

A tutorial at **CGO 2020** on

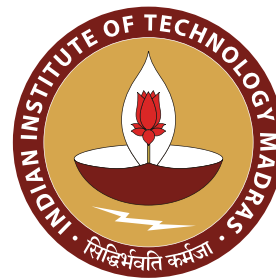
IMOP

IIT Madras OpenMP Compiler Framework

Aman Nougrihiya

V. Krishna Nandivada

PACE Lab
Dept of CS&E, IIT Madras



February 22nd, 2020

Webpage: bit.ly/imop-iitm

A World of Compiler Frameworks



Famous compiler frameworks, with diverse compilation goals^[1].

Wow, so many! These must be enough..

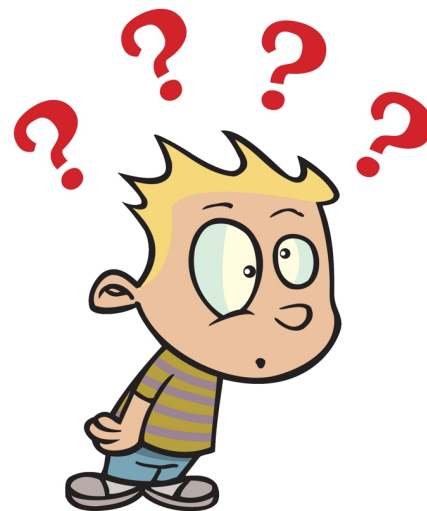
We don't need any New Compiler Framework!

[1] This figure has been created using Wordle.

Welcome to this CGO Tutorial on a *New Compiler Framework!*

IMOP

Okay, but... *WHY?!*



[2] Clipart taken from www.clipartstation.com

WHY?!

Do the current ones not suffice?

Major issues with existing compiler frameworks for OpenMP :

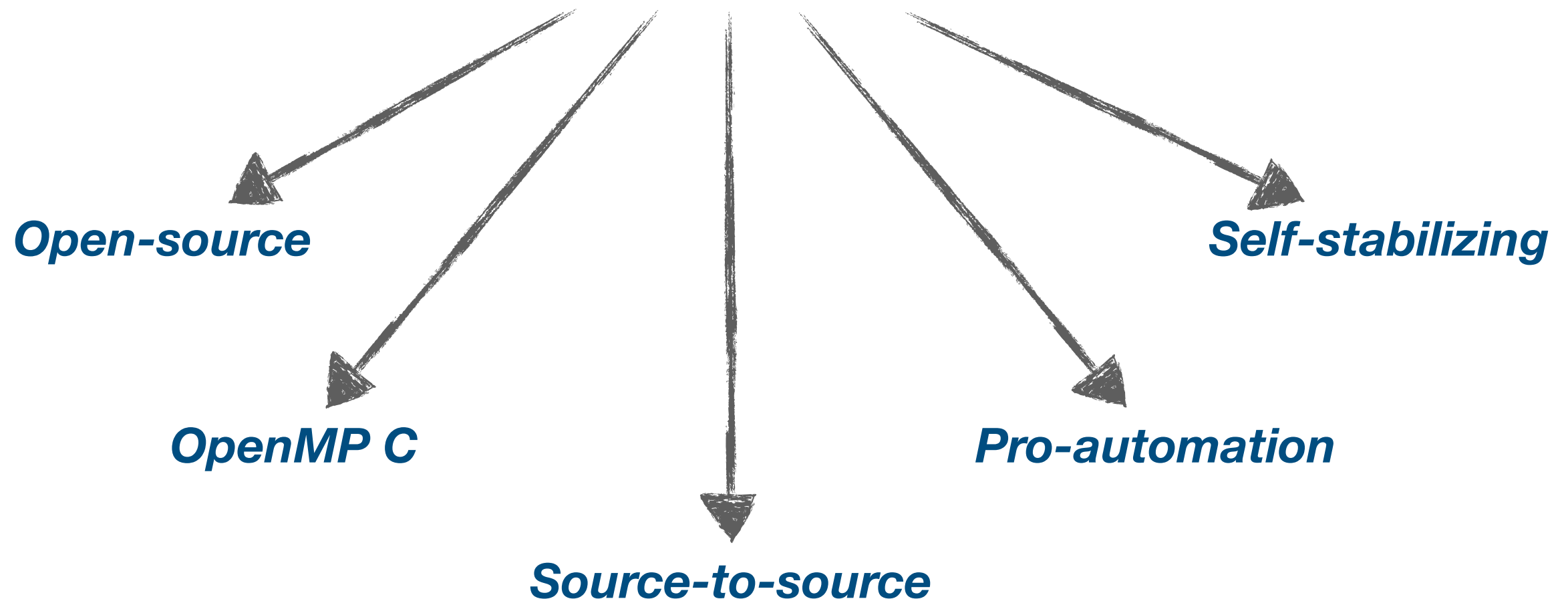
- Most of them (like GCC and LLVM) **work at low-level** of program representations.

OpenMP is expressed and better understood at higher-level representations.

- Originally **built for serial programs**, hence their implementations of analyses and transformations may yield **incorrect results under parallel semantics**, as
 - *static analyses do not assume multiple threads of execution, and*
 - *inter-task data flow is not modelled.*
- Further, **significant manual efforts** are required by the compiler writers to keep the compiler **stable** (or, *consistent*) in response to program changes.

To address these issues, we have developed

IMOP



Key Guiding Principle:

***Ease** the task of implementing various analyses and transformation tools for (OpenMP) C programs, by **automating** the tasks of a compiler writer.*

IMOP is licensed under the MIT License.

Tutorial Objectives

By the end of this tutorial, the audience shall be able to...

Appreciate **IMOP** as a **useful** framework **to quickly implement** their program analysis and optimization **prototypes**.

Tutorial Objectives

By the end of this tutorial, the audience shall be able to...

- Use the fundamental building blocks provided by IMOP for writing new analysis and optimization passes for (OpenMP) C programs.
- Understand (*some of*) the key design principles and implementation details of IMOP.
- Write
 - a loop-unrolling pass for while-loops in just about **8 lines of code**;
 - a code-instrumentation pass for adding write barriers for a variable in just **about 12 lines of code**;
 - an OpenMP optimization pass, in about **30 lines of code**, for simple removal of redundant OpenMP barriers; *and much more*.

Let us begin, then..

Hands-On Session #0

Testing the setup

We have already created ready-to-use virtual machines for your use during this tutorial.

- (A) Connect to the provided VM on cloud; test IMOP setup.
- (B) Run the project **MainProject** on the example program given in the folder `runner/cgo-eg/example.c`

```
$ cd bin
```

```
$ java cgo20.MainProject -f ../runner/cgo-eg/example.c
```

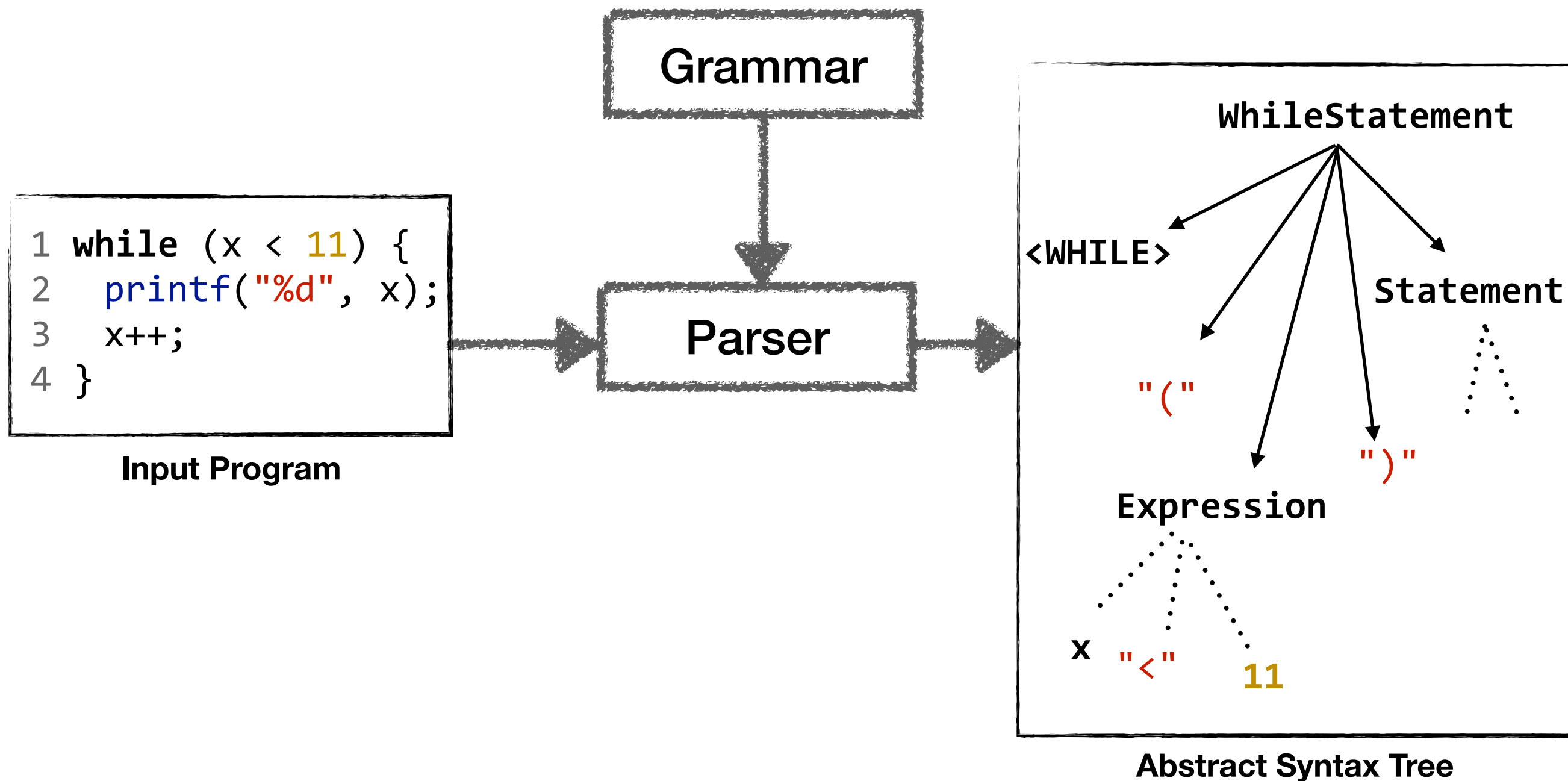
In this tutorial, we will learn how to implement this project (and some more), step-by-step.

Note: Detailed walkthrough is present in the provided handouts.

Abstract Syntax Tree

```

WhileStatement ::= <WHILE> "(" Expression ")" Statement
Statement ::= ExpressionStatement | CompoundStatement
              | IfStatement | . . .
  
```



Grammar of IMOP

```
WhileStatement ::= <WHILE> "(" Expression ")" Statement  
Statement ::= ExpressionStatement | CompoundStatement  
              | IfStatement | . . .
```

Grammar

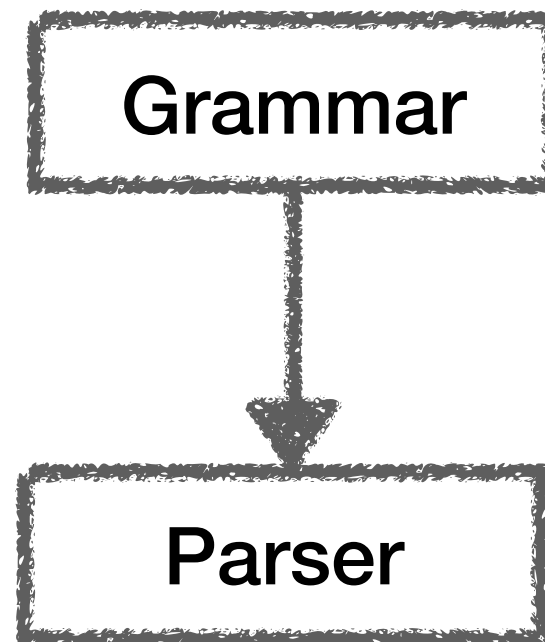
```
1 while (x < 11) {  
2   printf("%d", x);  
3   x++;  
4 }
```

Input Program

- IMOP accepts programs written in standard **C (ANSI)**, and (*almost all of*) **OpenMP 4.0**.
- Can handle all standard benchmarks—*NPB, SPECOMP, Sequoia*, etc.
- IMOP is itself written in **Java**.

Parser used in IMOP

```
WhileStatement ::= <WHILE> "(" Expression ")" Statement  
Statement ::= ExpressionStatement | CompoundStatement  
              | IfStatement | . . .
```



Parser of IMOP has been created using
JavaCC/JTB^[3]

[3] <http://compilers.cs.ucla.edu/jtb/>

Invoking the parser

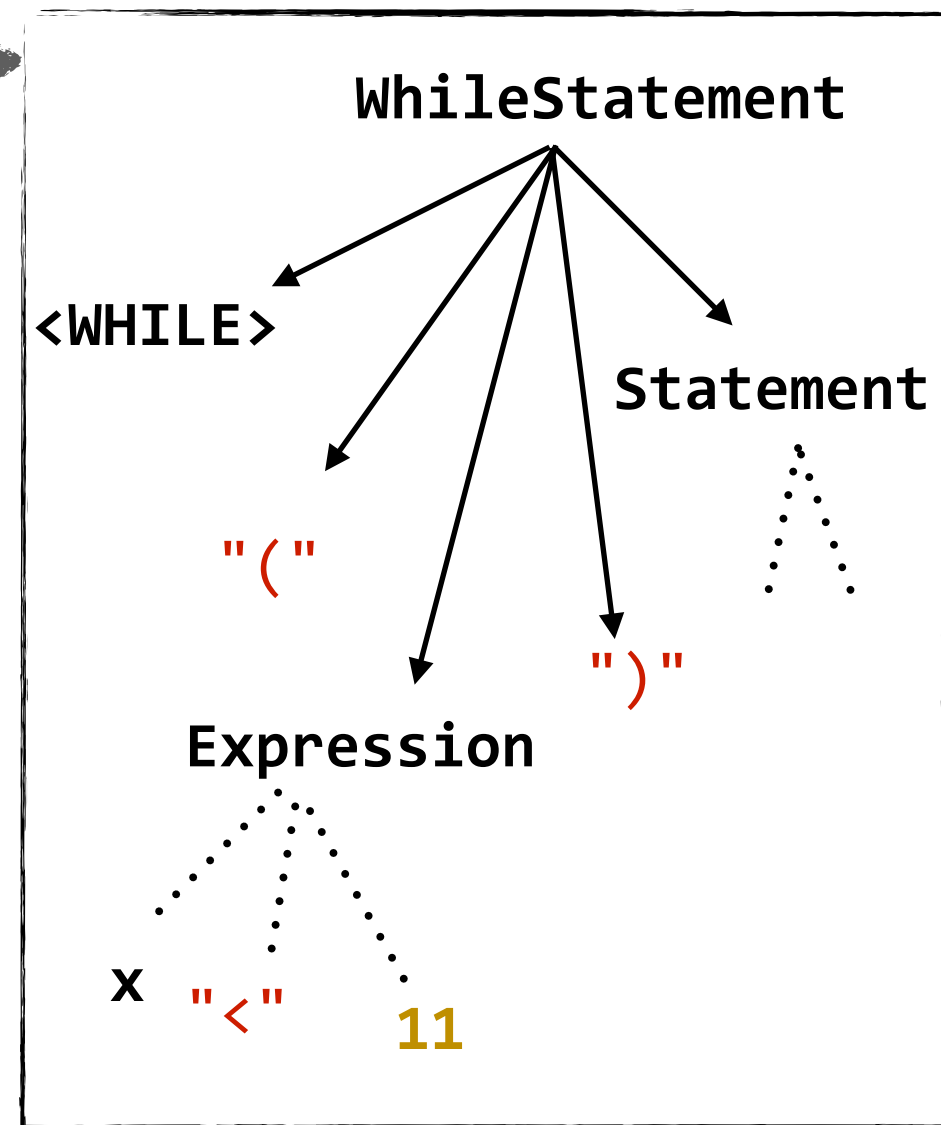
Program.**parseNormalizeInput**(args);

```

1 while (x < 11) {
2   printf("%d", x);
3   x++;
4 }

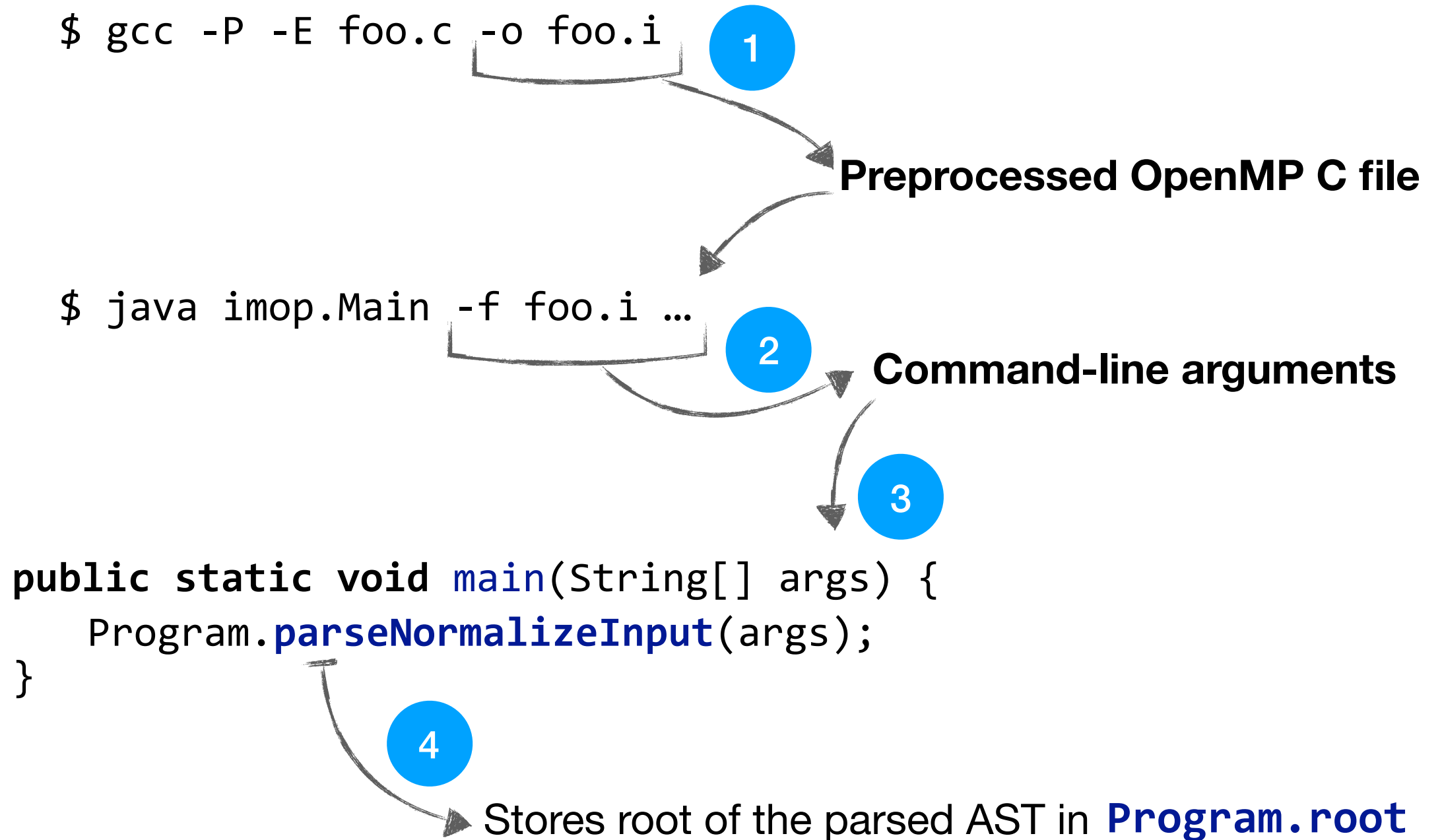
```

Input Program



Abstract Syntax Tree

Invoking the parser

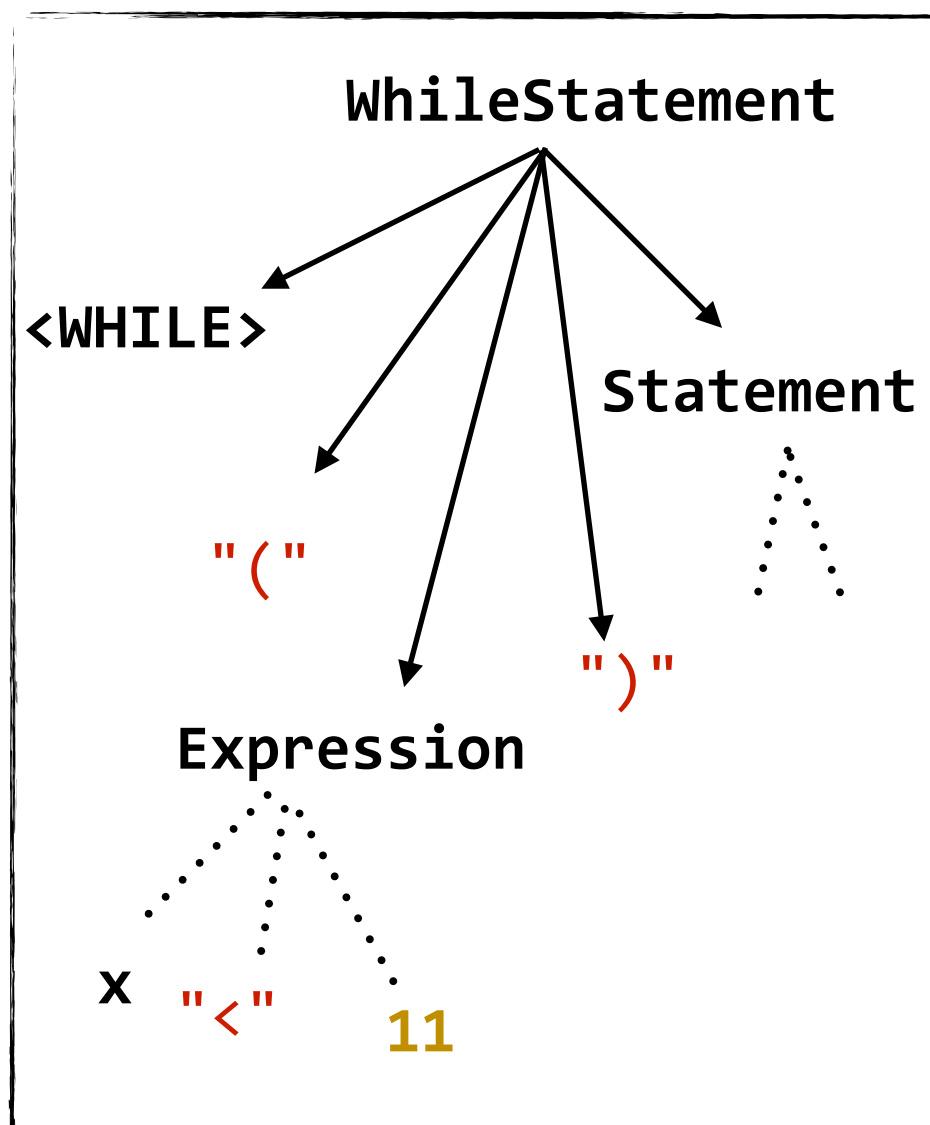


Print my program!


```

1 public static void main(String[] args) {
2     Program.parseNormalizeInput(args);
3
4     // Print the program to the terminal.
5     System.out.println(Program.getRoot());
6
7     // Print to a new file named foo-f1.i
8     DumpSnapshot.dumpRoot("f1");
9 }

```



Abstract Syntax Tree



```

1 while (x < 11) {
2     printf("%d", x);
3     x++;
4 }

```

Output Program

Information Objects

With each important AST node, we maintain a specific subclass of **NodeInfo**.

- Contains node-specific information and operations.
- Obtained using **getInfo()** invocations.
- Examples:
 - ▶ WhileStatementInfo::**unrollLoop(int)**
 - ▶ DeclarationInfo::**getInitializer()**

Querying the AST

- Simple depth-first traversals over the AST.
 - Too low-level; not recommended.
- Better alternative: Use higher-level query functions, such as:

| | |
|--|---|
| Get all functions of a program | <code>Program.getRoot().getInfo(). getAllFunctionDefinitions()</code> |
| Get all <i>if</i> -statements within a node <i>n</i> | <code>Misc.getInheritedEnclosee(n, IfStatement.class)</code> |
| Get statement with label <i>L</i> within a node <i>n</i> | <code>n.getInfo().getStatementWithLabel("L")</code> |

Example: Querying the AST

// Print all statements having the given label.

```
public static void demo1(String label) {
    for (FunctionDefinition func : Program.getRoot().getInfo().
        getAllFunctionDefinitions()) {
        Statement stmt = func.getInfo().getStatementWithLabel(label);
        if (stmt != null) System.out.println(stmt);
    }
    System.exit(0);
}
```

1 Iterate over all the function definitions.

2 Get statement with the given label.

3 Print the statement.

Suggestion:

Look into the other methods present in various subclasses of NodeInfo.

Hands-On Session #1

Working with the AST

- (A) Parse a sample program using IMOP.
- (B) From the generated AST, print the program to a file/terminal.

Note some simplifications performed by IMOP.

- (C) Find and print all those statements that have a given label.
- (D) Invoke loop unrolling on the given while-statement, and print.

Note: Detailed walkthrough is present in the provided handouts.

Control-Flow Graphs

- Control-flow graphs (CFGs) — approximations of how control would flow among executable nodes of a function in runtime.

Useful for performing static analyses.

- Unlike traditional frameworks, IMOP uses **nested** CFGs.
 - Resembles higher-level representation of (OpenMP) C programs.
 - Useful in modeling scope information (used by OpenMP clauses).
- Each CFG node is also an AST node.

Types of nodes in Nested CFGs

IMOP models two types of CFG nodes:

- ***Non-leaf nodes***, corresponding to nesting constructs of C and OpenMP, such as

`FunctionDefinition`, `WhileStatement`, and `CriticalConstruct`.

- ***Leaf nodes***, for other executable nodes in the program, such as,

`ExpressionStatement`, `GotoStatement`, and `BarrierDirective`.

Example CFG

```

1 int main() {
2     int x = 0;
3     while (x > 10) x++;
4 }

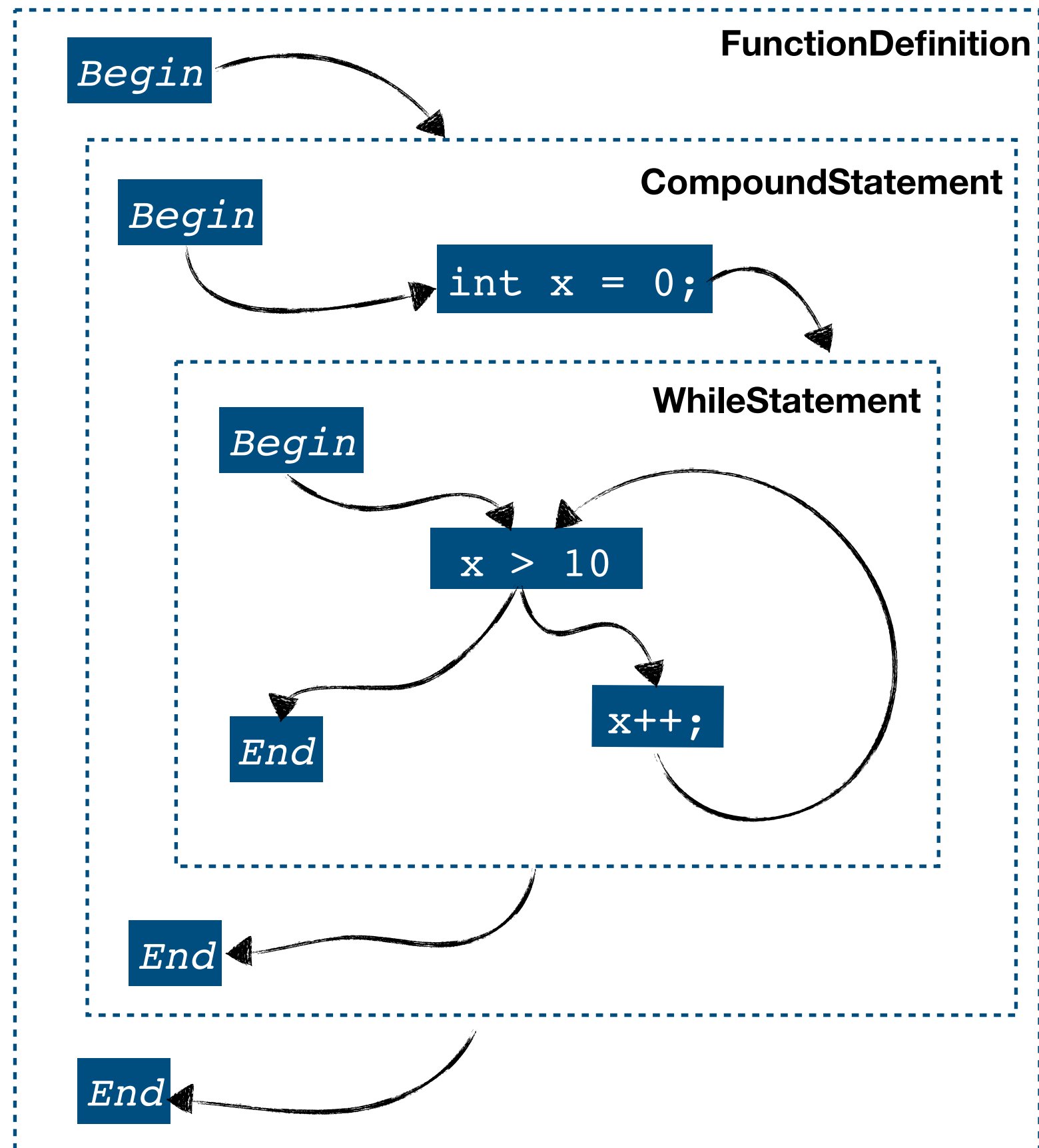
```

Non-leaf nodes

- Contain other CFG nodes.
- Entry/Exit points denoted by special nodes: *Begin*/*End*.
- Define control flow as per the semantics of OpenMP and C.

Leaf nodes

- Do not contain nested CFG nodes.



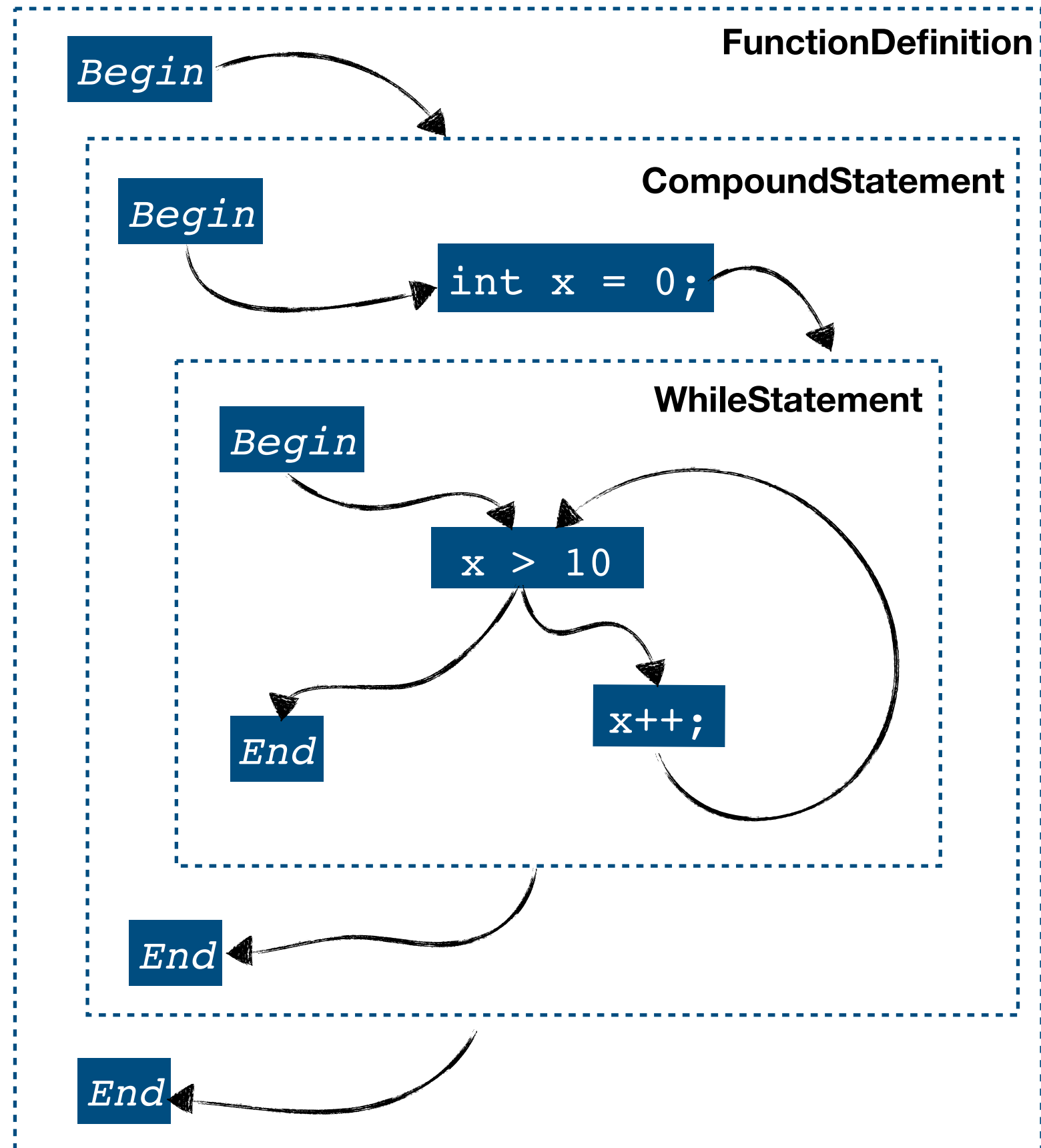
Example CFG

```

1 int main() {
2     int x = 0;
3     while (x > 10) x++;
4 }

```

- IMOP automatically creates CFG upon parsing.
- A DOT-file representation of the CFG is dumped at `output-dump/foonestedDotGraph.gv`



Traversals on CFG

Intra-procedural traversals

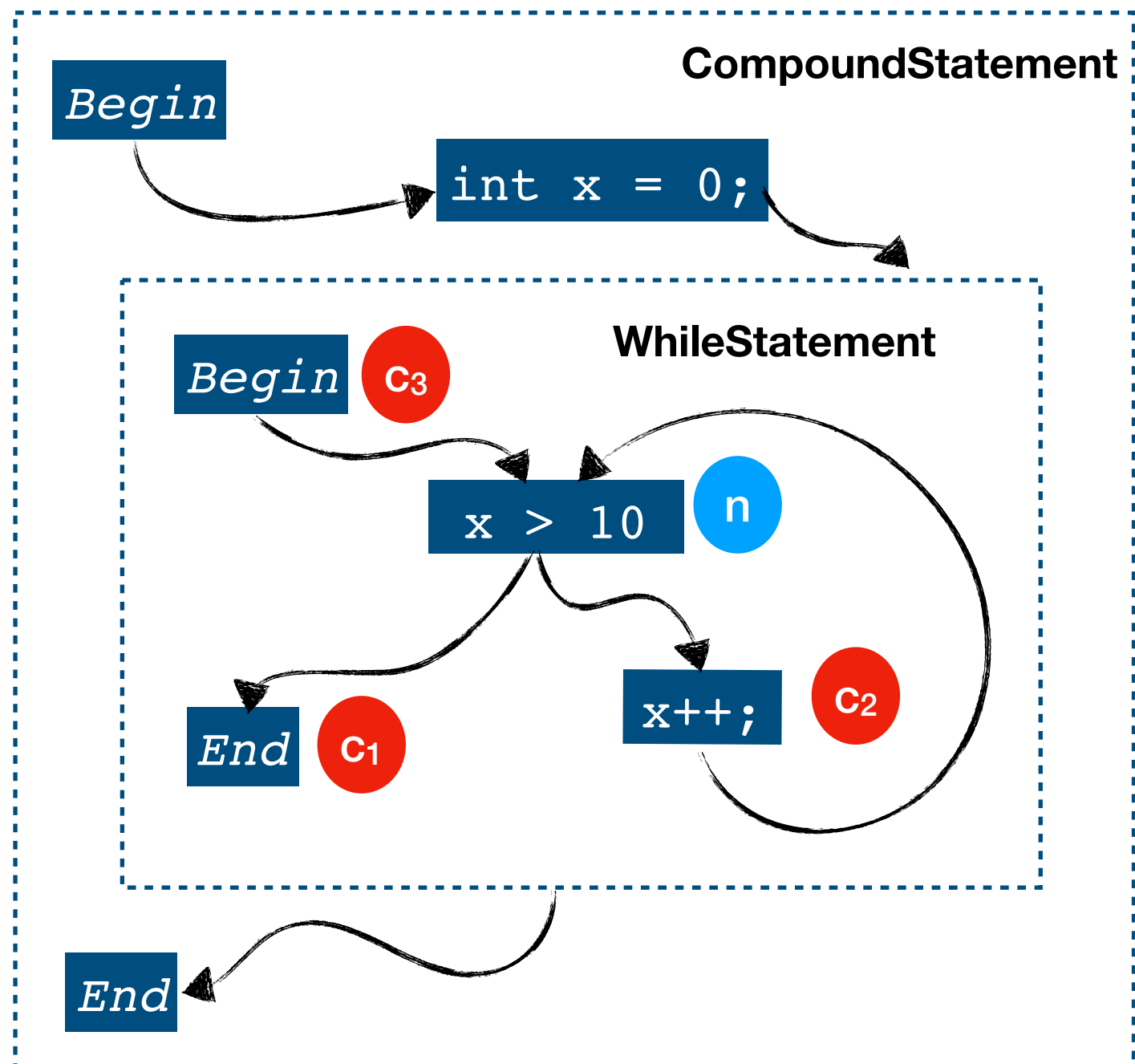
Successors of a node

`node.getInfo().getCFGInfo().getSuccessors()`



Predecessors of a node

`node.getInfo().getCFGInfo().getPredecessors()`



CFG Components and Links

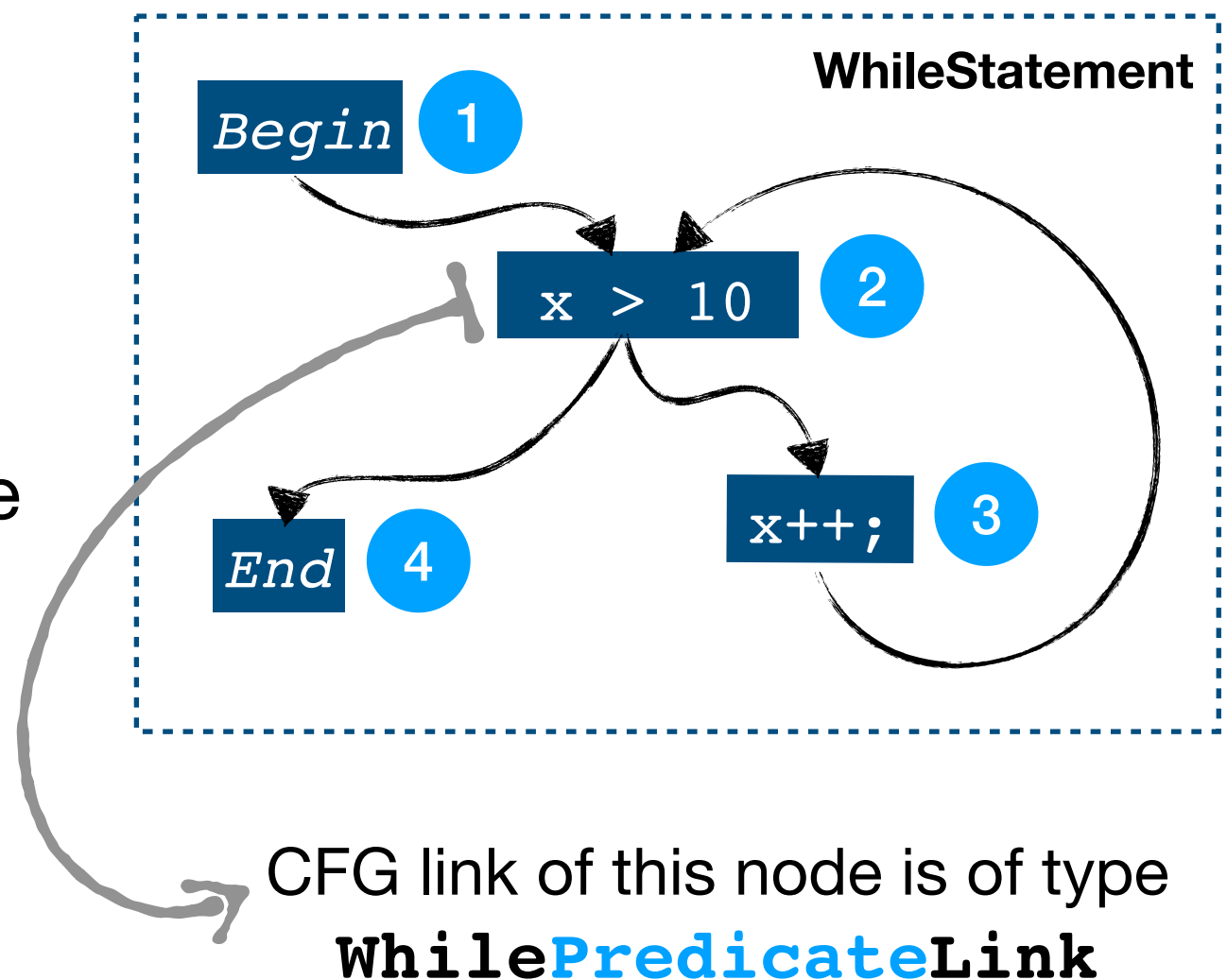
- Different parts of a non-leaf node, such as body and predicate of various constructs, are termed as its **CFG components**.

Note: Each CFG component is a CFG node.

- The nesting relation of a node with its enclosing non-leaf node is denoted with a **CFG Link** object.

Four CFG components of a WhileStatement

- 1 Begin node.
- 2 Predicate node.
- 3 Body node.
- 4 End node.



Querying CFG Components and Links

- CFG components of a non-leaf node can be obtained using pre-defined methods in its **CFGInfo** object.

| | |
|--------------------------------------|--|
| Get predicate of a while-statement | <code>whileStmt.getInfo().getCFGInfo().getPredicate()</code> |
| Get elements of a compound-statement | <code>compStmt.getInfo(). getCFGInfo().getElementList()</code> |
| Get false branch of an if-statement | <code>ifStmt.getInfo().getCFGInfo().getElseBody()</code> |

- CFG link corresponding to a node can be obtained by invoking **CFGLinkFinder.getCFGLinkFor(node)**.

Example: Querying the CFG Components

// Query CFG components

```
public static void demo2() {  
    for (WhileStatement whileStmt : Misc.getInheritedEnclosee(Program.getRoot(),  
        WhileStatement.class)) {  
        System.out.println(whileStmt.getInfo().getCFGInfo().getPredicate());  
    }  
  
    for (IfStatement ifStmt : Misc.getInheritedEnclosee(Program.getRoot(),  
        IfStatement.class)) {  
        if (!ifStmt.getInfo().getCFGInfo().hasElseBody()) {  
            System.out.println(ifStmt);  
        }  
    }  
}
```

Print predicate of a while-statement.

Check whether an else-branch exists for an if-statement.

Querying nested CFG nodes

- To obtain the set of all CFG nodes, and not just the immediate components, that are present lexically within a given node.

```
node.getInfo().getCFGInfo().getLexicalCFGContents()
```

- The above set, but only with leaf nodes in it:

```
node.getInfo().getCFGInfo().getLexicalCFGLeafContents()
```

- To obtain the set of all leaf CFG nodes that may be present either lexically within a given node, or in any of the functions reachable from any call-sites in the given node.

```
node.getInfo().getCFGInfo().getIntraTaskCFGLeafContents()
```

Hands-On Session #2

Working with CFG Components and Traversals

- (A) Print predicate and body (separately) of each while-statement in the program.
- (B) Print *successors* of those if-statements which do not have an else-body.

Note: Detailed walkthrough is present in the provided handouts.

Call Graphs

Call Statements

- During parsing, IMOP simplifies each call-site to one of the two forms:

foo(s_1, s_2, \dots, s_n);

$x =$ **foo**(s_1, s_2, \dots, s_n);

where,

foo is a function designator,

x is a temporary, and


s_1, s_2, \dots, s_n are compile-time constants or temporaries.

- The resulting call-site is a non-leaf CFG node of type **CallStatement**.

Call Graphs

Simplification of calls

```
i[3] = fptr[3](a * 10,  
              bar (2, p[3]));
```



```
fp = fptr[3];  
t1 = p[3];  
t2 = bar (2, t1);  
t3 = a * 10;  
t4 = fp(t3, t2);  
i[3] = t4;
```

- Such simplifications ease representations of call-graphs and data-flow analyses.

Call Graphs

Structure of a CallStatement

There are two components in a CallStatement:

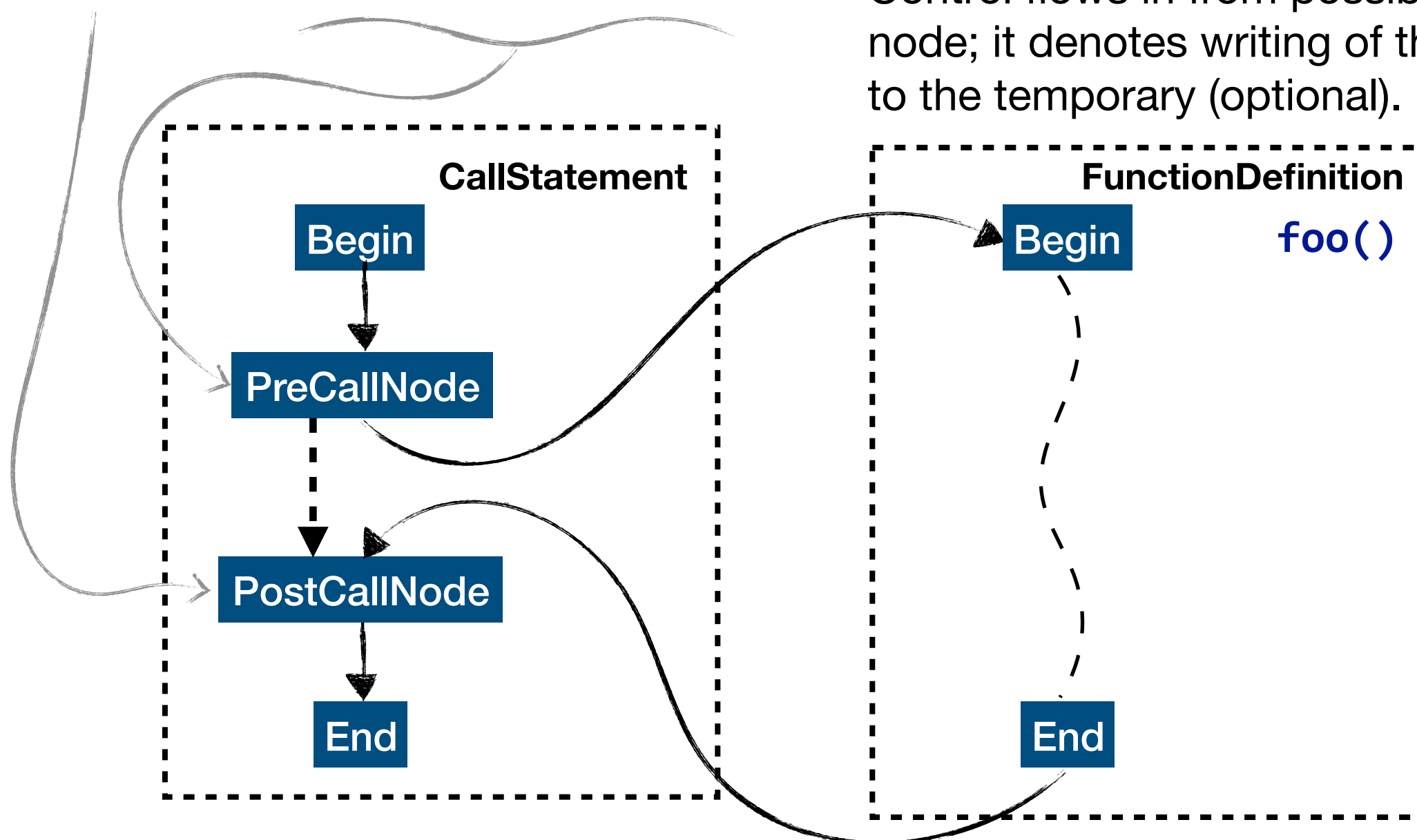
- **PreCallNode**

Denotes argument reads; after this node, control flows to possible target functions.

- **PostCallNode**

Control flows in from possible targets into this node; it denotes writing of the returned value to the temporary (optional).

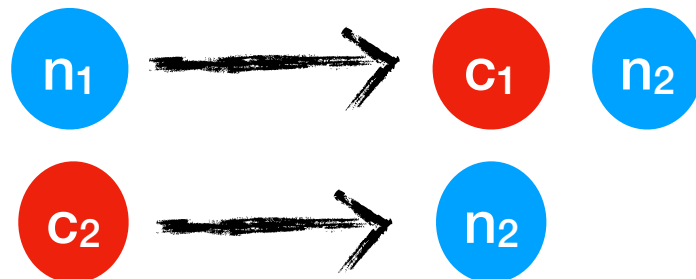
$x = \text{foo}(s_1, s_2, \dots, s_n);$



Traversals on Call Graphs

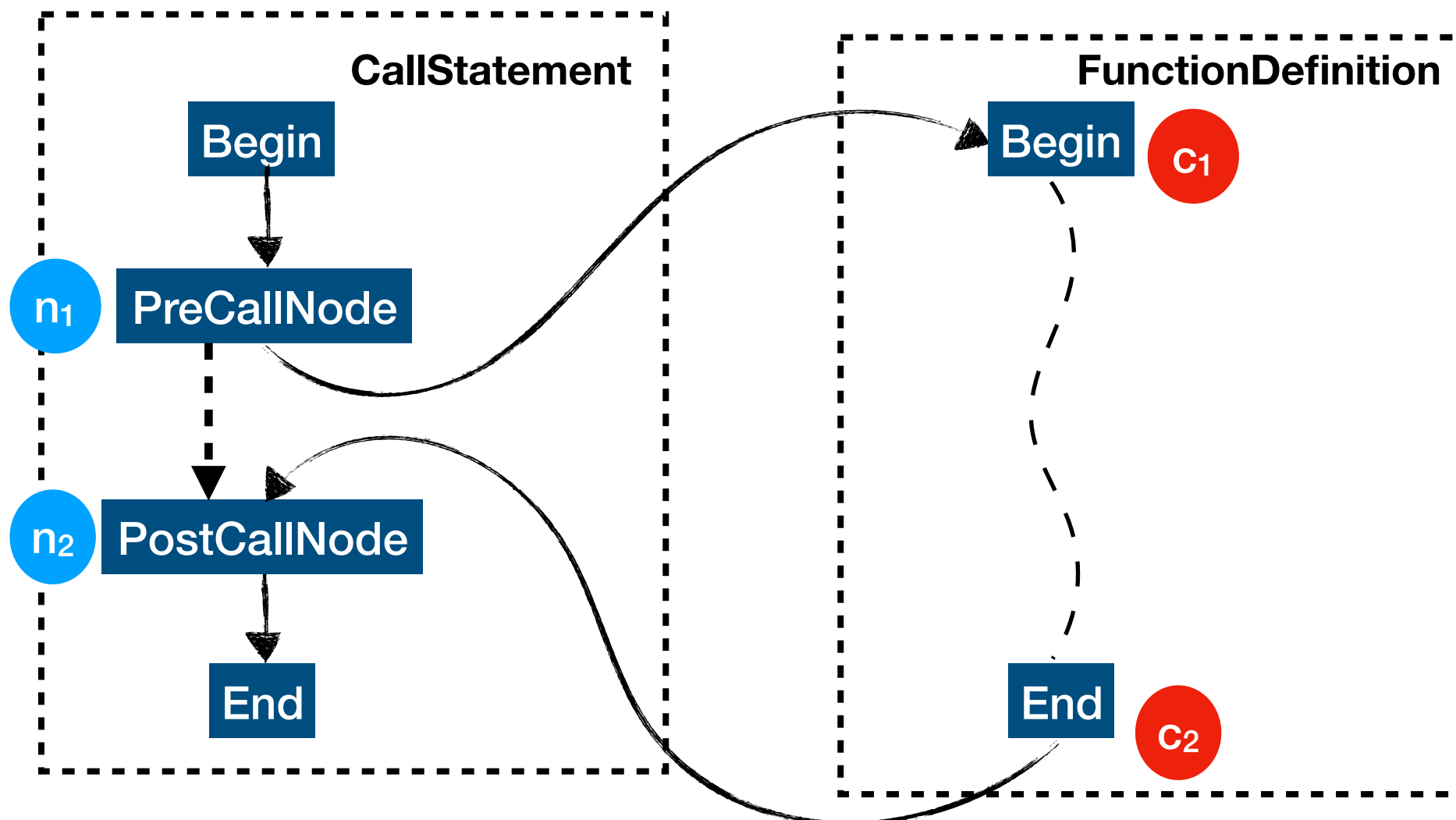
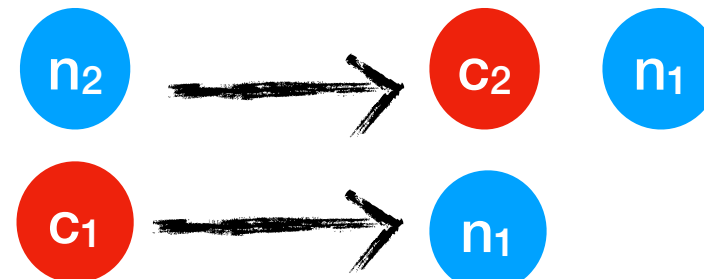
Successors of a node

```
node.getInfo().getCFGInfo().  
getInterProceduralSuccessors()
```



Predecessors of a node

```
node.getInfo().getCFGInfo().  
getInterProceduralPredecessors()
```



Queries on Call Graphs

- To obtain all `CallStatement`'s in a node:

```
Misc.getInheritedEnclosee(node, CallStatement.class))
```

- To obtain all `CallStatement`'s that may target a function:

```
func.getInfo().getCallersOfThis()
```

- To obtain all target `FunctionDefinition`'s of a `CallStatement`:

```
callStmt.getInfo().getCalledDefinitions()
```

- To obtain arguments of a `CallStatement`:

```
callStmt.getPreCallNode().getArgumentList()
```

- To check if a function is recursive:

```
func.getInfo().isRecursive()
```

Hands-On Session #3

Working with Call Graphs (CGs)

- (A) Print all call-sites present lexically within a given function.
- (B) Print all call-sites in the program that may have a given function as their target.
- (C) For a given call-statement: (i) print its target function(s), and (ii) print all its arguments.
- (D) Test whether a given method is recursive.

Note: Detailed walkthrough is present in the provided handouts.

Scopes, Symbols, and Types

- There are two kinds of symbols — (i) **variables**, and (ii) **functions**.
- Each symbol has following key attributes: (i) a **name**, (ii) a **type**, (iii) a **declaration**, and (iv) a **scope** that declares the symbol.
- IMOP models 3 kinds of scopes (Scopeable) —
 - *global* (TranslationUnit), *function* (FunctionDefinition), and *local* (CompoundStatement).
- Each scope maintains a symbol table for symbols defined in it.

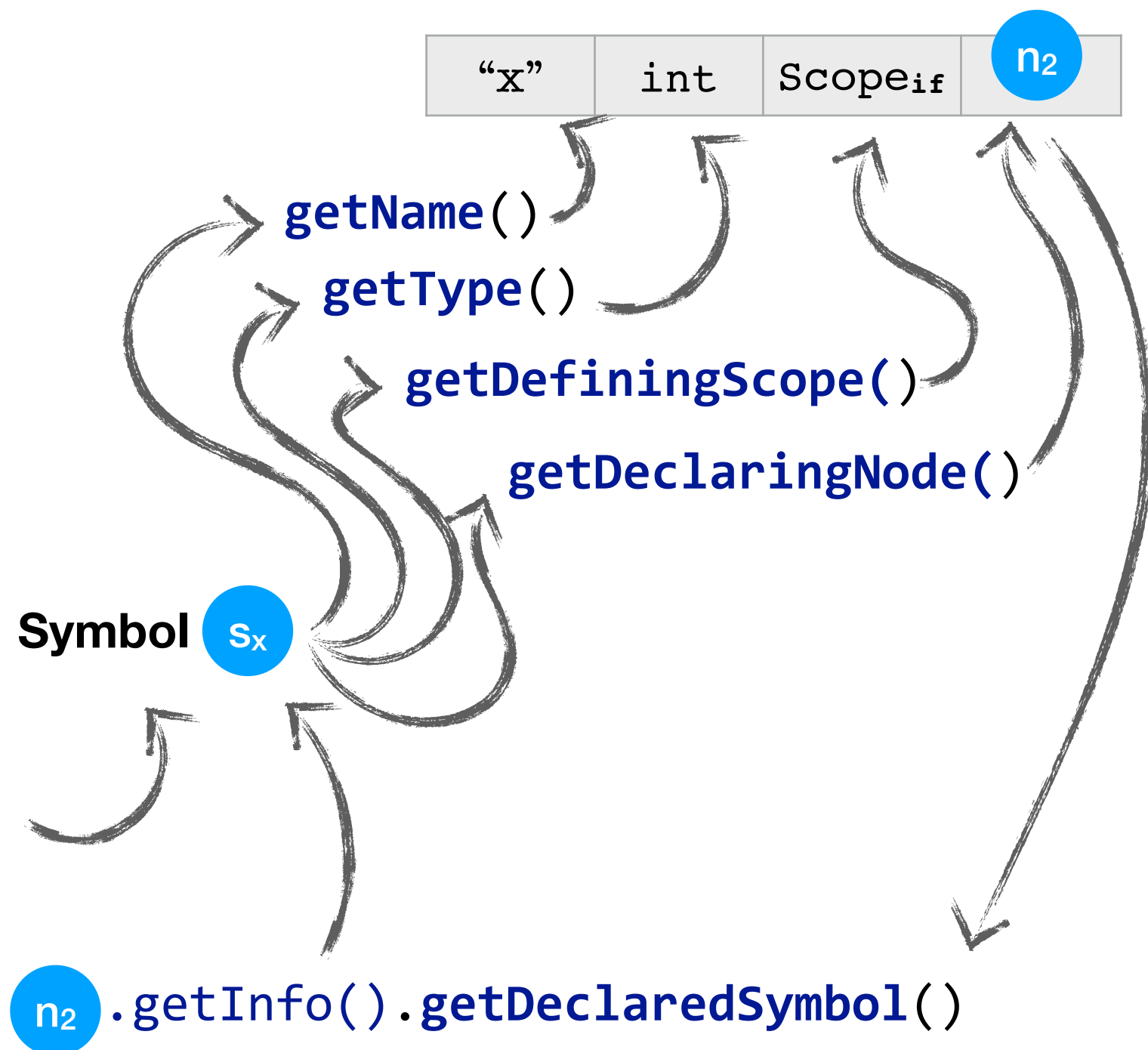
`scope.getInfo().getSymbolTable()`

Working with Symbols

```

int bar(float x);
int foo() {
    float x = 0.1; n1
    if (x == 0) {
        int x = 10; n2
        long int y = 10 + x;
        bar(x + y); n3
    }
}

```



Hands-On Session #4

On Scopes, Symbols, and Types

- (A) Print the names and types of symbols declared in a given function.
- (B) Print the scopes for each argument passed to a given call-statement.

Note: Detailed walkthrough is present in the provided handouts.

Memory Abstractions

- There are two main components of the data environment, from the perspective of static analyses: (i) **stack**, and (ii) **heap**.
- In IMOP, we term each element of these abstract components as **Cell**.
 - A stack-cell corresponding to a scalar is denoted by its **Symbol**.
 - For each aggregate stack-cell, IMOP uses a field-insensitive abstraction, **FieldCell**.
 - For each syntactic heap-allocation site, we model a single **HeapCell**.
- For efficiency, IMOP also maintains a single fixed `GenericCell`, which is used to model the universal set of cells.

Cell Accesses in a Node

- For each leaf/non-leaf CFG node, the list of cells that may be accessed, read, and written, by that node can be obtained using

```
int main() {
    int *a, b = 0;
    int c = 10;
    a = &b;
    *a = 10 + c++; n1
}
```

node.**getInfo().getReads()**



node.**getInfo().getWrites()**



node.**getInfo().getAccesses()**



Hands-On Session #5

Working with the Memory Abstractions

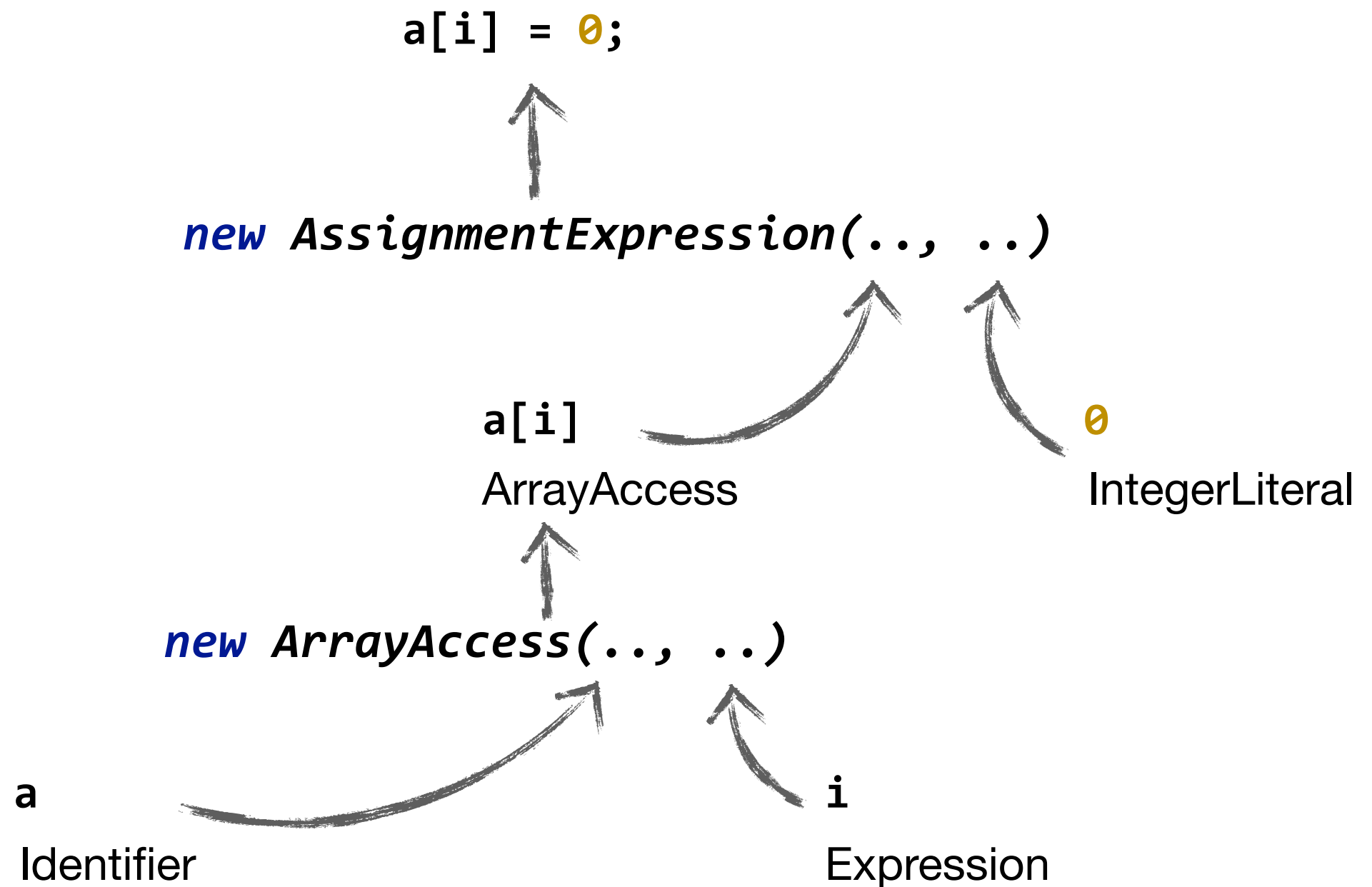
- (A) Write a pass that prints the set of cells (read and/or written) in a given expression-statement.
- (B) Write a pass that prints all those expression-statements which may write to a variable with given name.
- (C) Write a pass that prints the kind of data-dependences (RAW, WAR, or WAW), if any, that given two statements may have.

Note: Detailed walkthrough is present in the provided handouts.

Creating New Code Snippets

- Creation of new snippets of code during compiler passes is quite common.
e.g., creation of code snippets to be inserted in the program for instrumentation purposes.
- In almost all standard compiler frameworks, such as GCC, LLVM, Cetus, etc., *compiler writers need to create the AST denoting the snippet, **manually**!*
- IMOP utilizes the underlying parser to **automatically** create the AST for requested snippet using:
 - string of the snippet to be created,
 - type of the non-terminal at the root of the snippet AST.

Creating Snippets: Traditional Way



Creating Snippets in IMOP

`a[i] = 0;`



FrontEnd.parseAndNormalize("a[i] = 0;", *Statement.class*);

In IMOP, given the **string-equivalent** of a snippet to be generated, the actual AST can be obtained in a single step, as shown above.

Creating an unrolled loop

Code Snippet Generation: An example

// Print the loop-unrolled form of a while-statement.

```
public void printUnrolledLoop(WhileStatement loop) {
    WhileStatementCFGInfo cfgInfo = loop.getInfo().getCFGInfo();
    String snippetStr = "while (" + cfgInfo.getPredicate() + ") {"
        + cfgInfo.getBody() + "if (!("
        + cfgInfo.getPredicate() + ")) {break;}"
        + cfgInfo.getBody() + "}";

    Statement newBody = FrontEnd.parseAndNormalize(snippetStr, Statement.class);
    CompoundStatementNormalizer.removeExtraScopes(loop);
    System.out.println(newBody);
}
```

1 Create the string-equivalent of the string to be generated.

2 Invoke the parser to create AST of the new snippet.

3 Optionally, remove extra { } from the generated code.

Elementary Transformations

Let us revise the notion of CFG Components.

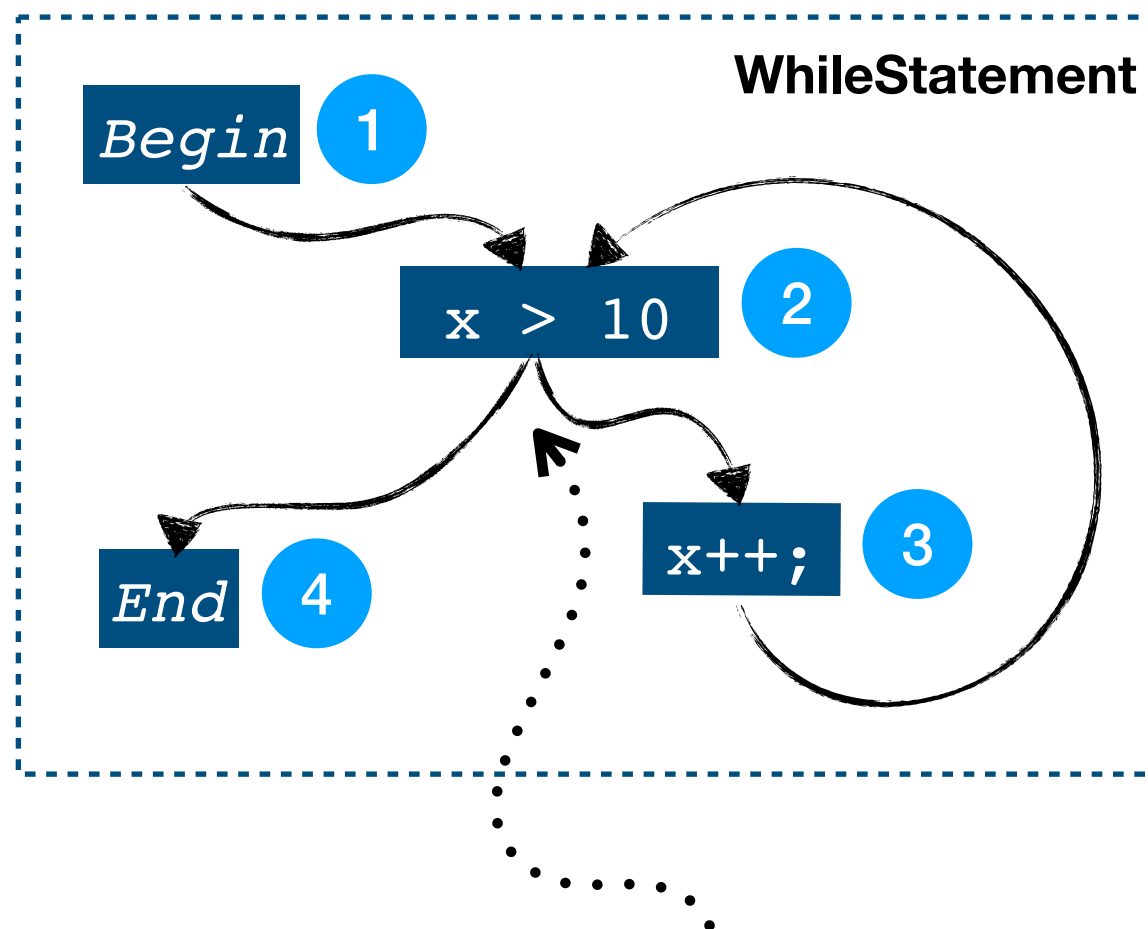
- Different parts of a non-leaf node, such as body and predicate of various constructs, are termed as its **CFG components**.
- **Elementary transformations** of a non-leaf node are those which can add/remove/replace its *CFG components*.

For example, one that replaces the predicate of a while-loop.

```
whileSmt.getInfo().getCFGInfo().setPredicate(newPred);
```

Four CFG components of a WhileStatement

- ① Begin node. ② Predicate node.
- ③ Body node. ④ End node.



Example: Simple Transformations

```
// Add empty else body to the if-statement if none exists.
public static void addEmptyElse(IfStatement ifStmt) {
    IfStatementCFGInfo cfgInfo = ifStmt.getInfo().getCFGInfo();
    if (cfgInfo.hasElseBody()) {
        return;
    }
    Statement emptyElse = FrontEnd.parseAndNormalize("{} ", Statement.class);
    cfgInfo.setElseBody(emptyElse);
}
```

Hands-On Session #6

Simple Program Transformations

- (A) Write a pass to perform loop unrolling for *while* loops.
- (B) Write a pass that translates a *do-while* loop to a *while* loop.
(Assume that no jump statements exist in the body.)

Note: Detailed walkthrough is present in the provided handouts.

Higher-level CFG transformations

```

1 int foo(int rb) {
2   int x = rb;
3   while (x < 10 + rb) {
4     x += 2;
5     if (x == 4) {
6 11: continue;
7     } else {
8         if (x == 5) goto 11;
9         else {x--; continue;}
10    }
11  }
12 }

```

Assume that we wish to add a read barrier for each read of parameter ***rb*** of *foo()*.

There are 3 key tasks involved:

- *Creation of snippet* for read barrier; **simple** and straightforward.
- *Detection of nodes* that may be reading from the parameter *rb*; this too is **simple**.
- *Performing the actual instrumentation*; **where shall we add the code?**

In this example

- There are two reads for *rb* -- at Line 2 and 3.
- For Line 2, we can simply add the read barrier code immediately before the line.
- But for Line #3, we need to insert read barriers at 3 positions!

Handling of such corner cases can be **difficult, repetitive, time-consuming, and error-prone**.

Higher-level CFG transformations

```

1 int foo(int rb) {
2     int x = rb;
3     while (x < 10 + rb) {
4         x += 2;
5         if (x == 4) {
6 11: continue;
7         } else {
8             if (x == 5) goto 11;
9             else {x--; continue;}
10        }
11    }
12 }

```

Handling of such corner cases can be **difficult, repetitive, time-consuming, and error-prone.**

IMOP resolves this issue by letting the programmer use one of the following **five** CFG transformations:

1. **Insert immediate predecessor.**
2. **Insert immediate successor.**
3. **Insert on the edge.**
4. **Node remover.**
5. **Node replacer.**

These transformations hide the syntactic forms/placements of nodes involved in the process.

In our example, we will require **just one invocation of InsertImmediatePredecessor to (automatically) handle all the possible cases.**

```
InsertImmediatePredecessor.insert(node, instrumentationCode);
```

Using Higher-Level CFG Transformations

Write barrier: Write a pass that instruments a program such that immediately before write to a scalar variable *thisVar* at runtime, a notification is displayed.

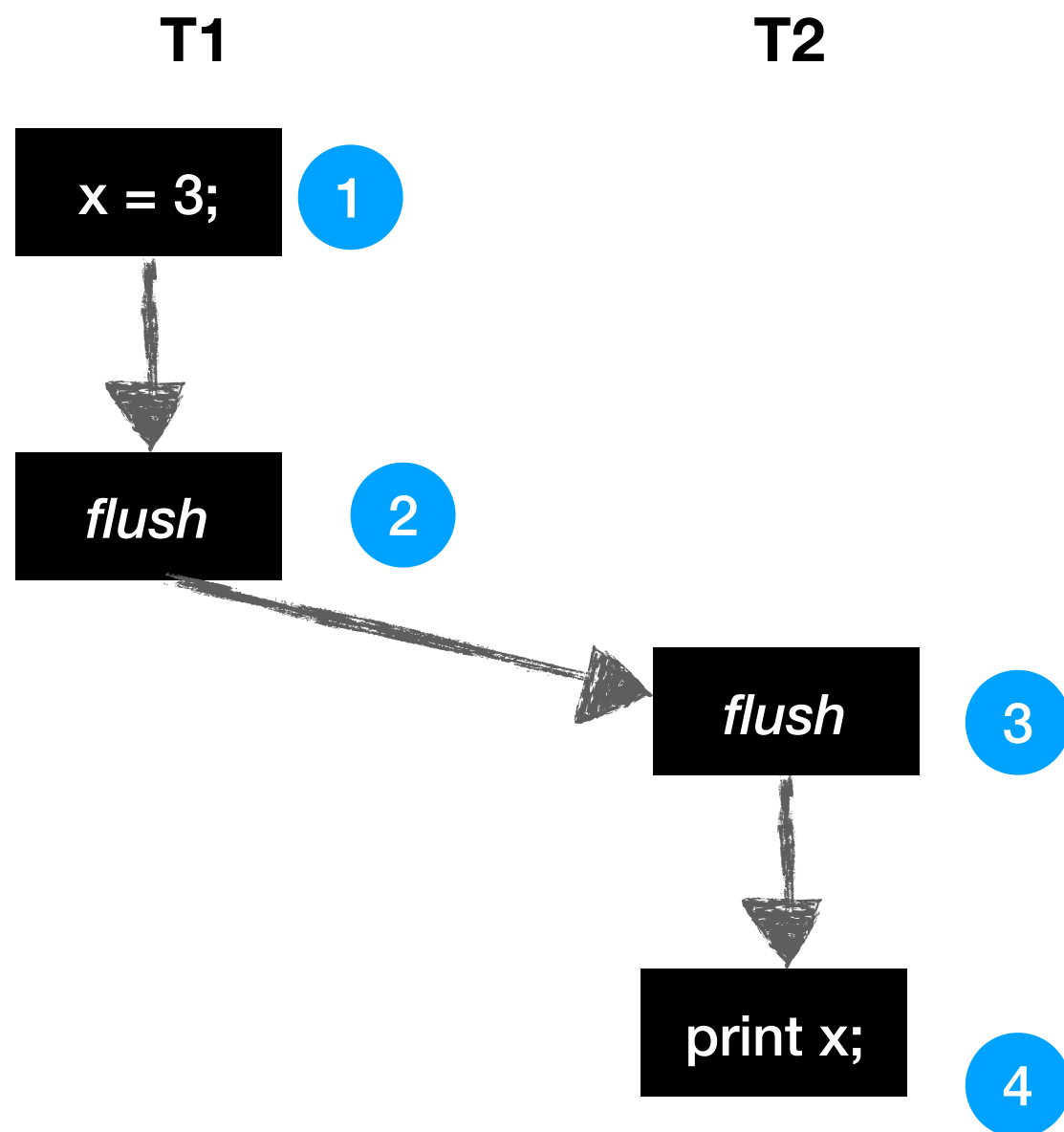
- a) Detect all those leaf CFG nodes that may write to *thisVar*.
- b) Create a notification message as a `printf()` statement.
- c) Insert the newly created statement immediately before the detected node.

Note: Detailed walkthrough is present in the provided handouts.

OpenMP

- IMOP has been written with **OpenMP semantics in mind**, since as early as its design phase.
- All program **abstractions of IMOP preserve OpenMP semantics**.
- For data-flow analyses, IMOP provides a framework where compiler writer needs to provide **only that data which is required for serial C programs**. OpenMP semantics are handled internally by IMOP.
- Many new program abstractions for parallelism, such as concurrency analysis, and LockSet analysis, are **already present** in IMOP.

Inter-Task Communications



- OpenMP API specifies that following four events are necessary for a communication to happen between two threads: (i) T1 writes to a shared location, (ii) T1 flushes that location, (iii) T2 flushes that location, and (iv) T2 reads from that location.
- We model these communications in IMOP by creating **edges** between those (implicit/explicit) flushes that may observe this pattern.
- *The resulting modified CFG is used in all data-flow and other static analyses to ensure that **OpenMP semantics are preserved**.*

Concurrency Analysis

- **Barrier directives** are program points where threads of a team wait for each other, before any thread is allowed to proceed.
- A ***phase*** is defined as a collection of statements from one set of barriers to the next set of barriers.
- Two statements can **never run in parallel** if they do not share any common phase.

*This observation helps **improve the precision/efficiency** of static analyses of parallel programs using phase (or concurrency) analysis.*

- **IMOP provides various interfaces** that can be used for answering related queries.

Concurrency Analysis

A glimpse into the interface

- To obtain all the static phases that are present in a `ParallelConstruct`.

`parCons.getInfo().getConnectedPhases()`

- To obtain the set of statements that may belong to a phase.

`ph.getNodeSet()`

- To obtain the set of barriers that may end a phase.

`ph.getEndpoints()`

- To obtain all the phases in which a statement may run.

`stmt.getInfo().getNodePhaseInfo().getPhaseSet()`

Hands-On Session #8

bit.ly/imop-iitm

Analyzing Concurrency in OpenMP programs

- (A) Print the number of static phases in every parallel-construct.
- (B) Print the highest number of statements in any static phase in the system.
- (C) Print the set of all those CFG leaf nodes that may run in parallel with the given expression statement.

Note: Detailed walkthrough is present in the provided handouts.

Hands-On Session #9

Project: Remove redundant barriers.

Check if a barrier-directive is required to preserve dependences among phases across it. If not, then delete the barrier.

- a) For any given barrier, get the set of phases that it may end, and the set of phases that may start after it.
- b) For each pair of phases from the sets in the last step, see if the pair conflicts, i.e. see if there exists any conflicting accesses between two phases of the pair.
- c) If no conflicts are found across a barrier, remove it from the program.

Note: Detailed walkthrough is present in the provided handouts.

Other Key Features of IMOP

Data-Flow Analysis

Generic Iterative Data-Flow Passes

- Iterative data-flow analysis is an important class of compiler passes.
- IMOP provides a set of generic *inter-thread inter-procedural flow-sensitive data-flow analysis* passes.
- For every existing and new instantiations of these generic passes, following guarantees are automatically ensured:
 - They respect the OpenMP semantics.
 - Their internal states are self-stable, in response to any existing or new program changes.

Data-Flow Analysis

Instantiating Generic IDFA Passes

- To instantiate the generic passes, compiler writers need to provide only the following information:
 - Structure of the flow-facts;
 - Value of the TOP element of lattice;
 - Meet operation on two flow-facts;
 - Notion of equality of two flow-facts; and
 - Transfer function for various kinds of CFG node.
- No additional information is needed from the compiler writers to ensure self-stabilization and adherence to OpenMP semantics.
- Various instantiations of the generic passes exist in IMOP — points-to, reaching definitions, liveness, dominator, lockset, copy propagation, etc.

Need for Compiler Stabilization

- Optimization passes involve program analyses and transformations.
- Transformations rely on program analyses for correctness, efficiency, precision, etc.
- In general, mainstream compilers do not automatically update program abstractions (analyses and representations) in response to program transformations — ***this can lead to incorrect application of downstream passes!***
- To resolve this challenge, compiler writers need to **manually** address these questions:
 - (a) What abstractions need stabilization in response to transformations?
 - (b) How to stabilize an abstraction?
 - (c) Where to write the stabilization code?

These queries need to be handled upon addition of each new analysis or optimization to the compiler.

IMOP: a Self-Stabilizing Compiler

Tackling Stabilization Behind the Scene

- In IMOP, compiler writers **need not write any code** to ensure self-stabilization while adding any
 - **new transformations**, and/or
 - **any new data-flow passes**.
- For other kinds of abstractions, compiler writers need to handle only a subset of stabilization tasks themselves.

Other interesting features of IMOP

- Lambda-based generic graph collectors.
 - Set of functionalities that *automate the graph-traversal mechanism* for frequently-occurring traversal patterns.
- Interface for Z3 SMT solver (by Microsoft).
 - Many analyses can be represented as a system of constraints.
 - IMOP can **automatically generate underlying system of inequations**, given one or more seed constraint(s).
 - It then *internally invokes **Z3** SMT solver* to return a conservative analysis result.
 - Used for auto-parallelization, dead-code removal, adding field-sensitivity to analyses, etc.

Maura et al., *Z3: an efficient SMT solver*, (TACAS 2008).

Use Cases of IMOP

Pedagogical Usage of IMOP

IMOP in a graduate-level course assignment

- In a graduate-level course, “Program Analysis, EVEN 16” (offered by Dr. Rupesh Nasre, Dept of CSE, IIT Madras), we used IMOP as an alternative framework for LLVM in one of the assignments.
- In the assignment, students had to *remove unsafe pointer dereferences from the program*.
- Total 10 out of 25 students used IMOP; rest used LLVM.
- None of the students had more than 1 lecture-hour exposure to IMOP in the past; all students had used LLVM for previous 3 assignments in that course.
- **Average marks** obtained by students who used IMOP were comparable to (rather, negligibly higher than) that of students who used LLVM.
- Average **submission-size** in IMOP ranged between **30-100 LOC**; in contrast, for LLVM, the range was **350-550 LOC**.

Pedagogical Usage of IMOP

Feedback scores from the students

We conducted a survey, asking students about their experiences working with IMOP.

Total 7 out of 10 students took the survey.

Average Ratings from the Survey:

- On **readability of code** written in IMOP versus that of LLVM: **4.14/5**
- On **ease of coding** in IMOP over that in LLVM: **4.57/5**
- On **ease of debugging** in IMOP over that in LLVM: **4/5**
- **Overall rating** of IMOP: **4.29/5**

Pedagogical Usage of IMOP

Feedback comments from the students.

Following were the comments given by 4 out of 10 students who used IMOP:

- ◆ “...I’d say working with it is **less difficult than LLVM...**”
- ◆ “...the ratio of how much one can achieve using the framework vs initial overhead of learning it, seems to be much higher for IMOP from my experience...”
- ◆ “...the nested CFG structure allowed **easy traversals of the CFG...**”
- ◆ “...I think the main aim of IMOP to **make writing compiler passes very easy** has been achieved compared to LLVM...IMOP **increases developer productivity.**”

IMOP in Published Projects

*IMOP has been used in the following published works, by **other** research groups:*

- (1) Jyothi Krishna Viswakaran Sreelatha, and Shankar Balachandran. IIT Madras. **Compiler Enhanced Scheduling for OpenMP for Heterogeneous Multiprocessors**. In Workshop on Energy Efficiency with Heterogeneous Computing (EEHCO 2016). ACM, Prague, Czech Republic.
- (2) Jyothi Krishna Viswakaran Sreelatha, Shankar Balachandran, and Rupesh Nasre. IIT Madras. **CHOAMP: Cost Based Hardware Optimization for Asymmetric Multicore Processors**. IEEE Transactions on Multi-Scale Computing Systems 4, 2 (TMSCS 2018), 163-176.
- (3) Jyothi Krishna Viswakaran Sreelatha, and Rupesh Nasre. IIT Madras. **Optimizing Graph Algorithms in Asymmetric Multicore Processors**. IEEE Transactions on CAD of Integrated Circuits and Systems 37, 11 (2018), 2673-2684.
- (4) Gnanambikai Krishnakumar, Alekhya Reddy Kommuru, Chester Rebeiro. IIT Madras. **ALEXIA: A Processor with Light Weight Extensions for Memory Safety**. ACM Transactions on Embedded Computing Systems, (2019).

*Note: IMOP is currently being used in **3 other projects** by 2 research groups (ours included).*

How far have we walked so far?

- ❖ Used the parser of IMOP, for a simple pretty-printing pass; perform simple queries on the **AST**.
- ❖ Understood the nested control-flow graphs (**CFGs**) and call graphs (**CGs**) of IMOP; perform traversals and queries on them.
- ❖ Worked with scopes, **symbols**, and **types**.
- ❖ Learnt how to query various **memory abstractions** maintained by IMOP.
- ❖ Created **new code snippets**, and perform **simple transformations** on the program.
- ❖ Performed **higher-level (semantic-level) transformations**, hiding the syntactic complexities of C and OpenMP.
- ❖ Utilized the existing **concurrency analysis** in IMOP.
- ❖ *Taken a glimpse into some other important **features**, and **use-cases**, of IMOP.*

Miles To Go

- IMOP is a source-to-source compiler framework, for OpenMP C programs.
In this tutorial, we have learnt how to use some of its key features.
- Developers of IMOP : Aman Nougrihiya, and V. Krishna Nandivada.
Total size of IMOP : ~**154 kLOC** (about ~**127 kLOC** manually written by Aman; rest generated by JavaCC/JTB).
Release date: **22-February-2020**
- Official website : bit.ly/imop-iitm
Github public repository: <https://github.com/amannougrihiya/imop-compiler>

What next?

For further development of IMOP, **YOUR contributions would really matter:**

Kindly try out IMOP, contribute towards enhancing its quality and capabilities, and let us know your suggestions for improvements.

Thanks! Questions and Feedback are Most Welcome! :)

IMOP Compiler Framework

A Self-Stabilizing Source-to-Source Compiler Framework for OpenMP C programs

The ever-growing need of faster, secure, and reliable programs across real-world domains has compelled continued improvements to compilers — a process that has seldom been a cake-walk for the compiler writers, specially when dealing with parallel programs. Clearly, compiler development needs to be made easier.

With its plethora of novel, well-thought out, design schemes, IMOP can significantly simplify the tasks of compiler writers, for both serial and parallel (OpenMP) C programs.

Pro-Automation

A key principle of IMOP is to automate as much of the tasks of compiler writers as feasible. IMOP achieves this automation along various dimensions, whether it be conversion of specifications of data-flow passes from serial to parallel, or stabilisation of program analyses in response to program transformations.

Self-Stabilization

Each program transformation may render the results of various analyses invalid. Mainstream compilers either require compiler writers to understand the analyses code and manually specify what analyses could be *preserved* or *destroyed*, or the compilers conservatively invalidate *all* the existing analyses results at the end of each pass.

IMOP handles this self-stabilization challenge without requiring any intervention from the compiler writers, for passes old and new.

Integration with Z3 SMT Solver (by Microsoft)

Complex compiler analyses (such as the ones that are *field-sensitive*) can be easily tackled using SMT solvers, once the analyses are expressed as systems of constraints. Unfortunately, it's hard, error-prone, and tedious for compiler writers to generate these constraints on their own, given any analysis query.

IMOP handles this challenge by automating both, the generation of system of inequations and the invocation of Z3 solver.

OpenMP-Aware Compilation

Unlike mainstream compilers, *each* component of IMOP has been built by taking OpenMP syntax/ semantics into account. This ensures that parallel semantics are preserved in each step of the compilation.

Lightweight
Pro-Automation
Self-Stabilizing
OpenMP-C
Open-Source
Ease-to-use
Z3-SMT-Solver-Integration
Source-to-Source
MidLevel-IR

Why IMOP?

Each of the 15+ people who have used the *pre-release* IMOP for different purposes, including 4 research publications, has given a consistent feedback — “*It is much easier to use IMOP than other alternatives such as LLVM*”.

Regardless, *IMOP exists to complement*, not compete with, the other compiler frameworks — IMOP's output remains to be a valid (OpenMP) C program, which can be fed to any other framework (like GCC or LLVM).

Developers

- ◆ Developed by Aman Nougrihiya (PhD Scholar), Email: amannoug@cse.iitm.ac.in.
 - Current size of the code-base: >150 kLOC
- ◆ Thesis/project advisor: V. Krishna Nandivada, Email: nvk@iitm.ac.in.