University of Amsterdam

Assignment 4

Semantic Analysis

March 7, 2021

Students:
René Kok
13671146

Aram Mutlu
13574116

Lecturer:
Dhr. dr. C.U. Grelck

Course:
Compiler Construction

Course code:
5062COMP6Y

1 Introduction

This report provides information regarding the fourth assignment "Semantic Analysis" of the course Compiler Construction lectured by Dhr. dr. C.U. Grelck. The goal of this assignment is to derive scoping and symbol tables, apply lambda lifting and resolving function overloading.

2 Scoping and symbol tables

Consider the following CiviC nested function definition:

```
int d = 2;

int foo(int a) {
  int b = 1;
  int c;

  int f(int x) {
      int b = x + b;
      return b;
  }

  int g(int x) {
      c = f( x - b);
      return c + a;
  }

  c = c + g( d - b);
  return c;
}
```

Listing 1: CiviC nested function definition

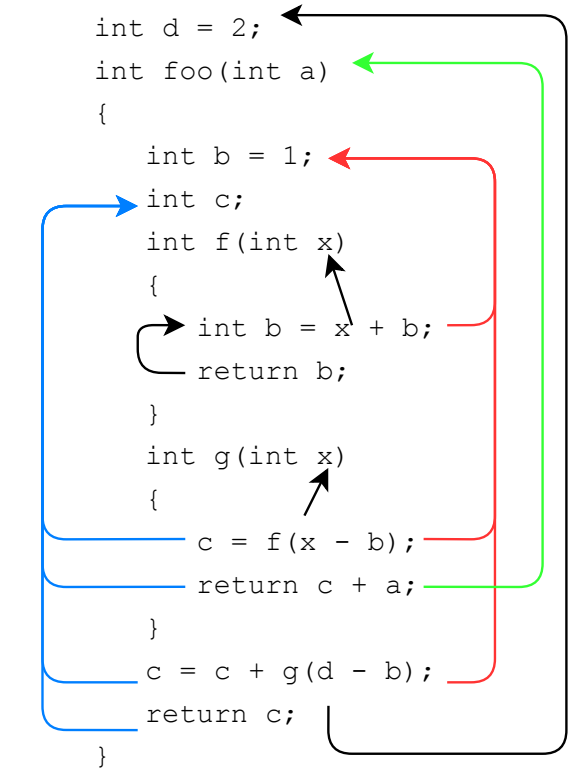
a). What is the value of foo(8), and, more importantly, why?

The result of foo(8), or any given integer as argument yields an undefined behaviour. This is because c is not static, not defined locally or because c is not initialized.

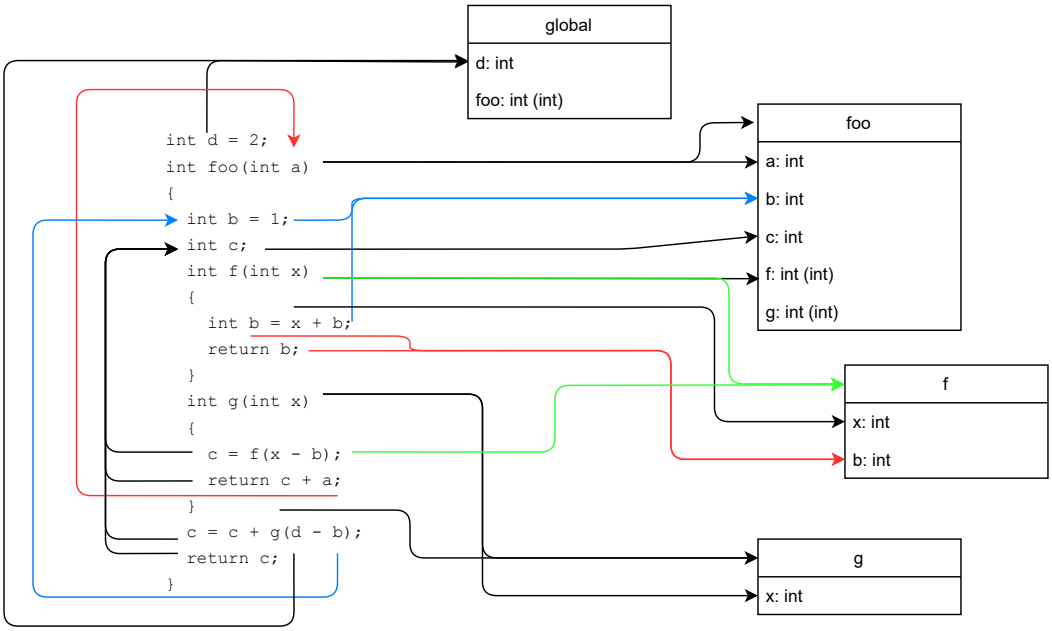
```
int d = 2;
foo(8) {
    int b = 1;
    int c;
    g(1) {
        f(0) {
            c = 1
            return local b = 1;
        }

        return 1 + 8;
    }

    c = undefined behaviour + 9;
    return c = undefined behaviour;
}
```

Listing 2: Execution steps of running foo(8)

b). Mark every occurrence of a variable identifier in statement position by an arrow to the declaration



c). Annotate each scope (level) with its symbol table.



d). Annotate each variable identifier in statement position by a number indicating the relative scope distance to the corresponding declaration.

```
      00      int d = 2;
    01  00    int foo(int a)
              {
      01          int b = 1;
      02          int c;
    03  00        int f(int x)
                  {
  01  00  11          int b = x + b;
      01              return b;
                  }
    04  00        int g(int x)
                  {
 22 23 00 21          c = f(x - b);
    22 20             return c + a;
                  }
22 22 24 30 21      c = c + g(d - b);
      22            return c;
               }
```

global
d: int
foo: int (int)

foo
a: int
b: int
c: int
f: int (int)
g: int (int)

f
x: int
b: int

g
x: int

3 Lambda lifting

Manually apply the lambda lifting transformation to the code example of listing 1.

```
int d = 2;

int f(int x, int b) {
        int b = x + b;
        return b;
}

int g(int x, int a, int b, int *c) {
        *c = f(x - b);
        return *c + a;
}

int foo(int a, int d) {
        int b = 1;
        int c;

        c = c + g(d - b, *c, b, a);
        return c;
}
```

Listing 3: Lambda lifting transformation

4 Function overloading

Assume we would extend CiviC by function overloading. Describe how this extension would affect semantic analysis in the CiviC compiler in general, and how you would solve the corresponding problems in detail.
The compiler wouldn't be able to distinguish similarly named functions. The determination of which function to use for a particular call should be resolved at compile time by adding the name of the parameter types to the function name as shown in table 1 below.

Original overloaded	**Compiler resolved**
`int foo(int a);`	`int foo_int(int a);`
`int foo(float a);`	`int foo_float(float a);`
`int foo(const char* a);`	`int foo_char_p(const char* a);`
`int foo(int a, int b);`	`int foo_int_int(int a, int b);`
`int foo(float a, float b);`	`int foo_float_float(float a, float b);`

Table 1: Implementing function overloading