Exercise: Data Flow Analysis

1 Available Expressions [38 points]

Consider the following program in a toy language with syntax inspired by Python. Assume all variables are integers and operators have the obvious semantics.

```
x = a - 3
y = a + 3
if x > a + 3:
      a = a * 3
else:
      x = a + 3
end
y = a - 3
```

Your task is to perform the *Available Expressions* data flow analysis, as presented in the lecture. Complete the following subtasks.

1.1 Control-Flow Graph [8 points]

Complete the following drawing of the control-flow graph (CFG) of the given program. As in the lecture, nodes are individual statements (and not basic blocks, as in other definitions of CFGs). Label each statement/node in the graph with a unique number, e.g., the line number in the source program, to help with the following subtasks. Do not include `else` or `end` in the CFG.

Check your result: The number of nodes (including entry and exit nodes) plus the number of edges (including edges from/to entry/exit nodes) should be 16.

1.2 Transfer Function [10 points]

First, list all *non-trivial, arithmetic expressions* in the program, i.e., expressions involving the operators: `+`, `-`, and `*`.

Expressions: { }

Next, fill the following table with the *gen* and *kill* sets of each statement in the program. For the first column, use the numbers from the CFG for identifiying statements.

Statement s	$gen(s)$	$kill(s)$

1.3 Solving Data Flow Equations [16 points]

Now, use the iterative algorithm from the lecture to solve the data flow equations for each statement in the program. You can iteratively fill up the second and third column of the table below during solving.

Statement s	$AE_{entry}(s)$	$AE_{exit}(s)$

1.4 Understanding [4 points]

Finally, to show your wider understanding of the applications of this data flow analysis, answer the following question: How could a compiler use the results of *Available Expressions* analysis to optimize the original program? (1-2 sentences as answer suffices.)

2 Live Variables [38 points]

Consider the following program in a toy language with syntax inspired by Python. Assume all variables are integers and operators have the obvious semantics.

```
x = 5
y = 0
while x > 0:
    x = x - 1
    while y < 10:
        y = x + y
end
y = 3
```

Your task is to perform the *Live Variables* data flow analysis, as presented in the lecture. Complete the following subtasks.

2.1 Reversed Control-Flow Graph [8 points]

Complete the following drawing of the *reversed* CFG of the given program, i.e., where all edges are inverted compared with the regular CFG. As in the lecture, nodes are individual statements (and not basic blocks, as in other definitions of CFGs). Label each statement/node in the graph with a unique number, e.g., the line number in the source program, to help with the following subtasks. Do not include `end` in the CFG.

Check your result: The number of nodes (including entry and exit nodes) plus the number of edges (including edges from/to entry/exit nodes) should be 19.

2.2 Transfer Function [10 points]

First, write down the domain of the *Live Variables* analysis for the given program, i.e., complete the following set.

Domain: { }

Next, fill the following table with the *gen* and *kill* sets of each statement in the program. For the first column, use the numbers from the CFG for identifiying statements.

Statement s	$gen(s)$	$kill(s)$

2.3 Solving Data Flow Equations [16 points]

Now, use the iterative algorithm from the lecture to solve the data flow equations for each statement in the program. You do not need to write down the data flow equations themselves here; you can iteratively fill up the second and third column of the table below during solving.

Statement s	$LV_{entry}(s)$	$LV_{exit}(s)$

2.4 Understanding [4 points]

Finally, to show your wider understanding of the applications of this data flow analysis, answer the following question: How could a compiler use the results of *Live Variables* analysis to optimize the original program? (1-2 sentences as answer suffices.)

3 Constant Propagation [24 points]

3.1 Example [6 points]

The following is about a data flow analysis that was *not* discussed in the lecture: *Constant Propagation.* This analysis is commonly used by compilers to avoid unnecessary computations by recognizing and replacing expressions that depend only on constants, and hence, can be computed at compile time rather than at runtime.

First, write down an original (!) example of a simple program in pseudocode (similar to the programs above), without the optimization applied and which exposes an opportunity for a compiler to optimize via a *Constant Propagation* analysis:

Second, write down the optimized program, after *Constant Propagation* has been applied:

3.2 Defining a Data Flow Analysis [18 points]

Please describe a *Constant Propagation* data flow analysis using the six properties discussed in the lecture. For any formal elements in your description, explain their meaning in a short sentence. You can assume programs with only integer variables.

Analysis domain:

Analysis direction:

$gen(s)$ function definition:

$kill(s)$ function definition:

Transfer function definition:

Meet operator:

Boundary condition (i.e., value of the transfer function at the start/final node of the CFG):

Initial values (i.e., value of the transfer function of nodes at the start of iterative solving):