory?

Incentive Structure
from [Fagan 86]

- Faults found in inspection are not used in personnel evaluation
  - Programmer has no incentive to hide faults
- Faults found in testing (after inspection) are used in personnel evaluation
  - Programmer has incentive to find faults in inspection, but not by inserting more

Why does inspection work?

- The evidence says it is cost-effective. Why?
  - Detailed, formal process, with record keeping
  - Check-lists; self-improving process
  - Social aspects of process, esp. for author
  - Consideration of whole input space
  - Applies to incomplete programs
- Limitations
  - Scale: Inherently a unit-level technique
  - Non-incremental; what about evolution?

Some figures

- Experiments over several projects have shown:
  - can be 4 times more effective than testing and
  - 2 times more effective than walkthroughs

  in finding errors
- Aetna Insurance Company:
  - Formal Review found 82% of errors, 25% cost reduction
- Bell-Northern Research:
  - Inspection cost: 1 hour per defect.
  - Testing cost: 2-4 hours per defect.
  - Post-release cost: 33 hours per defect
- Hewlett-Packard
  - Est. inspection savings (1993): $21,454,000

Document inspections

- Inspections can be applied on the whole documentation set build during the software lifecycle
- See the NASA document for guidelines, examples for the checklist
  - Is the design functionally cohesive?
  - Have the assumptions been documented?
  - Will the selected design or algorithm meet all of its requirements?
- https://standards.nasa.gov/standard/nasa/nasa-std-87399

Caveat

- Team members must read product before meeting.
- Meeting must not be longer than two hours.
- Author's boss cannot be present at the meeting.
- Inspectors must not consider solutions during meeting.
- Inspectors must not attack author; they can criticize product.

Modern code reviews

- Supported by tools
- Idea
  - Make the code visible to all team
  - Facilitate off-line discussion threads on the code
  - Keep track of ideas exchange around a piece of code in a durable way
- Advantages
  - Find faults
  - Most important, create a shared understanding of the code
  - Trigger a positive feedback mechanism within the team
- See for instance https://github.com/apache/incubator-brooklyn/pull/1093

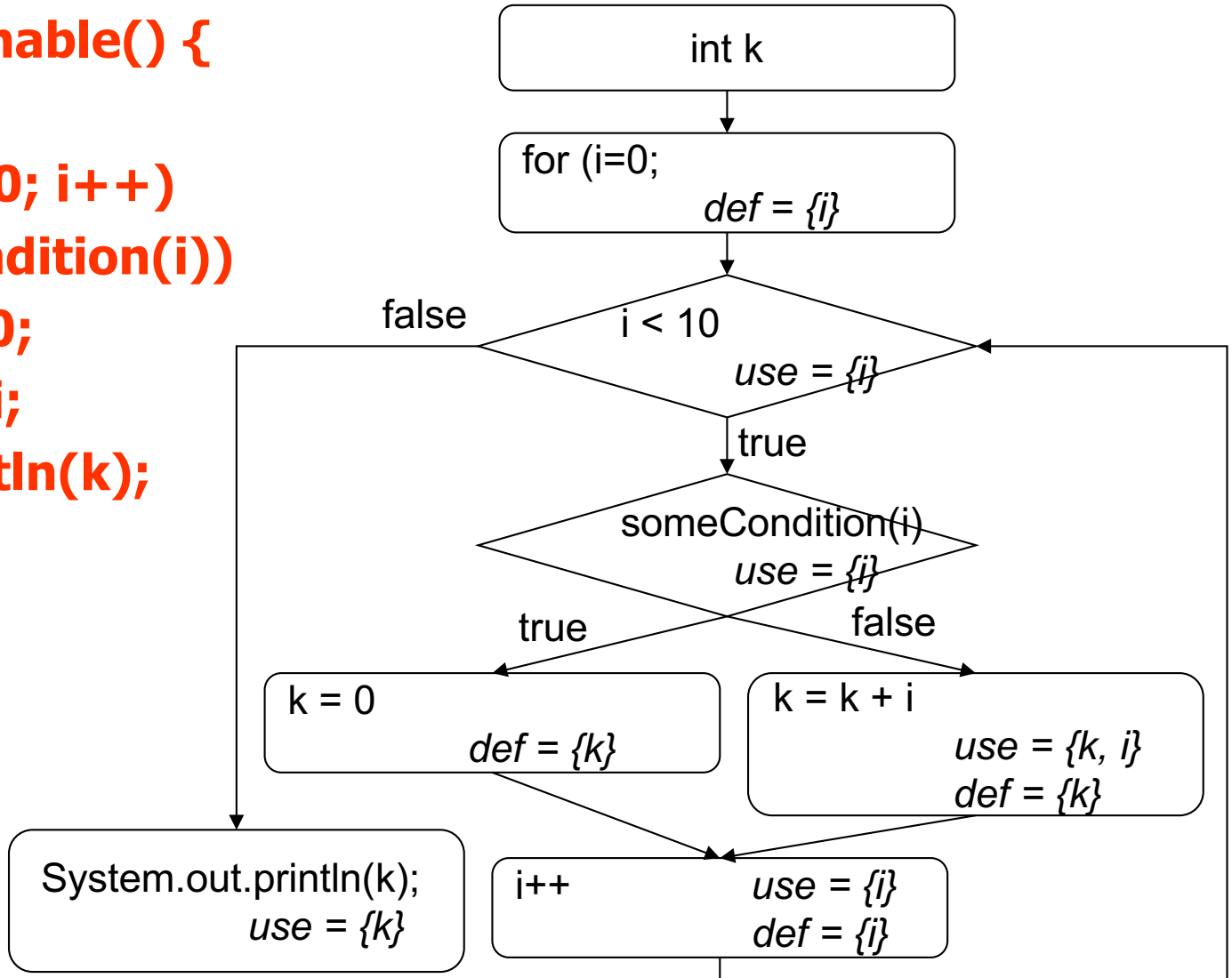
Analysis: Automated Approaches

Data flow analysis

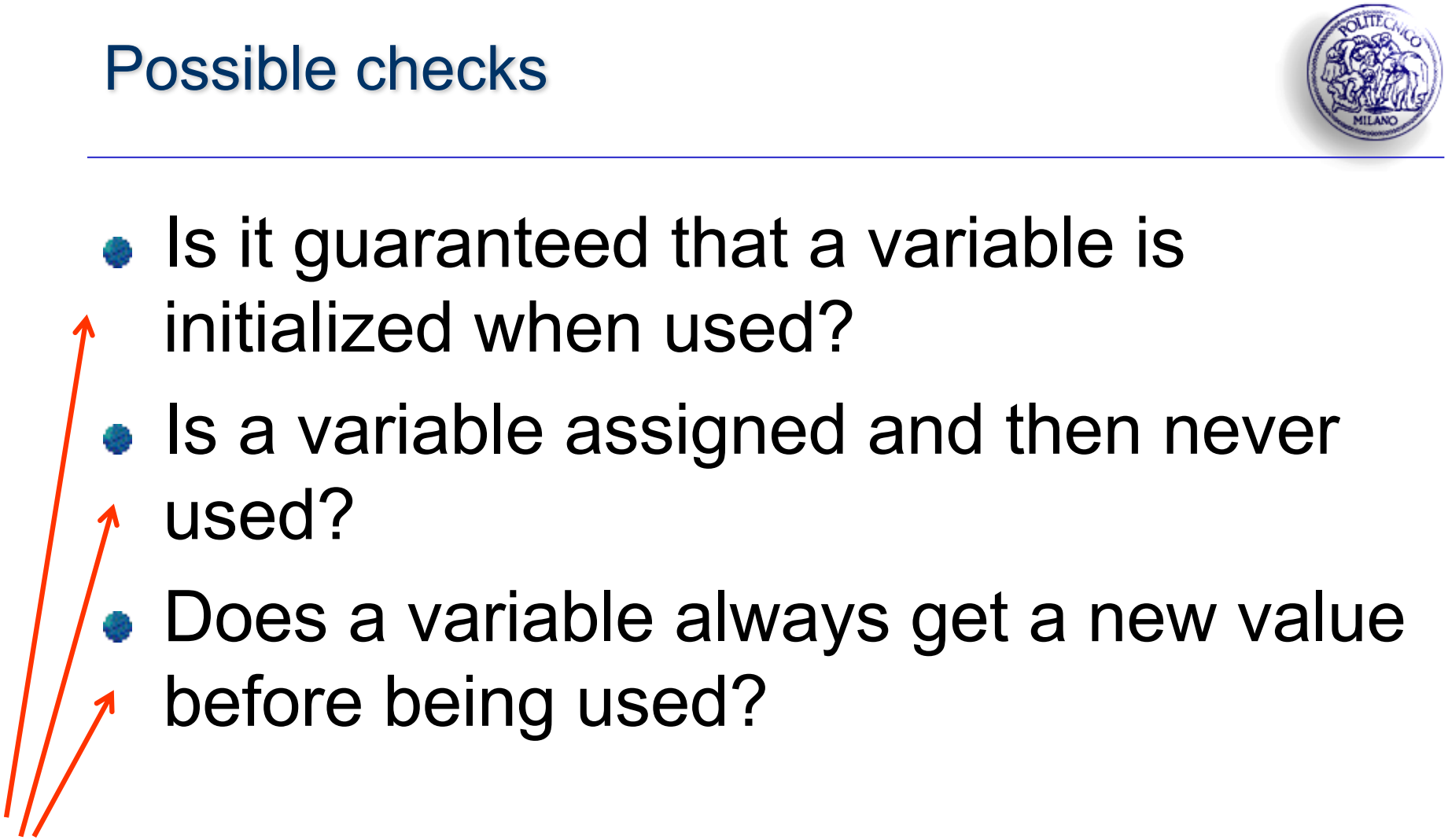
- Based on the identification of variables' definitions and use
- Typically used by compilers
  - Check for possible errors (e.g., a variable being used but not initialized)
  - Code optimization (e.g., computation results that can be used later on in the execution are saved)

Variables defs and uses

```
static void questionable() {
    int k;
    for (int i=0; i<10; i++)
        if (someCondition(i))
            k = 0;
        else k = k+i;
    System.out.println(k);
}
```



Possible checks



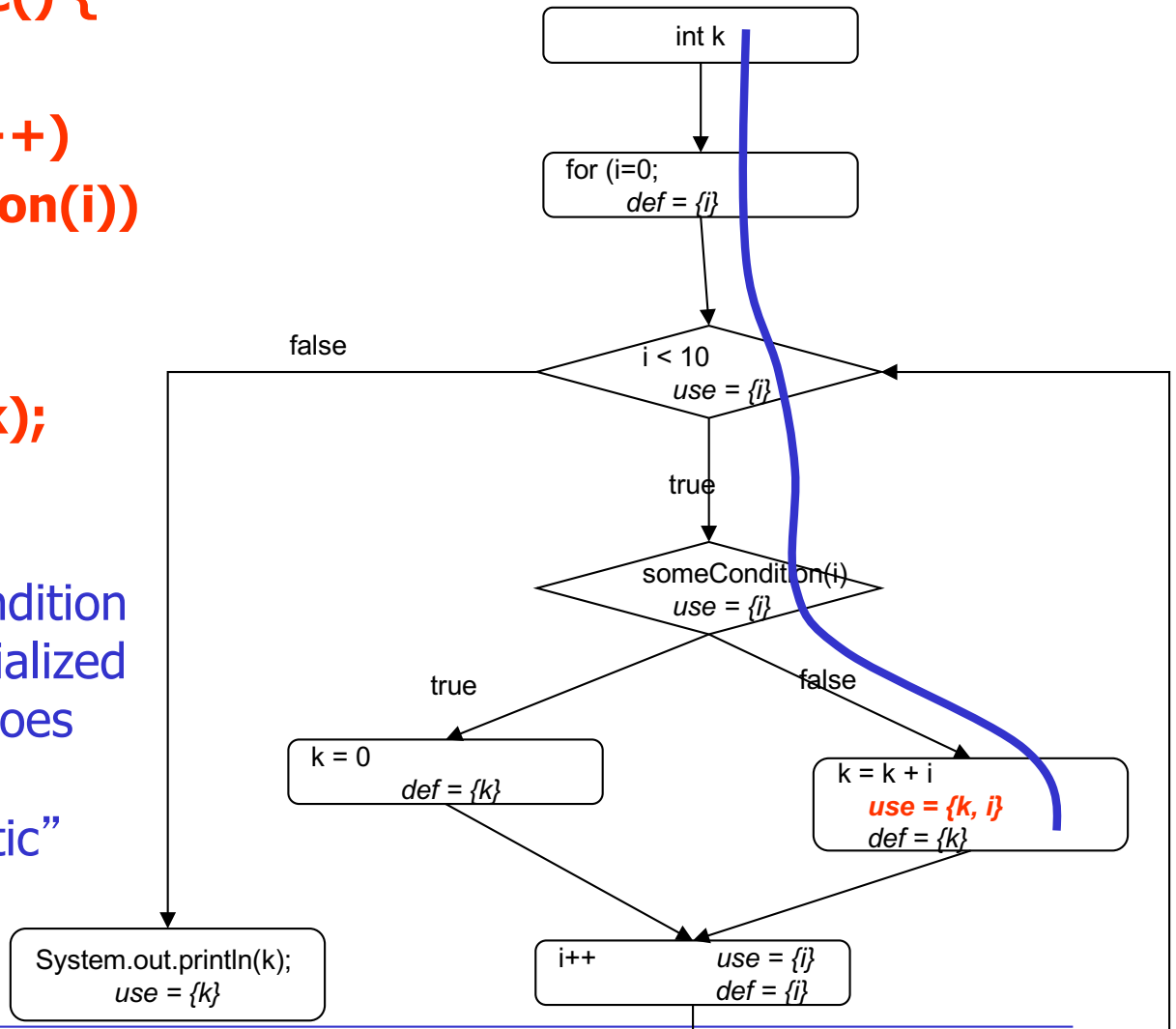
- Is it guaranteed that a variable is initialized when used?
- Is a variable assigned and then never used?
- Does a variable always get a new value before being used?

Note that these are not errors, but symptoms of possible errors

Is it guaranteed that a variable is initialized when used?

```
static void questionable() {
    int k;
    for (int i=0; i<10; i++)
        if (someCondition(i))
            k = 0;
        else k = k+i;
    System.out.println(k);
}
```

- Assuming that someCondition yields false, k is not initialized
- Maybe in practice this does not happen
- Dataflow is a "pessimistic" tool

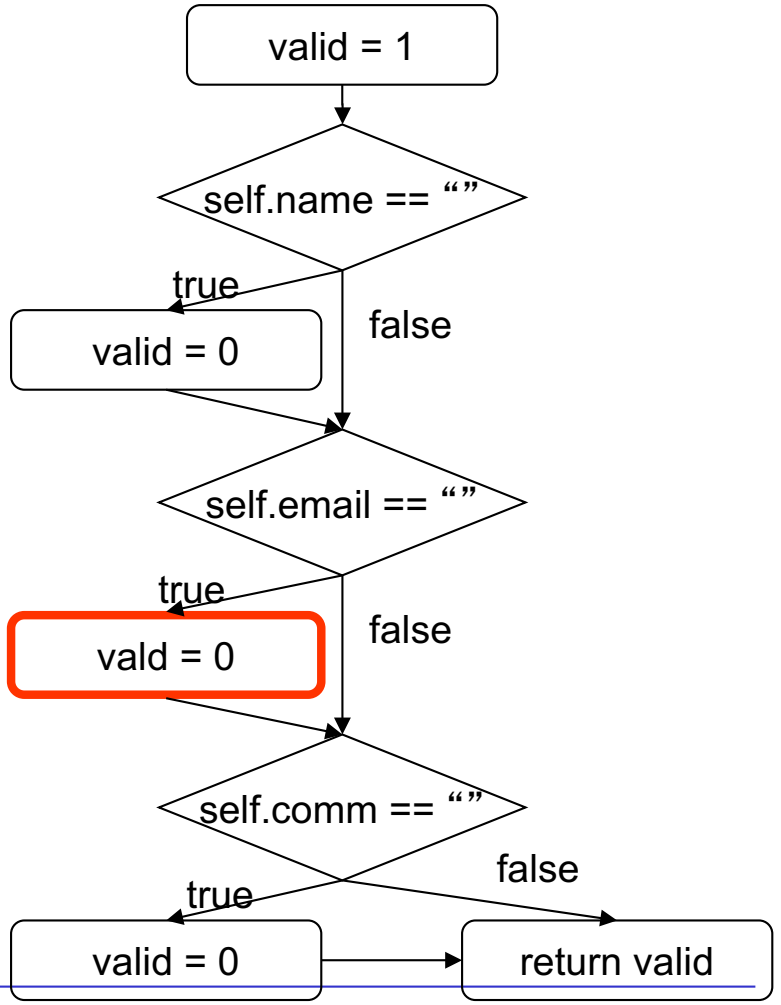


Checking for useless definitions

CGI script

```
class SampleForm(FormData):
 fieldnames=( 'name' , 'email' ,
      'comment' )
 def validate(self):
   valid = 1;
   if self.name==“”: valid = 0
   if self.email==“”: vald = 0
   if self.comment==“”: valid = 0
   return valid
```

dataflow discovers typing error as an unused variable “vald”



How to automate these checks?

- Derive the control flow diagram
- Identify points where variables are defined and used
  - Note: i++ contains both a use and a definition for i
- Identify def-use pairs and analyse the pairs to answer to questions in slide 48
- Example

```
1 int i, k = 0;
2 i = k;
3 while (i<10)
4   i++;
```

def-use pairs for k

<1, 2>

def-use pairs for i

<2, 3>, <2, 4>, <4, 3>, <4, 4>

Def-use pairs through an example

- Consider the following fragment of code:

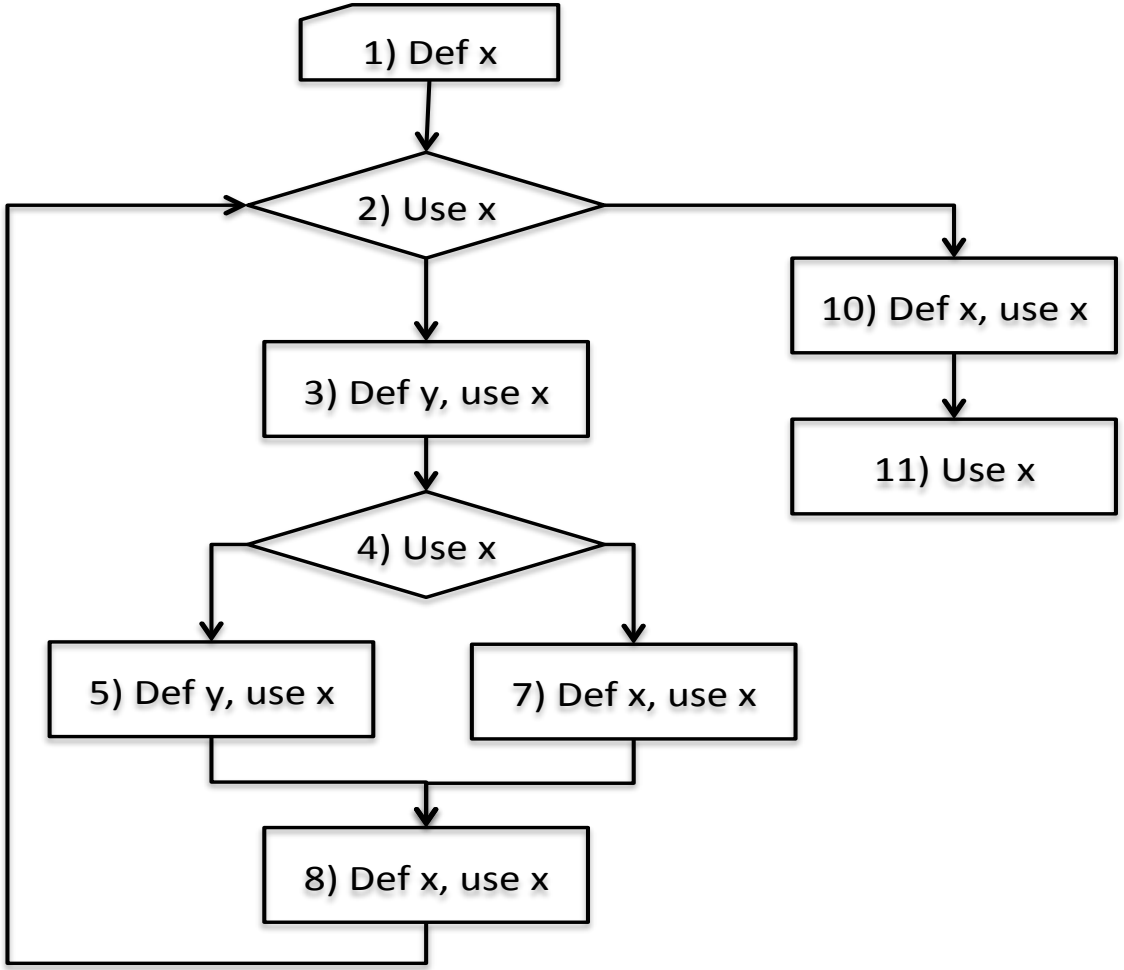
```
int foo() {
1   x = input();
2   while (x > 0) {
3     y = 2 * x;
4     if (x > 10)
5       y = x - 1;
6     else
7       x = x + 2;
8     x = x - 1;
9   }
10  x = x - 1;
11  return x;
}
```

You are to accomplish the following:

1. draw the control flow graph of the program;
2. provide the use-definition information for variables x and y;
3. provide the def-use pairs
4. point out a potential issue with this code that data flow analysis would be able to spot;

1., 2. Control flow graph and def/use

```
int foo() {
1   x = input();
2   while (x > 0) {
3     y = 2 * x;
4     if (x > 10)
5       y = x - 1;
6     else
7       x = x + 2;
8     x = x - 1;
9   }
10  x = x - 1;
11  return x;
}
```



3. Def-use pairs

x	**<1, 2> <1, 3> <1, 4> <1, 5> <1, 7> <1, 8> <1, 10> <7,8> <8, 2> <8, 3> <8, 4> <8, 5> <8, 7> <8, 8> <8, 10> <10, 11>**
y	No pair exists

4. Potential issue

- The potential issue spotted by data flow analysis is exactly that y is defined, but it is never used in the program

Data flow analysis: limitations

- It is pessimistic: it cannot distinguish between paths that can actually be executed and paths that cannot
- It cannot always determine whether two names or expressions refer to the same object
  - Becomes a problem for languages that allow aliasing between variables!

Symbolic execution

- Builds predicates that characterize
  - Conditions for executing paths
  - Effects of the execution on program state
- Symbolic state:
  - <path-condition, symbolic bindings>
- Finds important applications in
  - program analysis
  - test data generation
  - formal verification (proofs) of program correctness

Symbolic state

Values are expressions over symbols

Executing statements computes new expressions, example:

```
mid = (high+low)/2
```

Execution with concrete values

before instruction

low	12
high	15
mid	-

after instruction

low	12
high	15
mid	13

Execution with symbolic values

before instruction

low	L
high	H
mid	-

after instruction

low	L
high	H
mid	(L+H)/2

Example 1

```
read (a); read (b);
x = a + b;
write (x);
```

- Let A and B be the symbolic values read for a, b
- x = A+B
- Printed result is A+B

More examples


```
read (a); read (b);
x = a + 1;
y = x * b;
write (y);
[res: (A + 1) * B]
```

```
read (a); read (b);
x = a + 1;
x = x + b + 2;
write (x);
[res: A + B + 3]
```

Path condition and branches

```
1   read (y, a);
2   x = y + 2;
3   if x > a
4     a = a + 2;
5   else
6     y = x + 3;
7   x = x + a + y;
```

How can the symbolic executor determine if x > a is true or false?

Execution is performed for a specific path, for instance:

Execution path: <1, 2, 3, 5, 6, 7>

Path condition: $Y + 2 \leq A$

Execution result:

$\{a=A, y=Y+5, x=2Y+A+7\}$

In short:

$\langle\{a = A, y=Y+5, x=2*Y+A+7\}\langle 1, 2, 3, 5, 6, 7\rangle\ Y + 2 \leq A\rangle$

Another example

using symbolic execution, identify the path condition that allows the program to follow the path 1 2 3 4 5 8 9 2 3 4 6 7 8 9 2 10 11

```
int foo() {
1   x = input();
2   while (x > 0) {
3     y = 2 * x;
4     if (x > 10)
5       y = x - 1;
6     else
7       x = x + 2;
8     x = x - 1;
9   }
10 x = x - 1;
11 return x;
}
```

1. x = X
2. X>0
3. y=2*X
4. X>10
5. y=X-1
8. x = X-1
2. X-1 > 0
3. y = 2*(X-1)
4. X-1<=10
7. x = X-1+2=X+1
8. x = X+1-1=X
2. X<=0

The path condition for the given path is then
X>10 and X<=11 and X<=0
This is clearly an inconsistent condition.
Thus, the given path is impossible