DCC888 - Static Analysis of Programs

Name: _____ **ID**: _____

Name: _____ **ID**: _____

By turning this exam in, I give my word that I have done it with my team only, on the understanding that we are allowed to consult any material publicly available, except material disclosed by other colleagues outside our team who are taking this course, or who have taken it in the past.

Goal: The goal of this exam is to solve a puzzle: you will be given a number of instructions. You must chain them into a control-flow graph, in such a way that the resulting program runs without errors. The only error that shall be considered is the possibility of an instruction reading a value that has not yet been defined. The instructions that you shall consider have the following semantics, as given by a Python interpreter available at
https://homepages.dcc.ufmg.br/~fernando/coisas/phiPuzzle.txt

```
class Inst:
    def __init__(s):
        s.next_inst = None
    def set_next(s, next_inst):
        s.next_inst = next_inst
    def get_next(s):
        return s.next_inst

class BinOp(Inst):
    def __init__(s, dst, src0, src1):
        s.dst = dst
        s.src0 = src0
        s.src1 = src1
        super().__init__()
    def definition(s):
        return s.dst
    def uses(s):
        return [s.src0, s.src1]

class Add(BinOp):
    def eval(s, env):
        env.set(s.dst, env.get(s.src0) + env.get(s.src1))

class Mul(BinOp):
    def eval(s, env):
        env.set(s.dst, env.get(s.src0) * env.get(s.src1))

class Lth(BinOp):
    def eval(s, env):
        env.set(s.dst, env.get(s.src0) < env.get(s.src1))

class Geq(BinOp):
    def eval(s, env):
        env.set(s.dst, env.get(s.src0) >= env.get(s.src1))
```

```
class Phi(Inst):
    def __init__(s, dst, args):
        s.dst = dst
        s.args = args
        super().__init__()
    def definition(s):
        return s.dst
    def uses(s):
        return s.args
    def eval(s, env):
        env.set(s.dst, env.get_first(s.uses()))

class Bt(Inst):
    def __init__(s, pred, true_dst, false_dst):
        s.pred = pred
        s.true_dst = true_dst
        s.false_dst = false_dst
        super().__init__()
    def definition(s):
        return None
    def uses(s):
        return [s.pred]
    def set_true_dst(s, true_dst):
        s.true_dst = true_dst
    def set_false_dst(s, false_dst):
        s.false_dst = false_dst
    def eval(s, env):
        if env.get(s.pred):
            super().set_next(s.true_dst)
        else:
            super().set_next(s.false_dst)

def interp(instruction, environment):
    if instruction:
        instruction.eval(environment)
        interp(instruction.get_next(), environment)
    else:
        environment.dump()
```

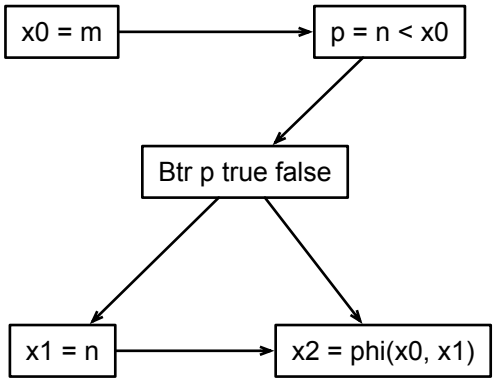
These instructions receive an environment E, and potentially modify it. For instance, the semantics of the Add instruction can be described as follows:

$$\frac{E[x0] = n0. \qquad E[x1] = n1}{val(Add(x, x0, x1, E)) \mapsto E[x] = n0 + n1}$$

But notice that if x0 does not exist in E, or if x1 does not exist in E, then the evaluation of the Add operation is stuck.

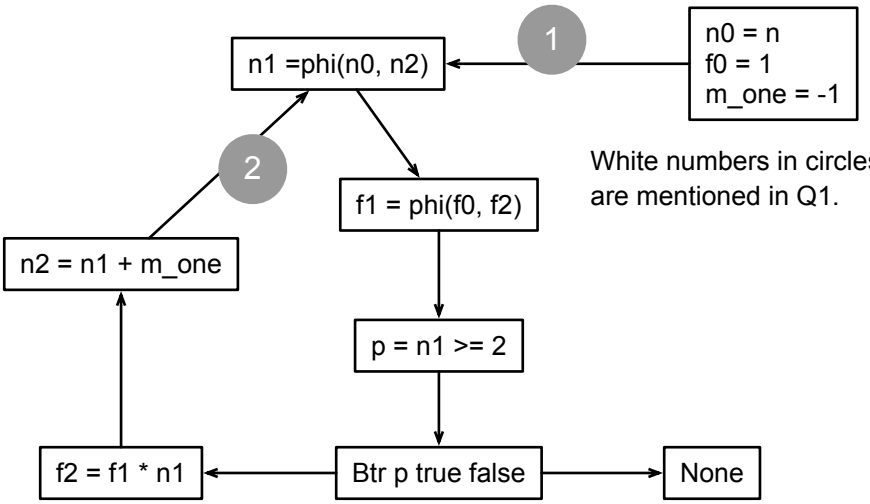
The implementation of the environment itself is not very important to this exercise. It suffices to imagine it as a table, that returns the value associated with some variable. The only interesting detail concerns the semantics of Phi-Functions: the evaluation of a phi-function such as x = phi(x1, x2, …, xn) in the environment E will search for the first occurrence of one of the arguments x1, …, xn in E. Thus, in this regard, E is implemented as a stack: the first valid definition of a variable is the last definition of it. For instance, if E = [(a, 1), (b, 2), (a, 3)], then E[a] = 1 and E[b] = 2 and E[c] is not defined. Some examples of programs are given below. We start with a program that computes the minimum of two values, m and n:

```
int min(int m, int n) {
    int x = m;
    if (x < n)
        x = n;
    return x;
}
```



The next program computes the factorial of a given integer n:

```
int fact(int n) {
    int f = 1;
    while (n >= 2) {
        f = f * n;
        n = n - 1;
    }
    return f;
}
```



Notice that the program flow is determined by an attribute next_inst, that is part of each object that represents an instruction. Interpretation ends if the interpreter receives an instruction whose next_inst field is the value None. As an example, the test below implements the fact function:

```
def test_fact(n):
    env = Env({"two": 2, "n0": n, "f0": 1, "m_one": -1})
    n1 = Phi("n1", ["n0", "n2"])
    f1 = Phi("f1", ["f0", "f2"])
    p = Geq("p", "n1", "two")
    f2 = Mul("f2", "f1", "n1")
    n2 = Add("n2", "n1", "m_one")
    b = Bt("p", f2, None)
    n1.set_next(f1)
    f1.set_next(p)
    p.set_next(b)
    f2.set_next(n2)
    n2.set_next(n1)
    interp(n1, env)
```

Branch instructions are initialized with three fields. The first, “p” in this example, is the predicate that controls the branch. The other two fields are the destination instructions in case the branch is true or false, respectively.

The code available at https://homepages.dcc.ufmg.br/~fernando/coisas/phiPuzzle.txt contains a few more examples of programs.

Question 1 [4 Points] The way phi-functions are implemented might look a bit mysterious in principle. To make it more concrete, check below the implementation of the get_first method in the environment, and the implementation of the evaluation of phi-functions:

```
class Env:
    def __init__(s, initial_args={}):
        s.env = deque()
        for var, value in initial_args.items():
            s.env.appendleft((var, value))

    def get(s, var):
        return s.get_or_raise(lambda var_name: var_name == var)

    def get_first(s, vars):
        return s.get_or_raise(lambda var_name: var_name in vars)

    def get_or_raise(self, pred):
        val = next((value for (e_var, value) in \
                            self.env if pred(e_var)), None)
        if val != None:
            return val
        else:
            raise LookupError(f"Absent key {val}")

    def set(s, var, value):
        s.env.appendleft((var, value))
```

```
class Phi(Inst):
    """
    Example:
        >>> a = Phi("a", ["b0", "b1"])
        >>> e = Env()
        >>> e.set("b0", 1)
        >>> e.set("b1", 3)
        >>> a.eval(e)
        >>> e.get("a")
        3
        >>> a = Phi("a", ["b0", "b1"])
        >>> e = Env()
        >>> e.set("b1", 3)
        >>> e.set("b0", 1)
        >>> a.eval(e)
        >>> e.get("a")
        1
    """
    def __init__(s, dst, args):
        s.dst = dst
        s.args = args
        super().__init__()
    def eval(s, env):
        env.set(s.dst, env.get_first(s.uses()))
```

And yet, it does works! Provide a argument (even if informal) explaining why the semantics of phi-functions are correct. In other words, if we have an instruction INST like “x = phi(x1, x2, …, xn)”, then INST will copy into ‘x’ the variable ‘xi’ that was lastly defined in the path that reaches INST. For instance, if the program flow reaches “n1 =phi(n0, n2)” in the factorial program above through Edge 1, then the interpreter will move n0 into n1. If the flow do it through Edge 2, then n2 will be copied into n1 instead. And that is the expected behavior of the phi-function. Why are we guaranteed to know that n2 will be defined ahead of n0 in the environment stack, in this particular example? Can you generalize your explanation for any program?

Question 2 [4 Points] Consider the following predicate, implemented in Prolog. We have that gcd(X, Y, Z) is true whenever Z is the maximum common divisor between integers X and Y.

```
gcd(X, Y, Z) :- X =:= Y, Z is X.
gcd(X, Y, Denom) :- X > Y, NewY is X - Y, gcd(Y, NewY, Denom).
gcd(X, Y, Denom) :- X < Y, gcd(Y, X, Denom).
```

Implement, in our low-level programming language, a function gcd(X, Y), which stored in a variable Z (which you must set in the environment) the value of the maximum divisor of X and Y. You must implement an actual Python function test_gcd, which builds the program (like test_fact() in Page 2) and runs it. Your Python code must follow the structure below:

```
def test_gcd(X, Y):
    env = Env({“zero”: 0, “minus_Y”: -y, ...})
    # TODO: create instructions.
    # TODO: connect instructions to form a CFG
    interp(first_instruction, env)
```

Question 3 [8 Points] Implement the function build_cfg(L), whose skeleton in available in https://homepages.dcc.ufmg.br/~fernando/coisas/phiPuzzle.txt. This function receives a list of instructions, and then connects these instructions into a control-flow graph G that is *valid*. By valid, we mean to say that interp(G, E) will never run into a stuck function, as long as all the free variables in L are defined in G. A variable V is free in L if L does not contain any instruction that defines V. Your implementation must contain a header comment explaining what is the algorithm that you have used to implement build_cfg(L). The reference implementation contains a number of tests, like the function test0 below, that you can use to verify that your code is correct:

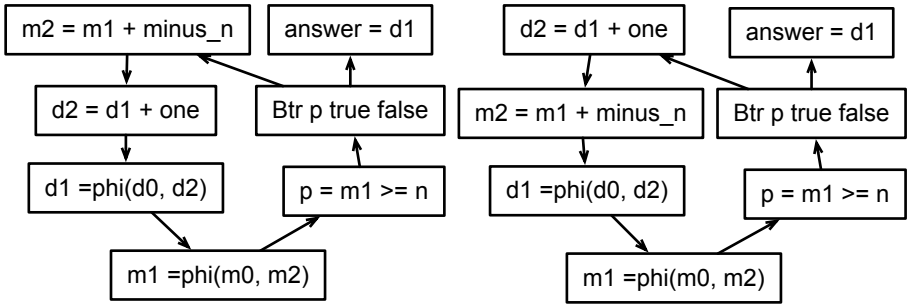
```
def test_build_cfg0():
    add0 = Add("a0", "i0", "i0")
    add1 = Add("a1", "a0", "a0")
    mul0 = Mul("m0", "a0", "a1")
    build_cfg([add0, add1, mul0])
    env = Env({"i0": 3, "i1": 5})
    interp(add0, env)
```

Notice that your algorithm does not produce a "correct" program, for the meaning of correctness, in this exercise, does not depend on what the program does. It depends, rather, on structural properties of the program. As an example, imagine that you are given the following input:

Instructions:

- d2 = d1 + one
- Btr p true false
- d1 =phi(d0, d2)
- m2 = m1 + minus_n
- m1 =phi(m0, m2)
- p = m1 >= n
- answer = d1

In this case, the following outputs out be valid. The first one computes, in answer, the integer division of m by n. The second does not do anything meaningful, but it is still structurally correct:




```
def test_div(m, n):
    env = Env({"d0": 0, "m0": m, "one": 1, "n": n, "minus_n": -n, "zero": 0})
    d1 = Phi("d1", ["d0", "d2"])
    m1 = Phi("m1", ["m0", "m2"])
    p = Geq("p", "m1", "n")
    m2 = Add("m2", "m1", "minus_n")
    d2 = Add("d2", "d1", "one")
    answer = Add("answer", "d1", "zero")
    b = Bt("p", m2, answer)
    d1.set_next(m1)
    m1.set_next(p)
    p.set_next(b)
    m2.set_next(d2)
    d2.set_next(d1)
    interp(d1, env)
```

Obs.: If you want to test the correct implementation of the div function (in the gray box below), you can use this low-level program on the left.

```
int div(int m, int n) {
    int d = 0;
    while (m >= n) {
        m = m - n
        d = d + 1
    }
    return d;
}
```

Question 4 [2 Points] In Question 3, above, we have been talking about "Structural Properties" of programs, which cause the reconstruction to be correct. State the key property that your program reconstruction algorithm must ensure. Think about programs in Static Single Assignment form: what are/is the key properties/y of such programs?

Question 5 [2 Points] Describe (in English/Portuguese, not in Python!) an algorithm that would verify if the properties/y stated in Question 4 are met by a given program. Consider that a program is a list of *connected instructions*, plus one particular instruction that is the *starting point* of the program, plus an environment that assigns values to variables that are free in the list of instructions.