Exercise 4

Compilation 0368-3133

Due 2/2/2025 before 23:59

1 Introduction

We continue our journey of building a compiler for the invented object-oriented language **L**. This time, we dive into the realm of the machine-independent optimizer.

2 Programming Assignment

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 an **L** program in the subset defined below, and the output is a (single) text file that contains an indication of whether the program does not access uninitialized variables or a list of such variables if it does.

Note: In the last exercise you will be required to translate *any* **L** program to IR and perform *live variables* (or *liveness*) analysis for arbitrary **L** programs. Keep this in mind while implementing the translation to IR and the DFA engine in this exercise.

3 The Input Programs

3.1 Syntax

To simplify the exercise, you may assume that the program can consist only of a main function with signature `void main()` and declarations of global variables of type `int`. Furthermore, `main` may only use (global and local) variables of type `int` and the library function `PrintInt`.

3.2 Semantics of Initializations

Recall that both global and local variables may be initialized when defined. The initial value can be any numerical expression over previously defined variables. Thus, when these initializations occur, they should be in the same order of appearance as in the original program. Specifically, before entering `main`, all global variables with initialized values should be evaluated.

4 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 9) or liveness analysis (see lecture 10). 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 path p does not contain an assignment which sets `x` to 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.

5 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.

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

6 Output

The output is a single text file that contains either the string `!OK` if there are no read accesses from uninitialized variables, or a lexicographically sorted list of the names of the uninitialized variables that were accessed.

7 Submission Guidelines

The skeleton for this exercise can be found here:
https://github.com/noa-schiller/compilation-tau/tree/main/skeletons/ex4.

The makefile must be located in the following path:

- ex4/Makefile

This makefile will build the analyzer (a runnable jar file) in the following path:

- ex4/ANALYZER

Feel free to reuse the makefile supplied in the skeleton, or write a new one if you want to.

7.1 Command-line usage

ANALYZER receives two parameters (file paths):

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

7.2 Skeleton

The skeleton contains a partial implementation of the IR, which you may adapt to your needs. 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 8)
- *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)

As in previous exercises, you need to use *your own* implementations from previous exercises, including the LEX and CUP configuration files, as well as your implementation of the AST nodes.

7.3 Compiling

To build the skeleton, run the following command (in the *src/ex4* directory):
`$ make compile`
This performs the following steps:

- Generates the relevant files using jflex/cup
- Compiles the modules into *ANALYZER*

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.