# Lab 4: Linear Constant Propagation Using IFDS/IDE or Full Constant Propagation Using the Call-strings Approach

In this lab, we can implement an inter-procedural linear constant propagation within the IFDS and IDE frameworks, using the Java/SootUp implementation for IFDS/IDE: *Heros*. A linear constant propagation propagates the values of constant variables or variables that depend on constants throughout the program. It is typically used in compiler optimization to reduce the amount of instructions that must be evaluated at runtime. In this lab, you can implement a linear constant propagation in IFDS. Then, you can implement an IDE version of the linear constant propagation, which is more complex, but yields a more efficient and scalable solution for real-world programs.

**General Instructions**

**Set up:** The code in folder `DECALab4-SootUp` contains a maven project readily set up. To set up your coding environment:

- Make sure that you have any version of Java running on your machine.
- Import the maven project into your favorite development environment.
- To run the test cases, run the project as a Maven build, with the goals set to "clean test".
- **Important: Make sure that your test cases run via the command line.** To do so, run `mvn clean test` in the `DECALab4-SootUp` directory.

**Project:** The project contains:

- `JUnit` tests in the package `test.exercises` of the directory `src/test/java/test/exercises`. Your goal is to make all of those test cases pass.
- target classes in the packages `target.exercise1and2` of the folder `src/test/java`. The test cases are run on those classes.
- two classes containing the flow functions of the two linear constant propagations you will develop in this lab. They are located in the package `analysis` of the folder `src/main/java`. For each exercise, you only need to provide correct implementations for the currently defaulted functions in the classes `IFDSLinearConstantAnalysisProblem.java` and `IDELinearConstantAnalysisProblem.java` in the package `analysis`. Do not edit any of the other files. There are `TODO` comments in the code that help you finding the right spot for your implementation.

**Exercise 1**

The IFDS analysis is already setup for you. The type of the data-flow facts used is `Pair<Local, Integer>`. Your IFDS linear constant propagation tracks tuples of `Local` $\times$ `Integer` in order to propagate constants throughout the program under analysis.

Observe the following code:

```
public static void main(String args[]) {
        int i = 10;
        int j = 20;
        i = 30;
}
```

Figure 1: Simple main

In this IFDS implementation of linear constant propagation, the lattice has been chosen such that the analysis stabilizes with the following data-flow facts after line 3 in Figure 1: $D = \{\langle i, 10\rangle, \langle j, 20\rangle, \langle i, 30\rangle\}$

Start working on the `getNormalFlowFunction()` factory method. This will make your implementation intra-procedural. You can leave the other flow function factories as is for the moment. When having implemented this flow function factory the first test cases should already pass.

Next, extend your linear constant propagation and make it an inter-procedural analysis by implementing the `getCallFlowFunction()` factory such that the data-flow facts from the caller's context are mapped into data-flow facts in the callee's context.

Subsequently, implement `getReturnFlowFunction()` to map the data-flow facts that hold at the end of a called function back into the caller's context.

Lastly, implement the `getCallToReturnFlowFunction()`. Restrict your implementation to transfer only the data-flow facts that are not involved in a call.

All test cases for the IFDS linear constant propagation should pass.

**Hints:**

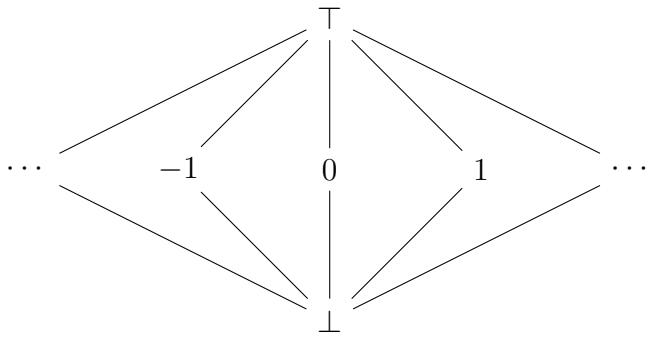
- You are free to use the pre-defined flow functions for common tasks that are pre-defined in the package `heros.flowfunc`. These pre-defined flow functions include `Gen`, `Kill`, `Identity`, etc..
- You can use the values `-1000` and `1000` as the lower and upper bounds of your analysis: a constant's value that exceeds this interval shall not be tracked. You will notice that the `JUnit` tests involving loops will emit an undesired behavior if you do not incorporate this special treatment.

(5 points)

**Exercise 2**

In this exercise, you will implement a linear constant propagation using the IDE framework. In this implementation of an IDE-based linear constant propagation, you will use variables as your data-flow facts. The values of the variables are tracked using the edge functions. This is more elegant and leads to increased performance. In this version of linear constant propagation, you

will use the following lattice; you will need to model it explicitly by overriding the adequate interface methods of the problem description:




The value domain in this exercise will be $\mathbb{Z}$ (or `Integer` in Java).

In order to encode your lattice, provide implementations for the interface factory methods `createJoinLattice` and `createAllTopFunction`. `createJoinLattice` lets you encode the lattice's values. The functions `topElement` and `bottomElement` refer to the top and bottom elements of the lattice. `join` tells the analysis framework how to join two values: "how to go up in the lattice".

Then, implement the flow function factories like in the IFDS implementation. Track the flow facts through the program and do not worry about the edge functions yet.

Finally, implement the edge functions that describe the value computation that is performed alongside the edges in the exploded super-graph.

- `EdgeFunction`s are function objects that describe a value computation in IDE alongside edges in the exploded super-graph that can be evaluated (using `computeTarget()`), composed (using `composeWith()`), joined (using `joinWith()`), and compared (using `equalTo()`).
- Like for the flow functions, implement `getNormalEdgeFunction()` first to obtain an intra-procedural analysis.
- Then, implement the functions `getCallEdgeFunction()`, `getReturnEdgeFunction()` and `getCallToReturnEdgeFunction()` in order to obtain a inter-procedural analysis.
- All these factory functions return an `EdgeFunction` that expresses how a statement affects the value of a data-flow fact.

All test cases should pass.

**Hints:**

- One way of determining the top and bottom values of the lattice is related to `Integer`'s `MIN_VALUE` and `MAX_VALUE`.
- In the literature, the meaning of the top and the bottom element are not used consistently. Sometimes, $\bot$ represents the element at which everything converges (over-approximation

reached after going up the lattice) and $\top$ represents the initial element (no information). In this course, and as shown in the Hasse diagram in this sheet, we use $\bot$ as the initial element and $\top$ as the over-approximated element. *Heros* does the contrary: it uses $\bot$ as the over-approximated element and $\top$ as the initial element. Thus, the interface method `createAllTopFunction()` must be implemented to return the function representation of the initial element (no information / $\bot$ in the course / $\top$ for *Heros*). Again, similar to the pre-defined flow functions you may wish to use the pre-defined edge functions such as `AllTop`.

- It is extremely helpful to use a debugger.
- When implementing the IDE analysis draw the exploded super-graph (IFDS/IDE's underlying data structure) for a given target program on a piece of paper before you start to encode flow- and edge-functions in Java.

(0 points - This exercise will not be graded. But it is a good way to strengthen your insight.)