CS 569
Selected Topics in Software Engineering:
Program Analysis & Evaluation

Invariant Detection

Oregon State University, Winter 2024

Announcements

- **Homework 1** will be released today (due **Monday, Jan 29, 2024, 11:59 PM**)
- **Project Idea Presentation and Report** will be released today (due before class on **Monday, Jan 29, 2024, 11:59 AM**)
- Projects will be completed in teams of **up to 5 students**. Each team is responsible for a single project.
- You should select a team by adding yourself to one of the "Project" groups on Canvas at https://canvas.oregonstate.edu/courses/1964338/groups#tab-110070 by **Wednesday, Jan 17, 2024, 11:59 PM PT**.
  Note: Students who don't create teams will be randomly grouped to form teams.

Recap

- Importance of knowing about Software Analysis and Testing
- Foundations
  - Programming language implementation, syntax, and semantics
- Program analysis overview
  - Static analysis (Overapproximation)
  - Dynamic analysis (Underapproximation)
  - Soundness vs Completeness
  - Test generation

Questions?

Todays Plan

- What are Program Invariants
- Techniques to detect Invariants
- Usefulness in practice

Dynamic Program Analysis

- Infer facts of the program by monitoring its runs
- Examples:
  - Array bound checking (Purify)
  - Memory leak detection (Valgrind)
  - Data race detection (Eraser)
  - Finding likely invariants (Diakon)

Static Program Analysis

- Infer facts of the program by inspecting its source (or binary) code
- Examples:
  - Suspicious error patterns (Lint, FindBugs, Coverity)
  - Checking API misuse (Microsoft SLAM)
  - Memory leak detection (Facebook Infer)
  - Verifying invariants (ESC/Java)

Program Invariants

- Useful facts about a program that hold true in **all program executions**
- Useful to infer program behavior and validate against specification

Program Invariants

```
int sqr(int x) {
  return x * x;
}

void main() {
  int res;
  if (getc() == ‘a’)
      res = sqr(6) + 6;
  else
      res = sqr(-7) – 7;

    res = c
}
```

An invariant at the end of the program is **(res == c)** for some constant c. **What is c?**

Program Invariants

```
int sqr(int x) {
  return x * x;
}

void main() {
  int res;
  if (getc() == ‘a’)
      res = sqr(6) + 6;
  else
      res = sqr(-7) - 7;

    res = 42
}
```

An invariant at the end of the program is **(res == c)** for some constant c. **What is c?**

For all executions of program, res:
{6*6 + 6 = 42, (-7)*(-7) – 7 = 42}

Therefore, c = 42

Program Invariants

```
int sqr(int x) {
  return x * x;
}

void main() {
  int res;
  if (getc() == 'a')
      res = sqr(6) + 6;
  else
      res = sqr(-7) - 7;
  if (res != 42)
     disaster();
}
```

Invariant discovered is a useful fact to prove that this program can never call disaster!

Discovering Invariants By Dynamic Analysis

```
int sqr(int x) {
  return x * x;
}

void main() {
  int res;
  if (getc() == ‘a’)
      res = sqr(6) + 6;
  else
      res = sqr(-7) – 7;
  if (res != 42)
     disaster();
}
```

(res == 42) *might be* an invariant

(res == 30) *is definitely not* an invariant

Daikon: Dynamic Discovery of **Likely** Invariants

Discovered properties are not stated in any part of the program

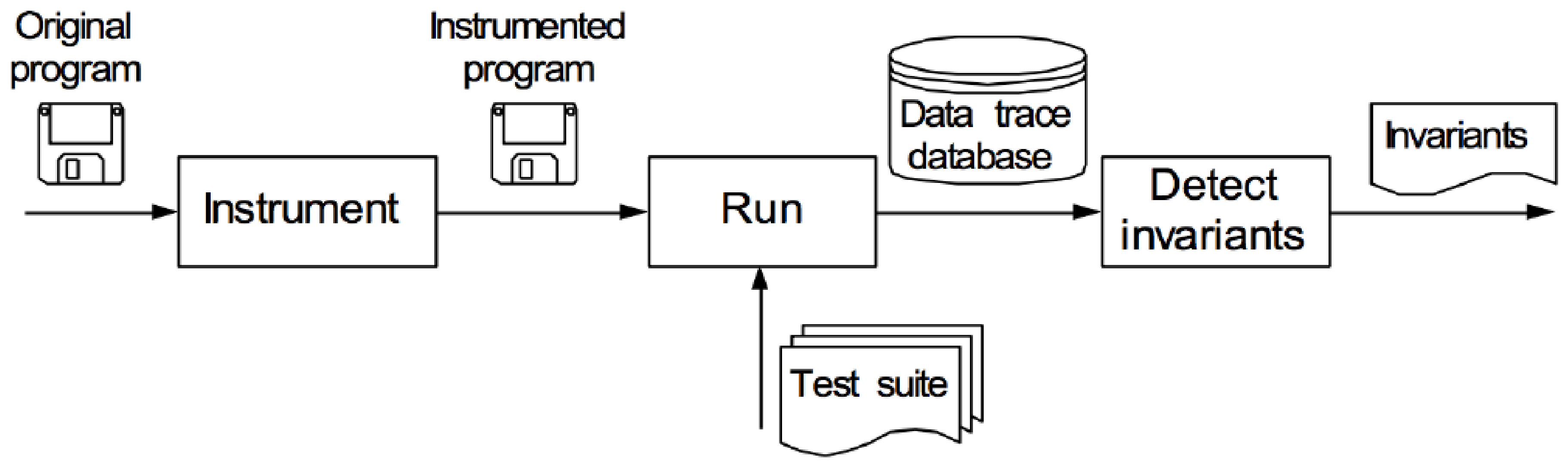
- They are discovered by monitoring the execution of the program on a set of inputs (a test set)
- The only thing that is guaranteed is that the discovered properties hold for all the inputs in the test set
- No guarantee of soundness or completeness

Identify Invariants in this Function

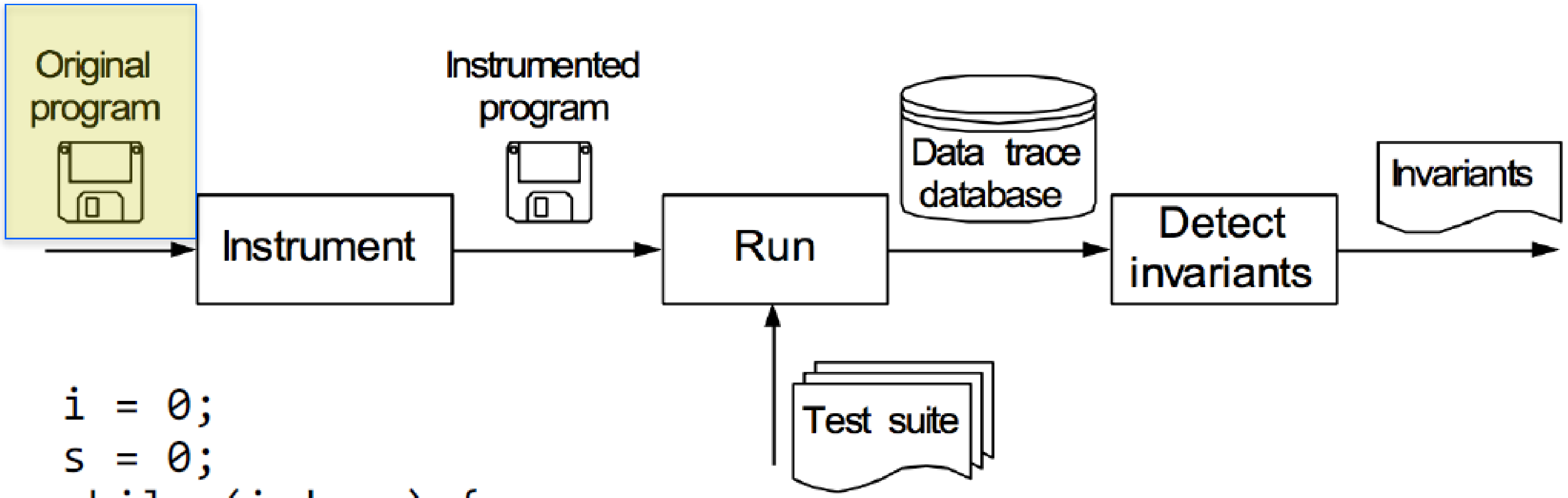
```
// Sum array b of length n into s
int sumArray(b, n) {
  i = 0;
  s = 0;
  while (i != n) {
    s = s + b[i];
    i = i+1;
  }

  return s;
}
```

Dynamic Invariant Detection



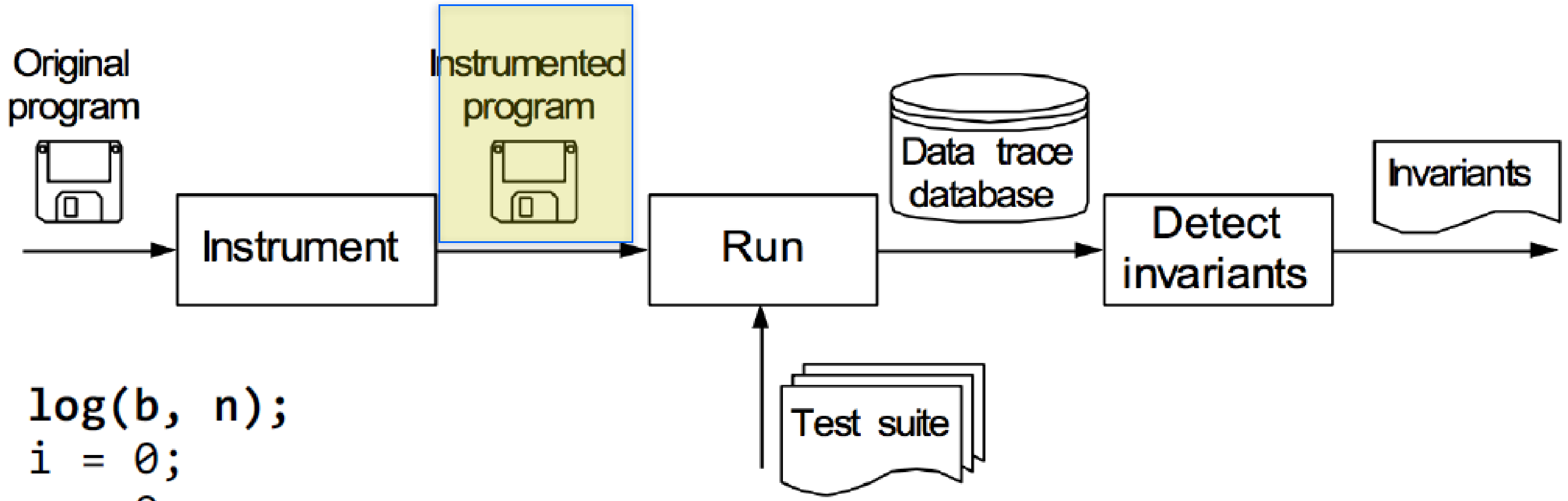
Dynamic Invariant Detection



```
i = 0;
s = 0;
while (i != n) {
  s = s + b[i];
  i = i+1;
}
```

```
// Sum array b of length
// n into variable s
```

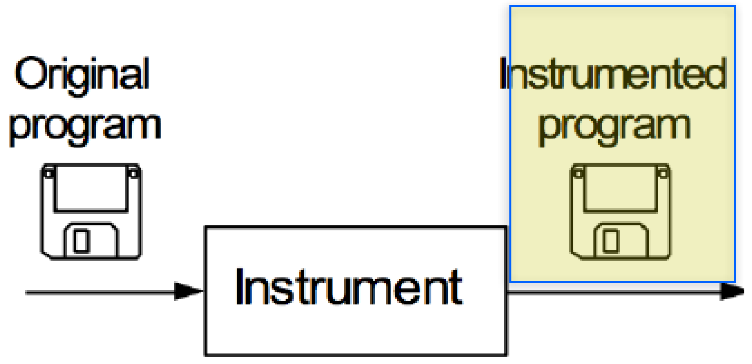
Dynamic Invariant Detection

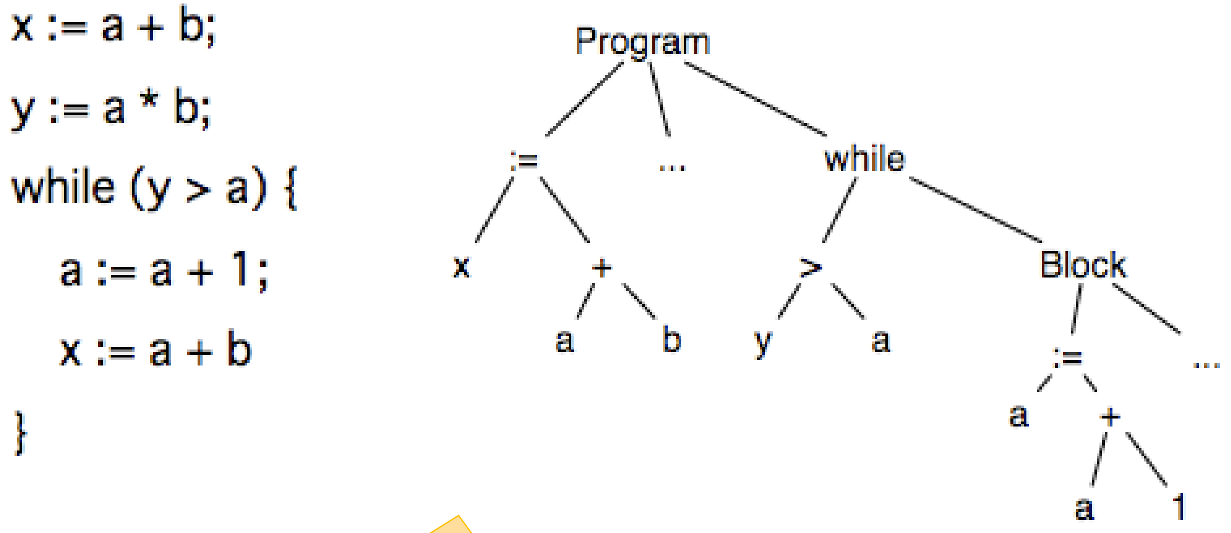


```
log(b, n);
i = 0;
s = 0;
while (i != n) {
  log(i, s, n, b[i]);
  s = s + b[i];
  i = i+1;
}
```

Original program instrumented at “interesting” points to collect traces

Dynamic Invariant Detection



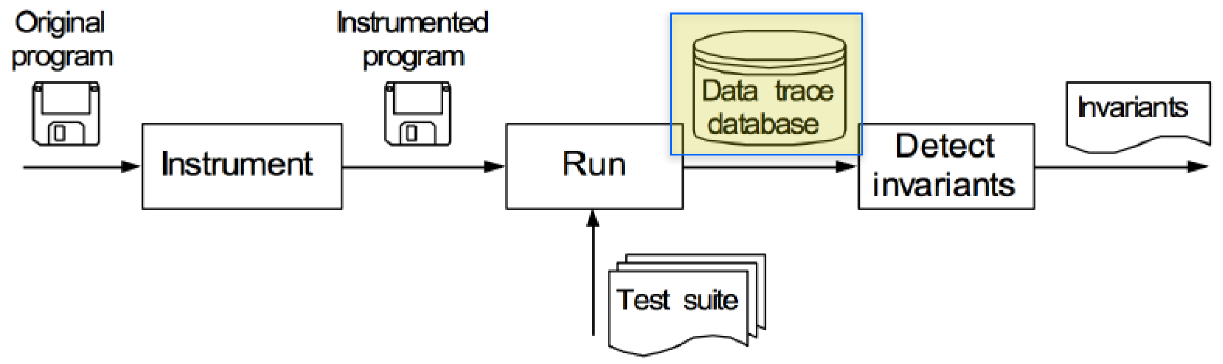


```
x := a + b;
y := a * b;
while (y > a) {
    a := a + 1;
    x := a + b
}
```

```
log(b, n);
i = 0;
s = 0;
while (i != n) {
   log(i, s, n, b[i]);
   s = s + b[i];
   i = i+1;
}
```

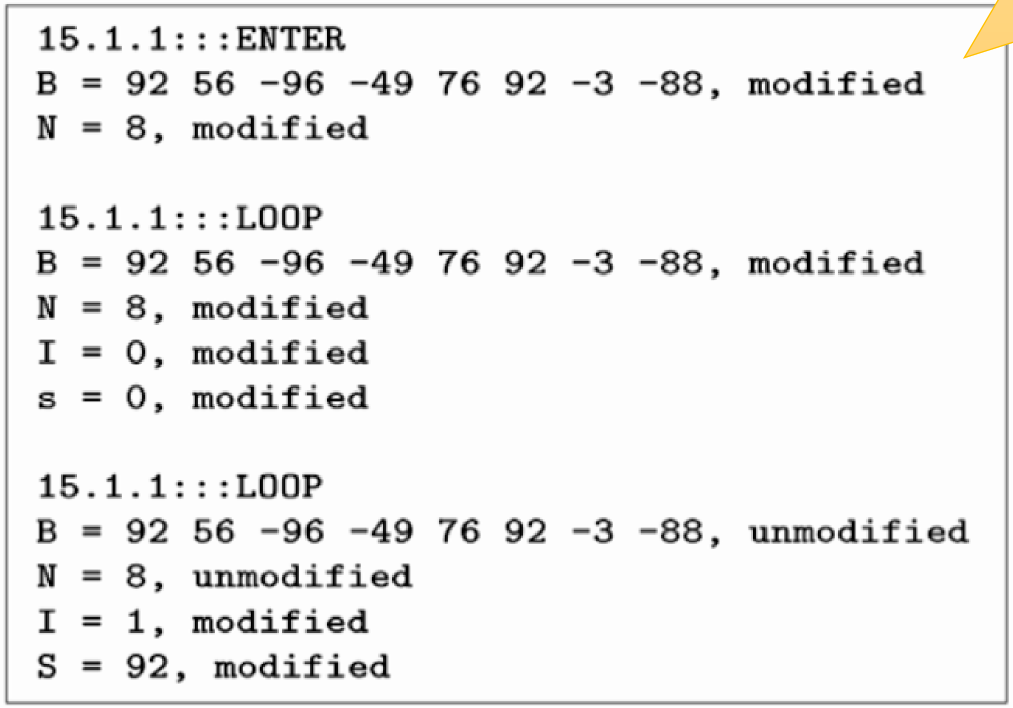
Instrumentation is done by parsing the AST of program to determine points of interest and add new nodes

Dynamic Invariant Detection



Execution trace collected at "interesting" points using test inputs

```
log(b, n);
i = 0;
s = 0;
while (i != n) {
  log(i, s, n, b[i]);
  s = s + b[i];
  i = i+1;
}
```

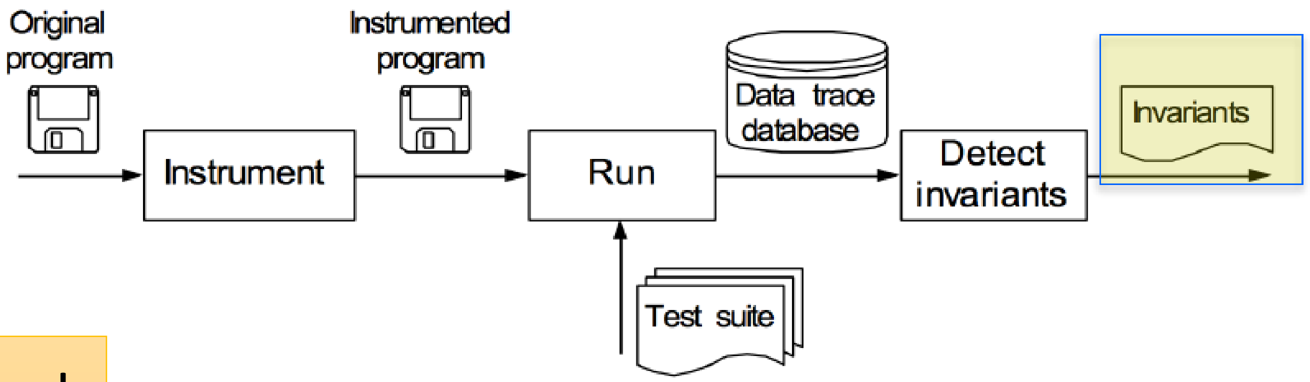


```
15.1.1:::ENTER
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified
I = 0, modified
s = 0, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, unmodified
N = 8, unmodified
I = 1, modified
S = 92, modified
```

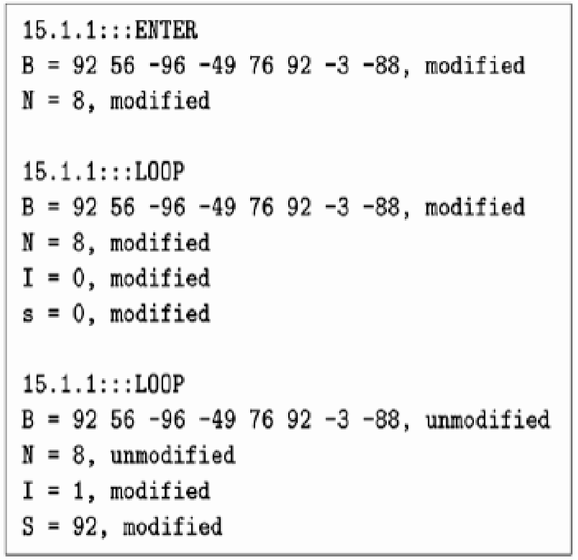
Dynamic Invariant Detection



Dynamically determined invariants

Determined Invariants

- i >=0
- s >= 0
- i <= n
- s != null
- b[ ] = orig(b[ ])
- Sum(b[ ]) == sum(orig(b[ ]))

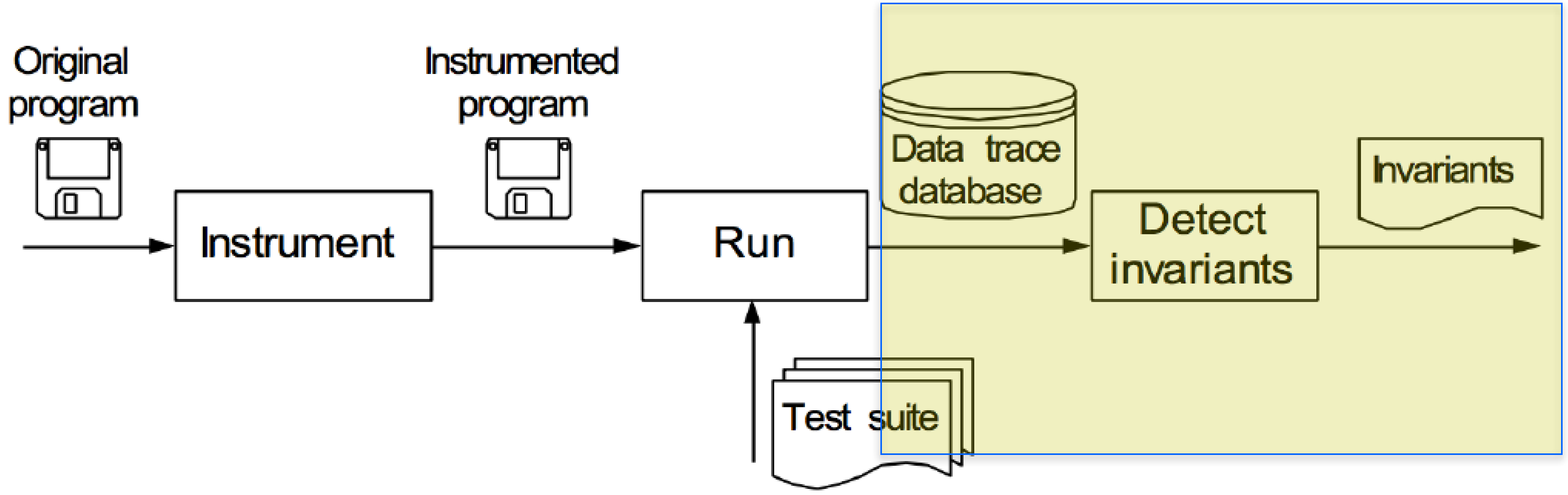


```
15.1.1:::ENTER
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, modified
N = 8, modified
I = 0, modified
s = 0, modified

15.1.1:::LOOP
B = 92 56 -96 -49 76 92 -3 -88, unmodified
N = 8, unmodified
I = 1, modified
S = 92, modified
```

Inferring Invariants



Example: Inferring *i <= n* loop Invariant

Pre-condition

Post-condition

Loop invariant

```
int sum(int *b,int n) {
  requires $n \geq 0$
  ensures $s=\sum_{0 \leq j<n} b[j]$
  i, s := 0, 0;
  while i ≠ n do
  inv $0 \leq i \leq n \wedge s=\sum_{0 \leq j<i} b[j]$
     i := i + 1
     s := s + b[i]
}
```

Possible relationships:

i<n *i≥n* *i≤n* *i=n* *i>n*

Cull relationships with traces

Trace: n=0

Example: Inferring *i <= n* loop Invariant

```
int sum(int *b,int n) {
  requires n ≥ 0
  ensures s=∑_{0≤j<n} b[j]
  i, s := 0, 0;
  while i ≠ n do
  inv 0 ≤ i ≤ n ∧ s=∑_{0≤j<i} b[j]
     i := i + 1
     s := s + b[i]
}
```

Loop invariant

Possible relationships:

$\cancel{i<n}$ $i \geq n$ $i \leq n$ $i=n$ $\cancel{i>n}$

Cull relationships with traces

Trace: n=0

n	i
0	0

Example: Inferring *i <= n* loop Invariant

```
int sum(int *b,int n) {
  requires n ≥ 0
  ensures s=Σ_{0≤j<n} b[j]
  i, s := 0, 0;
  while i ≠ n do
  inv 0 ≤ i ≤ n ∧ s=Σ_{0≤j<i} b[j]
     i := i + 1
     s := s + b[i]
}
```

Loop invariant

Possible relationships:

$\cancel{i<n}$ $\cancel{i>n}$ $i \leq n$ $\cancel{i=n}$ $\cancel{i \geq n}$

Cull relationships with traces

Trace: n=1

n	i
1	0
1	1

Example: Inferring *i <= n* loop Invariant

```
int sum(int *b,int n) {
  requires $n \geq 0$
  ensures $s = \sum_{0 \leq j < n} b[j]$
  i, s := 0, 0;
  while i ≠ n do
  inv $0 \leq i \leq n \land s = \sum_{0 \leq j < i} b[j]$
     i := i + 1
     s := s + b[i]
}
```

Loop invariant

Possible relationships:

$\cancel{i<n}$ $\cancel{i \geq n}$ $i \leq n$ $\cancel{i=n}$ $\cancel{i>n}$

Cull relationships with traces

Trace: n=2

n	i
2	0
2	1
2	2

Inferring Invariants

- There are two issues:
  - Choosing which invariants to infer
  - Inferring the invariants
- Daikon infers invariants at specific program points
  - procedure entries
  - procedure exits
  - loop heads (optional)
- Daikon can only infer certain types of invariants
  - it has a library of invariant patterns
  - it can only infer invariants which match to these patterns

Types of Invariants

Single Variables	
Constant Value	x = a
Uninitialized Value	x = uninit
Small Value Set	x in {a,b,c}
Single Numeric Variables	
Range Limits	x >= a, x <= b, etc…
Non-zero	x != 0
Modulus	x = a (mod b)
Non-Modulus	x != a (mod b)

Types of Invariants

Two Numeric Variables	
Linear Relationship	$y = ax + b$
Functional Relationship	$y = f(x)$
Comparison	$x > y$, $x = y$, etc…
Combinations of Single Numeric Values	$x+y = a \pmod b$
Three Numeric Variables	
Polynomial Relationship	$z = ax + by + c$

Invariant Confidence

- Not all unfalsified invariants should be reported
- There may be a lot of irrelevant invariants which may just reflect properties of the test suite
- The output may become unreadable if a lot of irrelevant invariants are reported
- Improving (increasing) the test set would reduce the number of irrelevant invariants

Invariant Confidence

An invariant is only *likely* if

- The observations do not disprove it

AND

- The relevant observations are statistically significant

How to compute statistical significance?

- For an invariant ***f(x,y)***, what is the probability that ***f(x,y)*** is satisfied under a random choice of ***x*** and ***y***?
- **Assume** 0 <= ***x,y*** *<= 1000* uniformly distributed
  - P(x==y) = 0.001
  - P(x < y) = 0.5
  - P(x!=y) = 0.999
- If this probability is less than a user defined confidence level, then ***f(x,y)*** is reported as an invariant

Cost of Detecting Invariants

- The cost of inferring invariants increases as follows:
  - linear in the number of test cases
  - linear in the number of program points
  - quadratic in the number of variables at a program point (**n^2** possible relationships between **n** variables at a program point)
- Typically takes a few minutes per procedure.

Drawbacks of Dynamic Invariant Detection

- Requires a reasonable test suite
- Invariants may not be true
  - May only be true for this test suite
- May detect uninteresting invariants
  - Some may actually tell you about the test suite, not the program (still useful)
- May miss some invariants
  - Detects all invariants in a procedure, but not all interesting invariants are in that procedure
  - Only reports invariants that are statistically unlikely to be coincidental

Note: easier to reject false or uninteresting invariants than to guess true ones!

In Practice

- This works!
- Finds interesting invariants for complex programs.
- Gives concise specifications
- Needs fewer executions than you’d think.

Diakon

- Implements dynamic invariant detection
- Open source, free to use
- Highly robust and customizable
- Takes some time to master but very powerful
- You’ll see it on homework 1

https://plse.cs.washington.edu/daikon/

Questions?

Discovering Invariants By Static Analysis

```
int sqr(int x) {
  return x * x;
}

void main() {
  int res;
  if (getc() == 'a')
      res = sqr(6) + 6;
  else
      res = sqr(-7) - 7;
  if (res != 42)
     disaster();
}
```

is definitely

(z == 42) ~~*might be*~~ an invariant

(z == 30) *is definitely not* an invariant

Detecting Invariants Using Static Analysis In Programs with Unbounded paths

```
void main() {
  int x = getInput();
  int y;
  int z = 3;
  while (true) {
      if (x == 1)
          y = 7;
      else
          y = z + 4;
      assert(y == 7);
 }
}
```

Static Analysis Problem:
Find variables that have a constant value at a given program point

Detecting Invariants Using Static Analysis In Programs with Unbounded paths

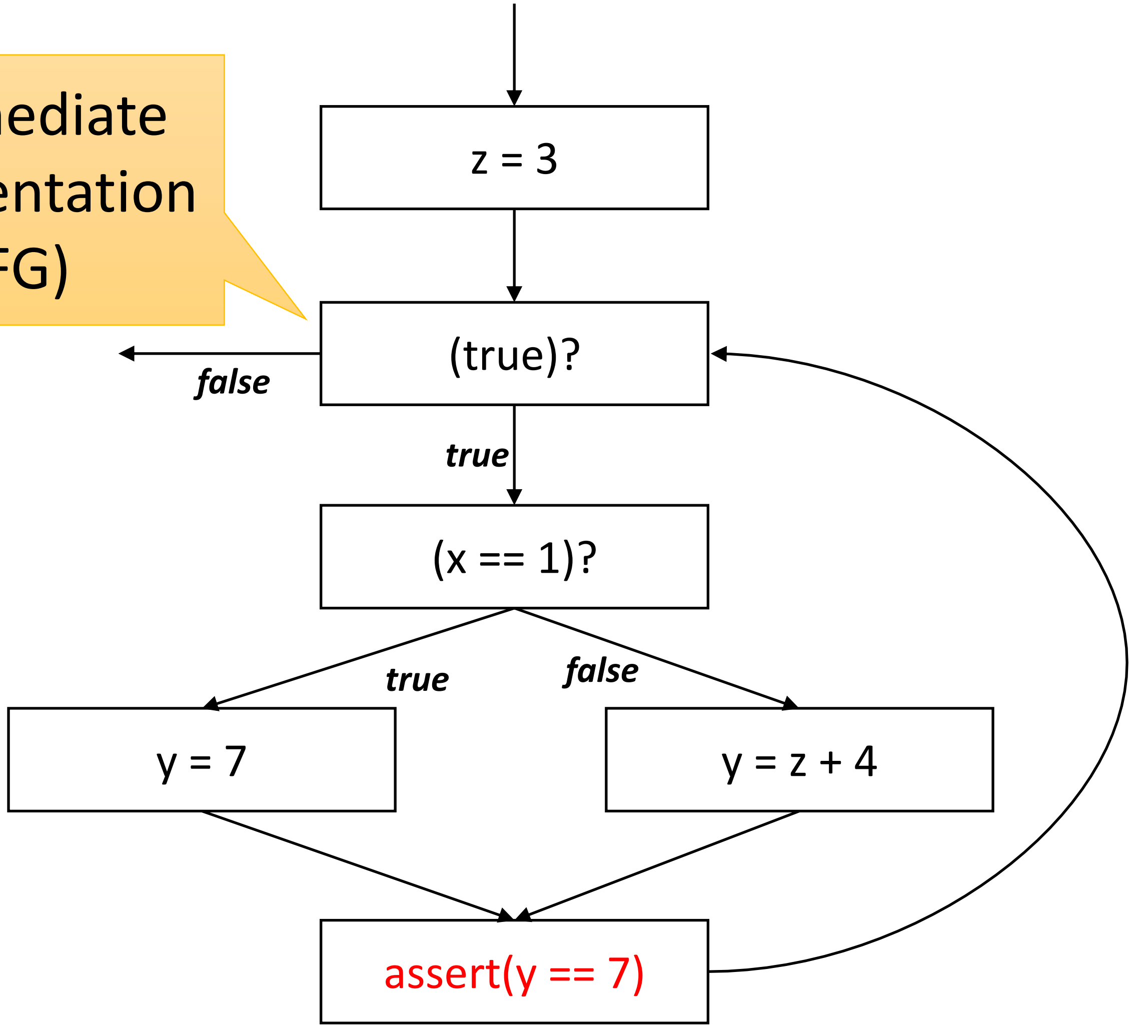
```
void main() {
  int x = getInput();
  int y;
  int z = 3;
  while (true) {
     if (x == 1)
        y = 7;
     else
        y = z + 4;
     assert(y == 7);
 }
}
```

Find variables that have a constant value at a given program point (end of loop)

Detecting Invariants Using Static Analysis In Programs with Unbounded paths

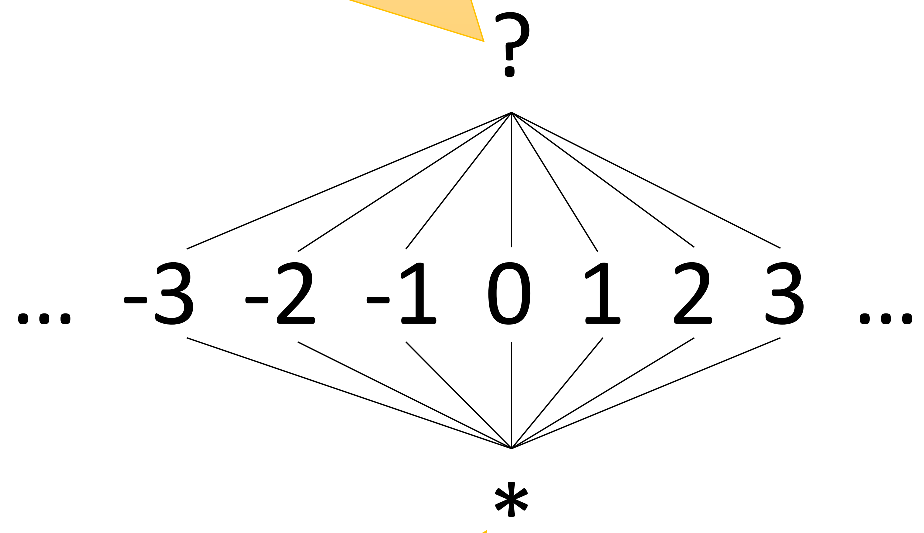
```
void main() {
  int x = getInput();
  int y;
  int z = 3;
  while (true) {
     if (x == 1)
        y = 7;
     else
        y = z + 4;
     assert(y == 7);
 }
}
```

Intermediate Representation (CFG)

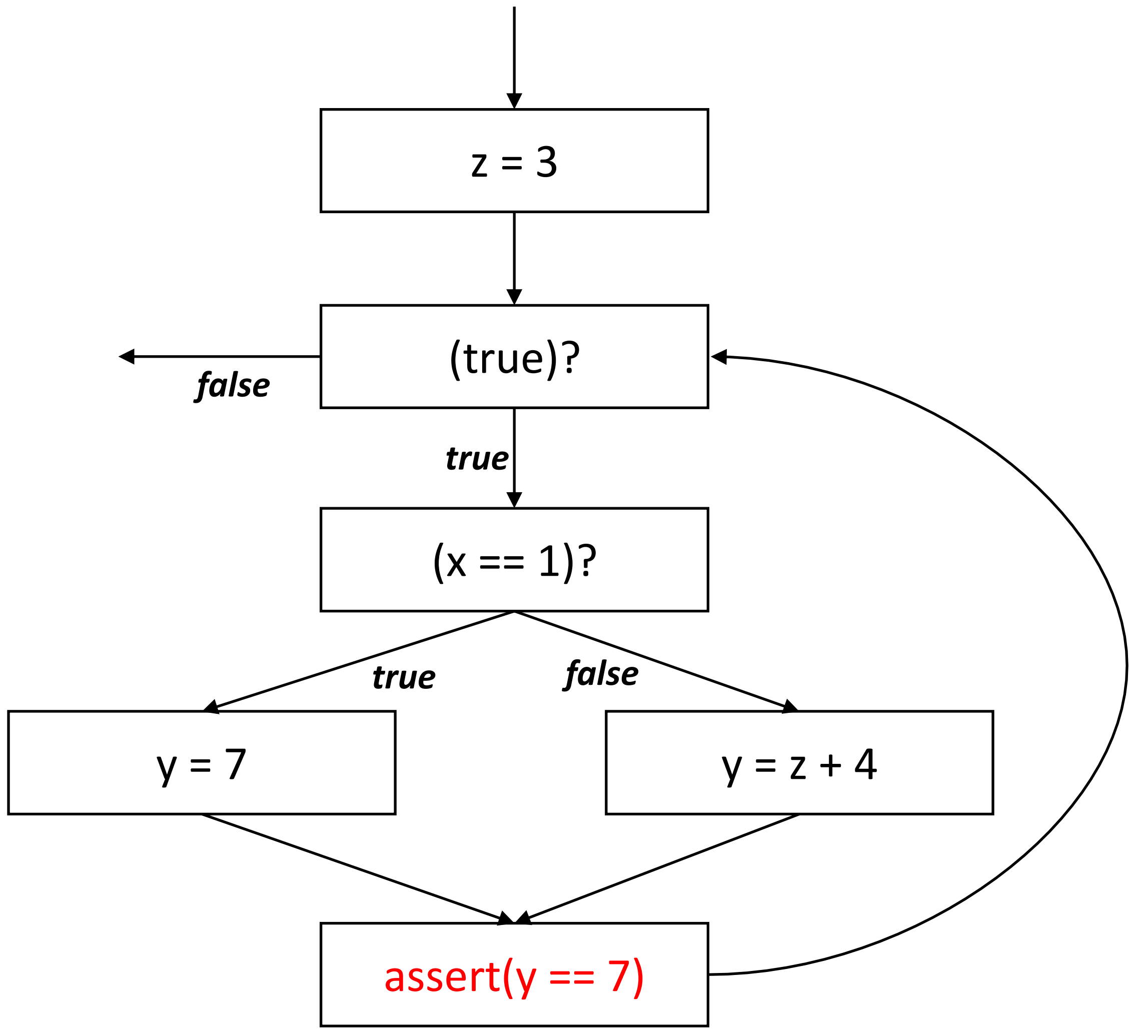


Abstract Domain

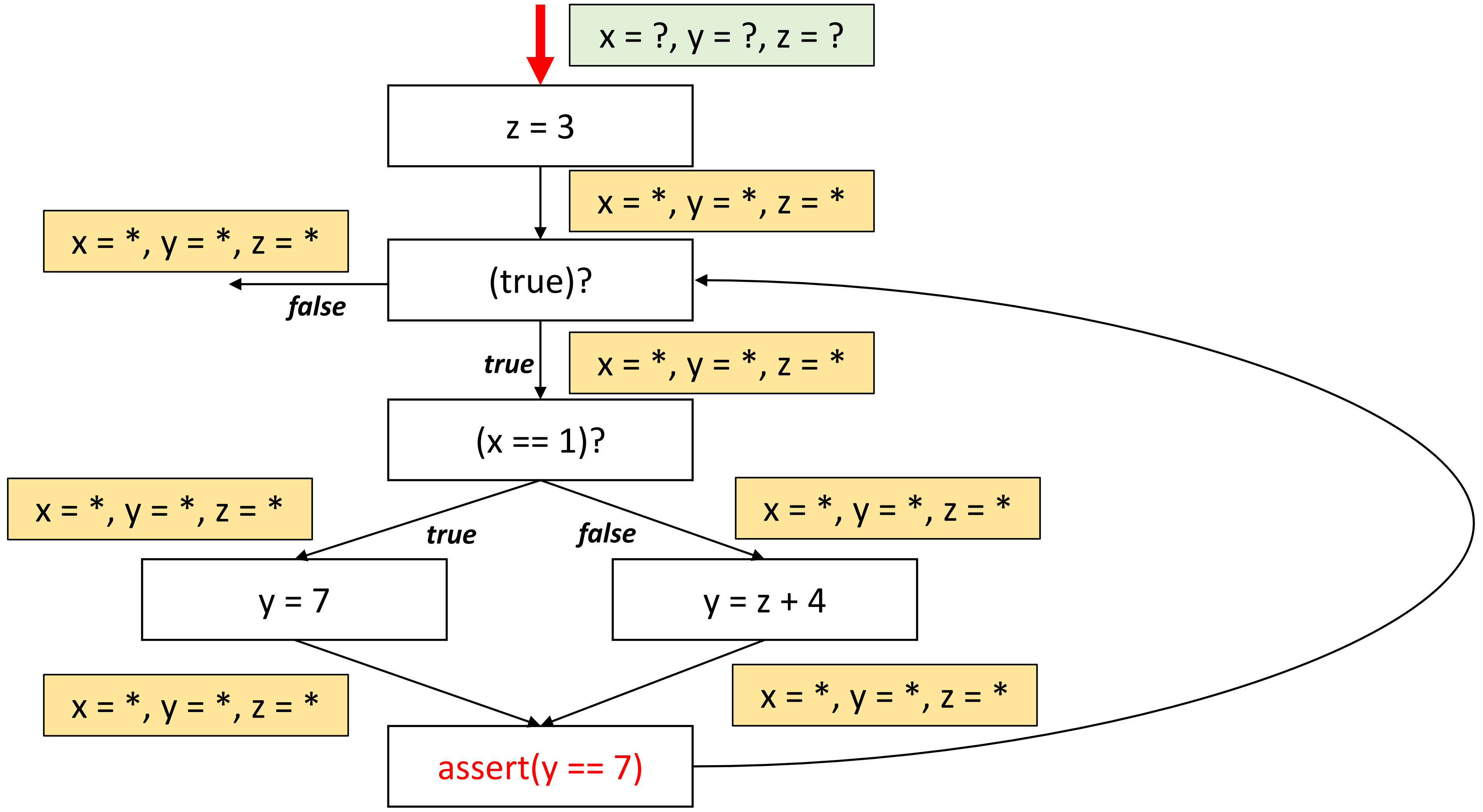
Value is unknown to the analysis



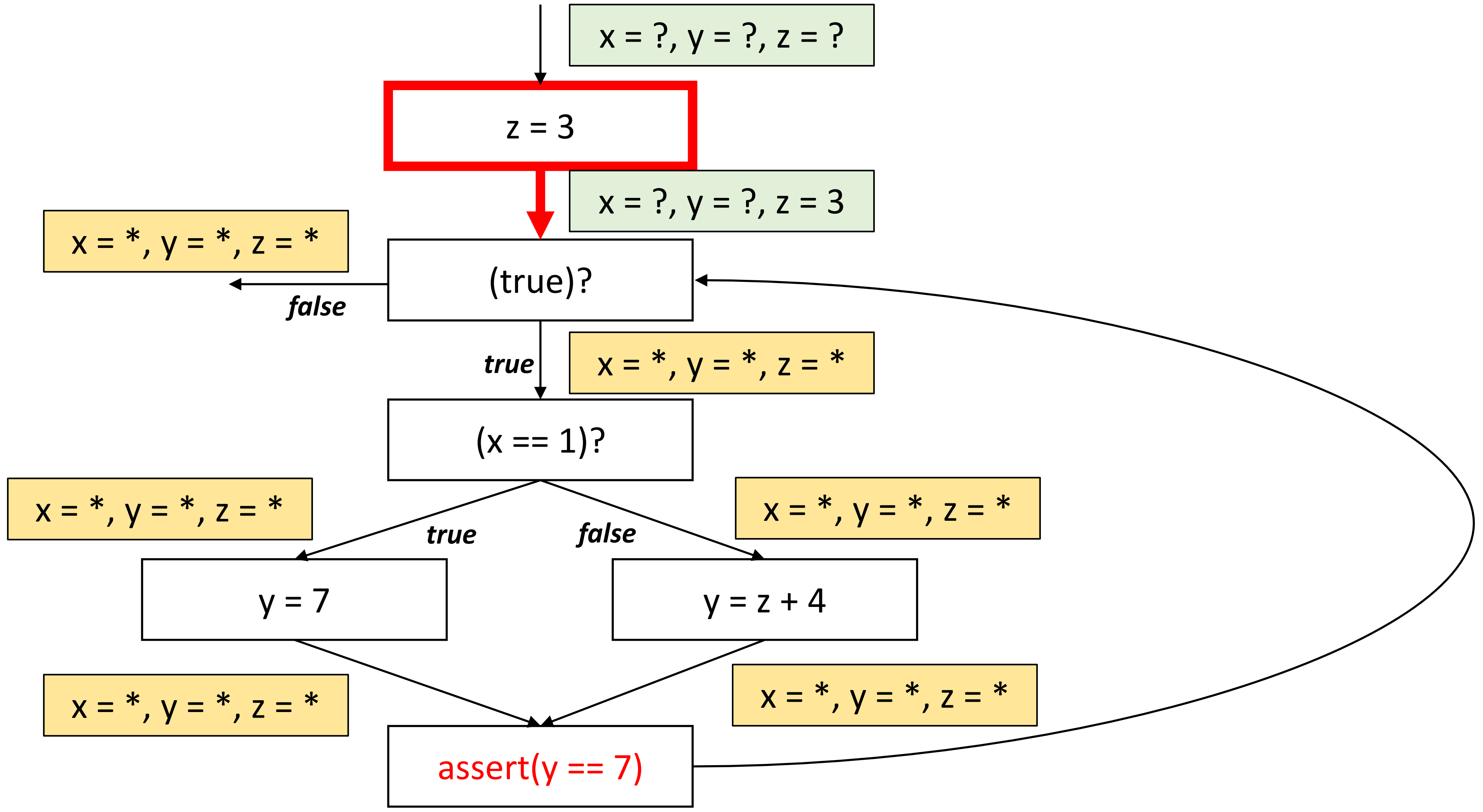
Value is undefined by the analysis



Iterative Approximation

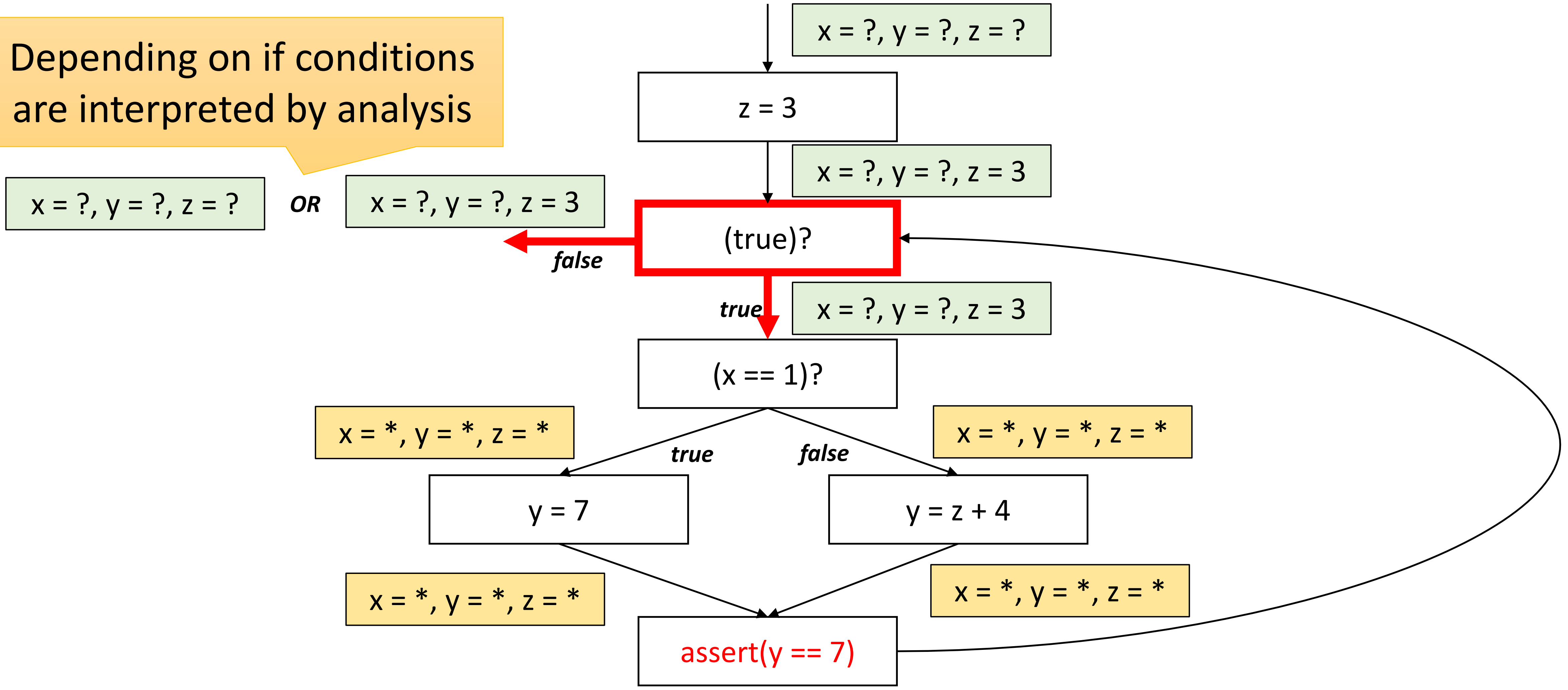


Iterative Approximation

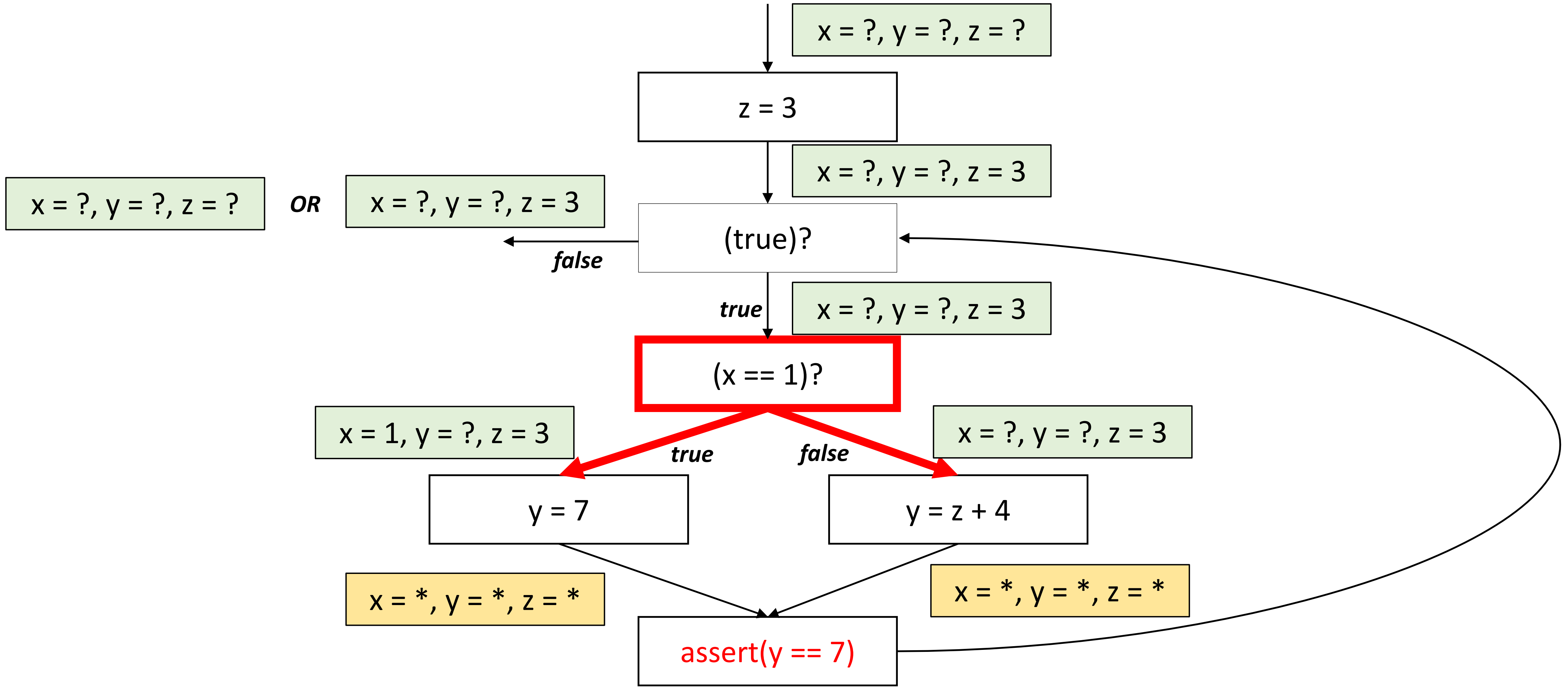


Iterative Approximation

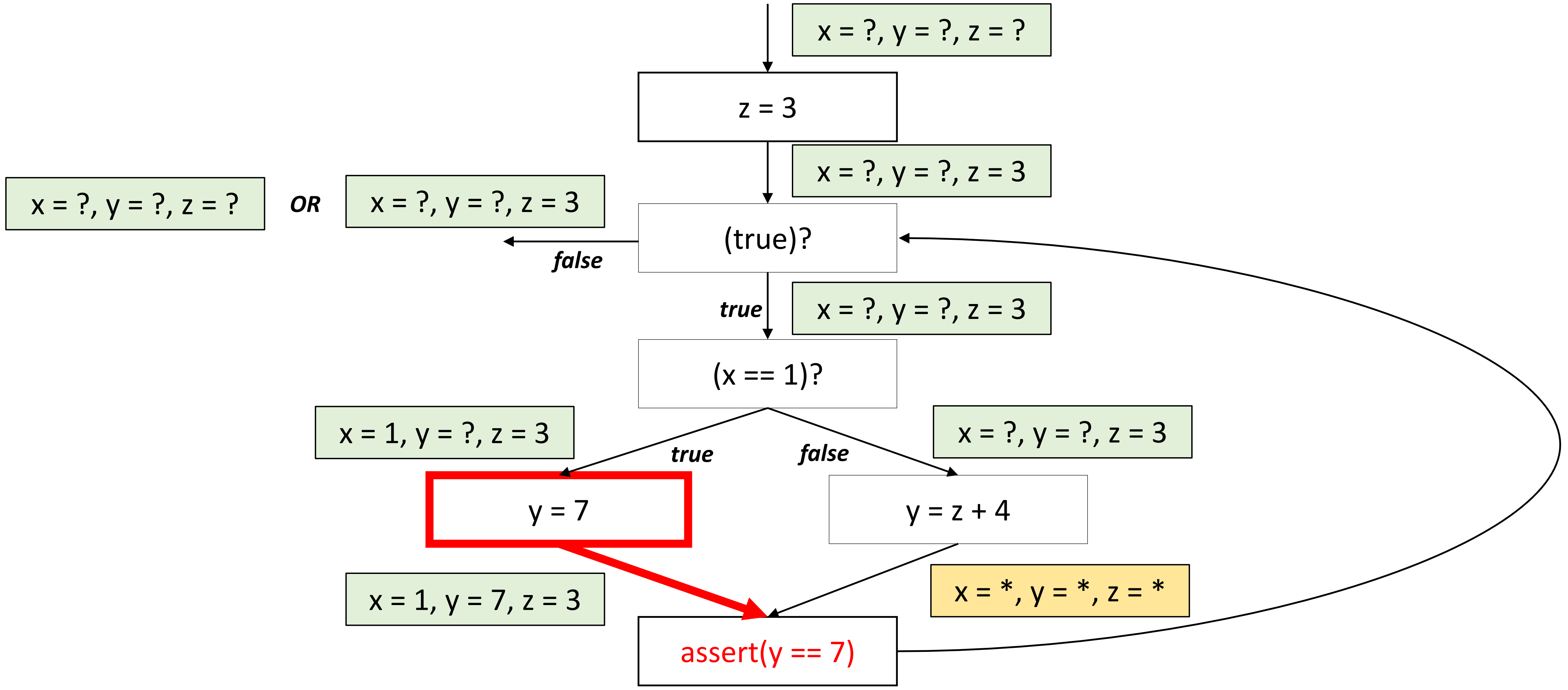
Depending on if conditions are interpreted by analysis



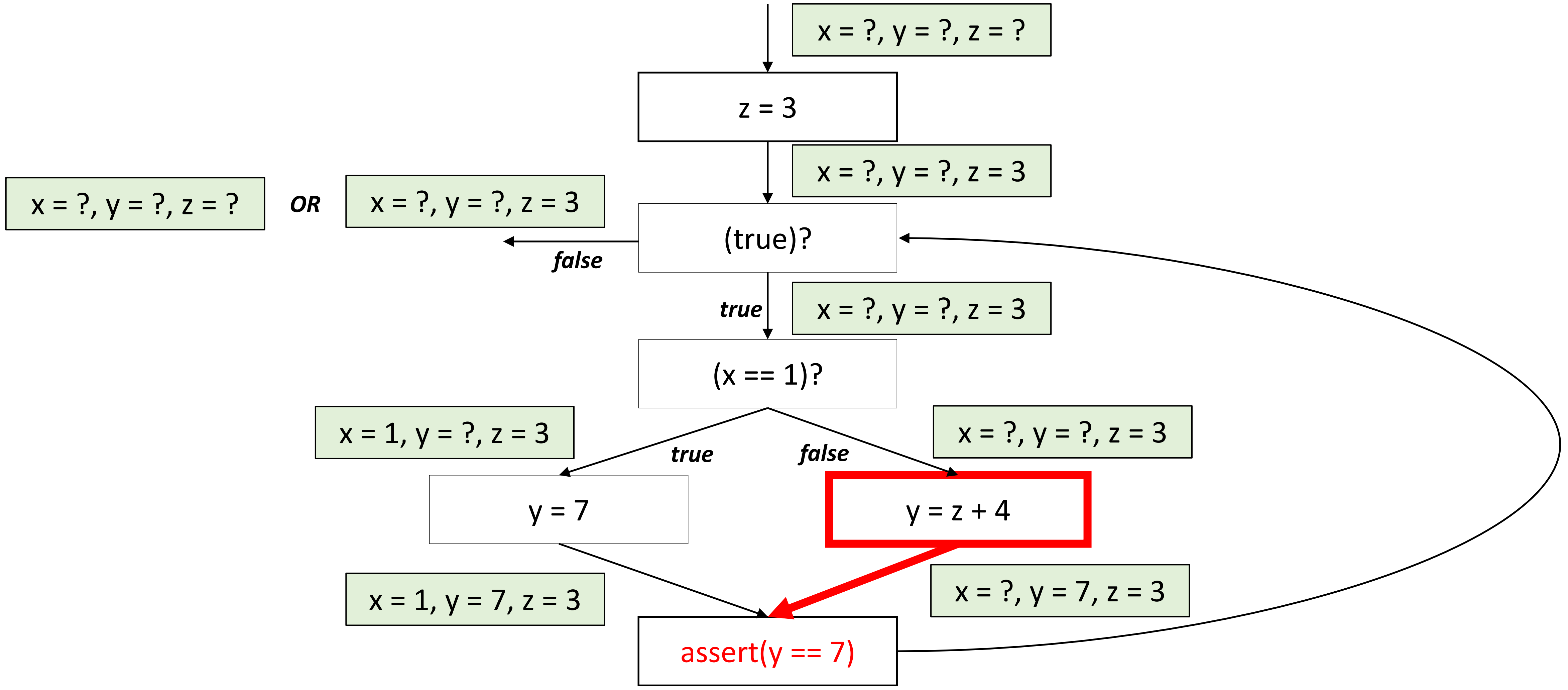
Iterative Approximation



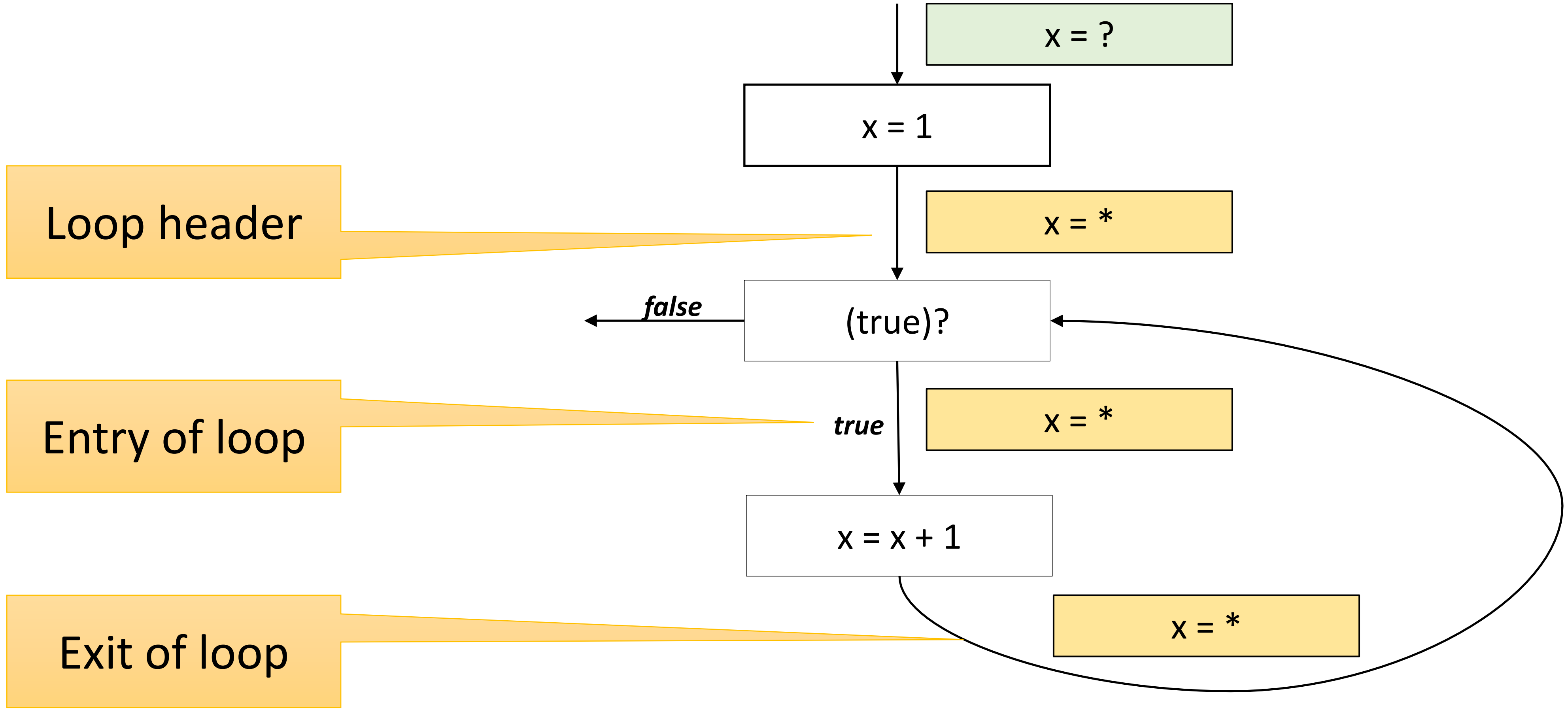
Iterative Approximation



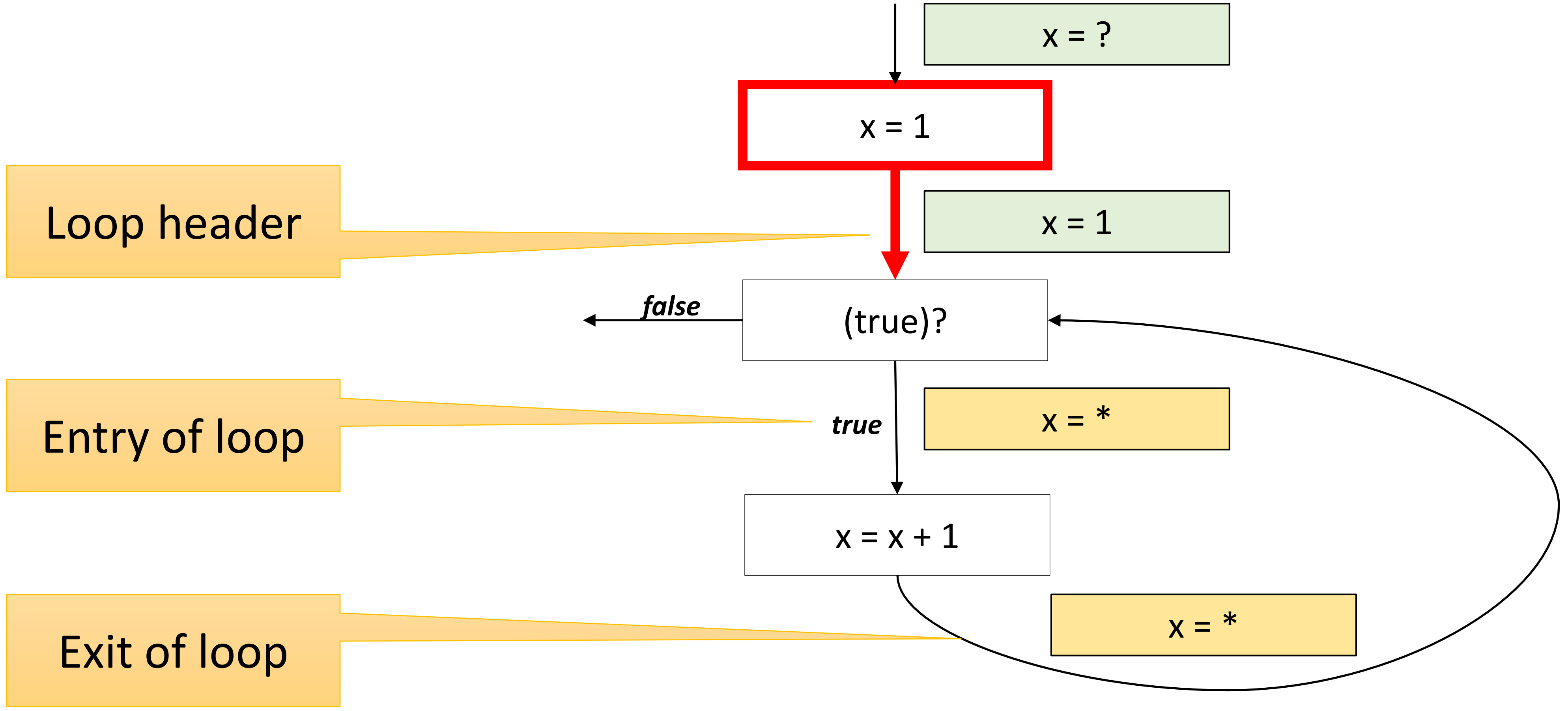
Iterative Approximation



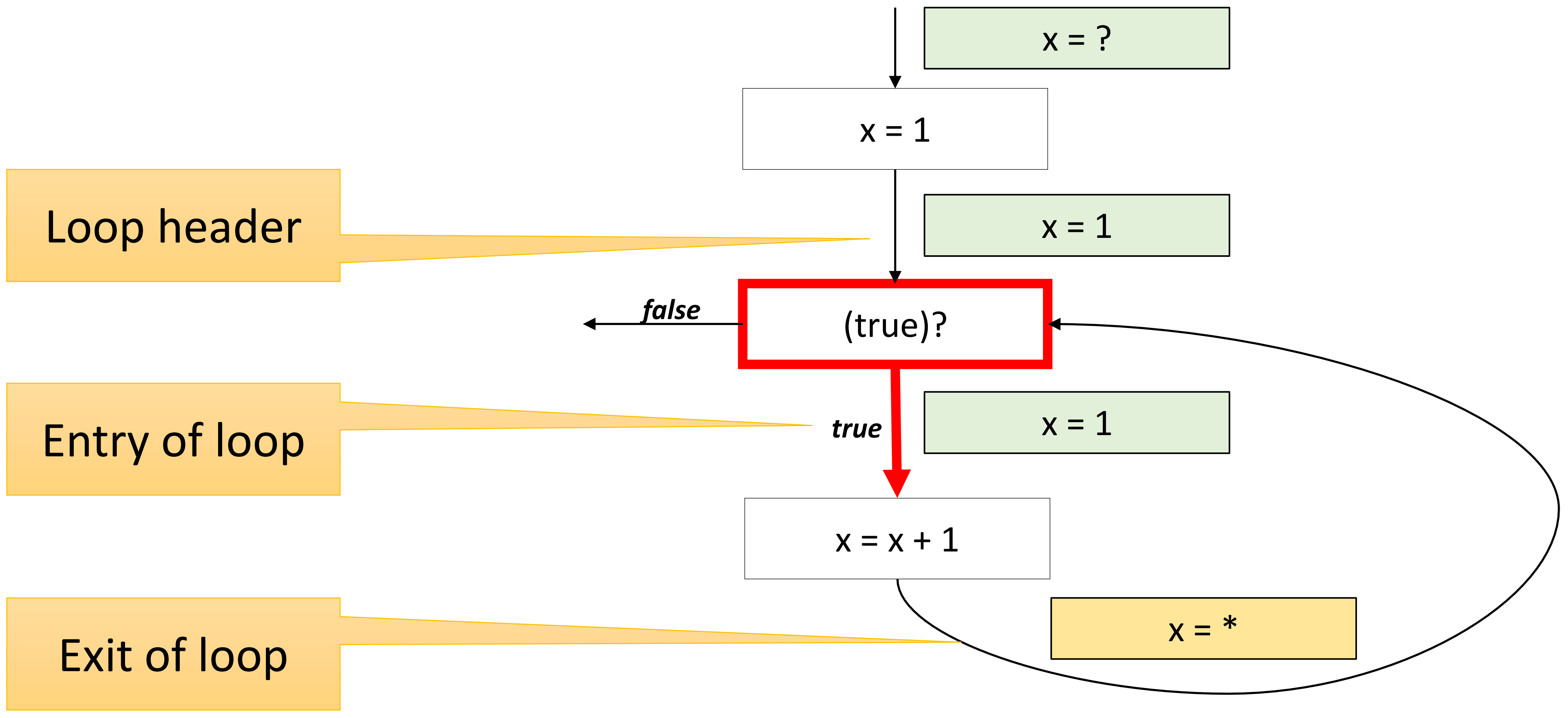
Another Example



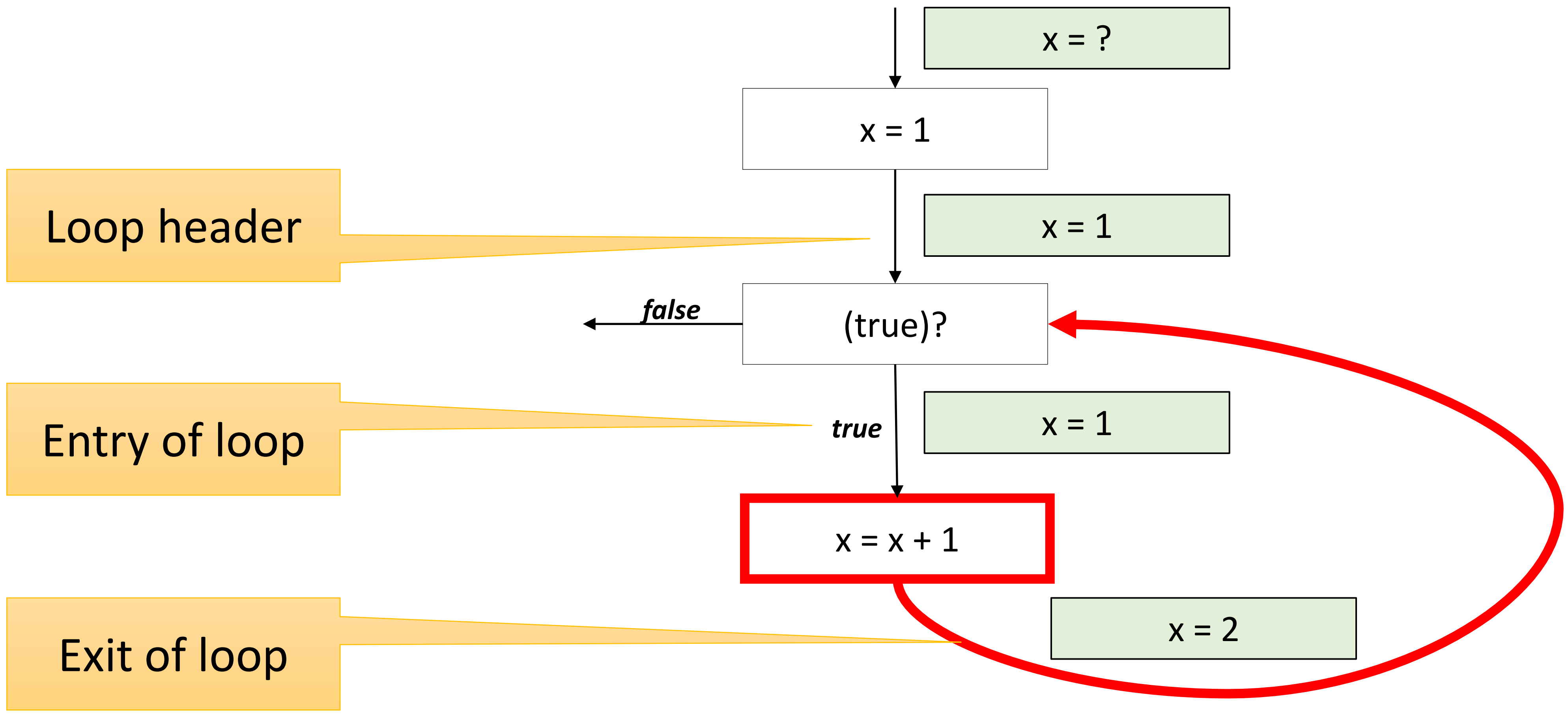
Another Example



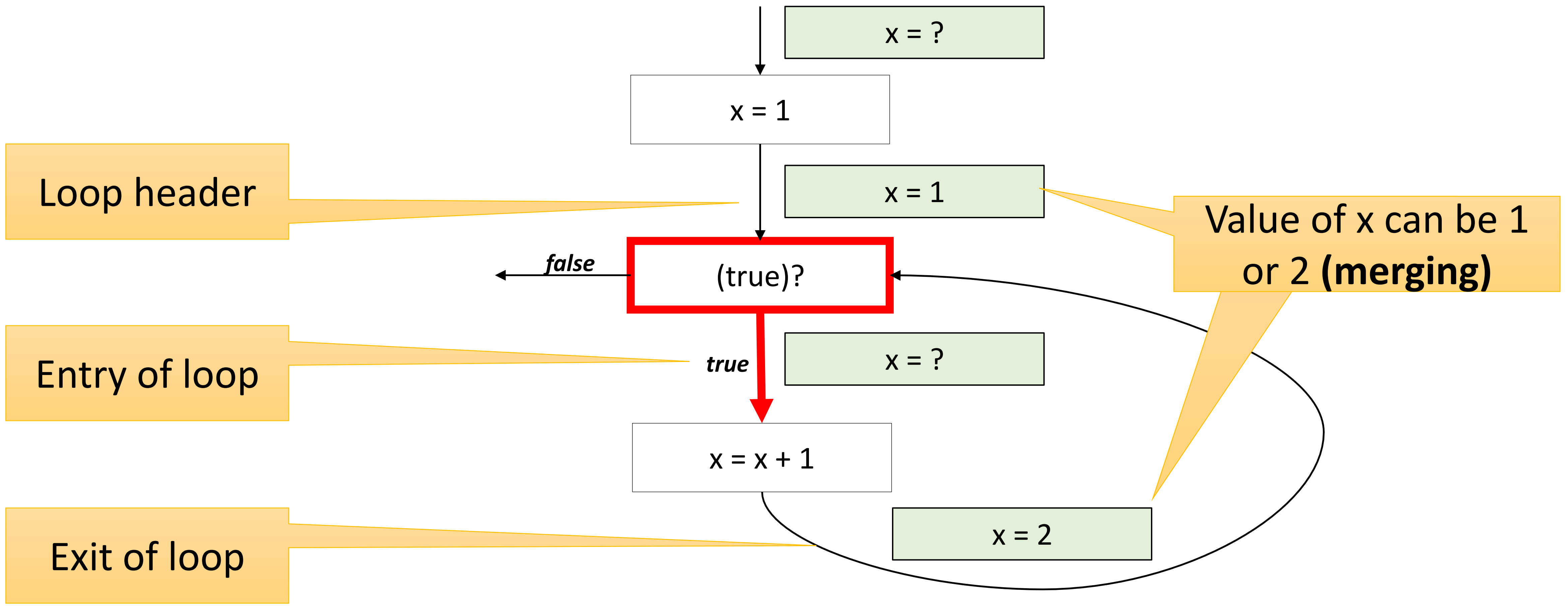
Another Example



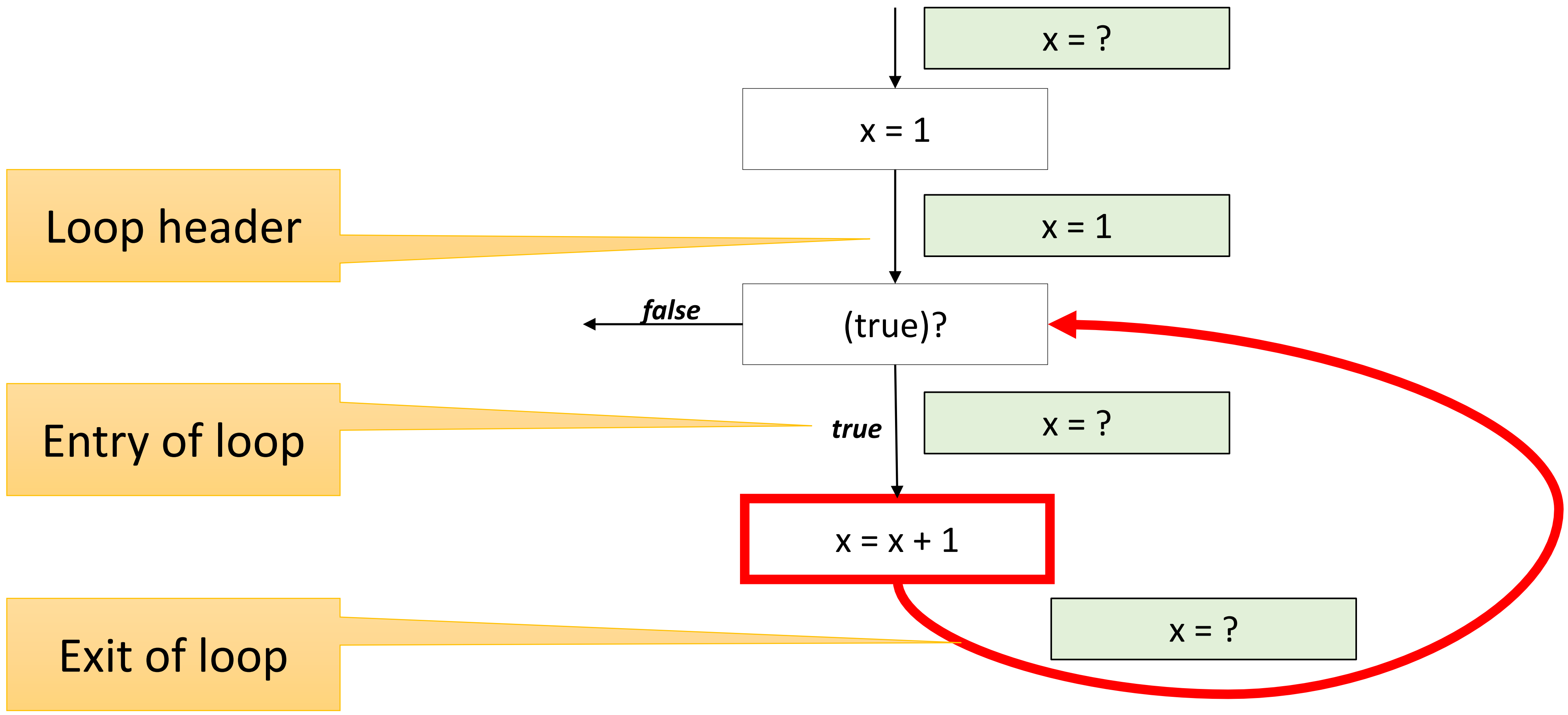
Another Example



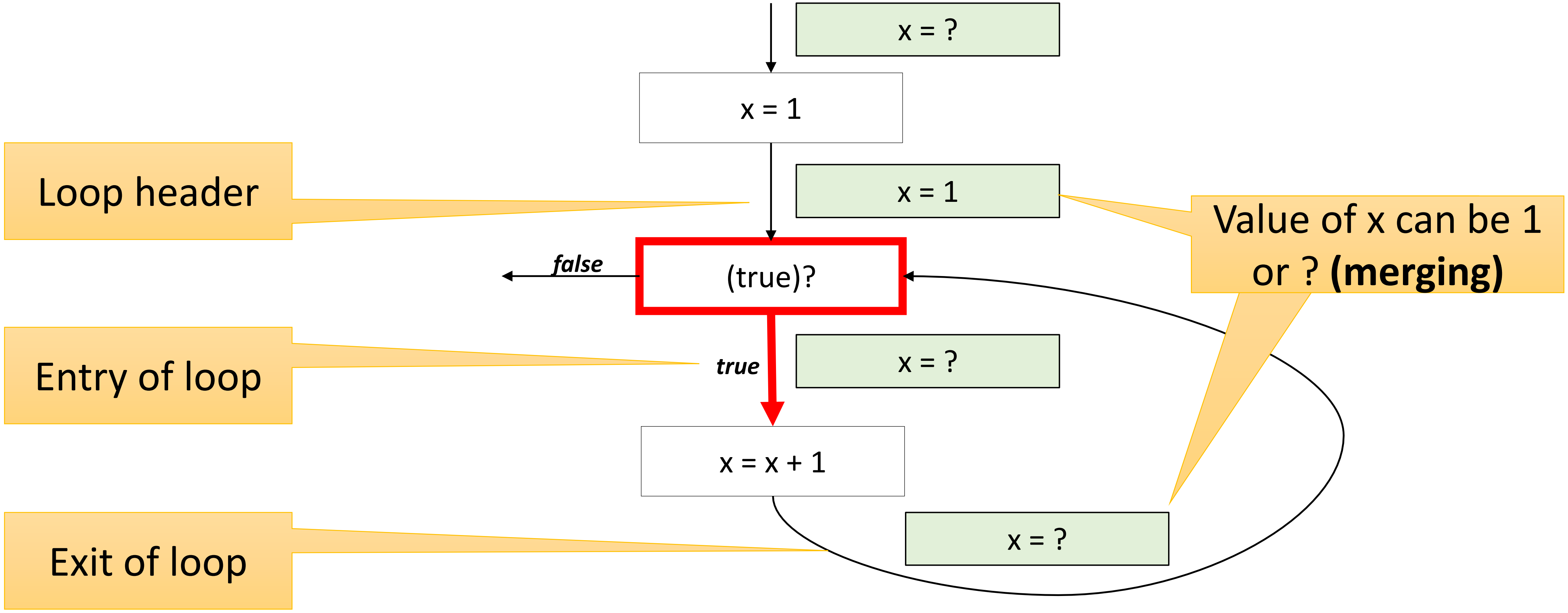
Another Example



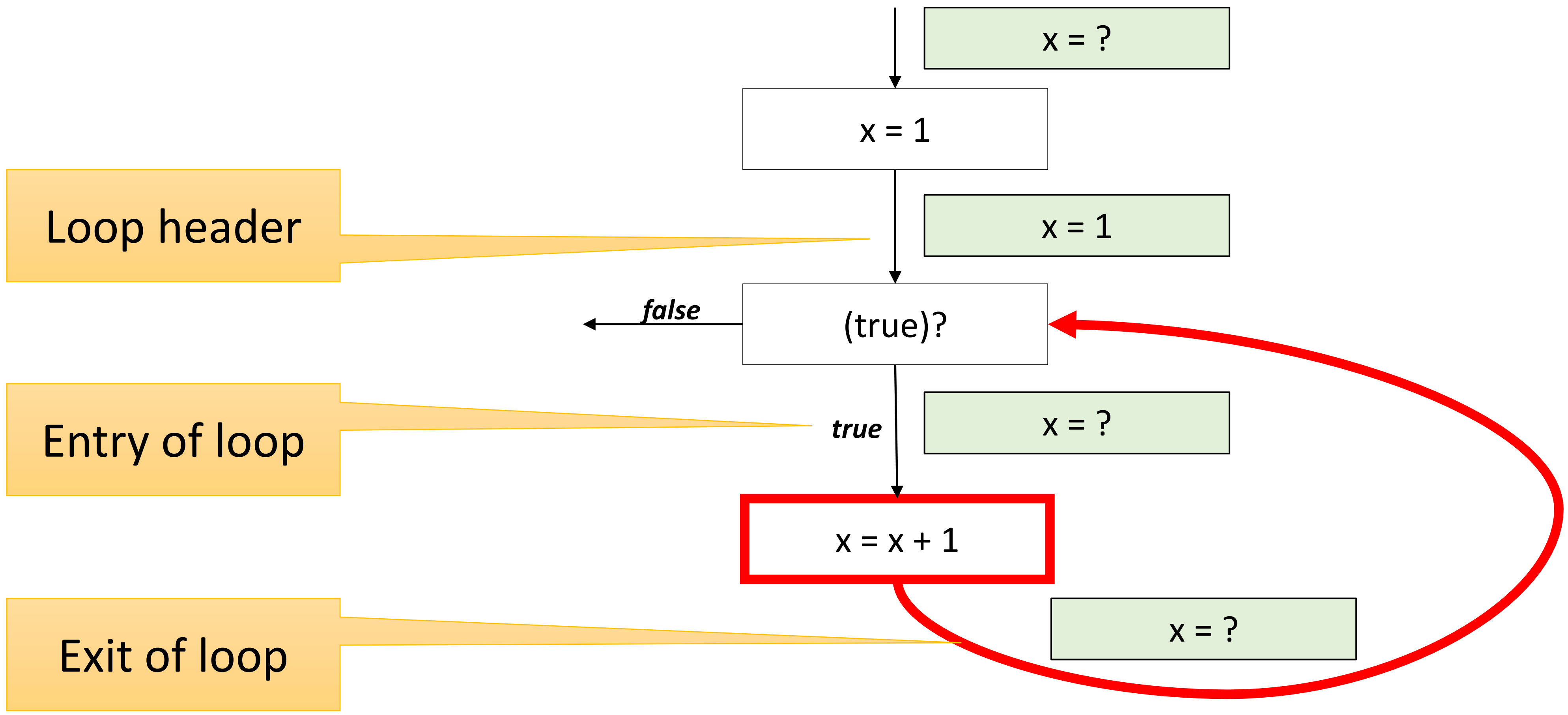
Another Example



Another Example



Another Example



Static Analysis Summary

- **Program representation**
  - How to represent program in a form suitable for analysis (e.g., CFG, AST, Bytecode)
- **Abstract domain**
  - How to approximate values of program variables
- **Transfer functions**
  - How to update values at assignments, conditionals, merge points, etc.
- **Fixed-point computation algorithm**
  - How to run the analysis (e.g., Iterative approximation)

Usefulness in Practice

- Compilers
- Software Quality Tools
- Integrated Development Environments

Compilers

```
int sqr(int x) {
  return x * x;
}

void main() {
  int res;
  if (getc() == ‘a’)
      res = sqr(6) + 6;
  else
      res = sqr(-7) - 7;

  print(res);          res = 42
}
```

```
int sqr(int x) {
  return x * x;
}

void main() {
  print(42);
}
```

Software Quality Tools

- Tools used by software engineers for testing, debugging, and verification
- Use program analysis to
  - Find programming errors
  - Prove program invariants
  - Generate test cases
  - Localize root causes of errors

```
int sqr(int x) {
  return x * x;
}

void main() {
  int res;
  if (getc() == 'a')
      res = sqr(6) + 6;
  else
      res = sqr(-7) - 7;
  if (res!=42)
      disaster();
}
```

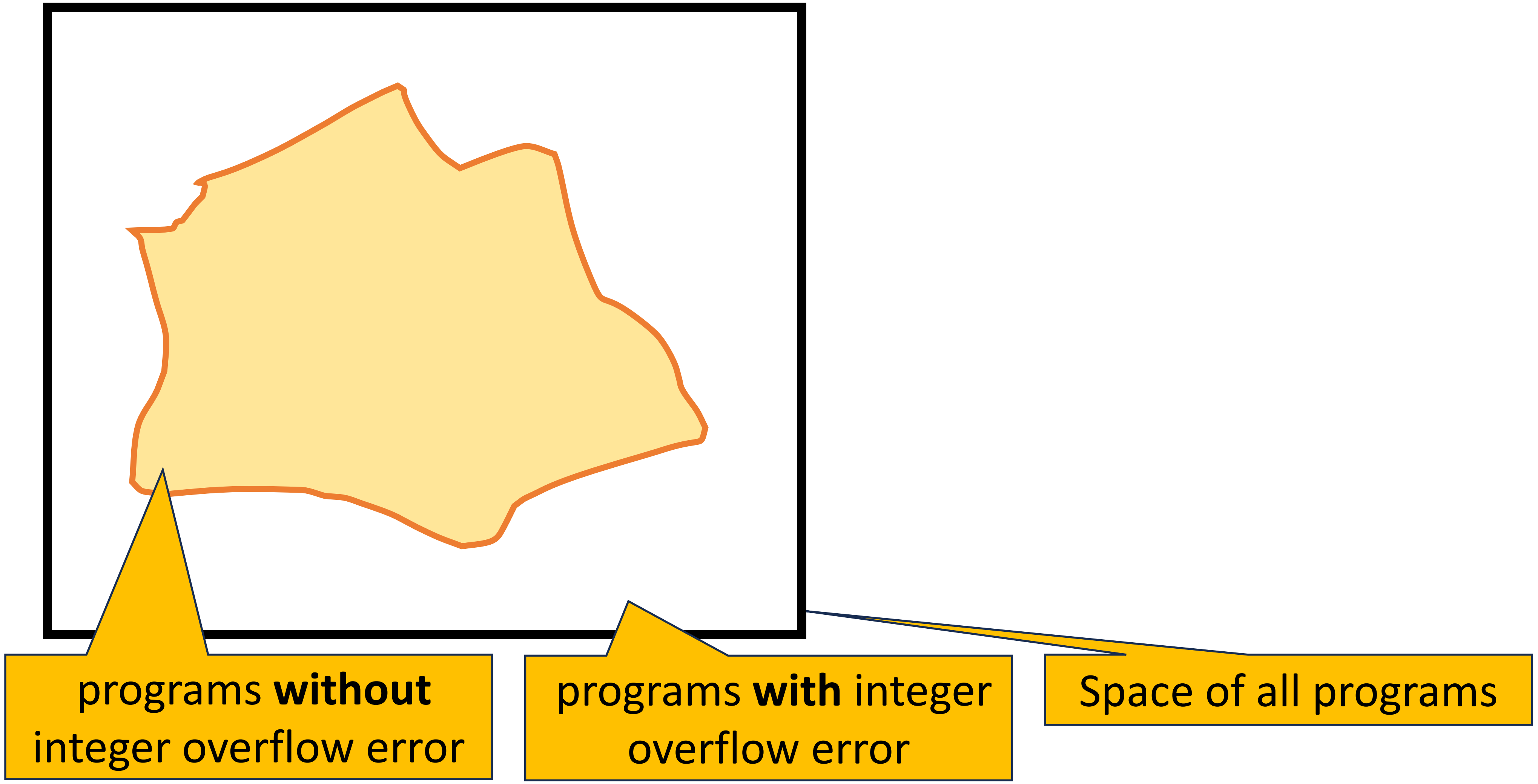
res = 42

Program verification tool can formally prove that disaster will never be invoked

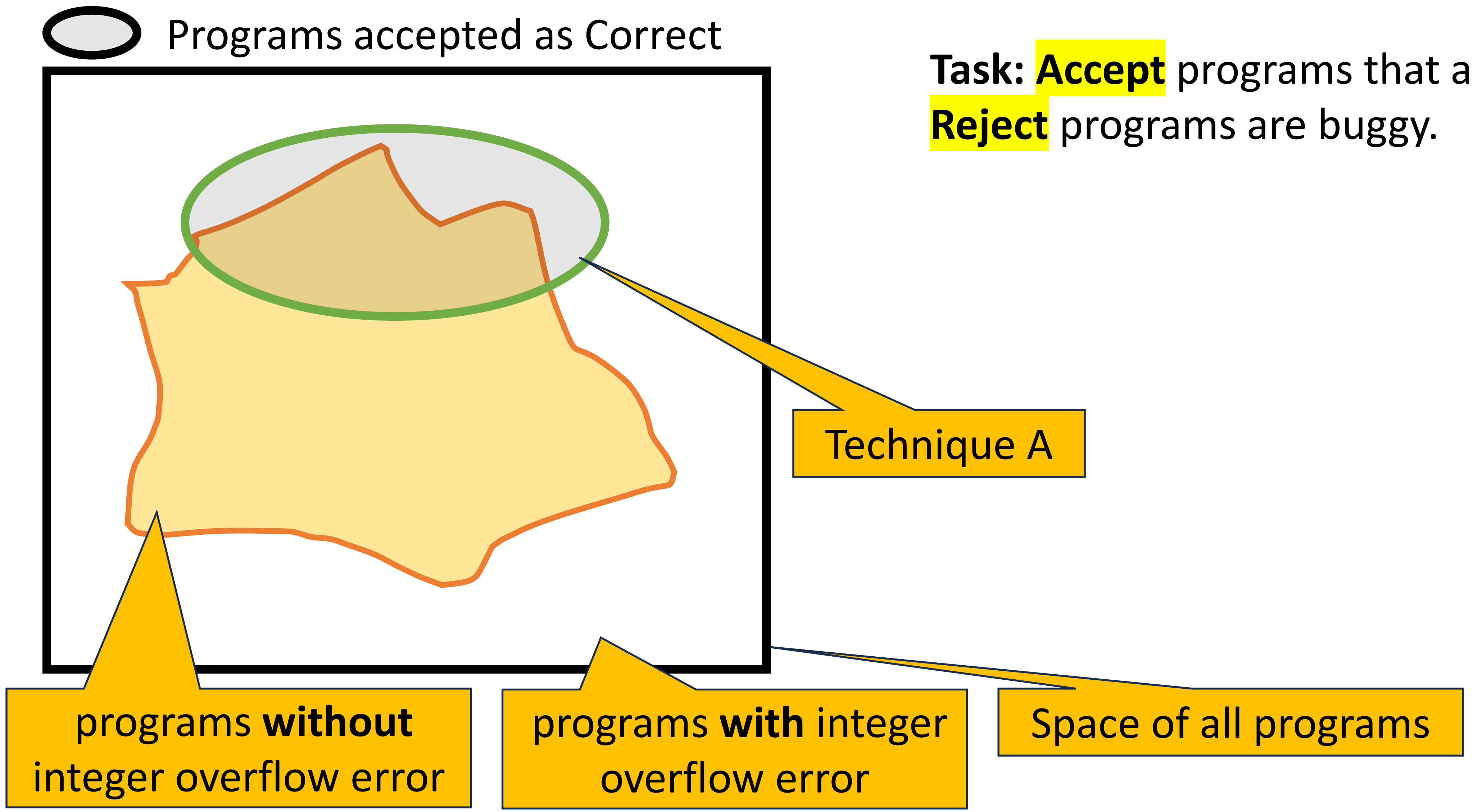
Integrated Development Environments

- Examples: Eclipse, IntelliJ, Microsoft Visual Studio Code, CLion
- Use program analysis to help programmers in:
  - Understanding programs
  - Refactoring programs
    - Restructuring program without changing its behavior
- Useful when working on large complex applications

Quiz: Choosing a Program Analysis


programs **without** integer overflow error
programs **with** integer overflow error
Space of all programs

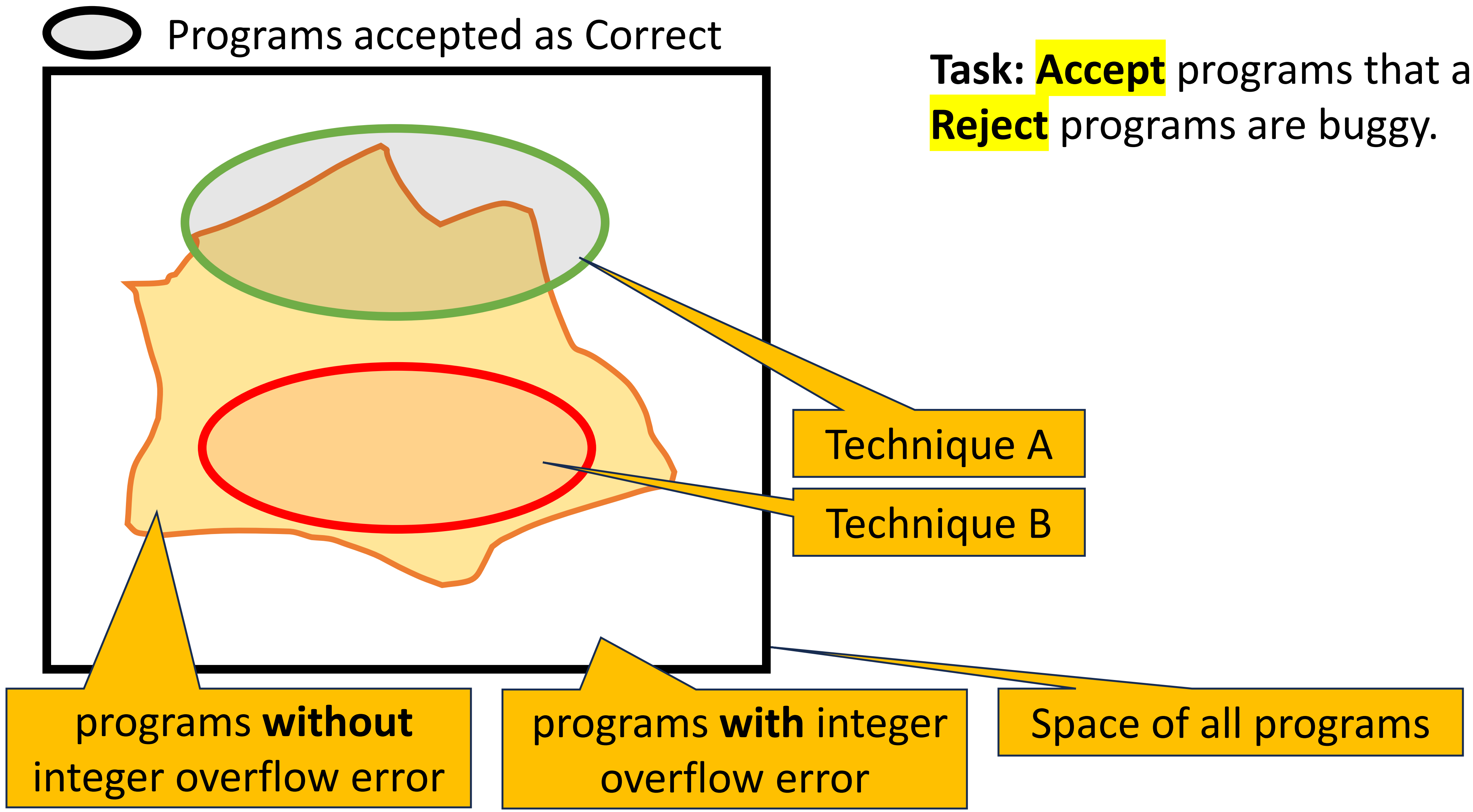
Quiz: Choosing a Program Analysis


Programs accepted as Correct
Technique A
programs **without** integer overflow error
programs **with** integer overflow error
Space of all programs

Task: Accept programs that are correct and **Reject** programs are buggy.

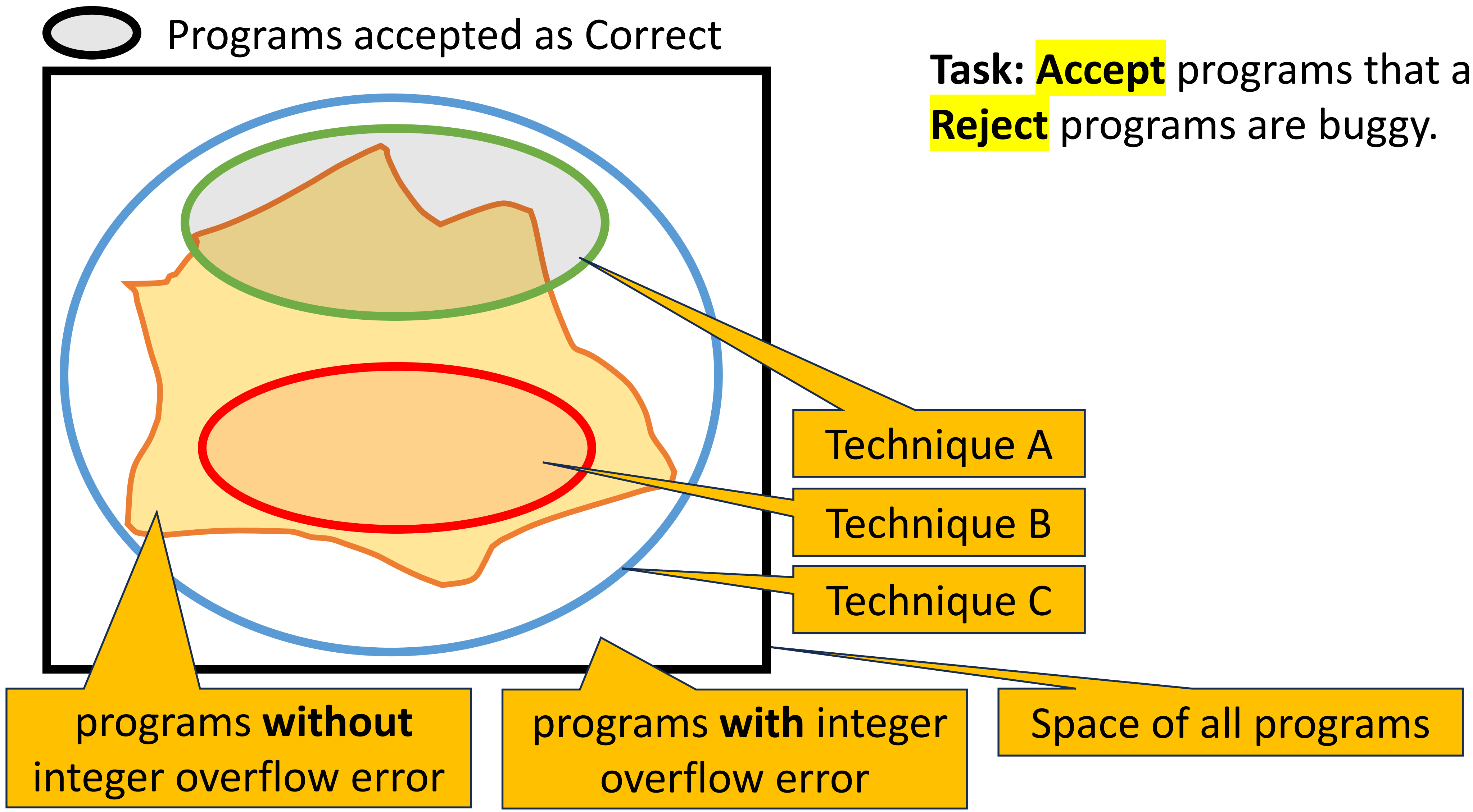
Quiz: Choosing a Program Analysis

Task: Accept programs that are correct and **Reject** programs are buggy.

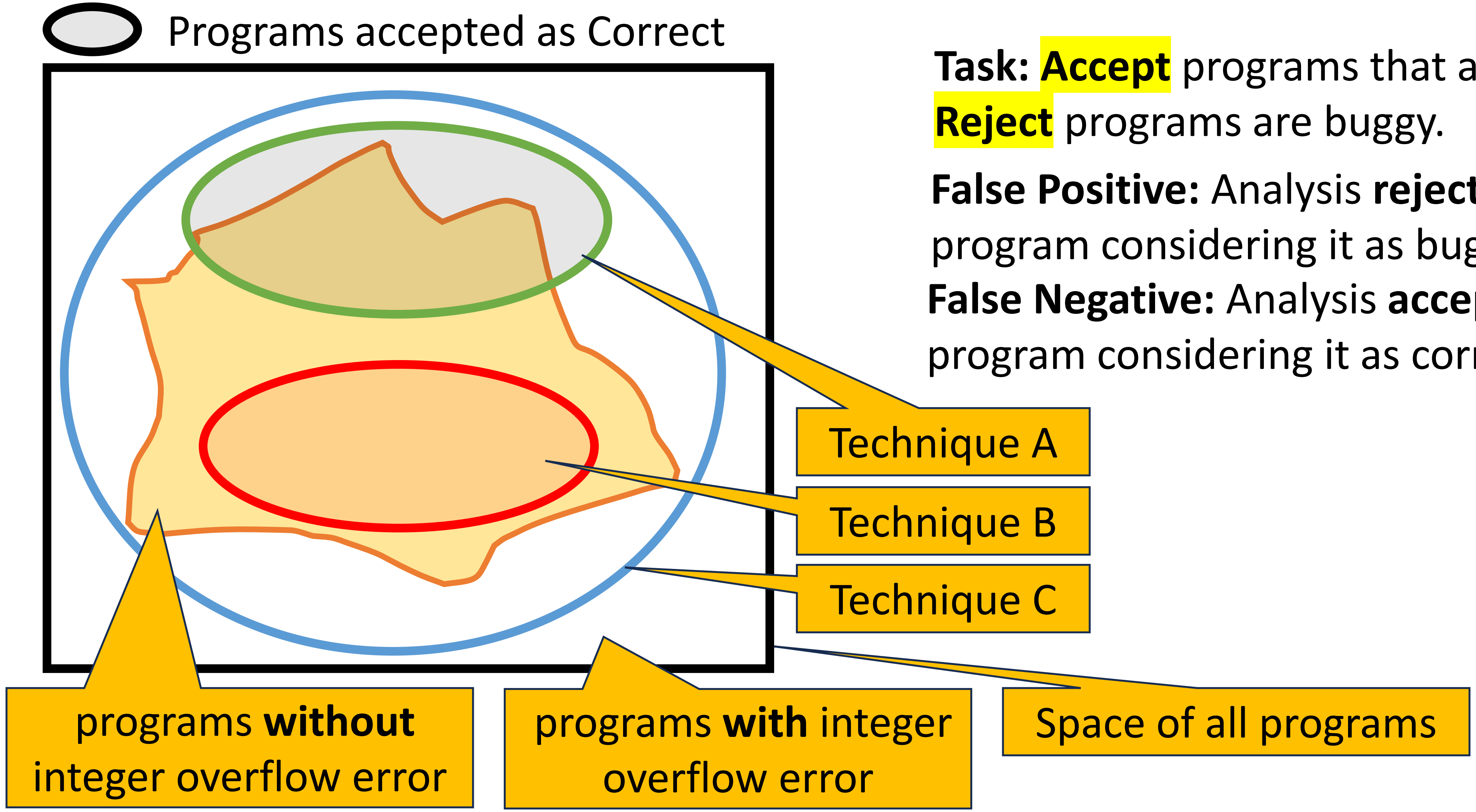

Programs accepted as Correct
Technique A
Technique B
programs **without** integer overflow error
programs **with** integer overflow error
Space of all programs

Quiz: Choosing a Program Analysis

Task: Accept programs that are correct and **Reject** programs are buggy.


Programs accepted as Correct
Technique A
Technique B
Technique C
programs **without** integer overflow error
programs **with** integer overflow error
Space of all programs

Quiz: Choosing a Program Analysis

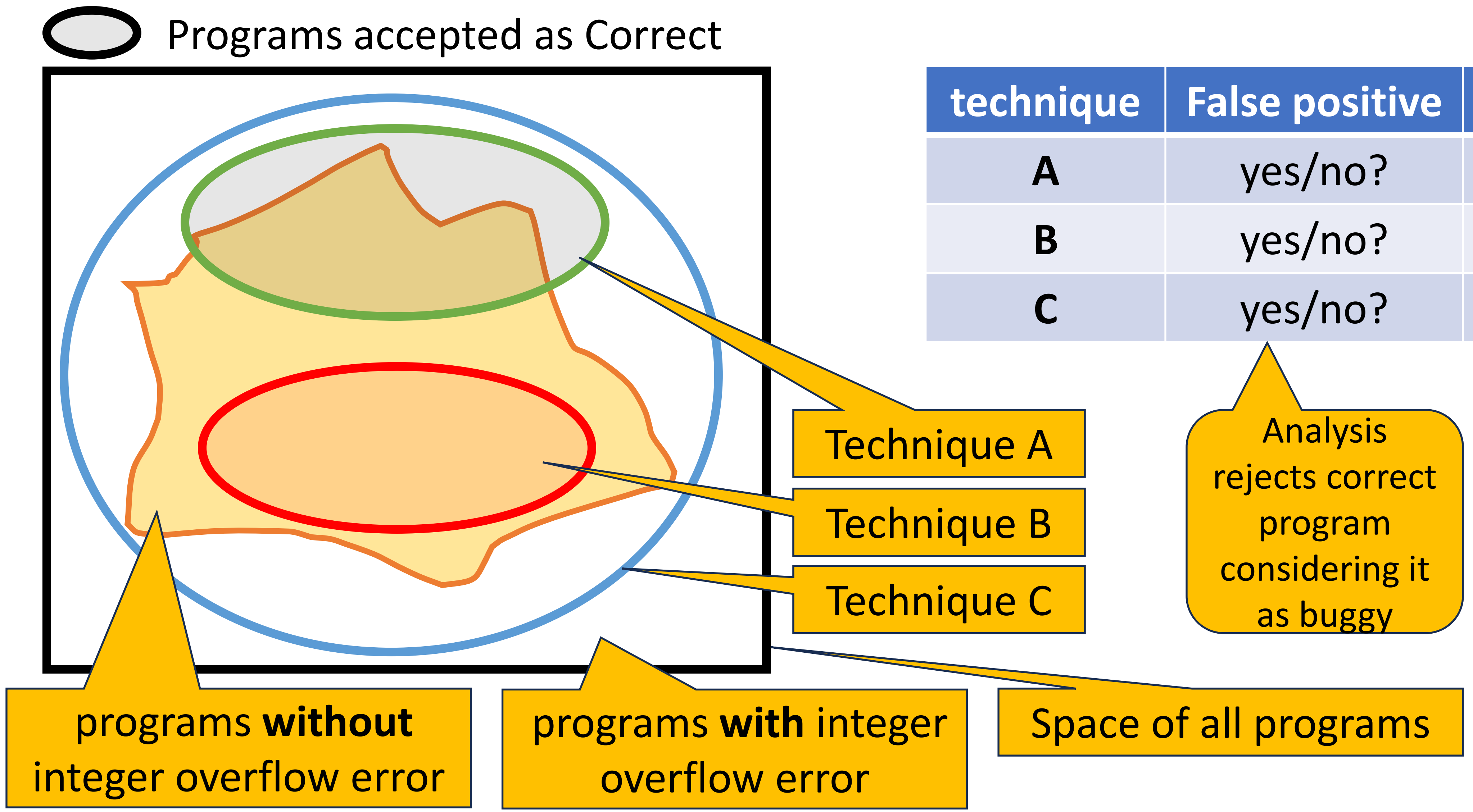

Programs accepted as Correct
Technique A
Technique B
Technique C
programs **without** integer overflow error
programs **with** integer overflow error
Space of all programs

Task: Accept programs that are correct and **Reject** programs are buggy.

False Positive: Analysis **rejects** correct program considering it as buggy

False Negative: Analysis **accepts** buggy program considering it as correct

Quiz (1/3): Choosing a Program Analysis

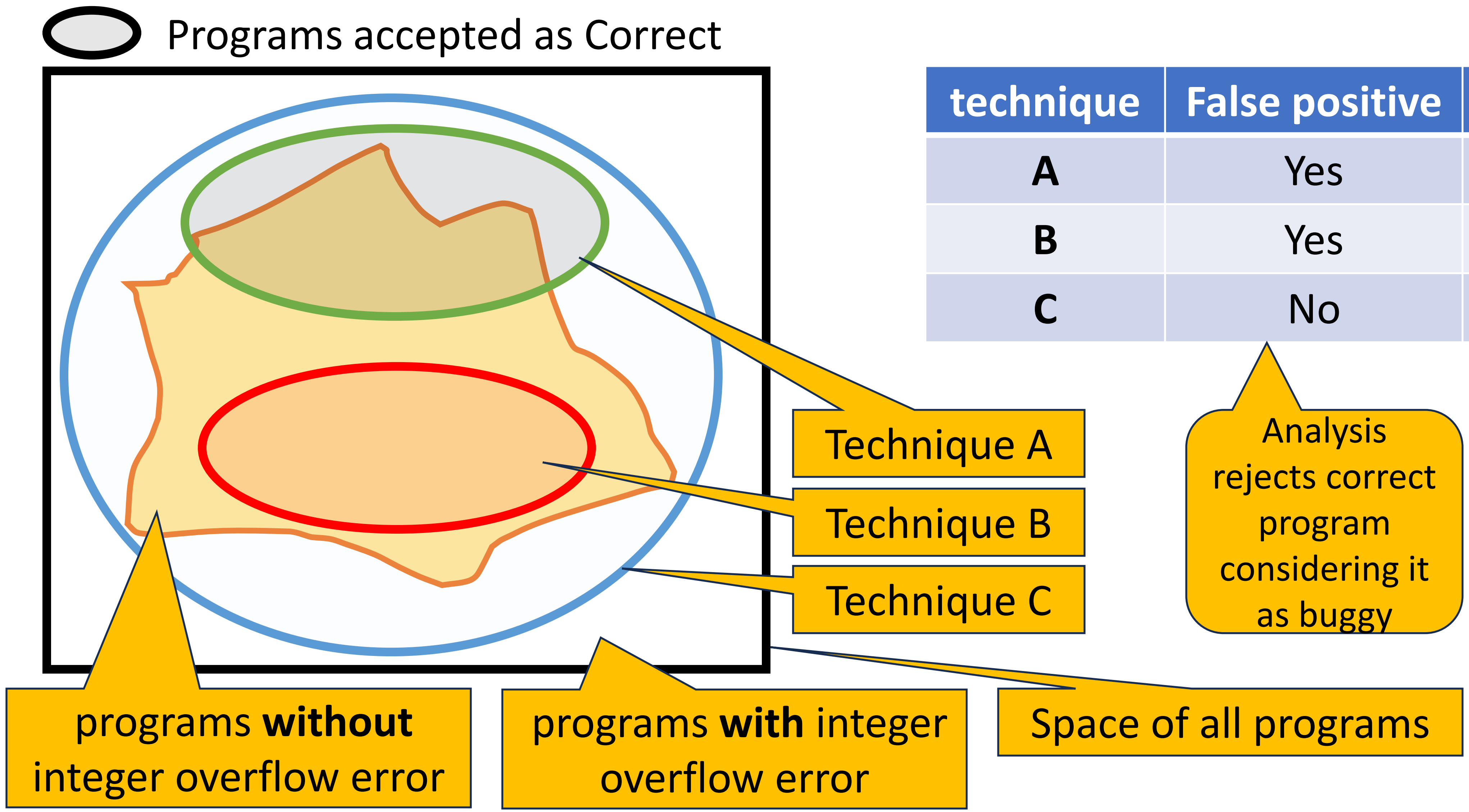

Programs accepted as Correct
Technique A
Technique B
Technique C
programs **without** integer overflow error
programs **with** integer overflow error
Space of all programs

technique	False positive	False negative
A	yes/no?	yes/no?
B	yes/no?	yes/no?
C	yes/no?	yes/no?

Analysis rejects correct program considering it as buggy

Analysis accepts buggy program considering it as correct

Quiz (1/3): Choosing a Program Analysis


Programs accepted as Correct
Technique A
Technique B
Technique C
programs **without** integer overflow error
programs **with** integer overflow error
Space of all programs

technique	False positive	False negative
A	Yes	Yes
B	Yes	No
C	No	Yes

Analysis rejects correct program considering it as buggy

Analysis accepts buggy program considering it as correct

Quiz (2/3): Choosing a Program Analysis

- NASA engineer Bob has been asked to apply program analysis to check for integer overflow errors in software that will control NASA's next billion-dollar space mission.
- Which analysis (A / B / C) should Bob use? Justify your answer.

technique	False positive	False negative
A	Yes	Yes
B	Yes	No
C	No	Yes

Quiz (2/3): Choosing a Program Analysis

- NASA engineer Bob has been asked to apply program analysis to check for integer overflow errors in software that will control NASA's next billion-dollar space mission.
- Which analysis (A / B / C) should Bob use? Justify your answer.

technique	False positive	False negative
A	Yes	Yes
B	Yes	No
C	No	Yes

Ans) B. *Safety critical systems like infrastructure systems in healthcare, aviation and aerospace software, and real time systems have high risk and cost -- lives depend on them! If the program is accepted by B, then Bob can be sure there are no integer overflow errors in the code.*

Quiz (3/3): Choosing a Program Analysis

- Microsoft developer Ann has agreed to apply program analysis to check for integer overflow errors in her programs.
- Which analysis (A / B / C) should Ann use? Justify your answer.

technique	False positive	False negative
A	Yes	Yes
B	Yes	No
C	No	Yes

Quiz (3/3): Choosing a Program Analysis

- Microsoft developer Ann has agreed to apply program analysis to check for integer overflow errors in her programs.
- Which analysis (A / B / C) should Ann use? Justify your answer.

technique	False positive	False negative
A	Yes	Yes
B	Yes	No
C	No	Yes

Ans) C*. Software developers quickly loose confidence in tools that throw many false alarms. If the program is rejected by C, then Ann can be sure there are integer overflow errors in that code.*

What have we learnt?

- Detecting invariants using dynamic analysis
- Detecting invariants using static analysis
- Useful of analysis techniques in practice
- How to decide which analysis is useful in a given context

More Announcements

- No class on Monday, 1/15 (MLK day), **next class will be on Wednesday, 1/17**
- **Paper Presentation** assignment will be released on **Monday, 1/15**.
- It will be done in groups of **up to three students**. The groups will be determined by who signs up for each paper.
- To sign up, add yourself to one of the "Paper Selection" groups at https://canvas.oregonstate.edu/courses/1964338/groups#tab-110068. For more details, see the assignment on canvas.
- Sign ups will open on **Monday, Jan 15, 2024, 2:00 PM PT** and will be done on a first-come first-served basis.
- Students who do not sign up by **Monday, Jan 22, 2024, 11:59 PM PT** will be randomly assigned to unassigned papers.

Reminder

Form your project team of **up to 5** by adding yourself to one of the "Project" groups at https://canvas.oregonstate.edu/courses/1964338/groups#tab-110070 by **Wednesday, Jan 17, 2024, 11:59 PM PT**.

Next Class

Software Specifications