Applied Static Analysis

Why Static Analysis?

Software Technology Group
Department of Computer Science
Technische Universität Darmstadt

Dr. Michael Eichberg

Questions that we may ask about a program:

- Will the variable **x** always contain the same value?
- Which objects can variable **x** points to?
- What is a lower/upper bound on the value of the integer variable **x**?
- Which methods are called by a method invocation?
- How much memory is required to execute the program?
- Will it throw a `NullPointerException`?
- Will it leak sensitive data? Will data from component `a` flow to component `b`?

Buggy C-Program:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, const char * argv[]) {
    char *p, *q;
    p = NULL;
    printf("%s",p);
    q = (char *)malloc(100);
    p = q;
    free(q);
    *p = 'x';
    free(p);
    p = (char *)malloc(100);
    q = (char *)malloc(100);
    q = p;
    strcat(p,q);
}
```

What is Static Analysis?

A static analysis of a program is a *sound*, *finite*, and *approximate* calculation of the program's execution semantics which helps us to solve practical problems.

Purposes of Code Analyses

- Finding code smells.
- Quality assessments.
- Improving the quality of the code.
- Support debugging of code.
- Optimizing the code.

Not every analysis which does not execute the code is a *static analysis*!

Finding Programming Bugs

```
class X {
    private long scale;
    X(long scale) { this.scale = scale; }
    void adapt(BigDecimal d){
        d.setScale(this.scale);
    }
}
```

There is typically more than one way to find certain bugs and not all require (sophisticated) static analyses!

Finding Bugs Using Bug Patterns

```
import org.opalj.br._
import org.opalj.br.instructions.{INVOKEVIRTUAL,POP}
val p = analyses.Project(org.opalj.bytecode.JRELibraryFolder) // <= analyze the JRE
p.allMethodsWithBody.foreach{m =>
  m.body.get.collectPair{
    case (
        i @ INVOKEVIRTUAL(ObjectType("java/math/BigDecimal"),"setScale",_),
        POP
    ) => i
  }
  .foreach(i => println(m.toJava(i.toString)))
}
```

Real instance in Java 8u191:
com.sun.rowset.CachedRowSetImpl.updateObject(int,Object,int)

Finding Bugs Using Bug Patterns (Assessment)

- Advantages:
  - very fast and scale well to very large programs
  - (usually) simple to implement
- Disadvantages:
  - typically highly specialized to specific language constructs and APIs
  - requires some understanding how the issue typically manifests itself in the (binary) code
  - small variations in the code may escape the anaylsis
  - to cover a broader range of similar issues significant effort is necessary

Finding Bugs Using Machine Learning

Finding Bugs Using Machine Learning

Guess the problem of the following JavaScript code snippet:

```
function setPoint(x, y) { ... } // <= given
```

```
var x_dim = 23;
var y_dim = 5;
setPoint(y_dim, x_dim);
```

Finding Bugs Using Machine Learning (Assessment)

- Finds bugs that are practically impossible to find using other approaches; hence, often complementary to classic static analyses and also bug pattern based analyses.
- Requires the analysis of a huge code base; it may be hard to find enough code examples for less frequently used APIs.

Finding Bugs by Mining Usage Patterns (Idea)

Is the following code buggy?

```
Iterator<?> it = ...
it.next();
while (it.hasNext()) {
      it.next();
      ...
}
```

Finding Bugs Using Generic Static Code Analysis

```
class com.sun.imageio.plugins.png.PNGMetadata{
    void mergeStandardTree(org.w3c.dom.Node) {
        [...]
        if (maxBits > 4 || maxBits < 8) {
            maxBits = 8;
        }
        if (maxBits > 8) {
            maxBits = 16;
        }
        [...]
    }
}
```

Finding Bugs Using Generic Static Code Analysis

```
class sun.font.StandardGlyphVector {
    private int[] glyphs; // always
    public int getGlyphCharIndex(int ix) {
        if (ix < 0 && ix >= glyphs.length) {
            throw new IndexOutOfBoundsException("" + ix);
        }
    }
}
```

Finding Bugs Using Generic Static Code Analysis

```
class sun.tracing.MultiplexProviderFactory {
    public void uncheckedTrigger(Object[] args) {
        […]
        Method m = Probe.class.getMethod(
            "trigger",
            Class.forName("[java.lang.Object")
        );
        m.invoke(p, args);
    }
}
```

Finding Bugs Using Generic Static Code Analysis

```
package com.sun.corba.se.impl.naming.pcosnaming;
class NamingContextImpl {
    public static String nameToString(NameComponent[] name)
        [...]
        if (name != null || name.length > 0) {
            [...]
        }
        […]
    }
}
```

Finding Bugs Using Static Code Analysis ?

```
private boolean ...isConsistent(
        String alg,
        String exemptionMechanism,
        Hashtable<String, Vector<String>> processedPermissions) {
    String thisExemptionMechanism =
        exemptionMechanism == null ? "none" : exemptionMechanism;
    if (processedPermissions == null) {
        processedPermissions = new Hashtable<String, Vector<String>>();
        Vector<String> exemptionMechanisms = new Vector<>(1);
        exemptionMechanisms.addElement(thisExemptionMechanism);
        processedPermissions.put(alg, exemptionMechanisms);
        return true;
    }
    […]
}
```

Finding Bugs Using (Highly) Specialized Static Analyses

Do you see the security issue?

```
Cipher c = Cipher.getInstance("DES/CBC/PKCS5PADDING")
```

(True|False) (Positives|Negatives)

- a true positive is the correct finding (of something relevant)
- a false positive is a finding that is incorrect (i.e., which can't be observed at runtime)
- a true negative is the correct finding of no issue.
- a false negative refers to those issues that are not reported.

Unguarded Access - True Positive ?

```
void printIt(String args[]) {
    if (args != null) {
        System.out.println("number of elements: " + args.length);
    }
    for (String arg : args) {
        System.out.println(arg);
    }
}
```

Implicitly Guarded Access

```
void printReverse(String args[]) {
    int argscount = 0;
    if (args != null) {
        argscount = args.length;
    }
    for (int i = argscount - 1; i >= 0; i--) {
        System.out.println(args[i]);
    }
}
```

Irrelevant True Positives

Let's assume that the following function is only called with **`non-null`** parameters.

```
private boolean isSmallEnough(Object i) {
    assert(i != null);
    Object o = " "+i;
    return o.length < 10;
}
```

Complex True Positives

The cast in line 5 will fail:

```
GeneralPath result = new GeneralPath(GeneralPath.WIND_NON_ZERO);
[…]
if (dx != 0 || dy != 0) {
    AffineTransform tx = AffineTransform.getTranslateInstance(dx, dy);
    result = (GeneralPath)tx.createTransformedShape(result);
}
```

Complex True Positives - Assessment

The sad reality:

> […] the general trend holds; a not-understood bug report is commonly labeled a false positive, rather than spurring the programmer to delve deeper. The result? We have completely abandoned some analyses that might generate difficult to understand reports. [^FindBugsInTheRealWorld]

Soundiness

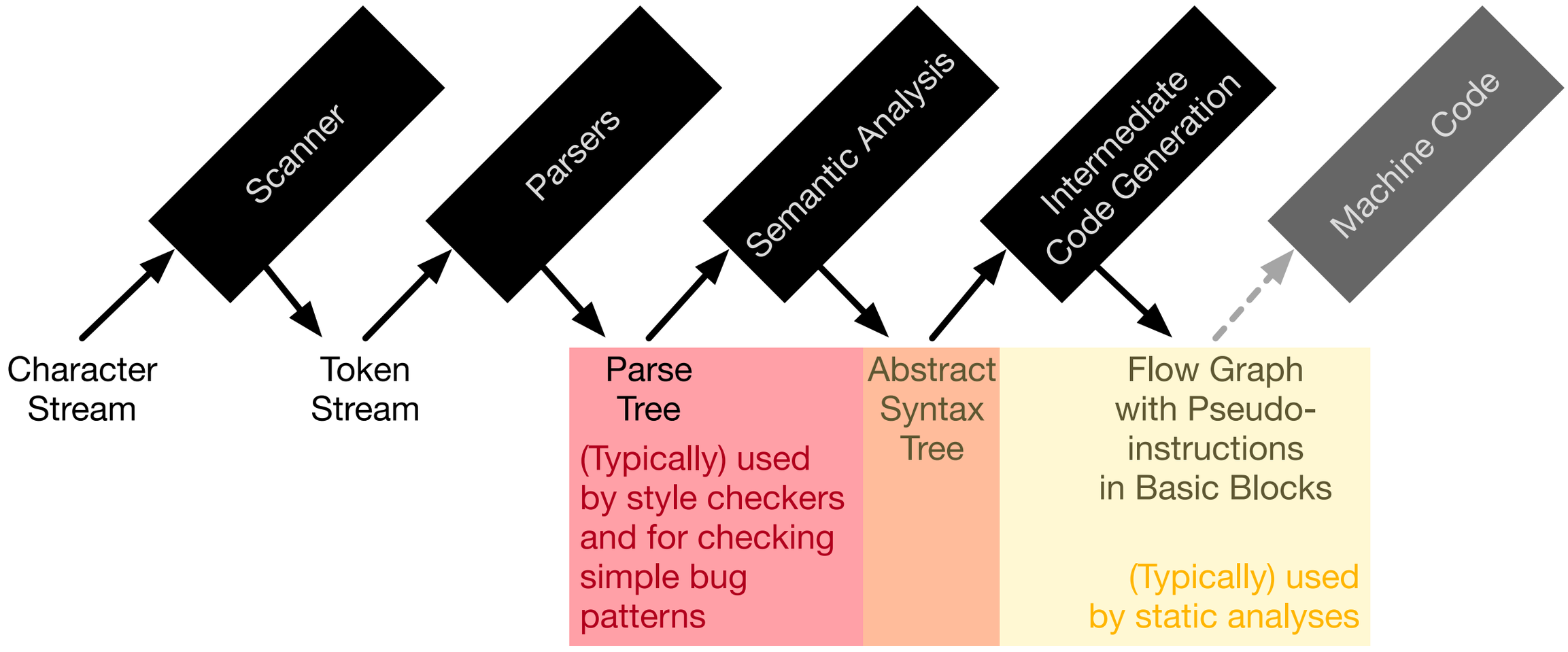
[…] in practice, soundness is commonly eschewed: we [the authors] are not aware of a single realistic whole-programa analysis tool […] that does not purposely make unsound choices.
[…]
Soundness is not even necessary for most modern analysis applications, however, as many clients can tolerate unsoundness. [^Soundiness]

Soundiness - Java

Common features that are often not soundly handled in Java:

1. Intents (in Android Programs)
2. Reflection (*often mentioned in research papers*)
3. Native methods (*often mentioned in research papers*)
4. Dynamic Class Loading / Class Loaders (*sometimes mentioned in research papers*)
5. (De)Serialization (*often not considered at all*)
6. Special System Hooks (e.g., `shutdownHooks`) (*often not considered at all*)
7. "Newer" language features

The Relation between Compilers and Static Analyses



The Relation between Compilers and Static Analyses

Source Code:

```
i = j + 1;
```

Tokens:

```
Ident(i) WS Assign WS Ident(j) WS Operator(+) WS Const(1)
Semicolon
```

AST with (type) annotations:

```
AssignmentStatement(
    target     = Var(name=i,type=Int),
    expression = AddExpression(
                     type  = Int,
                     left  = Var(name=j,type=Int),
                     right = Const(1)))
```

(Classical) Compiler Optimizations

- common subexpression elimination
- constant folding and propagation
- loop fusion
- loop invariant code motion
- code-block reordering
- dead-code elimination
- ...

often rely on the identification of the so-called *control dependencies* which are computed using control-flow graphs (CFGs). Though the optimizations as such are already relevant (e.g., to help deobfuscate code), the underlying techniques, such as, loop identification or live variable analyses are also relevant for static analyses on their own.

(Control-)Flow Graph (CFG)

Given a labeled, directed control-flow graph $\boldsymbol{G = (N, E, n_o)}$ which consists of the finite set $\boldsymbol{N}$ of the statements of the program and the set of edges $\boldsymbol{E}$ which represent the control flow between statements.

- $\boldsymbol{N}$ is partitioned into two subsets:
  - $\boldsymbol{N^S}$ – *statement nodes* with at most one successor and
  - $\boldsymbol{N^P}$ – *branch nodes* with at least two *distinct* successors
- $\boldsymbol{N^E \subseteq N^S}$ denotes the nodes in $\boldsymbol{N^S}$ that have no successors (the exit nodes)
- the start node $\boldsymbol{n_0}$ has no incoming edges and all nodes in $\boldsymbol{N}$ are reachable.
- if $\boldsymbol{N^E}$ contains only one element and this element is reachable from all other nodes of G then G satisfies the *unique end node property*

CFG - Nodes

```
public static long sumOfInts(int upTo) {
    long s = 1l;            // stmt node (assign)
    int i = 2;              // stmt node (assign)
    while (i < upTo) {      // branch node (if)
        s += i;             // stmt node (add followed by assign)
        i++;                // stmt node (inc)
    } // goto loop start - stmt node; control-transfer instruction
    return s;               // stmt node
}
```

CFGs - Paths

A CFG path π from n_i to n_k is a sequence of nodes $n_i, n_{i+1}, \ldots, n_k$ such that for each consecutive pair (n_i, n_j) in the path an edge from n_i to n_j exists.

A path is *nontrivial* if it contains at least two nodes.

A path is *maximal* if it is infinite or terminates in an end node.

Dominance

A Node n dominates node m in G (dom(n,m)) if *every* path from the start node n_0 to m passes through n (hence, dom is reflexive).

Dominance - Example

```
public static int abs(int i) {
    if(i == Integer.MIN_VALUE)
        throw new ArithmeticException();
    if(i < 0)
        i = -i;
    return i;
}
```

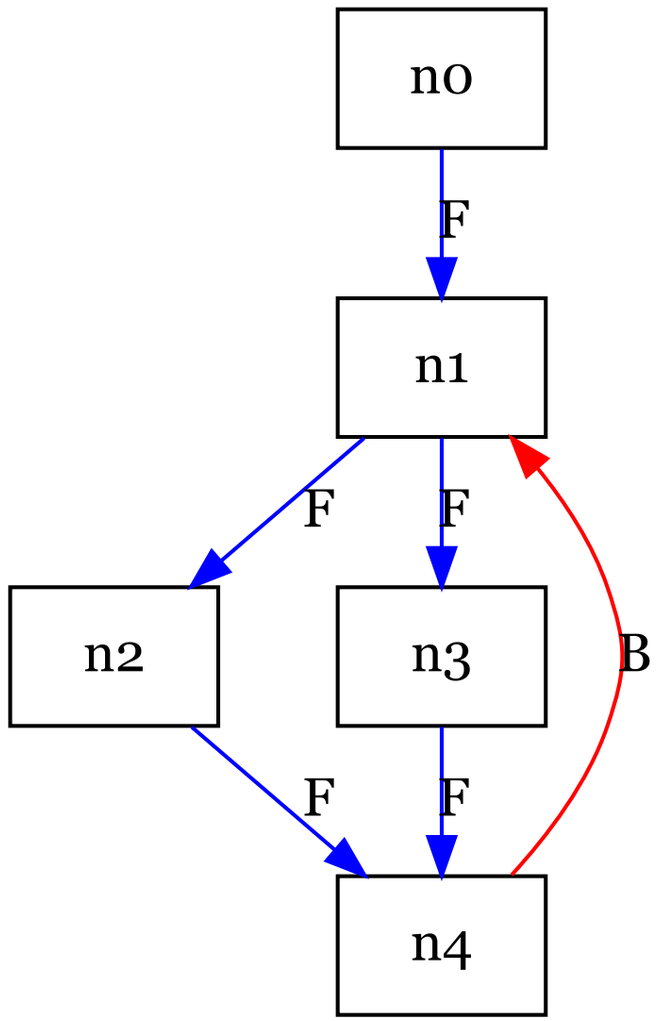
(Ir)Reducible CFGs

A CFG $\boldsymbol{G = (N, E, n0)}$ is reducible if ...

- E can be partitioned into disjoint sets $\boldsymbol{E_f}$ (forward edges) and $\boldsymbol{E_b}$ (backward edges)
- such that $\boldsymbol{(N, E_f)}$ forms a directed-acyclic graph (DAG) in which each node can be reached from the start node $\boldsymbol{n_0}$
- and for all edges $\boldsymbol{e \in E_b}$, the target of e dominates the source of e.

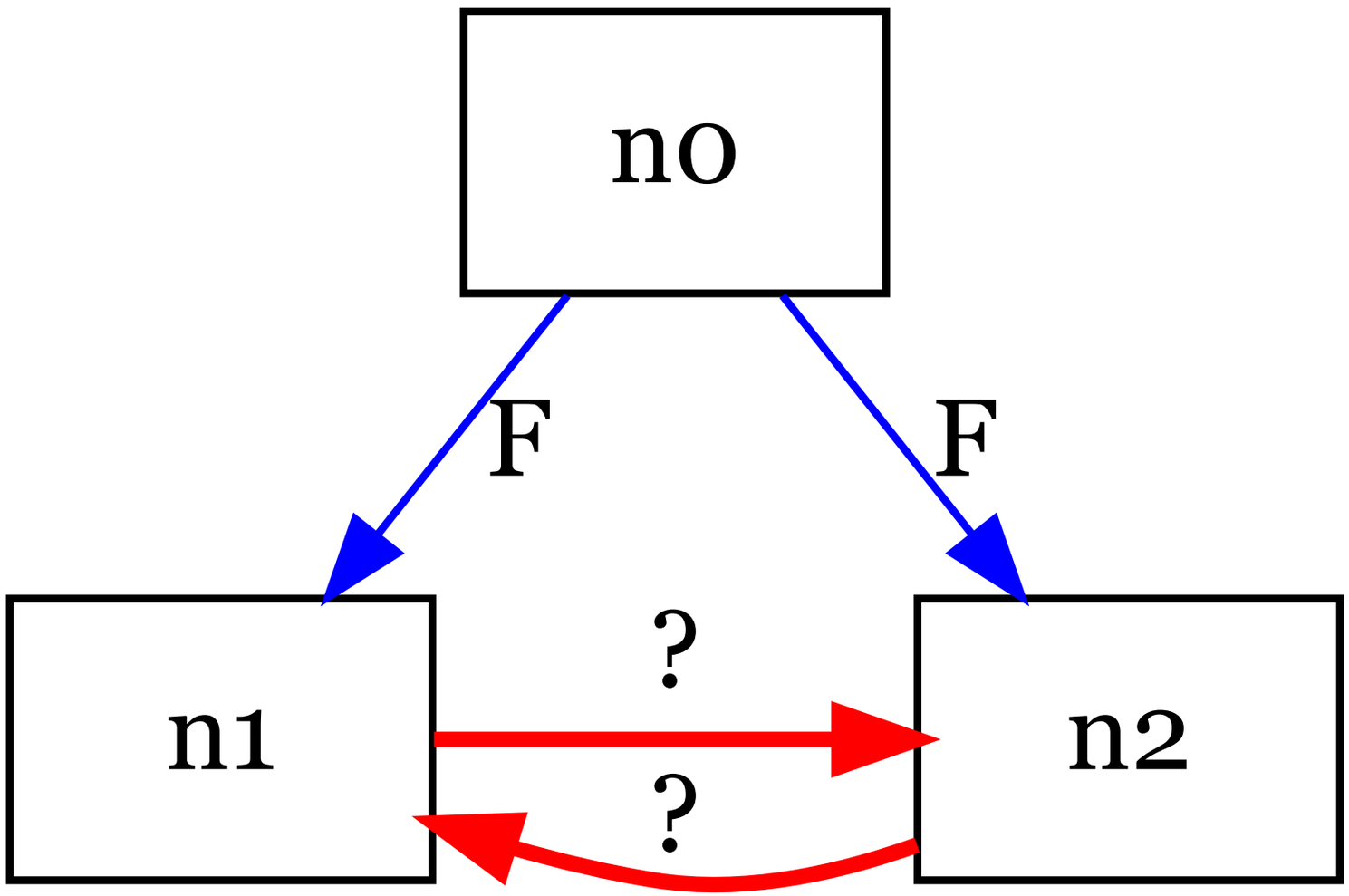
Reducible CFGs - Example

B = Backward Edge; F = Forward Edge



Irreducible CFGs - Example

B = Backward Edge; F = Forward Edge



CFGs based on *Basic Blocks*

In real-word CFGs the nodes are typically based on using basic blocks.

- A basic block is a maximal-length sequence of statements without jumps in and out (and no exceptions are thrown by intermediate instructions).
- A basic blocks based CFG (still) represents the control flow of a single method.

CFGs based on *Basic Blocks* - Example

```
public static int abs(int i) {
    if(i == Integer.MIN_VALUE)
        throw new ArithmeticException();
    if(i < 0)
        i = -i;
    return i;
}
```