Static Analysis

Software Reengineering
(COM3523 / COM6523)

The University of Sheffield

Source Code

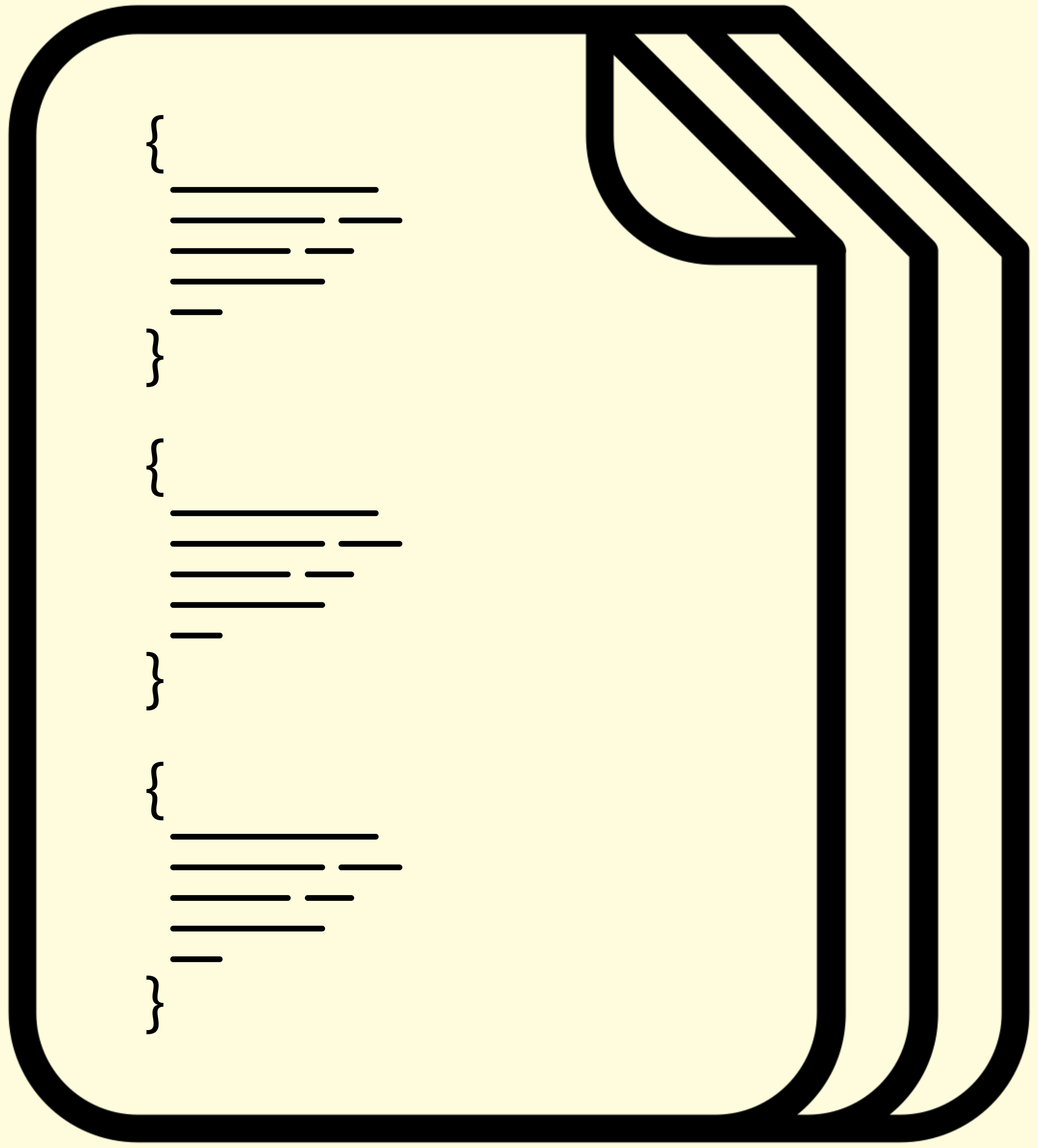
Definitive record of software structure and behaviour.

Commonly changed when system is reengineered.

Source Code

Definitive record of software structure and behaviour.

Commonly changed when system is reengineered.



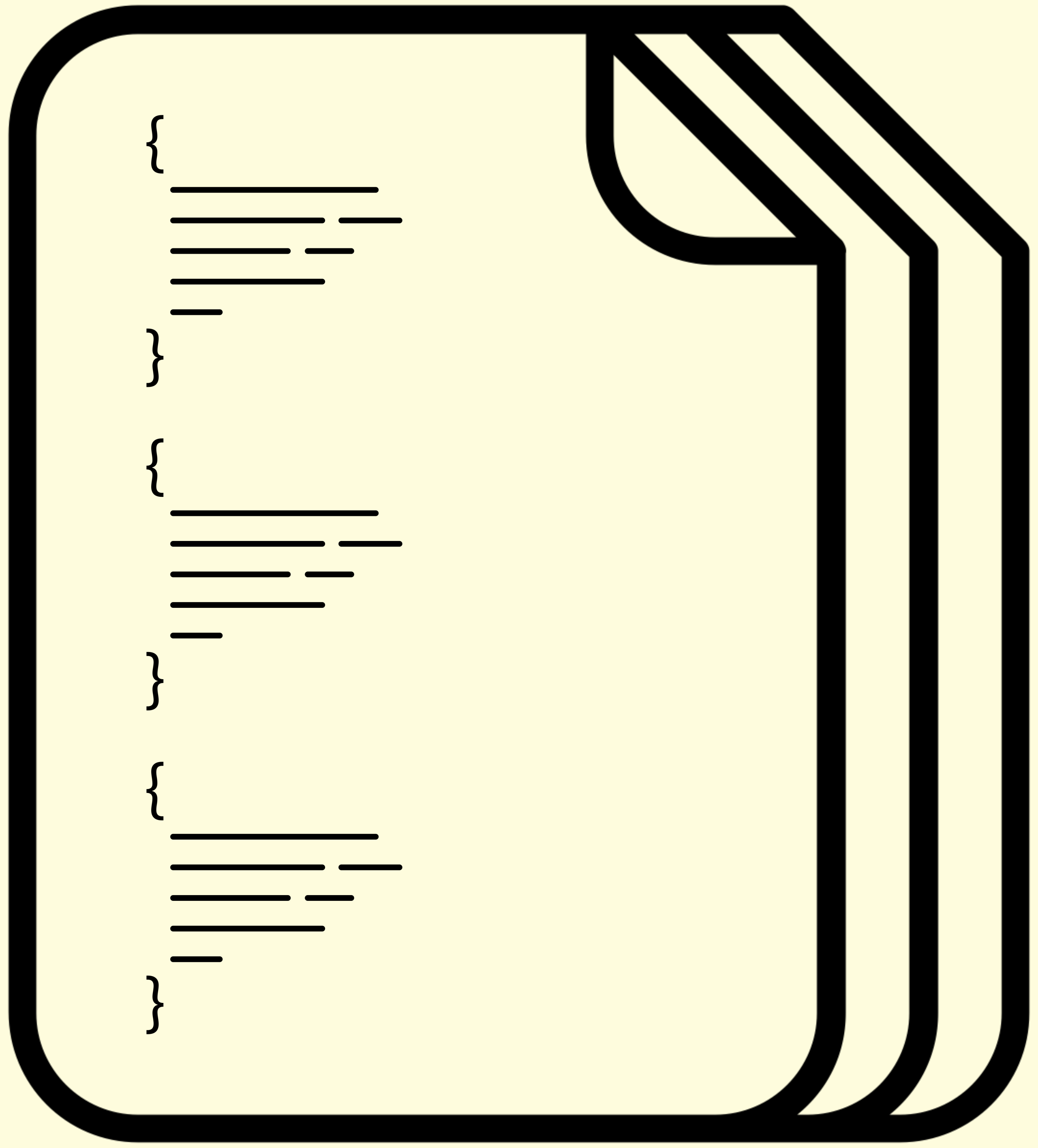
Source Code

Definitive record of software structure and behaviour.

Commonly changed when system is reengineered.

Difficult to understand because it is:

- Big - hundreds of thousands or millions of lines of code.
- Complex - highly interconnected.
- Poorly designed - having deteriorated over decades.



Source Code

Definitive record of software structure and behaviour.

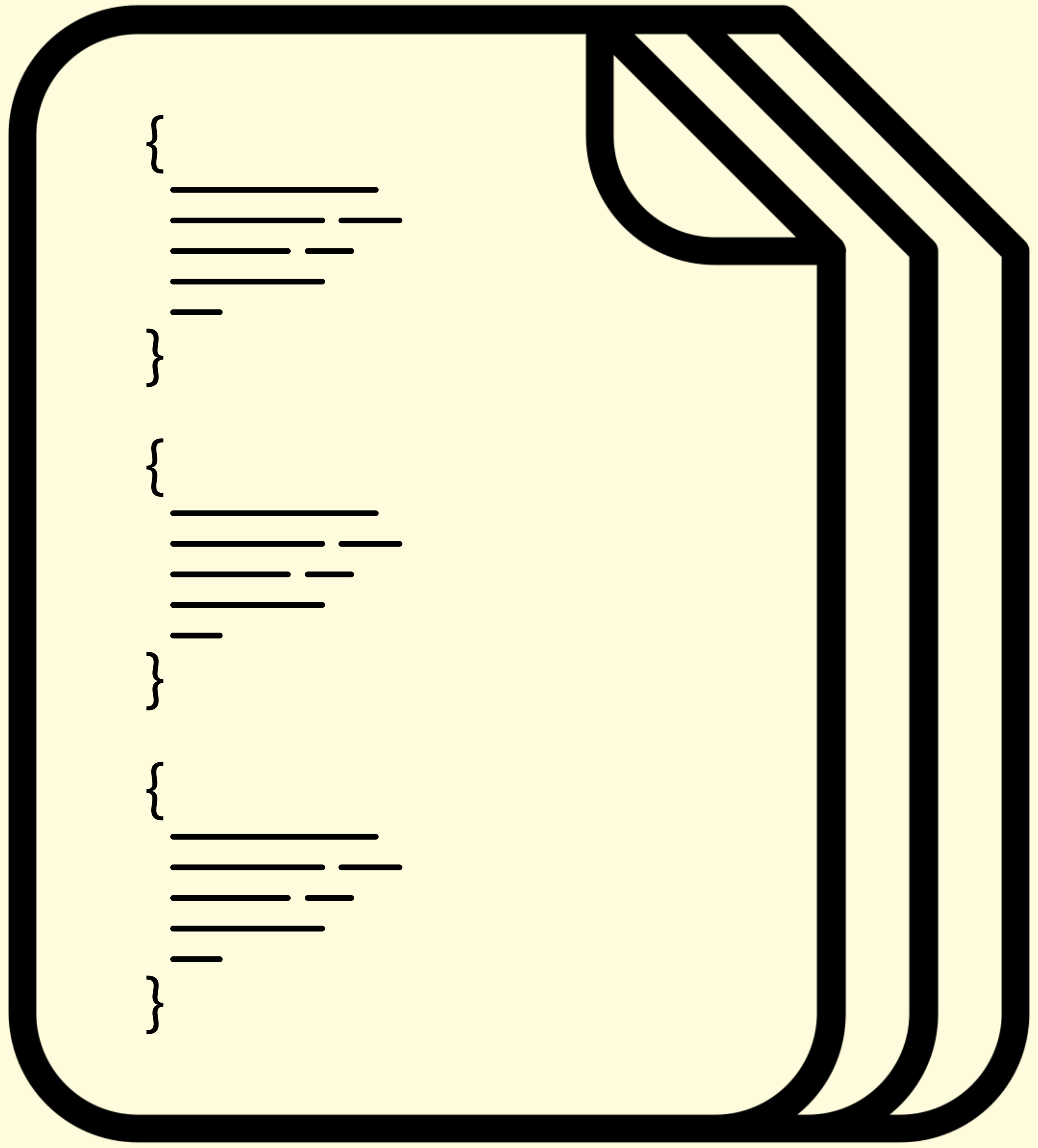
Commonly changed when system is reengineered.

Difficult to understand because it is:

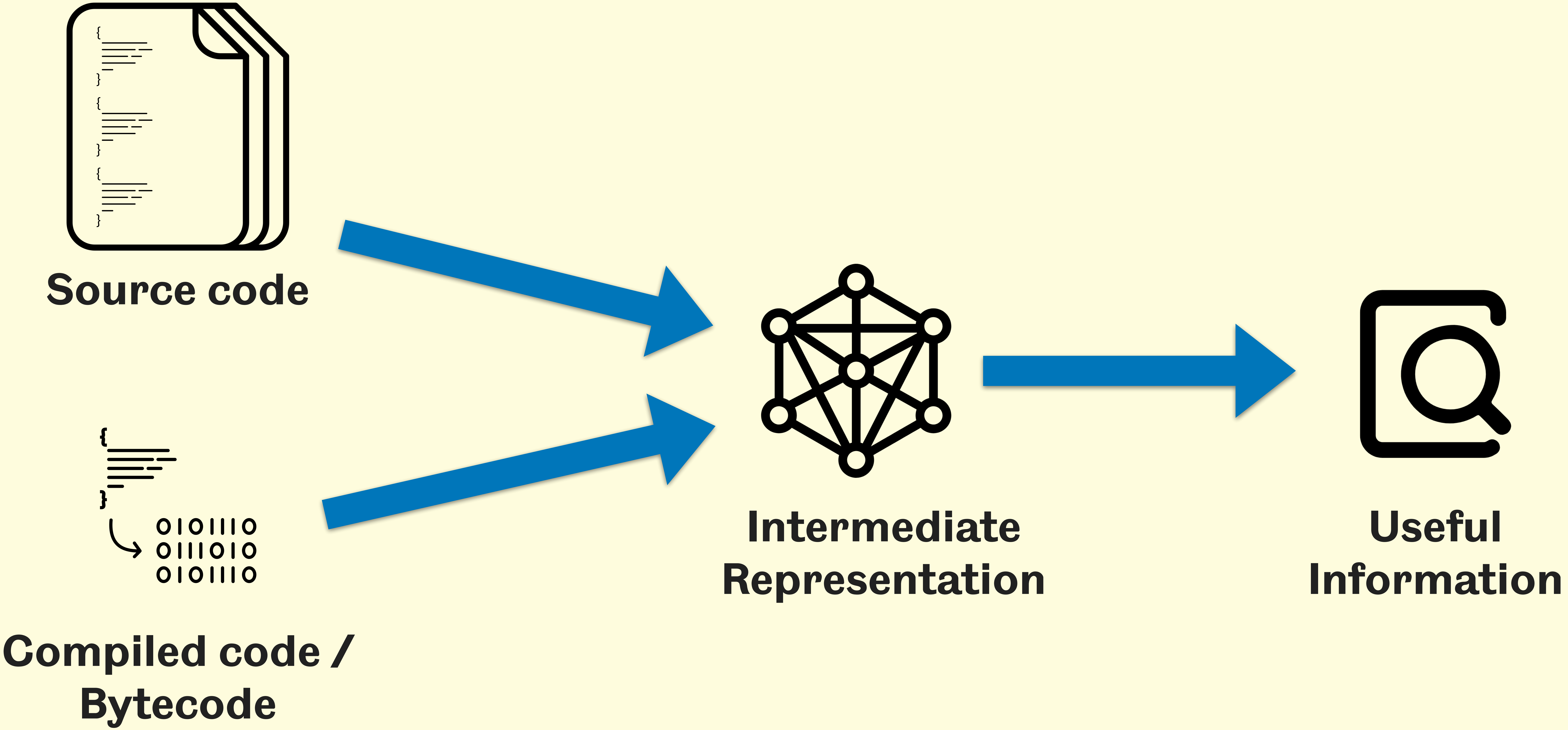
- Big - hundreds of thousands or millions of lines of code.
- Complex - highly interconnected.
- Poorly designed - having deteriorated over decades.

Loaded with latent information about what the system *should* be doing.

- Identifiers - variable, method, and class names.
- Dependencies - calls between functions indicate hidden relationships…



The source code analysis process



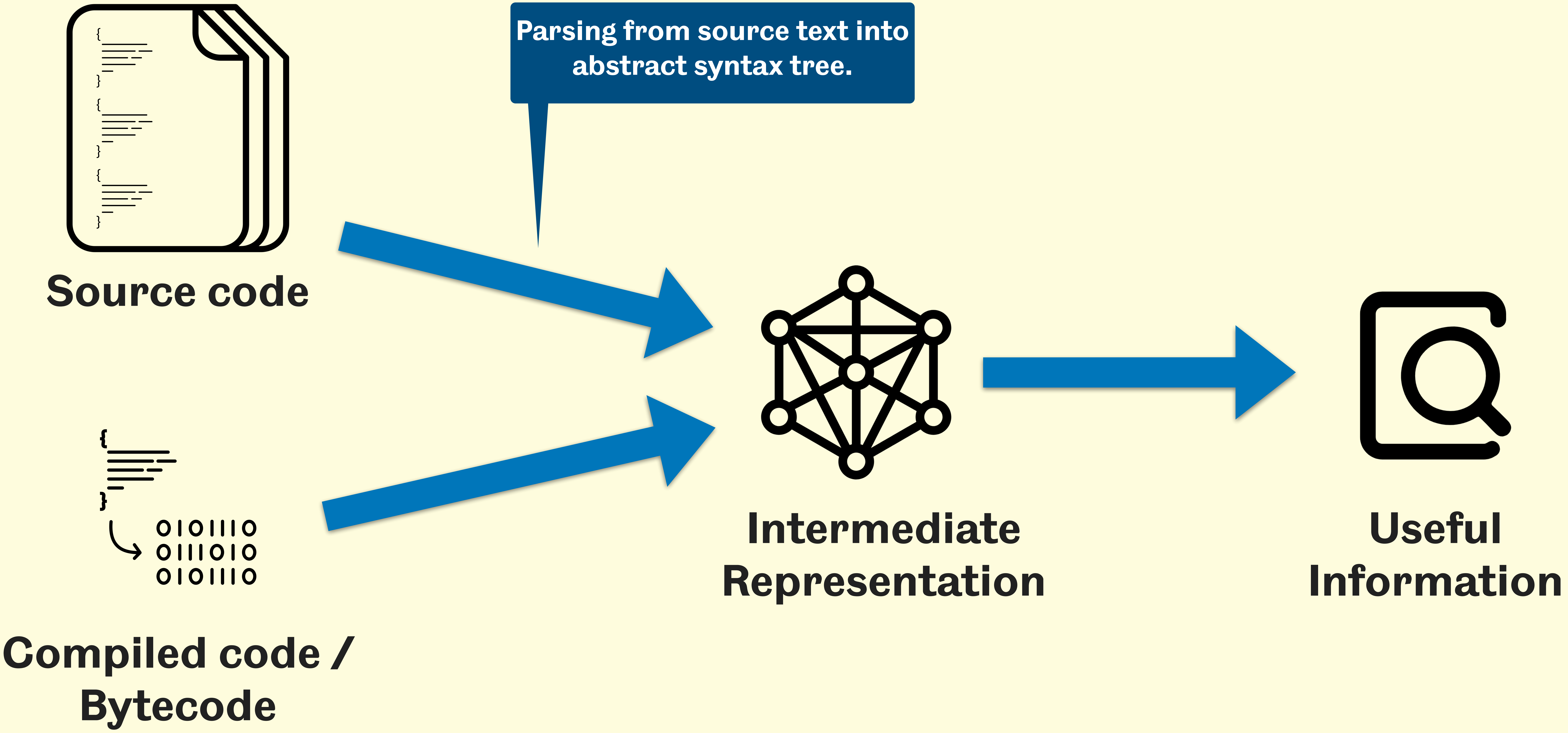
Source code

Compiled code / Bytecode

Intermediate Representation

Useful Information

The source code analysis process



Parsing from source text into abstract syntax tree.

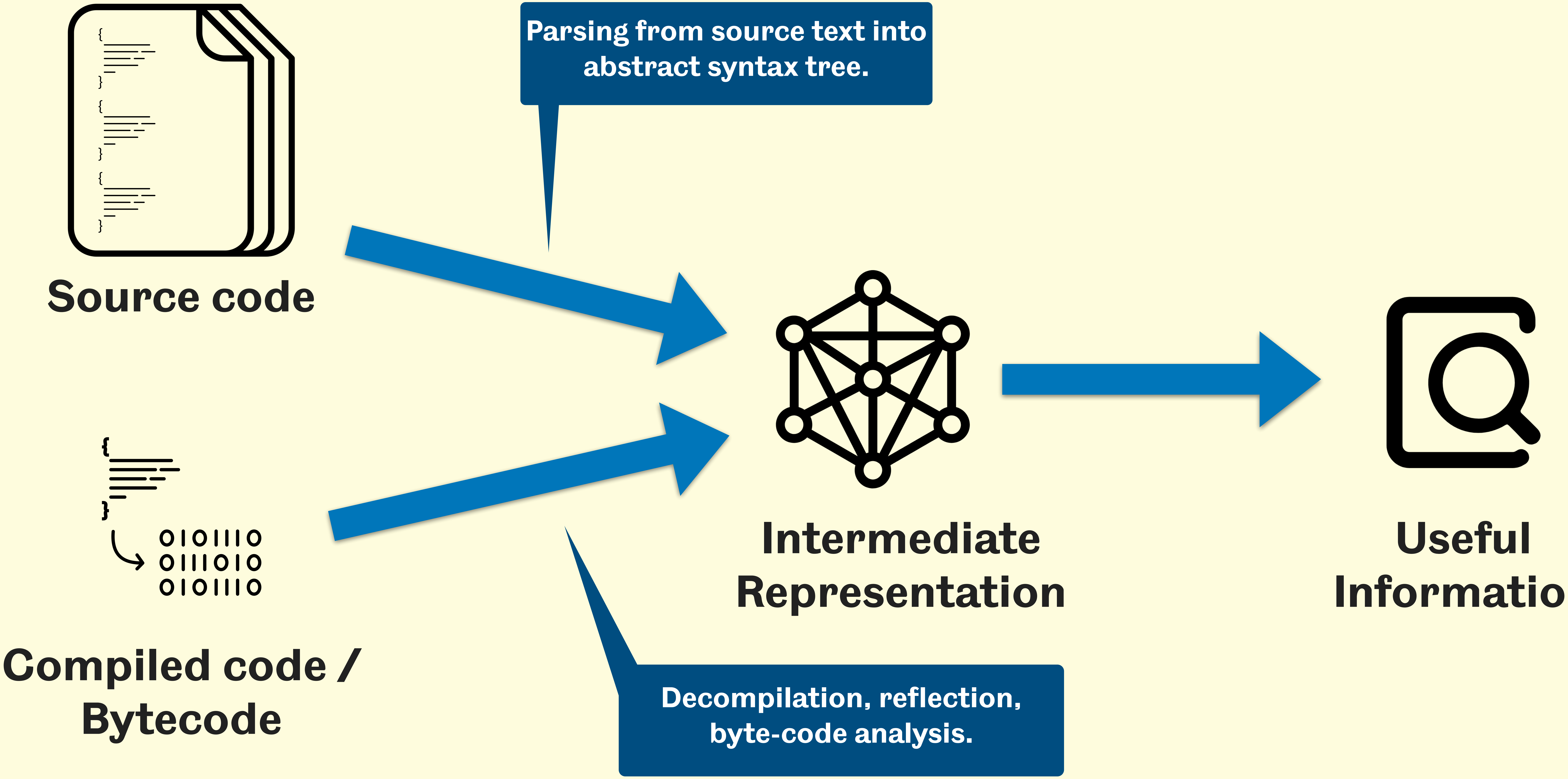
Source code

Compiled code / Bytecode

Intermediate Representation

Useful Information

The source code analysis process



Parsing from source text into abstract syntax tree.

Source code

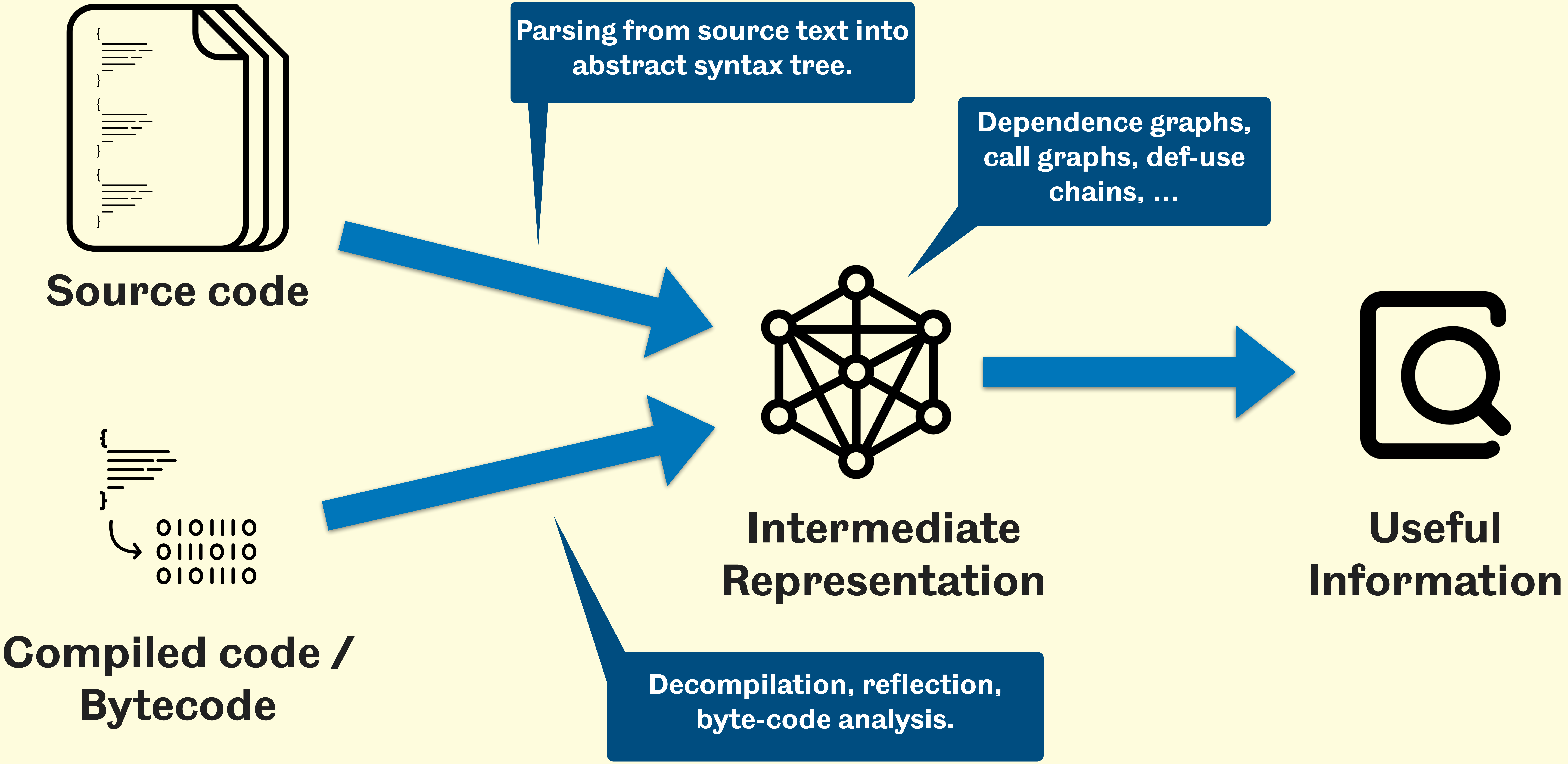
Compiled code / Bytecode

Intermediate Representation

Useful Information

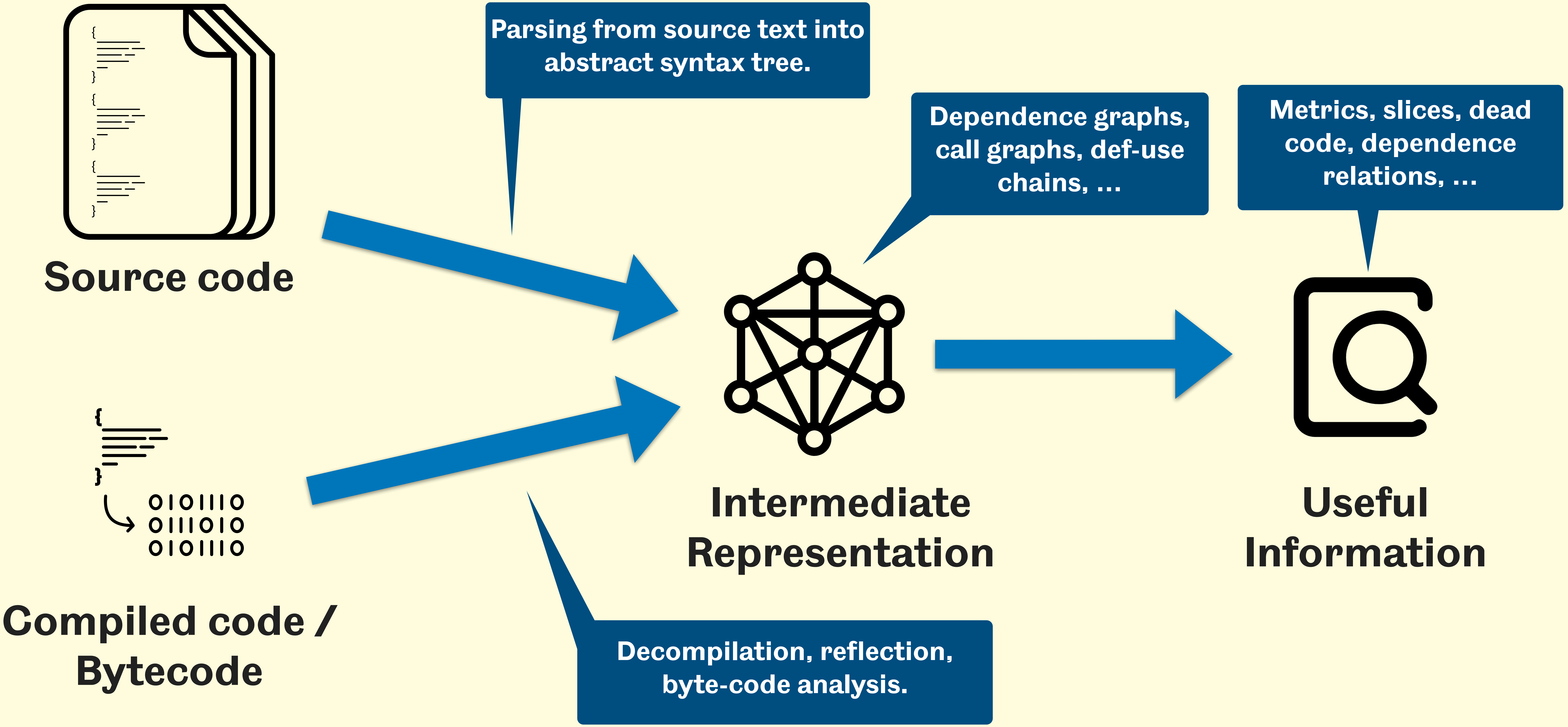
Decompilation, reflection, byte-code analysis.

The source code analysis process



Parsing from source text into abstract syntax tree.
Dependence graphs, call graphs, def-use chains, …
Source code
Intermediate Representation
Useful Information
Compiled code / Bytecode
Decompilation, reflection, byte-code analysis.
0101110
0111010
0101110

The source code analysis process



Source code

Compiled code / Bytecode

Parsing from source text into abstract syntax tree.

Decompilation, reflection, byte-code analysis.

Intermediate Representation

Dependence graphs, call graphs, def-use chains, ...

Useful Information

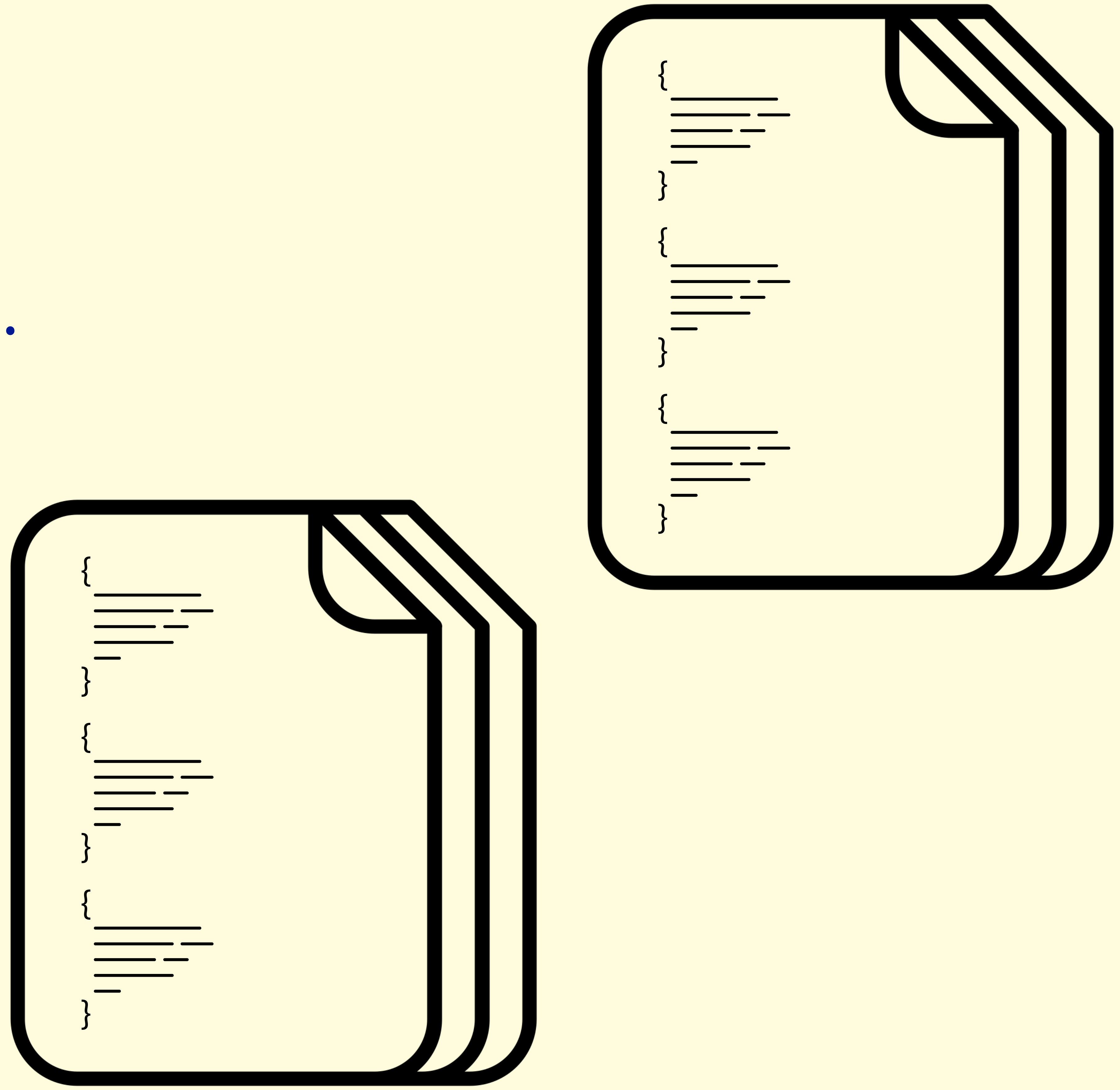
Metrics, slices, dead code, dependence relations, ...

Two levels of analysis

Function-level analysis

Examine control and data-flow within a function.

Useful for identifying information-flow vulnerabilities.

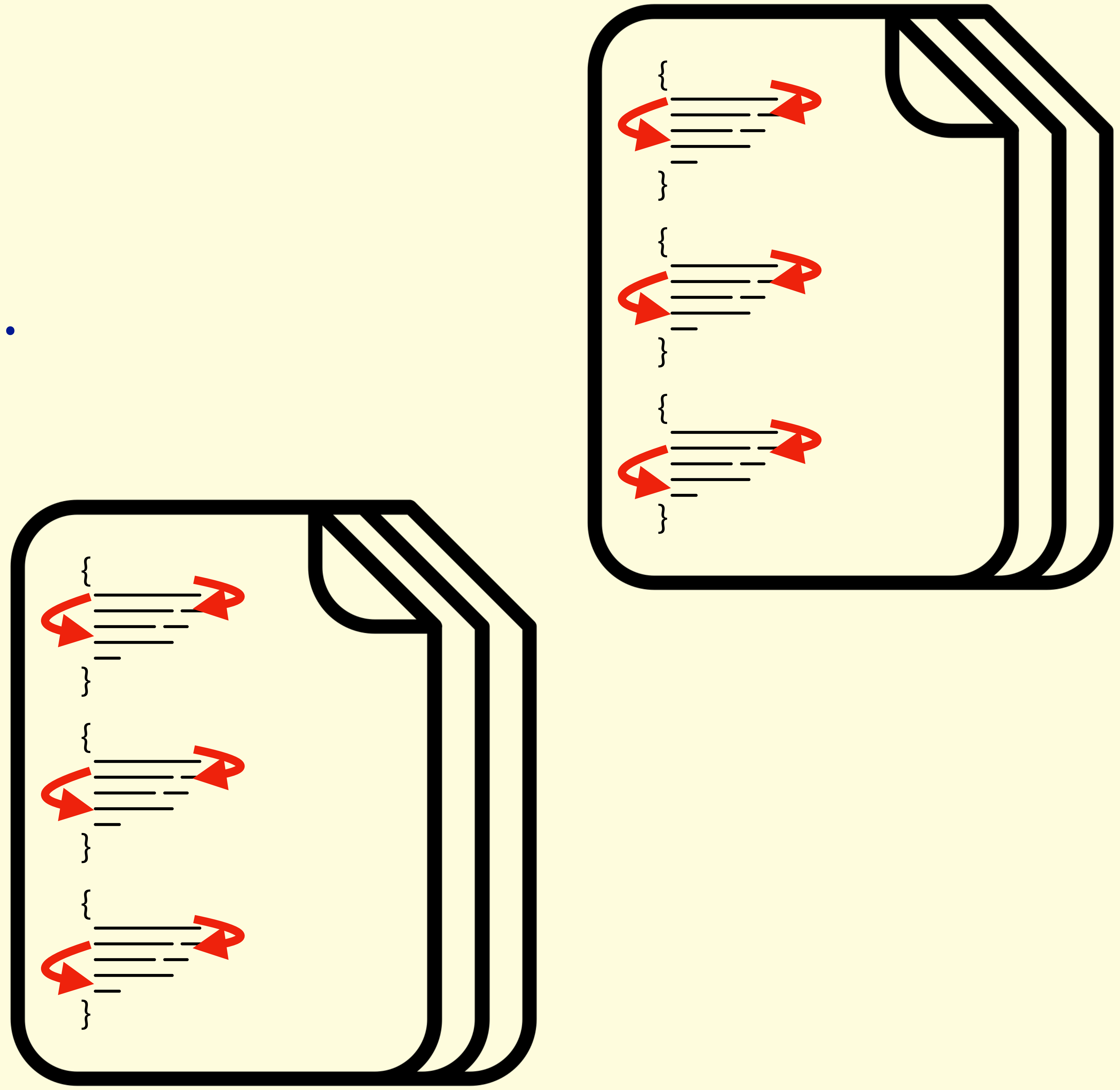


Two levels of analysis

Function-level analysis

Examine control and data-flow within a function.

Useful for identifying information-flow vulnerabilities.



Two levels of analysis

Function-level analysis

Examine control and data-flow within a function.

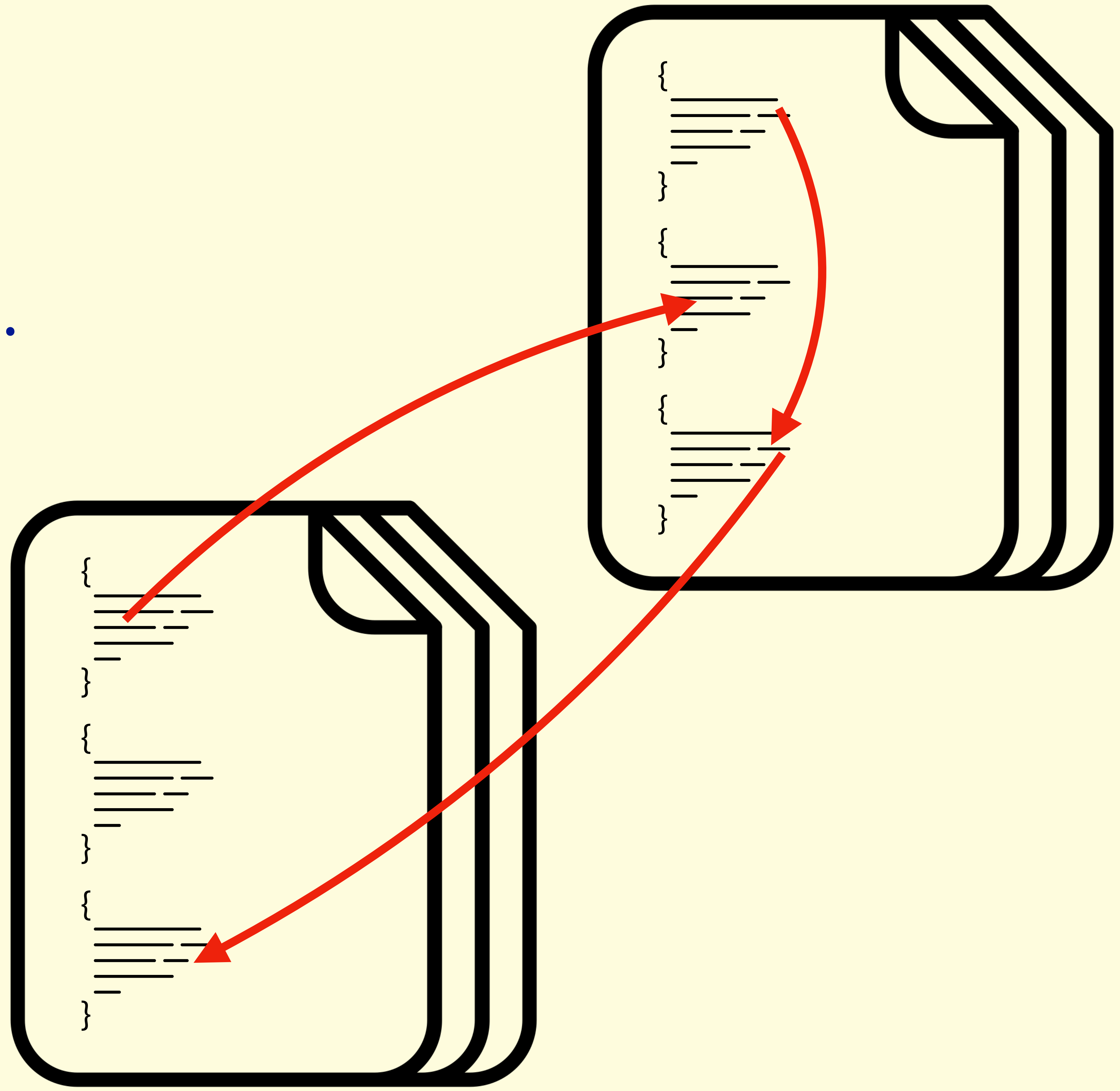
Useful for identifying information-flow vulnerabilities.

System-level analysis

Which procedures call each other?

How are classes connected to each other?

More useful for obtaining a high-level overview.



Intra-procedural Analysis

Control Flow Graph (CFG)

Each statement is a node.

- Some statements may need to be split up, if they comprise multiple instructions.
- Do not count statements that are not executable code (comments, curly brackets, etc.)

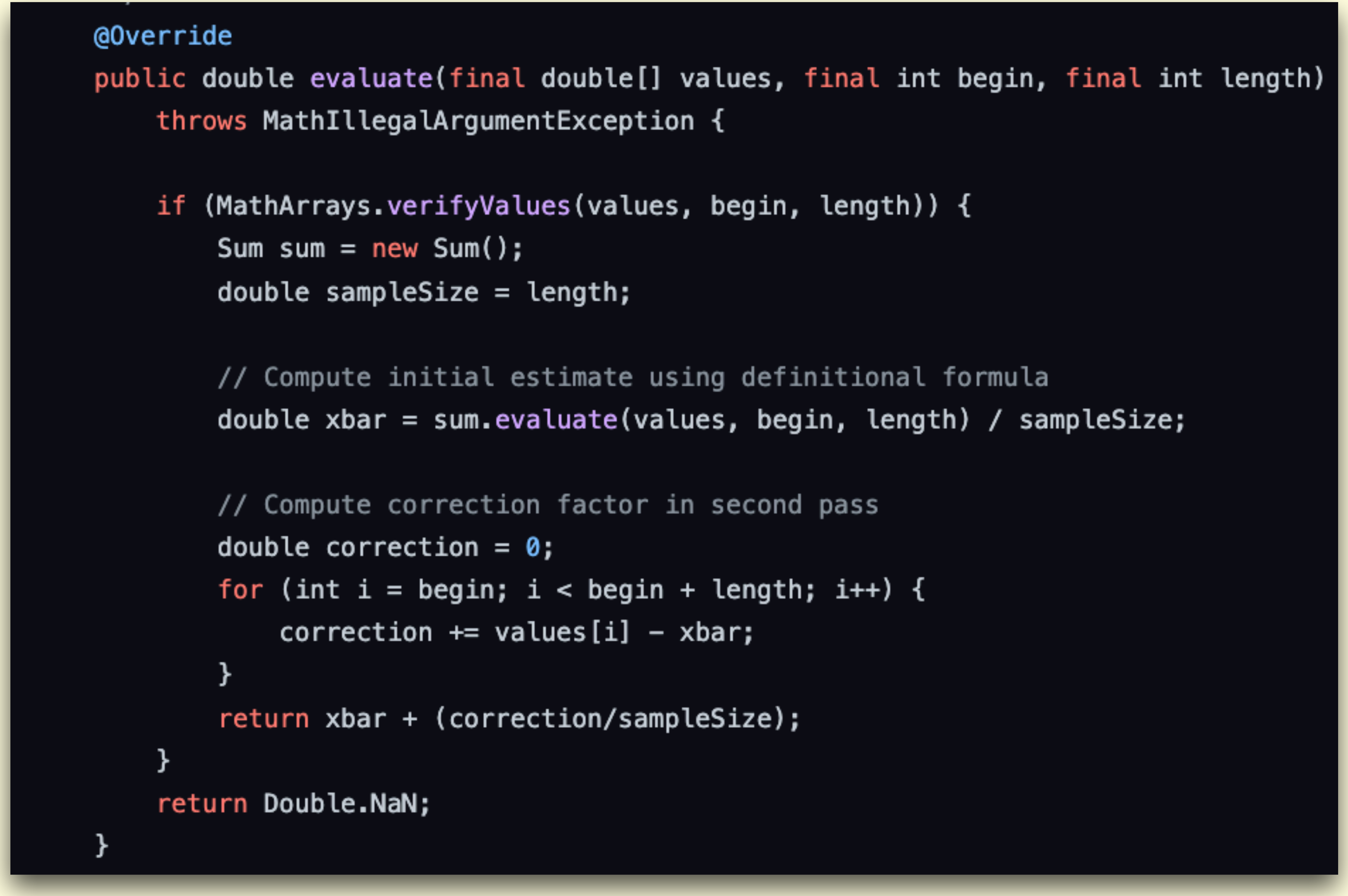
Flow from one statement to the next is represented as an edge.

Two special additional nodes that do not correspond to real statements:

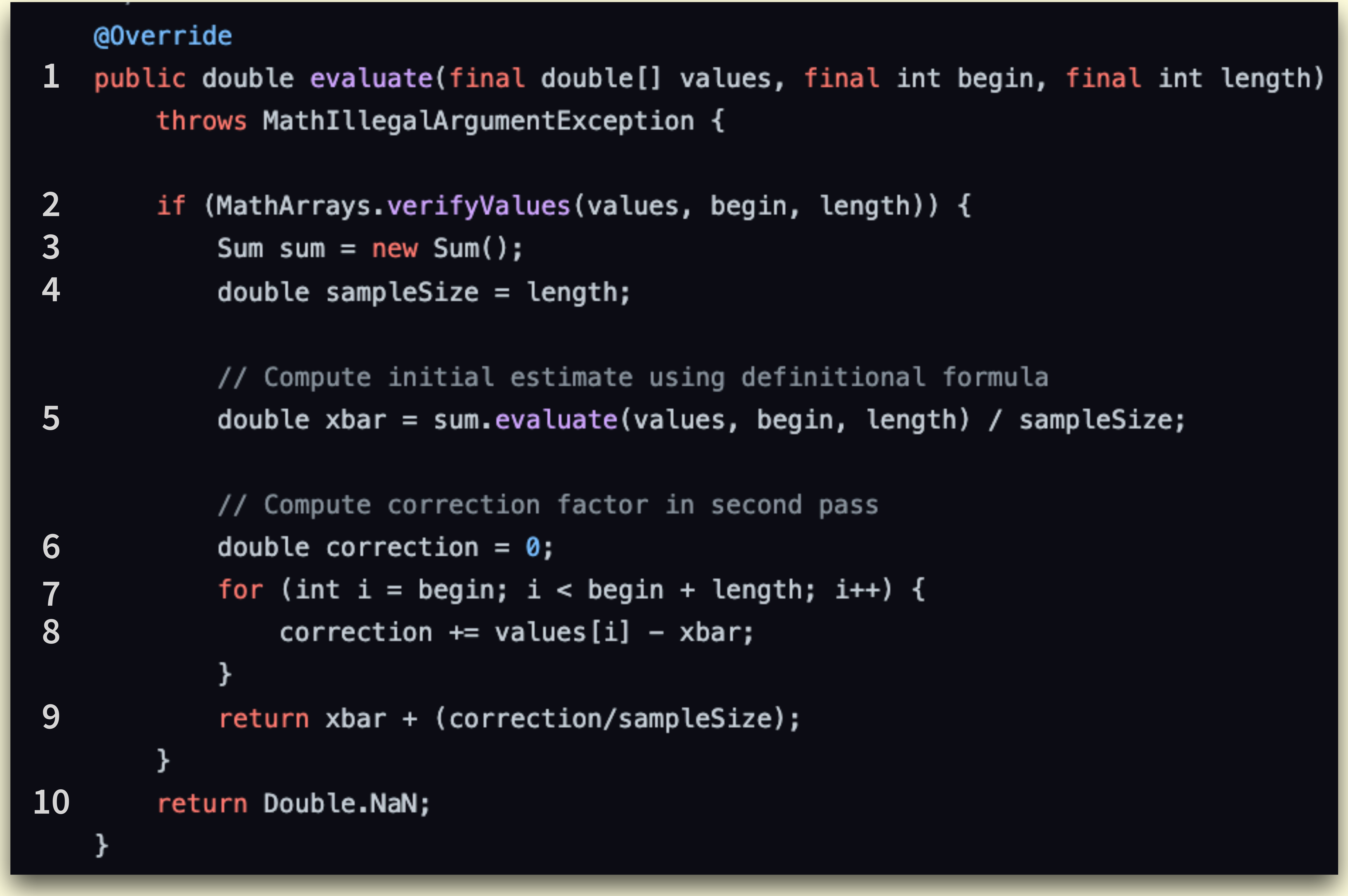
- “Entry” and “Exit” nodes.
- Represent points at which control “arrives at” and “departs from” function.

Conditional statements are branching nodes.

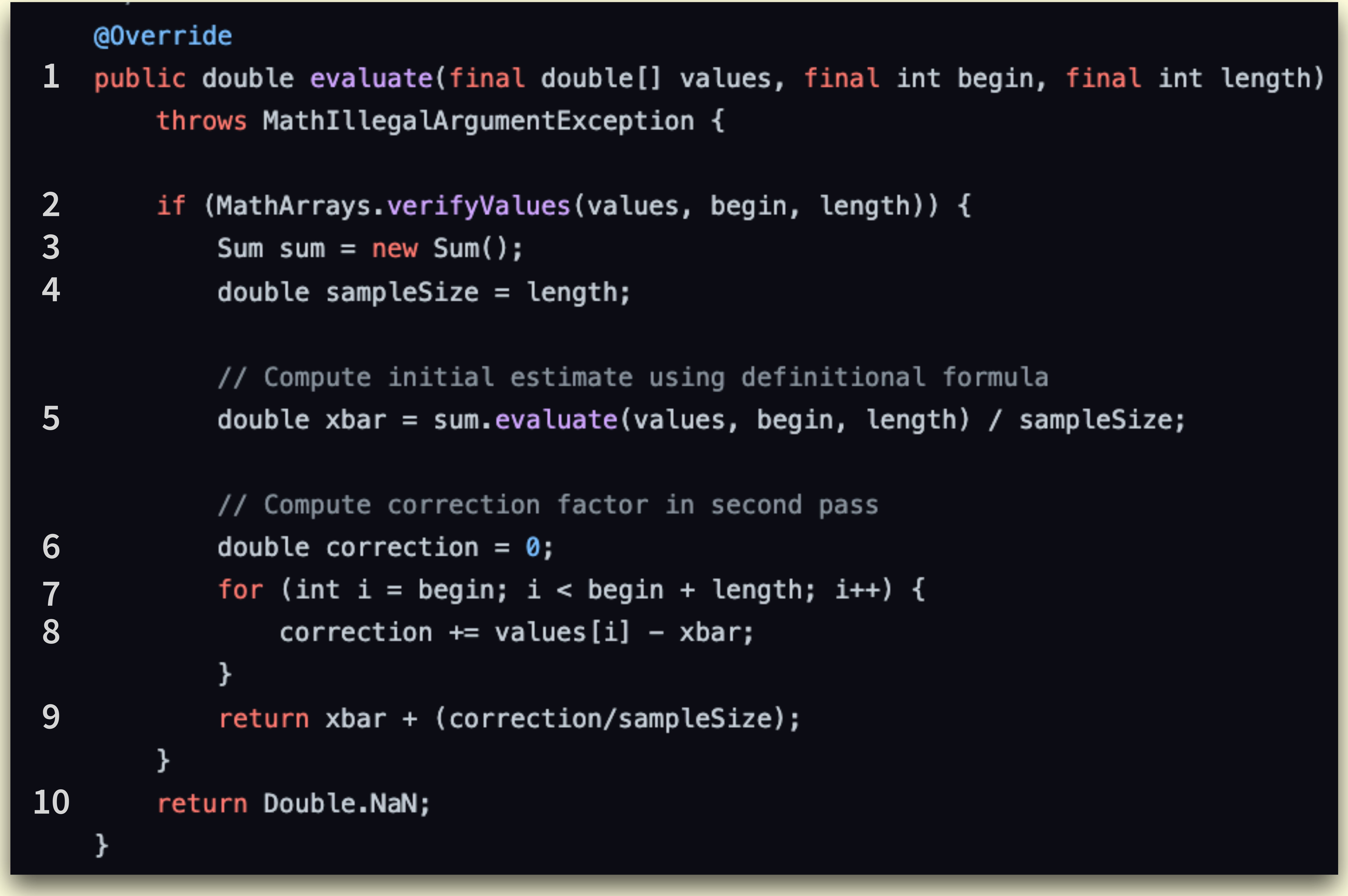
- If / while / for …



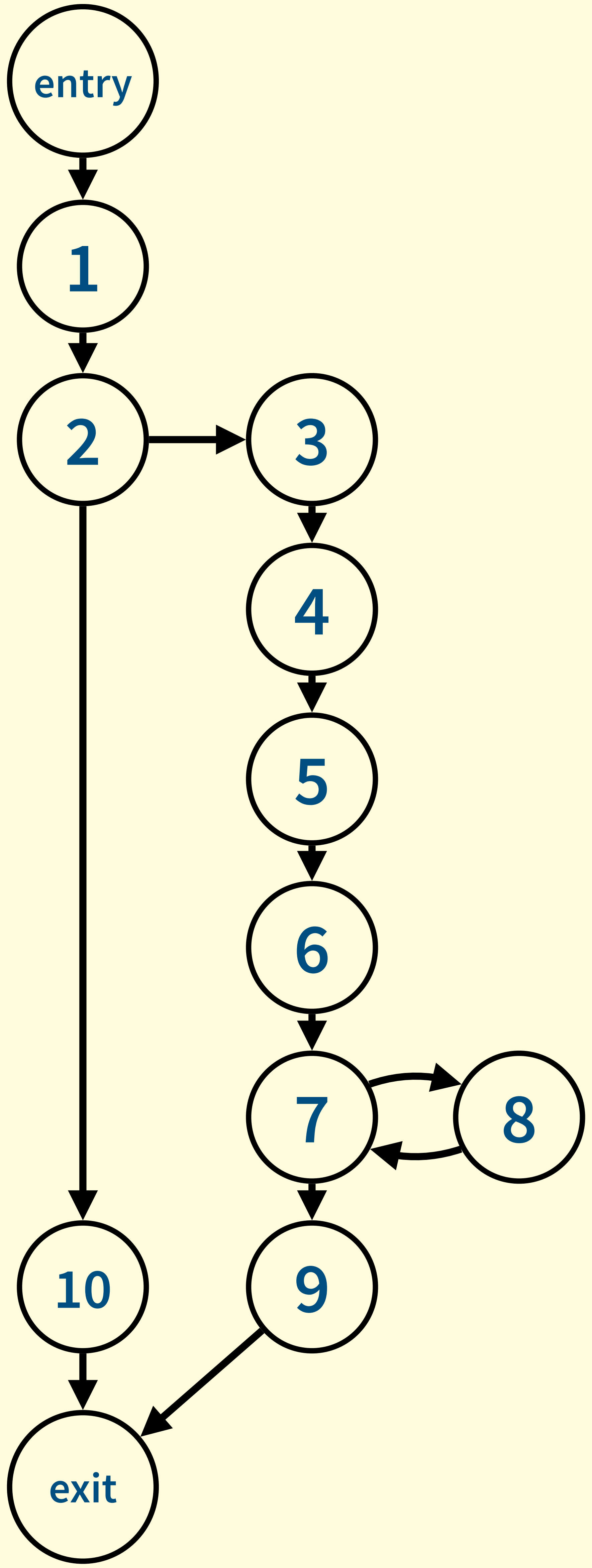
```java
    @Override
    public double evaluate(final double[] values, final int begin, final int length)
        throws MathIllegalArgumentException {

        if (MathArrays.verifyValues(values, begin, length)) {
            Sum sum = new Sum();
            double sampleSize = length;

            // Compute initial estimate using definitional formula
            double xbar = sum.evaluate(values, begin, length) / sampleSize;

            // Compute correction factor in second pass
            double correction = 0;
            for (int i = begin; i < begin + length; i++) {
                correction += values[i] - xbar;
            }
            return xbar + (correction/sampleSize);
        }
        return Double.NaN;
    }
```



```
    @Override
1   public double evaluate(final double[] values, final int begin, final int length)
        throws MathIllegalArgumentException {

2       if (MathArrays.verifyValues(values, begin, length)) {
3           Sum sum = new Sum();
4           double sampleSize = length;

            // Compute initial estimate using definitional formula
5           double xbar = sum.evaluate(values, begin, length) / sampleSize;

            // Compute correction factor in second pass
6           double correction = 0;
7           for (int i = begin; i < begin + length; i++) {
8               correction += values[i] - xbar;
            }
9           return xbar + (correction/sampleSize);
        }
10      return Double.NaN;
    }
```



```
    @Override
1   public double evaluate(final double[] values, final int begin, final int length)
        throws MathIllegalArgumentException {

2       if (MathArrays.verifyValues(values, begin, length)) {
3           Sum sum = new Sum();
4           double sampleSize = length;

            // Compute initial estimate using definitional formula
5           double xbar = sum.evaluate(values, begin, length) / sampleSize;

            // Compute correction factor in second pass
6           double correction = 0;
7           for (int i = begin; i < begin + length; i++) {
8               correction += values[i] - xbar;
            }
9           return xbar + (correction/sampleSize);
        }
10      return Double.NaN;
    }
```



Post-dominance and Control Dependence

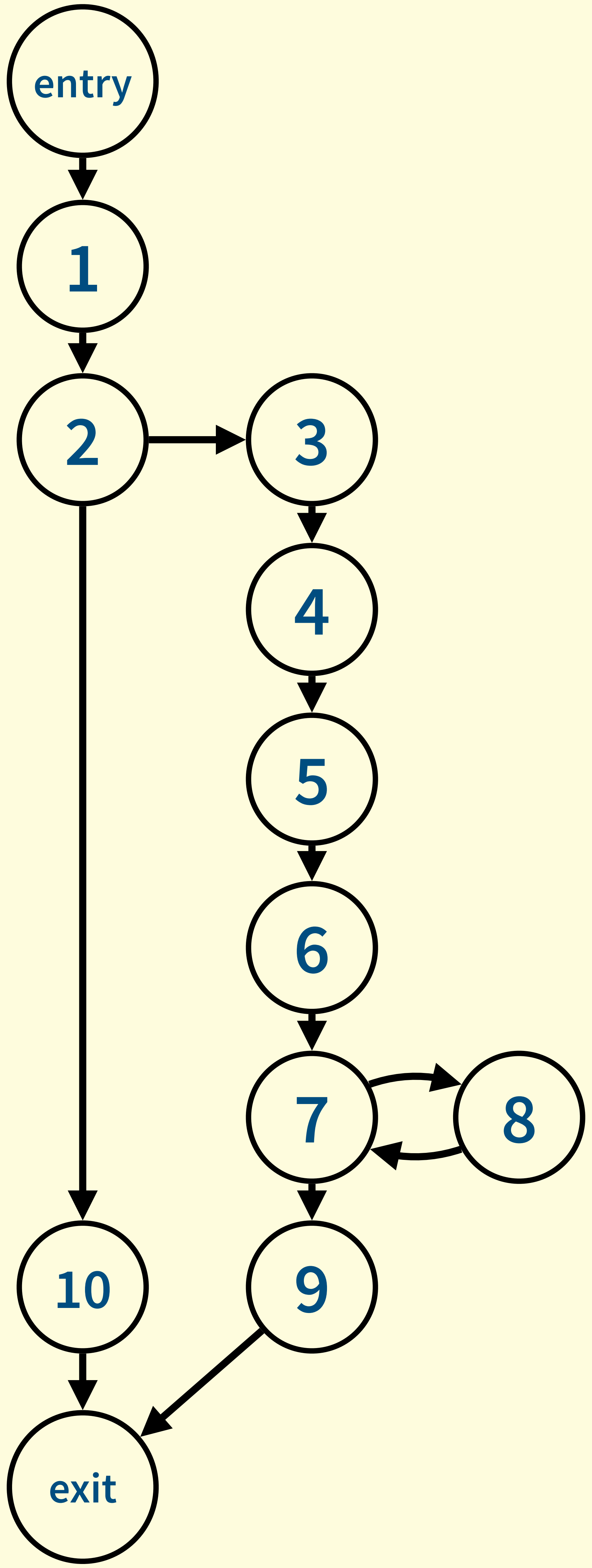
Statement B **post-dominates** A if every path from A to the exit passes through B.

- The first node on the path to the exit that post-dominates another node is known as the **immediate post-dominator**.

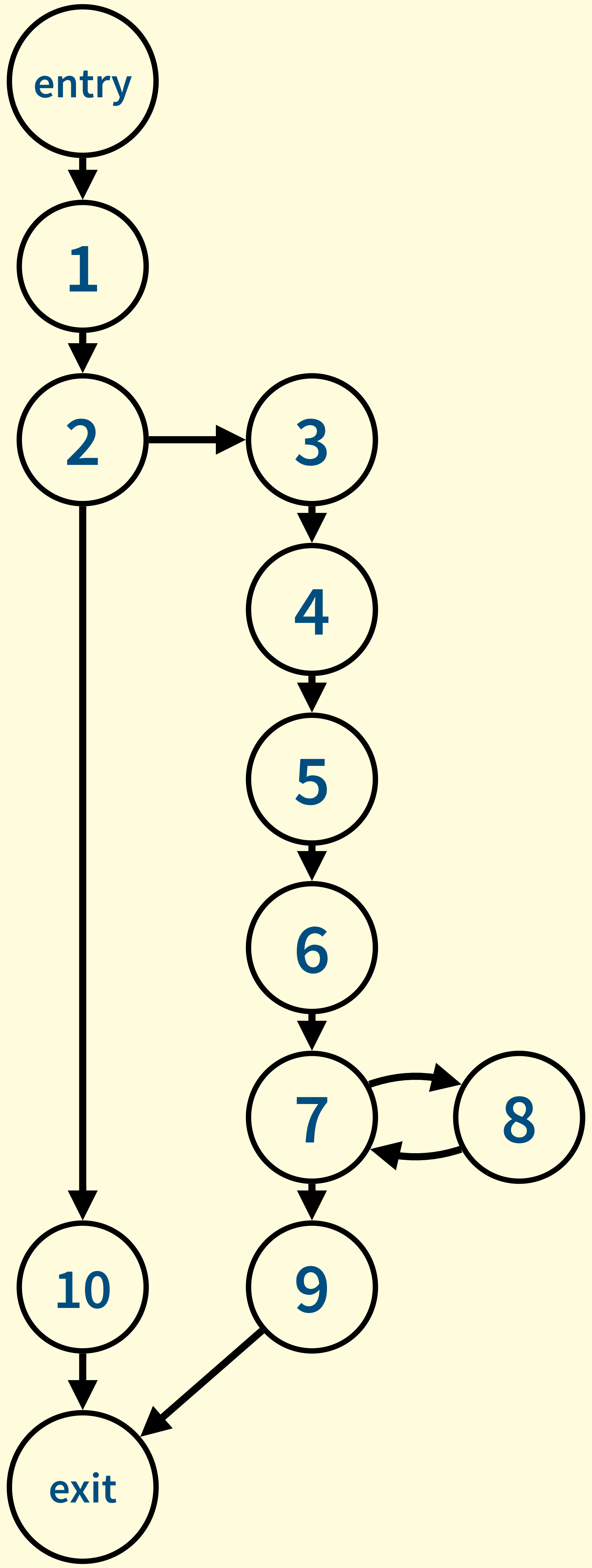
There is a **control dependence** relation from A to B if A directly determines whether B will be executed:

- There exists a path from A to B where all nodes on the path are post-dominated by B.
- A is not post-dominated by B.

Control dependence relations can be visualised as a tree.

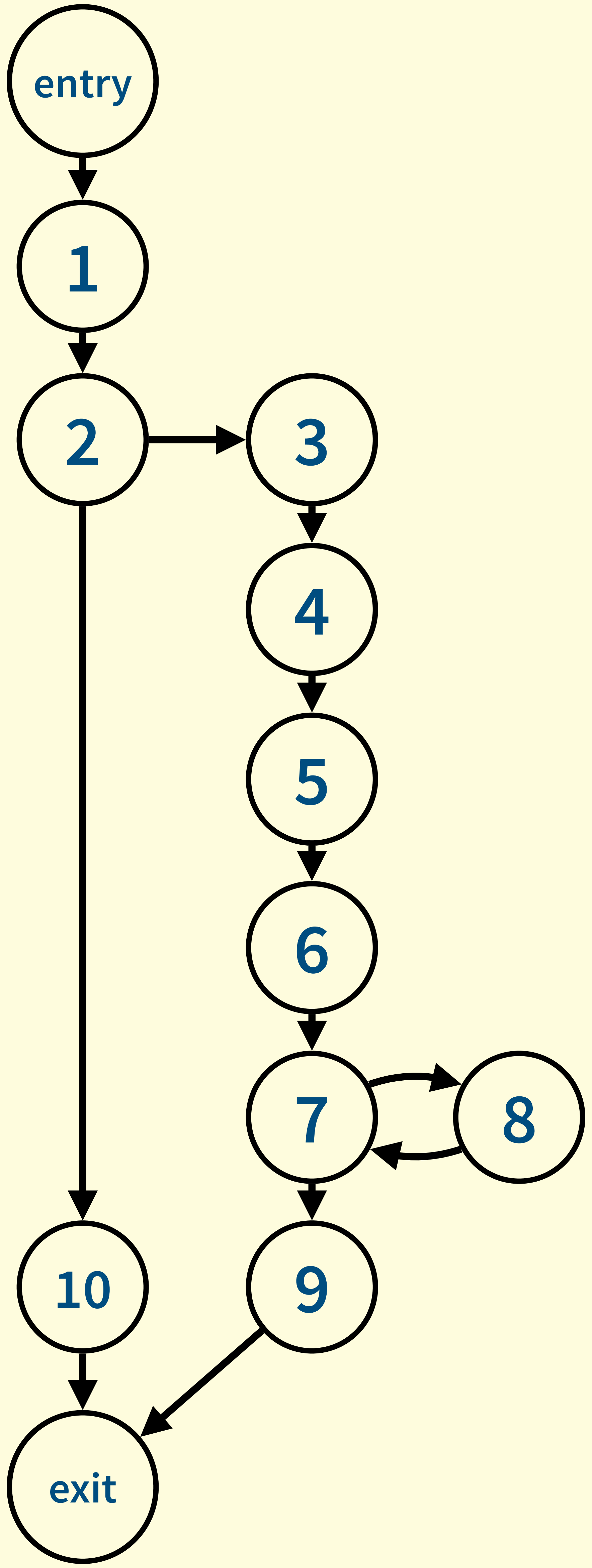


Does node 7 post-dominate node 3?



Does node 7 post-dominate node 3?

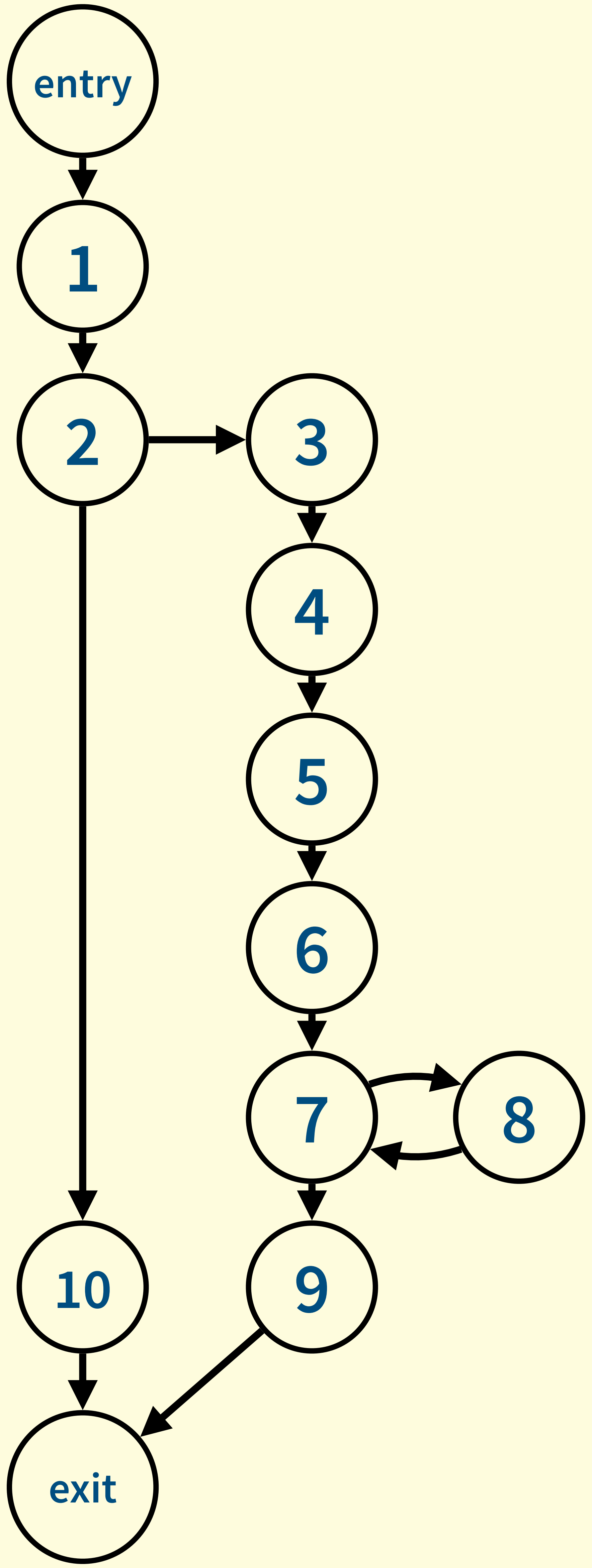
Does node 8 post-dominate node 7?



Does node 7 post-dominate node 3?

Does node 8 post-dominate node 7?

What is the immediate post-dominator for node 7?

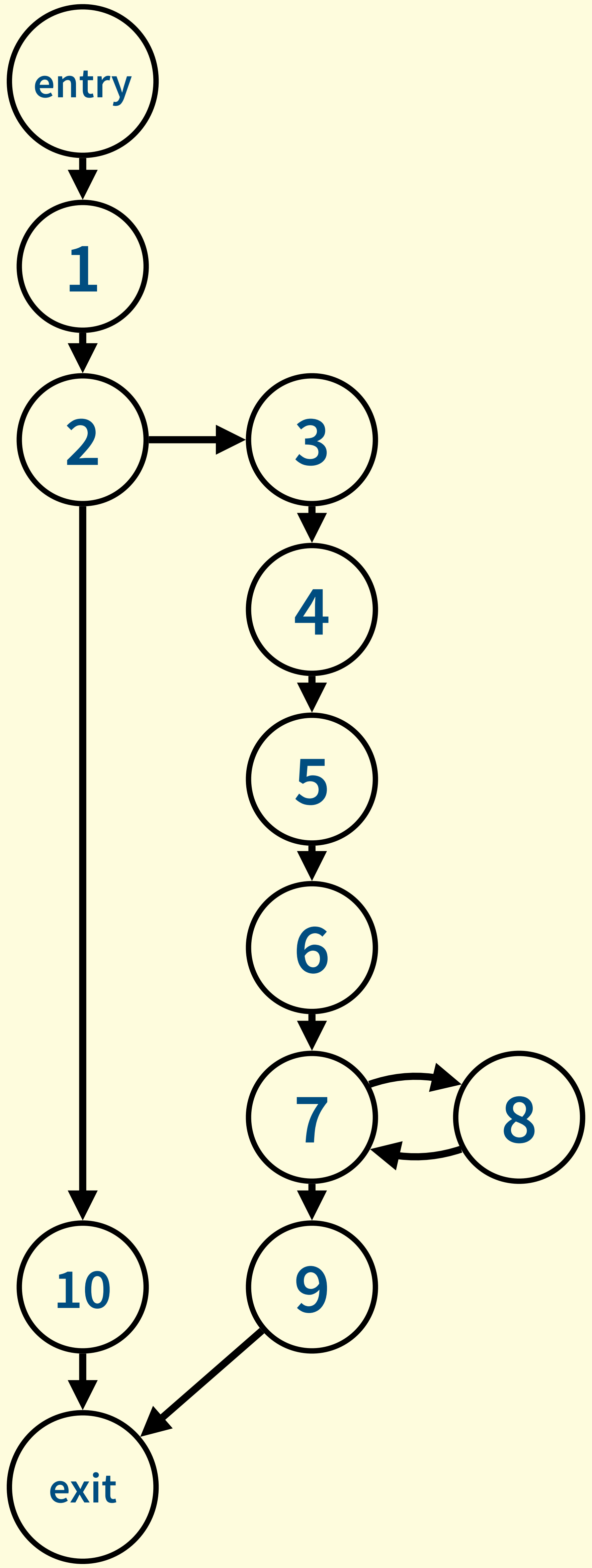


Does node 7 post-dominate node 3?

Does node 8 post-dominate node 7?

What is the immediate post-dominator for node 7?

Draw out the Control Dependence tree for this CFG.

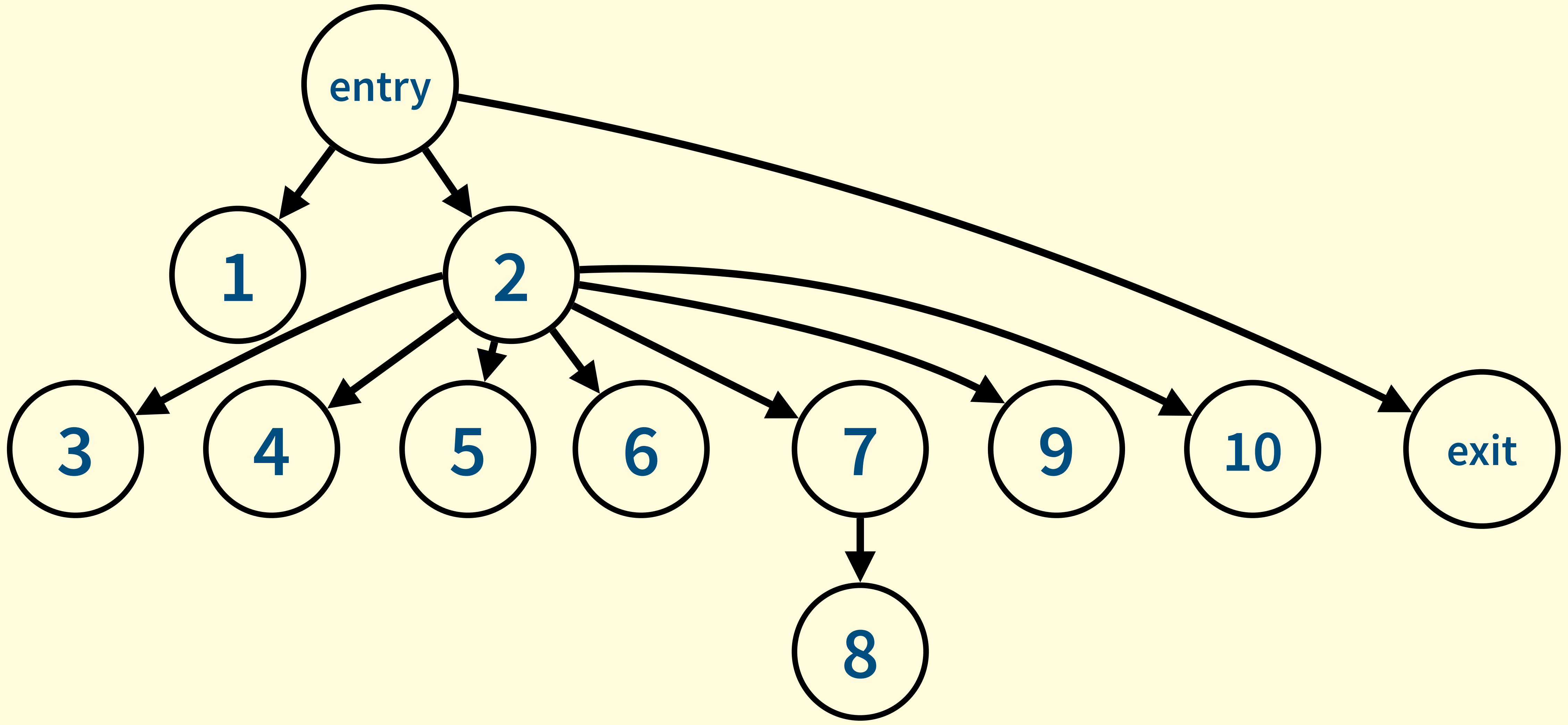


Does node 7 post-dominate node 3?

Does node 8 post-dominate node 7?

What is the immediate post-dominator for node 7?

Draw out the Control Dependence tree for this CFG.



Data Flow

The flow of data through a function can be characterised in terms of “defs” and “uses”.

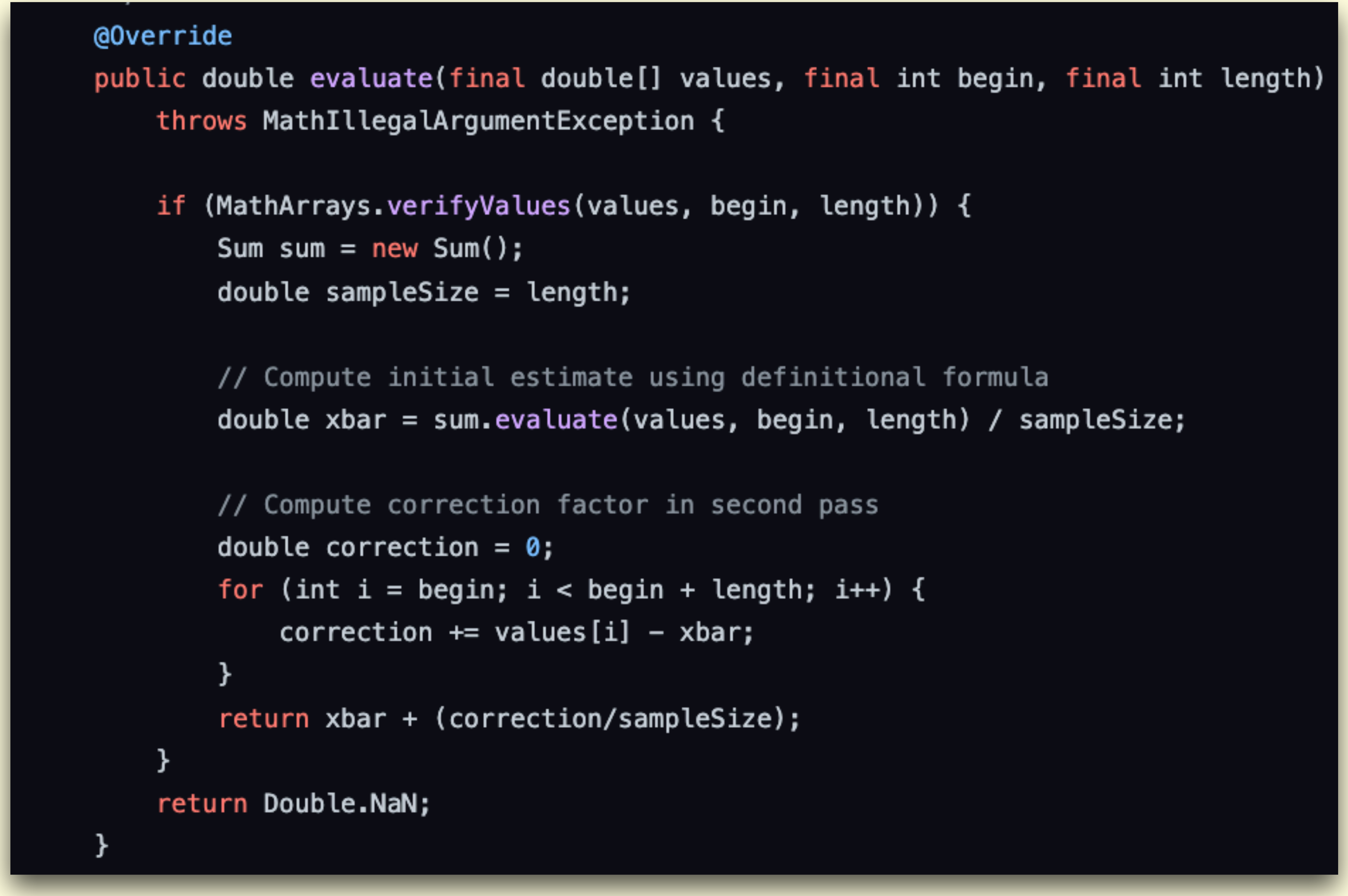
- A statement may “define” a variable by assigning a value to it.
- A statement may “use” a variable as a part of some computation.

A “def-use” relationship exists between statements A and B for variable v when:

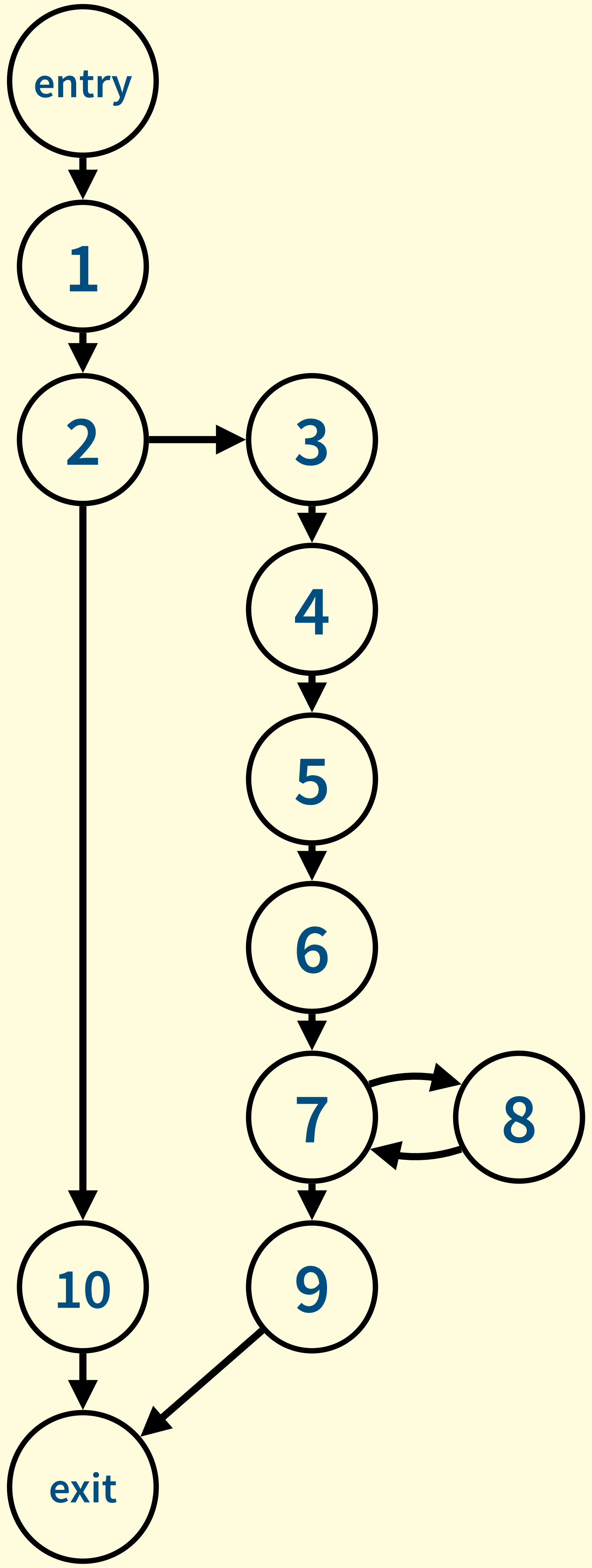
- A “defines” v by assigning a value to it.
- B “uses” v as a part of some computation.
- There is a path from A to B where v is not re-defined.

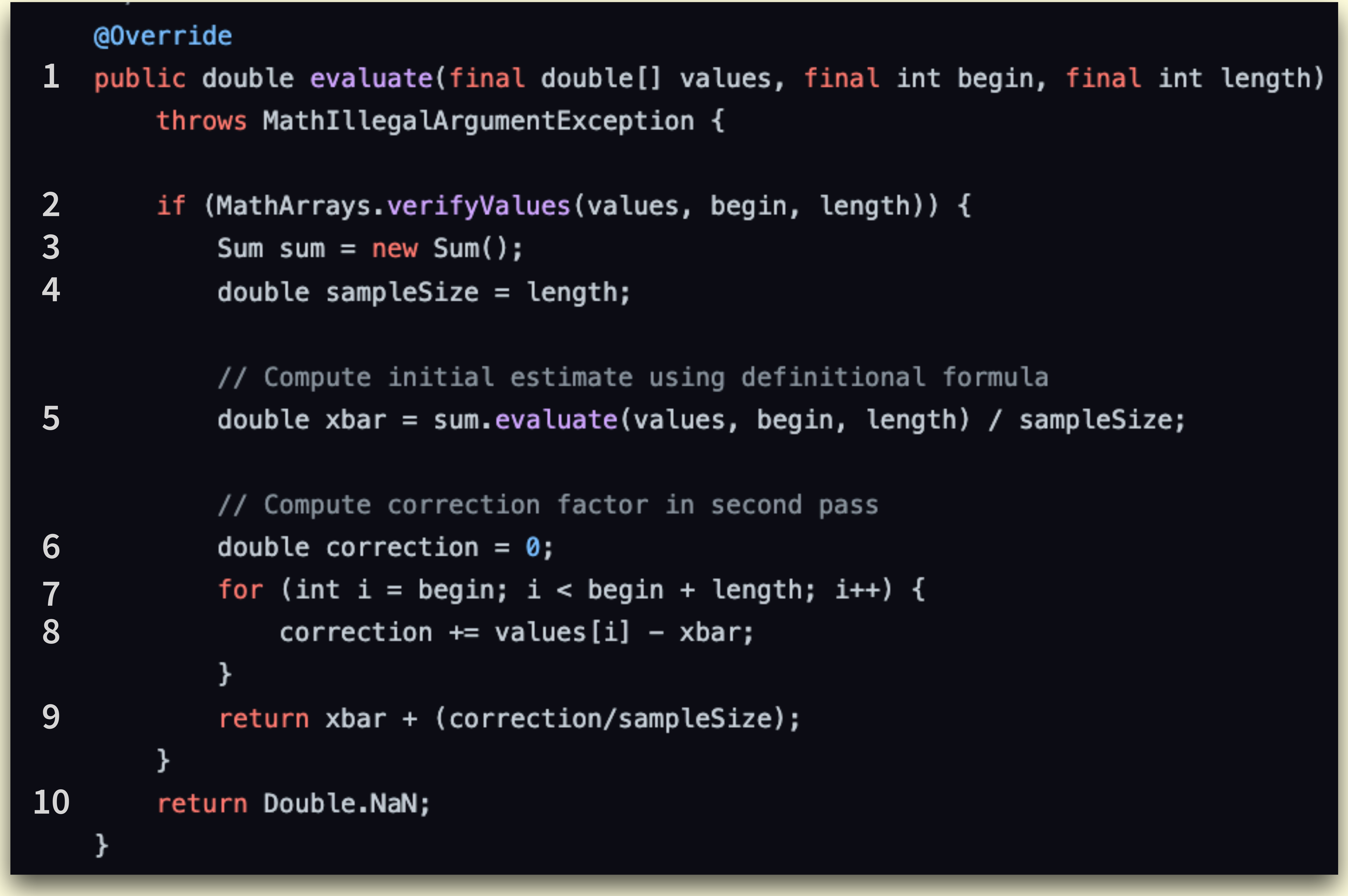
Not always obvious in Object-Oriented languages if an object is being “defined”.

- If statement calls a method on an object variable, does this method change its state?

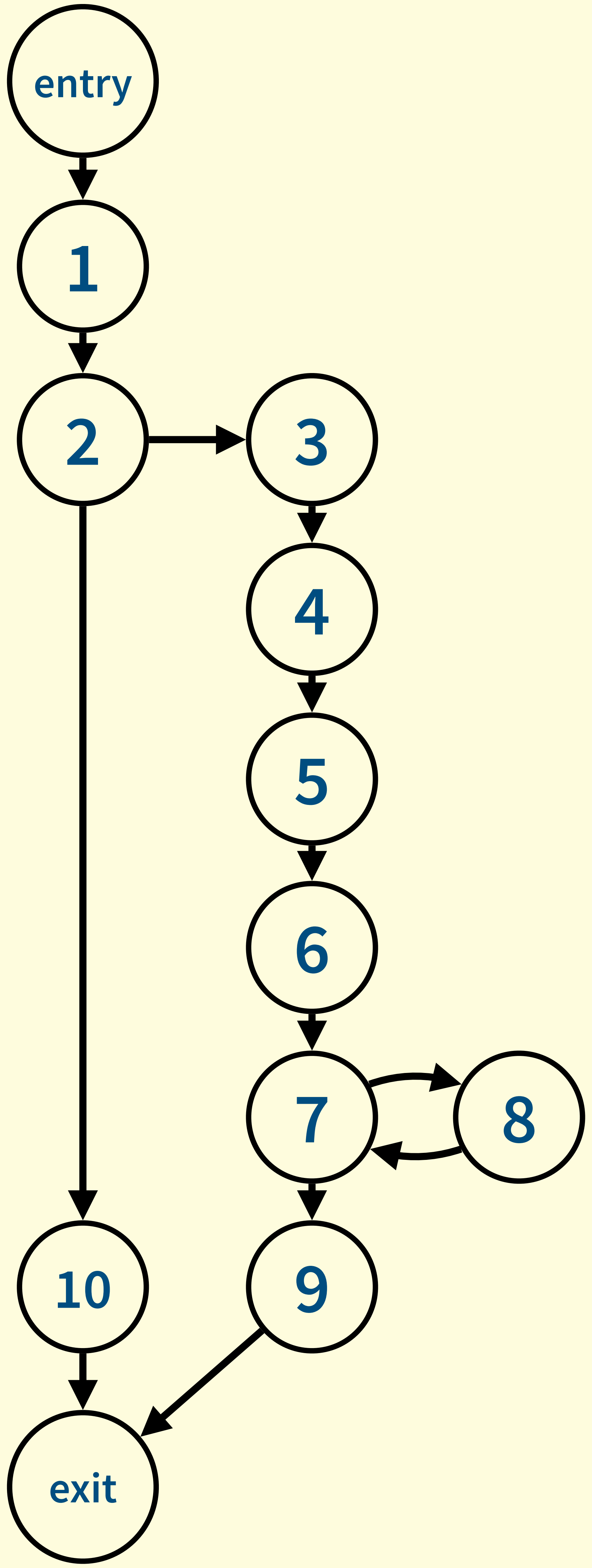


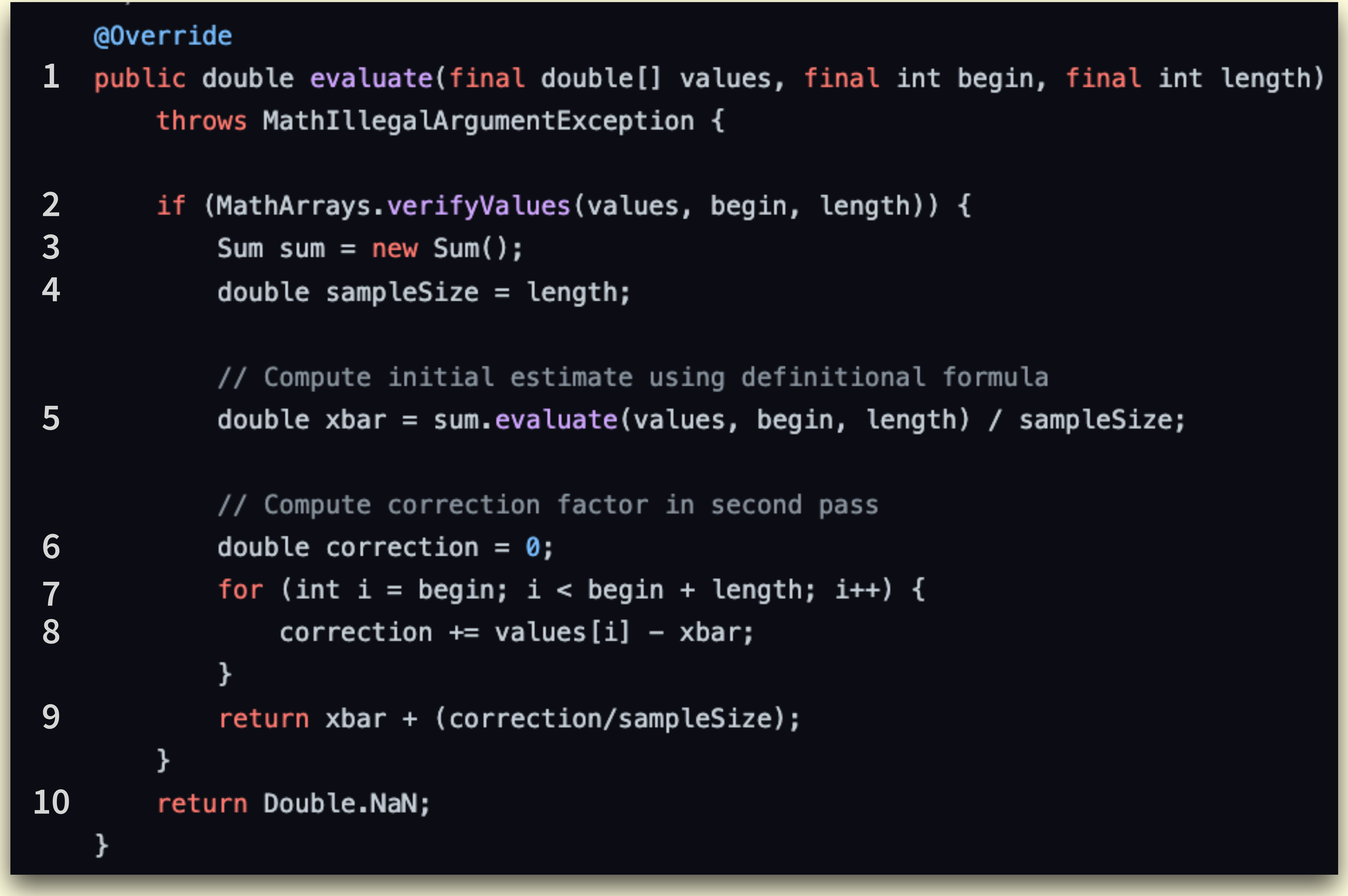
```java
@Override
public double evaluate(final double[] values, final int begin, final int length)
    throws MathIllegalArgumentException {

    if (MathArrays.verifyValues(values, begin, length)) {
        Sum sum = new Sum();
        double sampleSize = length;

        // Compute initial estimate using definitional formula
        double xbar = sum.evaluate(values, begin, length) / sampleSize;

        // Compute correction factor in second pass
        double correction = 0;
        for (int i = begin; i < begin + length; i++) {
            correction += values[i] - xbar;
        }
        return xbar + (correction/sampleSize);
    }
    return Double.NaN;
}
```



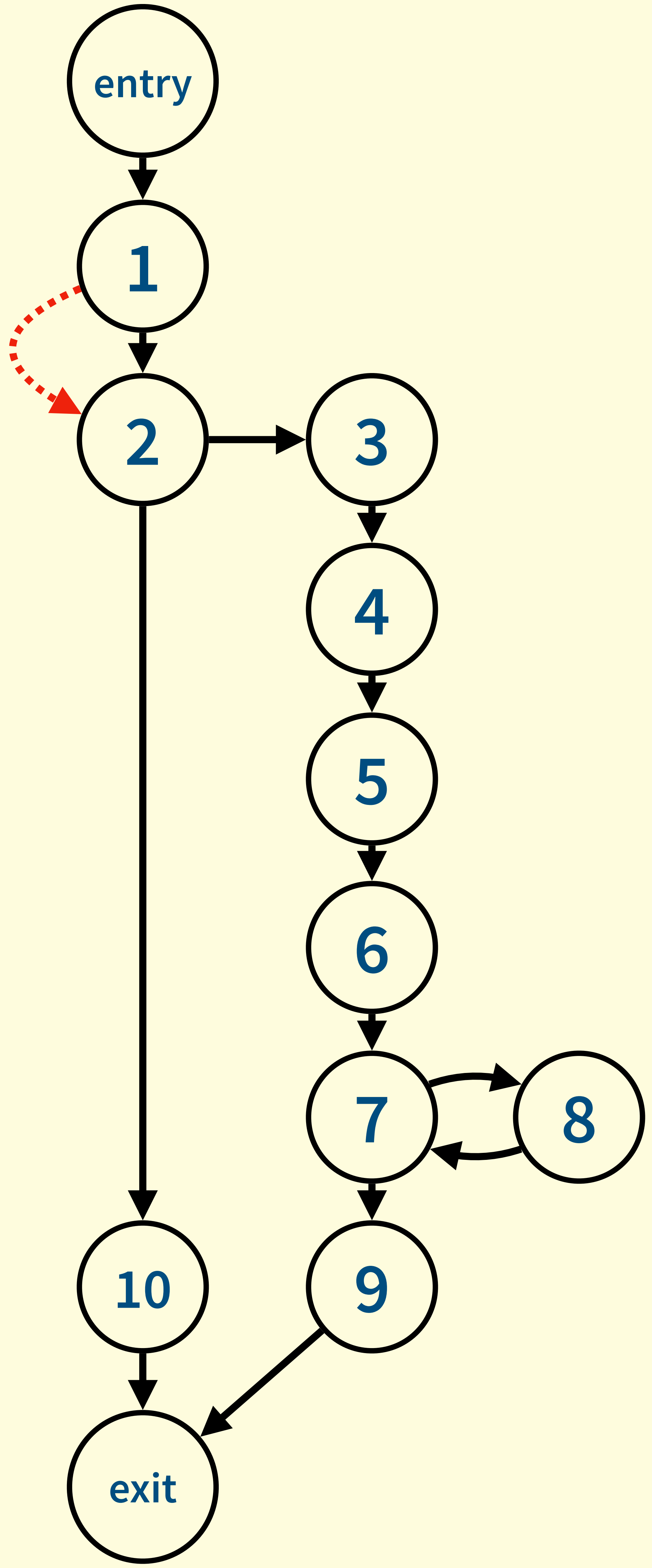


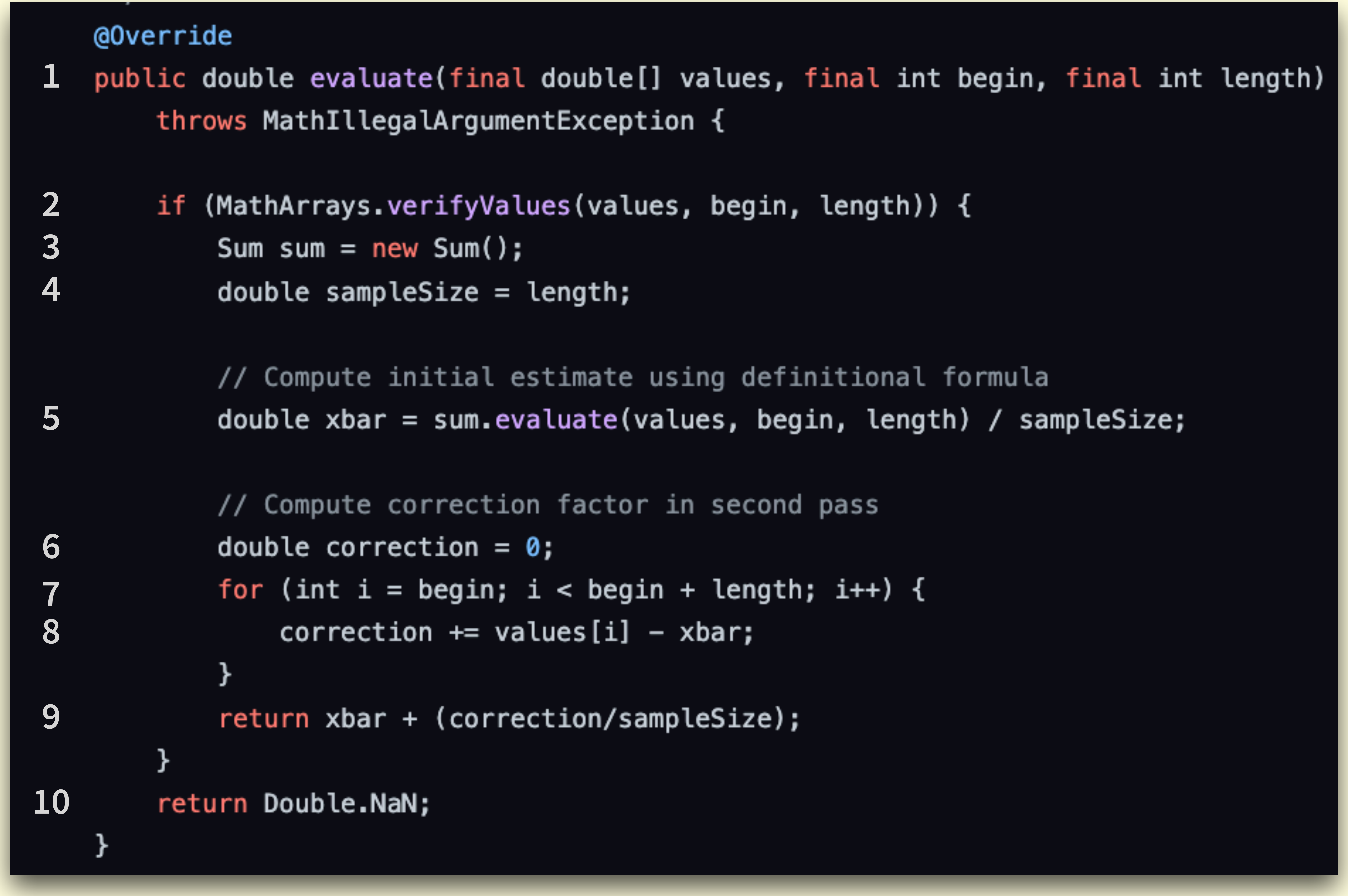
```
    @Override
1   public double evaluate(final double[] values, final int begin, final int length)
        throws MathIllegalArgumentException {

2       if (MathArrays.verifyValues(values, begin, length)) {
3           Sum sum = new Sum();
4           double sampleSize = length;

            // Compute initial estimate using definitional formula
5           double xbar = sum.evaluate(values, begin, length) / sampleSize;

            // Compute correction factor in second pass
6           double correction = 0;
7           for (int i = begin; i < begin + length; i++) {
8               correction += values[i] - xbar;
            }
9           return xbar + (correction/sampleSize);
        }
10      return Double.NaN;
    }
```



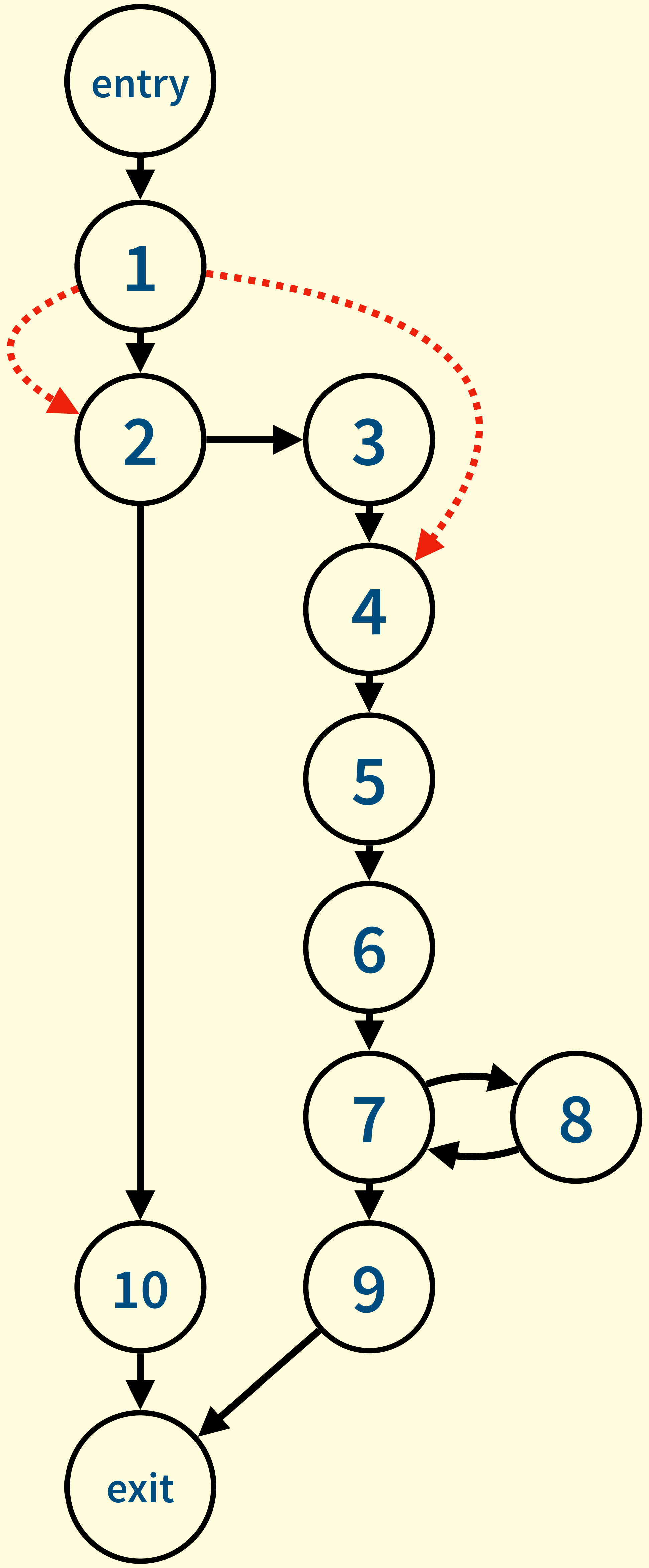


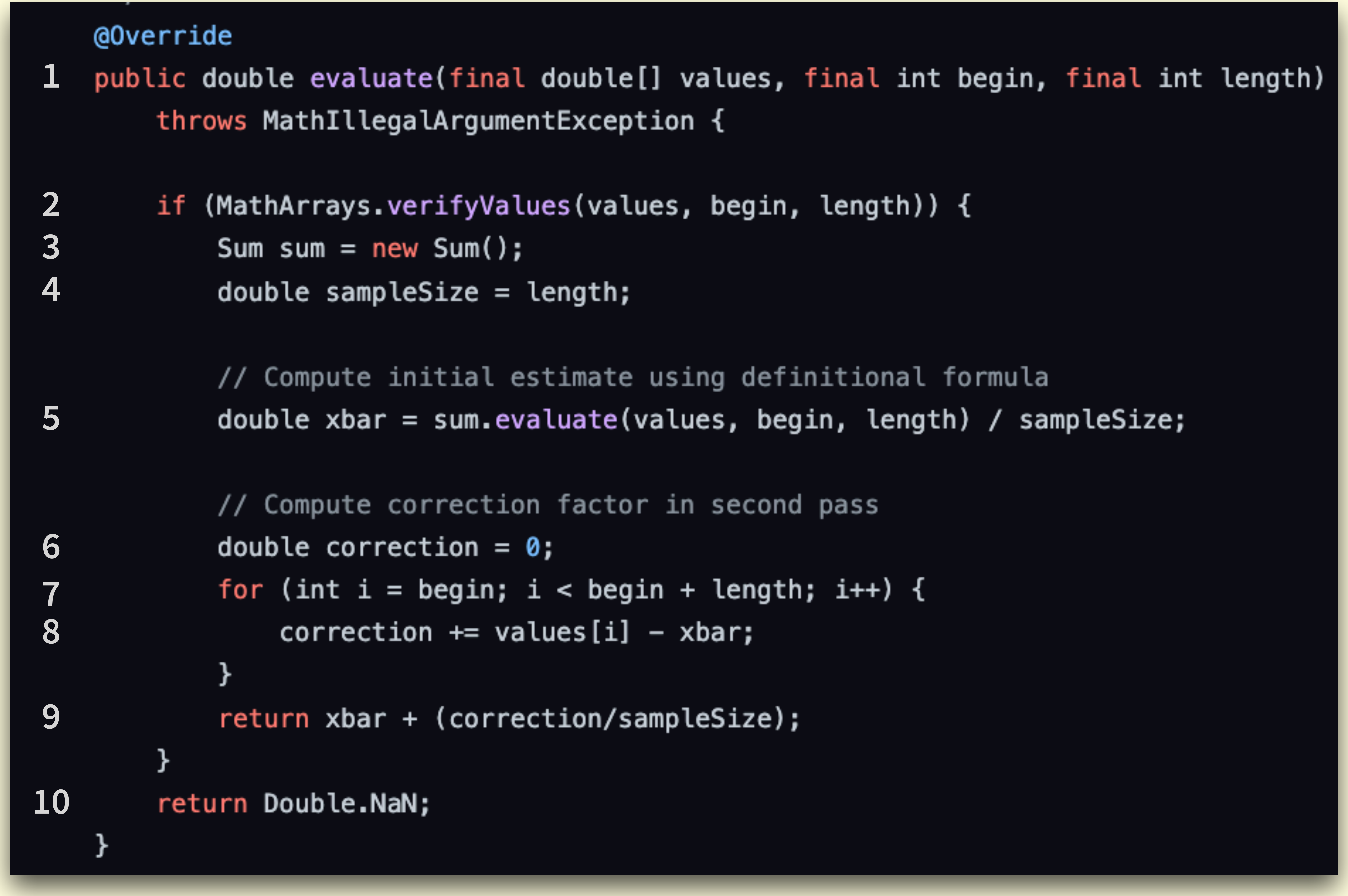
```
    @Override
1   public double evaluate(final double[] values, final int begin, final int length)
        throws MathIllegalArgumentException {

2       if (MathArrays.verifyValues(values, begin, length)) {
3           Sum sum = new Sum();
4           double sampleSize = length;

            // Compute initial estimate using definitional formula
5           double xbar = sum.evaluate(values, begin, length) / sampleSize;

            // Compute correction factor in second pass
6           double correction = 0;
7           for (int i = begin; i < begin + length; i++) {
8               correction += values[i] - xbar;
            }
9           return xbar + (correction/sampleSize);
        }
10      return Double.NaN;
    }
```



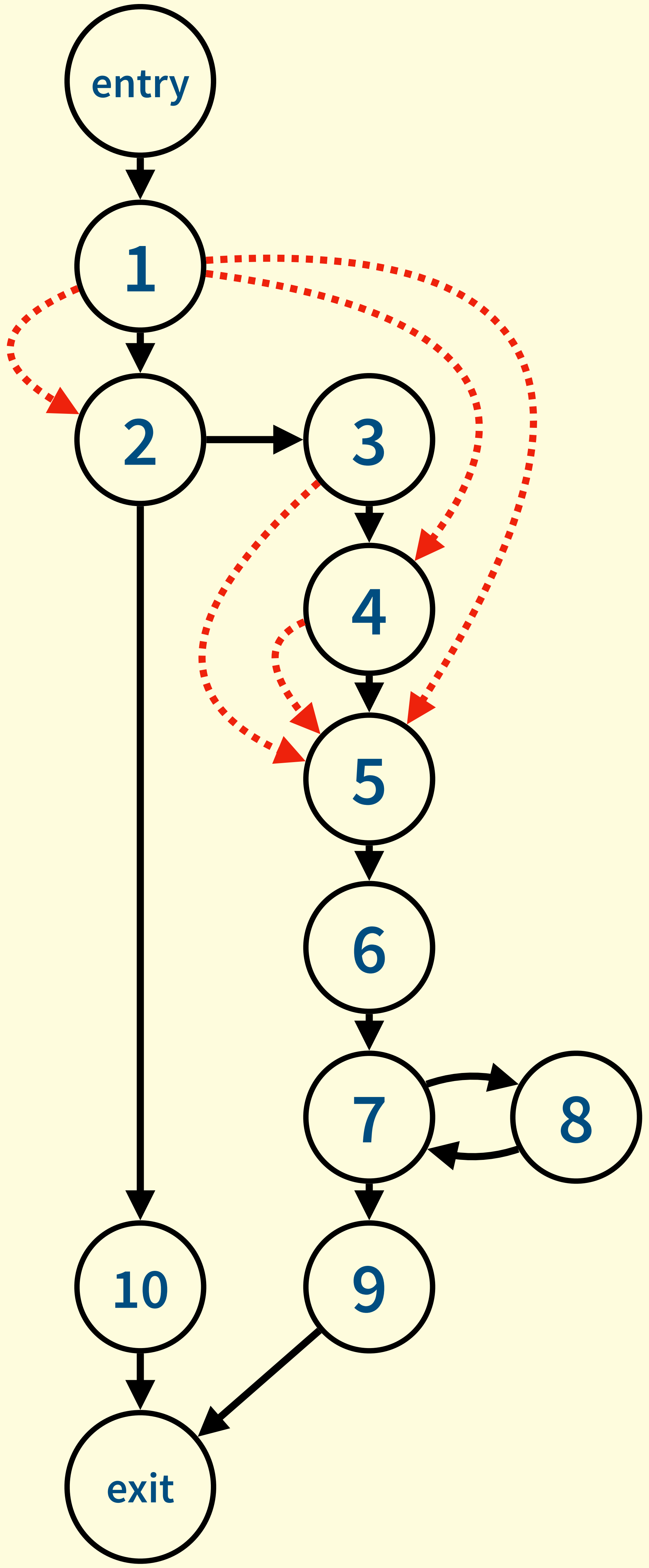


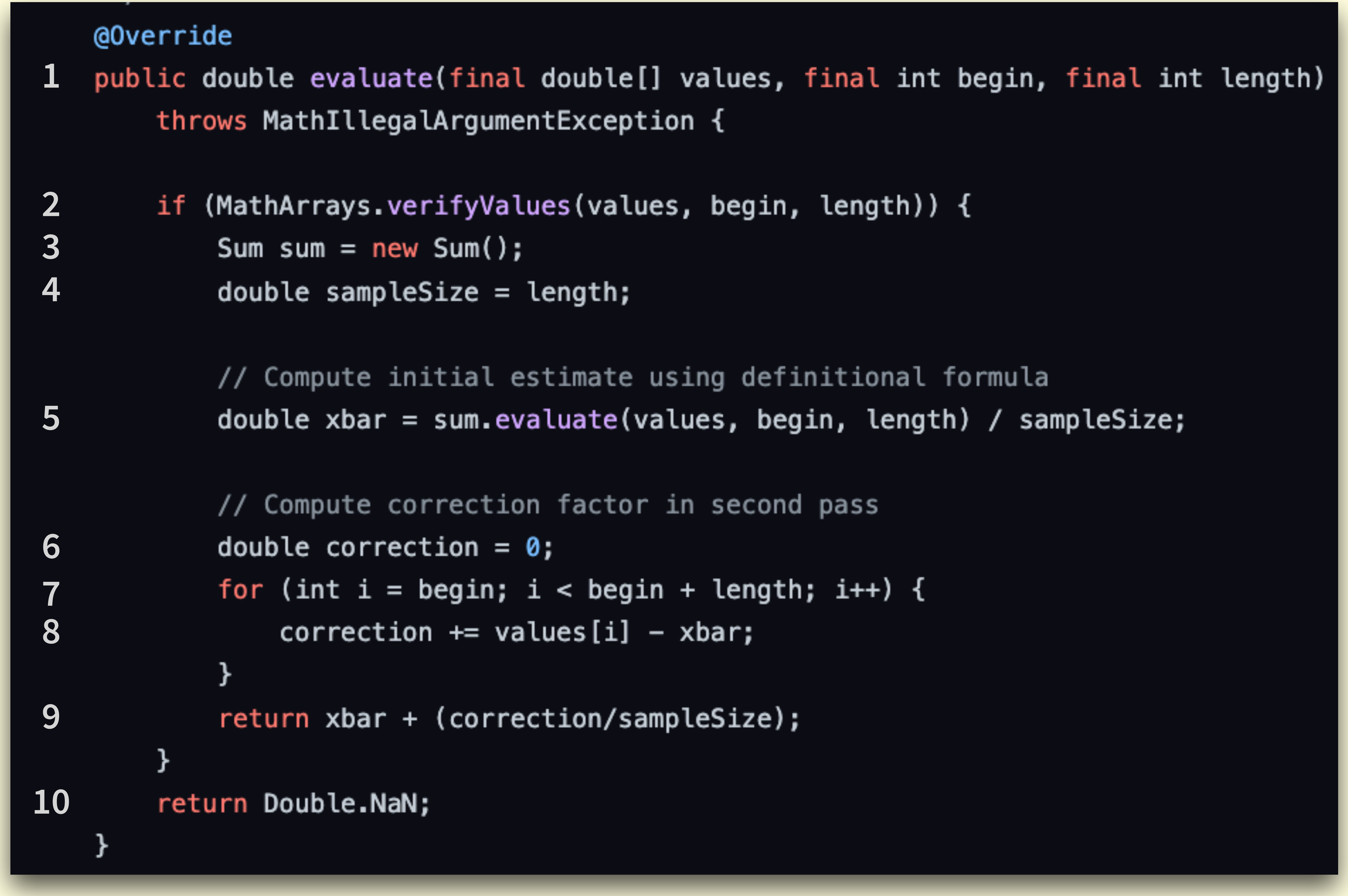
```
    @Override
1   public double evaluate(final double[] values, final int begin, final int length)
        throws MathIllegalArgumentException {

2       if (MathArrays.verifyValues(values, begin, length)) {
3           Sum sum = new Sum();
4           double sampleSize = length;

            // Compute initial estimate using definitional formula
5           double xbar = sum.evaluate(values, begin, length) / sampleSize;

            // Compute correction factor in second pass
6           double correction = 0;
7           for (int i = begin; i < begin + length; i++) {
8               correction += values[i] - xbar;
            }
9           return xbar + (correction/sampleSize);
        }
10      return Double.NaN;
    }
```



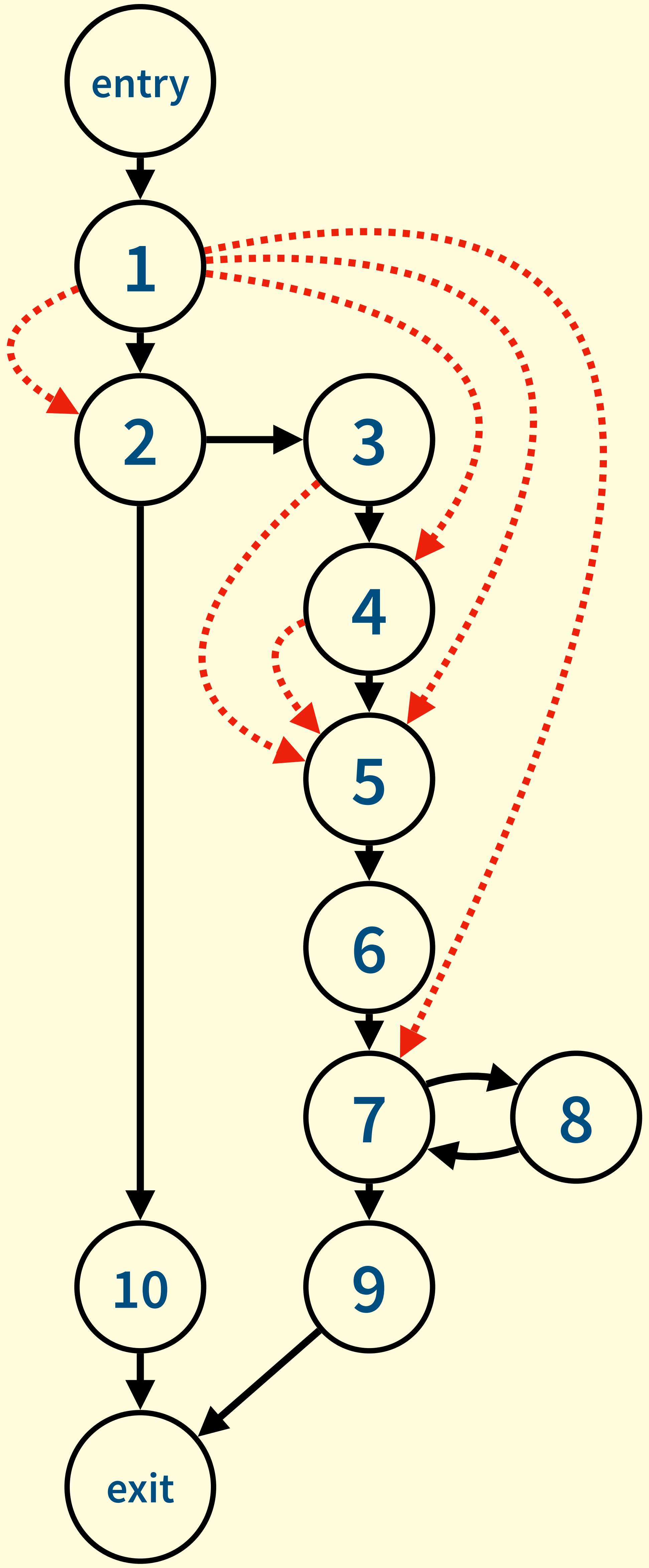


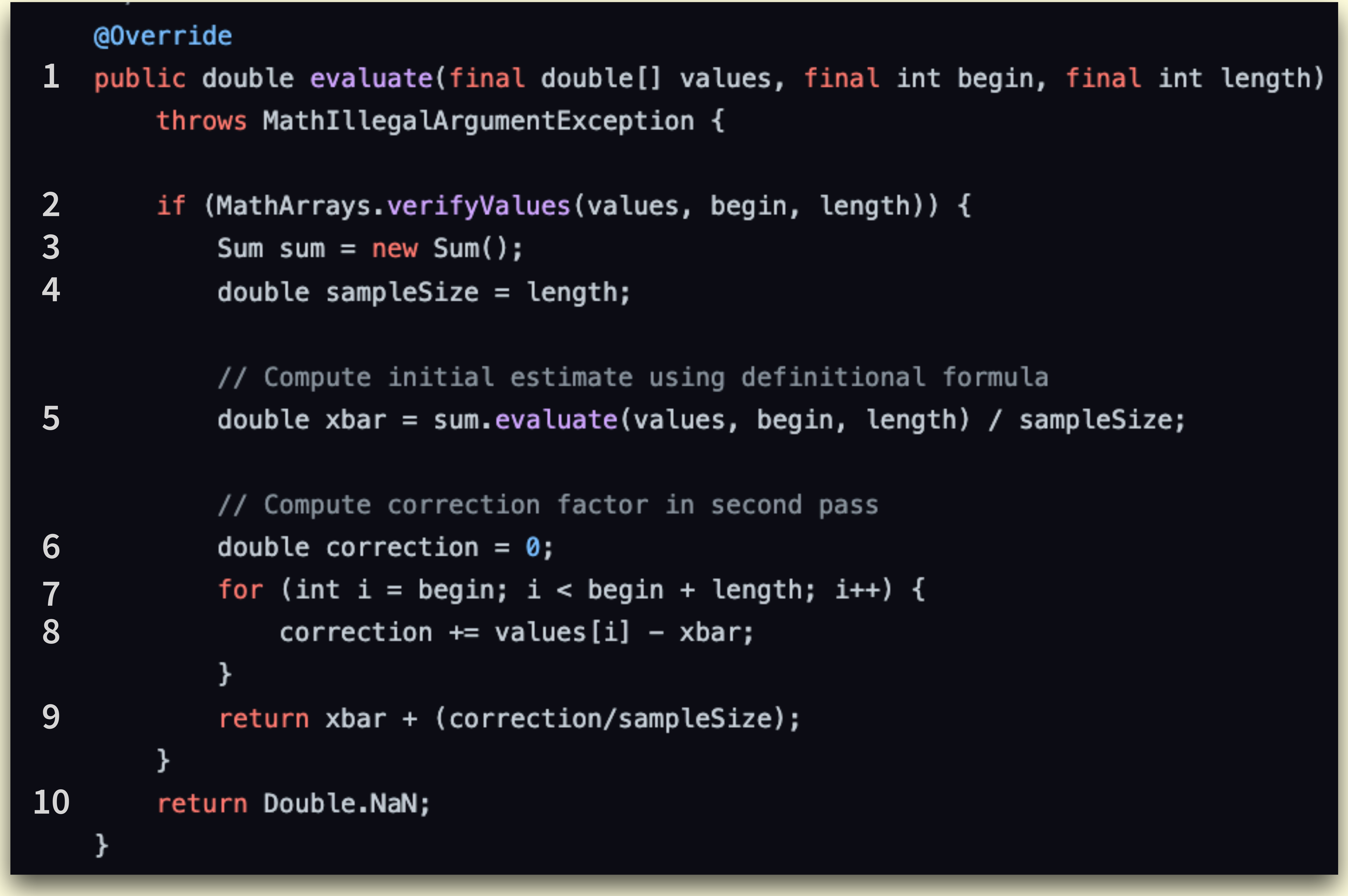
```
    @Override
1   public double evaluate(final double[] values, final int begin, final int length)
        throws MathIllegalArgumentException {

2       if (MathArrays.verifyValues(values, begin, length)) {
3           Sum sum = new Sum();
4           double sampleSize = length;

            // Compute initial estimate using definitional formula
5           double xbar = sum.evaluate(values, begin, length) / sampleSize;

            // Compute correction factor in second pass
6           double correction = 0;
7           for (int i = begin; i < begin + length; i++) {
8               correction += values[i] - xbar;
            }
9           return xbar + (correction/sampleSize);
        }
10      return Double.NaN;
    }
```



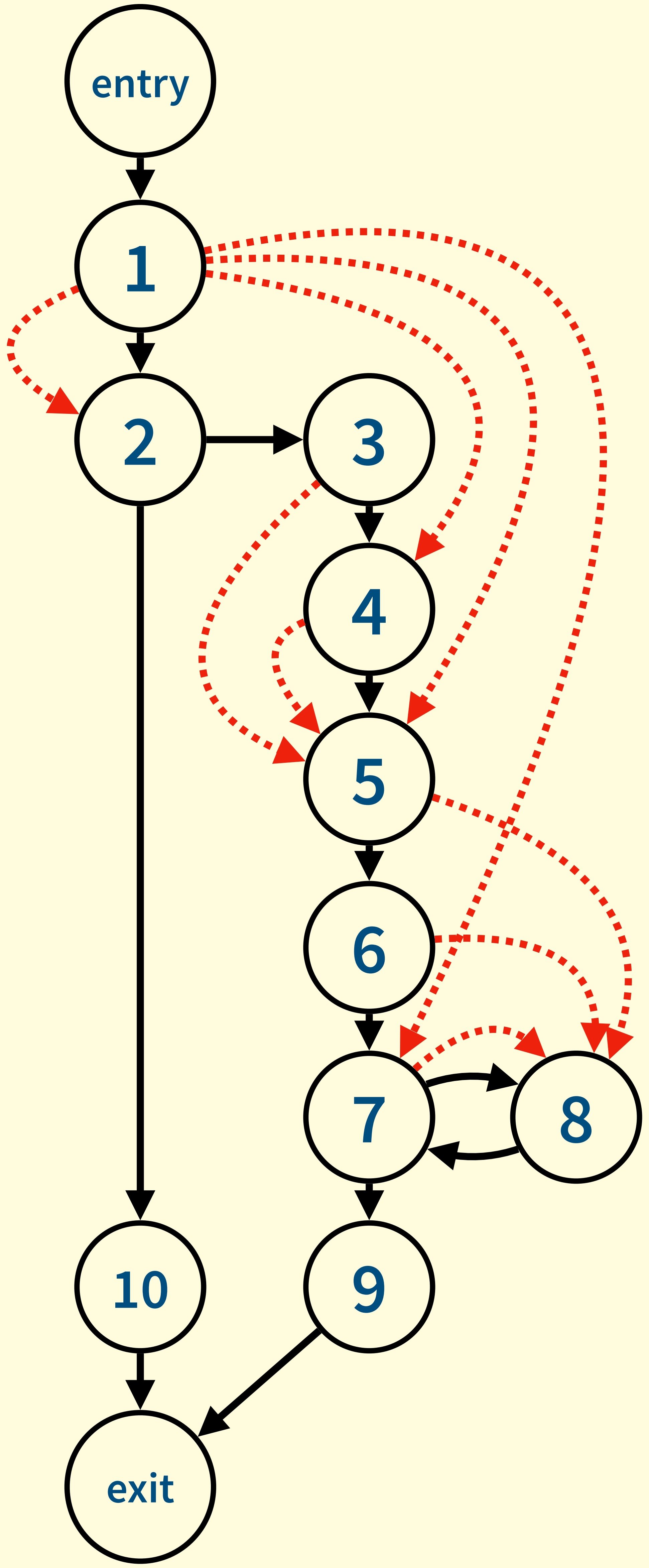


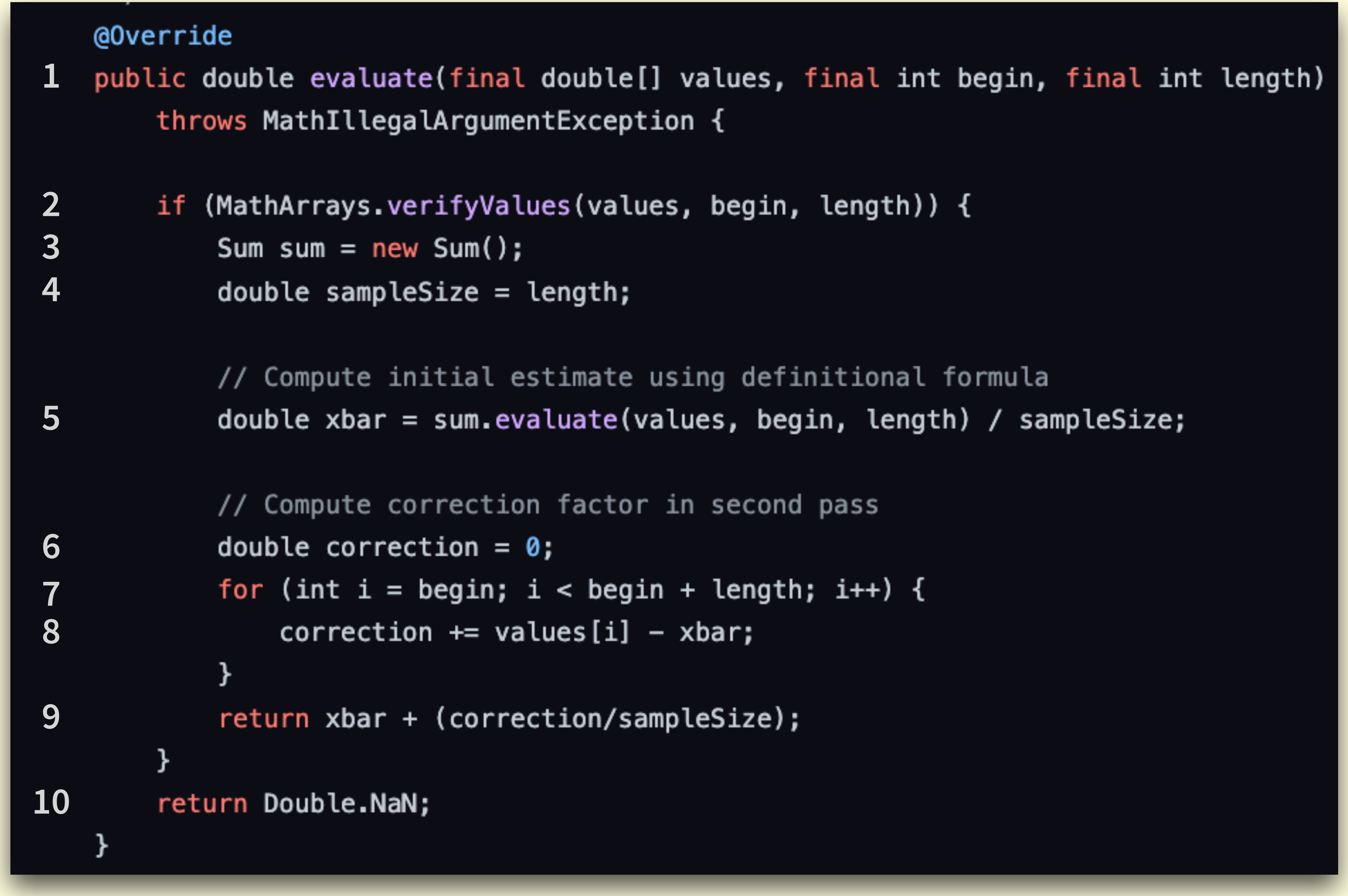
```
    @Override
1   public double evaluate(final double[] values, final int begin, final int length)
        throws MathIllegalArgumentException {

2       if (MathArrays.verifyValues(values, begin, length)) {
3           Sum sum = new Sum();
4           double sampleSize = length;

            // Compute initial estimate using definitional formula
5           double xbar = sum.evaluate(values, begin, length) / sampleSize;

            // Compute correction factor in second pass
6           double correction = 0;
7           for (int i = begin; i < begin + length; i++) {
8               correction += values[i] - xbar;
            }
9           return xbar + (correction/sampleSize);
        }
10      return Double.NaN;
    }
```



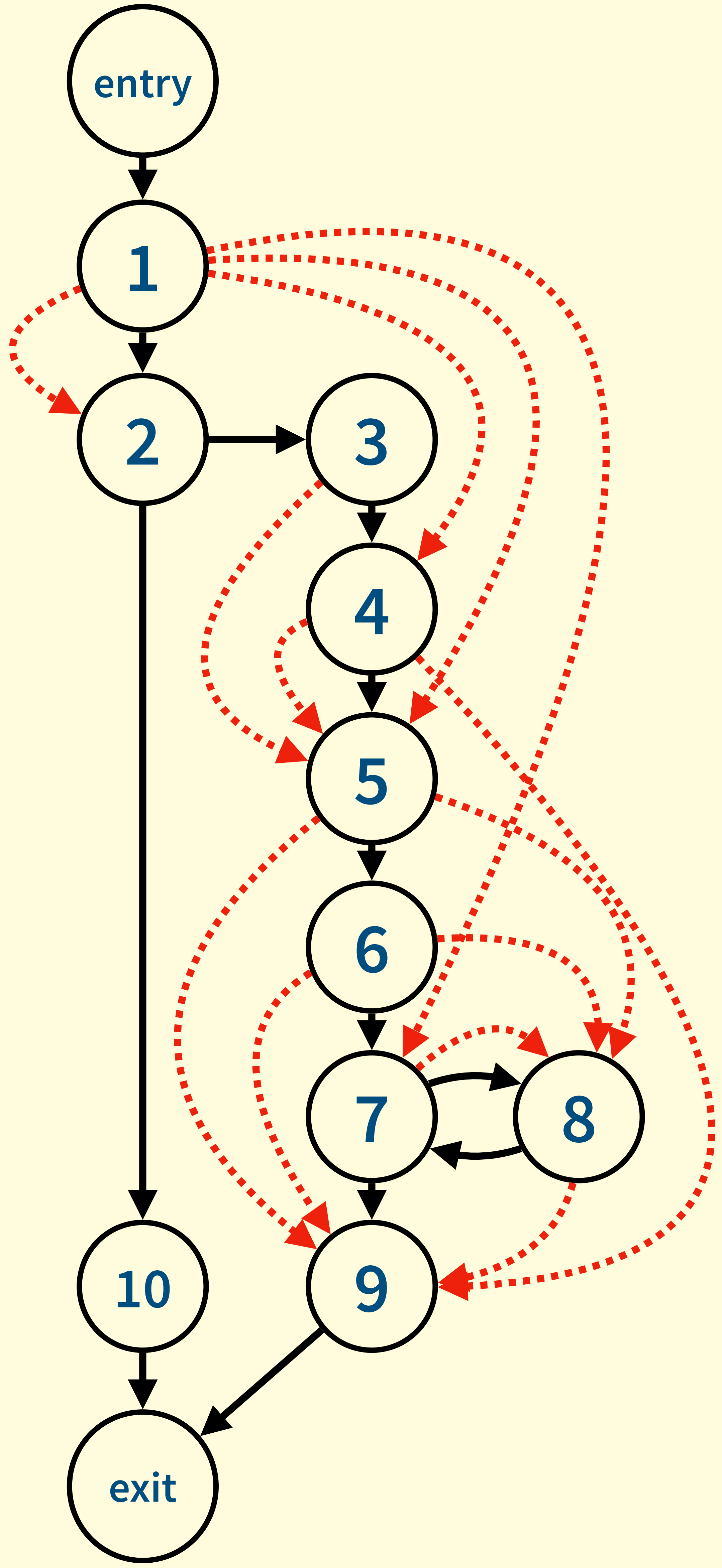


```
    @Override
1   public double evaluate(final double[] values, final int begin, final int length)
        throws MathIllegalArgumentException {

2       if (MathArrays.verifyValues(values, begin, length)) {
3           Sum sum = new Sum();
4           double sampleSize = length;

            // Compute initial estimate using definitional formula
5           double xbar = sum.evaluate(values, begin, length) / sampleSize;

            // Compute correction factor in second pass
6           double correction = 0;
7           for (int i = begin; i < begin + length; i++) {
8               correction += values[i] - xbar;
            }
9           return xbar + (correction/sampleSize);
        }
10      return Double.NaN;
    }
```

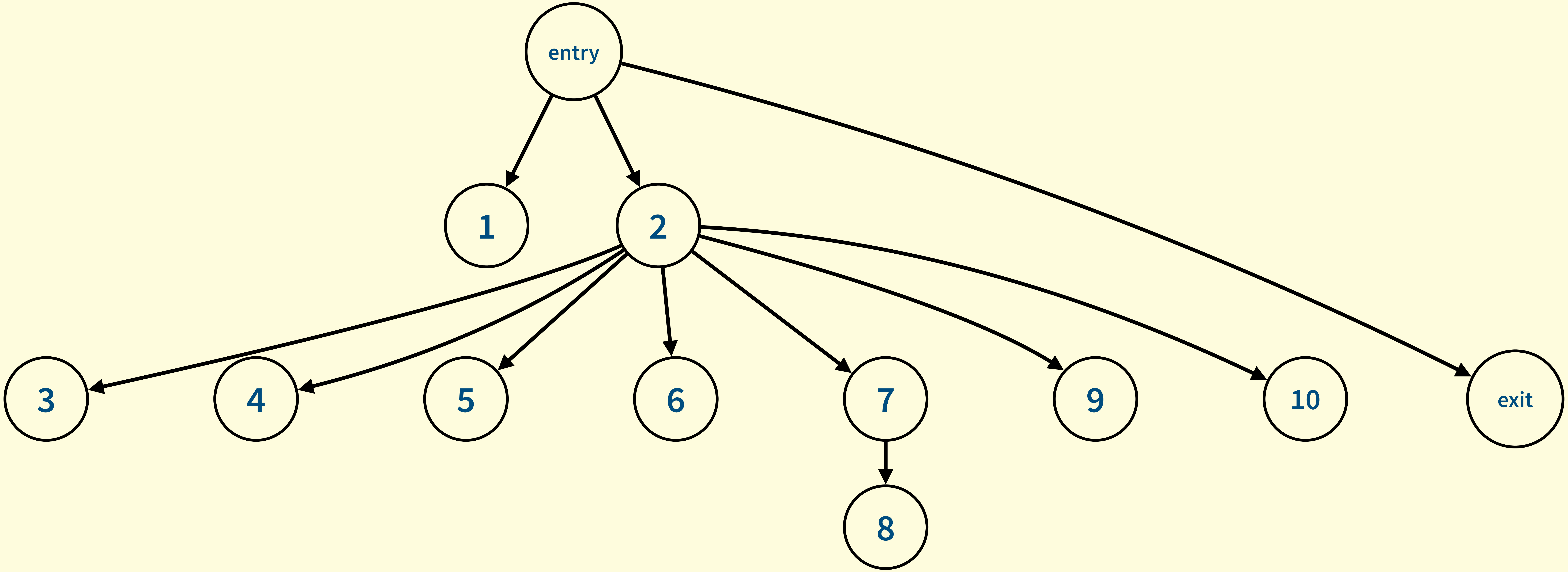




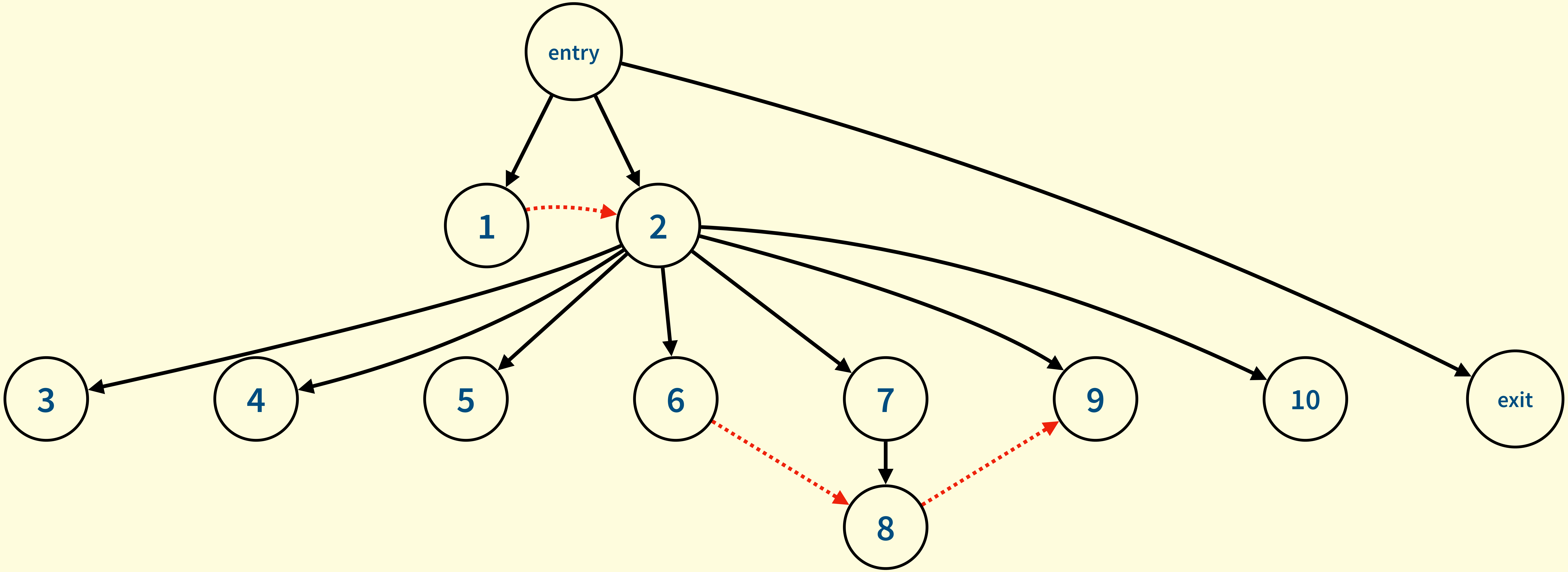
```
     @Override
 1   public double evaluate(final double[] values, final int begin, final int length)
         throws MathIllegalArgumentException {

 2       if (MathArrays.verifyValues(values, begin, length)) {
 3           Sum sum = new Sum();
 4           double sampleSize = length;

             // Compute initial estimate using definitional formula
 5           double xbar = sum.evaluate(values, begin, length) / sampleSize;

             // Compute correction factor in second pass
 6           double correction = 0;
 7           for (int i = begin; i < begin + length; i++) {
 8               correction += values[i] - xbar;
             }
 9           return xbar + (correction/sampleSize);
         }
10       return Double.NaN;
     }
```



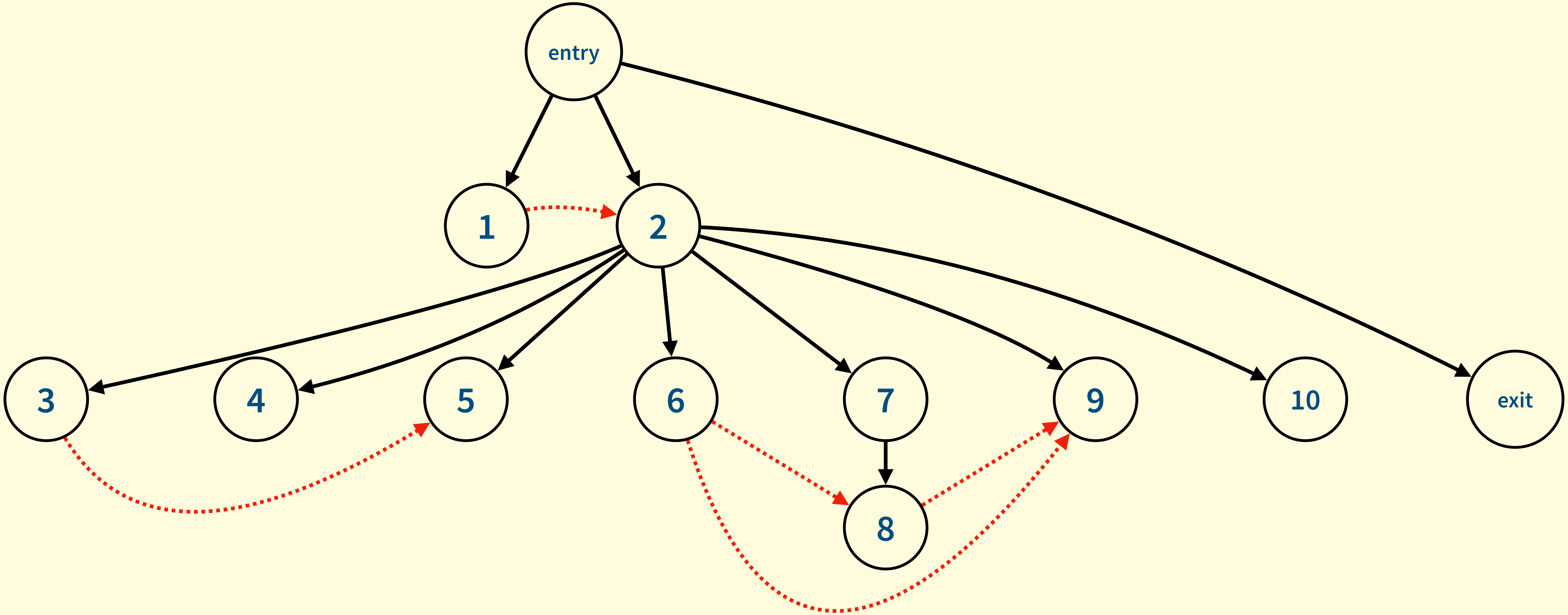
Putting it all together - the Program Dependence Graph



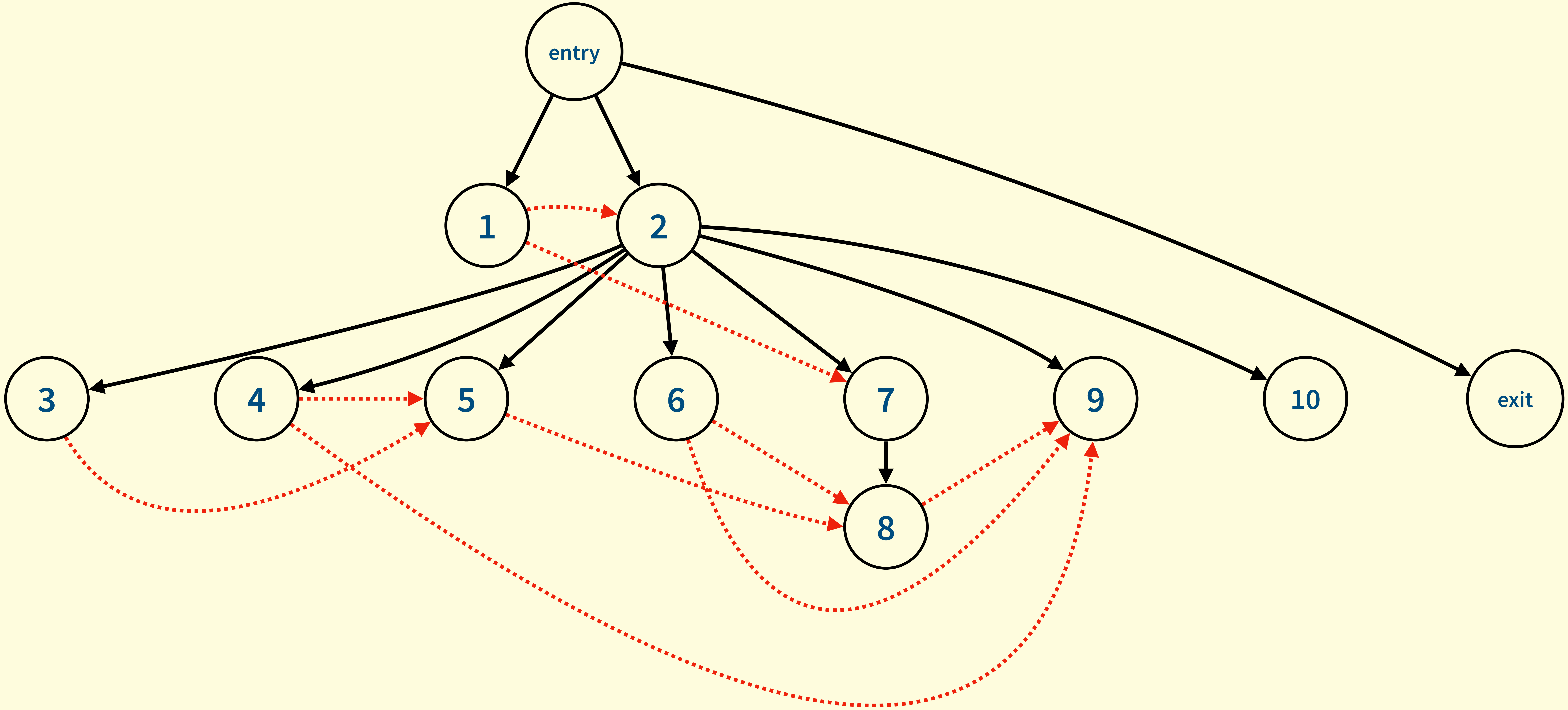
Putting it all together - the Program Dependence Graph



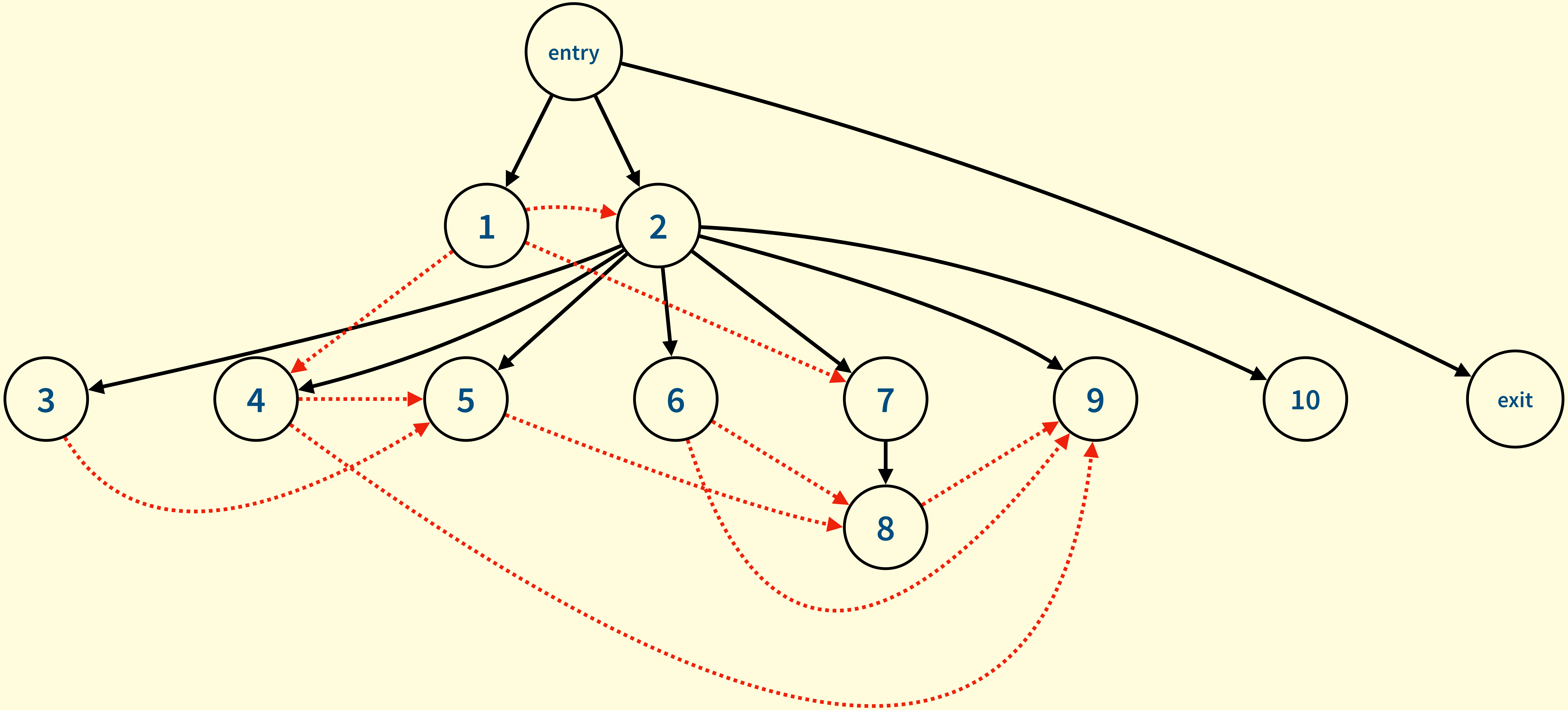
Putting it all together - the Program Dependence Graph



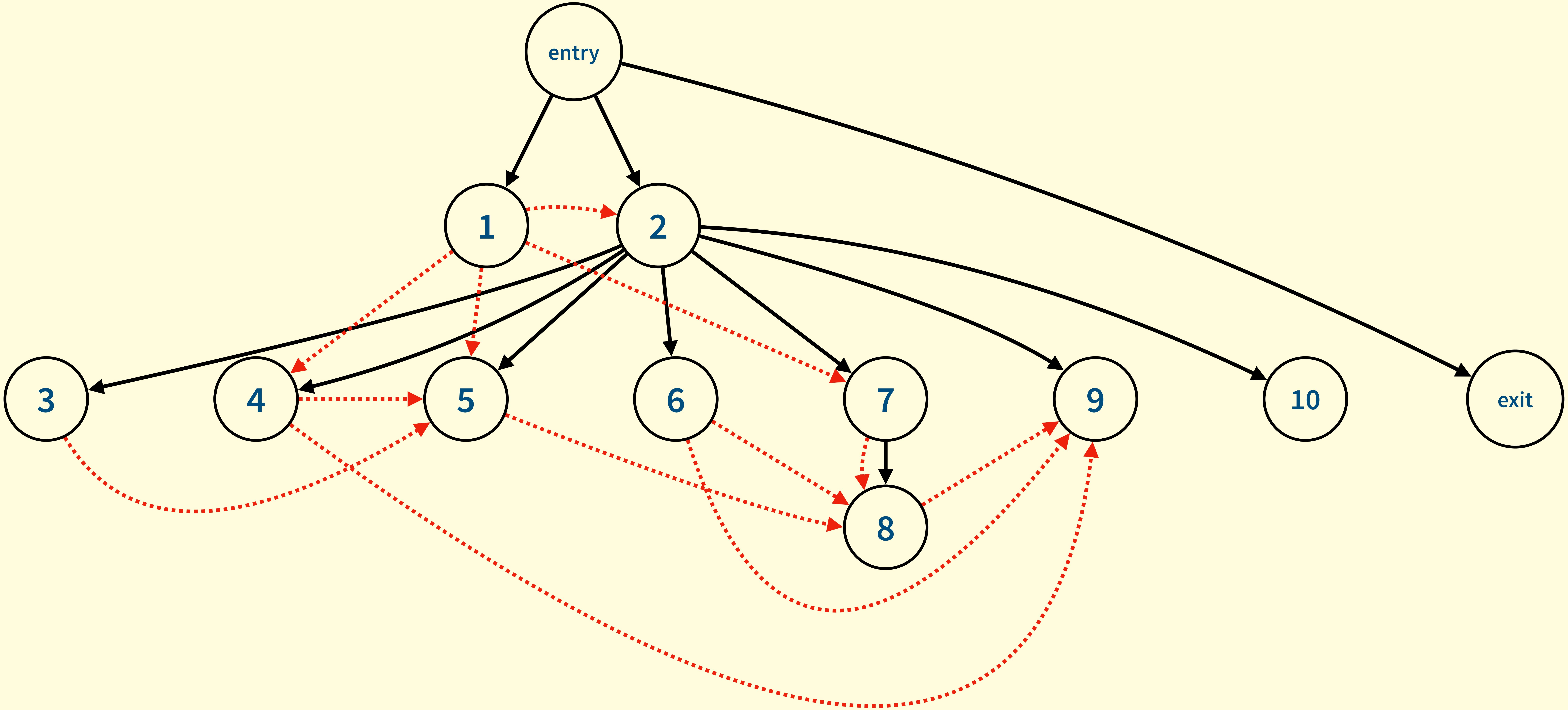
Putting it all together - the Program Dependence Graph



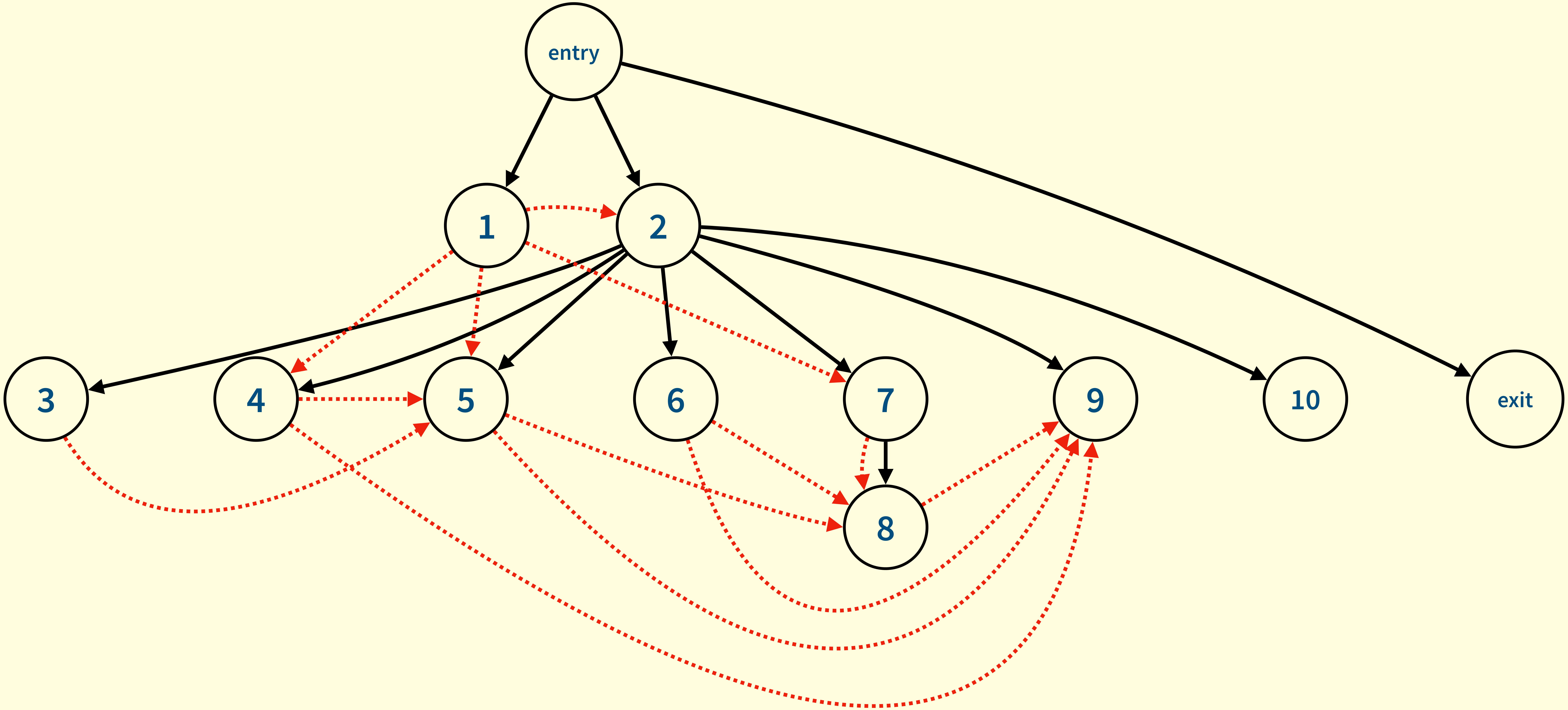
Putting it all together - the Program Dependence Graph



Putting it all together - the Program Dependence Graph



Putting it all together - the Program Dependence Graph



Code Slicing

A slice is a way of eliminating statements that are irrelevant to a slicing criterion.

A slicing criterion consists of a statement and a set of variables at that statement.

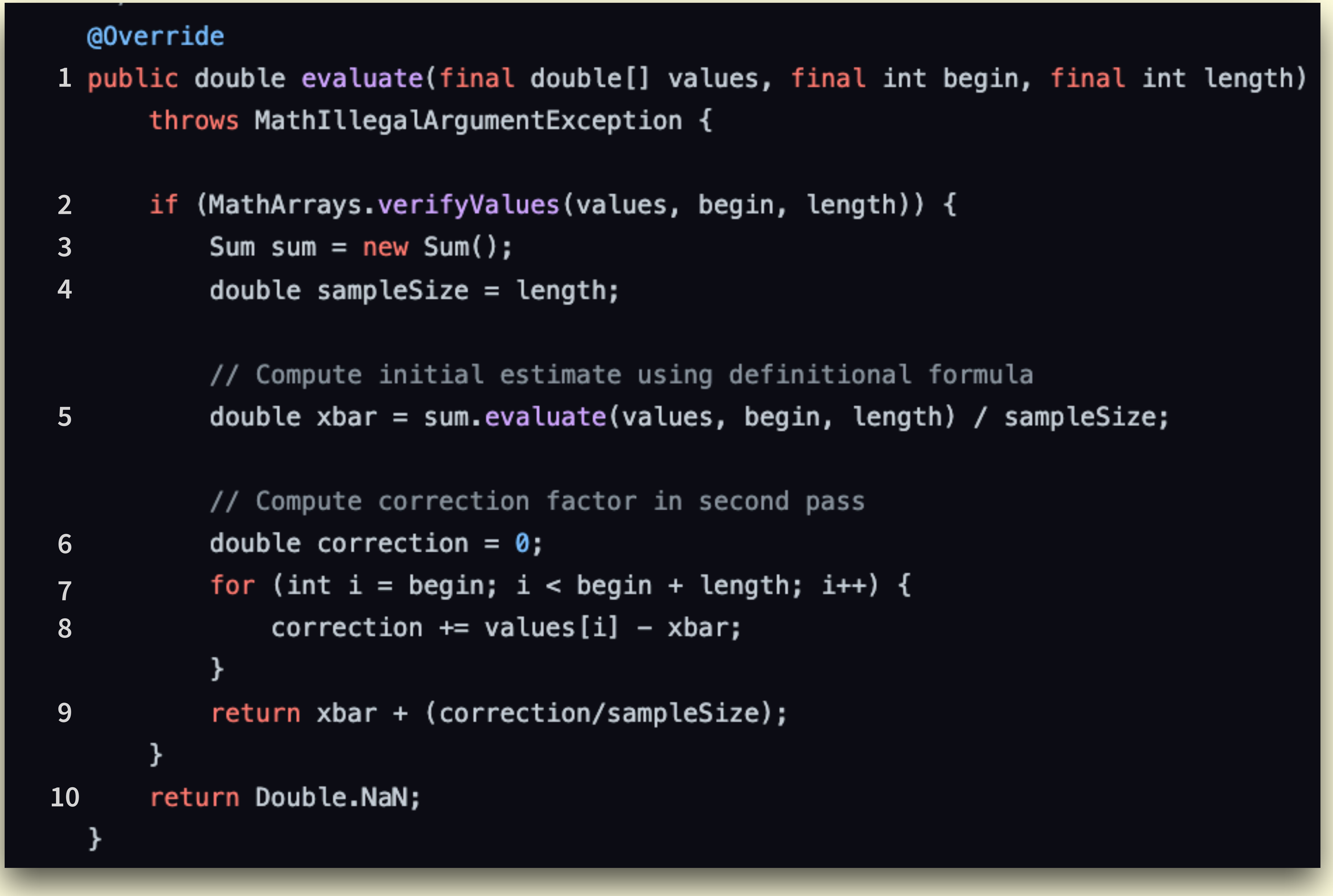
A backward slice is computed by identifying tracing all of the incoming paths to the criterion node in the PDG.

> Give me the statements that compute the slicing criterion.

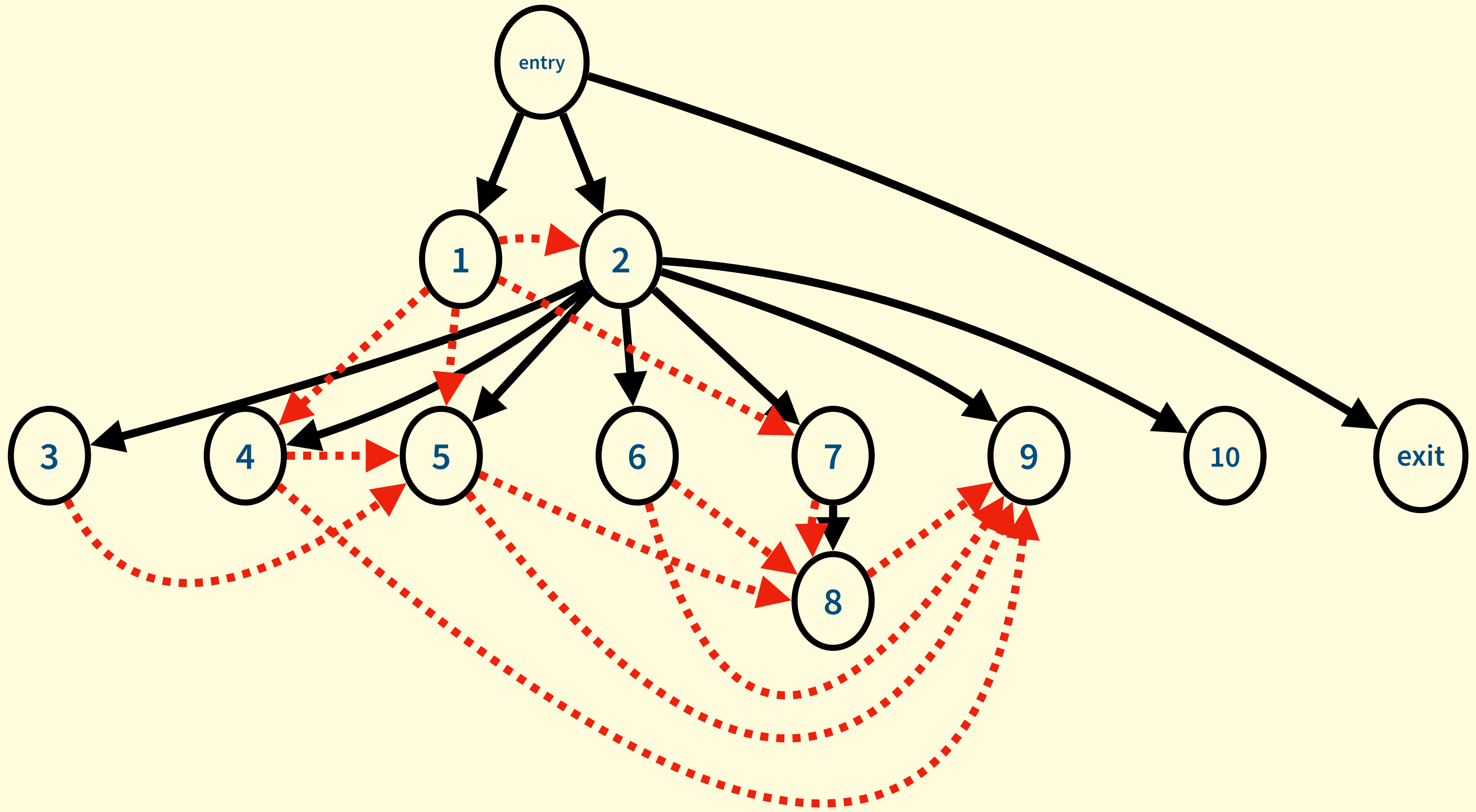
A forward slice is computed by identifying all outgoing paths from the criterion node in the PDG.

> Give me the sub-program that may be affected by the slicing criterion.

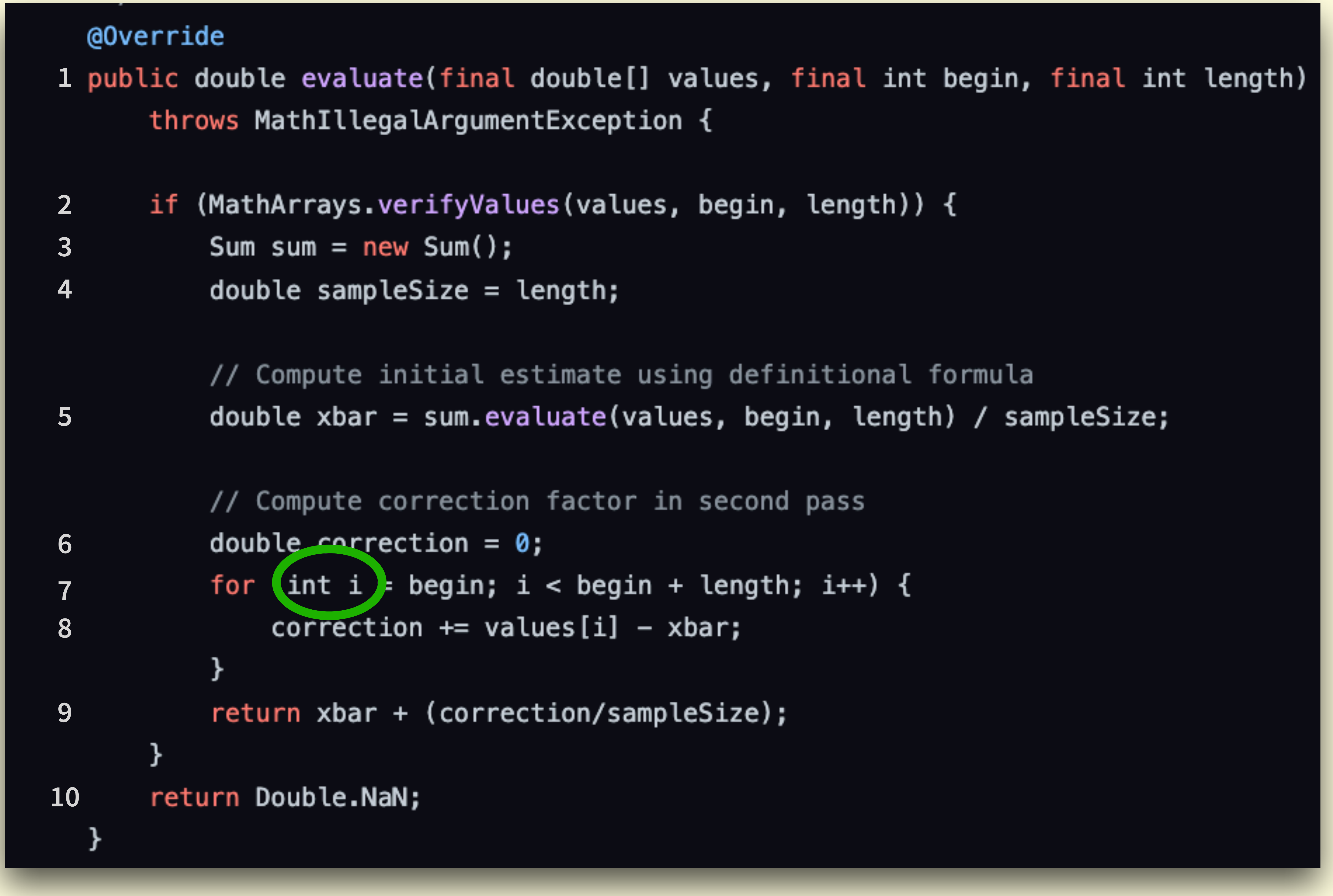
Putting it all together - the Program Dependence Graph



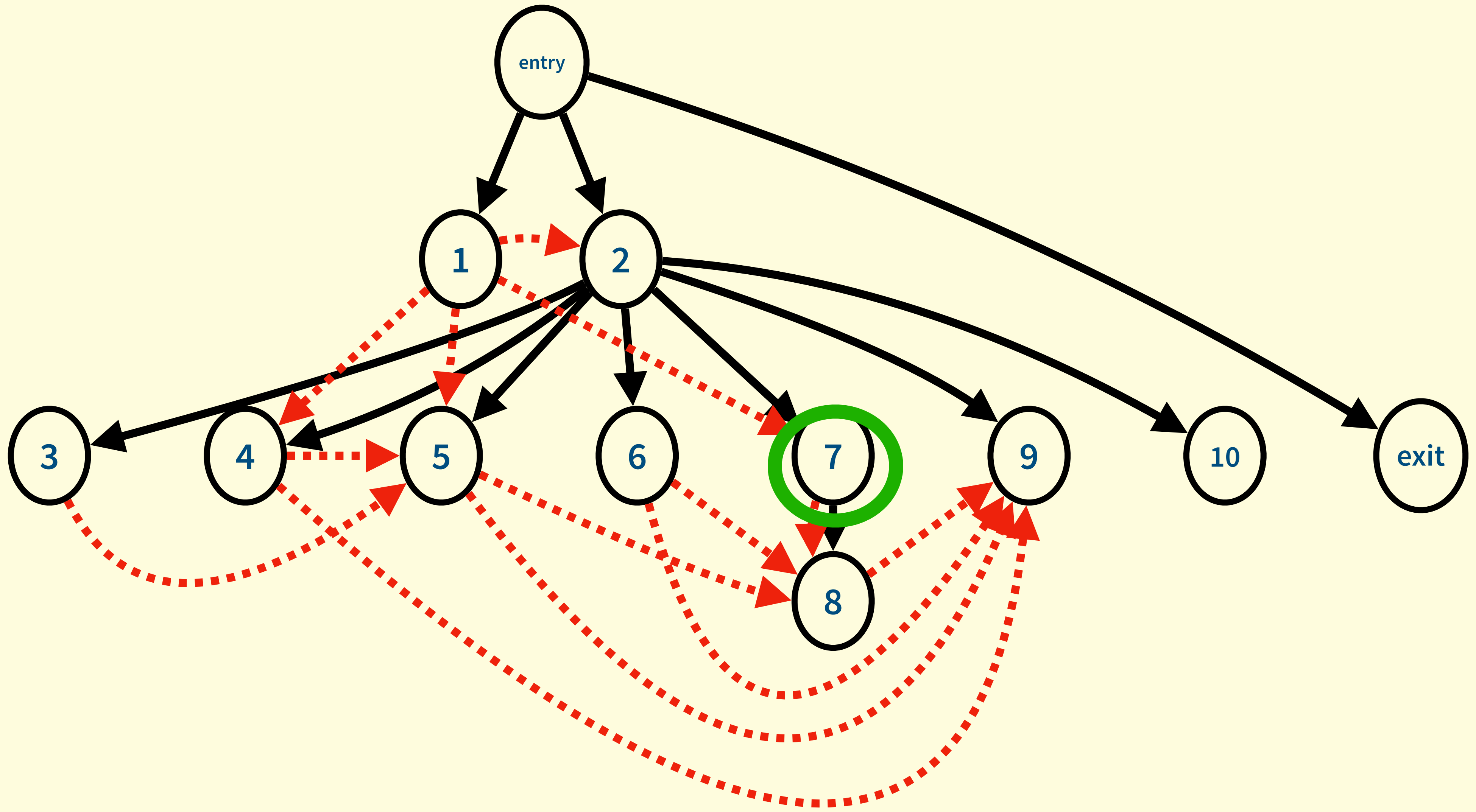
```
   @Override
1  public double evaluate(final double[] values, final int begin, final int length)
       throws MathIllegalArgumentException {

2      if (MathArrays.verifyValues(values, begin, length)) {
3          Sum sum = new Sum();
4          double sampleSize = length;

           // Compute initial estimate using definitional formula
5          double xbar = sum.evaluate(values, begin, length) / sampleSize;

           // Compute correction factor in second pass
6          double correction = 0;
7          for (int i = begin; i < begin + length; i++) {
8              correction += values[i] - xbar;
           }
9          return xbar + (correction/sampleSize);
       }
10     return Double.NaN;
   }
```



Putting it all together - the Program Dependence Graph

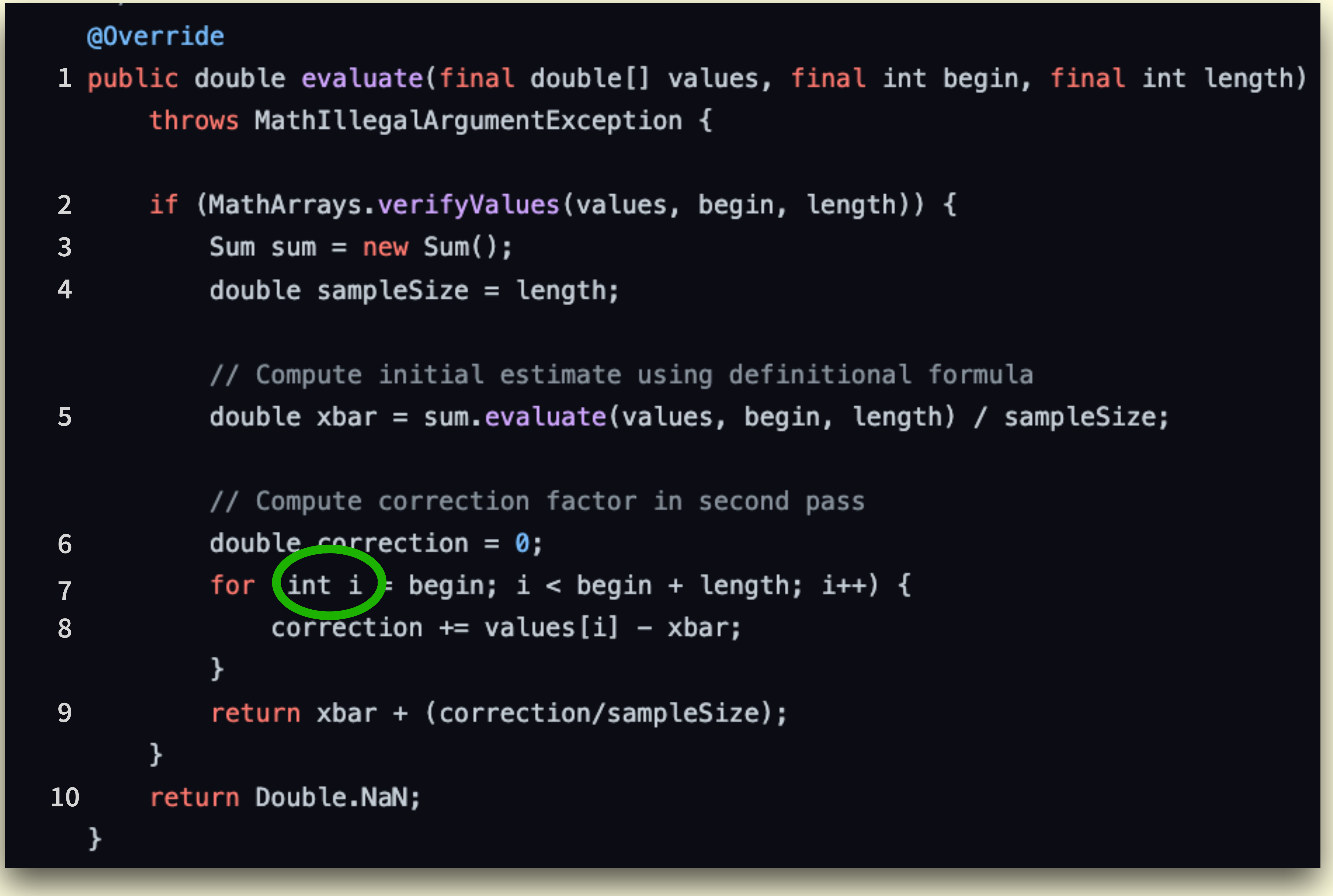


```
    @Override
1   public double evaluate(final double[] values, final int begin, final int length)
        throws MathIllegalArgumentException {

2       if (MathArrays.verifyValues(values, begin, length)) {
3           Sum sum = new Sum();
4           double sampleSize = length;

            // Compute initial estimate using definitional formula
5           double xbar = sum.evaluate(values, begin, length) / sampleSize;

            // Compute correction factor in second pass
6           double correction = 0;
7           for (int i = begin; i < begin + length; i++) {
8               correction += values[i] - xbar;
            }
9           return xbar + (correction/sampleSize);
        }
10      return Double.NaN;
    }
```

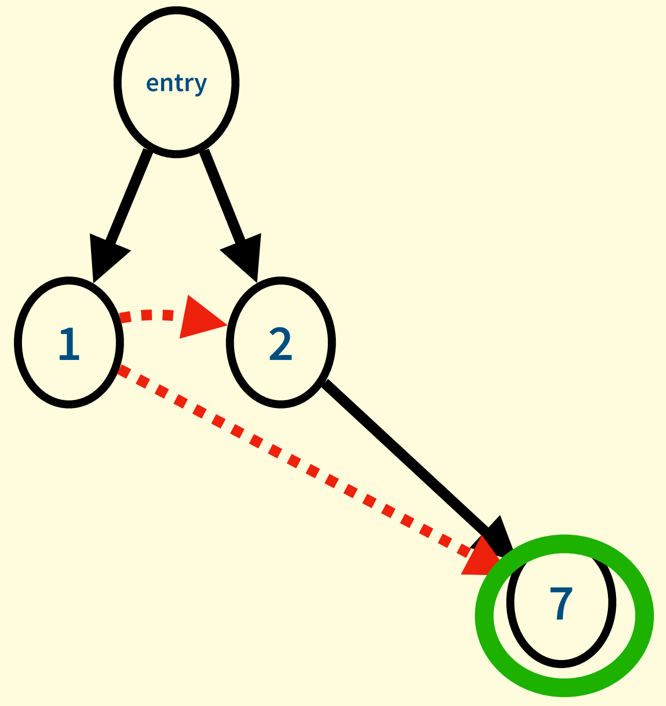


Slicing criterion: <7,{i}>

Putting it all together - the Program Dependence Graph

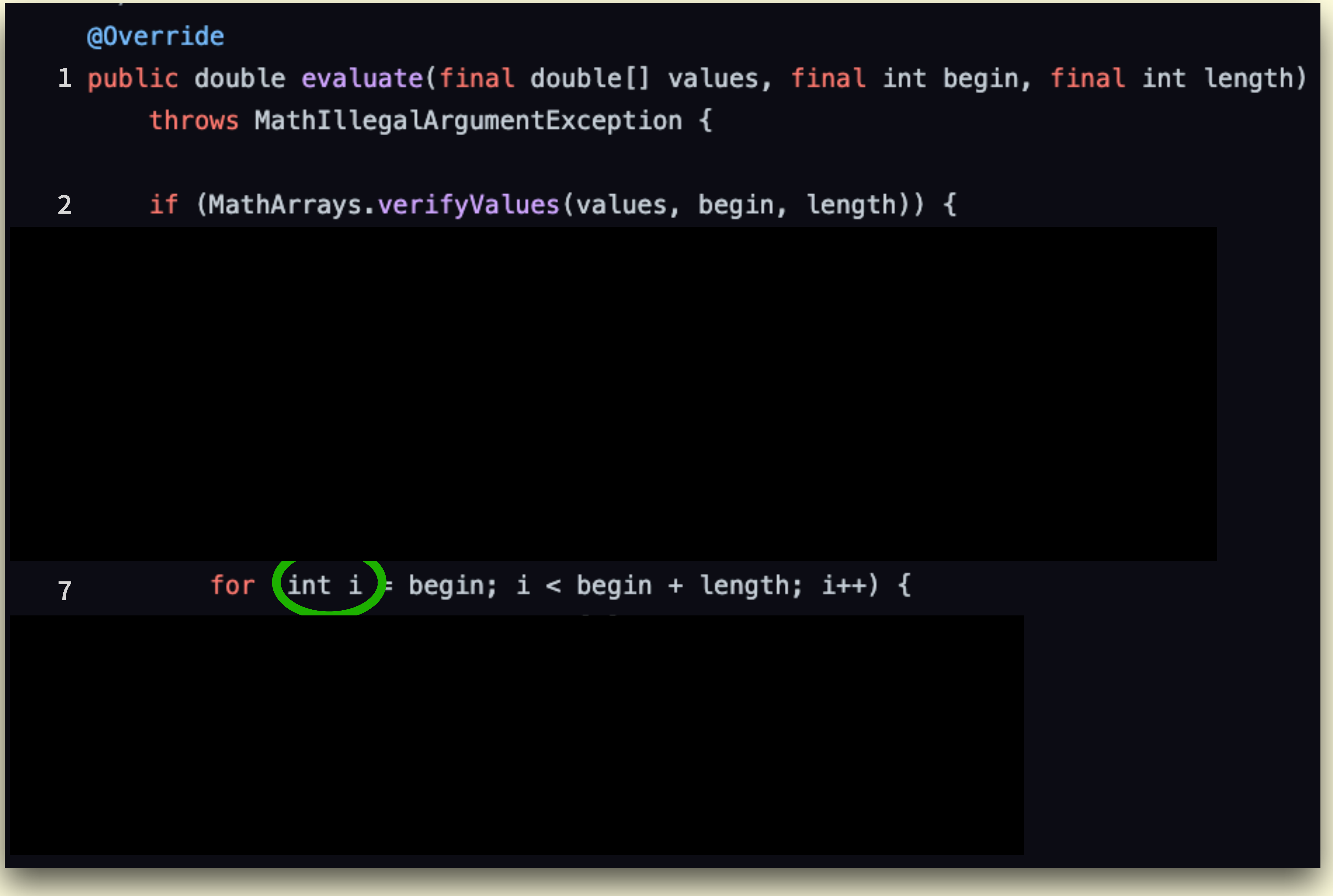


```
    @Override
 1  public double evaluate(final double[] values, final int begin, final int length)
        throws MathIllegalArgumentException {

 2      if (MathArrays.verifyValues(values, begin, length)) {
 3          Sum sum = new Sum();
 4          double sampleSize = length;

            // Compute initial estimate using definitional formula
 5          double xbar = sum.evaluate(values, begin, length) / sampleSize;

            // Compute correction factor in second pass
 6          double correction = 0;
 7          for (int i = begin; i < begin + length; i++) {
 8              correction += values[i] - xbar;
            }
 9          return xbar + (correction/sampleSize);
        }
10      return Double.NaN;
    }
```

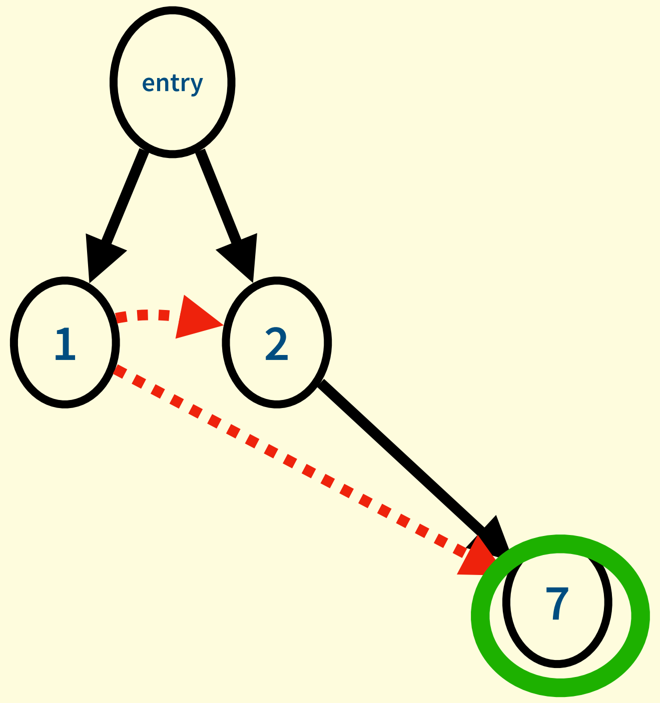


Slicing criterion: <7,{i}>

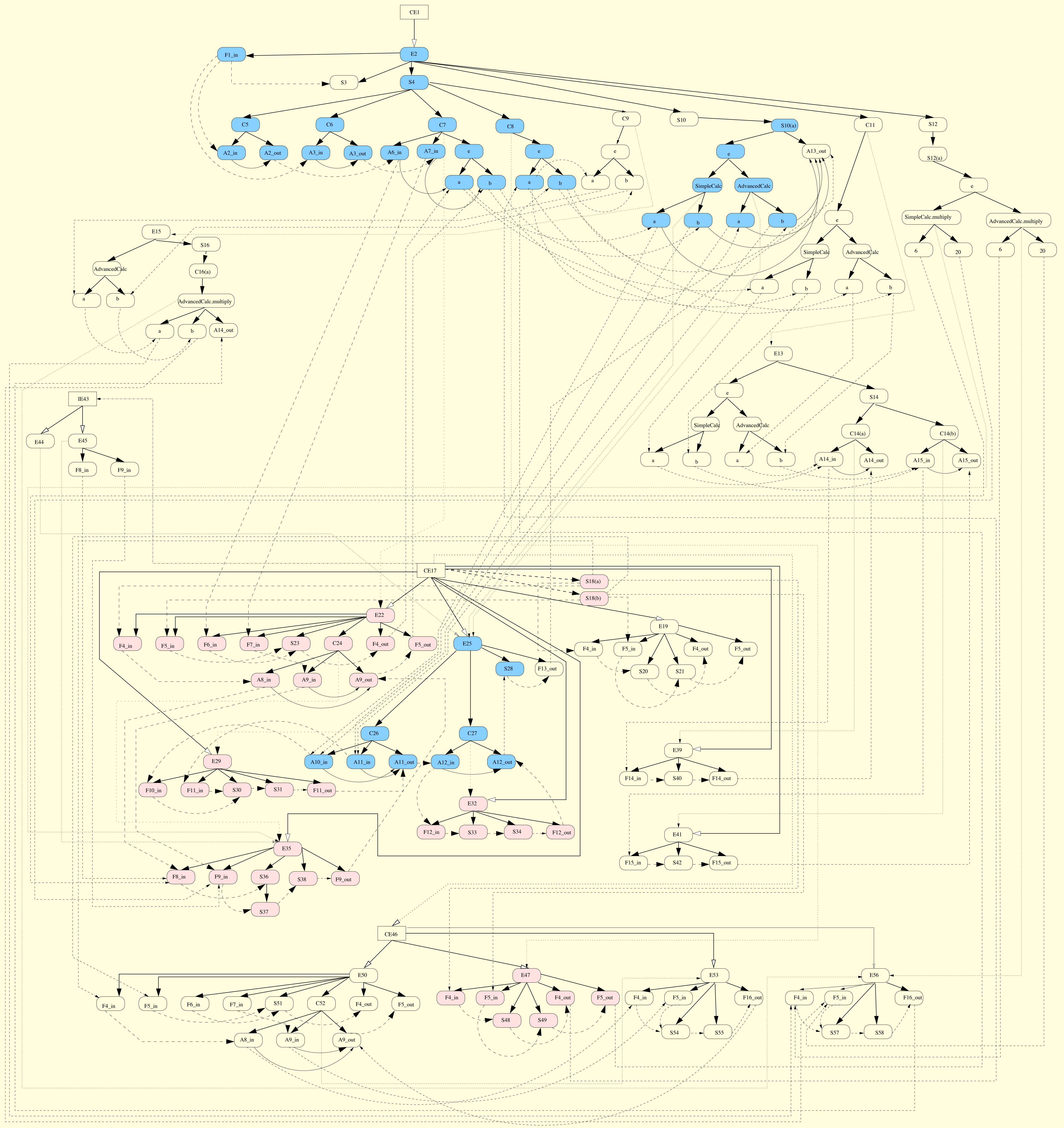
Putting it all together - the Program Dependence Graph



```
  @Override
1 public double evaluate(final double[] values, final int begin, final int length)
      throws MathIllegalArgumentException {

2     if (MathArrays.verifyValues(values, begin, length)) {
[redacted]
7         for (int i = begin; i < begin + length; i++) {
[redacted]
```



Slicing criterion: <7,{i}>



```
CE1   public class Execute{
E2        public static void main(String args[]){
S3            SimpleCalc e;
S4            if(args.length > 0){
C5                int a = Integer.parseInt(args[0]);
C6                int b = Integer.parseInt(args[1]);
C7                e = new SimpleCalc(a, b);
              }
              else
              {
C8                e = new AdvancedCalc();
C9                computePower(e);
              }
S10           System.out.println(e.average());
C11           getStats(e);
S12           System.out.println(e.multiply(6,20));
          }
E13       public void getStats(SimpleCalc e){
S14           System.out.println(“a: “+ e.getA() + “ b: “ + e.getB());
          }
E15       public void computePower(AdvancedCalc e){
S16           System.out.println(e.power());
          }
      }
```

```
CE17  public class SimpleCalc implements Calculator{
S18       int a,b;
E19       public SimpleCalc(){
S20           a = 6;
S21           b = 20;
          }
E22       public SimpleCalc(int aIn, int bIn){
S23           a = aIn;
C24           b = multiply(a, bIn);
          }
E25       public int average(){
C26           int added = add(a,b);
C27           int divided = divide(added);
S28           return divided;
          }
E29       private int add(int c, int d){
S30           int result = c+d;
S31           return result;
          }
E32       private int divide(int c){
S33           int result = c/2;
S34           return result;
          }
E35       protected int multiply(int c, int d){
S36           for(int i=0; i<c; i++){
S37               d=d+d;
              }
S38           return d;
          }
E39       public int getA(){
S40           return a;
          }
E41       public int getB(){
S42           return b;
          }
      }
```

```
IE43  interface Calculator{
E44       int average();
E45       int multiply(int c, int d);
      }

CE46  public class AdvancedCalc extends SimpleCalc{
E47       public AdvancedCalc(){
S48           a = 6;
S49           b = 20;
          }
E50       public AdvancedCalc(int aIn, int bIn){
S51           a = aIn;
C52           b = multiply(a, bIn);
          }
E53       protected int multiply(int c, int d){
S54           int result = c*d;
S55           return result
          }
E56       public int power(){
S57           int result=a^b;
S58           return result
          }
      }
```

What does this allow us to do?

Detect code clones (by identifying patterns in the PDG).

Debug - find the code that might be responsible for a faulty variable value.

Check for vulnerabilities.

- Could this variable possibly affect critical code?

Compute metrics from graph.

- Overlap between slices for the same code indicates cohesion.

Inter-procedural analysis

Calls

When a method invokes another method.

May be in the same class.

May be “static”.

- The target method can be executed from anywhere.
- Does not read or write the object data state.

May be in a different class.

- May be in a class hierarchy with multiple (overridden) implementations of the same signature.
- Can only know the target at runtime.

Calls

When a method invokes another method.

May be in the same class.

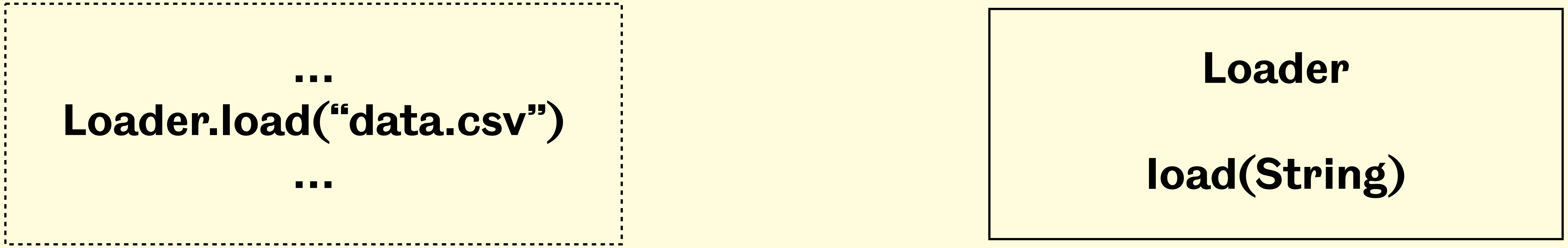
May be “static”.

- The target method can be executed from anywhere.
- Does not read or write the object data state.

May be in a different class.

- May be in a class hierarchy with multiple (overridden) implementations of the same signature.
- Can only know the target at runtime.

Points-To Analysis

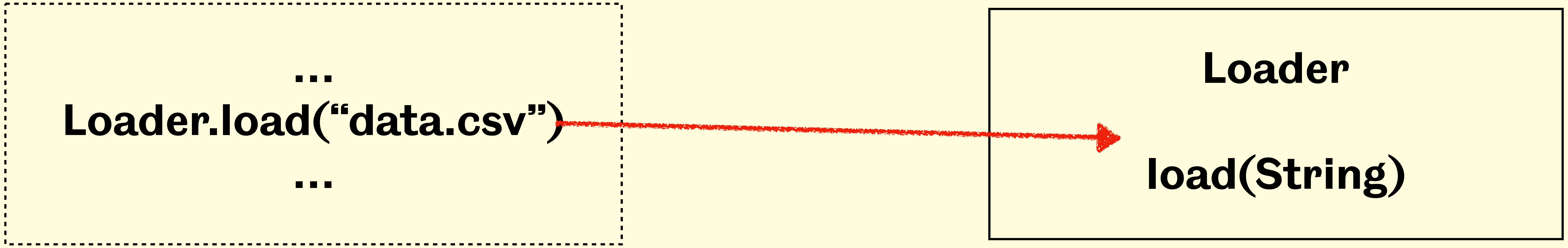


...
Loader.load(“data.csv”)
...

Loader

load(String)

Points-To Analysis



Points-To Analysis

Identify the possible destination(s) of a reference.

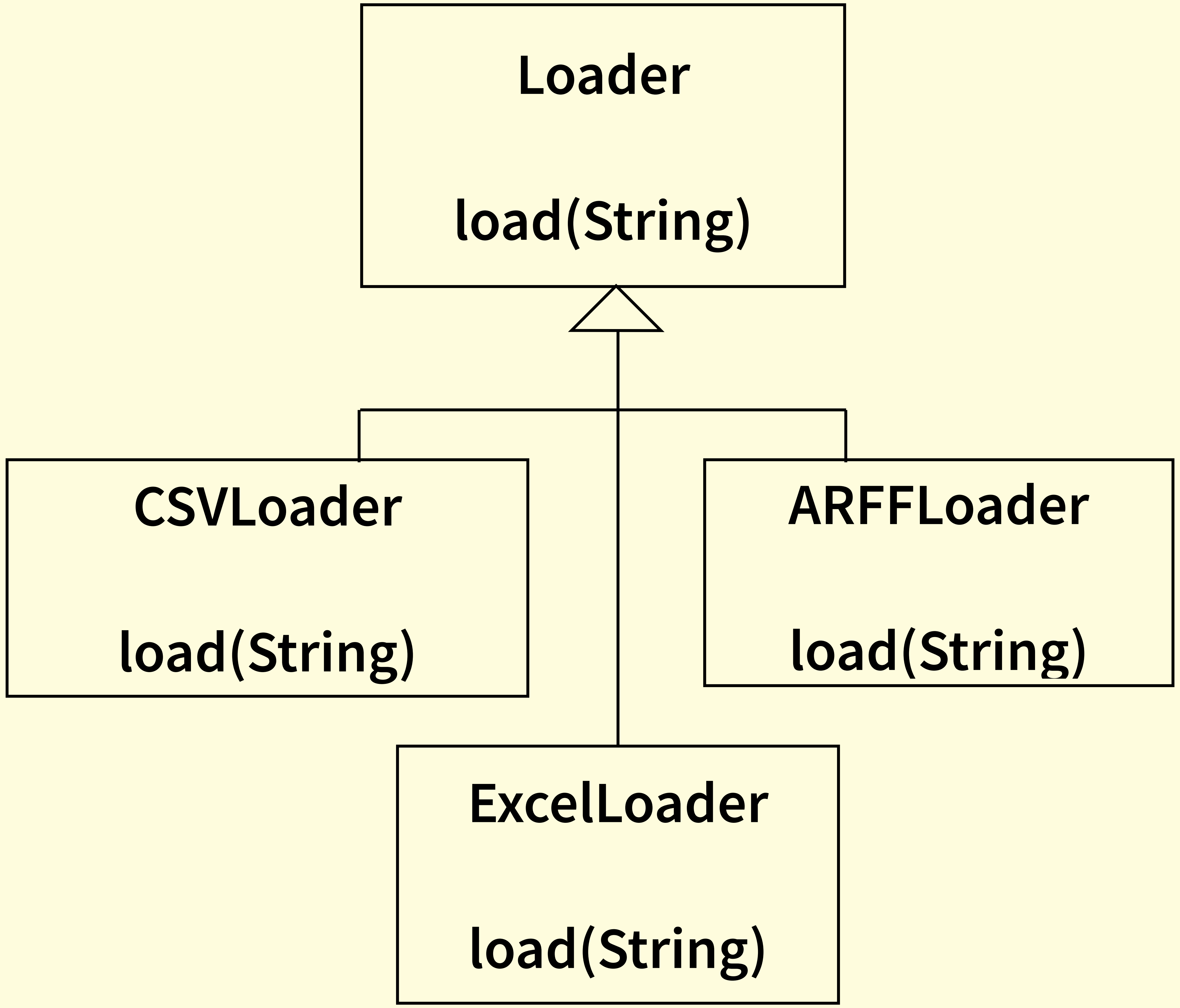
- Lots of possible algorithms.
- Tend to trade-off efficiency against accuracy.

Class Hierarchy Analysis (CHA)

- For any class that is the target of a call, identify any sub-classes with overriding methods.
- Make these methods potential targets.

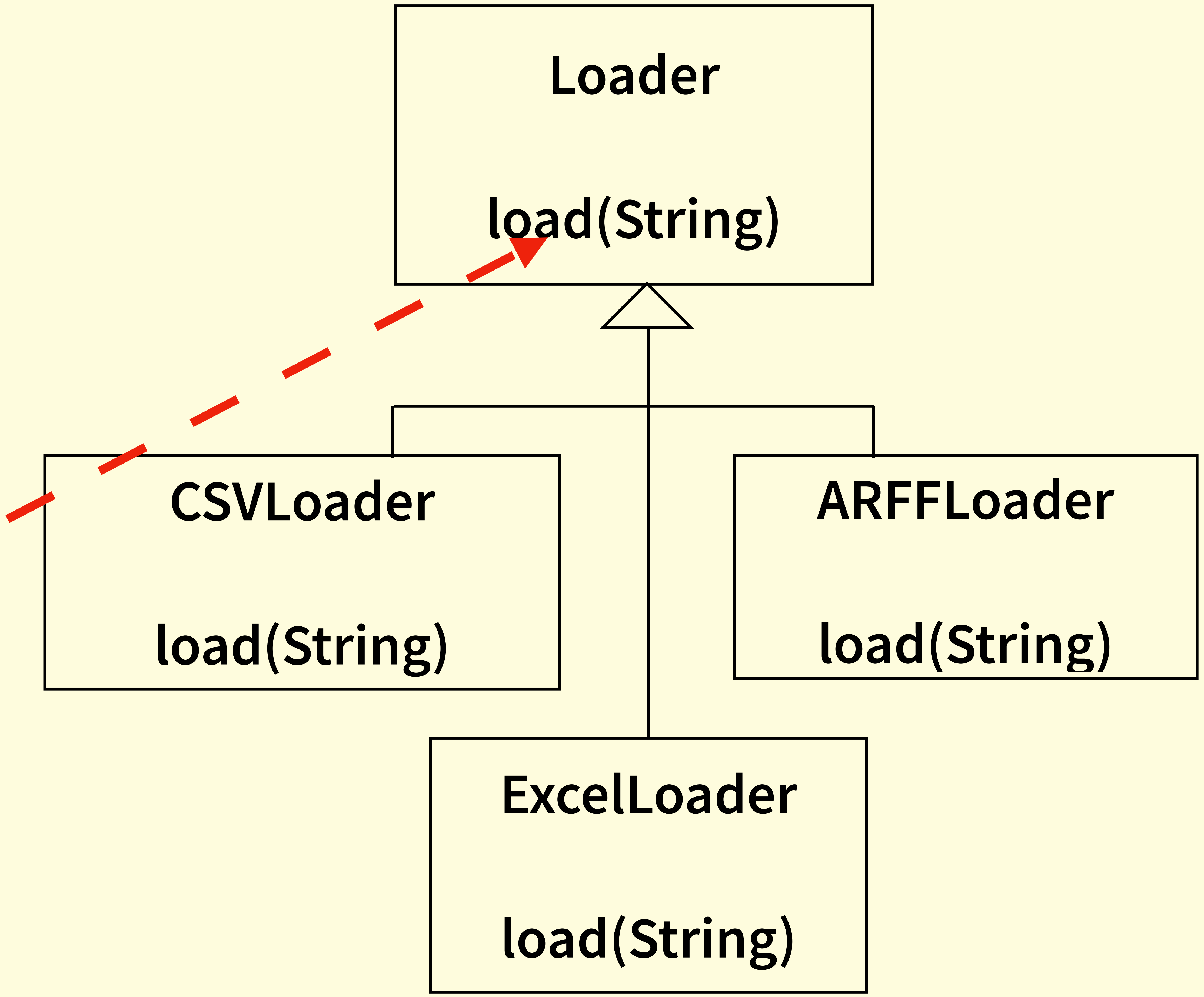
Points-To Analysis

```
public static void main(String[] args){
   Loader l = new CSVLoader();
   loadFile(l);
}

public void arffLoad(){
   Loader l = new ARFFLoader();
   loadFile(l);
}

protected void loadFile(Loader l){
   l.load(“data.csv”);
}
```



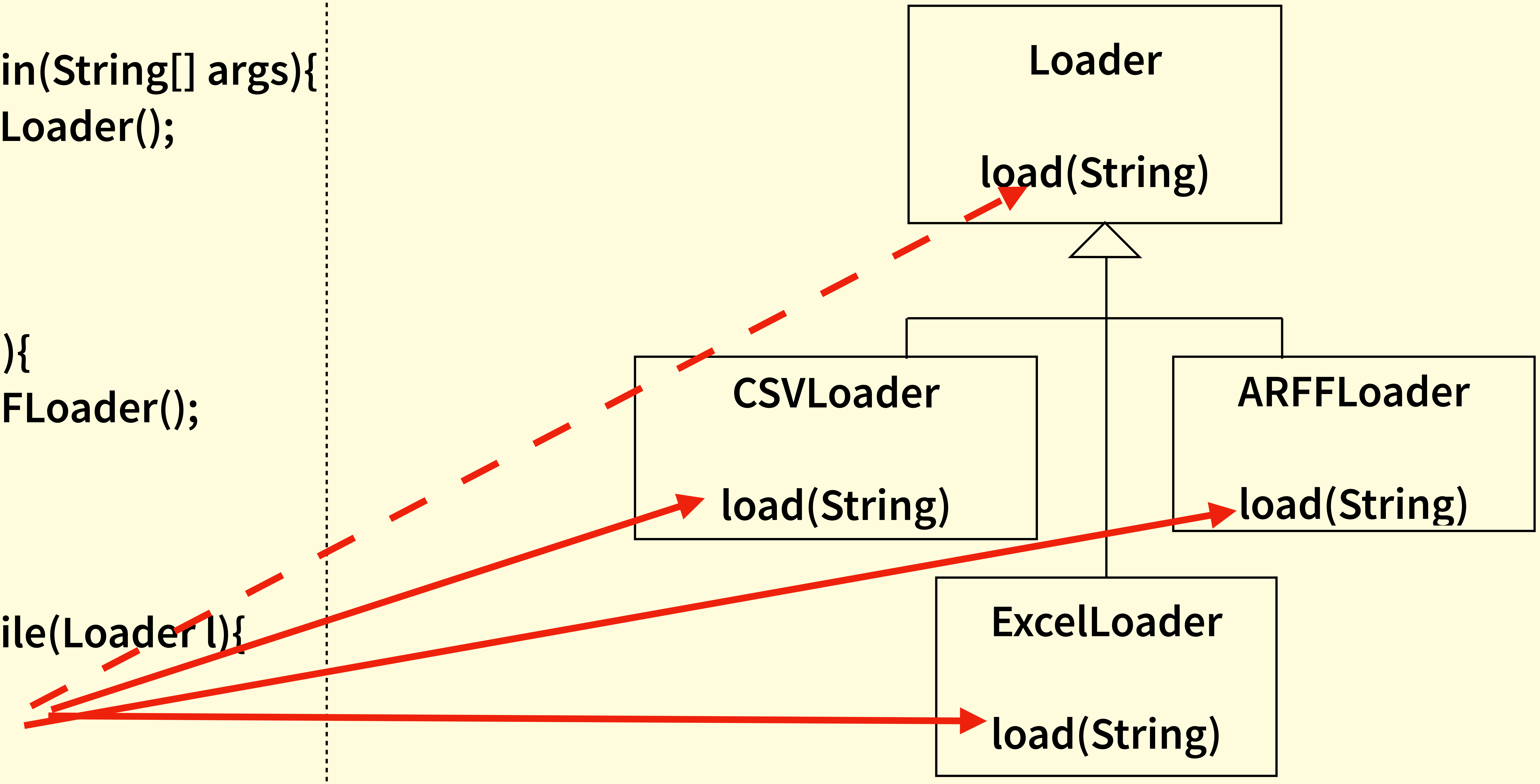
Points-To Analysis

```
public static void main(String[] args){
   Loader l = new CSVLoader();
   loadFile(l);
}

public void arffLoad(){
   Loader l = new ARFFLoader();
   loadFile(l);
}

protected void loadFile(Loader l){
   l.load(“data.csv”);
}
```



Points-To Analysis

```
public static void main(String[] args){
   Loader l = new CSVLoader();
   loadFile(l);
}

public void arffLoad(){
   Loader l = new ARFFLoader();
   loadFile(l);
}

protected void loadFile(Loader l){
   l.load(“data.csv”);
}
```



Fan-in and Fan-out metrics

Call graph can be used to quantify this interconnectedness via metrics.

Fan-in:

Number of incoming calls to a method or a class.

Provides an idea of how “critical” or “useful” a class or method is.

Fan-out:

Equivalent of fan-in with outgoing edges.

Can be computed at a method / function level, or at an entire class level

For a class, sum of number of incoming / outgoing edges for all methods

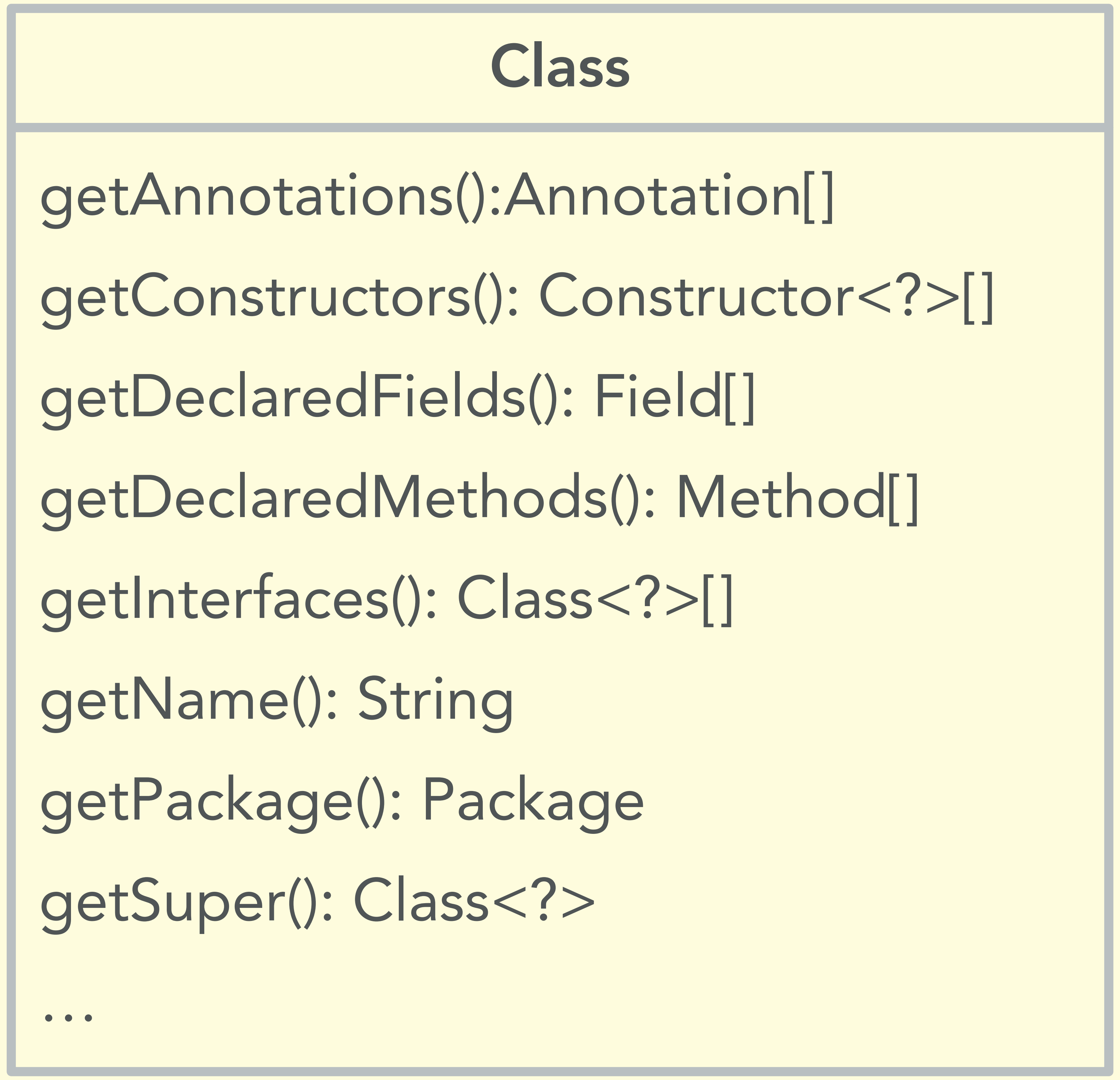
Call must come from (or go to) a different class.

Tools for Static Analysis

Reflection

The ability of a program to inspect itself at runtime.

Very useful for analysis tasks, e.g. reverse-engineering.



Class

getAnnotations():Annotation[]

getConstructors(): Constructor<?>[]

getDeclaredFields(): Field[]

getDeclaredMethods(): Method[]

getInterfaces(): Class<?>[]

getName(): String

getPackage(): Package

getSuper(): Class<?>

…

Some reflection methods in java.lang.Class

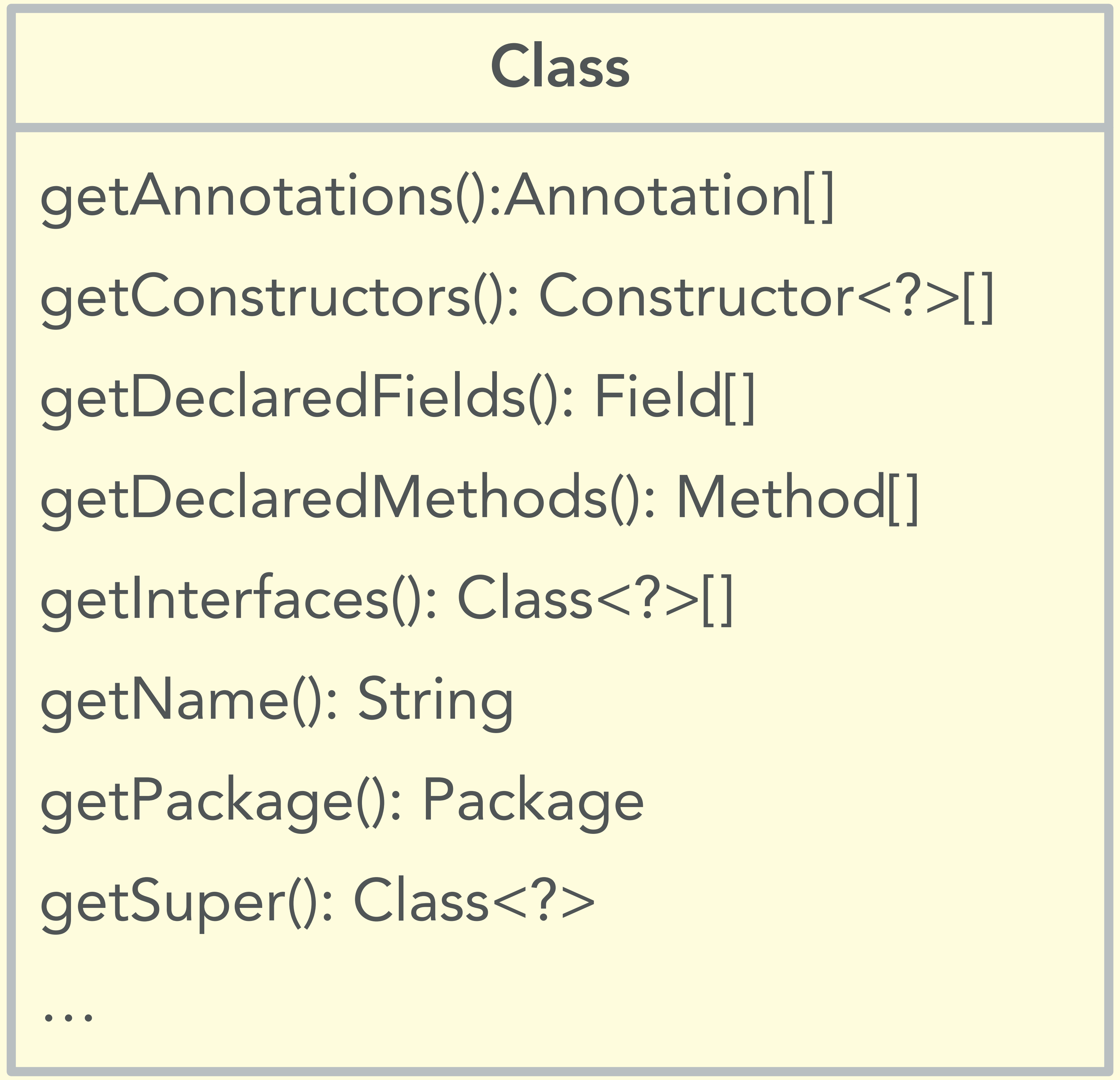
Reflection

The ability of a program to inspect itself at runtime.

Very useful for analysis tasks, e.g. reverse-engineering.

A feature that is incorporated into interpreters / VMs.

Java, C# (and other .NET languages), Go, Julia, Lisp, Perl, Python, R, Ruby, **Smalltalk**, …



Class

getAnnotations():Annotation[]
getConstructors(): Constructor<?>[]
getDeclaredFields(): Field[]
getDeclaredMethods(): Method[]
getInterfaces(): Class<?>[]
getName(): String
getPackage(): Package
getSuper(): Class<?>
…

Some reflection methods in java.lang.Class

Reflection

The ability of a program to inspect itself at runtime.

Very useful for analysis tasks, e.g. reverse-engineering.

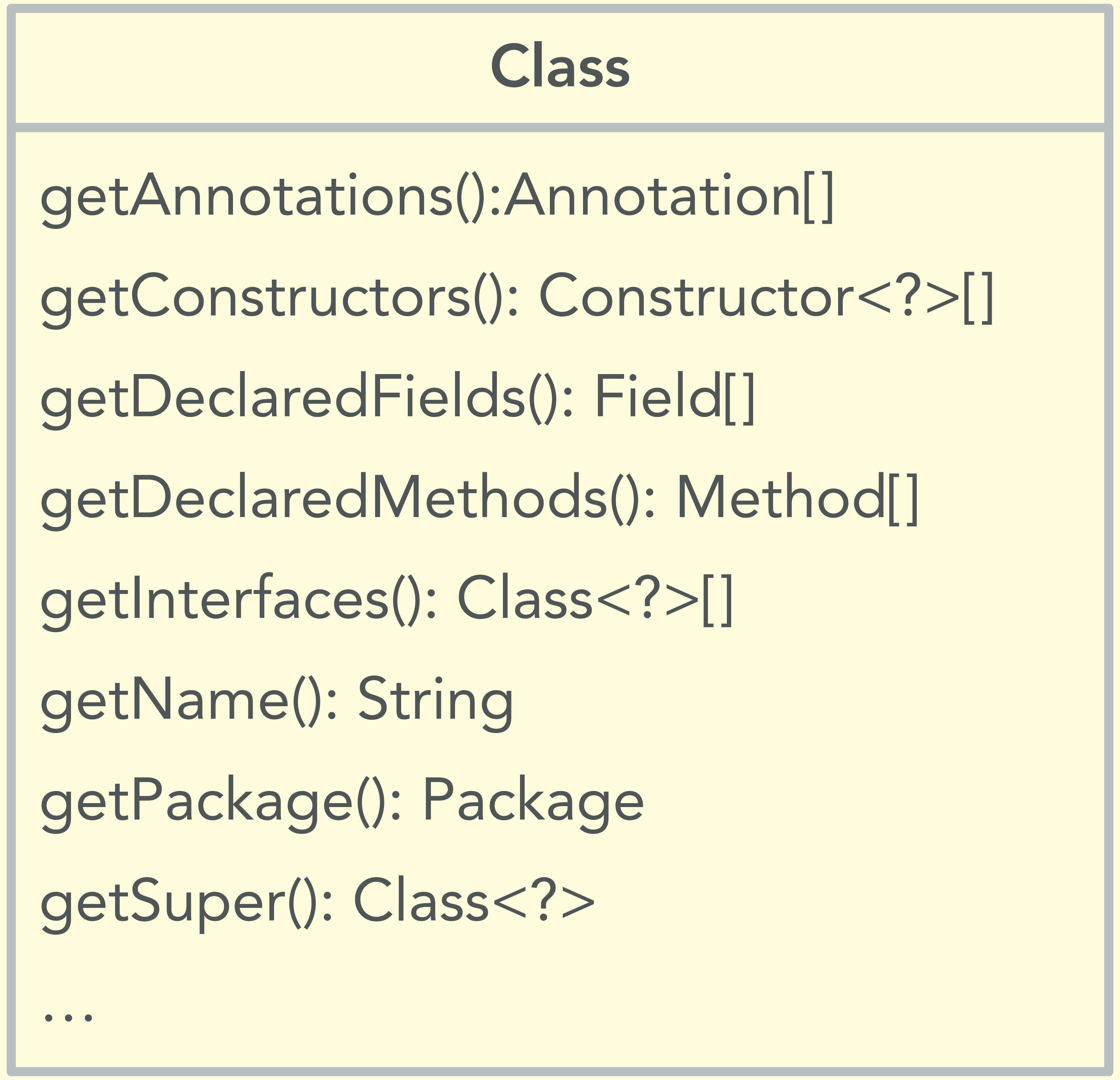
A feature that is incorporated into interpreters / VMs.

Java, C# (and other .NET languages), Go, Julia, Lisp, Perl, Python, R, Ruby, **Smalltalk**, …

Particularly useful for analysing class structure.

E.g. Reverse-engineering a class diagram.

Less useful for statement-level code analysis.



Class

getAnnotations():Annotation[]
getConstructors(): Constructor<?>[]
getDeclaredFields(): Field[]
getDeclaredMethods(): Method[]
getInterfaces(): Class<?>[]
getName(): String
getPackage(): Package
getSuper(): Class<?>
…

Some reflection methods in java.lang.Class

Reverse-engineering a class diagram with Reflection

Iterate through classes in the system and for each class X:

Create a “class” node corresponding to X.

Load X via reflection.

For each type of relationship from X to some other class Y:

- Create a “class” node for Y if it doesn’t exist already.
- Create an edge X → Y (using the appropriate edge notation for the relationship type).

Decompilation

Commonly done with 3rd party tools.

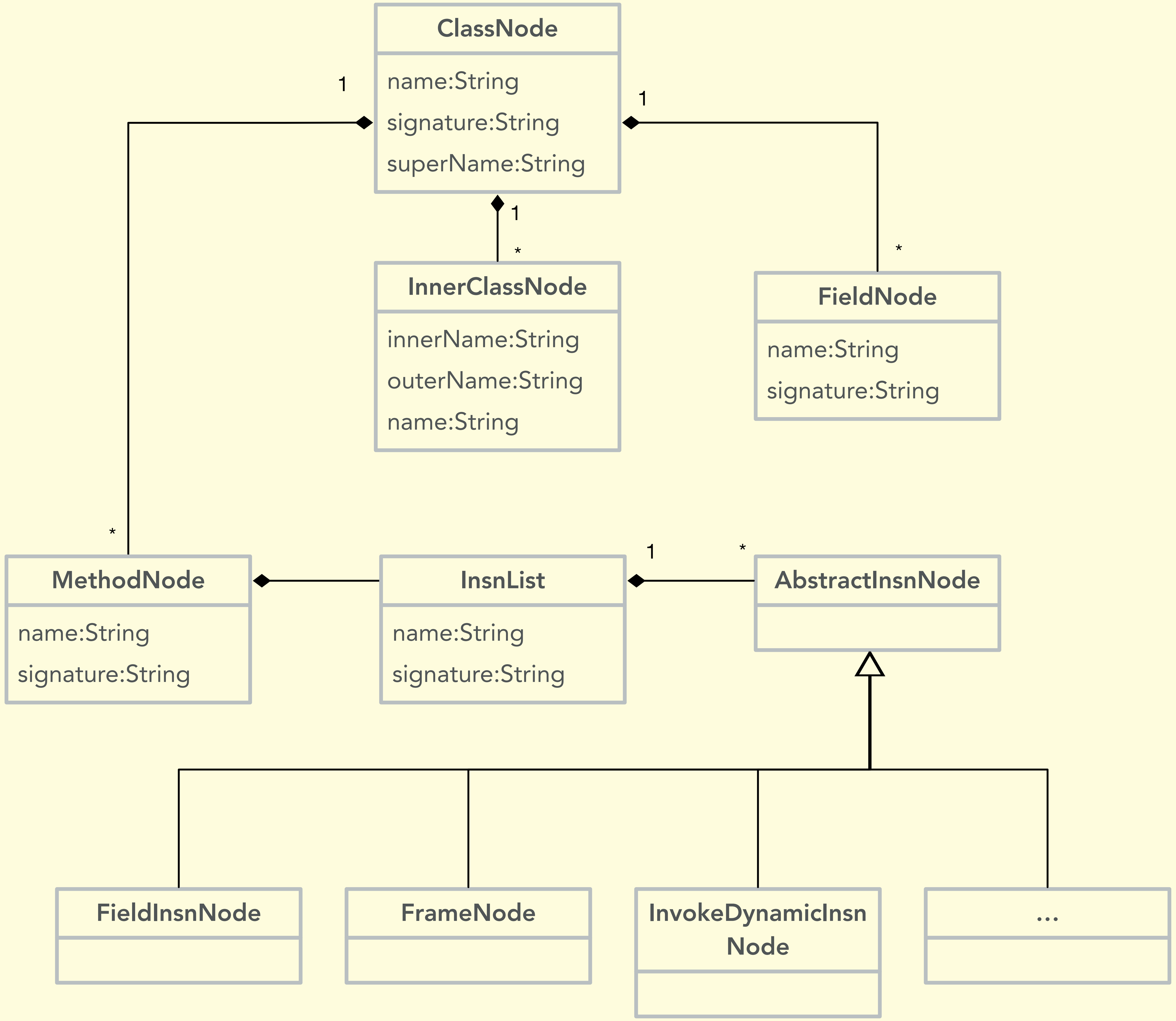
E.g. For Java: ASM, Apache BCEL, Soot, …

Compiled code is parsed.

Down to instruction-level.

Can often be inspected and manipulated.

Can often be changed and written to new classes.



Static analysis is conservative

Returns *everything* by default.

- Every single class or method in a system.
- Every single *potential* call (even calls that are infeasible in practice).

How useful is a class diagram with >700 classes?

Key strategies:

- For visual outputs (e.g. class diagrams) - **focus** on specific packages / classes.
- For non-visual outputs (e.g. call graphs) - summarise data into key metrics.

Key take-aways

Intra-procedural analysis is concerned with the analysis of individual functions.

Dominance, data-flow, control dependence, slicing.

Inter-procedural analysis is concerned dependencies between functions.

Call graphs.

Two useful technologies: Reflection and Bytecode analysis.

Reflection is useful for structural analysis - e.g. class diagrams.

Byte code analysis is more useful for detailed analysis - e.g. call graphs.

Overarching challenge: Information overload - static analysis is conservative.