COL719 Assignment 01

Abstract Syntax Tree and Data Flow Graph Construction

Jaskaran Singh Bhalla (2021TT11139)
Madhav Manish Gulati (2021EE10139)

February 5, 2024

1 Background

During the synthesis procedure of digital systems, in the initial stages, there is a step where the mathematical operations need to be converted into an appropriate data structure (graphs or trees).

This is done so that the flow of data (operators and operands) becomes machine-friendly and can be mapped to clock cycles during the scheduling step. Moreover, during parsing, the grammar rules are in the form of a tree data structure.

Further, when the hardware circuit is being designed, the circuits can be modelled as graphs as well, so software-to-hardware synthesis becomes easier.

2 Objective

The code takes input in the form of a text file that contains multiple lines; each line is a statement with an assignment operator, with the LHS as one variable, the RHS as a combination of variables and operators, and the four mathematical operators (+,-,*,/).

The code first returns the AST of each statement as the output, followed by the DFG of the entire statements combined.

The DFG has a '=' at the root and captures dependencies and reassignments of variables. The edges of the DFG has variable names.

3 Format of the AST

For the sake of conformity in the code, the AST for every statement is being synthesised in the form of a 3-member nested array representation for a tree. Every array is of the form [left child, node, right child]. The node is a mathematical operator (+,-,*,/). The root node is a '='.

The left child and right child may point to other nodes as well, in which the nodes further have children. In that case, the left child and right child are arrays nested inside the array of their parent node. Otherwise, they are discrete elements (variable names)

For example, the tree represented as [[["1"],["2"],["3"]],["4"],["5"]] has 4 as the root, with 5 as the right chlid, and 2 as the left child. 2 further has 1 as its left child and 3 as the right child.

The AST for every line appears in a dictionary format in the output, where the key is the line number and the value contains the AST in the nested array format.

4 Format of the DFG generated by the code

The tree has either mathematical operators (+,-,*,/,=) or read/write at the nodes. The edges contain the variable names corresponding to the read/write operators. Though the mathematical operators are also mapped to numbers (eg. "1+", "2+" etc.), the number associated with the operator is omitted in the graph diagram.

The edges between an operator node and read/write node contains the variable name (indexed) i.e. "a1" written on the edge means that read/write operation is being performed on the variable "a" for the first time, "a2" means that read/write operation is being performed on the variable "a" is being performed for the second time and so on. Every edge is directed.

If there is a "= - =" linkage with a variable name on the edge, that means that there is a dependency between more than one statement (i.e. the variable on the LHS of an earlier statement appears in the RHS of another statement).

If there is no dependency between statements, then we get a forest of trees with non-connected components.

The spatial arrangement of the trees is controlled by a seed variable which is randomly generated and library controlled. We can refresh the program multiple times till we don't get the desired arrangement of the nodes, that is clearly and neatly visible.

Furthermore, the DFG for the entire statement is generated in an adjacency list format, in which every element (along with read/write - if the node has "read" and edge has "a1", then the element would be taken as "read-a1") is mapped to a key value pair, and the adjacency list is generated in terms of the keys and a key-value map is also generated as output.

The leftmost element of each line the adjacency list represents the node in picture, while the other elements represent the other nodes connected to it, with the arrow head pointing towards them and the arrow tail pointing towards the node in picture.

All the constant values have been considered as a node. Note, we are considering constant to be a numeric value only.

5 Working of the code - an overview

In this code, the input given is a text file, which contains multiple lines, each line having an LHS (the variable which is being written), the RHS (a sequence of mathematical operators: '+','-','*','/', and variables which are being read), both separated by a '=' sign.

The code first reads every line of the text file, and the line is converted into an array. All such arrays are stored into a dictionary.

We have used a recursion based algorithm for this, where we store array of arrays or dictionaries. Note that we are using dictionaries to store the dependencies within multiple lines of abstract syntax tree. The precedence of the operators has been chosen by using the if else statements and making recursion calls accordingly for the construction of ASTs The dictionary of ASTs is then generated, with every AST in the nested array format (because converting into this format looks quite easy as compared to other formats, and conversion of this format into a tree, top to bottom in terms of tree levels, and out to in, in terms of the array brackets.)

After generating the AST, the ASTs are stored into a file, so that we can read them directly during DFG construction. Then, the code moves towards the DFG construction. Every operator is assigned a number ("1+", "2+" etc.) when ever an operator is added to the graph. We are using networkx and matplot lib for creating and storing graphs and working with them. While constructing the DFG, the AST's are read from our previous construction stored in a variable and combined, one by one. Also, the operators with numbers, and read/write operations along with their corresponding variables (after the variables are assigned a number based on the number of read/write operations called on them) are given keys. The DFG adjacency list is constructed using the keys of the variables. We have implemented special logic for taking into account dependency and re-dependency of the variables they are written once. As mentioned by the teaching

assistant we have handled WAR/WAR issues using the variable naming and have implemented the logic to chose the correct variable out of multiple variables with same name whenever it becomes a dependency

6 Code Structure

The code consists of a number of functions and blocks, each having a different purpose. The tasks performed by these functions and blocks are described below:

7 Code Explanation

7.1 Packages/Libraries Imported

- **import glob**: Enables file path expansion using the `glob` module. It is basically used to read all the files in a given directory, matching with a keyword and file type.
- **import json**: Provides functionality to handle data.
- **import networkx as nx**: Used for creating and manipulating graphs and networks.
- **import matplotlib.pyplot as plt**: Facilitates the creation of plots and visualizations,
- **import random**: Allows the generation of random numbers. The random numbers are used to assign random seeds, that control the spatial arrangement of the DFG plotted using matplotlib.

7.2 Global Dependencies

- **statements**: An array storing input statements read from the test case file. If the statement has both LHS and RHS, it gets stored in the array.
- **ast_ll**: A dictionary storing Abstract Syntax Trees (ASTs) for each input statement.
- **op_count**: A dictionary keeping count of different operators ('=', '+', '-', '*', '/').
- **var_count**: A dictionary tracking the occurrence count of each variable.
- **curr_dependency**: A dictionary tracking the dependencies between variables and their line numbers in the AST. Later on it is used for tracking the latest occurrence of a variable name.
- **G**: A directed graph representing the Data Flow Graph (DFG).
- **id_map**: A mapping of node IDs to their corresponding labels in the graph.

7.3 Reading Inputs

- **read_inputs(file)**: Reads input statements from a specified file using the `glob` module and populates the `statements` array.

7.4 Create ASTs for All Statements

- **process_asts()**: Iterates through input statements, constructing Abstract Syntax Trees (ASTs) and updating the global dictionary accordingly.
- **construct_ast(rhs)**: Recursively constructs an AST for the given right-hand side expression, handling variables and dependencies. This is actually the helper function, which constructs ASTs in the 3-member array format as mentioned above in the report.

7.5 Print AST as Array of Arrays

- **print_write_ast()**: Writes the constructed ASTs to an output file in for later reference. W

7.6 Operator Count

- **operator_count(input, count)**: Increments the count of the operators in the `op_count` dictionary and returns a string with the updated count.

7.7 Process DFG

- **create_dfg()**: Constructs the Data Flow Graph (DFG) based on the ASTs, creating nodes and edges for each operation and variable.

7.8 Construct DFG

- **construct_dfg(AST, parent_id, op_count)**: Recursively constructs the Data Flow Graph (DFG) for a single AST, based on the array representation, connecting nodes, and updating the global variable that stores the graph.

7.9 Print and Write DFG

- **print_write_dfg()**: Visualizes the DFG using `matplotlib` and saves the graph, edge labels, and a key map to files. It also writes the graph's weighted edge list to a text file.

7.10 Main Function

- **Main**: The entry point of the program, that executes the sequence of function calls to read inputs, process ASTs, create DFGs, and output results.

8 Test cases

We run the code on 19 test cases, to check the correctness of the code, if the operators and operands are shown correctly, if the dependencies are captured, and if the mapping is done correctly.

The test cases and the output of the code for each test case, is shown below. The AST for every line, the key mapping, the DFG adjacency list format, and the DFG graph diagram is shown below.

8.1 The following checks are done via the test cases:

1. Check for single constant assignment
2. Check for single variable assignment
3. Check for multiple constant assignment
4. Check for multiple variable assignment
5. Check for same variable read once and assigned later
6. Check for two reads of the same variable
7. Check for two different connected components in the graph
8. Check for single dependency
9. Check for single dependency and read
10. Check for multiple dependencies
11. Check for reading a variable and assigning a different value to that variable during operation
12. Checking for multiple dependency of same variable
13. Check for the constants only expression
14. Check for complex statements
15. Check for some corner and abstract cases

    Please note that some things like 'a = a' and others have been ignored by us as we don't know how they can be handled.

9 Running the code on

In order to run the code, please create an "input.txt" file in the same folder as main.py file and add your test case to input.txt and run the code by using the command "python3 main.py"

COL719 Assignment 01 Test Case Analysis

Test Case 0: Constant Assignment to a variable

[1]

Input:

main.py | input.txt

A1 > testcases > test00 > input.txt

```
1  a = 1
```

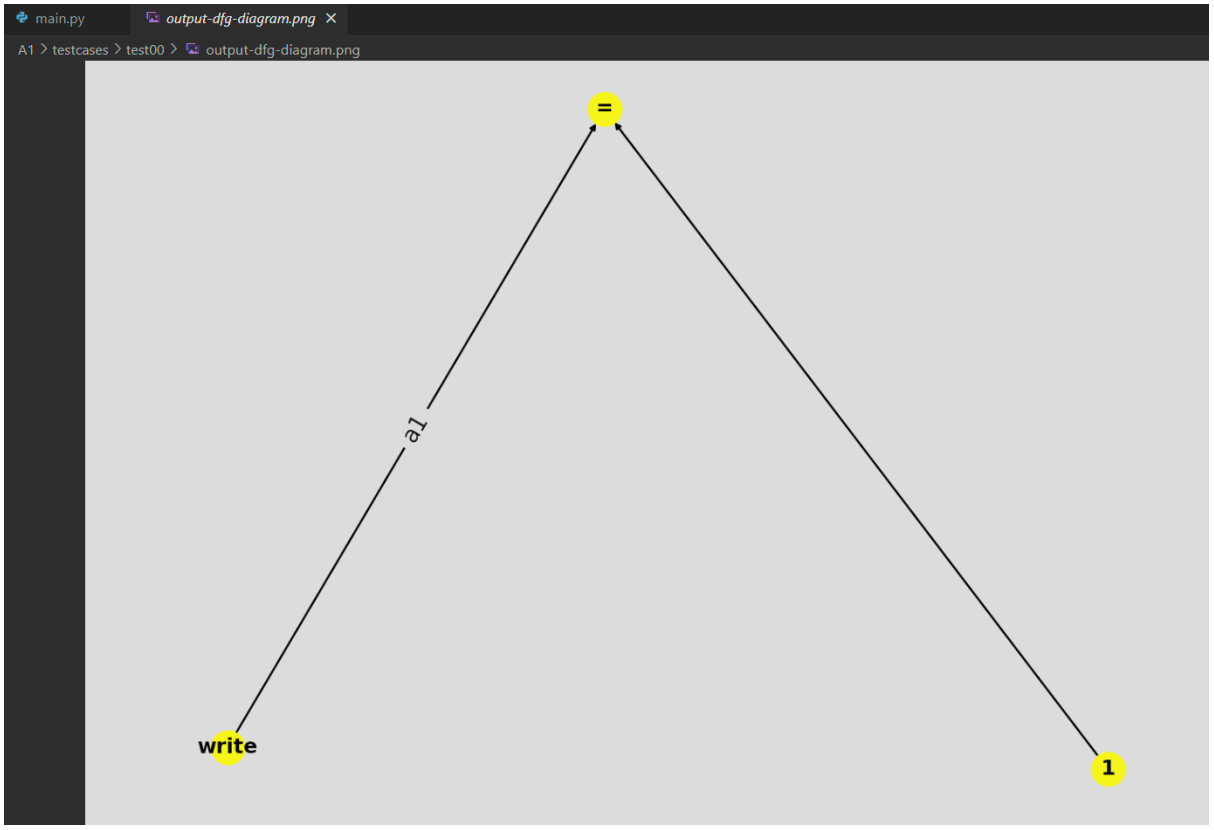
AST dictionary:

main.py | output-ast.txt

A1 > testcases > test00 > output-ast.txt

```
1  {"0": [["a"], ["="], ["1"]]}
```

DFG Diagram:



[1]

Key Map:

main.py | output-keymap.txt

A1 > testcases > test00 > output-keymap.txt

```
1  {"1=": 4638206448, "write-a1": 4642405424, "1": 4298689840}
```

DFG Adjacency list

main.py | output-dfg.txt

A1 > testcases > test00 > output-dfg.txt

```
1  4642405424 4638206448
2  4298689840 4638206448
3
```

Test Case 1: Assigning a variable to another variable

Input:

main.py | input.txt

A1 > testcases > test01 > input.txt

```
1  a = b
```

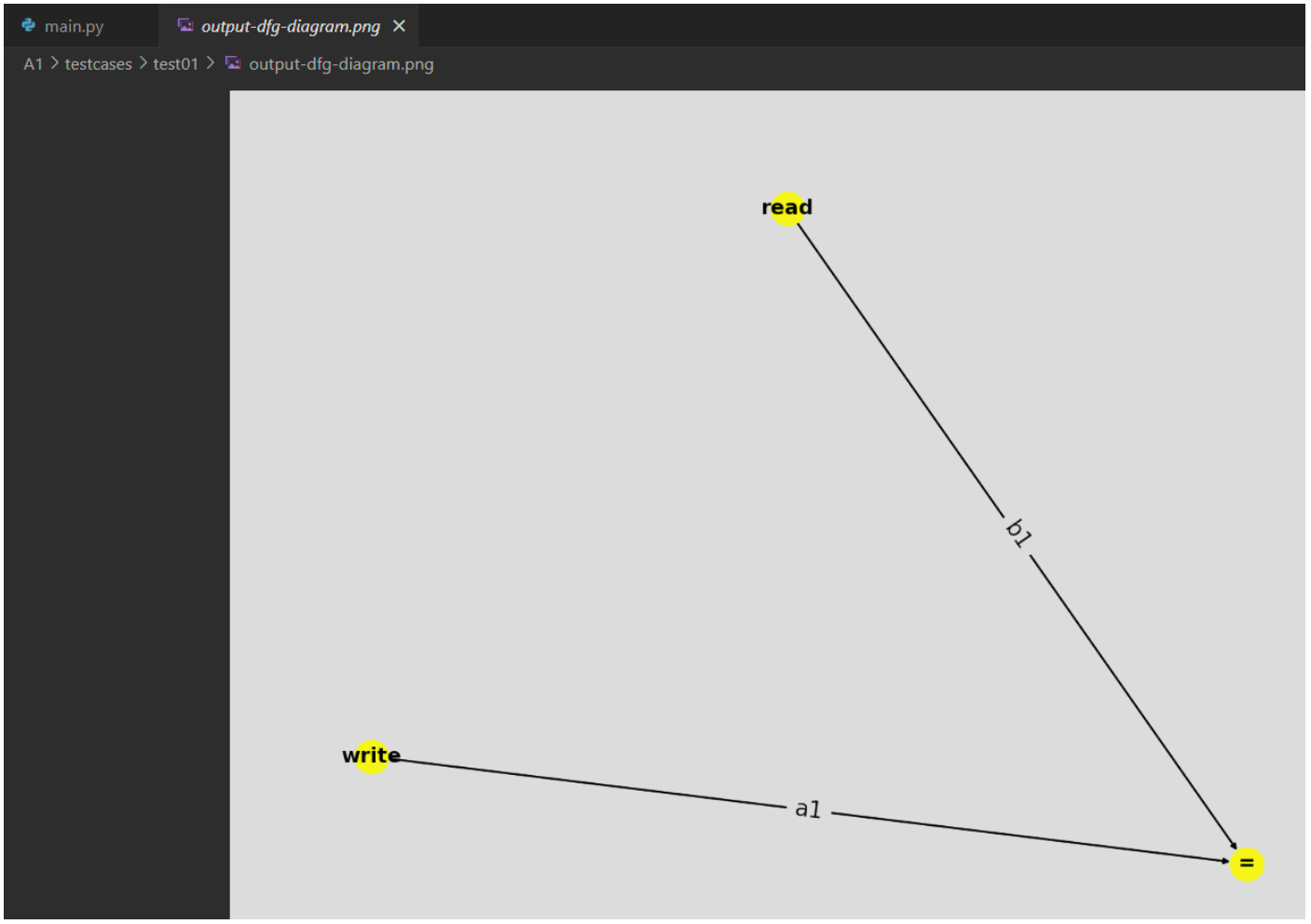
AST dictionary:

main.py | output-ast.txt

A1 > testcases > test01 > output-ast.txt

```
1  {"0": [["a"], ["="], ["b"]]}
```

DFG Diagram:



Key Map:

main.py | output-keymap.txt

A1 > testcases > test01 > output-keymap.txt

```
1  {"1=": 4809248560, "write-a1": 4554072368, "read-b1": 4554072240}
```

DFG Adjacency list

main.py | output-dfg.txt

A1 > testcases > test01 > output-dfg.txt

```
1  4554072368 4809248560
2  4554072240 4809248560
3
```

Test Case 2: Adding 2 constants and writing the value to a variable

Input:

main.py | input.txt

A1 > testcases > test02 > input.txt

```
1  a = 2 + 3
```

AST dictionary:

main.py | output-ast.txt

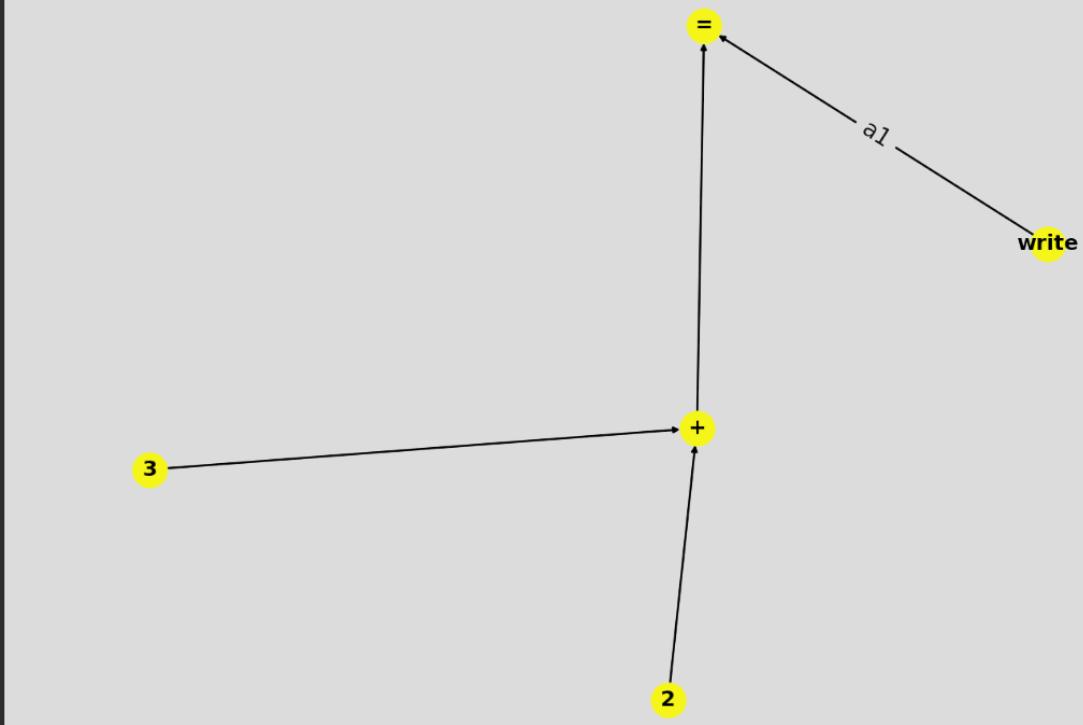
A1 > testcases > test02 > output-ast.txt

```
1  {"0": [["a"], ["="], [["2"], ["+"], ["3"]]]}
```

DFG diagram:

main.py | output-dfg-diagram.png

A1 > testcases > test02 > output-dfg-diagram.png



Key Map:

main.py | output-keymap.txt

A1 > testcases > test02 > output-keymap.txt

```
1  {"1=": 4652294192, "write-a1": 4650988912, "1+": 4651083632, "2": 4306244208, "3": 4306244272}
```

DFG Adjacency List:

main.py | output-dfg.txt

A1 > testcases > test02 > output-dfg.txt

```
1  4650988912 4652294192
2  4651083632 4652294192
3  4306244208 4651083632
4  4306244272 4651083632
5
```

Test Case 3: Adding 2 variables and writing the value to a third variable

Input:

main.py | input.txt

A1 > testcases > test03 > input.txt

```
1  a = b + c
```

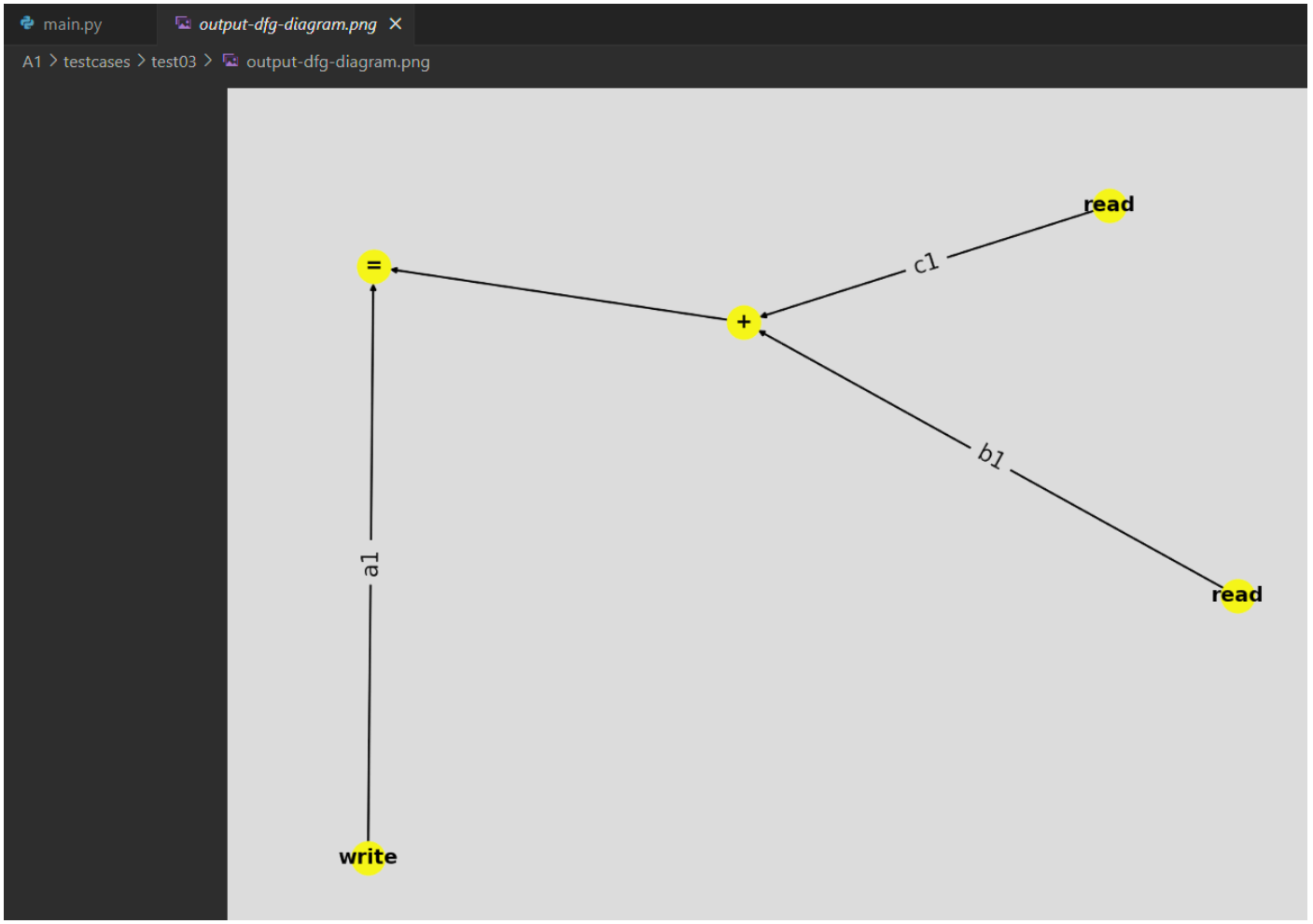
AST dictionary:

main.py | output-ast.txt

A1 > testcases > test03 > output-ast.txt

```
1  {"0": [["a"], ["="], [["b"], ["+"], ["c"]]]}
```

DFG diagram:



Key Map:

main.py output-keymap.txt

A1 > testcases > test03 > output-keymap.txt

```
1  {"1=": 4671579504, "write-a1": 4671604016, "1+": 4671604080, "read-b1": 4396586992, "read-c1": 4671997296}
```

DFG Adjacency List:

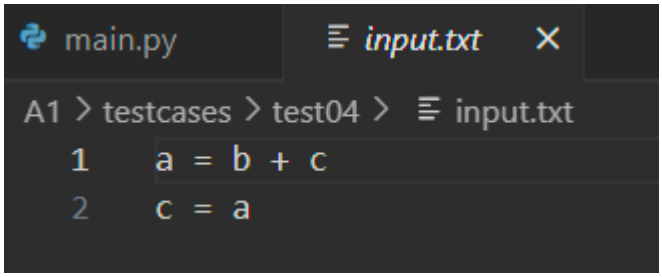
main.py output-dfg.txt

A1 > testcases > test03 > output-dfg.txt

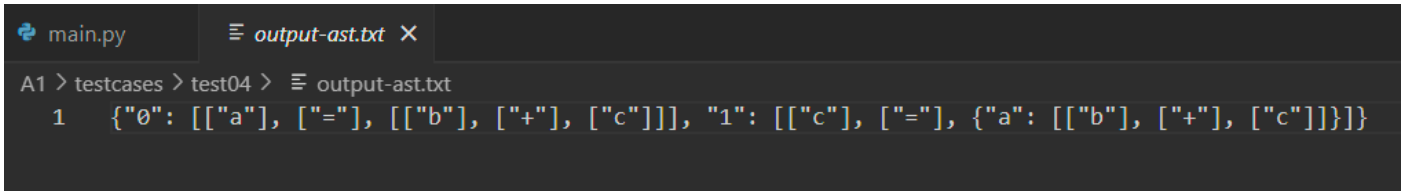
```
1  4671604016 4671579504
2  4671604080 4671579504
3  4396586992 4671604080
4  4671997296 4671604080
5
```

Test Case 4: Adding 2 variables and writing the value to a third variable, and further, writing the value of the third variable to a variable which has been read before (a dependency is created).

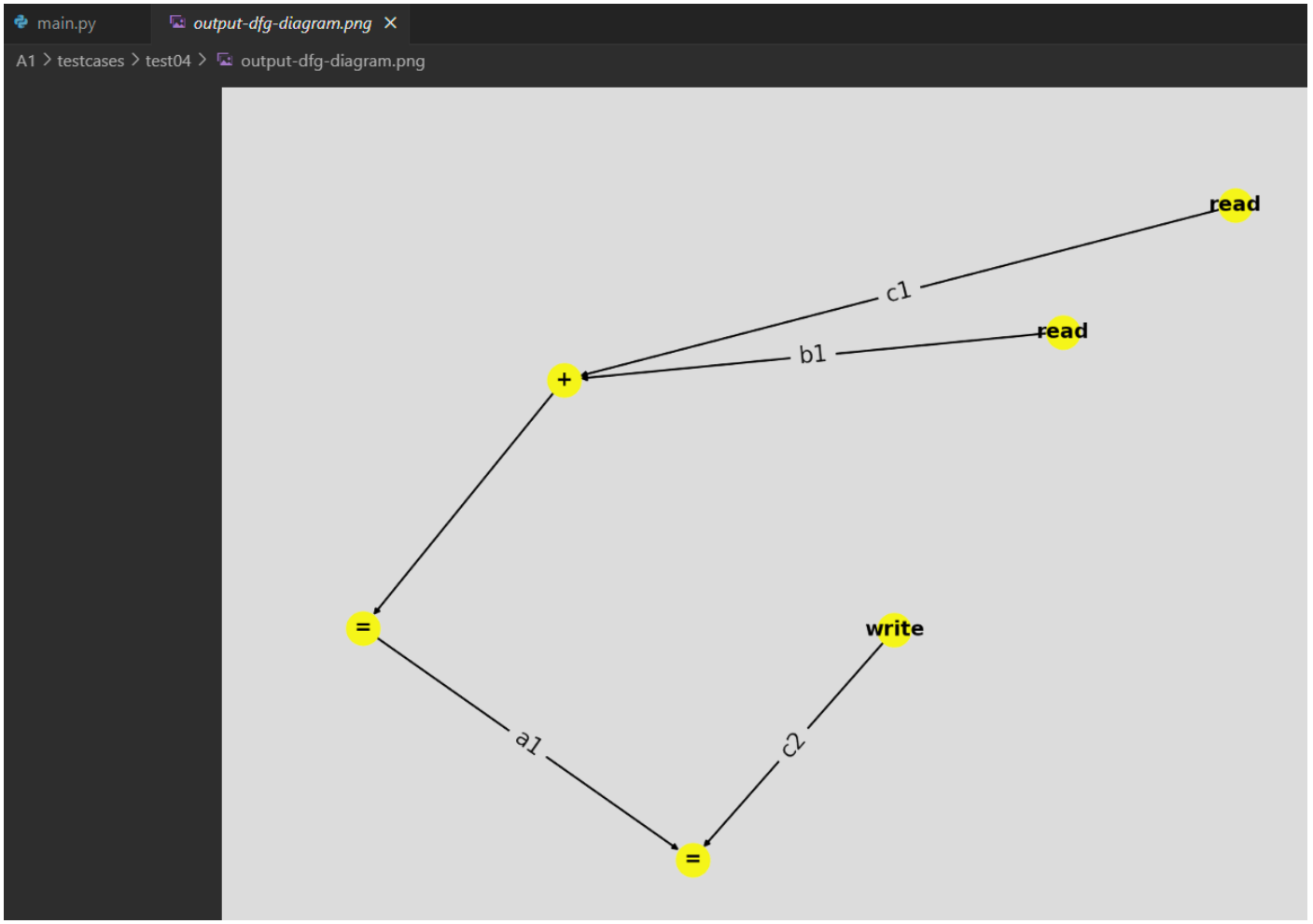
Input:



AST dictionary:



DFG diagram:



Key Map:

main.py output-keymap.txt

A1 > testcases > test04 > output-keymap.txt

```
1  {"1=": 4912824752, "1+": 4902283056, "read-b1": 4913151344, "read-c1": 4913151728, "2=": 4913151792, "write-c2": 4913181104}
```

DFG Adjacency List:

main.py output-dfg.txt

A1 > testcases > test04 > output-dfg.txt

```
1  4912824752 4913151792
2  4902283056 4912824752
3  4913151344 4902283056
4  4913151728 4902283056
5  4913181104 4913151792
6
```

Test Case 5: Adding 2 variables and writing the value to a third variable, and further, rewriting the value of a fourth variable to the third variable (no dependency is created)

Input:

main.py | input.txt

A1 > testcases > test05 > input.txt

```
1   a = b + c
2   a = c
```

AST dictionary:

main.py | output-ast.txt

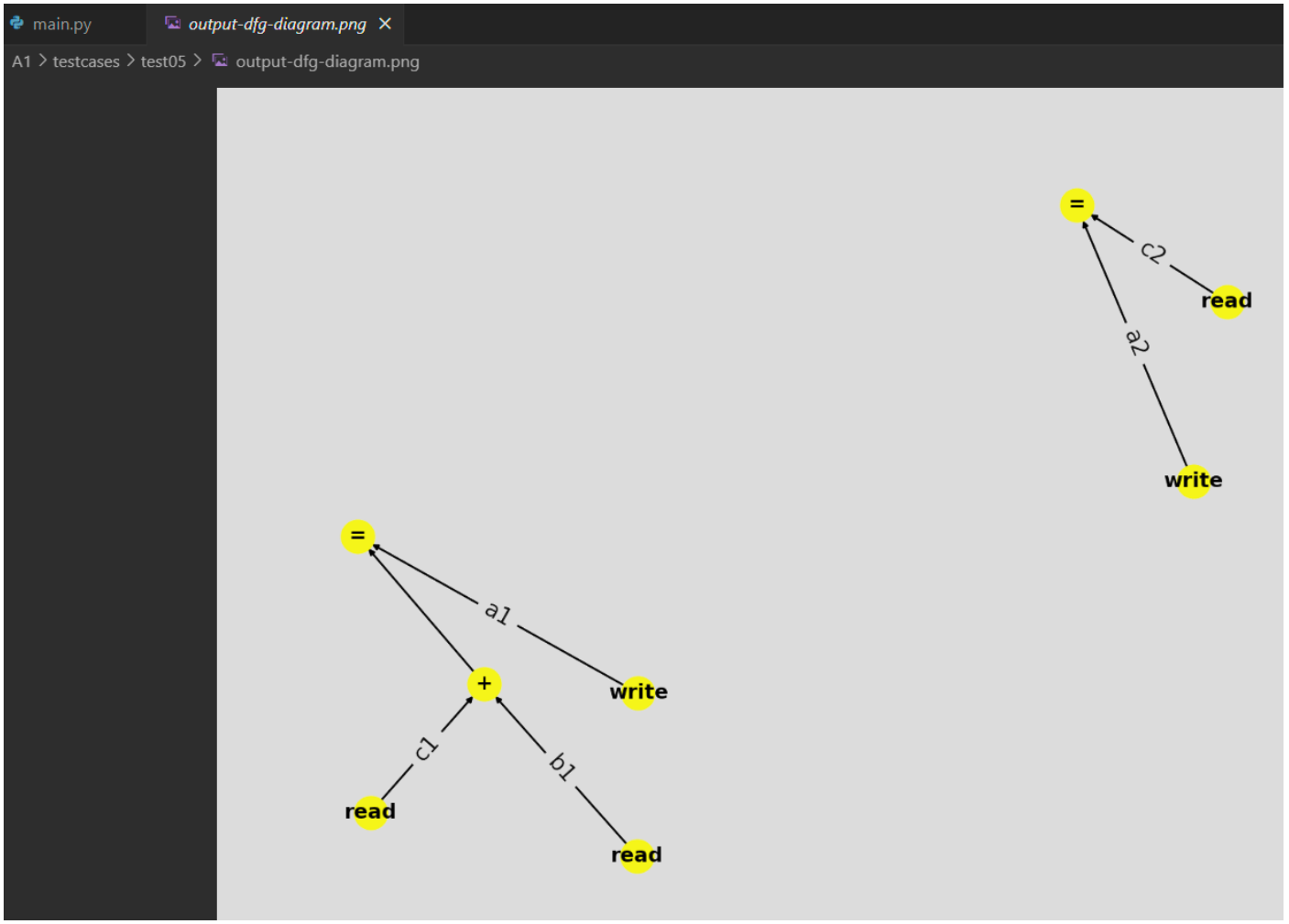
A1 > testcases > test05 > output-ast.txt

```
1   {"0": [["a"], ["="], [["b"], ["+"], ["c"]]], "1": [["a"], ["="], ["c"]]}
```

DFG diagram:

main.py | output-dfg-diagram.png

A1 > testcases > test05 > output-dfg-diagram.png



Key Map:

main.py | output-keymap.txt

A1 > testcases > test05 > output-keymap.txt

```
1  {"1=": 4722983344, "write-a1": 4704883184, "1+": 4704883120, "read-b1": 4704906352, "read-c1": 4723114864, "2=": 4723114928,
   "write-a2": 4723115504, "read-c2": 4723115888}
```

DFG Adjacency List:

main.py | output-dfg.txt

A1 > testcases > test05 > output-dfg.txt

```
1  4704883184 4722983344
2  4704883120 4722983344
3  4704906352 4704883120
4  4723114864 4704883120
5  4723115504 4723114928
6  4723115888 4723114928
7
```

Test Case 6: Adding 2 variables and writing the value to a third variable. Writing the value of 2 new variables to another new third variable.

Input:

main.py | input.txt

A1 > testcases > test06 > input.txt

```
1  a = b + c
2  d = e + f
```

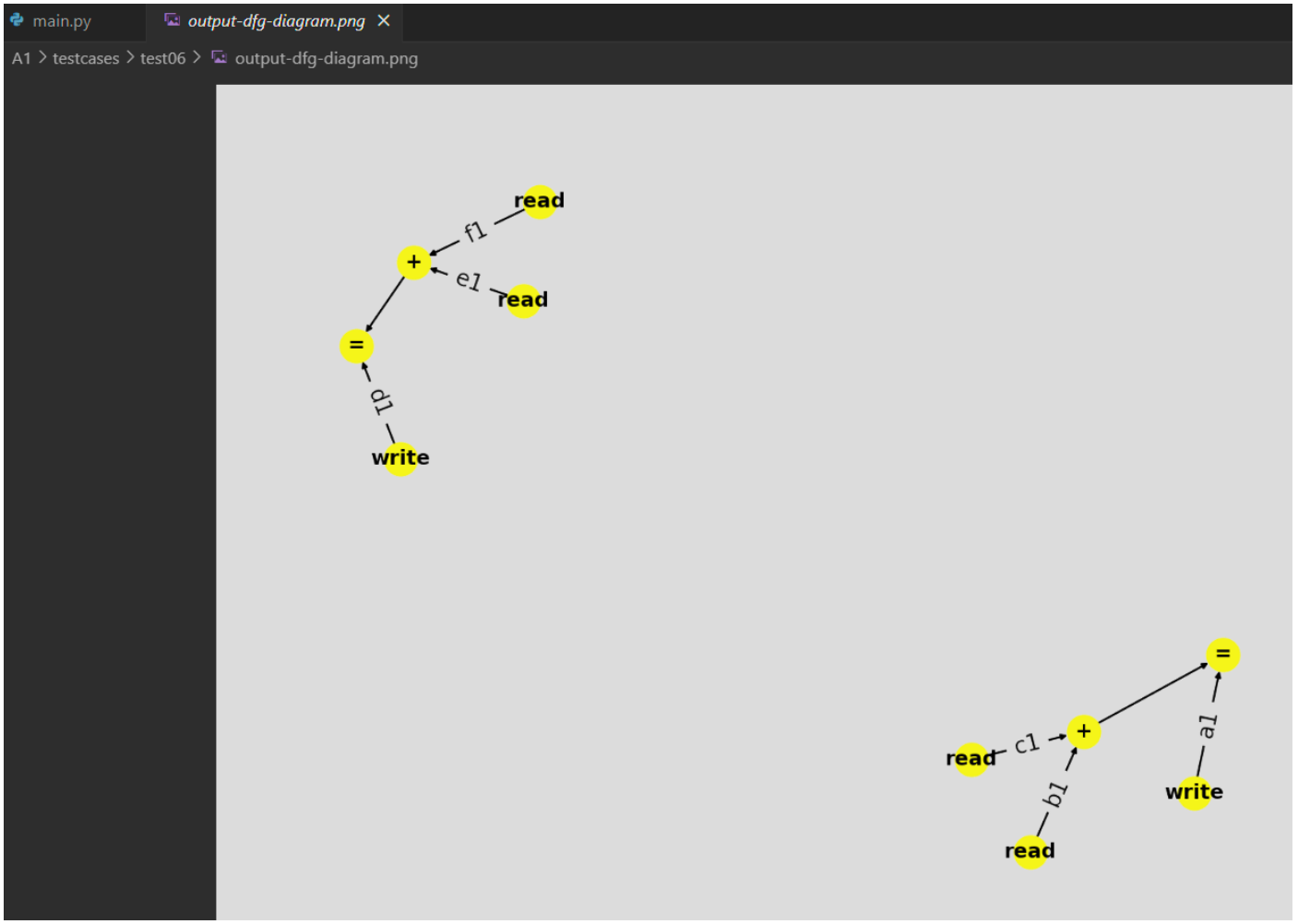
AST dictionary:

main.py | output-ast.txt

A1 > testcases > test06 > output-ast.txt

```
1  {"0": [["a"], ["="], [["b"], ["+"], ["c"]]], "1": [["d"], ["="], [["e"], ["+"], ["f"]]]}
```

DFG diagram:



Key Map:

main.py output-keymap.txt

A1 > testcases > test06 > output-keymap.txt


```
{"1=": 4858251952, "Write-a1": 4857994608, "1+": 4857994864, "read-b1": 4385859824, "read-c1": 4858377072, "2=": 4858377136,
"write-d1": 4858377712, "2+": 4858377776, "read-e1": 4858378416, "read-f1": 4858378800}
```


DFG Adjacency List:

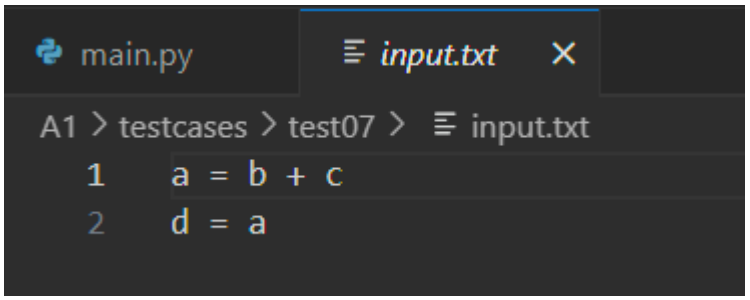
main.py output-dfg.txt

A1 > testcases > test06 > output-dfg.txt

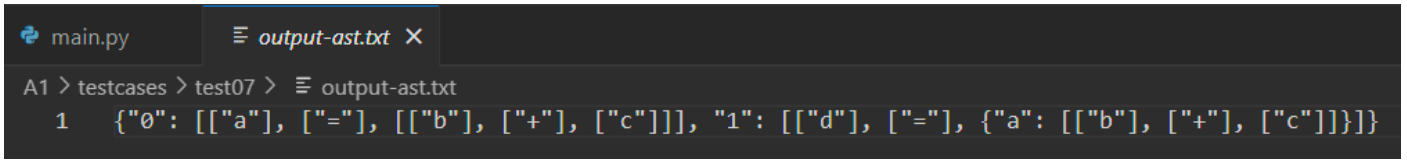

```
4857994608 4858251952
4857994864 4858251952
4385859824 4857994864
4858377072 4857994864
4858377712 4858377136
4858377776 4858377136
4858378416 4858377776
4858378800 4858377776
```

Test Case 7: Adding 2 variables and writing the value to a third variable, and further, rewriting the value of the third variable to a fourth variable

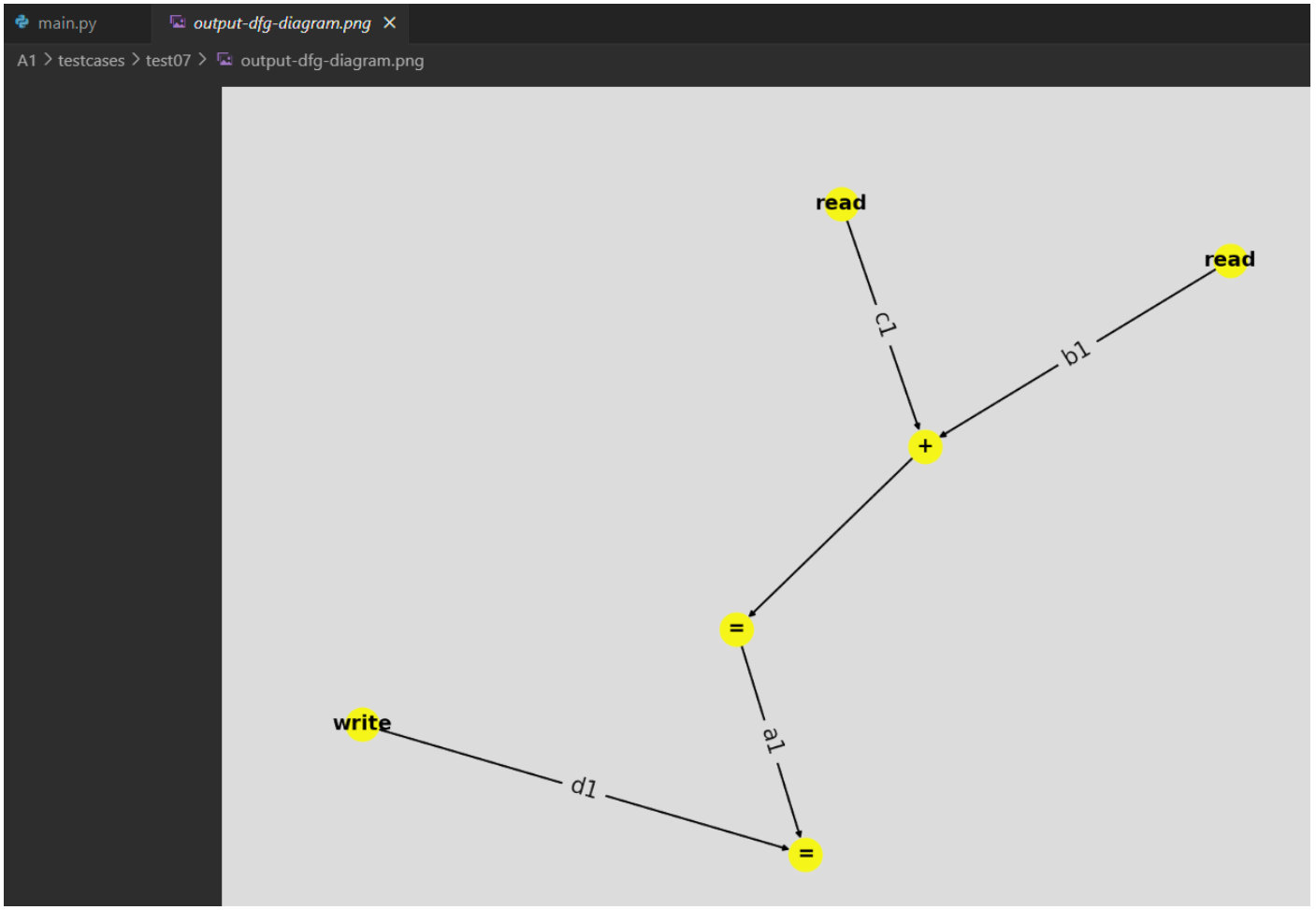
Input:



AST dictionary:

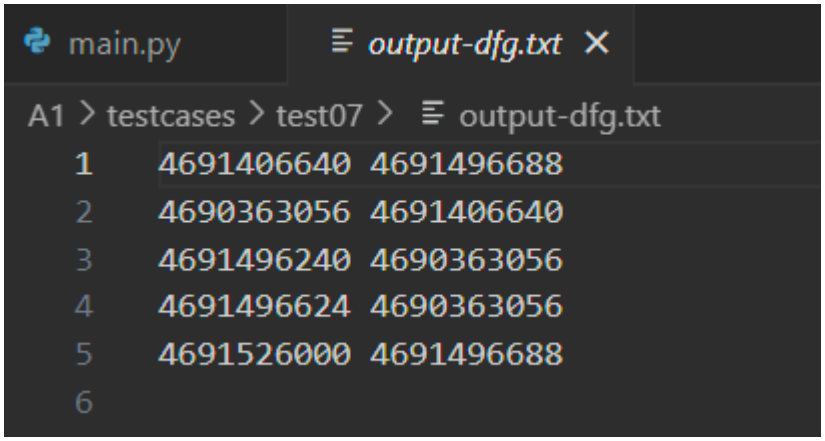


DFG diagram:



Key Map:

DFG Adjacency List:



Test Case 8: Adding 2 variables and writing the value to a third variable. The next statement adds the third variable to another variable, creating a dependency.

Input:

main.py | input.txt

A1 > testcases > test08 > input.txt

```
1  a = b + c
2  d = a + k
```

AST dictionary:

main.py | output-ast.txt

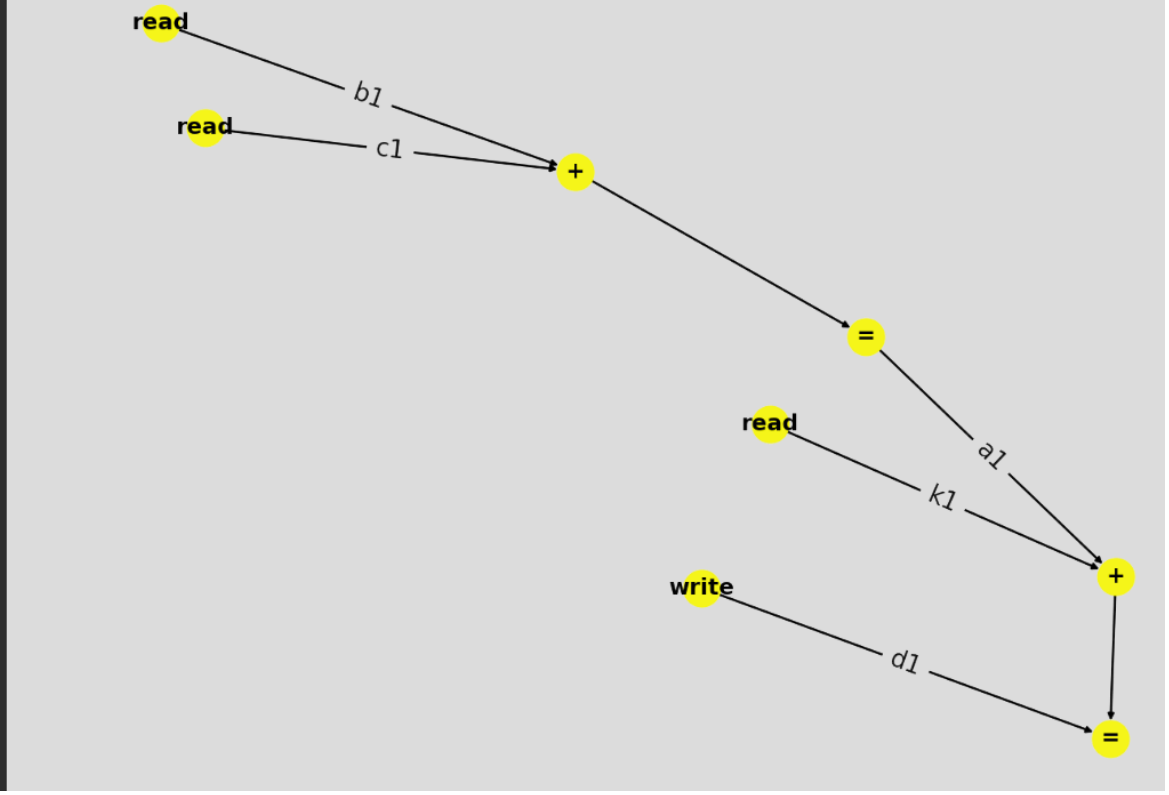
A1 > testcases > test08 > output-ast.txt


```
{"0": [["a"], ["="], [["b"], ["+"], ["c"]]], "1": [["d"], ["="], [{"a": [["b"], ["+"], ["c"]]}, ["+"], ["k"]]]}
```


DFG diagram:

main.py | output-dfg-diagram.png

A1 > testcases > test08 > output-dfg-diagram.png



Key Map:

main.py output-keymap.txt

A1 > testcases > test08 > output-keymap.txt

```
1  {"1=": 4732449904, "1+": 4695222832, "read-b1": 4695258480, "read-c1": 4733586416, "2=": 4733586480, "write-d1": 4733587056, "2+": 4733587120, "read-k1": 4733273072}
```

DFG Adjacency List:

main.py output-dfg.txt

A1 > testcases > test08 > output-dfg.txt

```
1  4732449904 4733587120
2  4695222832 4732449904
3  4695258480 4695222832
4  4733586416 4695222832
5  4733587056 4733586480
6  4733587120 4733586480
7  4733273072 4733587120
8
```

Test Case 9: 3 addition statements, with the third being dependent on the first and the second one

Input:

main.py input.txt

A1 > testcases > test09 > input.txt

```
1  a = b + c
2  d = e + f
3  g = a + d
```

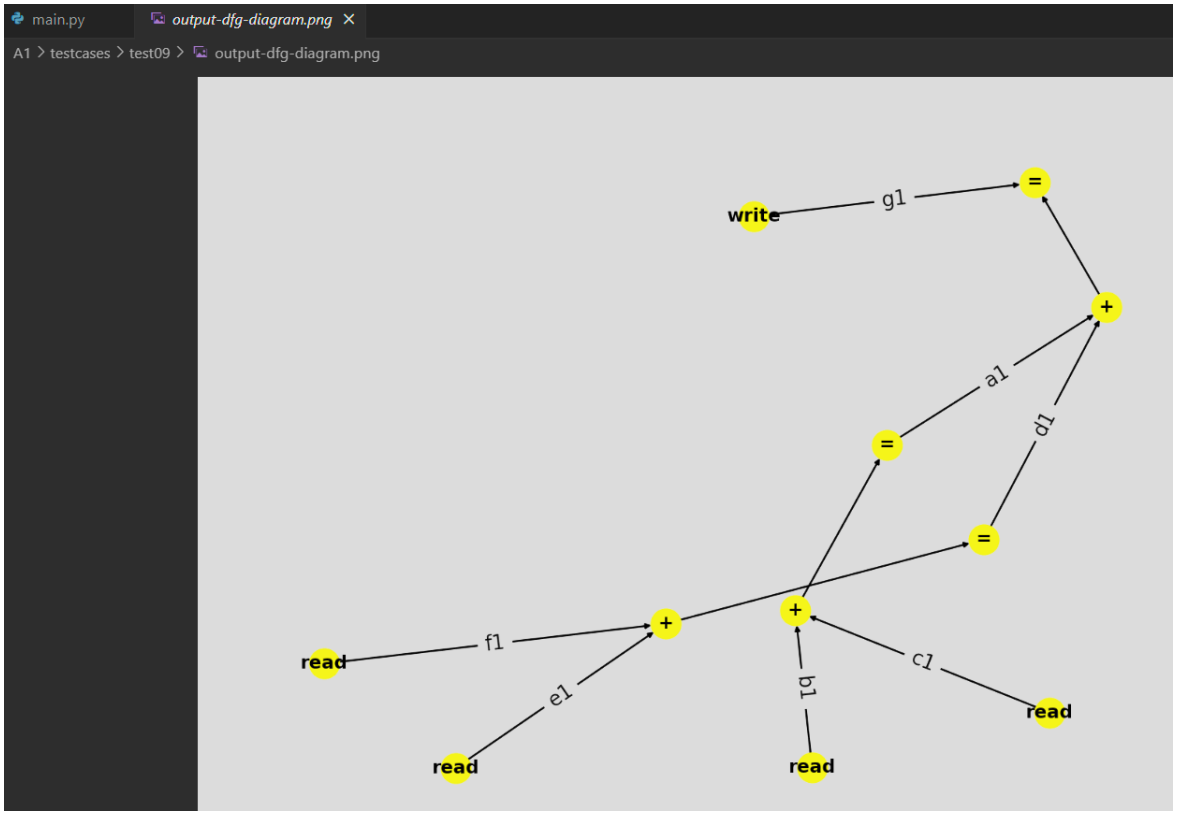
AST dictionary:

main.py output-ast.txt

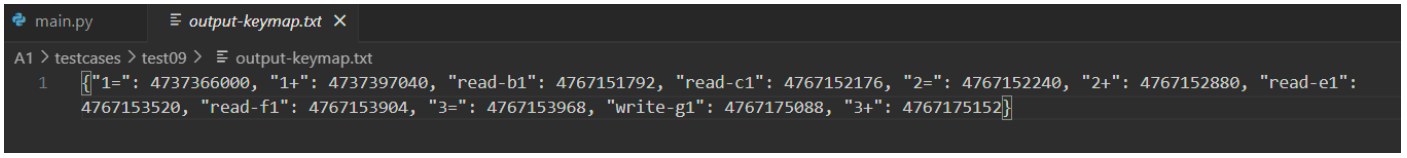
A1 > testcases > test09 > output-ast.txt

```
1  {"0": [["a"], ["="], [["b"], ["+"], ["c"]]], "1": [["d"], ["="], [["e"], ["+"], ["f"]]], "2": [["g"], ["="], [{"a": [["b"], ["+"], ["c"]]}, ["+"], {"d": [["e"], ["+"], ["f"]]}]]}
```

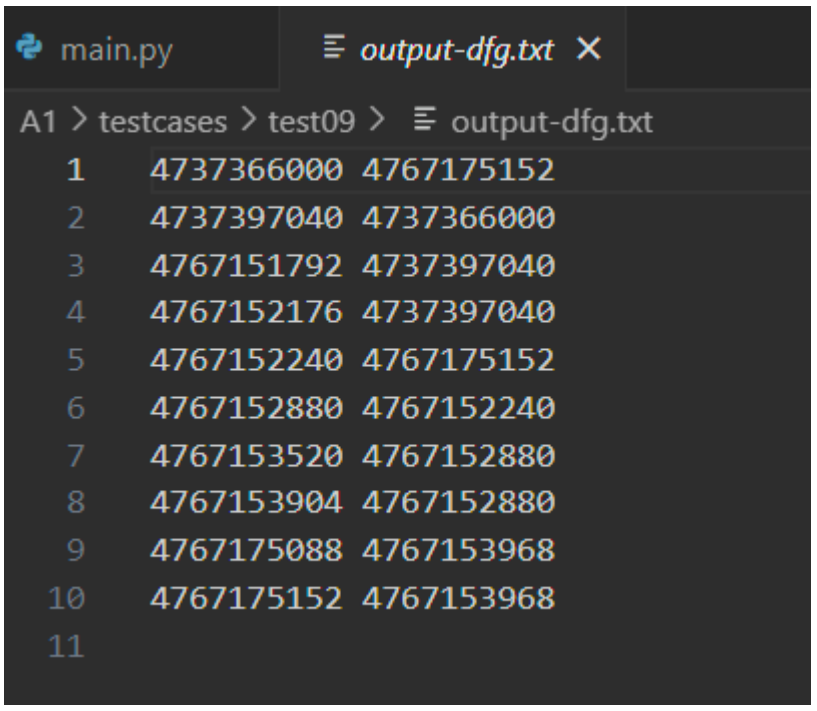
DFG diagram:



Key Map:



DFG Adjacency List:



Test Case 10: 2 independent addition statements, in which a read variable is written in the next statement

Input:

main.py | input.txt

A1 > testcases > test10 > input.txt

```
1  a = b + c
2  c = k
```

AST dictionary:

main.py | output-ast.txt

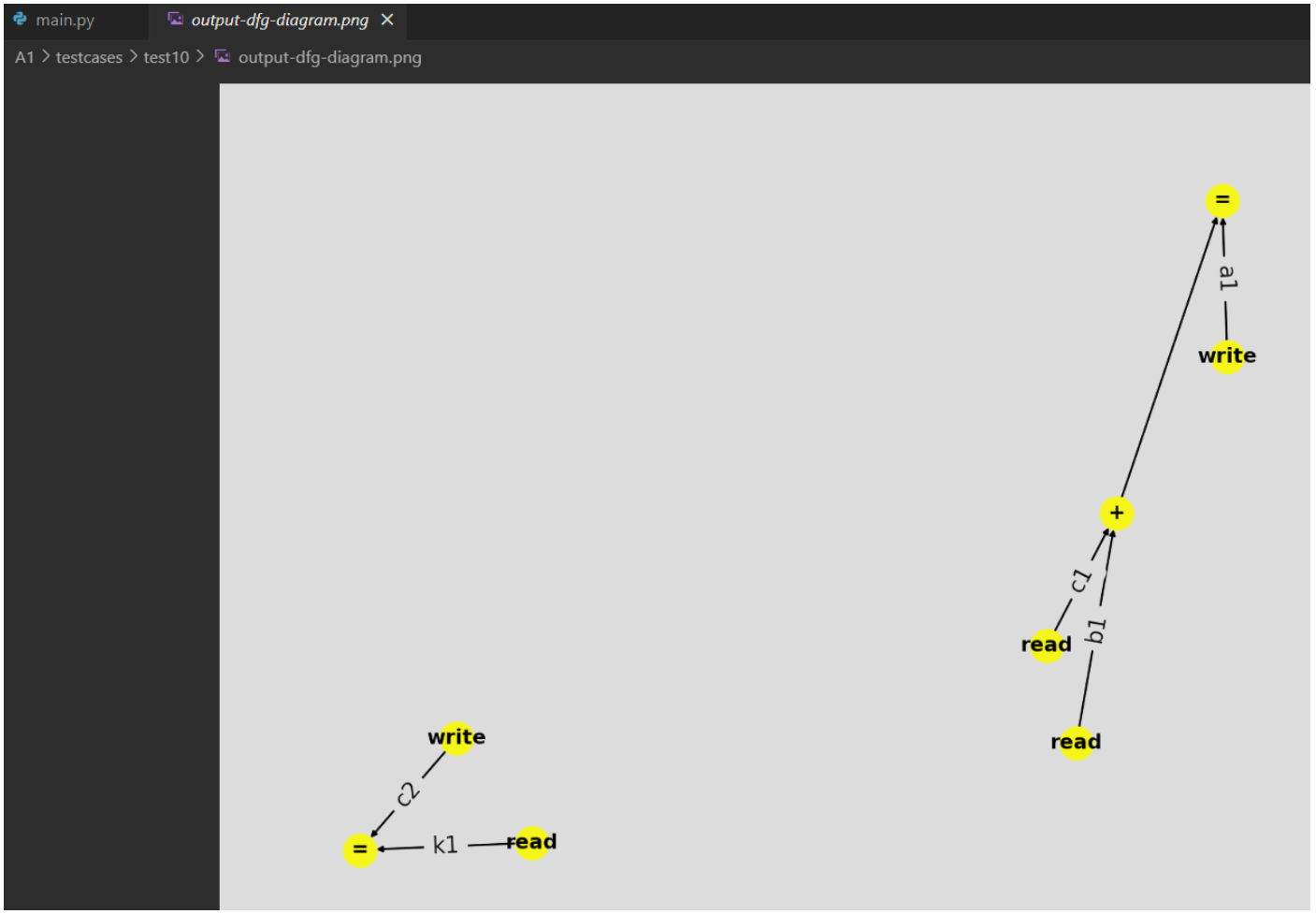
A1 > testcases > test10 > output-ast.txt

```
1  {"0": [["a"], ["="], [["b"], ["+"], ["c"]]], "1": [["c"], ["="], ["k"]]}
```

DFG diagram:

main.py | output-dfg-diagram.png

A1 > testcases > test10 > output-dfg-diagram.png



Key Map:

main.py | output-keymap.txt

A1 > testcases > test10 > output-keymap.txt

```
1  {"1=": 4800147248, "write-a1": 4800025072, "1+": 4800024880, "read-b1": 5050263216, "read-c1": 5050263600, "2=": 5050263664,
   "write-c2": 5050264240, "read-k1": 5050264624}
```

DFG Adjacency List:

main.py | output-dfg.txt

A1 > testcases > test10 > output-dfg.txt

```
1  4800025072 4800147248
2  4800024880 4800147248
3  5050263216 4800024880
4  5050263600 4800024880
5  5050264240 5050263664
6  5050264624 5050263664
7
```

Test Case 11: Adding a variable to itself, and then storing the result in a second variable.

Input:

main.py | input.txt

A1 > testcases > test11 > input.txt

```
1  a = b + b
```

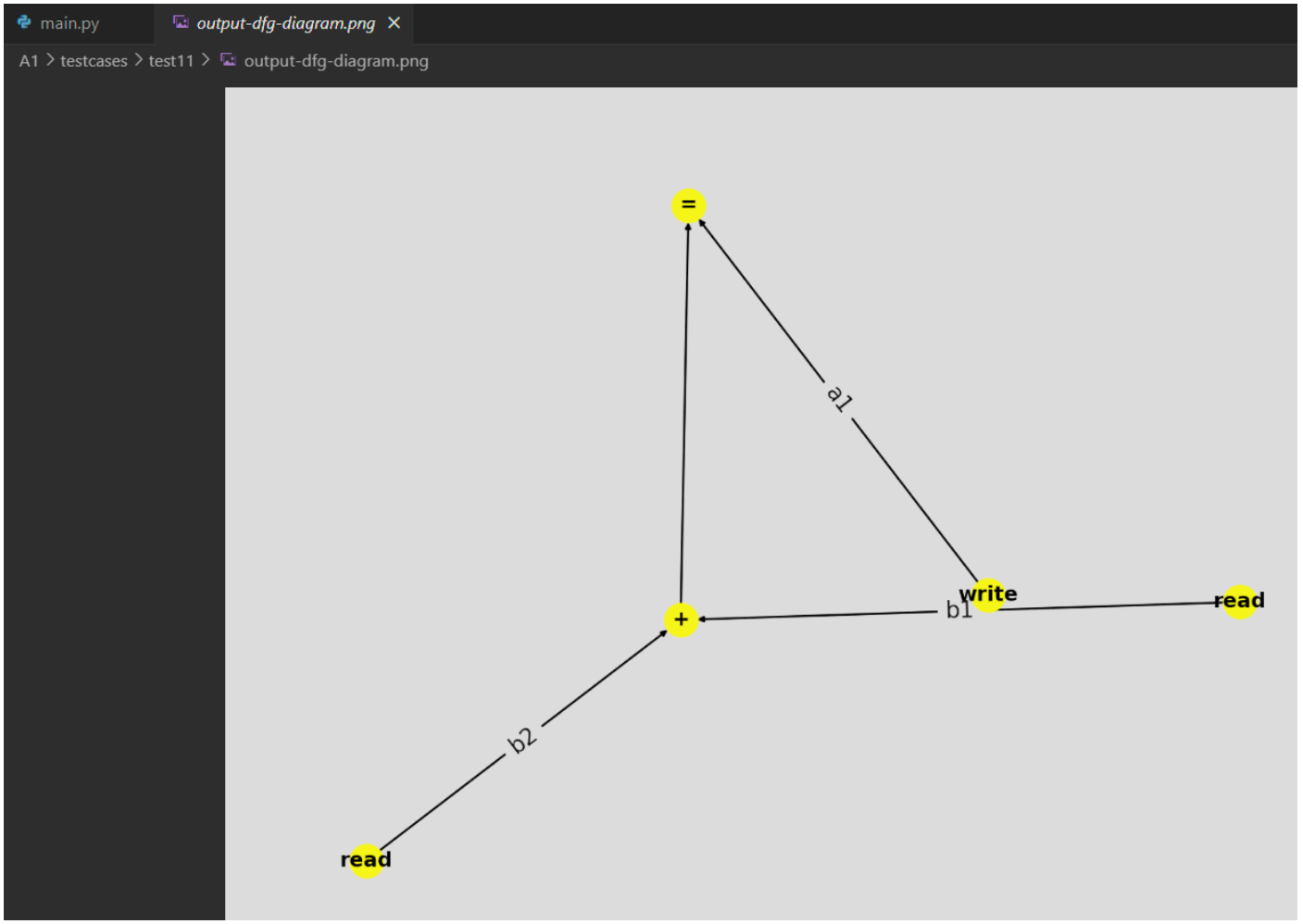
AST dictionary:

main.py | output-ast.txt

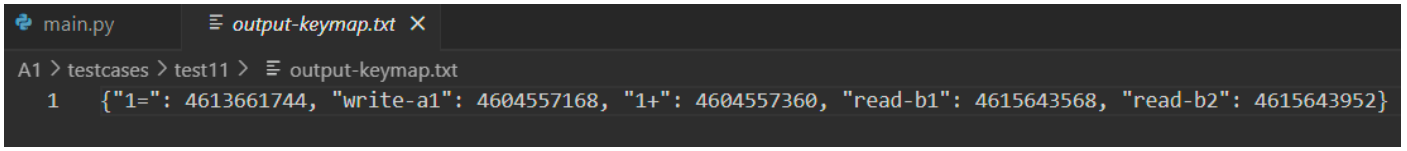
A1 > testcases > test11 > output-ast.txt

```
1  {"0": [["a"], ["="], [["b"], ["+"], ["b"]]]}
```

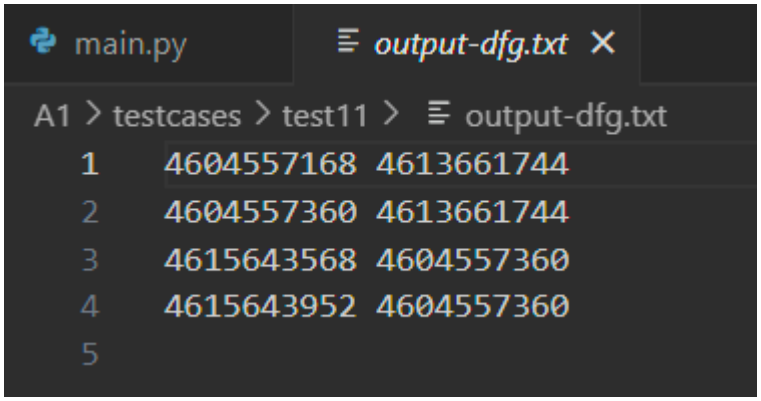
DFG diagram:



Key Map:



DFG Adjacency List:



Test Case 12: Storing the value of a variable to itself

Input:

main.py | input.txt

A1 > testcases > test12 > input.txt

```
1  a = a
```

AST dictionary:

main.py | output-ast.txt

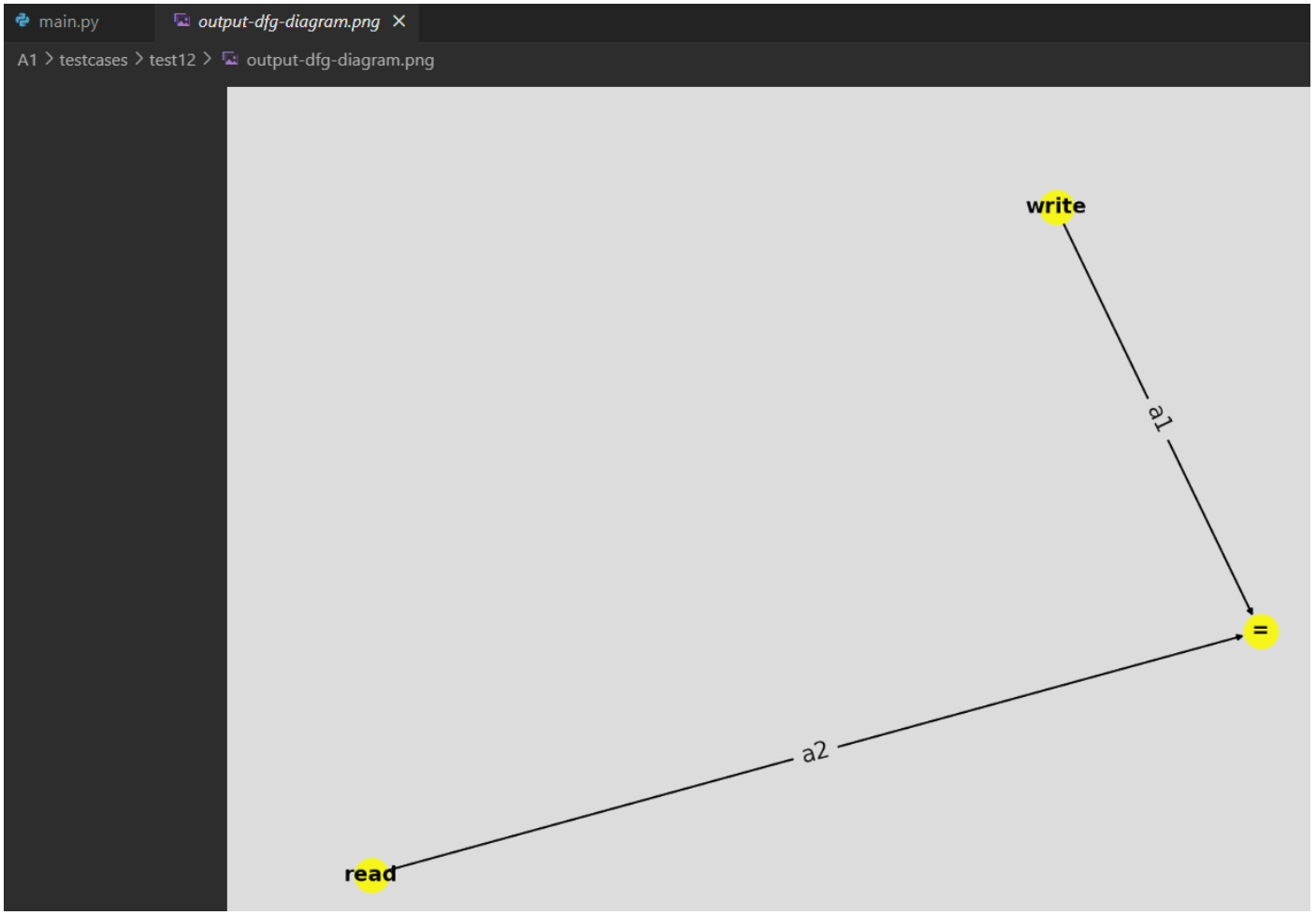
A1 > testcases > test12 > output-ast.txt

```
1  {"0": [["a"], ["="], ["a"]]}
```

DFG diagram:

main.py | output-dfg-diagram.png

A1 > testcases > test12 > output-dfg-diagram.png



Key Map:

main.py | output-keymap.txt

A1 > testcases > test12 > output-keymap.txt

```
1  {"1=": 4550038704, "write-a1": 4555531568, "read-a2": 4352612912}
```

DFG Adjacency List:

main.py | output-dfg.txt

A1 > testcases > test12 > output-dfg.txt

```
1  4555531568 4550038704
2  4352612912 4550038704
3
```

Test Case 13: 2 independent assignments

Input:

main.py | input.txt

A1 > testcases > test13 > input.txt

```
1  a = b
2  b = c
```

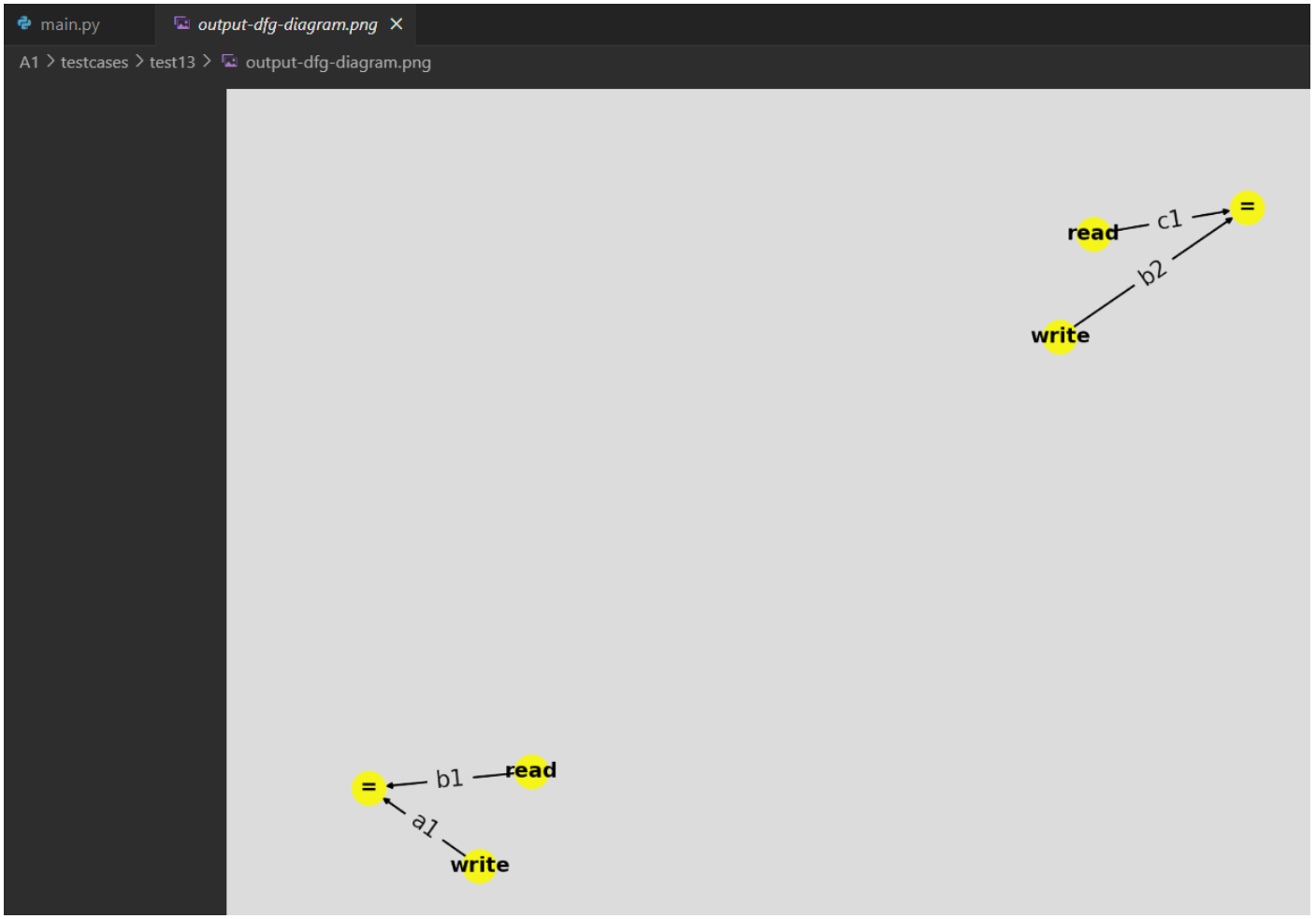
AST dictionary:

main.py | output-ast.txt

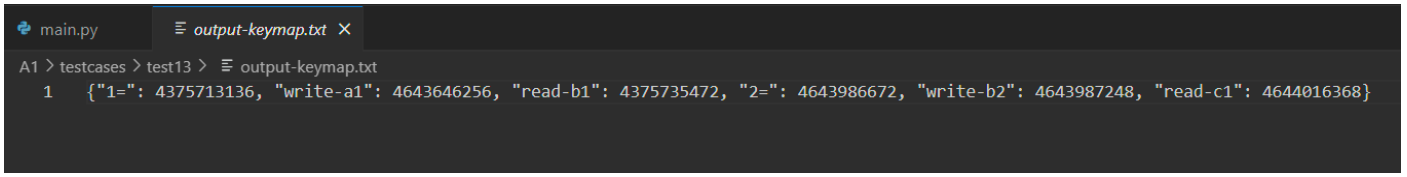
A1 > testcases > test13 > output-ast.txt

```
1  {"0": [["a"], ["="], ["b"]], "1": [["b"], ["="], ["c"]]}
```

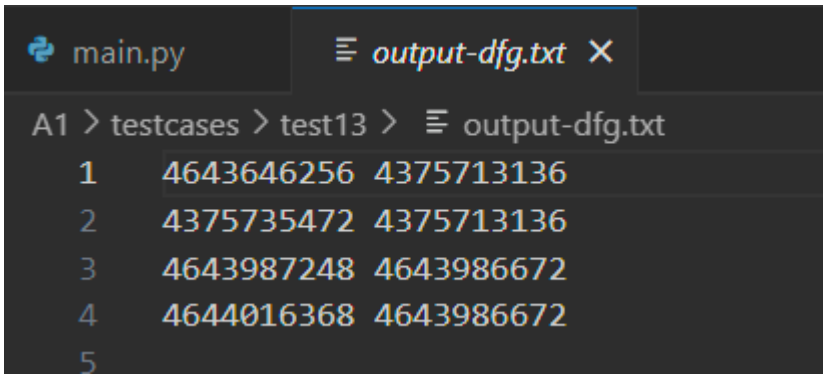
DFG diagram:



Key Map:



DFG Adjacency List:



Test Case 14: 2 dependent addition statements, in which one the variable to be written in the second statement has already been read once.

Input:

main.py | input.txt

A1 > testcases > test14 > input.txt

```
1  a = b + c
2  b = a
```

AST dictionary:

main.py | output-ast.txt

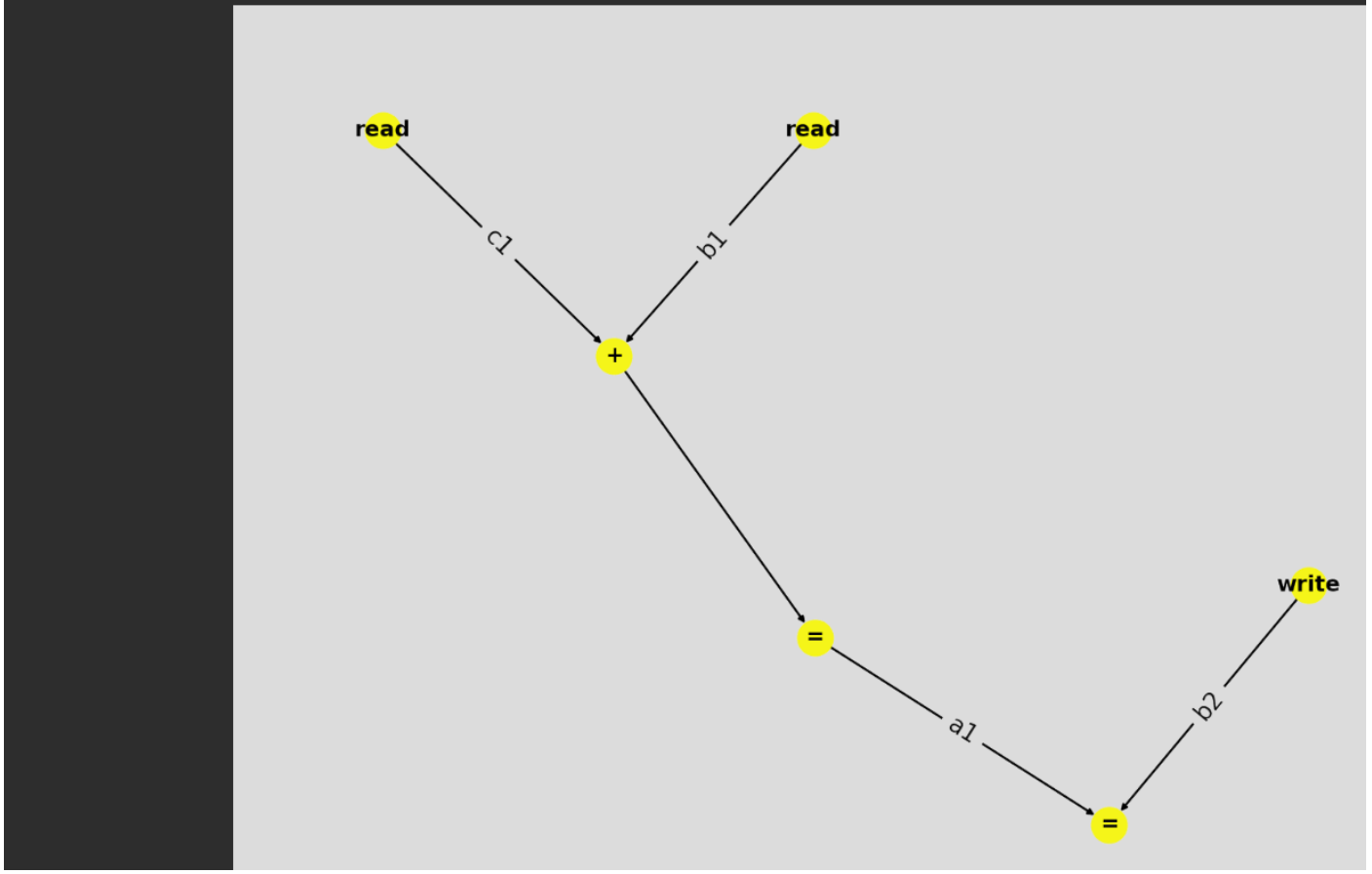
A1 > testcases > test14 > output-ast.txt

```
1  {"0": [["a"], ["="], [["b"], ["+"], ["c"]]], "1": [["b"], ["="], {"a": [["b"], ["+"], ["c"]]}]}
```

DFG diagram:

main.py | output-dfg-diagram.png

A1 > testcases > test14 > output-dfg-diagram.png



Key Map:

main.py | output-keymap.txt

A1 > testcases > test14 > output-keymap.txt

```
1  {"1=": 4667085232, "1+": 4347813360, "read-b1": 4347848944, "read-c1": 4667409200, "2=": 4667409264, "write-b2": 4667409840}
```

DFG Adjacency List:

main.py | output-dfg.txt

A1 > testcases > test14 > output-dfg.txt

```
1  4667085232 4667409264
2  4347813360 4667085232
3  4347848944 4347813360
4  4667409200 4347813360
5  4667409840 4667409264
6
```

Test Case 15: 2 variables dependent on a variable

Input:

main.py | input.txt

A1 > testcases > test16 > input.txt

```
1  a = b
2  d = a
3  c = a
```

AST dictionary:

main.py | output-ast.txt

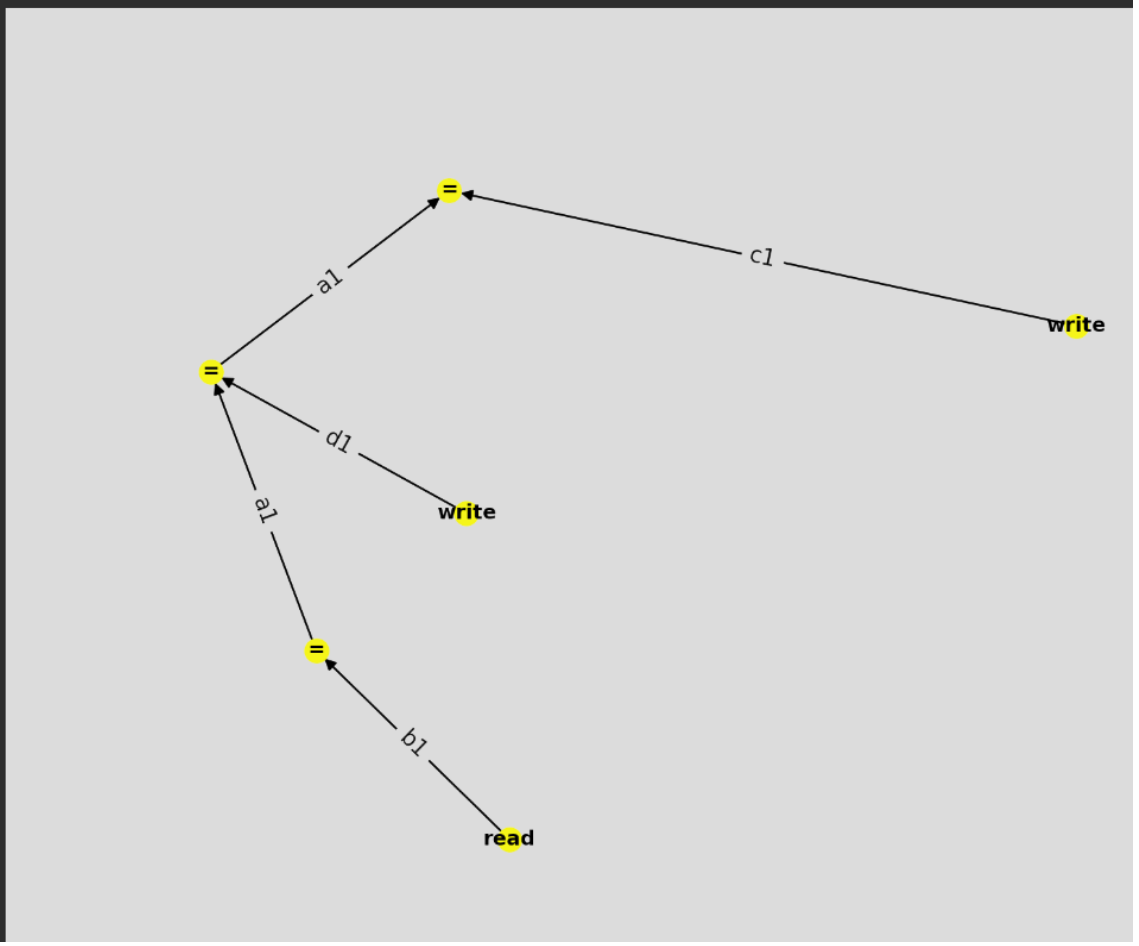
A1 > testcases > test16 > output-ast.txt

```
1  {"0": [["a"], ["="], ["b"]], "1": [["d"], ["="], {"a": ["b"]}], "2": [["c"], ["="], {"a": ["b"]}]}
```

DFG diagram:

main.py output-dfg-diagram.png

A1 > testcases > test16 > output-dfg-diagram.png



Key Map:

main.py output-keymap.txt

A1 > testcases > test16 > output-keymap.txt

```
1   {"1=": 4980690672, "read-b1": 4969902832, "2=": 4969925744, "write-d1": 4980925040, "3=": 4980213296, "write-c1": 4980925680}
```

DFG Adjacency List:

main.py output-dfg.txt

A1 > testcases > test16 > output-dfg.txt

```
1   4980690672 4969925744
2   4969902832 4980690672
3   4969925744 4980213296
4   4980925040 4969925744
5   4980925680 4980213296
6
```

Test Case 17: Addition of 2 constants

Input:

main.py | input.txt

A1 > testcases > test18 > input.txt

```
1    a = 2 + 3
```

AST dictionary:

main.py | output-ast.txt

A1 > testcases > test18 > output-ast.txt

```
1    {"0": [["a"], ["="], [["2"], ["+"], ["3"]]]}
```

DFG diagram:

main.py | output-dfg-diagram.png

A1 > testcases > test18 > output-dfg-diagram.png

Key Map:

main.py | output-keymap.txt

A1 > testcases > test18 > output-keymap.txt

```
1  {"1=": 4686647408, "write-a1": 4686683120, "1+": 4686843376, "2": 4376106096, "3": 4376106160}
```

DFG Adjacency List:

main.py | output-dfg.txt

A1 > testcases > test18 > output-dfg.txt

```
1  4686683120 4686647408
2  4686843376 4686647408
3  4376106096 4686843376
4  4376106160 4686843376
5
```

Test Case 18: Multiple statements having combination of the four operations

Input:

main.py | input.txt

A1 > testcases > test19 > input.txt

```
1  a = b + c
2  d = e * k - b
3  p = q + r
4  m = c / b + a
```

AST dictionary:

main.py | output-ast.txt

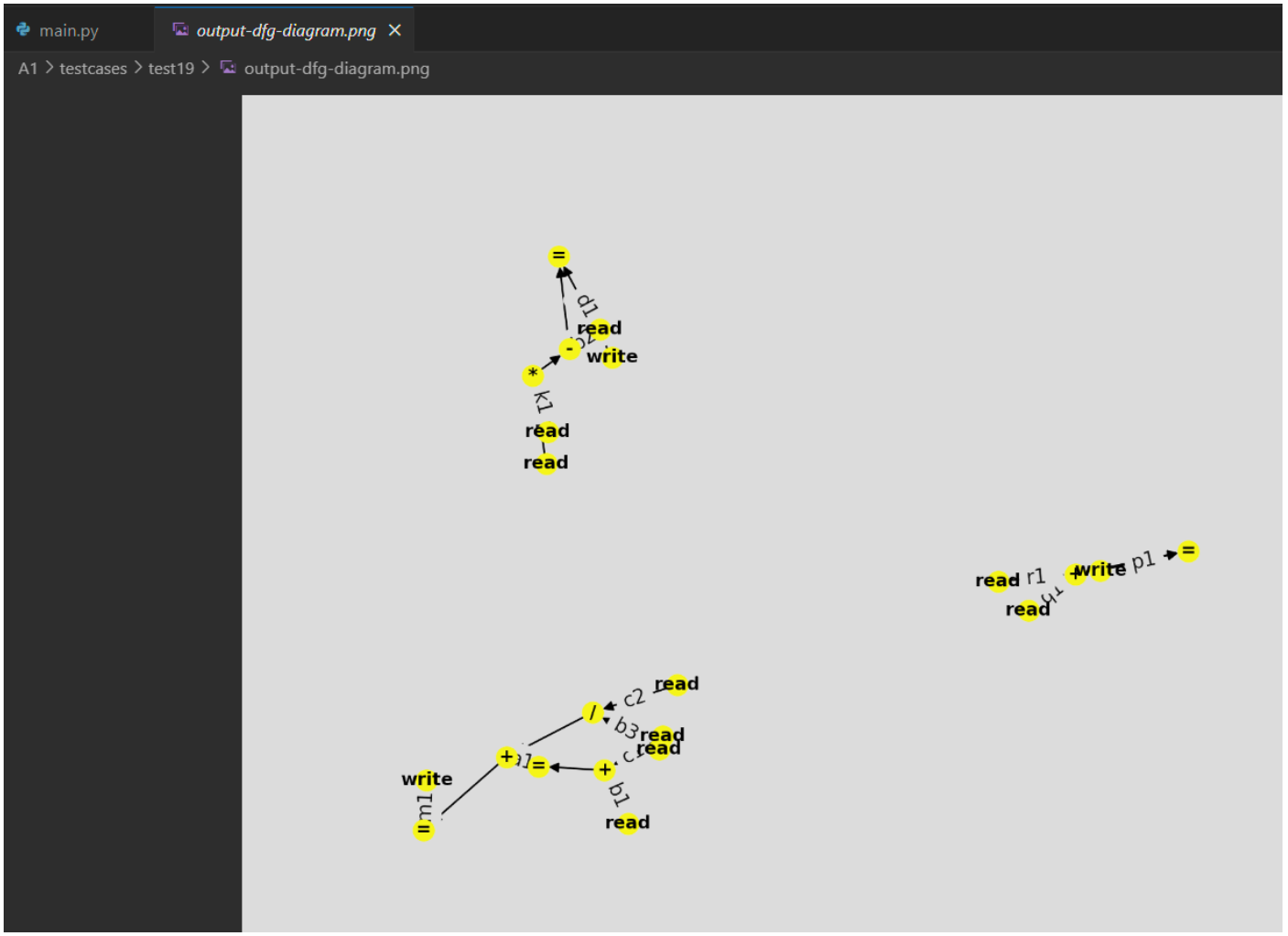
A1 > testcases > test19 > output-ast.txt

```
1  {"0": [["a"], ["="], [["b"], ["+"], ["c"]]], "1": [["d"], ["="], [[["e"], ["*"], ["k"]], ["-"], ["b"]]], "2": [["p"], ["="], [["q"], ["
   +"], ["r"]]], "3": [["m"], ["="], [[["c"], ["/"], ["b"]], ["+"], {"a": [["b"], ["+"], ["c"]]}]]}
```

DFG diagram:

main.py | output-dfg-diagram.png

A1 > testcases > test19 > output-dfg-diagram.png



Key Map:

main.py | output-keymap.txt

A1 > testcases > test19 > output-keymap.txt


```
{"1=": 4713891504, "1+": 4713891888, "read-b1": 4713892528, "read-c1": 4713892912, "2=": 4713892976, "write-d1": 4713893552, "1-": 4713893616, "1*": 4713910640, "read-e1": 4713911024, "read-k1": 4713911408, "read-b2": 4713911792, "3=": 4713911856, "write-p1": 4713912432, "2+": 4713912496, "read-q1": 4713913136, "read-r1": 4713913520, "4=": 4713913584, "write-m1": 4713914160, "3+": 4713914224, "1/": 4713923056, "read-c2": 4713923440, "read-b3": 4713923824}
```

DFG Adjacency List:

main.py | output-dfg.txt

A1 > testcases > test19 > output-dfg.txt

```
4713891504 4713914224
4713891888 4713891504
4713892528 4713891888
4713892912 4713891888
4713893552 4713892976
4713893616 4713892976
4713910640 4713893616
4713911024 4713910640
4713911408 4713910640
4713911792 4713893616
4713912432 4713911856
4713912496 4713911856
4713913136 4713912496
4713913520 4713912496
4713914160 4713913584
4713914224 4713913584
4713923056 4713914224
4713923440 4713923056
4713923824 4713923056
```

Test Case 19: Single statement combining multiple operations and variables

Input:

main.py | input.txt

A1 > testcases > test20 > input.txt

```
a = b + c + d / e * f - g - h * i - j
```

AST dictionary:

main.py | output-ast.txt

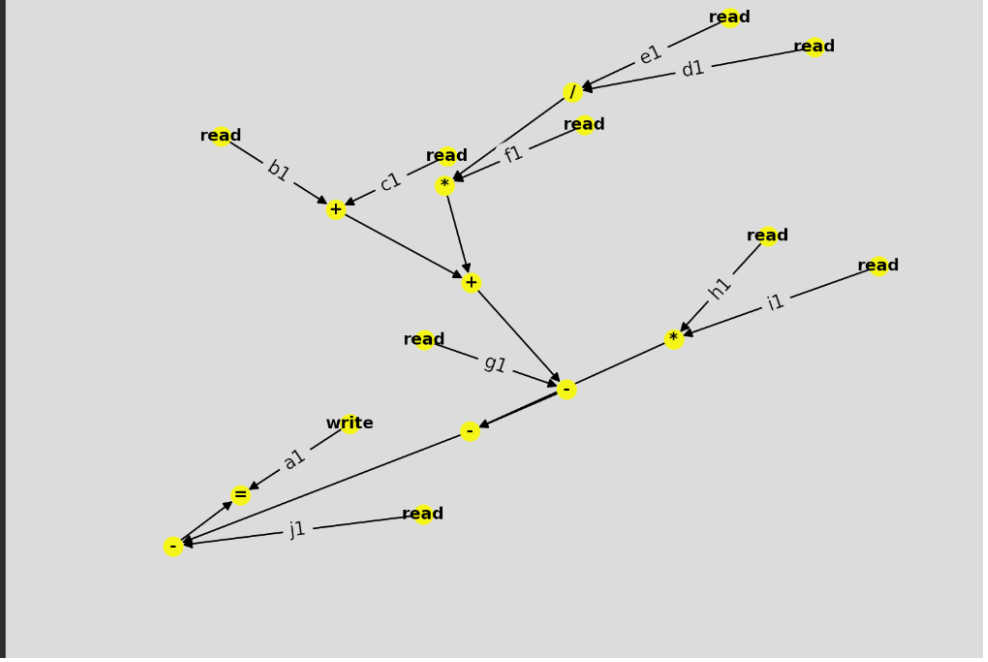
A1 > testcases > test20 > output-ast.txt

```
1  {"0": [["a"], ["="], [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"],
   ["i"]]], ["-"], ["j"]]]}
```

DFG Diagram:

main.py | output-dfg-diagram.png

A1 > testcases > test20 > output-dfg-diagram.png



Key Map:

main.py | output-keymap.txt

A1 > testcases > test20 > output-keymap.txt


```
1  {"1=": 4866254128, "write-a1": 4866254448, "1-": 4866254512, "2-": 4866255088, "3-": 4866255408, "1+": 4866255728, "2+": 4866256048,
   "read-b1": 4866256432, "read-c1": 4866256816, "1*": 4866269232, "1/": 4866269808, "read-d1": 4866270192, "read-e1": 4866270576,
   "read-f1": 4866270960, "read-g1": 4866271344, "2*": 4866271408, "read-h1": 4866272048, "read-i1": 4866272432, "read-j1": 4866272816}
```

DFG Adjacency list

main.py | output-dfg.txt

A1 > testcases > test20 > output-dfg.txt

```
4866254448 4866254128
4866254512 4866254128
4866255088 4866254512
4866255408 4866255088
4866255728 4866255408
4866256048 4866255728
4866256432 4866256048
4866256816 4866256048
4866269232 4866255728
4866269808 4866269232
4866270192 4866269808
4866270576 4866269808
4866270960 4866269232
4866271344 4866255408
4866271408 4866255088
4866272048 4866271408
4866272432 4866271408
4866272816 4866254512

```

Test Case 20: Multiple statements combining multiple operations and variables, and dependencies

Input:

main.py | input.txt

A1 > testcases > test21 > input.txt

```
a = b + c + d / e * f - g - h * i - j
y = a + k
z = a * y + d + y / x - y
```

AST dictionary:

main.py | output-ast.txt

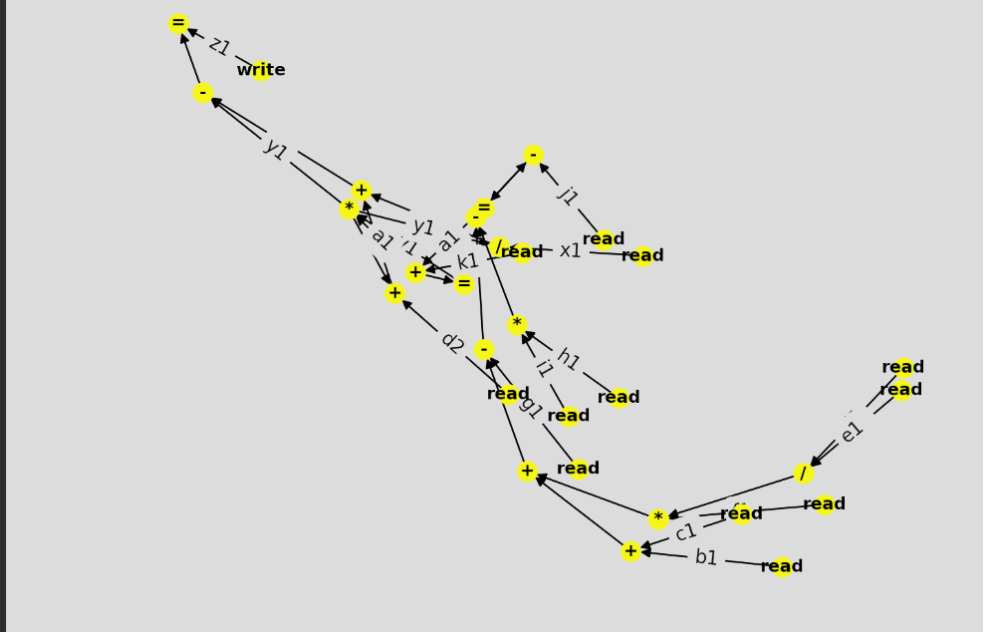
A1 > testcases > test21 > output-ast.txt


```
{"0": [["a"], ["="], [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"],
["i"]]], ["-"], ["j"]]], "1": [["y"], ["="], [{"a": [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"],
["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"], ["k"]]], "2": [["z"], ["="], [[[[{"a": [[[[[["b"], ["+"], ["c"]], ["+"],
[[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["*"], {"y": [{"a": [[[[[["b"], ["
+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"],
["k"]]}], ["+"], ["d"]], ["+"], [{"y": [{"a": [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]],
["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"], ["k"]]}, ["/"], ["x"]]], ["-"], {"y": [{"a": [[[[[["b"], ["+"], ["c"]], ["+"],
[[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"], ["k"]]]]]}
```


DFG Diagram:

main.py | output-dfg-diagram.png

A1 > testcases > test21 > output-dfg-diagram.png



Key Map:

main.py | output-keymap.txt

A1 > testcases > test21 > output-keymap.txt


```
{"1=": 4693761008, "1-": 4693761456, "2-": 4693778480, "3-": 4693778800, "1+": 4693779120, "2+": 4693779440, "read-b1": 4693779824,
"read-c1": 4693780208, "1*": 4693780272, "1/": 4693780848, "read-d1": 4693781232, "read-e1": 4693781616, "read-f1": 4693782000,
"read-g1": 4693782384, "2*": 4693782448, "read-h1": 4693791344, "read-i1": 4693791728, "read-j1": 4693792112, "2=": 4693792176, "3+":
4693792816, "read-k1": 4693656752, "3=": 4692617328, "write-z1": 4693793904, "4-": 4693793968, "4+": 4693794544, "5+": 4693803120,
"3*": 4693803440, "read-d2": 4693803888, "2/": 4693804080, "read-x1": 4693804912}
```

DFG Adjacency list

main.py | output-dfg.txt

A1 > testcases > test21 > output-dfg.txt

```
4693761008 4693792816
4693761456 4693761008
4693778480 4693761456
4693778800 4693778480
4693779120 4693778800
4693779440 4693779120
4693779824 4693779440
4693780208 4693779440
4693780272 4693779120
4693780848 4693780272
4693781232 4693780848
4693781616 4693780848
4693782000 4693780272
4693782384 4693778800
4693782448 4693778480
4693791344 4693782448
4693791728 4693782448
4693792112 4693761456
4693792176 4693803440
4693792816 4693792176
4693792816 4693803440
4693656752 4693792816
4693793904 4692617328
4693793968 4692617328
4693794544 4693793968
4693803120 4693794544
4693803440 4693803120
4693803440 4693804080
4693803440 4693793968
4693803888 4693803120
4693804080 4693794544
4693804912 4693804080
```

Test Case 21: Multiple complex statements combining multiple operations and variables, and dependencies

Input:

main.py | input.txt

A1 > testcases > test22 > input.txt

```
1  a = b + c + d / e * f - g - h * i - j
2  y = a + k
3  p = q + r + s + t
4  z = a * y + d + y / x - y
5  w = a - y * p / z
```

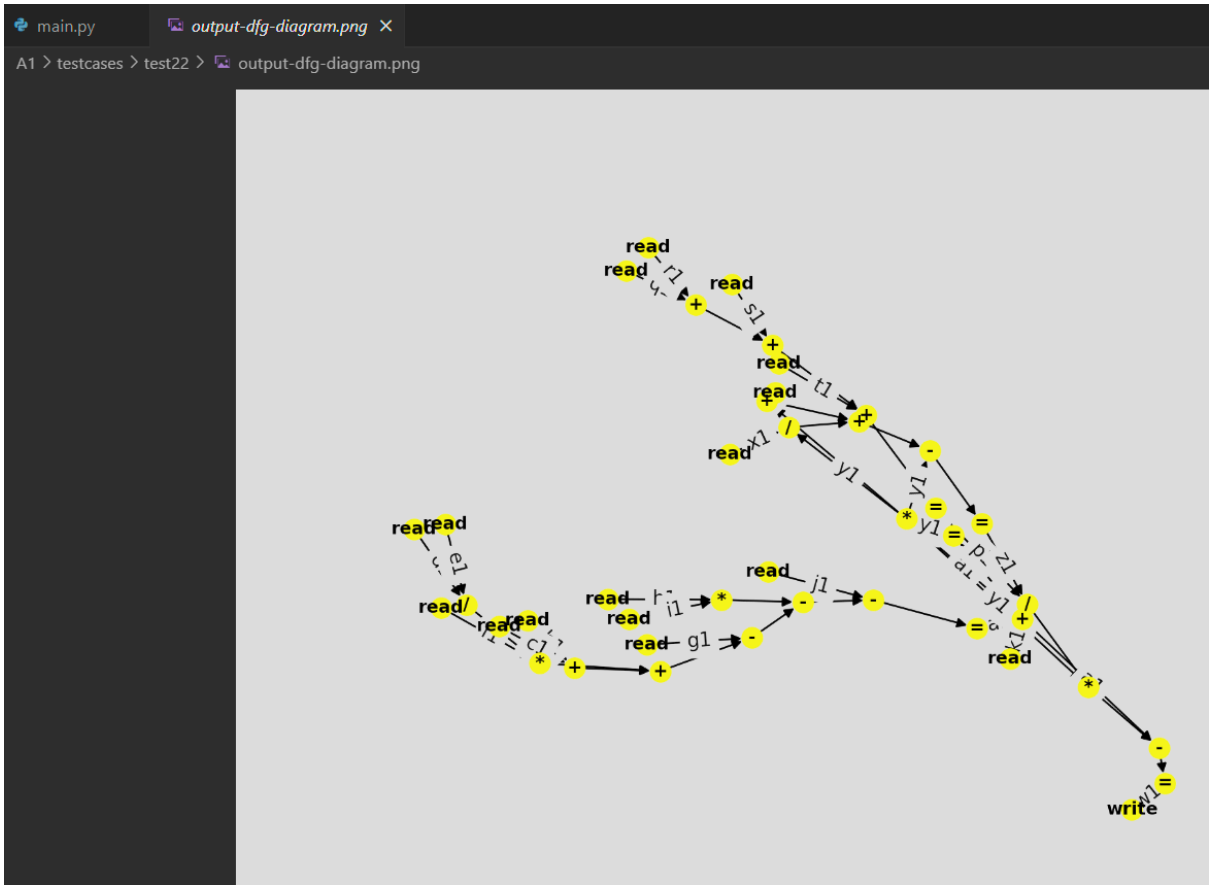
AST dictionary:

main.py | output-ast.txt

A1 > testcases > test22 > output-ast.txt


```
{"0": [["a"], ["="], [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"],
["i"]]], ["-"], ["j"]]], "1": [["y"], ["="], [{"a": [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"],
["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"], ["k"]]], "2": [["p"], ["="], [[[["q"], ["+"], ["r"]], ["+"], ["s"]], ["
+"], ["t"]]], "3": [["z"], ["="], [[[[{"a": [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]],
["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, "*"], {"y": [{"a": [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"],
["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"], ["k"]]}], ["+"], ["d"]], ["+"], [{"y": [{"a": [[[[[["b"],
["+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"],
["k"]]}, ["/"], ["x"]]], ["-"], {"y": [{"a": [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]],
["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"], ["k"]]}]], "4": [["w"], ["="], [{"a": [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"],
["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["-"], [{"y": [{"a": [[[[[["b"], ["+"],
["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"], ["k"]]},
["*"], [{"p": [[[["q"], ["+"], ["r"]], ["+"], ["s"]], ["+"], ["t"]]}, ["/"], {"z": [[[[{"a": [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"],
["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["*"], {"y": [{"a": [[[[[["b"], ["+"],
["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"], ["k"]]}], ["
+"], ["d"]], ["+"], [{"y": [{"a": [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"], ["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"],
[["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"], ["k"]]}, ["/"], ["x"]]], ["-"], {"y": [{"a": [[[[[["b"], ["+"], ["c"]], ["+"], [[["d"],
["/"], ["e"]], ["*"], ["f"]]], ["-"], ["g"]], ["-"], [["h"], ["*"], ["i"]]], ["-"], ["j"]]}, ["+"], ["k"]]}]}]}]}]}
```

DFG Diagram:



Key Map:

main.py output-keymap.txt

A1 > testcases > test22 > output-keymap.txt


```
{"1=": 4700206960, "1-": 4700208368, "2-": 4700208880, "3-": 4700213360, "1+": 4700213680, "2+": 4700214000, "read-b1": 4700214384,
"read-c1": 4700214768, "1*": 4700214832, "1/": 4700215408, "read-d1": 4700215792, "read-e1": 4700216176, "read-f1": 4700216560,
"read-g1": 4700216944, "2*": 4700217008, "read-h1": 4700225904, "read-i1": 4700226288, "read-j1": 4700226672, "2=": 4700226736, "3+":
4700227376, "read-k1": 4694540656, "3=": 4699045168, "4+": 4700228528, "5+": 4700229104, "6+": 4700229424, "read-q1": 4700238064,
"read-r1": 4700238448, "read-s1": 4700238832, "read-t1": 4700239216, "4=": 4700239280, "4-": 4700239920, "7+": 4700240496, "8+":
4700240816, "3*": 4700241136, "read-d2": 4700241584, "2/": 4700241776, "read-x1": 4700250864, "5=": 4700250608, "write-w1": 4700251568,
"5-": 4700251632, "4*": 4700252336, "3/": 4700252848}
```

DFG Adjacency list

main.py | output-dfg.txt

A1 > testcases > test22 > output-dfg.txt

```
4700206960 4700227376
4700208368 4700206960
4700208880 4700208368
4700213360 4700208880
4700213680 4700213360
4700214000 4700213680
4700214384 4700214000
4700214768 4700214000
4700214832 4700213680
4700215408 4700214832
4700215792 4700215408
4700216176 4700215408
4700216560 4700214832
4700216944 4700213360
4700217008 4700208880
4700225904 4700217008
4700226288 4700217008
4700226672 4700208368
4700226736 4700241136
4700227376 4700226736
4700227376 4700241136
4700227376 4700251632
4694540656 4700227376
4699045168 4700252848
4700228528 4699045168
4700229104 4700228528
4700229424 4700229104
4700238064 4700229424
4700238448 4700229424
4700238832 4700229104
4700239216 4700228528
4700239280 4700252848
4700239920 4700239280
4700240496 4700239920
4700240816 4700240496
4700241136 4700240816
4700241136 4700241776
4700241136 4700239920
4700241136 4700252336
4700241584 4700240816
4700241776 4700240496
4700250864 4700241776
4700251568 4700250608
4700251632 4700250608
4700252336 4700251632
4700252848 4700252336

```