Exercise 4 – Data Flow Analysis

Compilation 0368-3133

Due 8/1/2026 before 23:59

1 Assignment Overview

The fourth exercise implements a dataflow analysis for **L** programs. Specifically, you need to implement an uninitialized variable analysis for (a subset of) **L**. The exercise is roughly divided into three parts as follows:

1. Recursively traverse the AST to create an intermediate representation (IR) of the program.
2. Construct a control-flow graph.
3. Compute the results of the analysis using chaotic iterations.

The input for this exercise is a single text file containing a **L** program in the subset defined below. The output is a single text file indicating whether the program accesses uninitialized variables and listing them if it does.

2 The Input Programs

This section defines the valid input programs, which is a subset of **L**, and describes the semantics that your analysis must handle.

2.1 Syntax

To simplify the exercise, you may assume that the input program consists only of a single function with the signature, `void main()` and global variable declarations of type `int`. Furthermore, `main` uses only variables (global and local) of type `int` and the library function `PrintInt`. An input program is not required to contain the function `main` or any global variable declarations.

2.2 Semantics of Initializations

Recall that both global and local variables may be initialized upon definition. The initial value can be any numerical expression involving constants and previously defined variables. These initializations are executed in the order of their appearance in the source code. Specifically, before entering `main`, all global variables with initialized values should be evaluated, including those defined after `main`.

3 Uninitialized Variables Analysis

One of the most common programming errors is the use of a variable before its definition. This undefined value may produce incorrect results, memory violations, unpredictable behaviors, and program failure. More specifically, *used-before-set* refers to the error occurring when a program uses a variable that has not been

assigned an (initialized) value. This uninitialized variable, once used in a calculation, can be quickly propagated throughout the entire program. The program may produce different results each time it runs, may crash for no apparent reason, or may behave unpredictably [1].

In this exercise, you are asked to implement a *conservative* compile-time dataflow analysis that detects used-before-set errors. In class, we have seen a way to implement a *used-before-def* analysis based on reaching definitions (see Tutorial 8). Here, you are asked to design a more sophisticated analysis where a definition sets a variable `x` to an initialized value only if the value assigned to `x` is initialized. **Note that a simple used-before-def analysis, as seen in class, will not be sufficient for this case**. You will need to design an analysis that accurately captures this concept.

More formally, a program may have a used-before-set-a-valid-value error[1] involving variable `x` if there is a path p from the start of the program to a program point in which an expression using `x` is evaluated, and along path p either `x` is never assigned a value, or the most recent assignment to `x` does not assign it an *initialized value*. An assignment l : `x := exp` in path p sets `x` to an initialized value if `exp` consists of constants and variables that were assigned an initialized value prior to l : `x := exp` in p. Stated differently, assigning a variable with an expression involving an uninitialized value does not initialize that variable. These definitions consider variable *shadowing* and distinguish between different instances of `x` in different scopes (see the example in Fig. 4). Your analysis should detect and report all variables possibly involved in used-before-set errors. The following examples clarify these notions:

1. The program in Fig. 1 has use-before-set errors involving variables `g` and `y`. Note that `g` is used in an assignment command while `y` is used as part of a boolean condition.

2. The program in Fig. 2 has use-before-set errors involving variables `g`, `x`, and `y`. Note that `x` is assigned a value before it is used. This assignment, however, is to an expression containing `g`, which is uninitialized at that point.

3. The program in Fig. 3 also has use-before-set errors involving variables `g`, `x`, and `y`. Note that the use of `g` in `PrintInt(g)` is considered a use-before-set error because the loop body may never execute. Also, note that although `y` is involved in two potential use-before-set errors (one when evaluating the condition of the `if` statement and one during the assignment `y := y - 1`), it is printed only once.

   The analysis *should not* try to infer the infeasibility of paths at compile time. For example, if we replace the declaration of `y` in Figure 3 with:

   ```
   int y := 10;
   ```

   then the analysis should still report that variables `x` and `g` may be used-before-set, although the body of the loop in the program in Fig. 3 is always executed and thus assigns an initialized value to `g`.

4. The program in Fig. 4 has a use-before-set error involving variable `x`. Although `x` is initialized in the global scope, it is redefined again inside the function `main`, where it is uninitialized.

5. The program in Fig. 5 has no use-before-set errors.

4 Implementation Notes

- In the final exercise, you will be required to translate *any* **L** program into IR and perform *live variables* (or *liveness*) analysis on arbitrary **L** programs. The results of this analysis will be used to allocate the unbounded number of temporary variables to a fixed number of physical registers. Keep this in mind when implementing the translation to IR and the DFA engine in this exercise.

- Use the semantic information from the previous exercise to identify which variable declaration corresponds to each usage. The offset annotation is particularly useful to distinguish between variables with the same name.

[1] For brevity, from now on we will refer to such an error as a used-before-set error.

- The IR used in the project includes temporary variables. However, the final output must only list high-level variable names. You should consider how the use of these temporaries propagates uninitialized values through the code and design your analysis to track these dependencies accordingly.

5 Input and Output

Input: The input for this exercise is a single text file, the input program in the assumed subset of **L**. You can assume that the input program has no lexical, syntax or semantic errors.

Output: The output is a single text file containing one of the following:

- If no read accesses from uninitialized variables are detected: `!OK`
- Otherwise, a list of the names of the uninitialized variables that were accessed. These names must be sorted lexicographically and printed on separate lines.

6 Submission Guidelines

Each group should have **only one** student submit the solution via the course Moodle. Submit all your code in a single zip file named `<ID>.zip`, where `<ID>` is the ID of the submitting student. The zip file must have the following structure at the top level:

1. A text file named `ids.txt` containing the IDs of all team members (one ID per line).
2. A folder named `ex4/` containing all your source code.

Inside the `ex4` folder, include a makefile at `ex4/Makefile`. This makefile must build your source files into the static analyzer, which should be a runnable jar file located at `ex4/ANALYZER` (note the lack of .jar suffix). You may reuse the makefile provided in the skeleton, or create a new one if you prefer.

Command-line usage: `ANALYZER` receives 2 parameters (file paths):

- `input` (input file path)
- `output` (output file path containing the expected output)

Self-check script: Before submitting, you *must* run the self-check script provided with this exercise. This script verifies that your source code compiles successfully using your makefile and passes several provided tests. You must execute the self-check on your submission zip file within the school server (`nova.cs.tau.ac.il`) and ensure that your submission passes all checks. Follow these steps:

1. Download `self-check-ex4.zip` from the course Moodle and place it in the same directory as your submission zip file. Make sure no other files are present in that directory.
2. Run the following commands:
   ```
   unzip self-check-ex4.zip
   python self-check.py
   ```

Make sure that your `ids.txt` file follows the required format, and that your makefile does *not* contain hard-coded paths. The self-check script does **not** verify these aspects automatically and you are responsible for ensuring that your code compiles correctly in any directory.

Submissions that fail to compile or do not produce the required runnable will receive a grade of 0, and resubmissions will not be allowed.

7 Starter Code

The exercise skeleton can be found in the repository for this exercise. The skeleton provides a partial implementation of the IR translation, which you may modify as needed. Some files of interest in the provided skeleton:

- `src/ast/*.java` (classes for the AST nodes with an additional `irMe` method, implementing the AST visitor pattern described in tutorial 7)
- `src/TEMP/*.java` (contains a class that represents a temporary variable and implementation of the helper function `getFreshTEMP` (called `freshVar` in class) that generates a fresh unused before temporary variable)
- `src/IR/*.java` (classes for the IR commands)

Note that you need to use *your own* LEX and CUP configuration files from previous exercises, as well as your own AST nodes classes, as the ones in the skeleton provide only a partial implementation.

Building and running the skeleton: From the `ex4` directory, run:

```
$ make
```

This performs the following steps:

- Generates the relevant files using jflex/cup
- Compiles the modules into `ANALYZER`
- Runs `ANALYZER` on `input/Input.txt`

References

[1] Thi Viet Nga Nguyen, François Irigoin, Corinne Ancourt, and Fabien Coelho. Automatic detection of uninitialized variables. In Görel Hedin, editor, *Compiler Construction*, pages 217–231, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

```
int g;
void main() {
  int x;
  int y;
  int z;

  x := 1;
  if (y>0)  { z := x + g + 1;  x := x + 1;}
}

// expected output:
g
y
```

Figure 1: A program accessing uninitialized local and global variables.

```
int g;
void main() {
  int x := g * 2;
  int y;
  int z;

  g := 1;
  if (y>0)  { z := x + 1; }
}

// expected output:
g
x
y
```

Figure 2: A program accessing uninitialized local and global variables.

```
int g;
void main() {
  int x;
  int y;
  int z;

  while (y>0)  { z := x + 1;  g:=1; y:= y-1;}
  PrintInt(g);
}

// expected output:
g
x
y
```

Figure 3: A program accessing uninitialized local and global variables.

```
int x := 1;
void main() {
  int x;
  int y := x;
}

// expected output:
x
```

Figure 4: A program accessing uninitialized local variable.

```
int g0 := 0;
int g:= g0 + 1;
void main() {
  int x := g * 2 + 3;
  int y := 10;
  int z;
  int w;

  while (y>0)  { z := x + 1 + y;   g:=1; y:= y-1;}
  PrintInt(g);
}

// expected output:
!OK
```

Figure 5: A program which does not access uninitialized variables.