apl. Prof. Dr. Thomas Noll
Christoph Matheja

General Remarks

- If you have questions regarding the exercises and/or lecture, feel free to write me an email (`matheja@cs.rwth-aachen.de`) or visit me at the chair (E2, room 4206).
- Please hand in your solutions in groups of three. If you are still looking for a group or your group has less than three members, please use the L2P or contact me after the exercise class.
- Solutions to programming exercises have to be handed in via L2P.
- You can hand in your solutions of theoretical exercises online via L2P or at the beginning of the exercise class.
- Please do *not* use the L2P to hand in large high resolution photos/scans of handwritten solutions.
- If you cannot access the L2P due to registration issues please contact me as soon as possible.
- Solutions to all exercises will be published in L2P. All other material, such as slides and exercise sheets, are distributed on our webpage.

Exercise 1 (Available Expressions): (37 Points)

Extend the WHILE programming language of the lecture by a *do-while*-construct.

a) Adapt the init- and final-mapping as well as the flow-relation to capture the newly introduced construct.

b) Additionally, adapt all concepts needed to perform an available expression analysis on programs using the *do-while*-construct.

c) Perform an available expression analysis on the following program:

```
y := x + 1;
x := x + y;
do
   y := x + 1;
while (y < x);
y := y * x;
```

Exercise 2 (Live Variables): (10 Points)

Consider the following program:

```
x := 42;
x := x - 23;
x := 17;
```

a) At which labels is x a live variable?

b) Is the result of a Live Variable Analysis reasonable for the program from above? Explain your answer.

Exercise 3 (Control Flow of Intermediate Language): (21 Points)

Static program analyses are often not performed on high-level source code, but on an intermediate language that is translated into machine code after optimization. We thus also consider a more low-level intermediate language of labeled While programs as introduced in the lecture.
For this intermediate language, we assume the set of program labels Lab to consist of all natural numbers, i.e. $Lab = \mathbb{N}$, including zero. Let arithmetic expressions $a \in AExp$ and Boolean expressions $b \in BExp$ be defined as in the lecture (Definition 2.1). Moreover, let $\ell, \ell' \in Lab$ be program labels. Then the possible individual statements in our intermediate language are given by the following grammar:

$$\begin{aligned}[p]^\ell \ ::= \ & [\texttt{skip}]^\ell \\ & \mid [\texttt{x} := a]^\ell \\ & \mid [\texttt{goto}\ \ell']^\ell \\ & \mid [\texttt{if}\ (b)\ \texttt{goto}\ \ell']^\ell\end{aligned}$$

A program in our intermediate language is then a sequence of the form

$$c \ ::= \ [p]^0; [p]^1; [p]^2; \ldots; [p]^n \ ,$$

where n is some natural number. Notice that we use labels as program counters instead of an explicit statement for sequential composition ($c\,;c$). Hence, program execution is started at label 0 and proceeds to the next label unless there is an explicit goto statement leading to another label. Furthermore, notice that choosing a label $\ell' > n$ in a goto statement will immediately terminate the program.

a) Translate the following `WHILE` program into an equivalent program in intermediate language.

```
if (x > y) then y := x + 1 else y := x - 1 end;
while(x > 0) do
    x := x - 1;
    y := y - x;
end;
x := y
```

b) Define the control flow relation, i.e. $\mathsf{flow} \subseteq Lab \times Lab$, for programs in intermediate language.

Exercise 4 (Implementation Task): (32 Points)

Over the course of the lecture we will develop a small static analysis tool for (a very restricted fragment of) Java programs. More precisely, the tool will compile a given Java program into an intermediate language, called Jimple, which is an easier-to-analyze intermediate language than Java Bytecode. Intuitively, the Jimple fragment considered in the exercises is very similar to the language considered in exercise 3.
In order to concentrate on concepts relevant to static program analysis, we provide a small framework in L2P that takes care of most other tasks such as parsing, translation from Java to Jimple, etc. **Since we will reuse this framework a few times, please try to complete at least the setup.**

Setup Please make sure to satisfy the following system requirements first:

- Java JDK version 1.8 is installed.[1]
- The environment variable `JAVA_HOME` is properly set on your machine.
- Apache Maven (https://maven.apache.org/) is installed.

[1] Unfortunately, Java 9 and Java 10 are not properly supported by the required libraries yet.

After unpacking the code framework in L2P (`code-framework-01.zip`), run `mvn package` within the unpacked directory. On the first run this will require an internet connection to obtain missing dependencies. If the build succeeds, an executable .jar file `spa-0.1-jar-with-dependencies.jar` will be created in the target directory. You can test whether building the framework was successful by running

```
java -jar target/spa-0.1-jar-with-dependencies.jar
```

which should display all available command line options.
When running an analysis, you have to specify the classpath (`-cp`), the name of a class that should be analyzed (`-cn`), and the name of the method that is analyzed (`-m`). If no further command-line options are supplied, the program will just output the translated program.

a) Please test whether programs are correctly translated by running

```
java -jar target/spa-0.1-jar-with-dependencies.jar -cp examples -cn Example01 -m f
```

If no errors occurred, this should display the translated program of method `f` in class `Example01` located in directory `examples`. **Please add the resulting output to your solution.**

b) The first analysis we are going to implement is *Live Variable Analysis*. This analysis is executed if the command-line option `-lv` is set. It then adds a comment to each line containing the analysis result for the respective label. Our code framework already contains a skeleton for implementing this analysis in the package `de.rwth.i2.spa.liveVariables`. Your task is to implement all required transfer functions. The corresponding methods are found in class `LVTransferFunctions` and are marked with a `// TODO` comment. **Feel free to add your own variables and methods to the above class, but please do not change any method signatures.**

The methods you have to implement are each of the form

```
Set<Variable> apply(Statement stmt, Set<Variable> d) ,
```

where `stmt` is an object representing the program statement, i.e. c in terms of the lecture. Moreover, `d` is the *analysis domain*, i.e. a set of program variables in case of Live Variables Analysis. Each of these methods should return the result of the transfer function for the respective statement. Formal definitions of the transfer functions (for `WHILE` programs) are found in the slides of the lecture.

You can build the whole project using the command `mvn package`.[2] Furthermore, we provide a few test files in the directory `examples`. You can run your implemented analysis on an example program as follows:

```
java -jar target/spa-0.1-jar-with-dependencies.jar -cp examples -cn ExampleXY -m f -lv
```

where `ExampleXY` is the name of the example that should be analyzed.

Hint: Package `de.rwth.i2.spa.availableExpressions` already contains a naive implementation of Available Expressions Analysis that you can execute by adding the command-line option `-ae`. Feel free to look at the corresponding implementation of transfer functions in class `AETransferFunctions`.

c) Run your analysis for the class `Exercise2` in the examples directory, i.e. execute

```
java -jar target/spa-0.1-jar-with-dependencies.jar -cp examples -cn Exercise2 -m f -lv
```

after you finished implementing your analysis (please attach the output to your solution). Does the problem discussed in exercise 2 also show up in your implemented analysis? Propose an improvement of Live Variable Analysis to yield more reasonable results (you do not have to implement your improvements).

[2] Common IDEs such as eclipse and intelliJ are also able to import, build and run maven projects.