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 exercise is to design a code transformation that builds executable slices out of complete programs. We shall consider the following definitions:

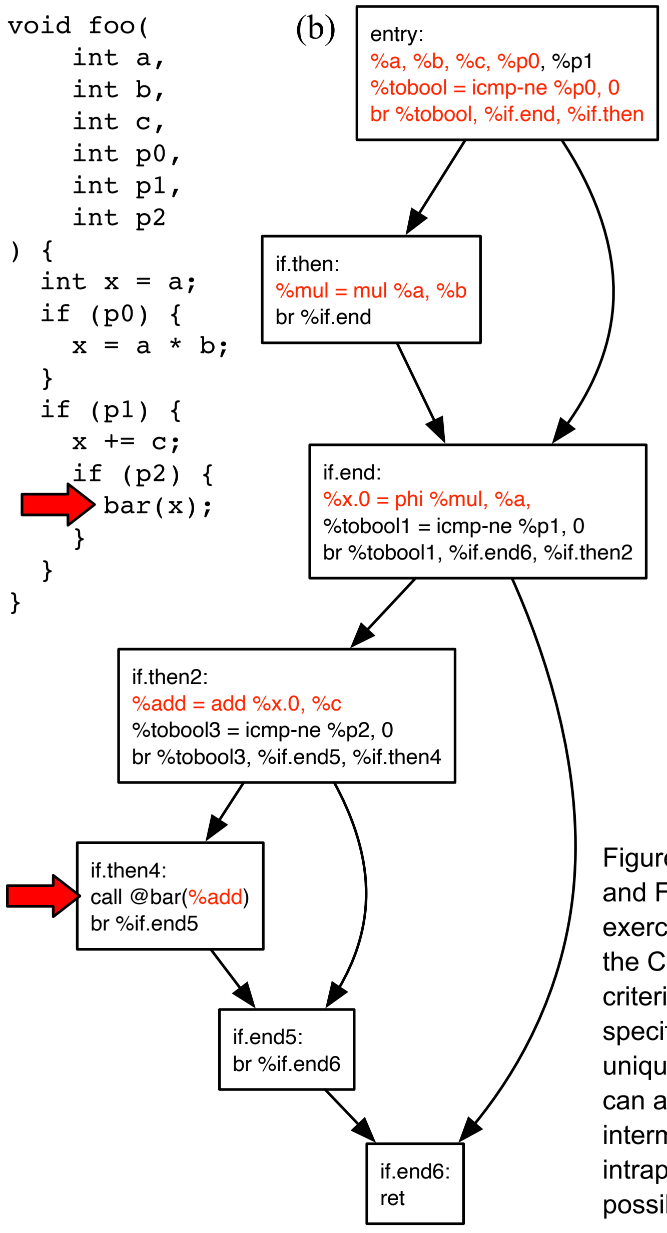
* A slice criterion is the use or definition site of a variable in an SSA-form program.
* Given a slice criterion "v = E", we call a slice a function F that computes E.

The example below clarifies these notions. Figure (a) shows a C program. Let's build a function that computes the value of x in line 16 of function foo. Once the program is converted to SSA-form, as shown in part (b) of the figure, the slice criterion becomes the definition "%add = add %x.0, %c", because that is the computation of the value that is used in line 16.

```
01 void foo(
02     int a,
03     int b,
04     int c,
05     int p0,
06     int p1,
07     int p2
08 ) {
09   int x = a;
10   if (p0) {
11     x = a * b;
12   }
13   if (p1) {
14     x += c;
15     if (p2) {
16       bar(x);
17     }
18   }
19 }
```

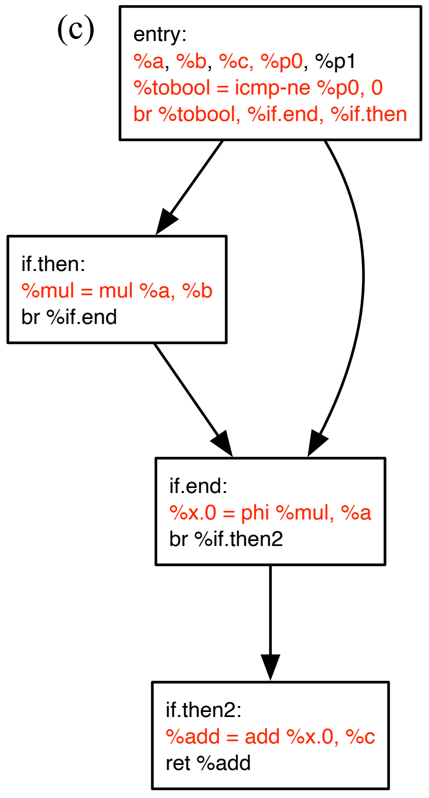
(a)

(b)



```
entry:
%a, %b, %c, %p0, %p1
%tobool = icmp-ne %p0, 0
br %tobool, %if.end, %if.then

if.then:
%mul = mul %a, %b
br %if.end

if.end:
%x.0 = phi %mul, %a,
%tobool1 = icmp-ne %p1, 0
br %tobool1, %if.end6, %if.then2

if.then2:
%add = add %x.0, %c
%tobool3 = icmp-ne %p2, 0
br %tobool3, %if.end5, %if.then4

if.then4:
call @bar(%add)
br %if.end5

if.end5:
br %if.end6

if.end6:
ret
```

(c)



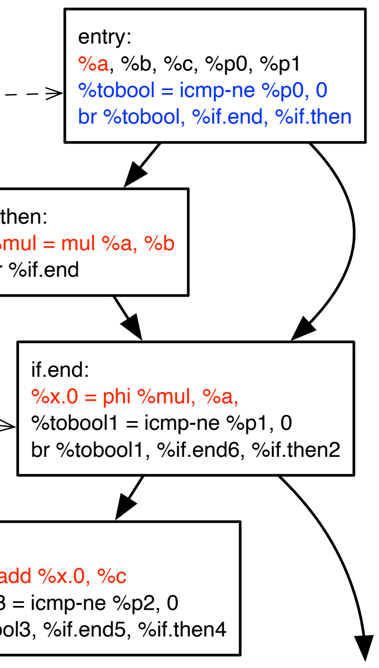
```
entry:
%a, %b, %c, %p0, %p1
%tobool = icmp-ne %p0, 0
br %tobool, %if.end, %if.then

if.then:
%mul = mul %a, %b
br %if.end

if.end:
%x.0 = phi %mul, %a
br %if.then2

if.then2:
%add = add %x.0, %c
ret %add
```

```
01 int foo_x_at_14(
02     int a,
03     int b,
04     int c,
05     int p0,
06 ) {
07   int x = a;
08   if (p0) {
09     x = a * b;
10   }
11   x += c;
12   return x;
13 }
```

(d)

Figure (c) shows the backward slice that computes the value of %add, and Figure (d) shows the corresponding C function. Your task in this exercise is to come up with a code-extraction algorithm that produces the CFG in figure (c) out of the CFG in figure (b), given the slice criterion "%add". Notice that, once in SSA form, we do not need to specify the entire definition of %add. Its name define the slice criterion uniquely. Your algorithm must work for any program in SSA form. You can assume that we are working on a language that follows the LLVM intermediate representation. You must produce maximal intraprocedural slices. In other words, your slices must be as large as possible, but do not have to leave the scope of the enclosing function.

Question 1 [2 Points] In the above example, the assignment "%add = add %x.0, %c" is controlled by two conditional branches. The outcome of the first conditional branch is determined by the predicate %**tobool**. The outcome of the second conditional branch is determined by the predicate %**tobool1**. However, once we build the slice, only the first branch is important. The second branch plays no role in this slice. Nevertheless, both control the assignment that defines the slice criterion. In other words, %tobool will be part of the ensuing slice, but %tobool1 will not be part of it. Explain why such is the case. Hint: perhaps you would like to take a look into the other questions. They might help you to better formalize your explanation.



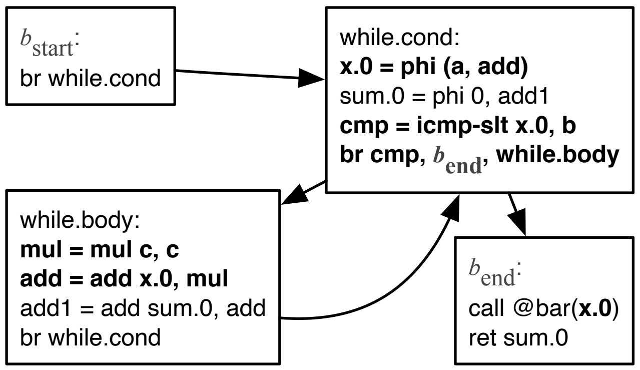
```
entry:
%a, %b, %c, %p0, %p1
%tobool = icmp-ne %p0, 0
br %tobool, %if.end, %if.then

if.then:
%mul = mul %a, %b
br %if.end

if.end:
%x.0 = phi %mul, %a,
%tobool1 = icmp-ne %p1, 0
br %tobool1, %if.end6, %if.then2

if.then2:
%add = add %x.0, %c
%tobool3 = icmp-ne %p2, 0
br %tobool3, %if.end5, %if.then4
```

Question 2 [4 Points] We would like to build slices that compute *Single Dynamic Assignments*. In other words, if F is a slice that computes the value of E in the assignment "v = E", we would like this value to be unique for the free variables in E. That amounts to say tht the assignment to v within F happens only once. Imagine that E contains two free variables, e.g., v0 and v1. Then, for any v0 and v1, the value F(v0, v1) must be unique. To understand this requirement, consider the example below:

(a) Slice criterion outside loop

```
int foo(int a, int b, int c){
  int x = a;
  int sum = 0;
  while (x < b) {
    x += c*c;
    sum += x;
  }
  bar(x);
  return sum;
}
```

(b) Control-flow graph



```
bstart:
br while.cond

while.cond:
x.0 = phi (a, add)
sum.0 = phi 0, add1
cmp = icmp-slt x.0, b
br cmp, bend, while.body

while.body:
mul = mul c, c
add = add x.0, mul
add1 = add sum.0, add
br while.cond

bend:
call @bar(x.0)
ret sum.0
```

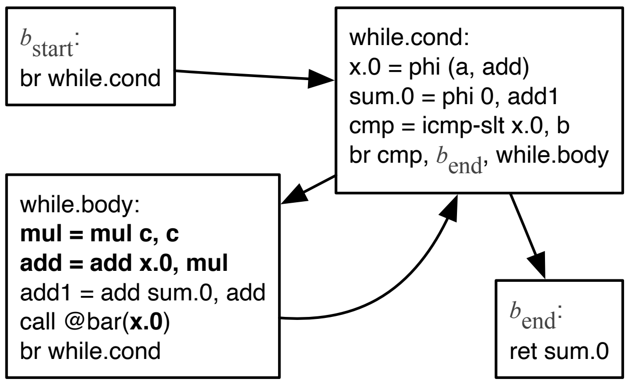
(c) Sliced program

```
int F(
      int a,
      int b,
      int c
   ) {
  int x = a;
  while (x < b) {
    x += c*c;
  }
  return x;
}
```

(d) Slice criterion inside loop

```
int foo(int a, int b, int c){
  int x = a;
  int sum = 0;
  while (x < b) {
    x += c*c;
    sum += x;
    bar(x);
  }
  return sum;
}
```

(e) Control-flow graph



```
bstart:
br while.cond

while.cond:
x.0 = phi (a, add)
sum.0 = phi 0, add1
cmp = icmp-slt x.0, b
br cmp, bend, while.body

while.body:
mul = mul c, c
add = add x.0, mul
add1 = add sum.0, add
call @bar(x.0)
br while.cond

bend:
ret sum.0
```

(f) Sliced program

```
int F(
      int x,
      int c
   ) {
  x += c*c;
  return x;
}
```

To respect the single dynamic assignment property, we ended up producing two different slices for programs in part (a) and (d) above. The slice of (d) is smaller: it includes only x.0 and c as free variables (names taken from part e). The first slice includes the whole loop (see part c), whereas the second is restricted to within the loop (see part f). In this question, you must determine the *scope* of a backward slice that ensures single dynamic assignment. That is to say: you must explain what is the maximal code region that must be included in the slice that still ensures single dynamic assignment.

Question 3 [4 Points] We shall devise an algorithm to compute the backward slice of a given slice criterion. For that, we must traverse the dependence graph of the program backwards. There are two kinds of dependencies: data and control:

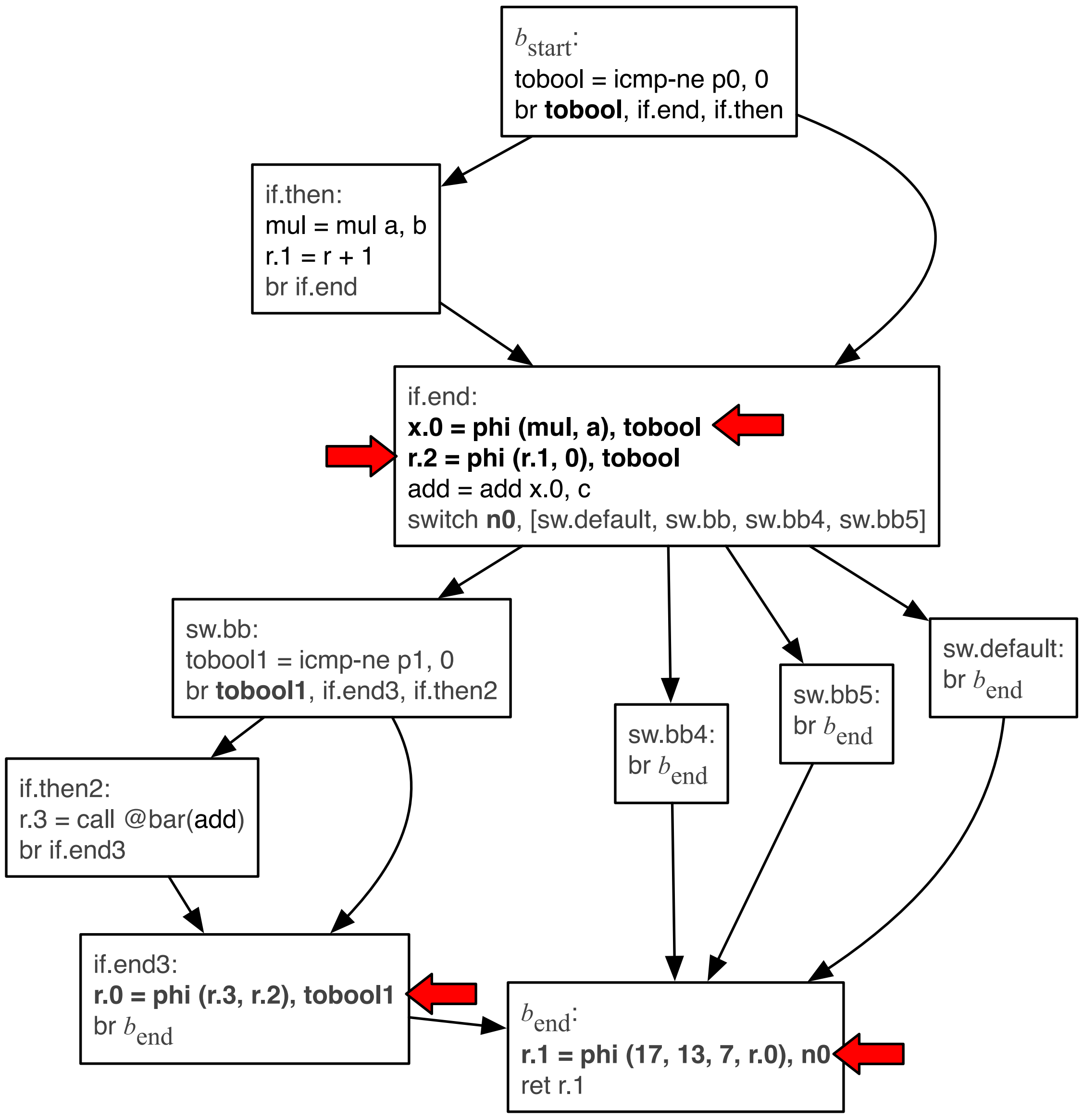
* We say that v is data-dependent on u if v is defined by an instruction that uses u.
* We say that v is control-dependent on u if the assignment to v is controlled by a branch whose outcome is determined by u.

There is a cheap trick to convert control dependencies into data dependencies: gate phi-functions. To gate a phi-function, we append the predicate that controls its incoming edges next to it. In this way, we only have to use the program syntax to compute data dependencies: everything on the left side of an assignment depends on everything on the right side of an assignment. As an example, the figure below shows the gates that have been added to the CFG of the program on the left.

(a) Original program

```
int foo(
    int a,
    int b,
    int c,
    int p0,
    int p1,
    int n0
  ) {
    int r = 0;
    int x = a;
    if (p0) {
      x = a * b;
      r = r + 1;
    }
    x += c;
    switch (n0) {
      case 1:
        if (p1) {
          r = bar(x);
        }
        break;
      case 2:
        r = 7;
        break;
      case 3:
        r = 13;
        break;
      default:
        r = 17;
        break;
    }
    return r;
}
```

(b) Gated CFG



```
b_start:
tobool = icmp-ne p0, 0
br tobool, if.end, if.then
```

```
if.then:
mul = mul a, b
r.1 = r + 1
br if.end
```

```
if.end:
x.0 = phi (mul, a), tobool
r.2 = phi (r.1, 0), tobool
add = add x.0, c
switch n0, [sw.default, sw.bb, sw.bb4, sw.bb5]
```

```
sw.bb:
tobool1 = icmp-ne p1, 0
br tobool1, if.end3, if.then2
```

```
if.then2:
r.3 = call @bar(add)
br if.end3
```

```
sw.bb4:
br b_end
```

```
sw.bb5:
br b_end
```

```
sw.default:
br b_end
```

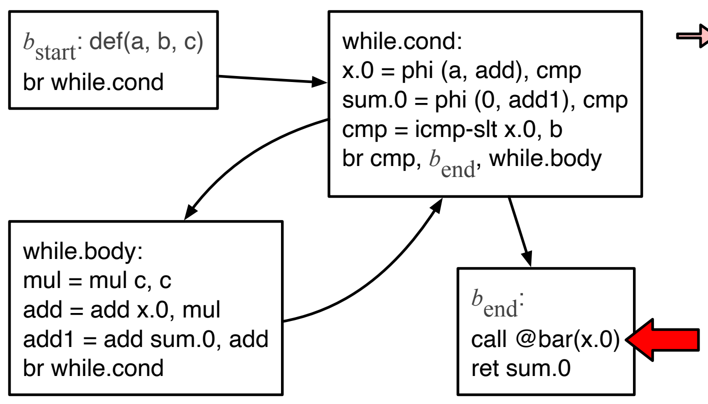
```
if.end3:
r.0 = phi (r.3, r.2), tobool1
br b_end
```

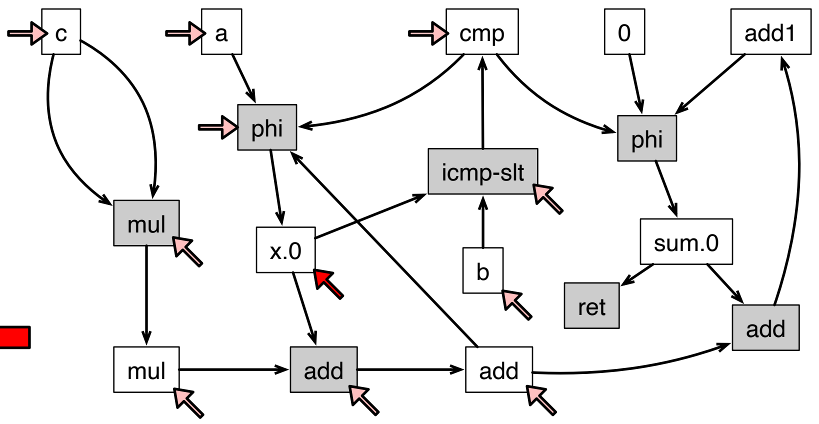
```
b_end:
r.1 = phi (17, 13, 7, r.0), n0
ret r.1
```

In this question, you must devise an algorithm to gate phi-functions. You can adopt either an imperative style (e.g., pseudocode) or a declarative style (e.g., explain which phi-functions are gated by which predicates). Notice that the same phi-function might be controlled by multiple predicates.

Question 4 [2 Point] Is it possible to have a program with a phi-function having two or more arguments whose assignment is not controlled by any predicate? If you can think on an example, draw it. Otherwise, explain why such case is not possible.

Question 5 [4 Points] Once we have a program in gated SSA form, we can build its dependence graph to find out what are the instructions and values that constitute the slice. Below we show a program, and its dependence graph, given the slicing criterion that the arrow indicates.





Design an algorithm that traverses the dependence graph to find out all the values and operations that belong into the backward slice of a given slicing criterion. In this question you must answer the following two subquestions:

(a) [2 Points] How does your algorithm stop? Remember question 3!

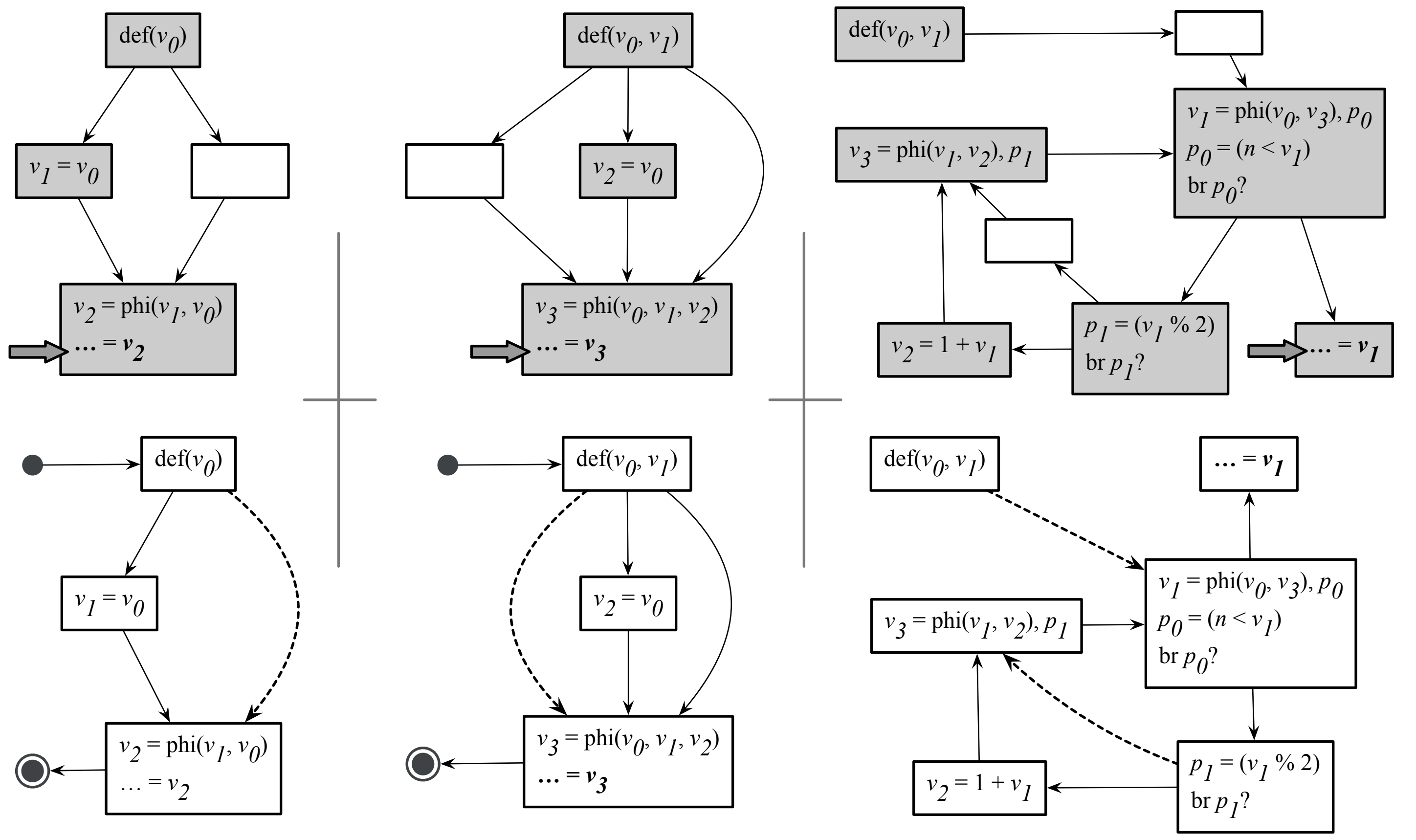
(b) [2 Points] What will be the free variables that your algorithm finds out? The free variables are the variables that will become arguments of the function F that implements the backward slice.

Question 6 [4 Point] Now, you must build a CFG that contains the instructions marked in the previous question. Notice that not all the basic blocks in the original program will be part of this CFG. You might have to add new edges to the CFG as well.

(a) [2 Points] Which edges from the original CFG will be preserved in the new CFG that implements the backward slice?

(a) [2 Points] Which new edges will be added to your new CFG?

To help you out, we have a few examples of slices that have been produced (in gray) and the ensuing CFGs that are created from these slices (on the bottom of the figure). The dashed edges are new.



def(v_0)
$v_1 = v_0$
v_2 = phi(v_1, v_0)
… = v_2
def(v_0, v_1)
$v_2 = v_0$
v_3 = phi(v_0, v_1, v_2)
… = v_3
def(v_0, v_1)
v_1 = phi(v_0, v_3), p_0
$p_0 = (n < v_1)$
br p_0?
v_3 = phi(v_1, v_2), p_1
$v_2 = 1 + v_1$
$p_1 = (v_1$ % 2)
br p_1?
… = v_1