ECE654 Final Project Review

Huijie Chu* Shiyun Qin†

ABSTRACT

In 1970, Frances E. Allen proposed the idea of control flow graph(CFG). It is a graphical representation of the flow of the computation during the program execution. Since it represents the flow of logic, it can be used for static analysis or compiler optimization, etc. In this project, my team aims to build a static control flow graph generator from scratch using the concept of an abstract syntax tree. The essential part of the program will be written in Python with the assistance of Graphviz.

he main idea of this project is to implement a control graph generator with a foundation build upon the concept of an abstract syntax tree. The project starts off using the python AST package as the parser to identify and parse the input program, which is a piece of read-in code written in python. Processed the code is then stored in the format of an abstract syntax tree, which essentially is a tree-like structure that contains partitioned statements along with their associated conditions and expressions. The control flow graph is generated by recursively iterate through the processed abstract syntax tree. Graphize is used as the drawing tool to create the final graph representation. Compared with the previous implementation, a lot more details were implemented in our code, such as list comprehension and try catch functions.

1 INTRODUCTION AND OVERVIEW

One of the initial purposes of this project is to simplify the time consumption on the preparatory work phase when a new piece of code is handed for understanding. Start by partition a program into an independent or a few aggregated functions. Feed aggregated function bundles respectively into the control graph generator enable readers to visualize the flow of each function, along with their internal logical conditions, statements, and expressions. A control flow graph is a graphical representation that consolidates statements as notation, split and link adjacent notations with edges that each epitomize a path with a correlated logical condition that might be traversed through during the execution phase. It refers to the order in which the individual statements, instructions, or function calls of an imperative or functional program are executed or evaluated.

Furthermore, a comprehensive control graph is essential in the phase of error detection. A missing or contradictory logical condition can create a bug that is laborious to fix. Without a thorough understanding and meticulously go through the logical transition in the program, the bug is arduous to trace. Paths on a control graph, along with their statements, instructions, or function calls, are visually represented. Thus, it is straightforward for a tester to go through suspicious paths without being anxious about getting lost in the flow of logic.

*e-mail: h25chu@uwaterloo.ca
†e-mail:s45qin@uwaterloo.ca

2 IMPLEMENTATION

2.1 Preprocessing

For the purpose of this project, which is to accurately generate a exhaustive control flow graph, there is no point to analyze codes with syntax errors. For the initial set up, the first step is to read and compile the source code. Error code that fails in the preparation step will be returned immediately after.

Most of the informal coding style of the input will not affect the output of ast module since it analyzes the code in a deeper level. As an example. consider the code below.

```
a=1; b = 1; c =\
1
d = [1,
2,
3]
```

As shown in the example, the Line separation symbol used above, \ or equivalent, may seem to be a strange usage, whereas it does not conduct any syntax error which may prevent compile of the program. The more commonly adapted programming habit is demonstrated below.

```
a = 1
b = 1
c = 1
d = [1, 2, 3]
```

In the above example, the spaces between variables and other variables/constants are ignored, the backslash to connect multiline codes are ignored and the brackets/parentheses that expands multiple lines are ignored. Also, the semicolons that enable multiple statements in one line are omitted. In order words, the compiler does not tend to standardize different programming styles. As long as it successfully passes the compilation. Typos, any symbols or usages from another coding language will be detected and handled as error behavior.

For anothe example, take a look at the code below.

```
a = 1 # this is a comment
```

The previous code is identical with the one below.

```
a = 1
```

In line comments that immediately follow the code part of the implementation will be read and ignored in the compiler. As an exception, multiline comments(doc strings), which are quoted between a pair of three single or double quotes are not ignored by python interpreter. They are treated as multi-line strings but assigned to any variable for display only. If the source code contains lots of doc strings, the control flow graph will be redundant and confusing.

Hence, the second step is to remove the these multiline comments. Using python's tokenize module, we only keep a string token if its previous token is not indentation, is not a new line and the token itself is not the first token in a module. In the context

below, standarized input stands for code blocks that can compile without raising any syntax error

2.2 AST Nodes

The standardized input source code is then parsed by the python AST module. AST stands for an abstract syntax tree. Therefore, a tree-like structure is generated after parsing, which means the result is either a basic subclass node of ast.AST, e.g. ast.Name, ast.Num, ast.Eq etc., or a recursive structure (an ast node inside a basic subclass node of ast.AST), e.g. ast.Delete(targets = [ast.Name(id='a', ctx=ast.Del())]).

By recursively visiting and processing the parsed result, we can build the control flow graph of the input source code. Since each subclass of ast.AST has its structure/variables, different visiting methods are required. Those methods will be elucidated in the following subsections.

2.3 Models

Class BasicBlock is the supermodule that every node in the control flow graph inherits from. Each basic block structure contains a unique block id, two lists of block id to represent the previous and the neighboring block(blocks), a list to store the functions that are called inside the block, and a list to store the statements inside the block.

Each block is assigned with a unique id. To generate unique block id for new blocks and to prevent from using numerous global variables, a singleton class BlockId is used. The singleton class will keep track of every existing blockid and next available block id in case of a new block is generated. Each time a new block id is to generate, the same BlockId instance is called.

Class CFG is the highest level representation of the control flow graph. Every function 'def' in the input code will be treated as an individual CFG. The CFG class keeps track of every block in the graph using a dictionary with the format of *blockid* $\rightarrow$ *basicblock*. It also keeps track of every edge in the graph using a dictionary in the format of *(fromID, toID)* $\rightarrow$ *condition of the edge*. Finally, it contains another dictionary to keep track of function calls in a function in the format of *function name* $\rightarrow$ *CFG*.

Class CFGVisiter, which is a subclass from ast.NodeVisitor is the main part of this project. According to the documentation of ast, ast.NodeVisitor is meant to be subclassed and we can define custom visit methods in the subclass. For example, if an ast node is of type *If*, its custom visit method should be defined in method *visit_If*. All visit functions must follow the conventional function naming style as required for the ast to recognize such function name. In the case, if there does not exist a customized visit method for a certain kind of node, this node will be visited using *generic_visit* method. Below is the implementation of those custom visiting methods.

2.4 visit_*

Some type of visit functions, commonly seen in a function of variable assignation, that generate leaf nodes. According to its simple structure, there will not be another new control flows generated inside the node. For example, the node of type augmentation assign, in which the value can only be Name(a += b), Constant(a += 1), Subscript(a += b[0]) or Attributes(a += c.val), will not lead to two or more successors in the CFG. For these kinds of nodes, we only have to add them to the current basic block and then move on. Those types of nodes are not mentioned below.

2.4.1 Assign

Normally, the value of ast.Assign class is identical with ast.AugAssign, which means we only have to add the assign statement to the current block. Whereas in special cases, in which the value is a list of comprehension, dictionary comprehension, set comprehension, generator expression, or lambda expression, we would tag a corresponding flag to those special nodes and recognize its type. Ensuring later in handling the corresponding visit method, e.g. *visit_ListComp*, we can get assertive information from the flag raised and perform the corresponding operation accordingly. (Section 2.4.6 demonstrate this point as an example in the variable assignment of a list comprehension)

2.4.2 Call

Except for the special case under the class of lambda function((lambda x: x+1)(1)), which will be expressed in a further detailed section below, we append the name of the function being called to a list in class CFG. Thus, we can show what functions are called inside the current function as needed.

2.4.3 For

For the ast.For class, there will be an arrow(edge) pointing backwards to indicate loops. So, a new block, we called it *h1*, must be added to be the head of that arrow. A *for_block* is created for the body part of the For statement. And an *after_for_block* is created for codes after the For statement. We know that there is an edge from *h1* to *for_block* and an edge from *for_block* to *h1*. And when the For loop is over, there will be an edge from *h1* to the *after_for_block*. Since there may be nested loops, we use a stack to keep track of the loop chain in order to identify a current block and the reciprocal edges in relation with it. In Python, both While and For have an Else section, this section will be executed when the test condition becomes false. Basically, it is just like the code after the While or For block except for the condition when there is a Break inside the loop. When the Break is executed, the code inside the Else section will be ignored. In our implementation, an extra block is added for Else section in For or While. Figure 3c shows a CFG of a For-Else statement.

2.4.4 If

The conditional statement is treated as a normal statement, and simply append it to the current block. There are at least two blocks generated from a statement: one for the statements in the context of the if block, and one for statements after the whole If expressions. And if there is an Else in the If expression, there should be the third block to handle statements inside the Else segment. We do not have to consider Elif because, in AST, all the logic is binary. A frequently implemented If-Elif-Else structure is treated as an If statement inside another If statement with their correlated else segment.

2.4.5 IfExp

IfExp is the expression like

```
2 * x if a > 0 else 3 * x
```

It is converted to a simple If-Else statement in our implementation if it is inside a Lambda function, otherwise we keep it as its original form.

2.4.6 Lambda

Since lambda function is also a function call, we convert it to a function definition by creating a new ast.FunctionDef class based on the lambda flag. So, the lambda function can be shown as an exclusive CFG but is displayed on the same graph subject to its principle measure. Figure 7a shows the CFG of a lambda function, which is transferred to a FunctionDef and plotted in a separated graph.

2.4.7 List Comprehension

List Comprehension can be splited and conducted as appending to a list in multiple for loops and if statements. For example:

```
a = [2*x for x in y if x > 0 for y in z if len(y) > 3]
```

The expression above is the same as the implementation written below.

```
a = []
for y in z:
    if len(y) > 3:
        for x in y:
            if x > 0:
                a.append(2*x)
```

So, we recursively construct a for loop if we confirm the list comprehension flag is set. The base case is that the generator field inside the list comprehension node starts as an empty list, under which condition we iteratively construct an append(). The recursive case is that the generator field is not empty, under which condition we construct a For statement(and an optional If statement) and make it the body of the outer level node. In the same way, set comprehension and dictionary comprehension can be constructed. The only difference is implemented within the base case. Thus, we assume their concept is already covered in this section. Figure 6a shows the CFG of a list comprehension, which is transferred to a nested for loop.

2.4.8 Try

We want the CFG to show the exception handling for each type of error. The format of the graph is similar to that of the For statement – after handling each type of error, there will always be an edge pointing backwards to the end of the Try statement. And finally an edge pointing outwards from the end of the Try statement indicates the control flow resumes. The actual implementation is slightly more complex because we have to consider multiple scenario – unique combination from potential elements Try, Except, Else, and Finally. Figure 5c shows the CFG of a Try-Catch-Else-Finally statement.

2.4.9 While

In substance, While statements are another form of loop statement. The implementation is essentially identical for the two types of loops, and we will not expand the section further.

2.4.10 Logic Inversion

In the control flow graph, it is important to show the condition when the program traversal through paths. Sometimes the condition is not intuitive like the condition of an Else statement. Based on different types of conditional statement, we proposed different methods to invert the logical condition to demonstrate the accurate condition, as shown on the edge in the CFG, when the program exits the loop or introduce the Else path. We use a dictionary to store the inversion of different comparators, e.g. ast.Gt() $\rightarrow$ ast.LtE(). If the condition node is a comparator, we return the dictionary lookup result. If the condition node is True/-False, we return the opposite. If the condition node is a boolean formula, e.g. $a > 1 \wedge b < 1 \vee c == 0$, we flip it using De Morgan's law. For comparisons like $a < b < c > d$, we splitted it into multiple single comparisons connected with *And*, and reverse it like a boolean formula. For other convoluted conditions but scarce practical value, we just add a Not in front of the condition and return it.

2.5 Plotting

As mentioned in the abstract session, Graphviz is used to plot the CFG after traversing the entire AST is complete. CFG is represented by a DiGraph object in Graphviz. Using a set to record the visited nodes, we can build a directed graph from the starting basic block and eliminate duplication.

3 Unit Testing

3.1 Blockid

We performed unit testing on two of the essential class, Blockid, and Basicblock. Each Basicblock associated with a bid (i.e. a unique id), and it interprets and stores information retrieved from an AST model. Because, some input code may generate more than one AST model, each would generate a control flow graph aggregated into one output file, we need a class Blockid, which is a singleton, to keep track of id and ensure there will not be any repeated ones. Therefore, it is critical to perform a unit test on class Blockid for the entire project is build upon it. Although simple, any mistake can cause a severe breakdown.

3.2 Basicblock

As introduced in the previous section, we can see this class serves as the backbone of the project. From an AST to a control graph, all class structure, function calls, statements, expression, and variables declaration are eventually represented as blocks. Class Basicblock contains not only information about itself, but also the adjacent blocks and its relationship with them. Thus, we performed unit test one the following function within the class to ensure code quality.

1. is_empty: Since there is no guarantee that the input file will contain any valid code to show a cfg. Initially, a new block is first created before we can know whether there will be any sufficient information to be put into the block or not. Image a class structure with a purely variable declaration. Since that is no splitting in the program, all variable declarations are sufficient to be stored in the same block with the class name. Thus this function is used to remove empty blocks and prevent them from being shown on the output.

2. has_next: The AST parses static python code. Picture a simple tree structure with nodes and leaves down to different branches. Therefore, it is important to different leaf with the rest of the nodes on the path.

3. has_previous: This function is nearly identical to the last class. Both of the functions are frequently used to handle a loop call.

4. remove_from_prev and remove_from_next: These two function is used in correlation with the previous functions. When we detect an empty block, we would first identify

whether it is the previous or the next block of the current block, and remove it from the chain of blocks.

5. stmts_to_code and calls_to_code: In an AST, input python code is read and stored as either a statement of a call. A CFG is built from the information given in the AST, but simply output strings retrieved is not enough to correctly build a CFG. In another word, we need a way to translate information retrieved from AST to mimic a comparable python syntax.

Unit tests on these two classes serve as concrete to ensure the correctness of fundamental functionalities on building each block. The output control flow graph is created by traversal through the blocks chain. Starting from the root node, we connect adjacent blocks and adds proposition logical expression that determines the path on the edge between each node correspondingly. Since there is a guarantee that the information in the AST is correctly stored inside each BasicBlock, we can visually exam whether the expressions, which are shown on the edge between nodes, on the output graph are in the logical equivalent form with the input code. We did not choose to perform a unit test on everything function involved in this project to reduce code redundancy. Only the essential functions that have no other way to directly check its correctness on the output are mentioned above are included in the unit test. In order words, we performed a unit test on functions that we can now do a showcase on the presentation to prove the correctness is checked by the unit test.

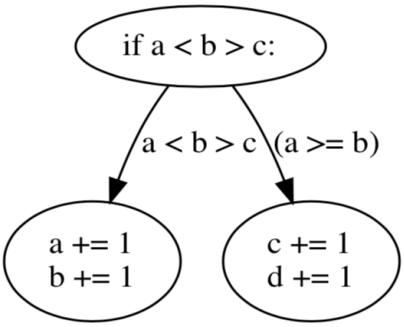
4 RESULTS

In this part, we make comparison between the original and our implementation of the CFG. Due to the limitation in length of this report, **only the improvements** are shown, the basic function will be shown in the presentation.

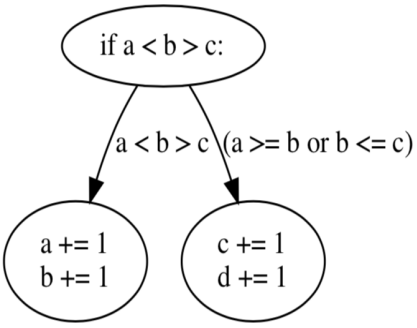
4.1 Logic Inversion

```
1  if a < b > c:
2      a += 1
3      b += 1
4  else:
5      c += 1
6      d += 1
```

(a) Code



(b) Before

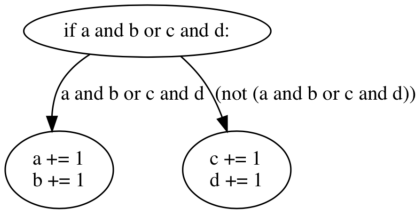


(c) After

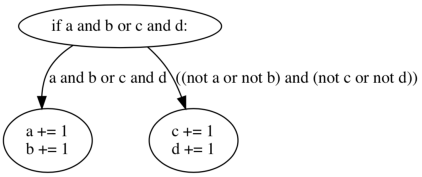
Figure 1: Fixed Logic Inversion Bug

```
if(a and b) or (c and d):
    a += 1
    b += 1
else:
    c += 1
    d += 1
```

(a) Code



(b) Before



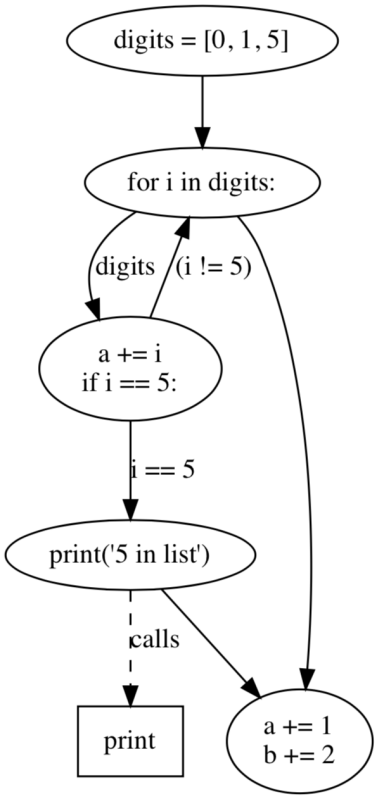
(c) After

Figure 2: Inversed Logic Using De Morgan's Law

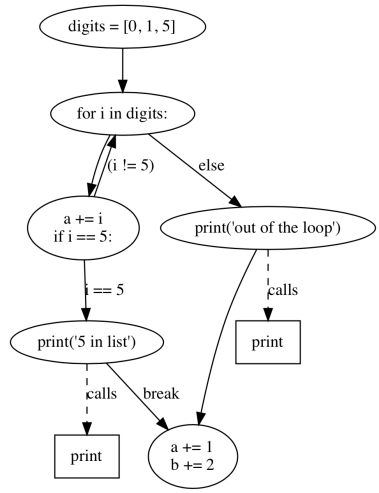
4.2 For Loop

```
digits = [0, 1, 5]

for i in digits:
    a += i
    if i == 5:
        print("5 in list")
        break
else:
    print("out of the loop")

a += 1
b += 2
```

(a) Code



(b) Before



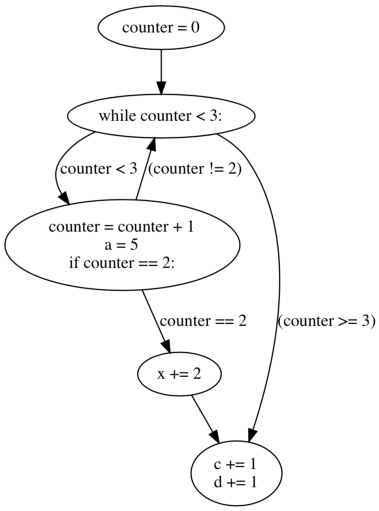
(c) After

Figure 3: Showed Else part and the Correct Logic in For Loop

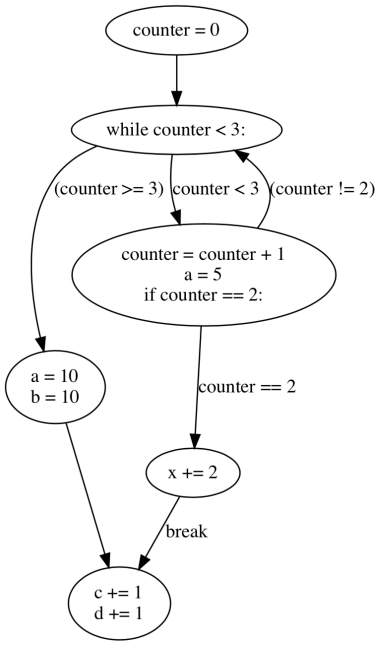
4.3 While Loop

```
counter = 0

while counter < 3:
    counter = counter + 1
    a = 5
    if counter == 2:
        x += 2
        break
else:
    a = 10
    b = 10

c += 1
d += 1
```

(a) Code



(b) Before



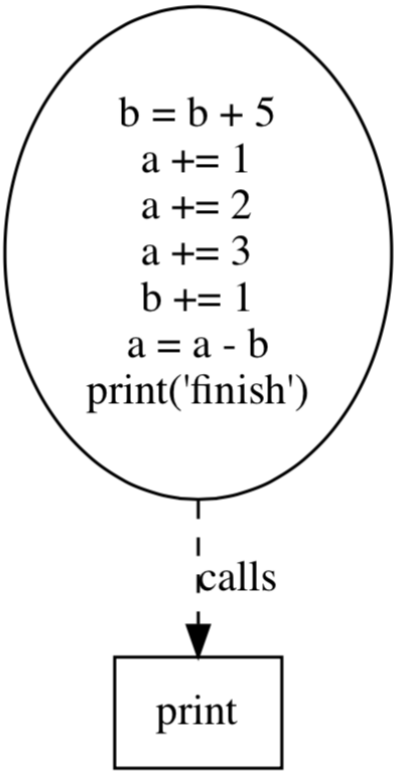
(c) After

Figure 4: Showed Else part and the Correct Logic in While Loop

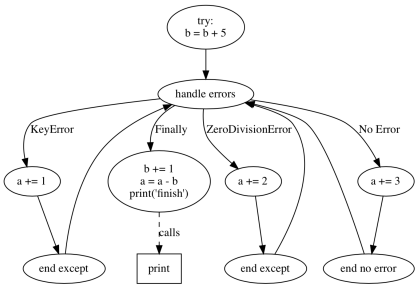
4.4 Try

```
1  try:
2      b = b + 5
3  except KeyError:
4      a += 1
5  except ZeroDivisionError:
6      a += 2
7  else:
8      a += 3
9  finally:
10     b += 1
11     a = a - b
12     print('finish')
```

(a) Code



(b) Before



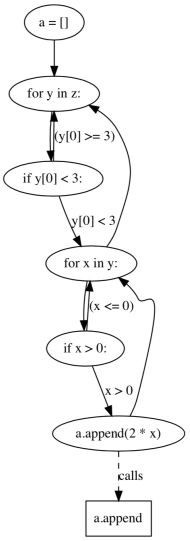
(c) After

Figure 5: Added Support for Try

4.5 Comprehensions

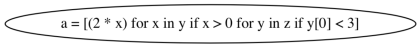
The original implementation will show list/dict/set comprehensions directly. For code:

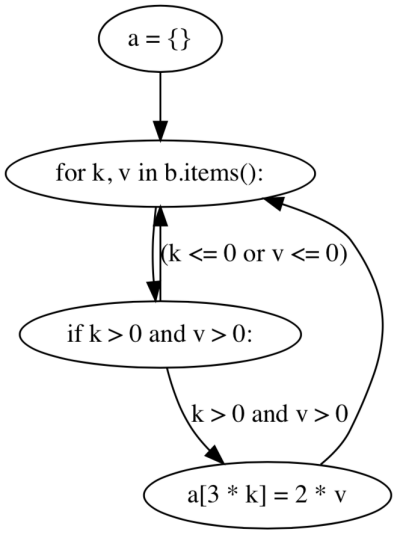
```
a = [2 * x for x in y if x > 0 for y in z if y[0] < 3]
a = {3 * k : 2*v for (k, v) in b.items() if k > 0 if v > 0}
a = {6-x for x in y if x > 5 for y in [1,2,3,4] if y==3}
```



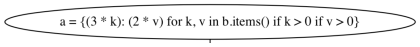
(a) List Comprehension

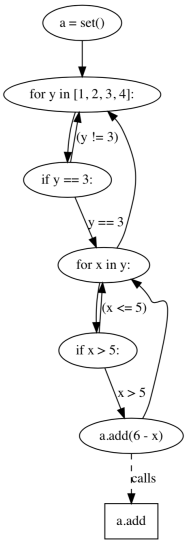
The output will be:





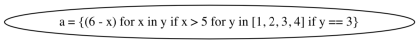
(b) Dict Comprehension





(c) Set Comprehension

Figure 6: Added Support for Comprehensions

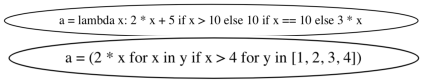


4.6 Lambda Expression and Generator Expression

The original implementation will show lambda expression and generator expression directly. For code:

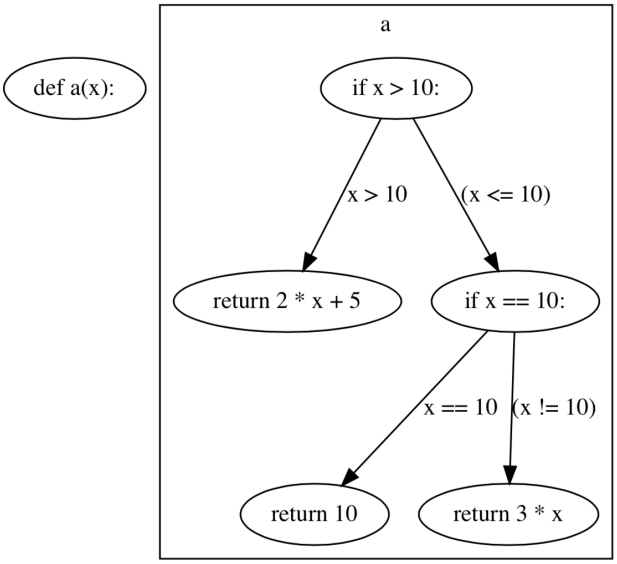
```
a = lambda x: 2 * x + 5 \n
    if x > 10 else 10 \n
    if x == 10 else 3 * x
a = (2*x for x in y if x > 4 \n
    for y in [1,2,3,4])
```

The output will be:

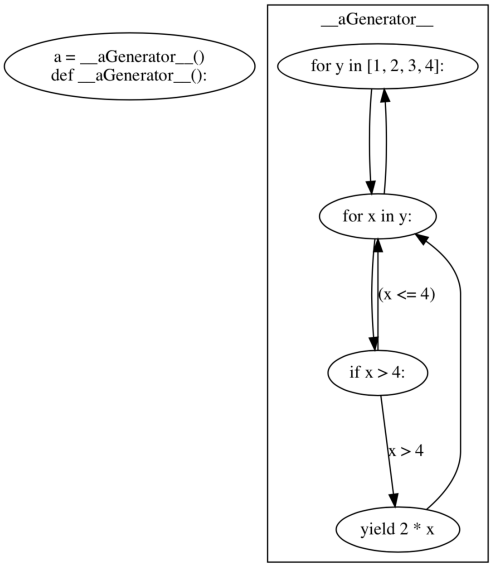


After the implementation, the outputs are:

After the implementation a separate function is defined. The outputs are:



(a) Lambda Expression



(b) Generator Expression

Figure 7: Added Support for Lambda

5 CONCLUSION

This report summarized the design and implementation of a control flow graph generator and briefly discussed its testing method. Comparing to other CFG generator, this one focuses on details like the visualization of the lambda function, try catch methods, list/set/dict comprehensions and special usage in While and For loops, etc. It serves as a great supplement for other CFG generators to generate graphs for special cases.