Software Testing, Verification and Validation

November 16, 2022
Week #10— Lecture #7

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Last week, we introduced condition coverage, condition + decision coverage, path coverage, and modified condition/decision coverage as part of our set of white-box techniques and closed the structural/logical coverage topic.

This week we will introduce dataflow coverage in particular: All Defs, All Uses, All Def-Uses-Paths, All P-Uses/Some C-Uses, All C-Uses/Some P-Uses, All P-Uses, and All C-Uses.

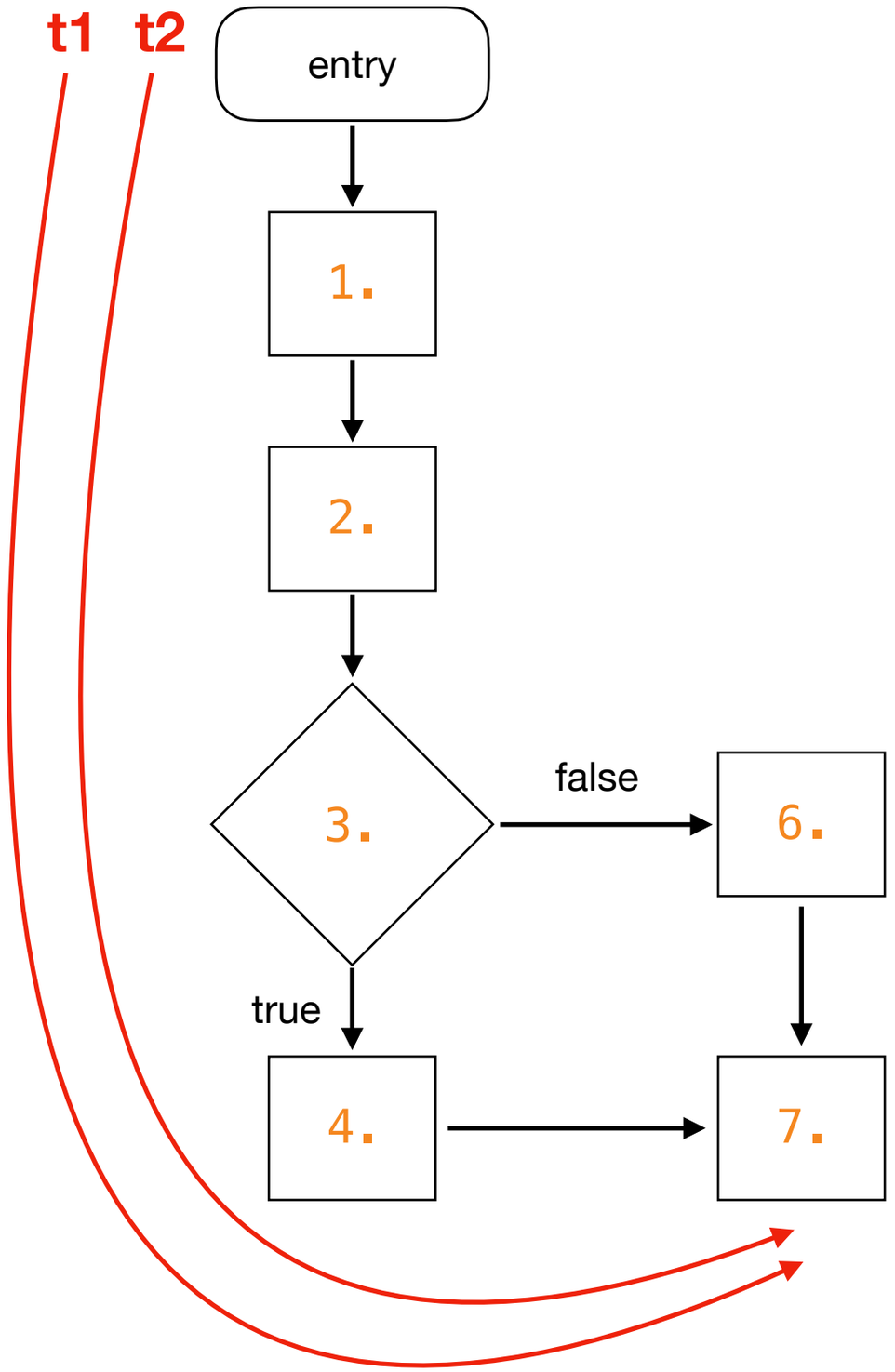
Dataflow Coverage

Are t1 and t2 identical? Although the paths are the same, different tests may have different variable values defined/used.

- Existing control-flow coverage criteria only consider the execution path (structure).

- In the program paths, which variables are defined and then used should also be covered (data).

- A family of dataflow criteria is then defined, each providing a different degree of data coverage.



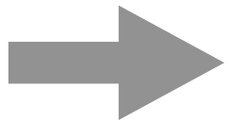
Dataflow Coverage

- Considers how data gets accessed and modified in the system and how it could get corrupted.

- Common access-related *bugs*:

- Using an undefined or uninitialized variable.

```
...
List l;
...
l.add(6); = y + 2
```



```
Exception in thread "main"
java.lang.NullPointerException
```

Dataflow Coverage

- Considers how data gets accessed and modified in the system and how it could get corrupted.

- Common access-related *bugs*:

  - Using an undefined or uninitialized variable.

  - Deallocating or reinitializing a variable before it is constructed, initialized, or used.

  - Deleting a collection object leaving its members unaccessible (garbage collection helps here).

Variable Definition

A program variable is DEFINED whenever its value is modified:

- on the left hand side of an assignment statement, e.g., `y = 17`.

- in an input statement, e.g., `read(y)`.

- as a call-by-reference parameter in a subroutine call, e.g., `update(x, &y)`.

Variable Use

A program variable is USED whenever its value is read:

- on the right hand side of an assignment statement, e.g., `y = x + 17`.

- as a call-by-value parameter in a subroutine or function call, e.g., `y = sqrt(x)`.

- in the predicate of a branch statement, e.g.,

```
if (x > 0) { … }.
```

Variable Use: p-use and c-use

- A variable used in the predicate of a branch statement is a predicate-use or "p-use".

- Any other use is a computation-use or "c-use".

```
if (x > 0) {
    print(y);
}
```

There is a p-use of x and a c-use of y.

Variable Use

A variable can also be used and then re-defined in a single statement when it appears:

- on both sides of an assignment statement, e.g.,

```
y = y + x.
```

- as a call-by-reference parameter in a subroutine call, e.g.,

```
increment(&y).
```

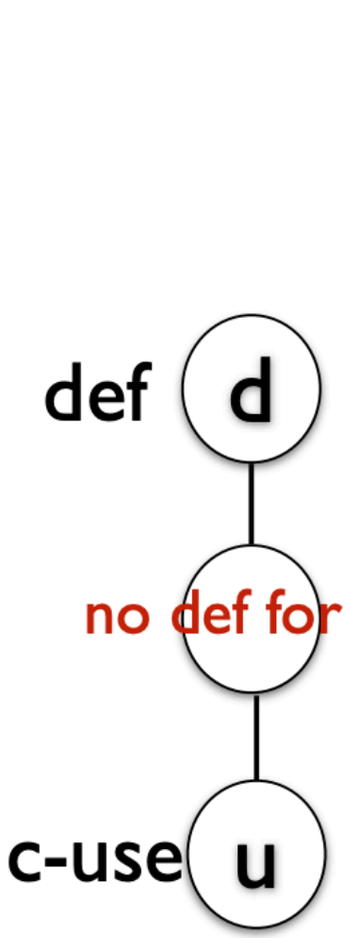
Dataflow Graph

- A data-flow graph of a program, also known as def-use graph, captures the flow of definitions (also known as defs) across basic blocks in a program.
- It is similar to a control flow graph of a program in that the nodes, edges, and all paths through the control flow graph are preserved in the data flow graph.

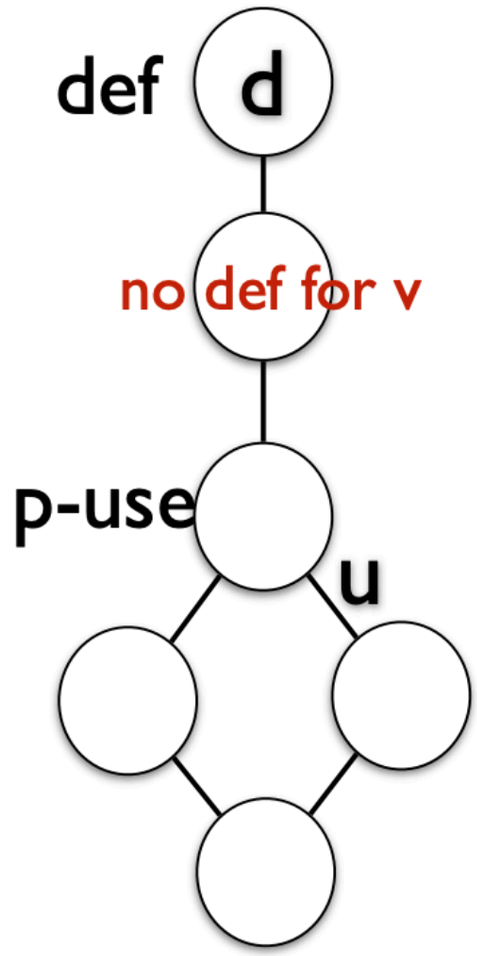
Def-Use Pair

A definition-use pair ("def-use" for short) with respect to a variable **v** is a pair (**d, u**) such that:

- **d** is a node defining **v**
- **u** is a node or edge using **v**
  - when it is a p-use of **v**, **u** is an outgoing edge of the predicate statement
- There is a def-clear path with respect to v from **d** to **u**



c-use example



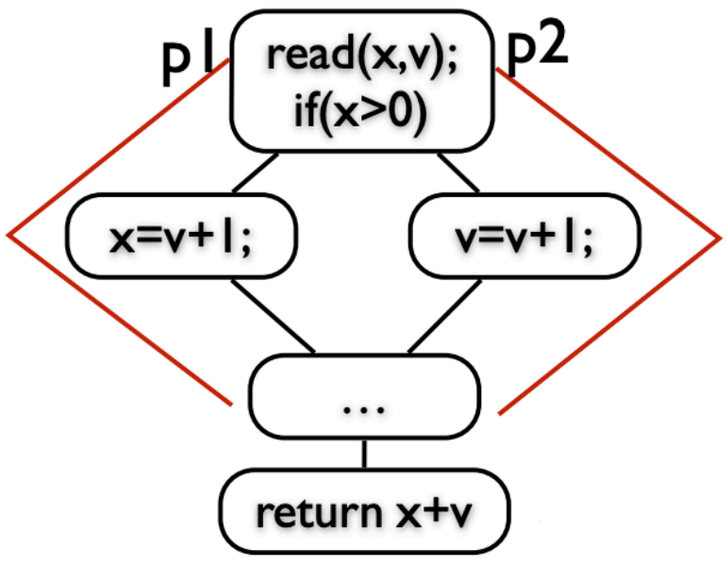
p-use example

Def-Clear

- A path is definition clear (“def-clear”) with respect to a variable **v** if it has no variable re-definition of **v** on the path. In other words, any path starting from a node at which variable **v** is defined and ending at a node at which **v** is used, without redefining **v** anywhere else along the path, is a def-clear path for **v**.

Def-Clear

- A path is definition clear ("def-clear") with respect to a variable **v** if it has no variable re-definition of **v** on the path. In other words, any path starting from a node at which variable **v** is defined and ending at a node at which **v** is used, without redefining **v** anywhere else along the path, is a def-clear path for **v**.



p1 is def-clear for **v**, while p2 is not

Procedure

- Given a program, construct its control-flow graph.

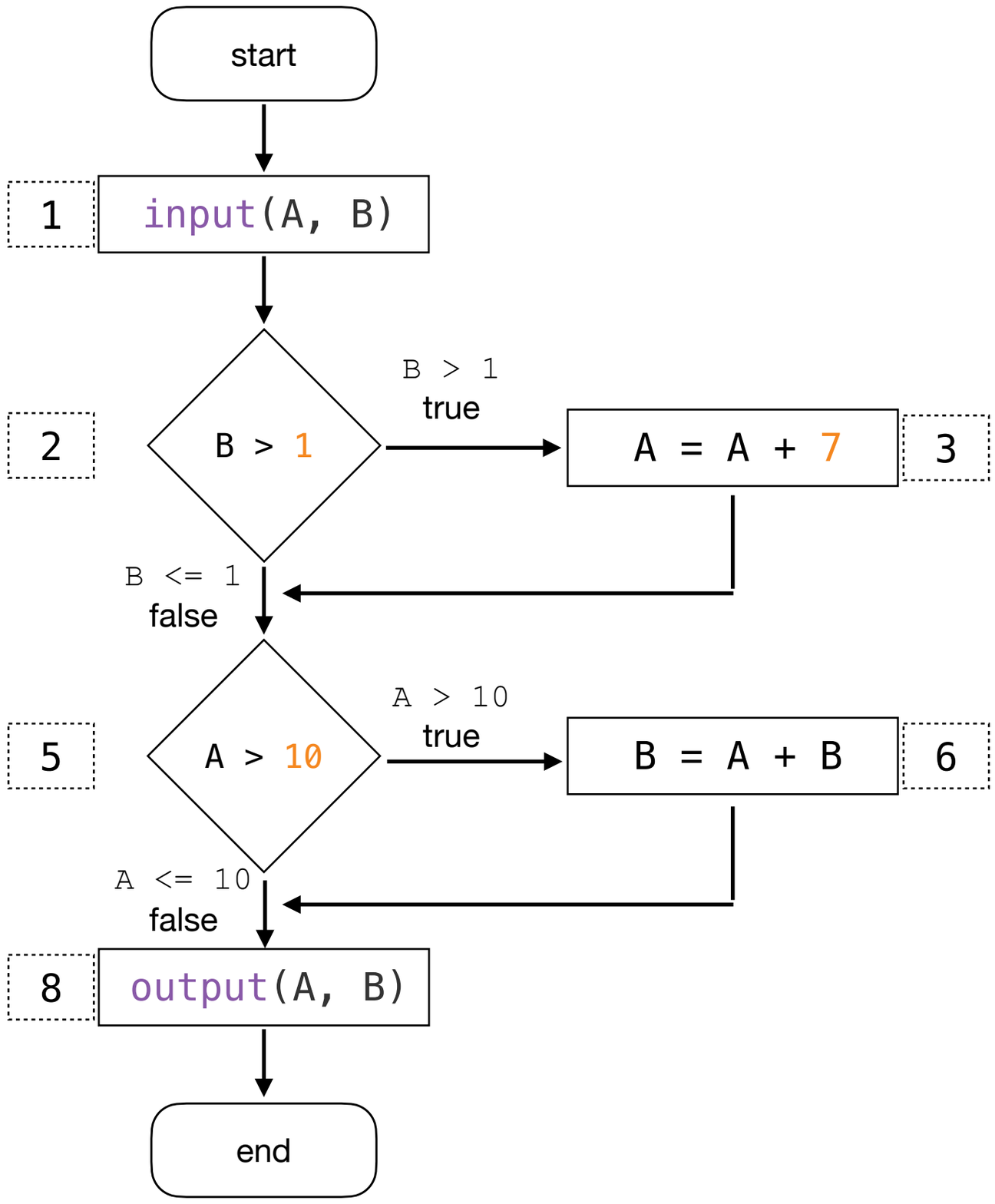
- For each block/node, identify and classify occurrences of all the variables in the software under test. In other words, compute defs, c-uses and p-uses in each block. Each block becomes a node in the def-use graph (this is similar to the control flow graph).

- Attach defs, c-use and p-use to each node in the graph. Label each edge with the condition which when true/false causes the edge to be taken.

- Create a tabular summary for each variable and derive test data such that all definitions and all uses for all of the variables are exercised during test.

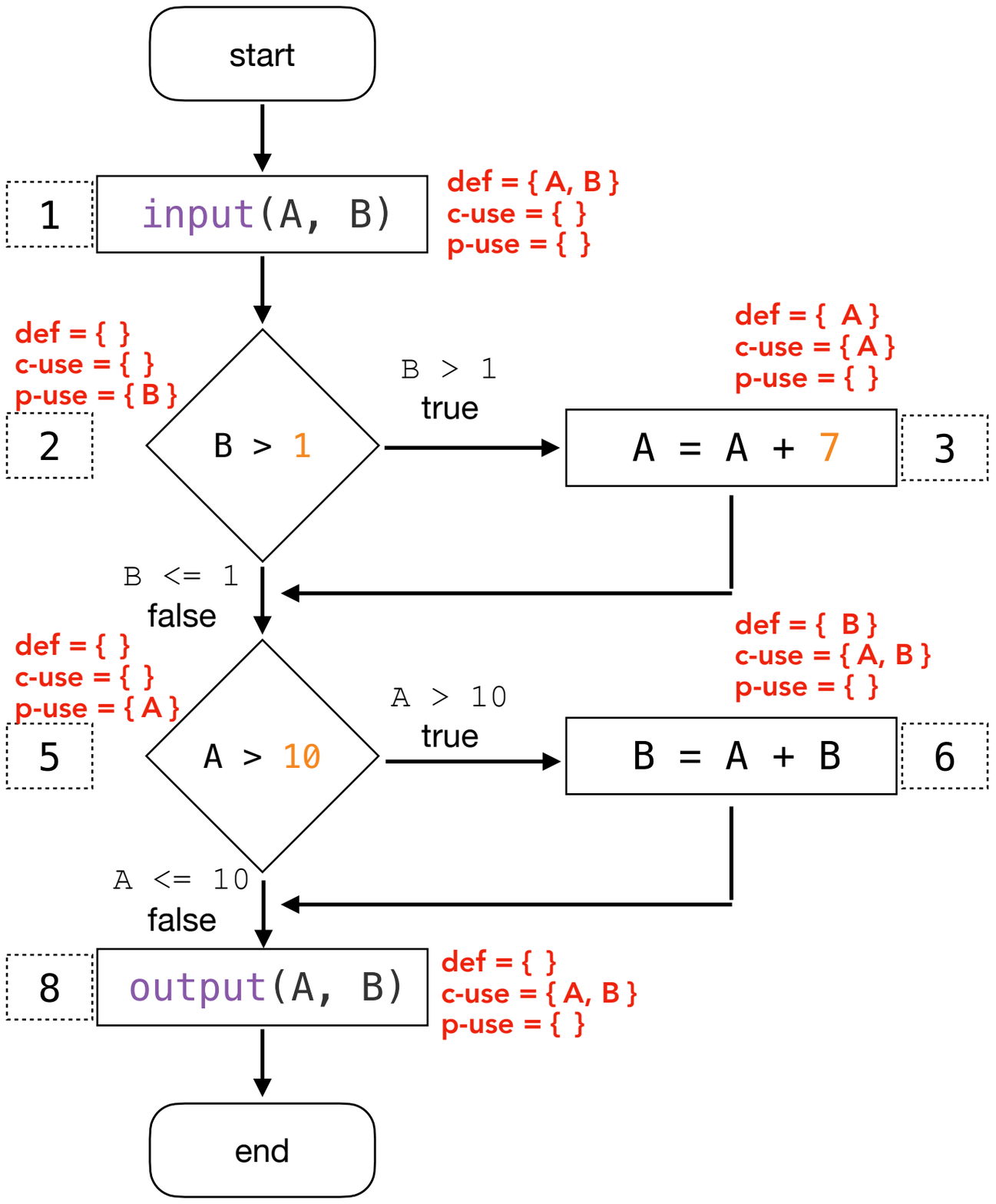
```
input(A, B)
if (B > 1) {
    A = A + 7
}
if (A > 10) {
    B = A + B
}
output(A, B)
```

```
1   input(A, B)
2   if (B > 1) {
3       A = A + 7
4   }
5   if (A > 10) {
6       B = A + B
7   }
8   output(A, B)
```



The variables of interest are **A** and **B**.

```
1   input(A, B)
2   if (B > 1) {
3       A = A + 7
4   }
5   if (A > 10) {
6       B = A + B
7   }
8   output(A, B)
```

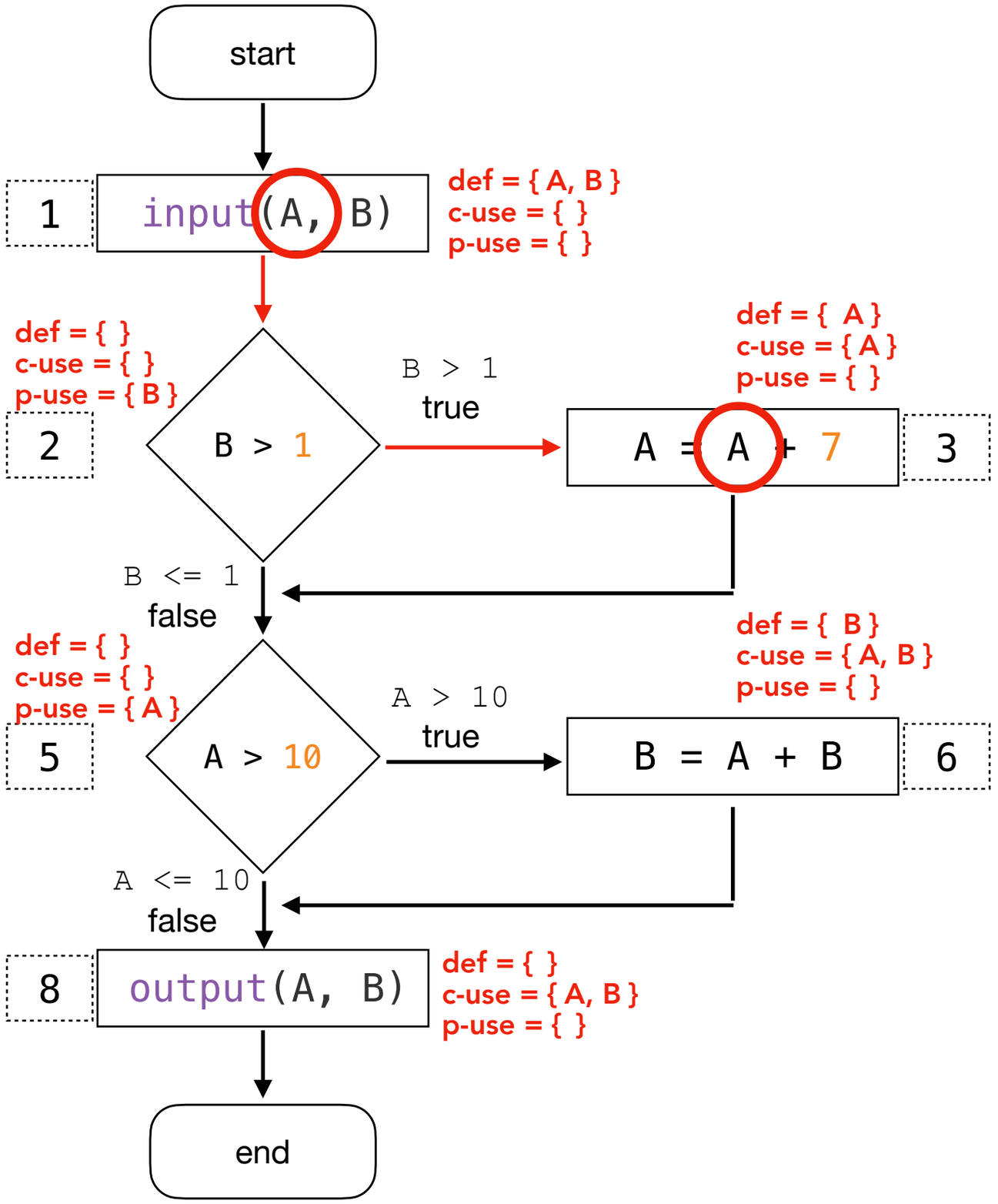


Identifying *def-use pairs* for variable `A`

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>

`A` defined in 1 and c-used in 3 through <1,2,3>.

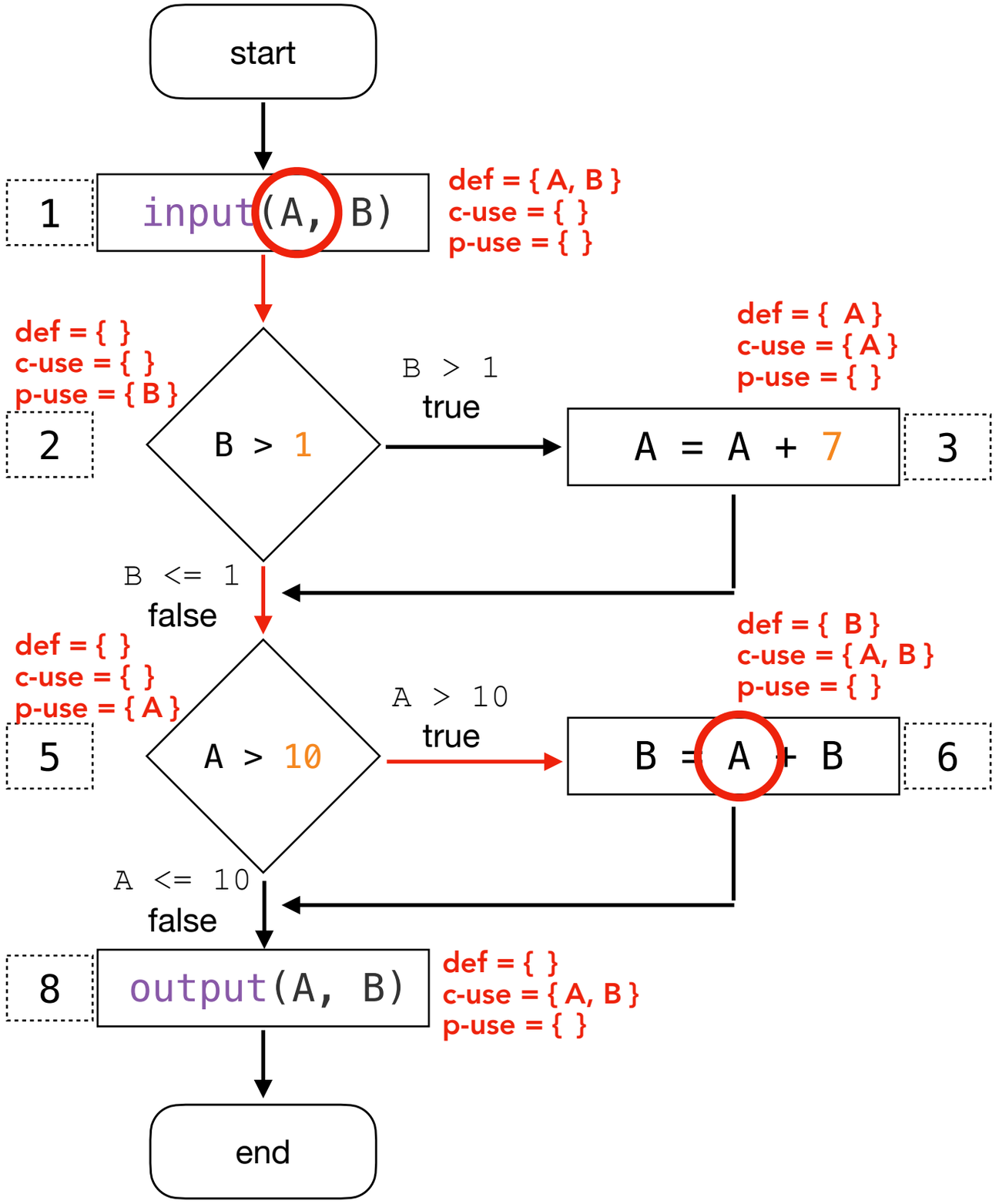


Identifying *def-use pairs* for variable `A`

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>

`A` defined in 1 and c-used in 6 through <1,2,5,6>.

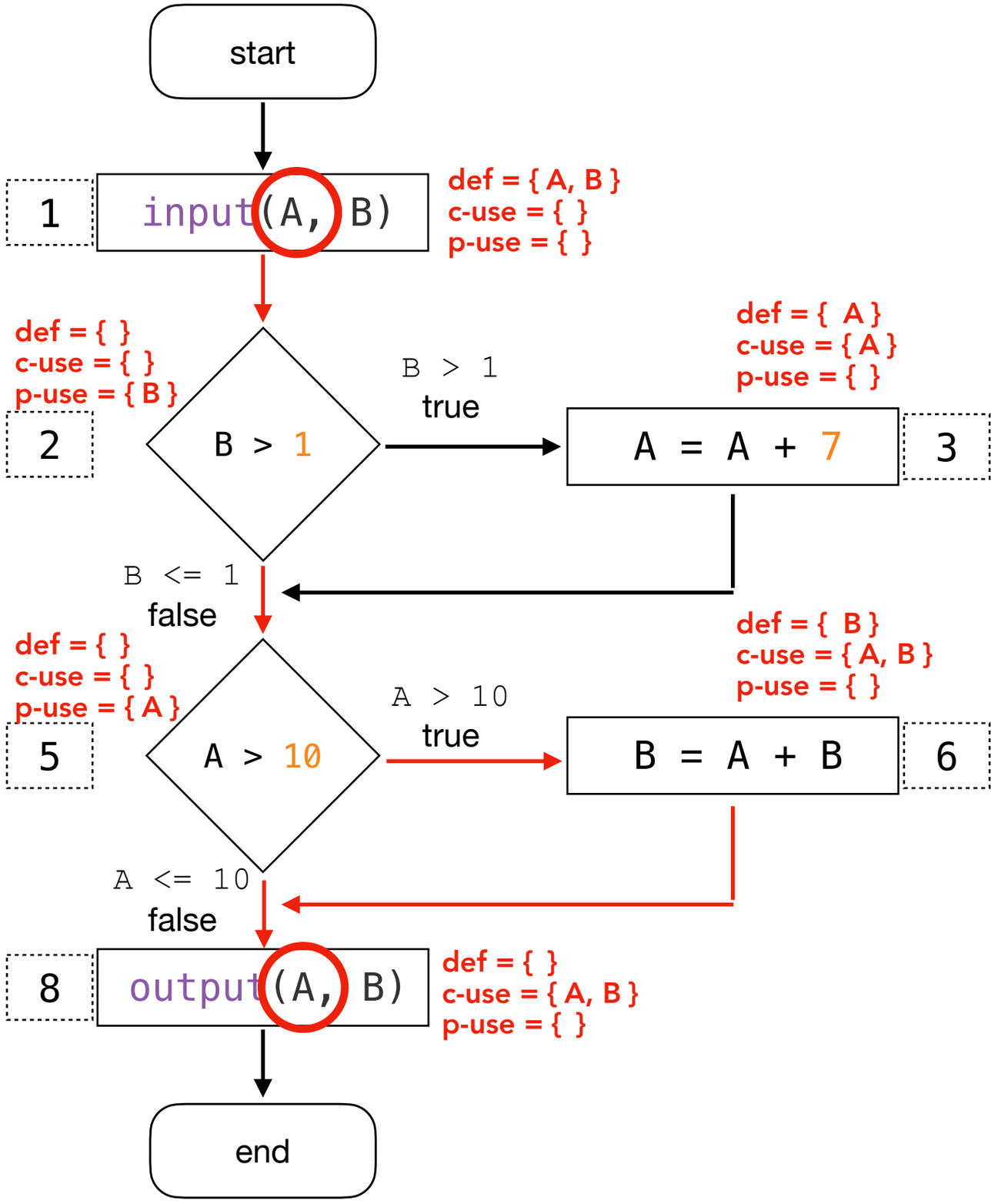


Identifying *def-use pairs* for variable `A`

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>

`A` defined in 1 and c-used in 8 through <1,2,5,6,8>.

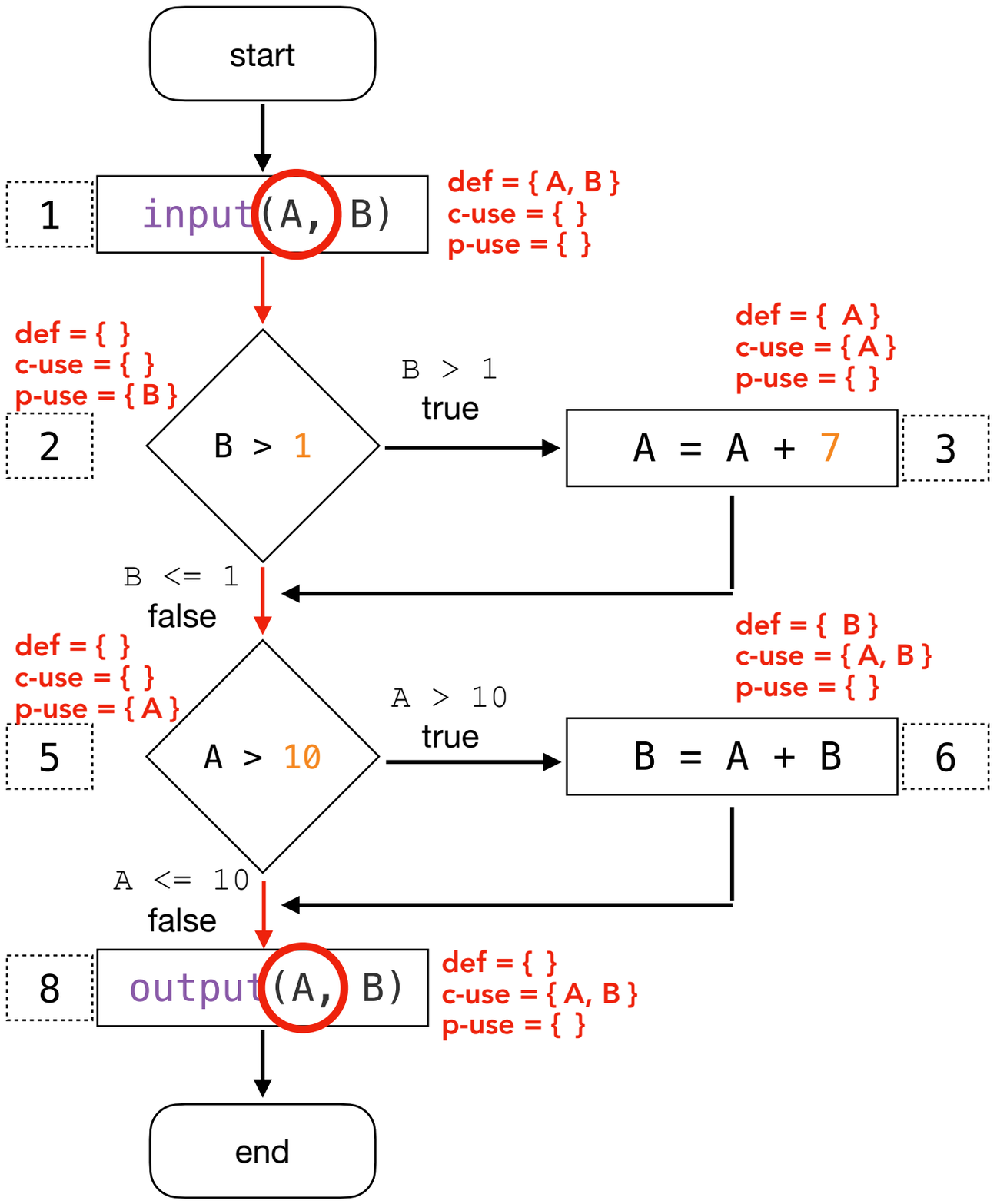


Identifying *def-use pairs* for variable `A`

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>

`A` defined in 1 and c-used in 8 through <1,2,5,8>.



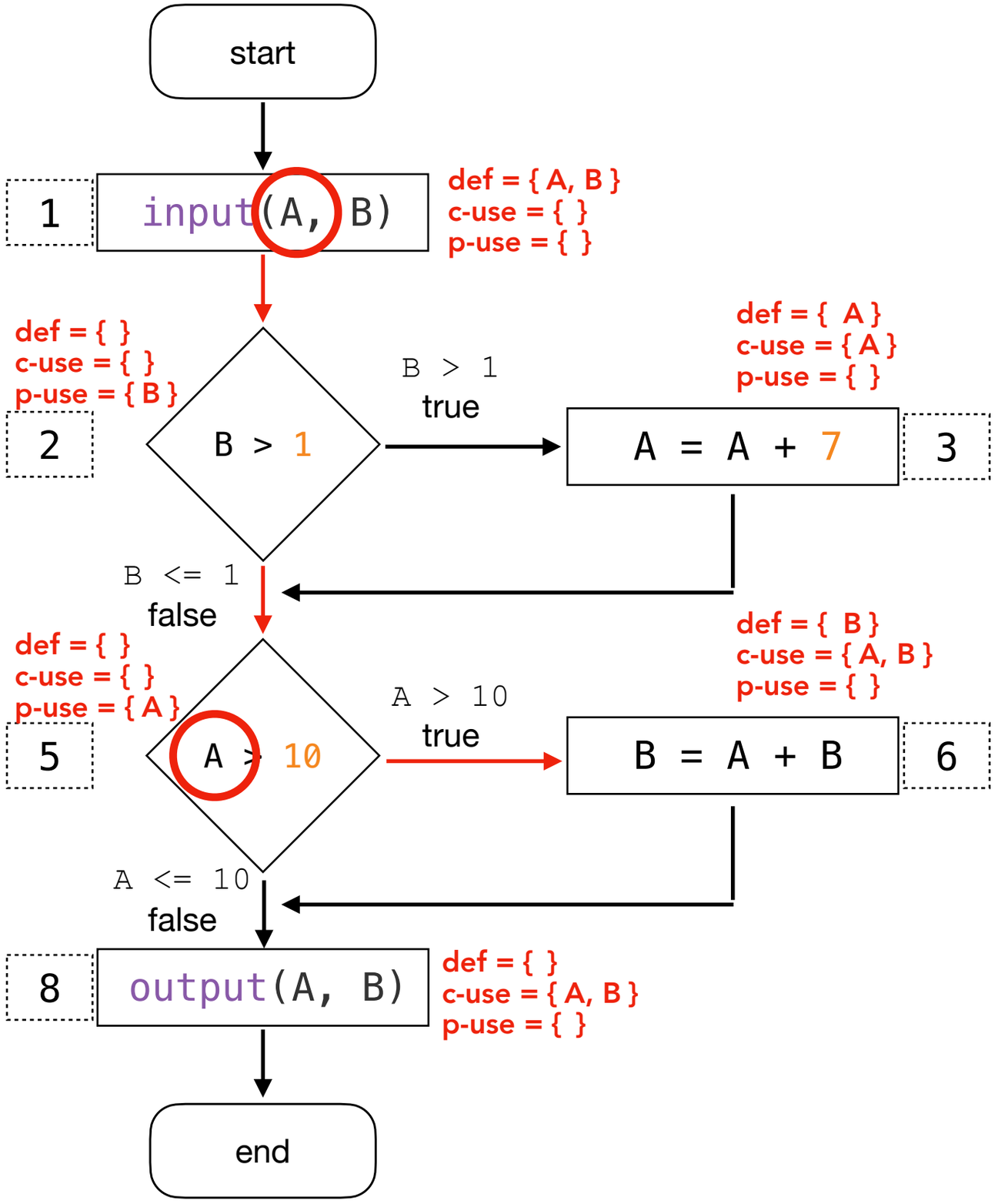
Identifying *def-use pairs* for variable `A`

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>

`A` defined in 1 and p-used in 5.

When it is a p-use of v, u is an outgoing edge of the predicate statement, i.e., True.



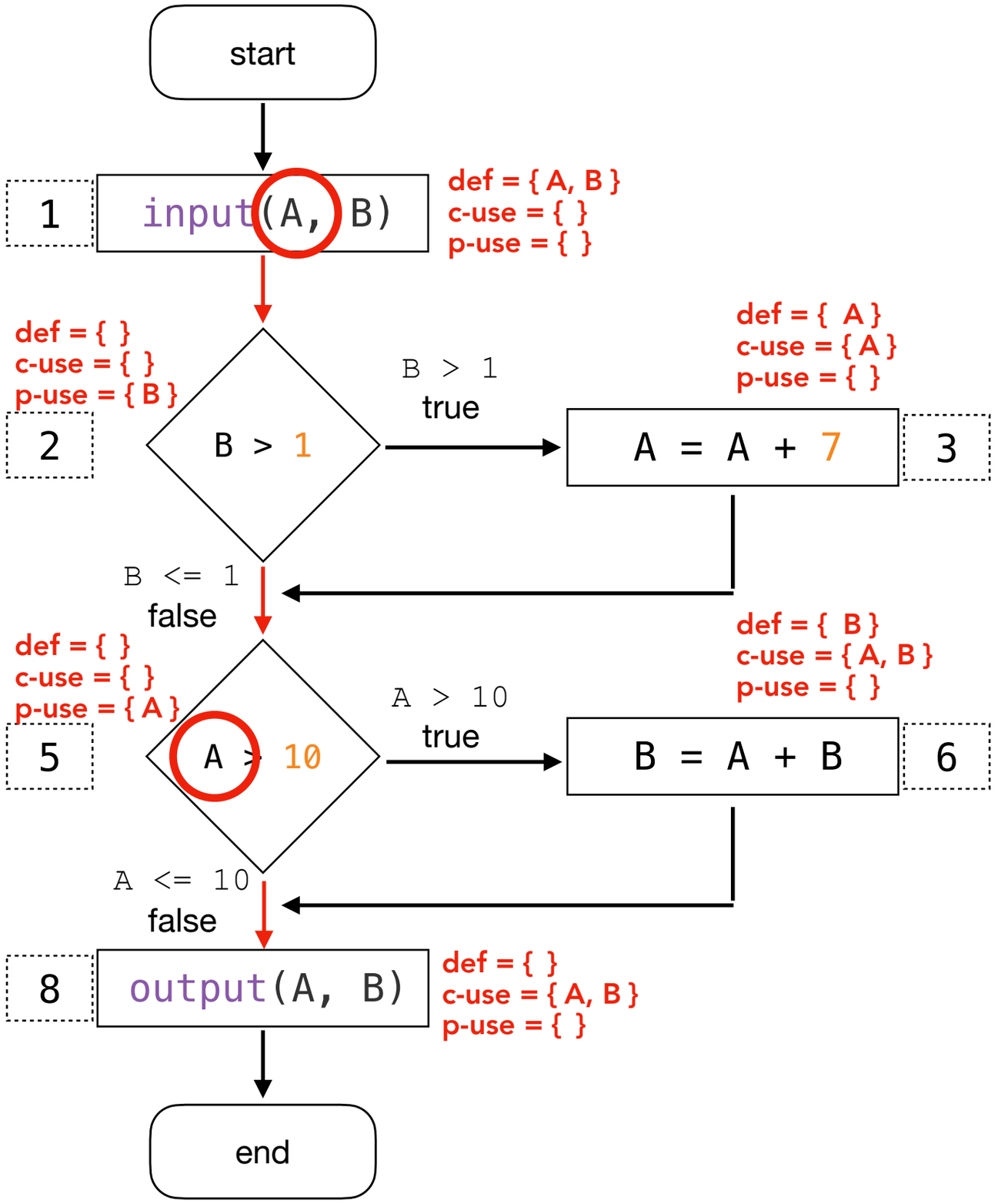
Identifying *def-use pairs* for variable `A`

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>

`A` defined in 1 and p-used in 5.

When it is a p-use of v, u is an outgoing edge of the predicate statement, i.e., False.

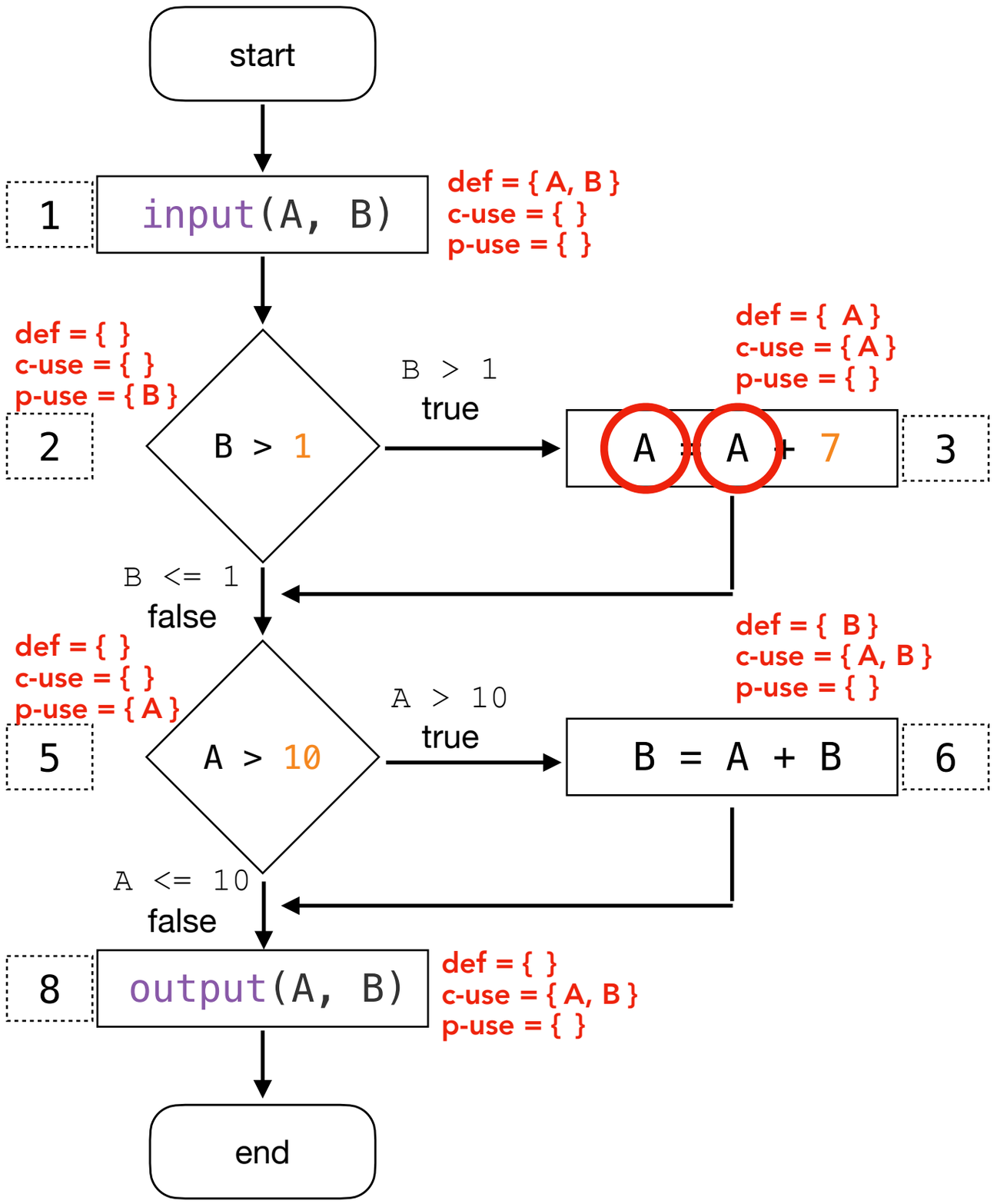


Identifying *def-use pairs* for variable `A`

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>

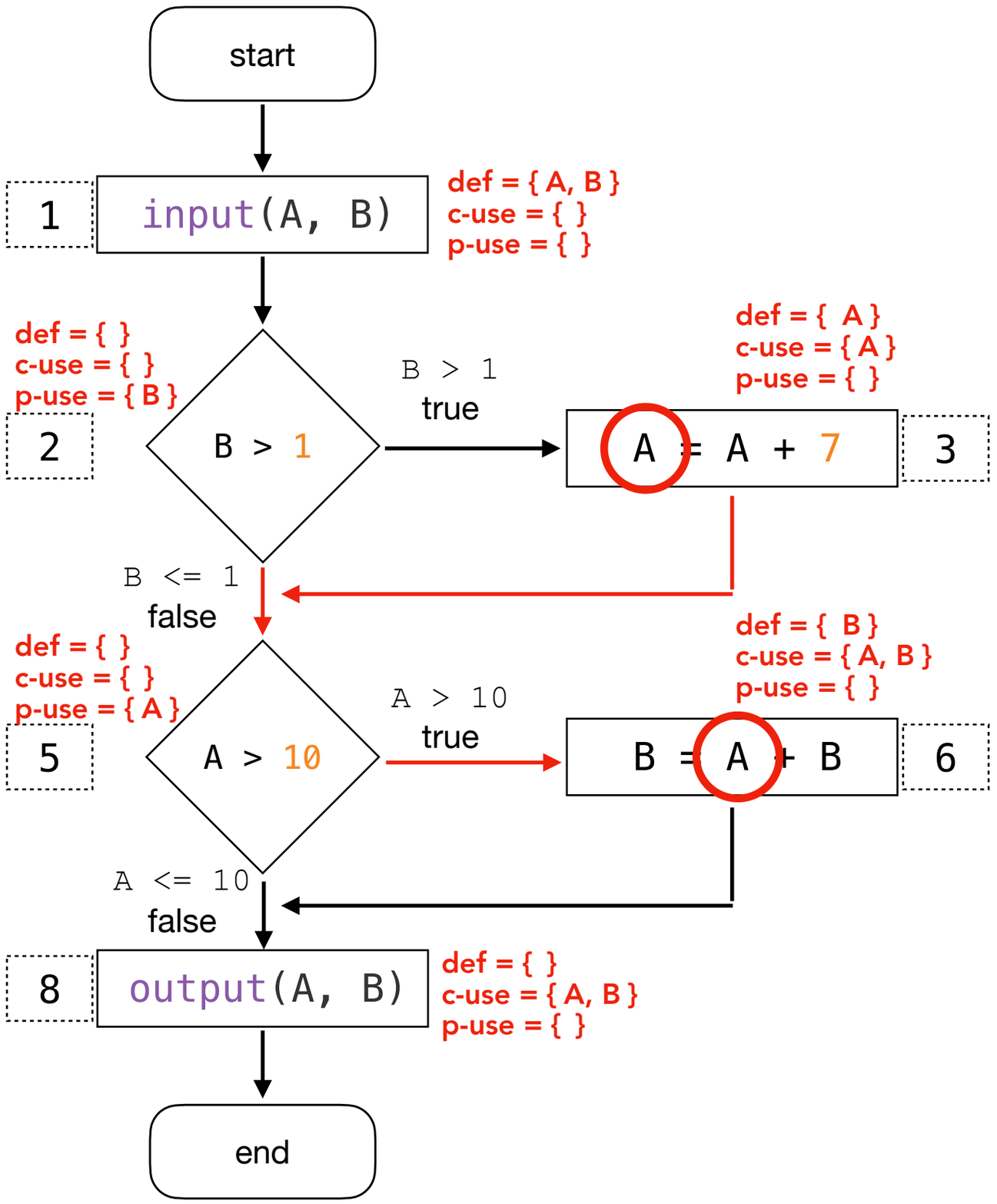
`A` defined in 3 and c-used in 3 through <3,3>.



Identifying *def-use pairs* for variable `A`

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>

`A` defined in 3 and c-used in 6 through <3,5,6>.

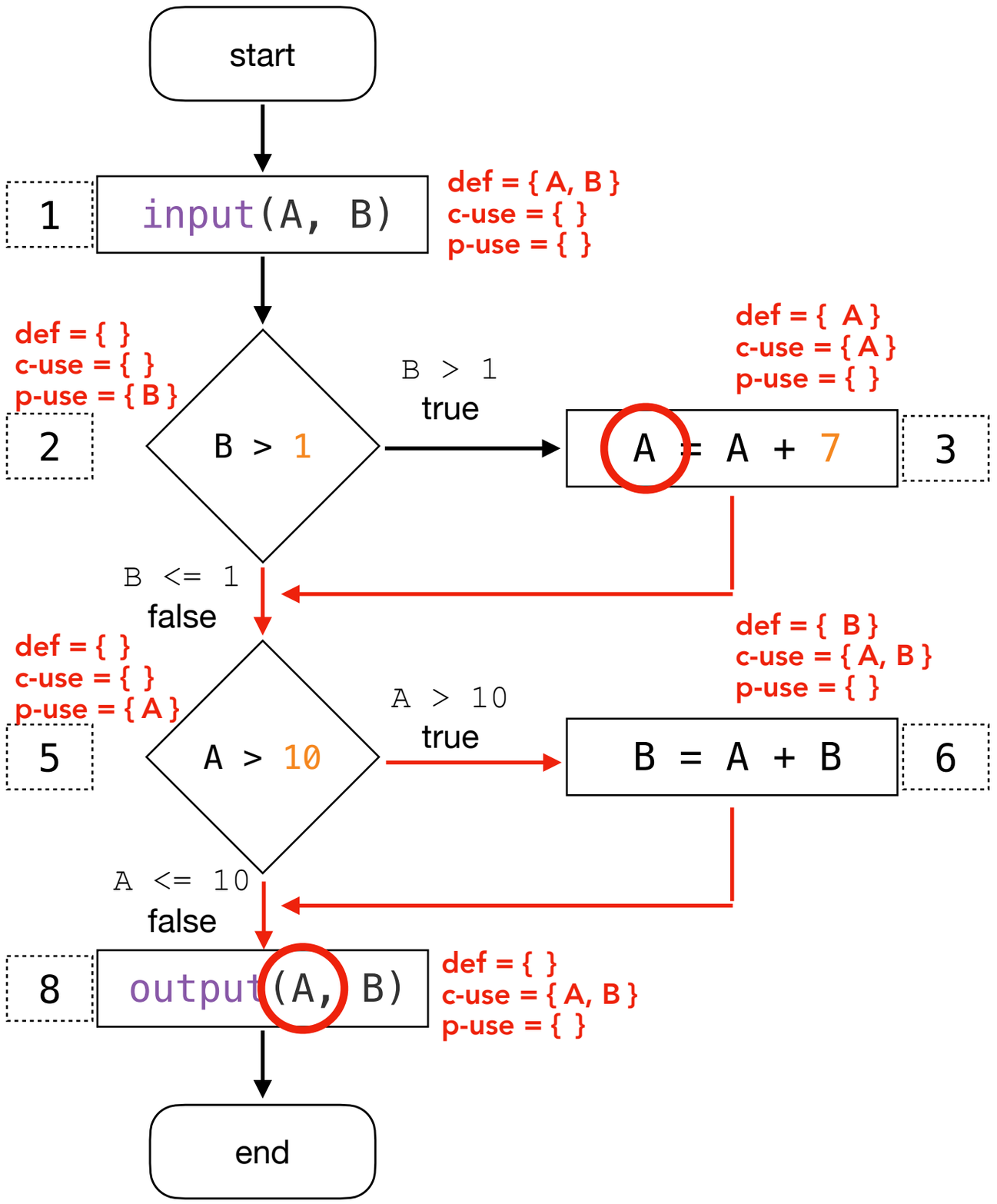


Identifying *def-use pairs* for variable `A`

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>

`A` defined in 3 and c-used in 8 through <3,5,6,8>.

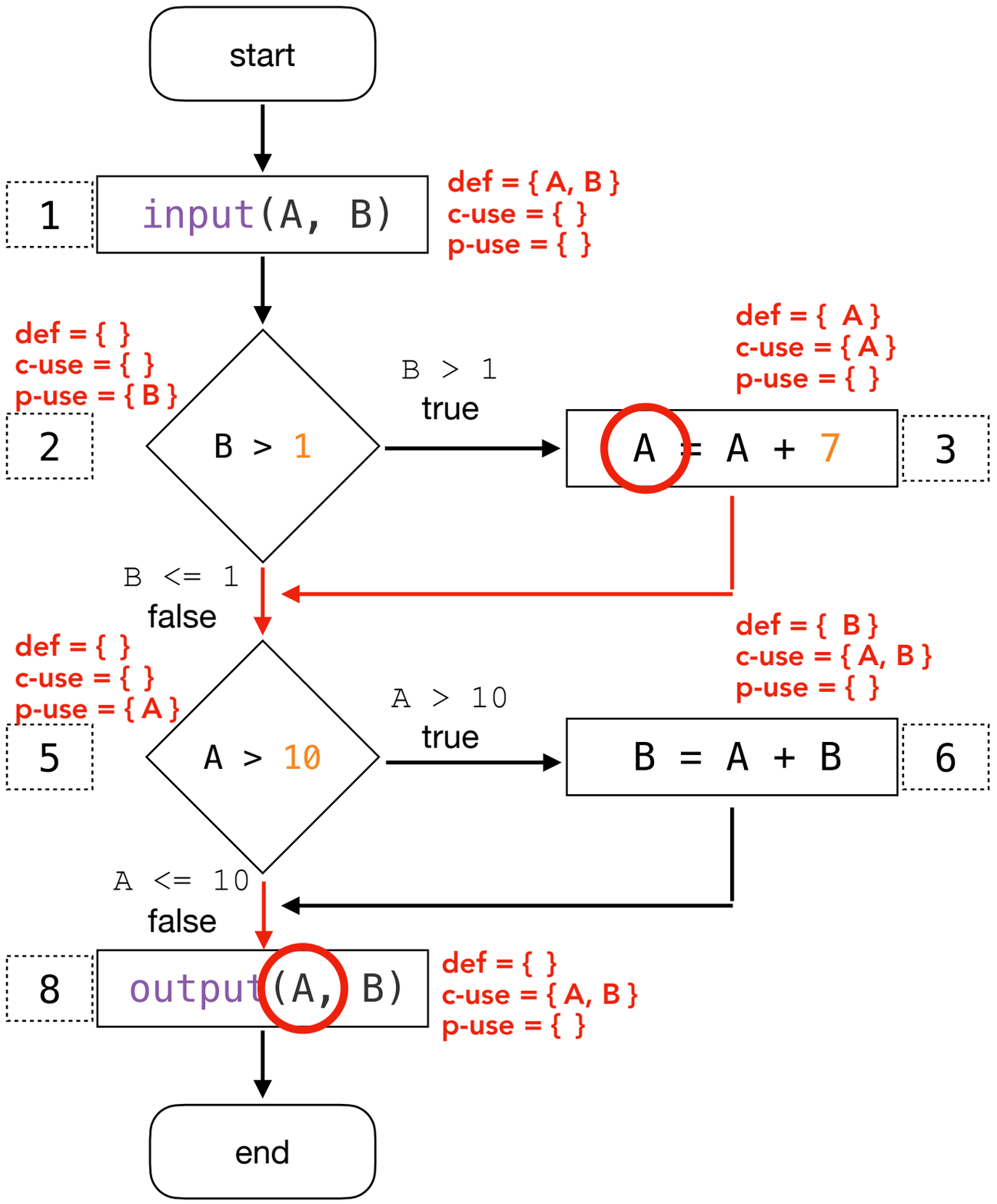


Identifying *def-use pairs* for variable `A`

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>

`A` defined in 3 and c-used in 8 through <3,5,8>.



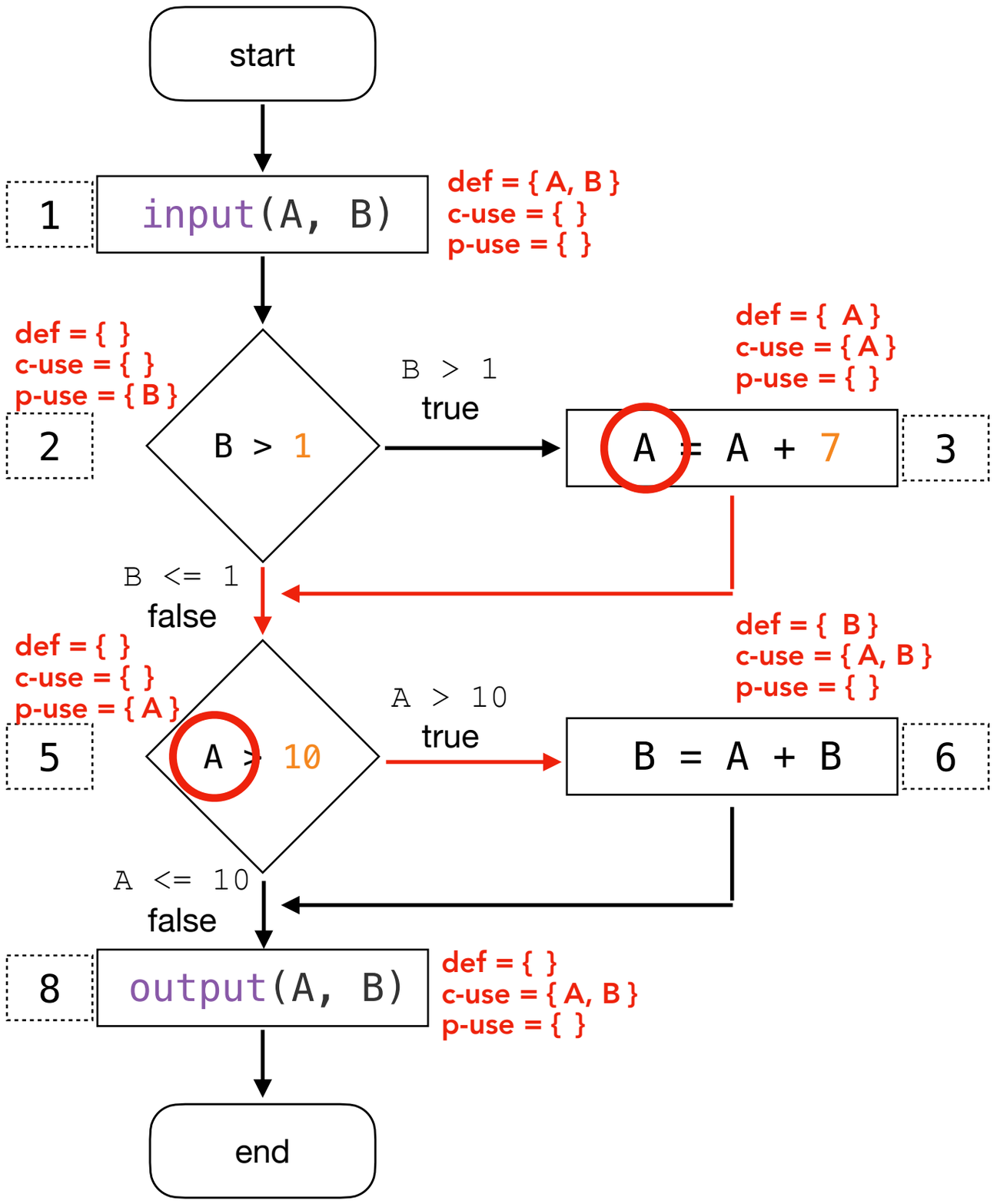
Identifying *def-use pairs* for variable `A`

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>

`A` defined in 3 and p-used in 5.

When it is a p-use of v, u is an outgoing edge of the predicate statement, i.e., True.



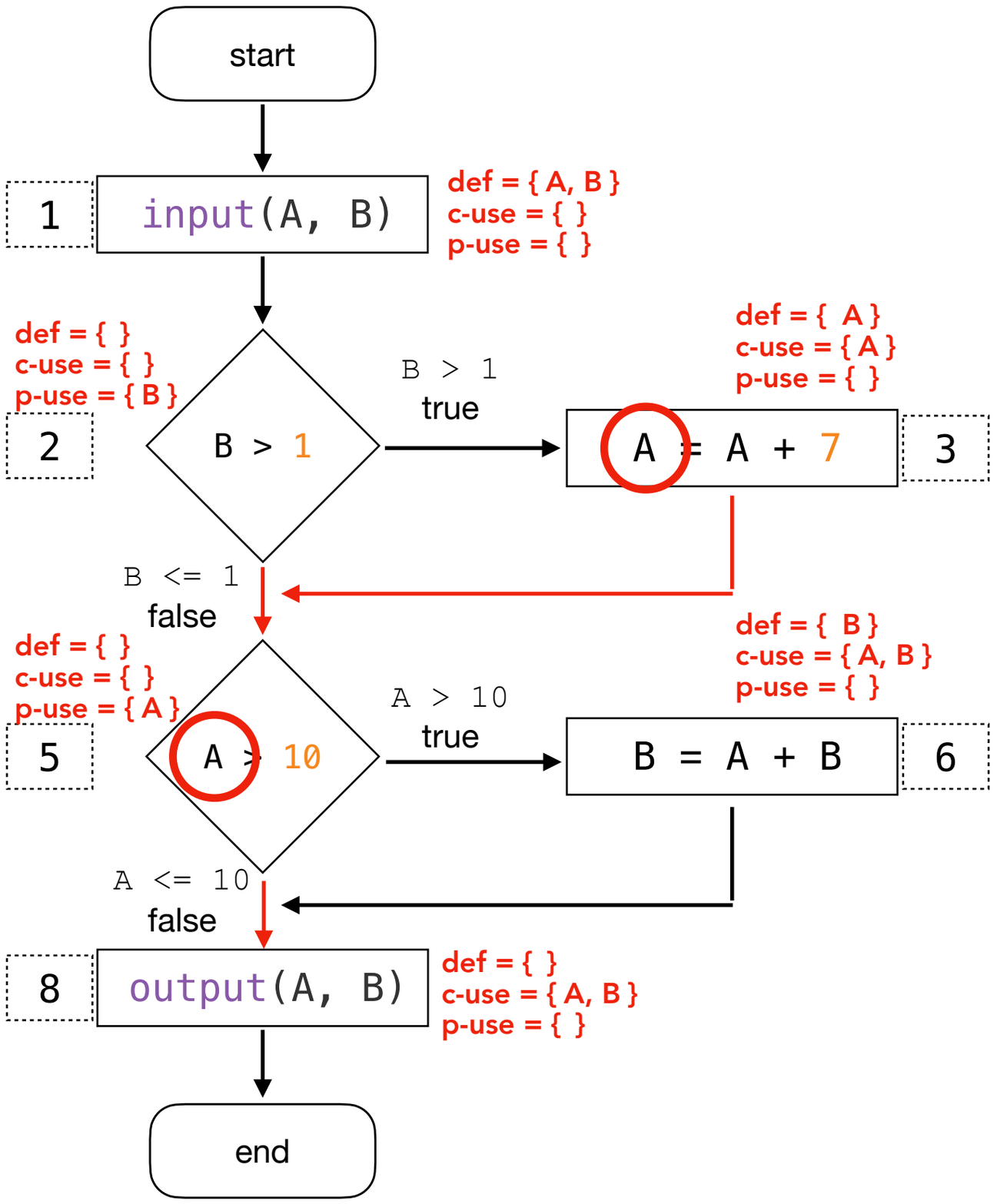
Identifying *def-use pairs* for variable `A`

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>
12	3	(5,F)	<3,5,8>

`A` defined in 3 and p-used in 5.

When it is a p-use of v, u is an outgoing edge of the predicate statement, i.e., False.

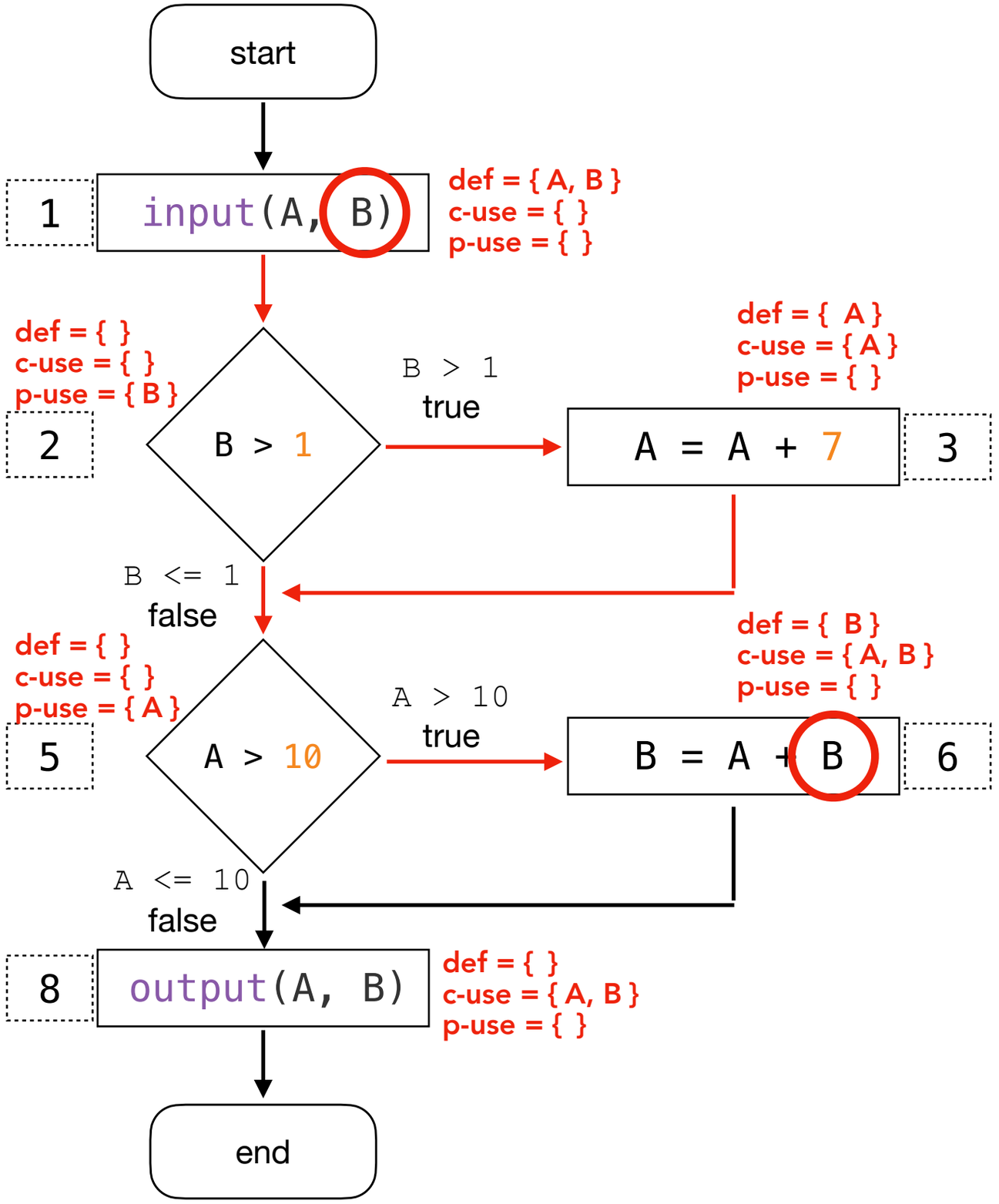


Identifying *def-use pairs* for variable `B`

Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>

`B` defined in 1 and c-used in 6 through <1,2,3,5,6>.

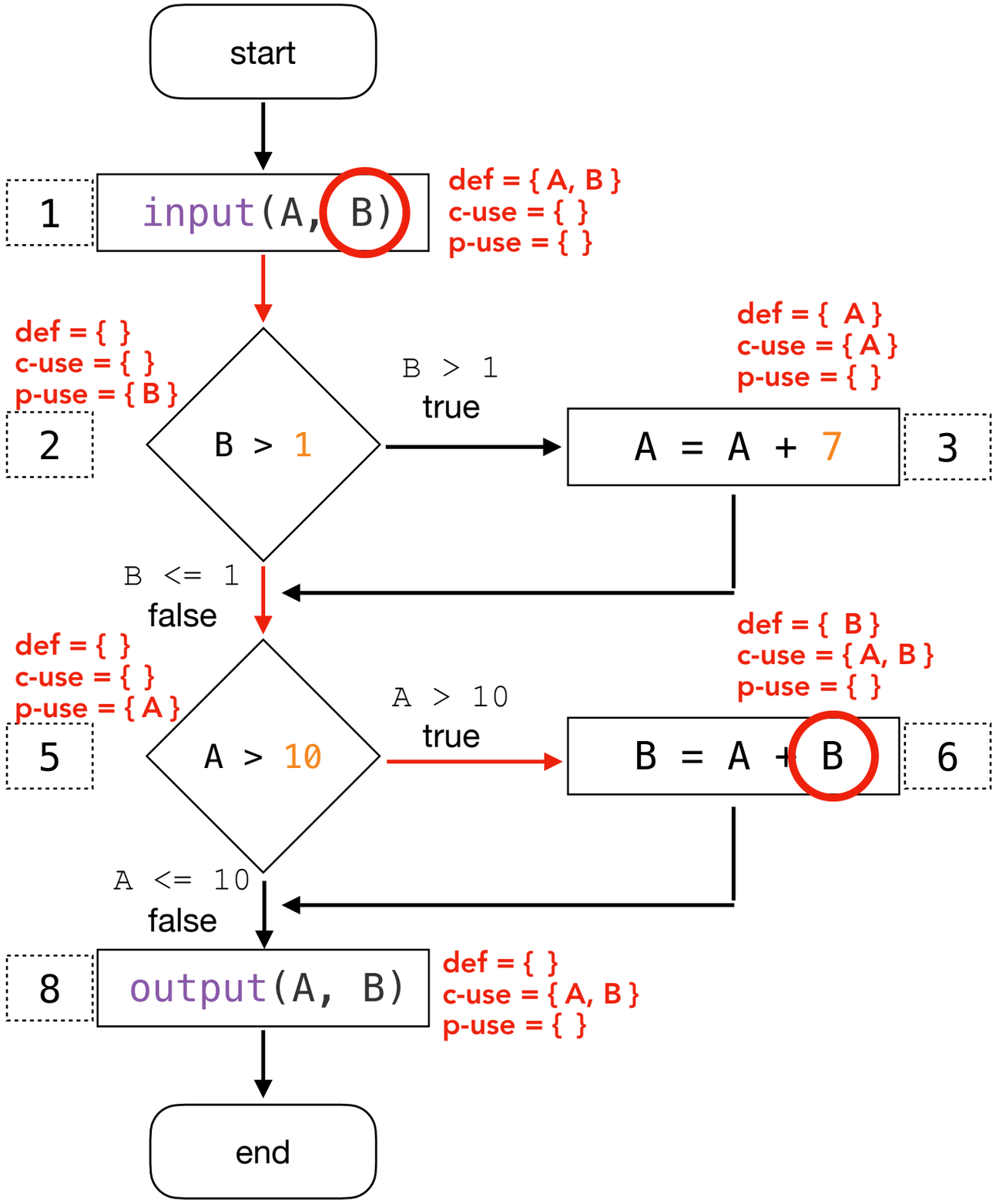


Identifying *def-use pairs* for variable `B`

Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>

`B` defined in 1 and c-used in 6 through <1,2,5,6>.

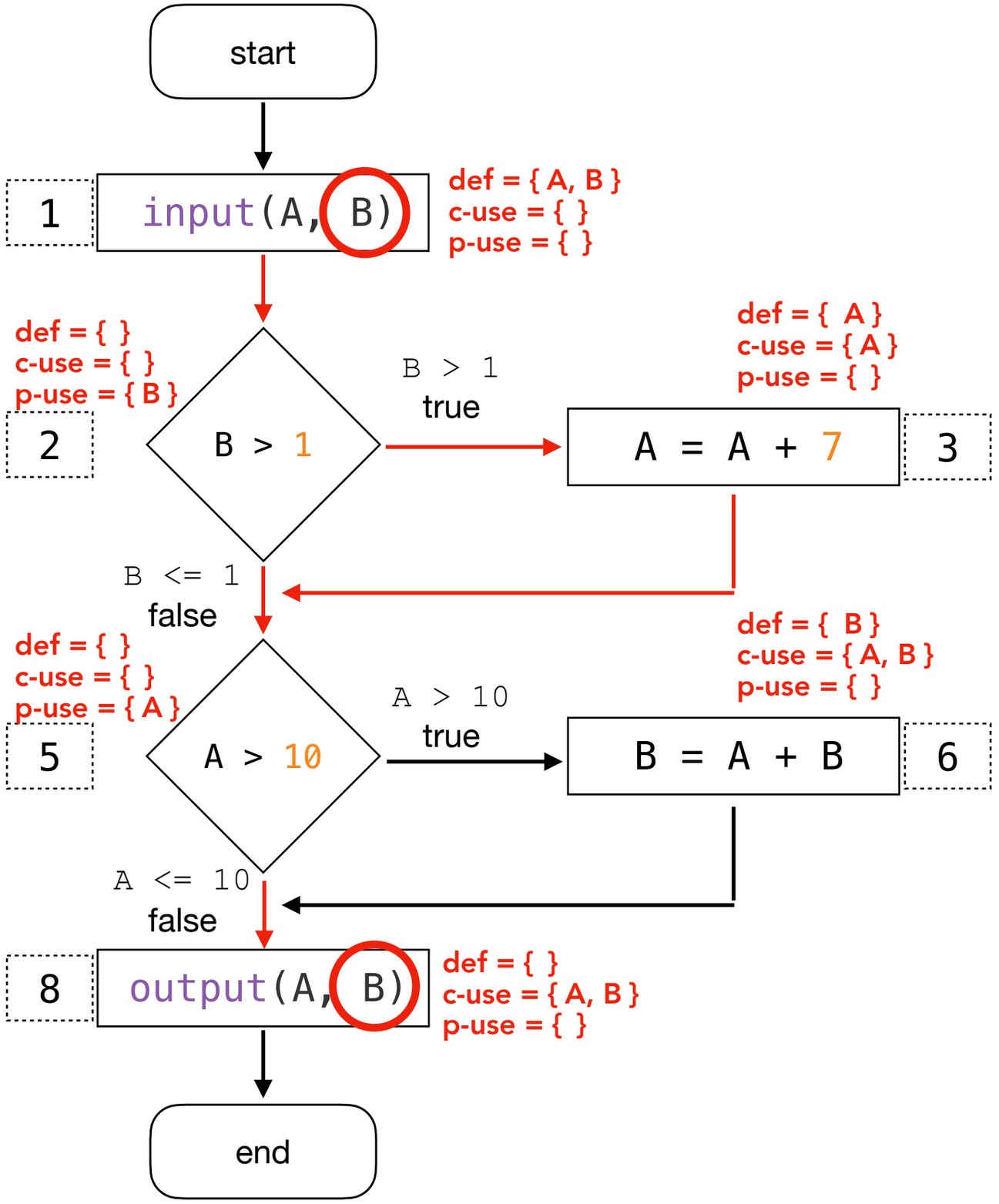


Identifying *def-use pairs* for variable `B`

Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>

`B` defined in 1 and c-used in 8 through <1,2,3,5,8>.

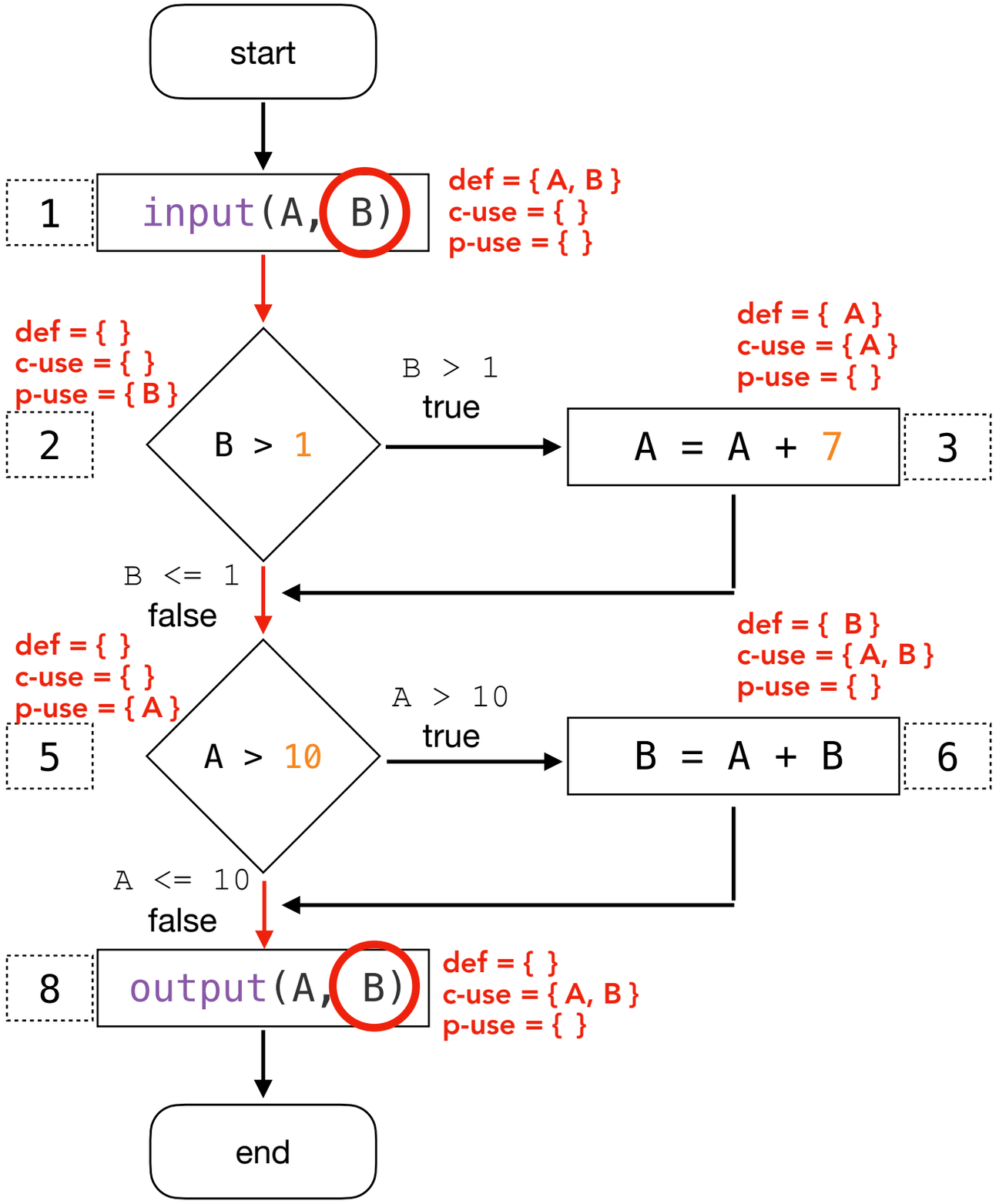


Identifying *def-use pairs* for variable `B`

Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>

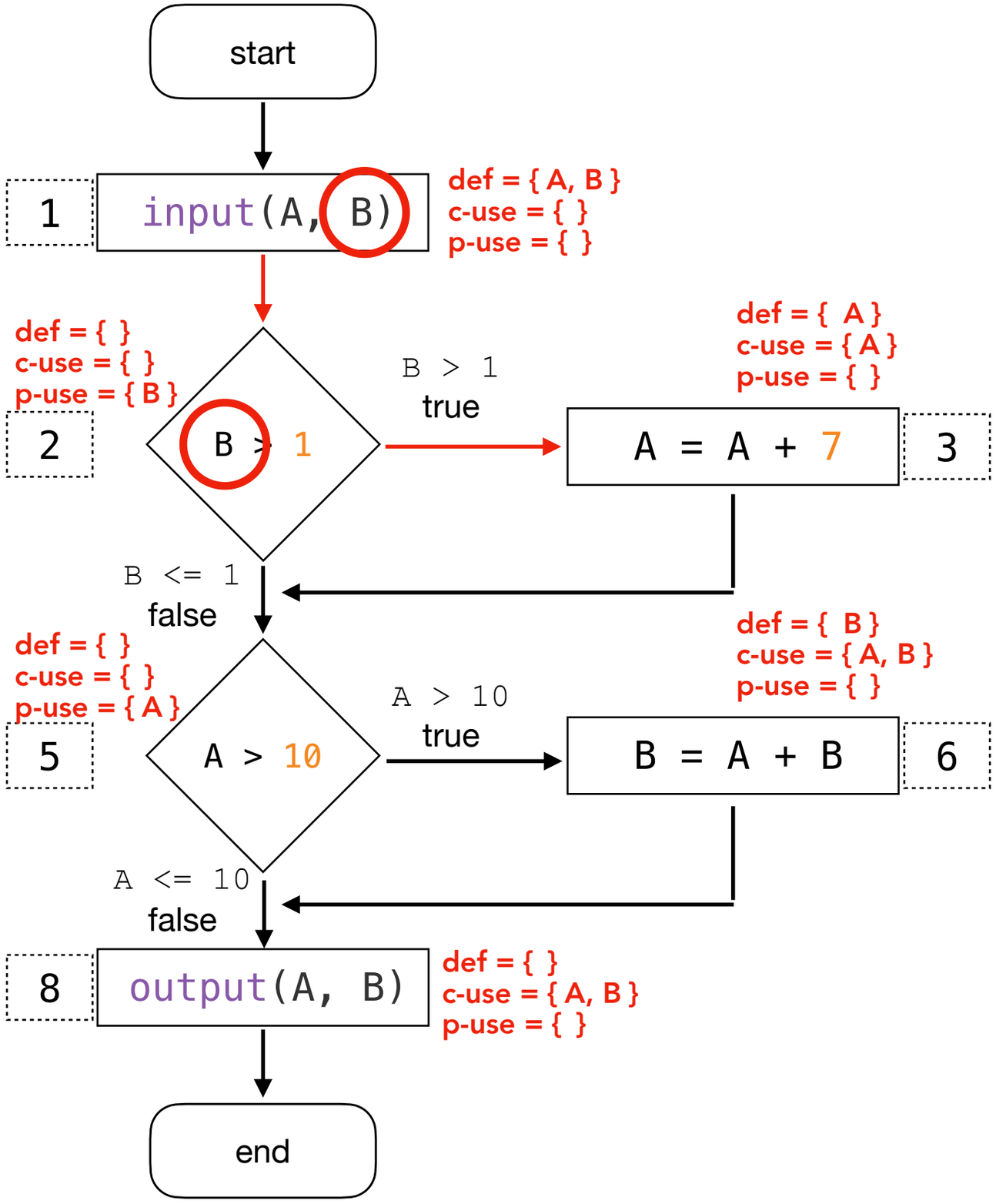
`B` defined in 1 and c-used in 8 through <1,2,5,8>.



Identifying *def-use pairs* for variable `B`

Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>



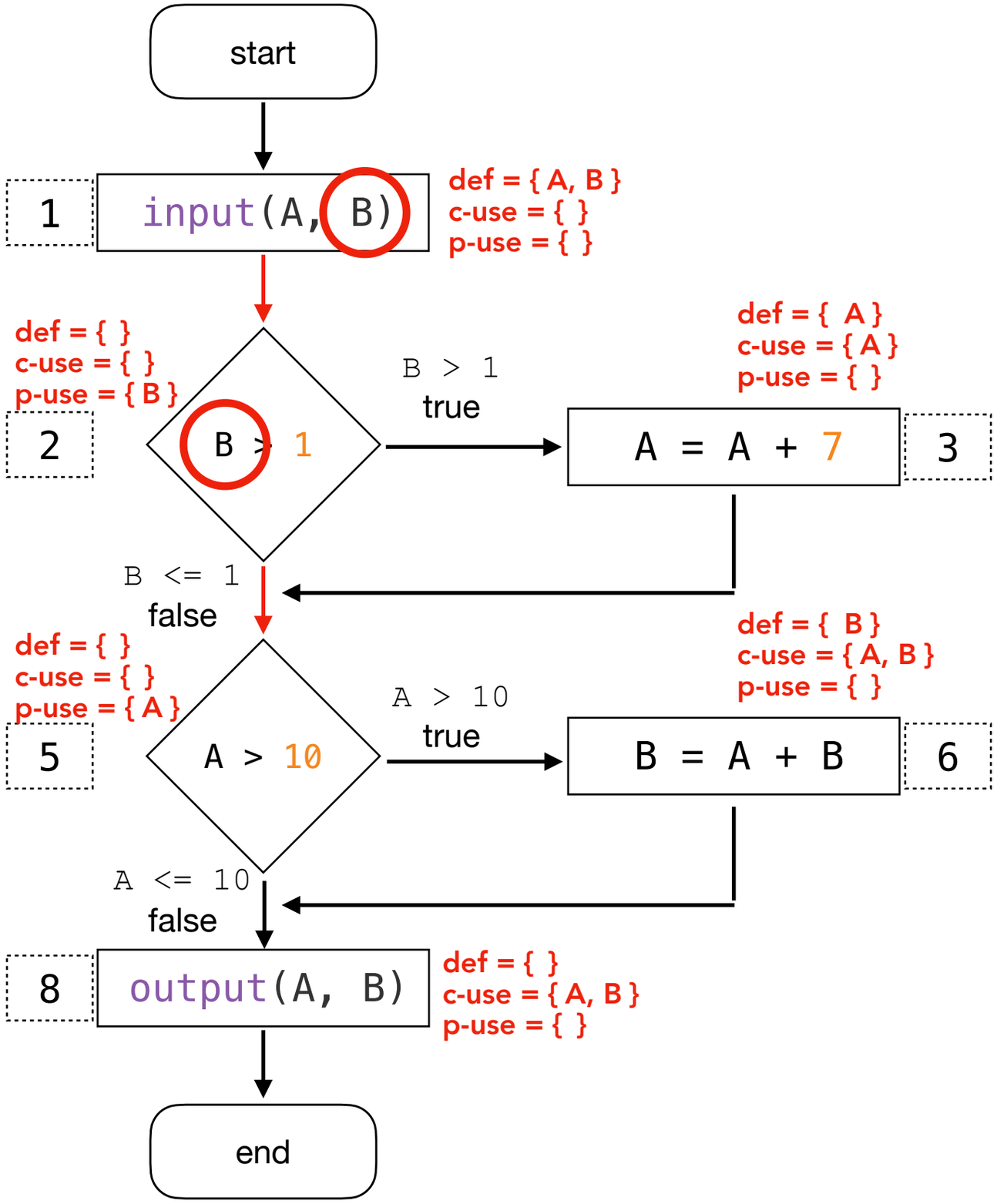
`B` defined in 1 and p-used in 2.

When it is a p-use of v, u is an outgoing edge of the predicate statement, i.e., True.

Identifying *def-use pairs* for variable `B`

Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>



`B` defined in 1 and p-used in 2.

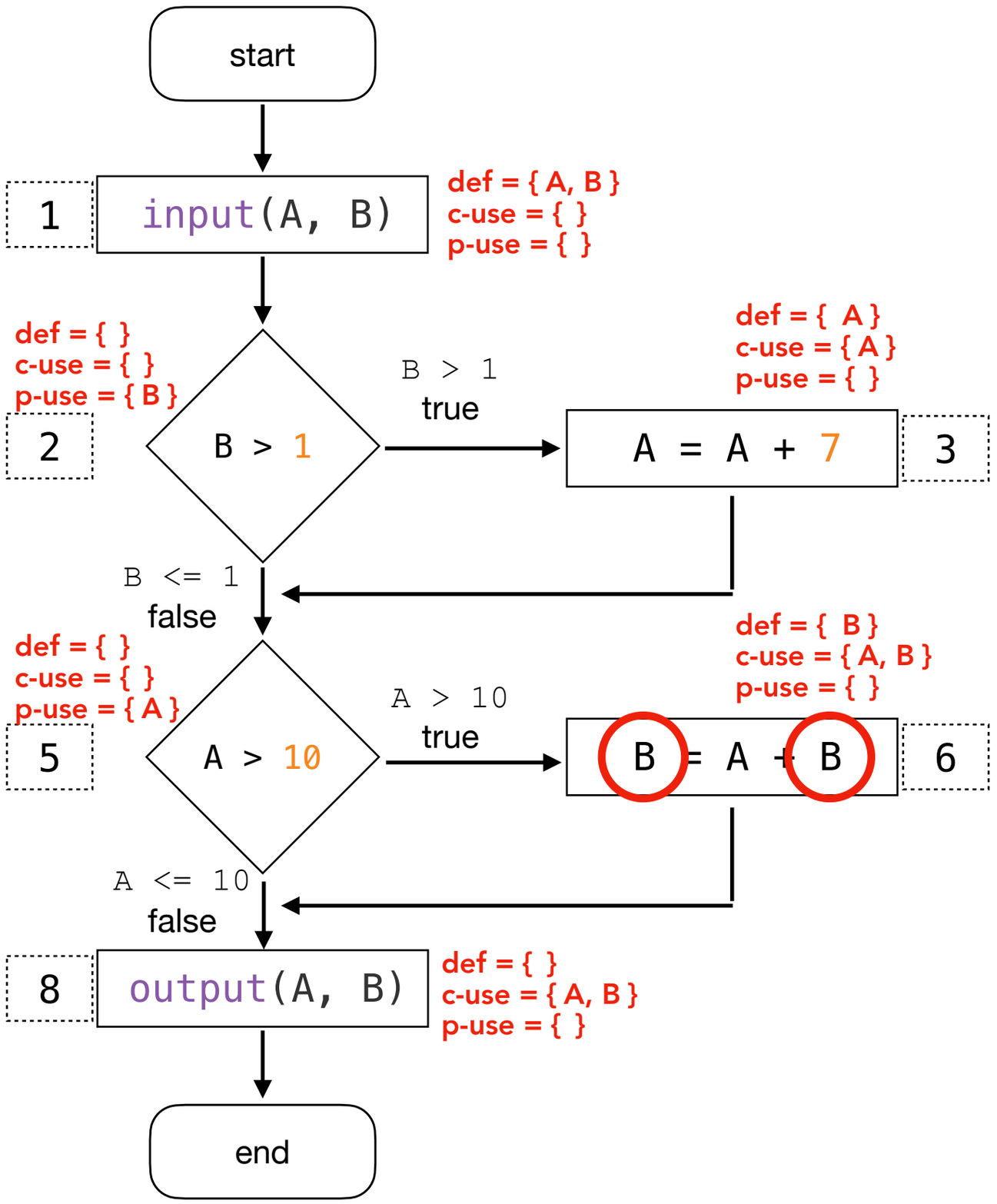
When it is a p-use of v, u is an outgoing edge of the predicate statement, i.e., False.

Identifying *def-use pairs* for variable `B`

Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>
7	6	6	<6,6>

`A` defined in 6 and c-used in 6 through <6,6>.

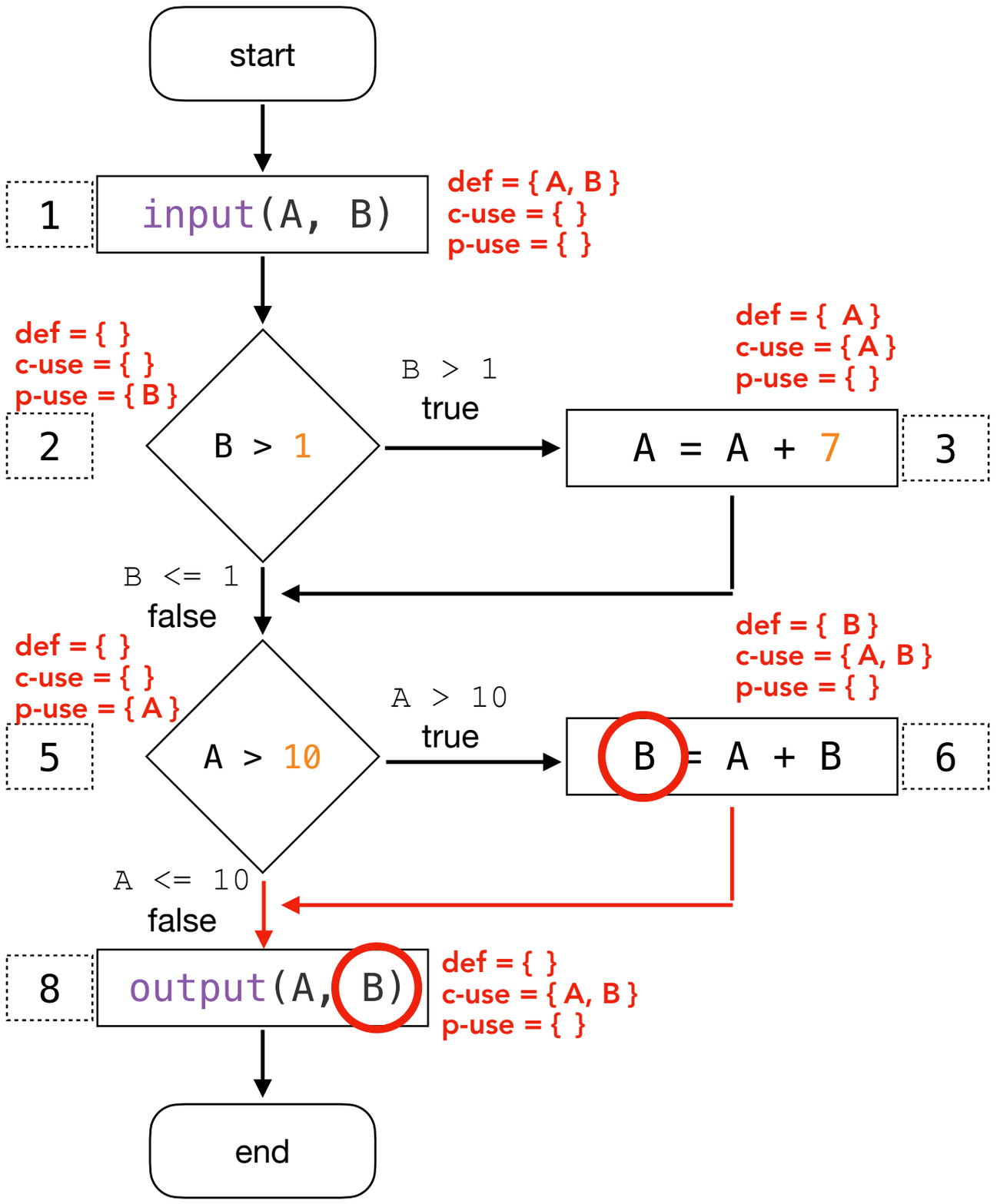


Identifying *def-use pairs* for variable `B`

Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>
7	6	6	<6,6>
8	6	8	<6,8>

`B` defined in 6 and c-used in 8 through <6,8>.



Dataflow Test Coverage Criteria I

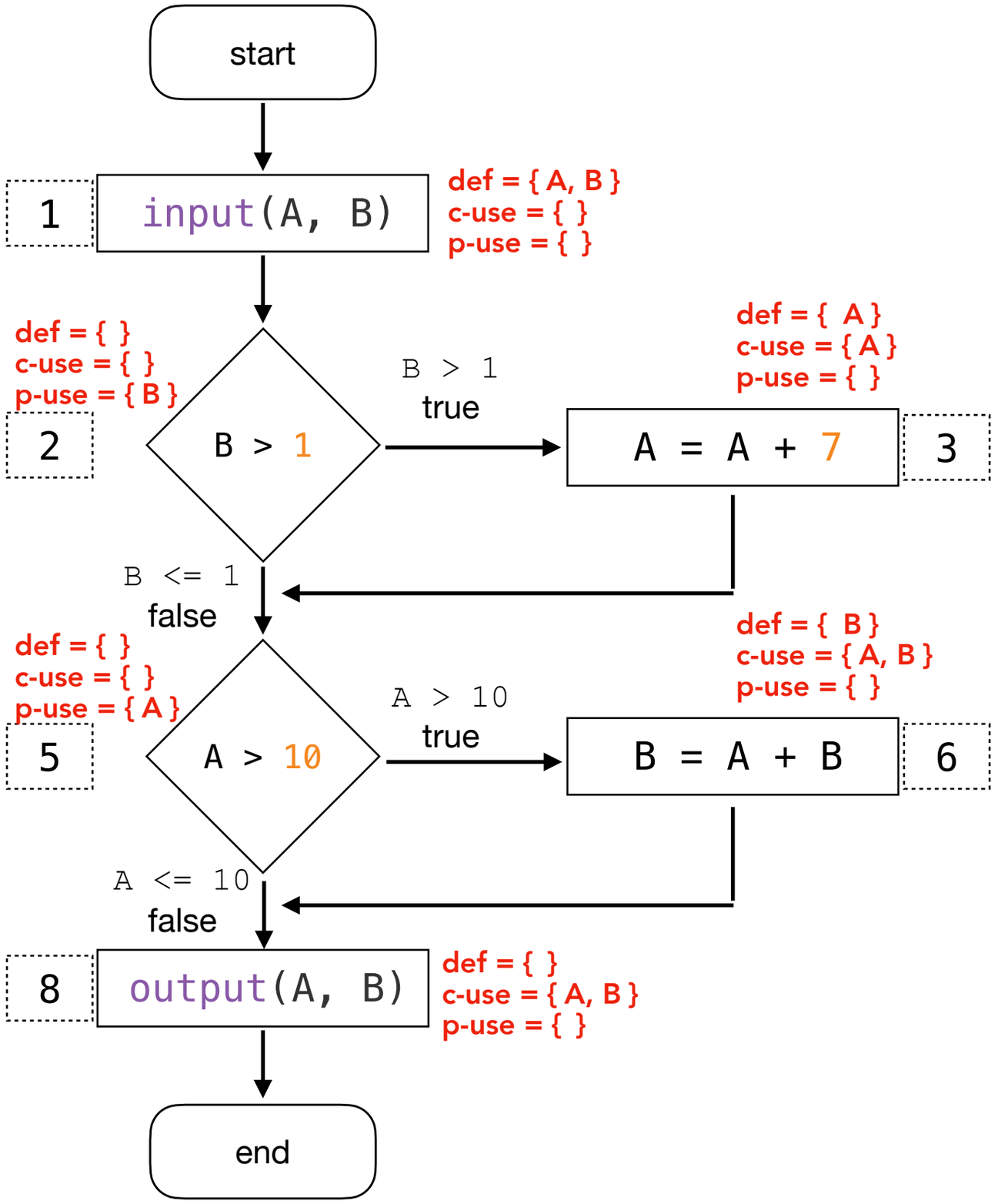
Rapps and Weyuker identified several dataflow based test adequacy criteria that map to corresponding coverage goals. These are based on test sets that exercise specific path segments.

All-defs: for every program variable **v**, at least one def-clear path from every definition of **v** to at least one c-use or one p-use of **v** must be covered. In other words, test cases include a def-clear path from every definition of **v** to some corresponding use (either c-use or p-use).

In our running example, are all definitions for each variable associated with at least one def-clear path?

Variable A

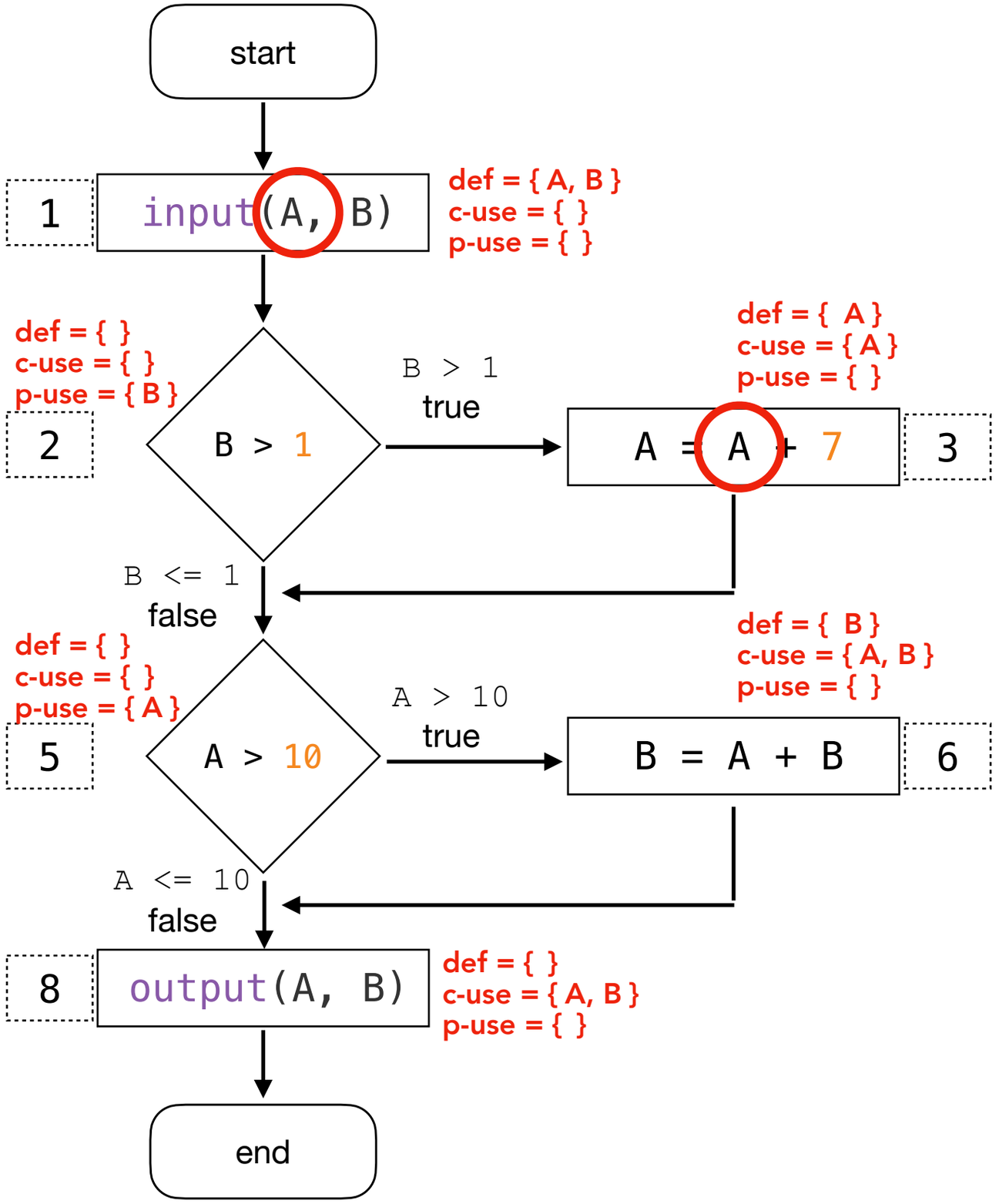
pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>
12	3	(5,F)	<3,5,8>



Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>
12	3	(5,F)	<3,5,8>

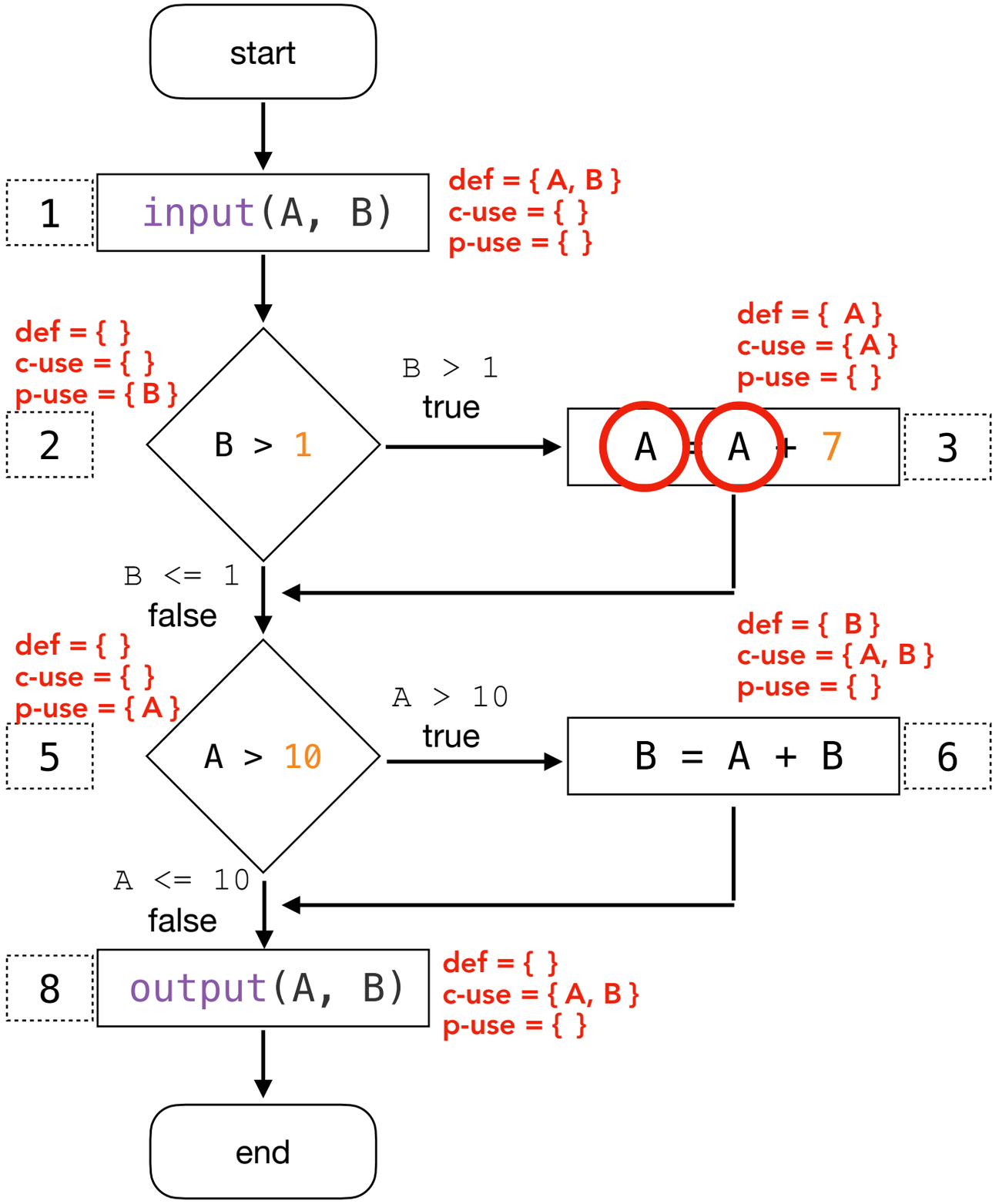
`A` defined in 1 and c-used in 3 through <1,2,3>.



Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>
12	3	(5,F)	<3,5,8>

`A` defined in 3 and c-used in 3 through <3,3>.

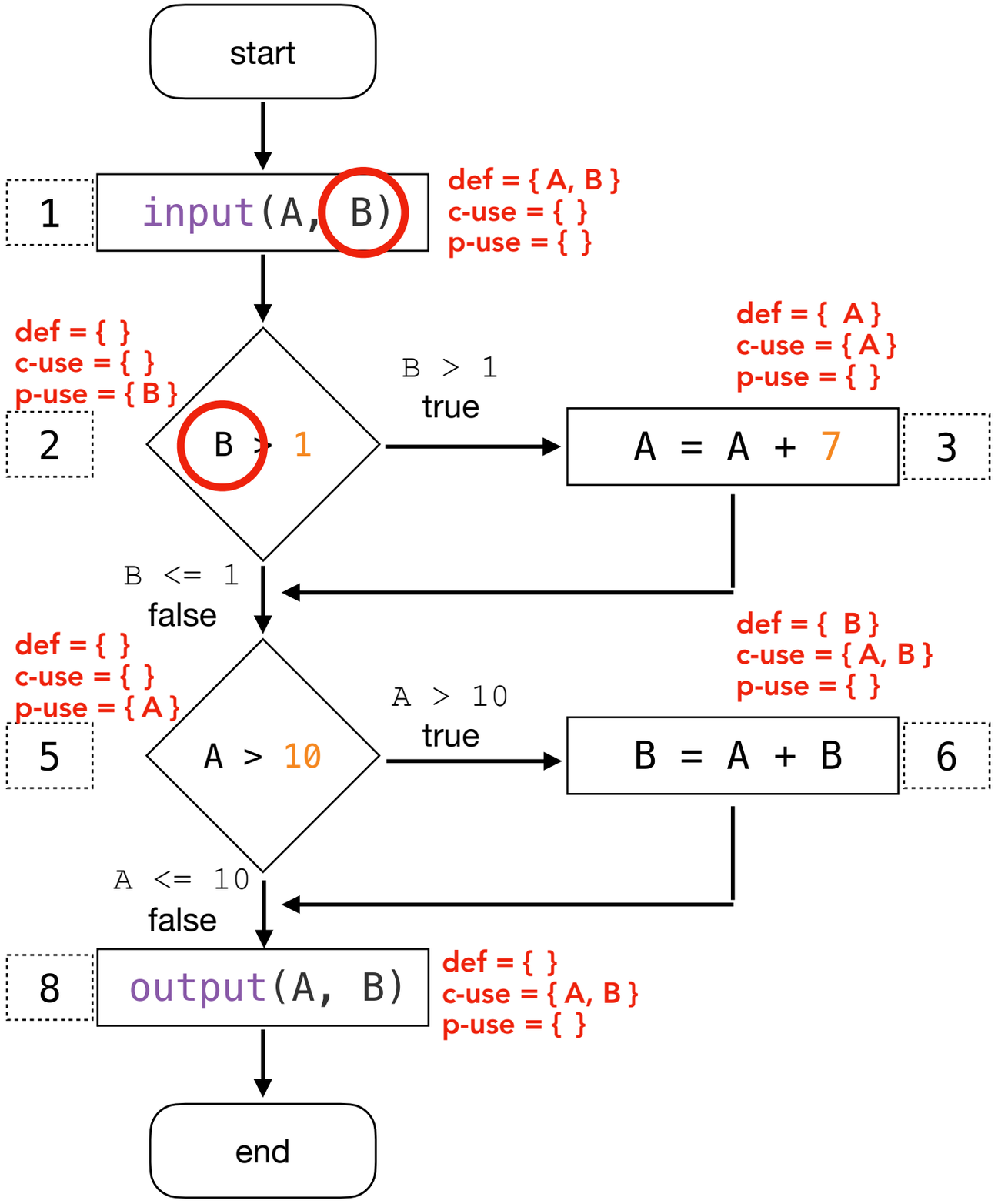


Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>
7	6	6	<6,6>
8	6	8	<6,8>

`B` defined in 1 and p-used in 2.

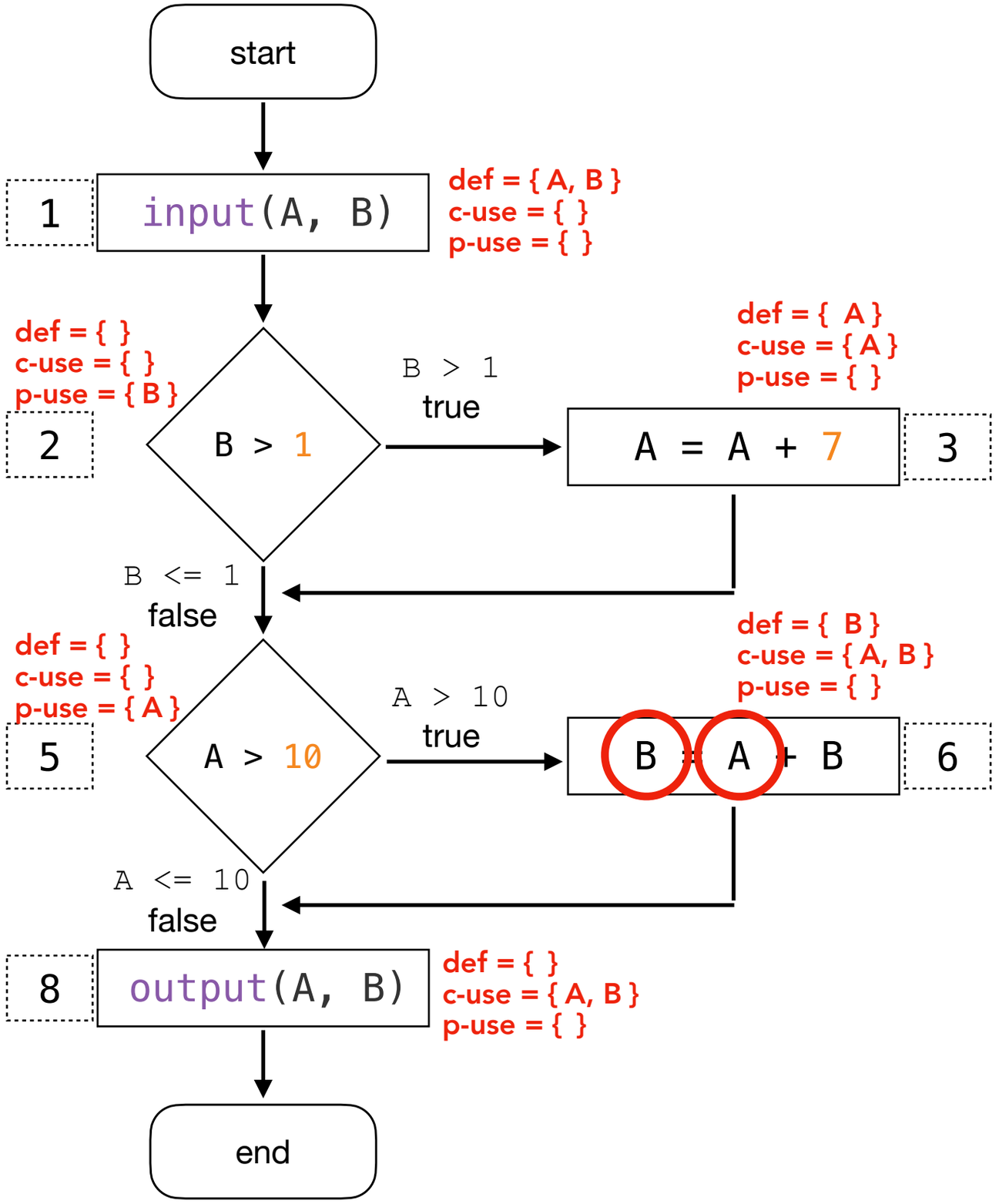
When it is a p-use of v, u is an outgoing edge of the predicate statement, i.e., True.



Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>
7	6	6	<6,6>
8	6	8	<6,8>

`B` defined in 6 and c-used in 6 through <6,6>.



Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>
12	3	(5,F)	<3,5,8>

Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>
7	6	6	<6,6>
8	6	8	<6,8>

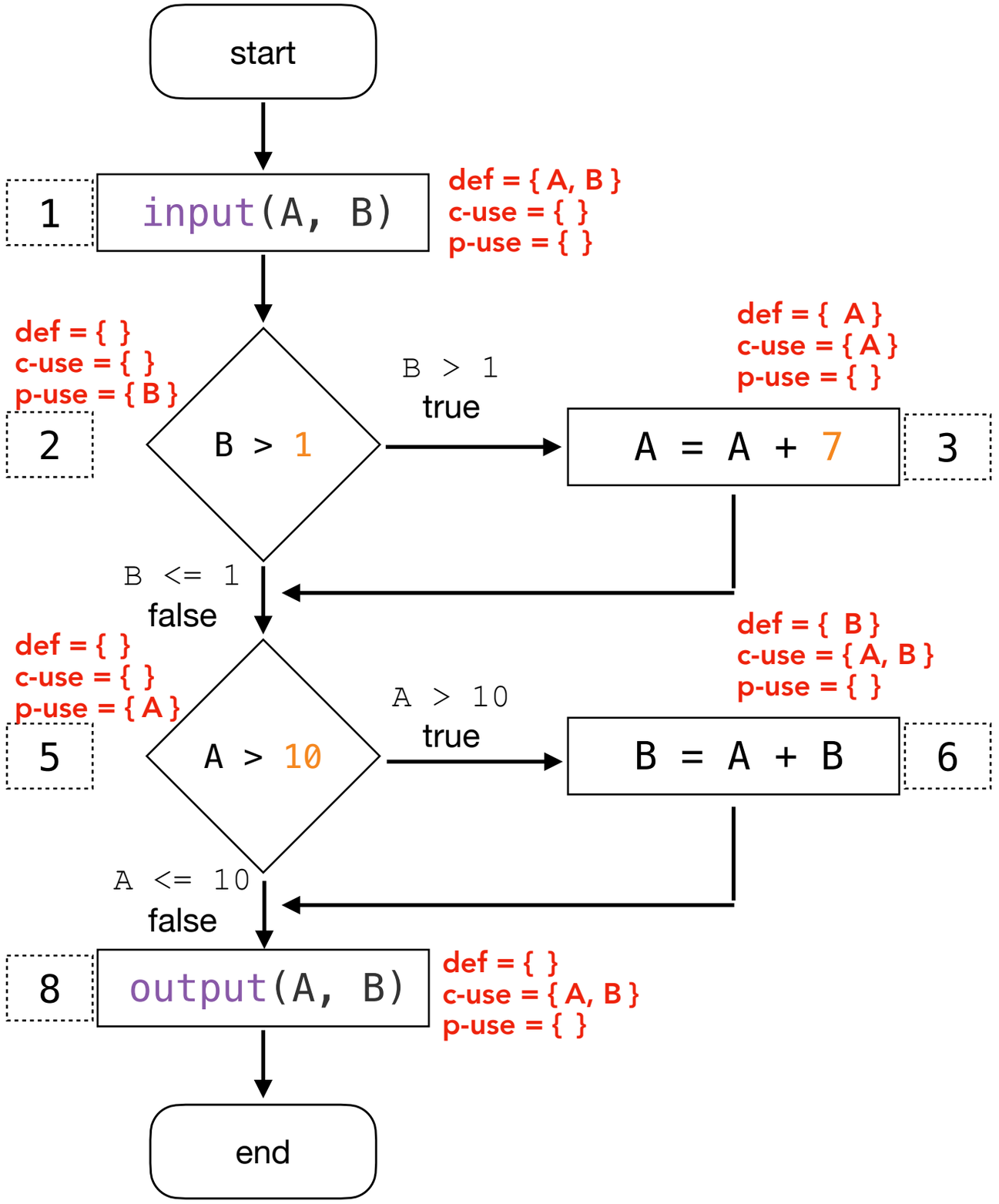
All-defs coverage is achieved as there is at least one def-clear path from every definition of **A** or **B** to at least one c-use or p-use of **A** or **B**.

Dataflow Test Coverage Criteria II

All-c-uses: for every program variable **v**, at least one def-clear path from every definition of **v** to every c-use of **v** must be covered.

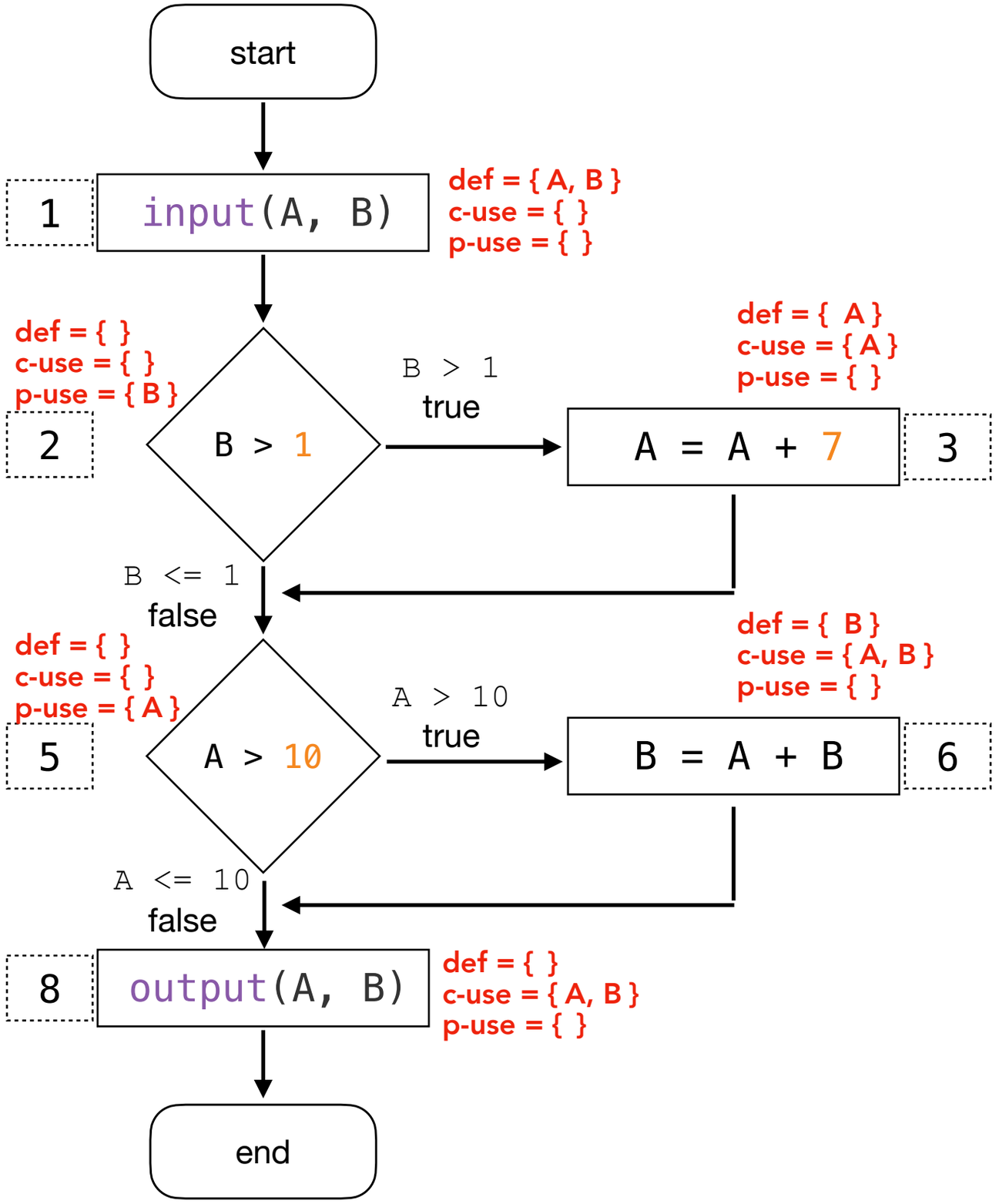
Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>
12	3	(5,F)	<3,5,8>



Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>
7	6	6	<6,6>
8	6	8	<6,8>

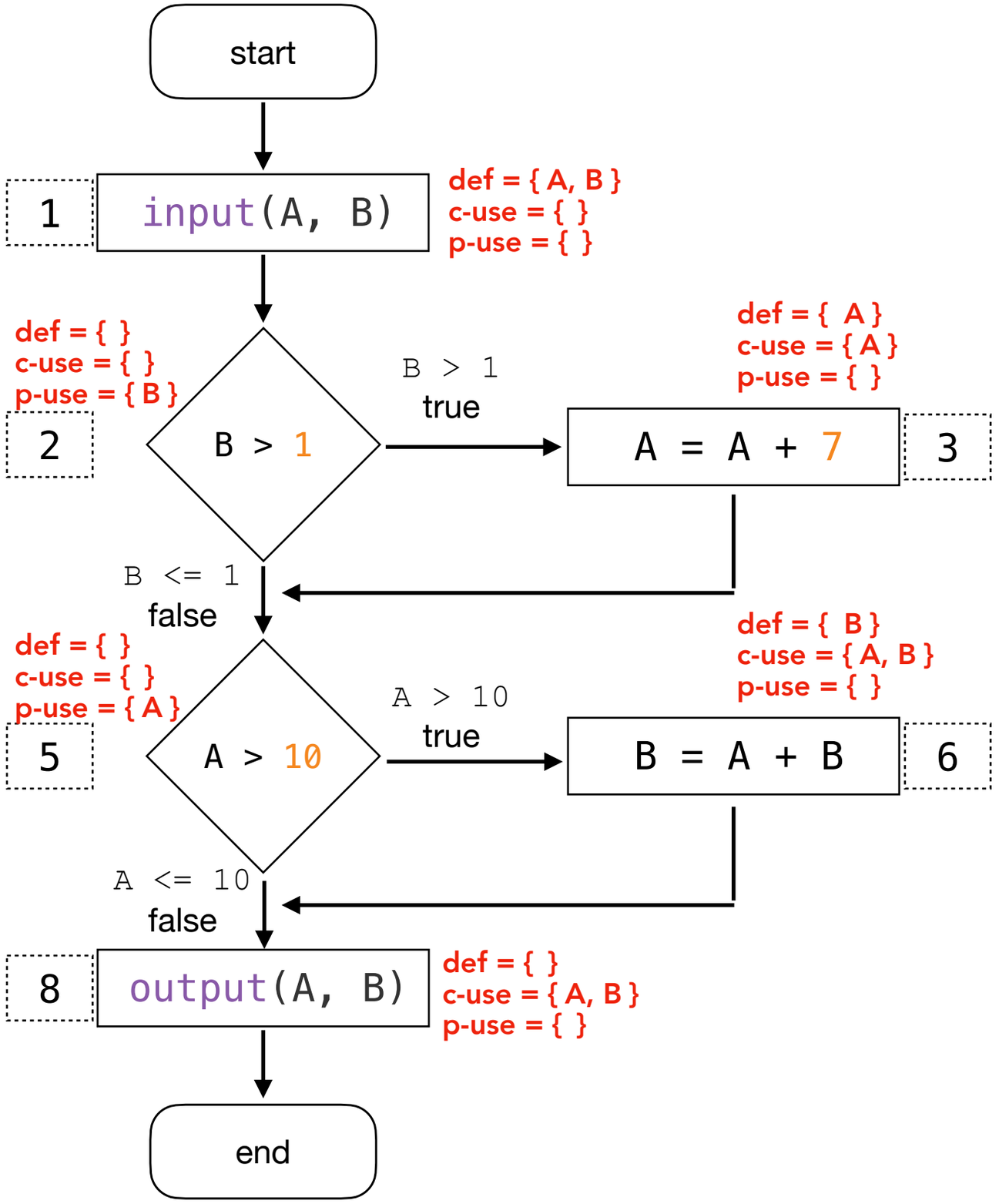


Dataflow Test Coverage Criteria III

All-p-uses: for every program variable **v**, at least one def-clear path from every definition of **v** to every p-use of **v** must be covered.

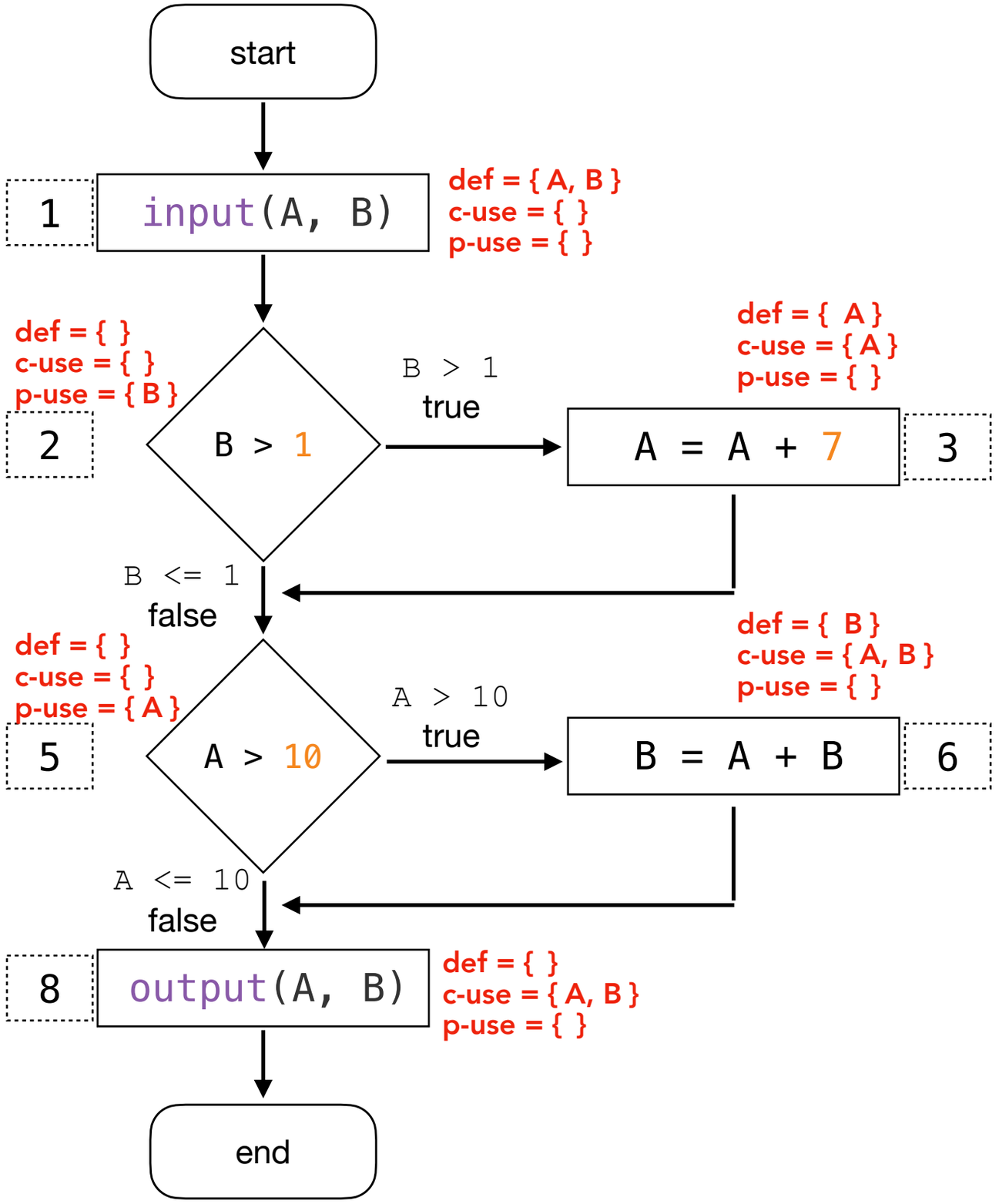
Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>
12	3	(5,F)	<3,5,8>



Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>
7	6	6	<6,6>
8	6	8	<6,8>



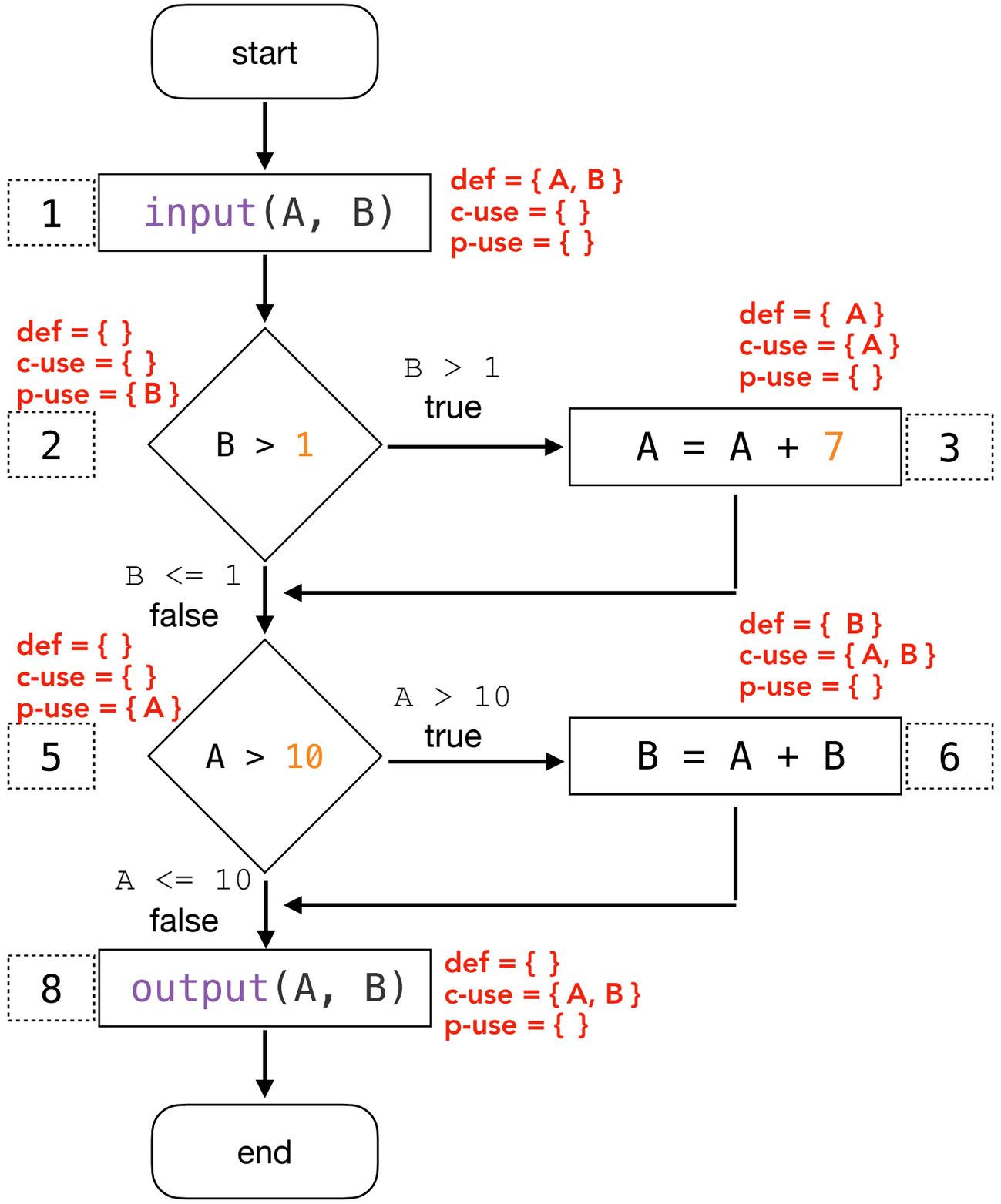
Dataflow Test Coverage Criteria IV

All-c-uses/some-p-uses: for every program variable **v**, at least one def-clear path from every definition of **v** to every c-use of **v** must be covered. If no c-use of **v** is available, at least one def-clear path to a p-use of **v** must be covered.

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>
12	3	(5,F)	<3,5,8>

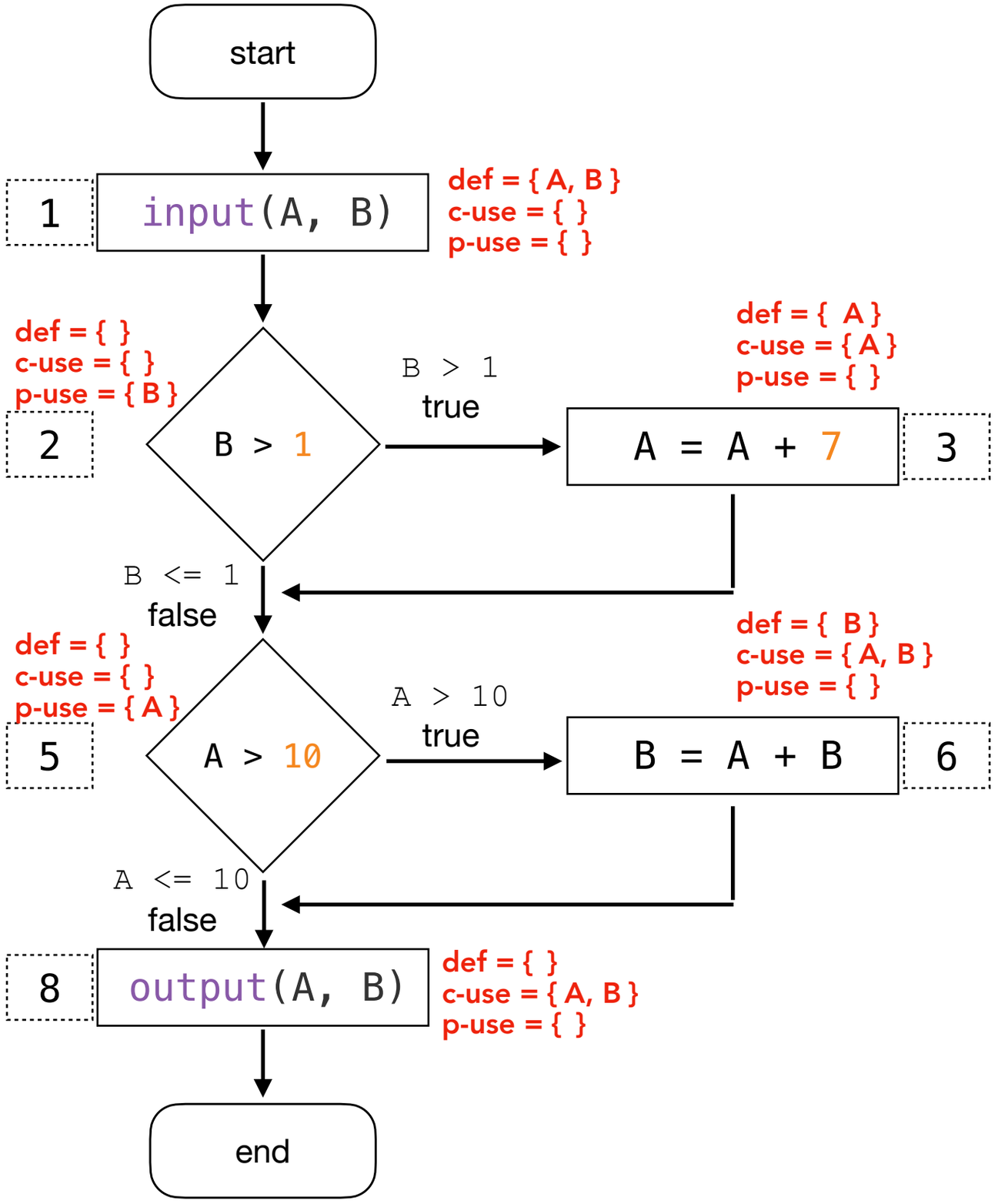
As there is at least one c-use for each definition of **A**, no p-use is used.



Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>
7	6	6	<6,6>
8	6	8	<6,8>

As there is at least one c-use for each definition of **B**, no p-use is used. If pair ids 1, 2, 3, or 4 did not exist, all-c-uses/some-p-uses would include pair id, e.g., 5.



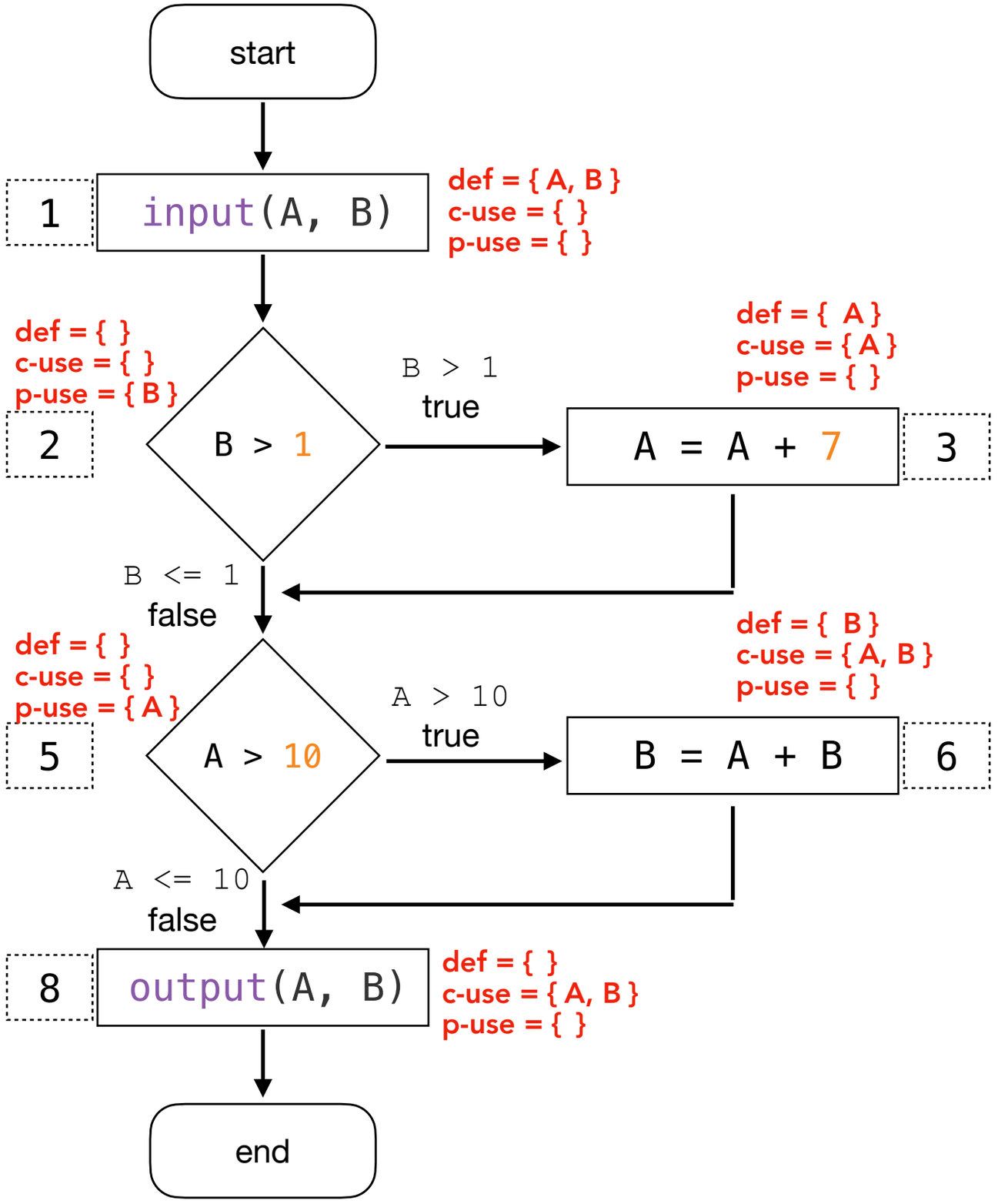
Dataflow Test Coverage Criteria V

All-p-uses/some c-uses: for every program variable **v**, at least one def-clear path from every definition of **v** to every p-use of **v** must be covered. If no p-use of **v** is available, at least one def-clear path to a c-use of **v** must be covered.

Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>
12	3	(5,F)	<3,5,8>

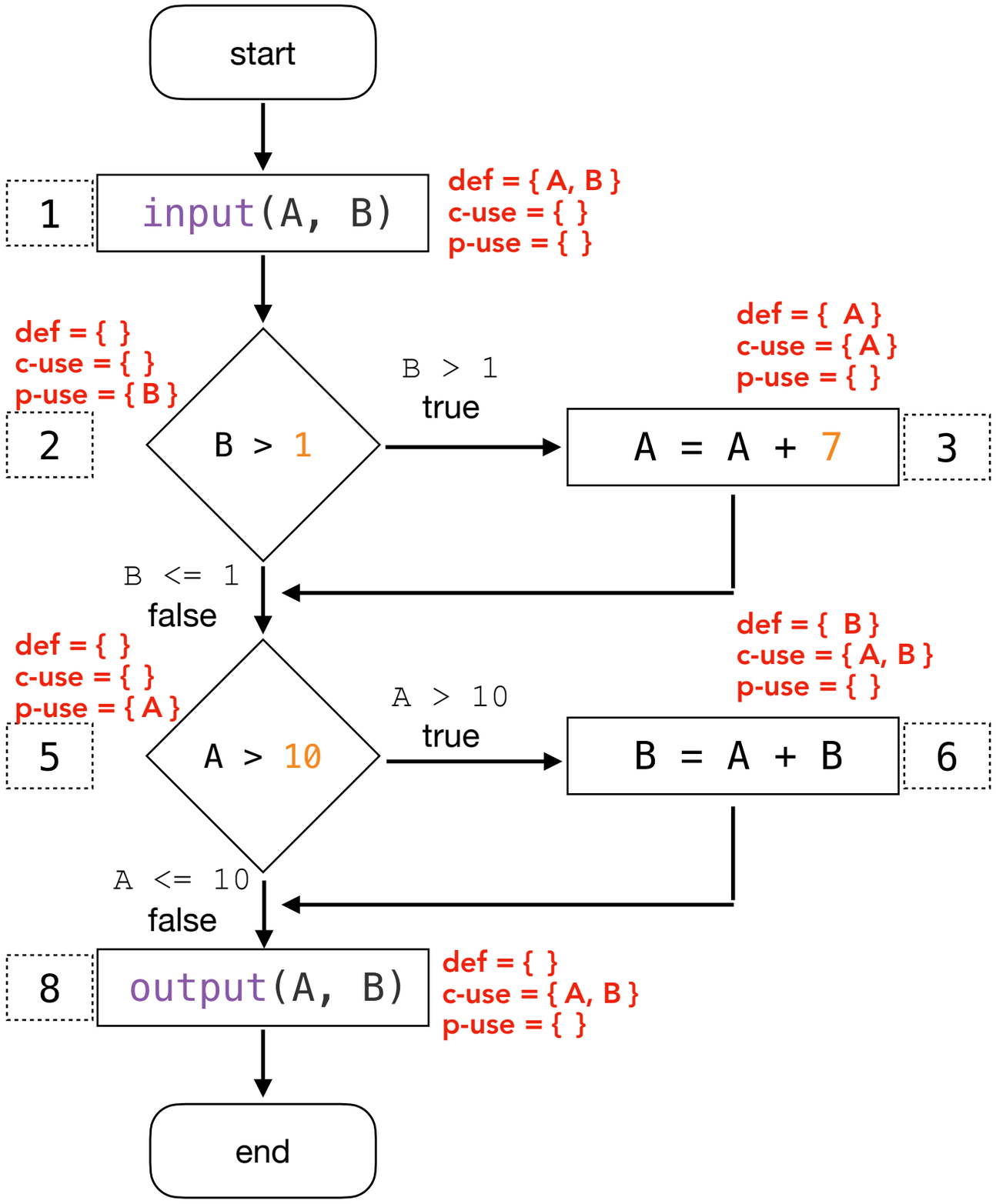
As there is at least one p-use for each definition of **A**, no c-use is used.



Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>
7	6	6	<6,6>
8	6	8	<6,8>

Note that from node 6 (definition of **B**) there is no other p-use node that could be reached, thus a c-use is considered by All-p-uses/some c-uses.

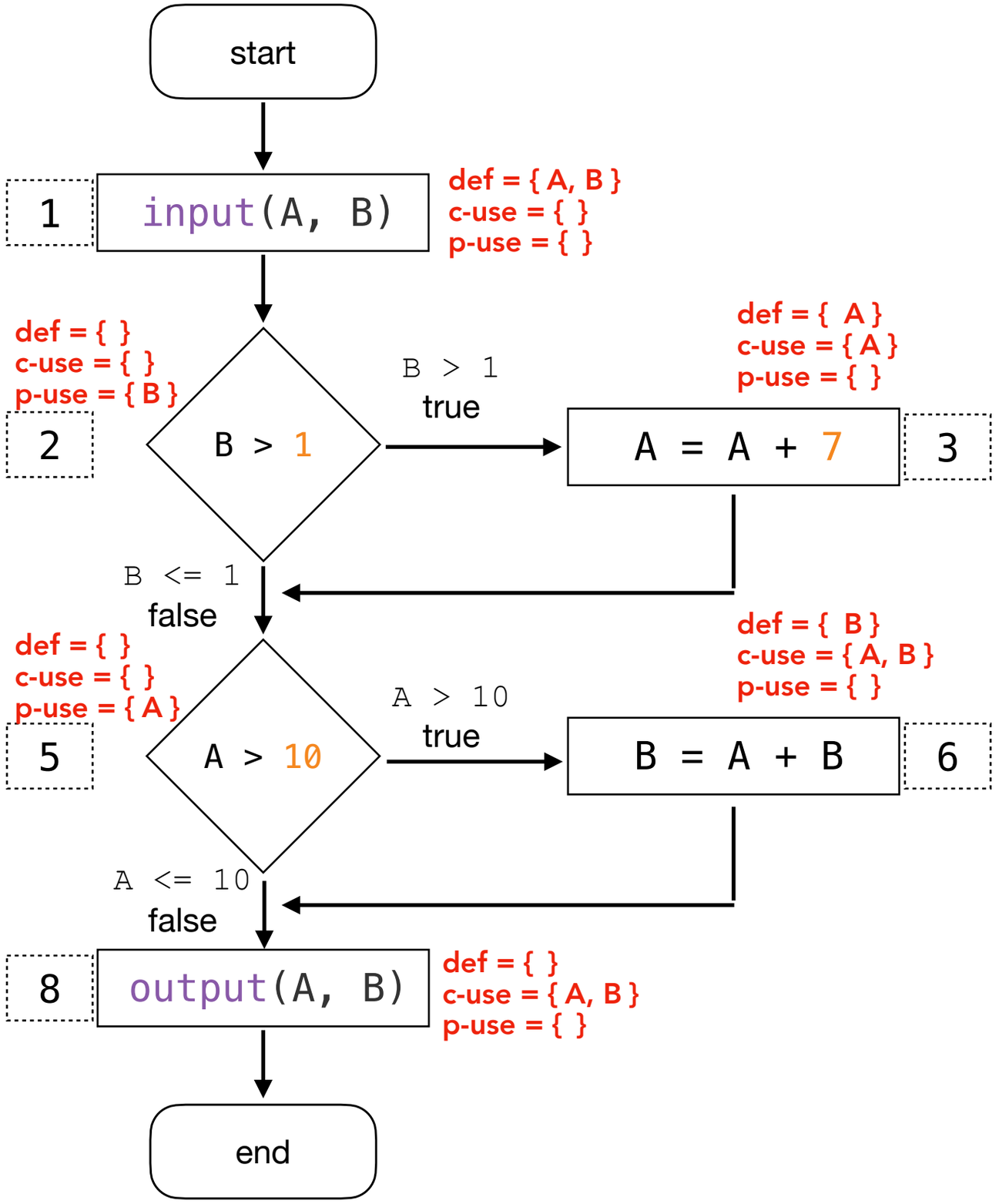


Dataflow Test Coverage Criteria VI

All-uses: for every program variable **v**, at least one def-clear path from every definition of **v** to every c-use and every p-use (including all outgoing edges of the predicate statement) of **v** must be covered. Requires that all def-use pairs are covered.

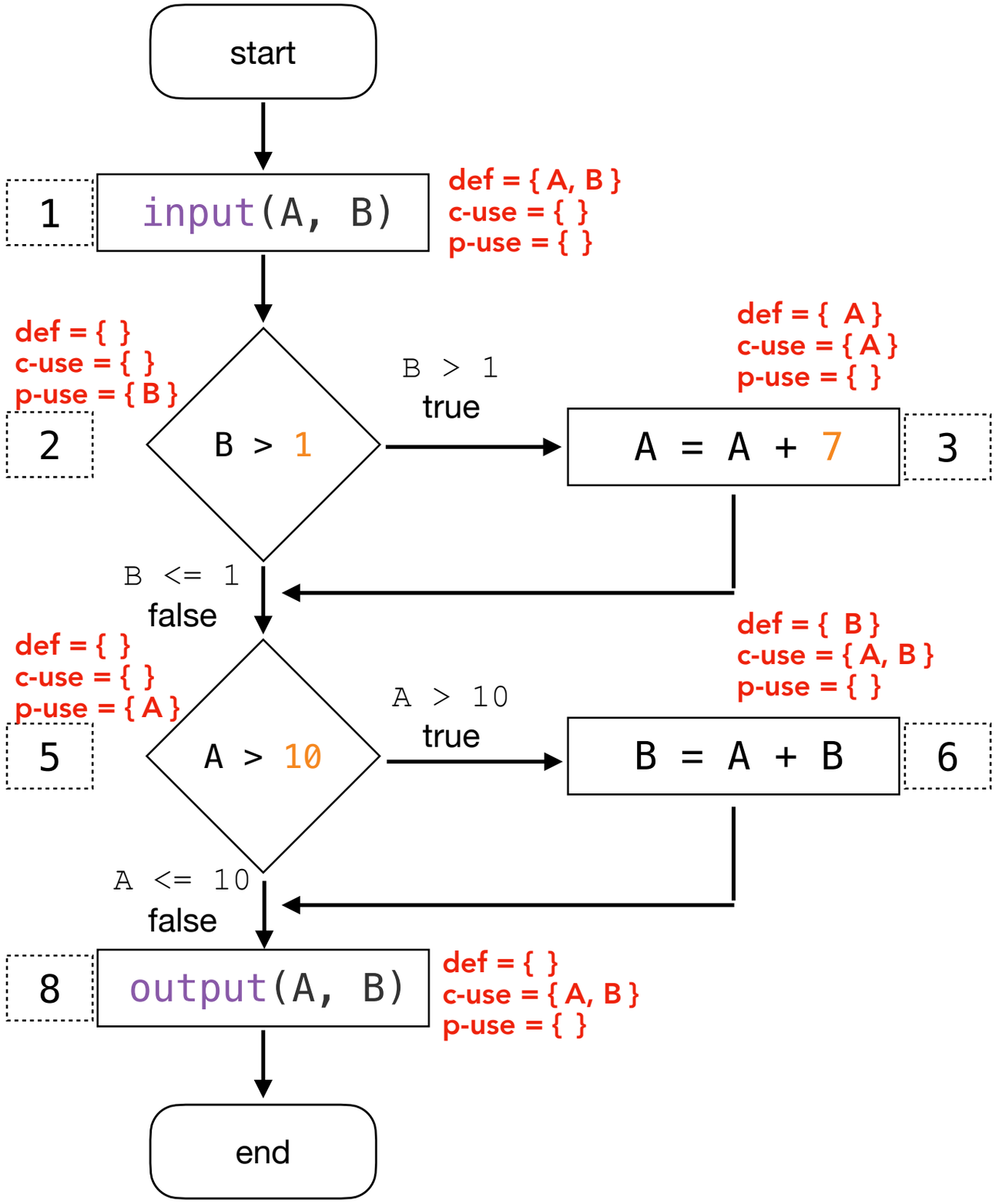
Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>
12	3	(5,F)	<3,5,8>



Variable B

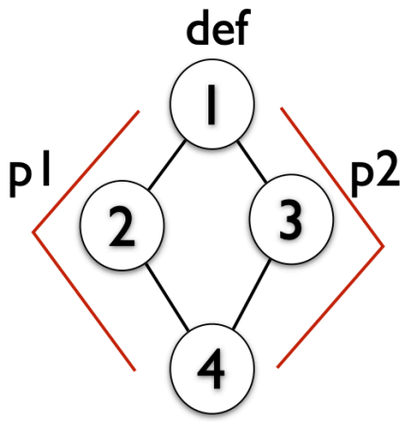
pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>
7	6	6	<6,6>
8	6	8	<6,8>



Dataflow Test Coverage Criteria VII

All-def-use paths: for every program variable **v**, every def-use path from every definition of **v** to every c-use and every p-use of **v** must be covered. Therefore, if there are multiple paths between a given definition and a use, they must all be included. It is the **strongest** data-flow testing criteria since it is a superset of all other data flow testing strategies. **All-def-use paths** is similar to **All-uses**, but instead of at least one path from each def to each use of that def, includes all paths from each def to each use of that def.

A path is a def-use path with respect to a variable **v**, if **v** is defined at node **n1** and either:
- there is a c-use of **v** at node **nk** and is a def-clear simple path (i.e., all edges within the path are distinct, i.e., different), or
- there is a p-use of **v** at edge and is a def-clear loop-free path (i.e., all nodes within the path are distinct, i.e., different).



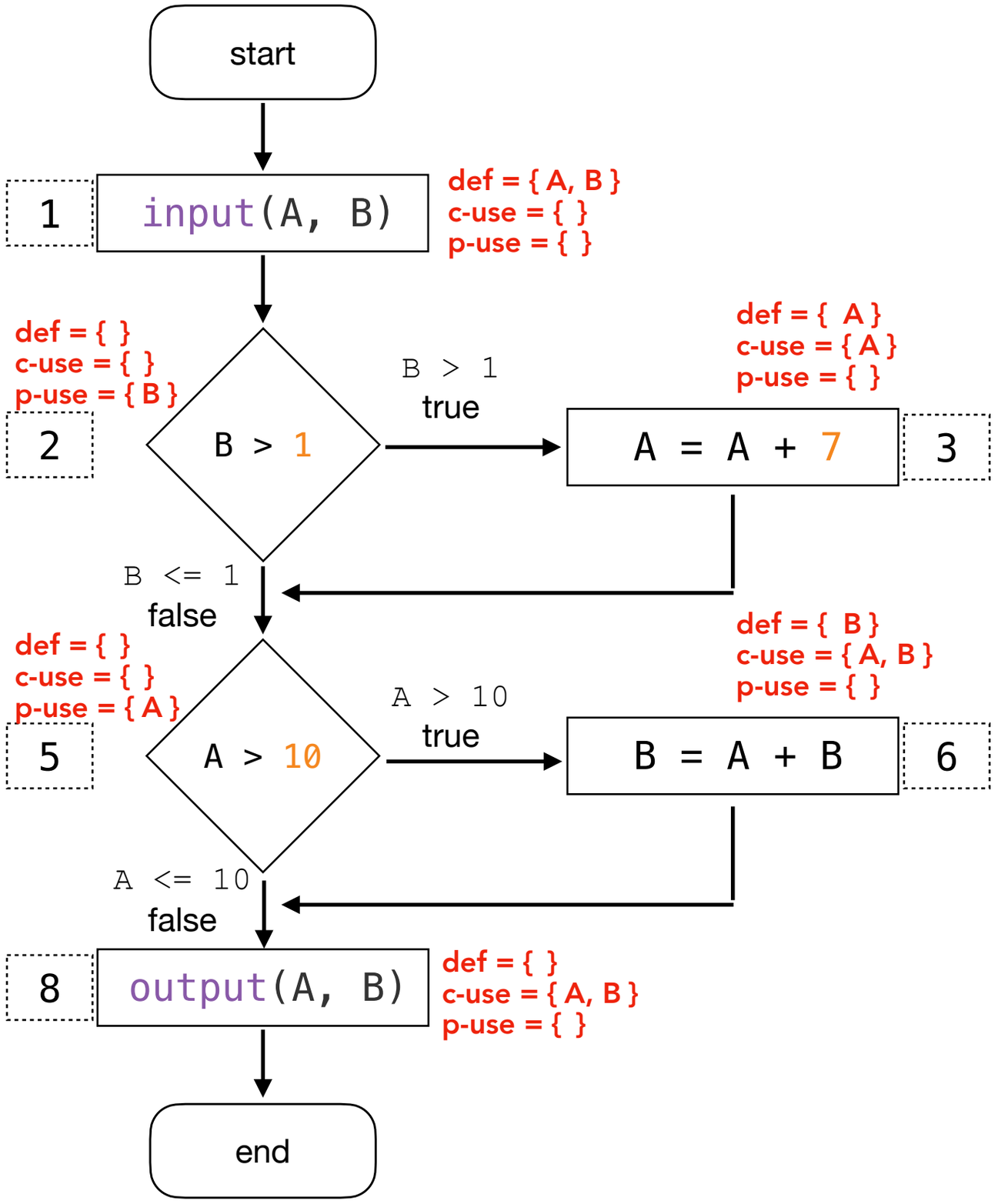
c-use
node 1 is the only def node, and 4 is the only use node for **v**

p1 satisfies all-defs and all-uses, but not all-du-paths

p1 and p2 together satisfy all-du-paths

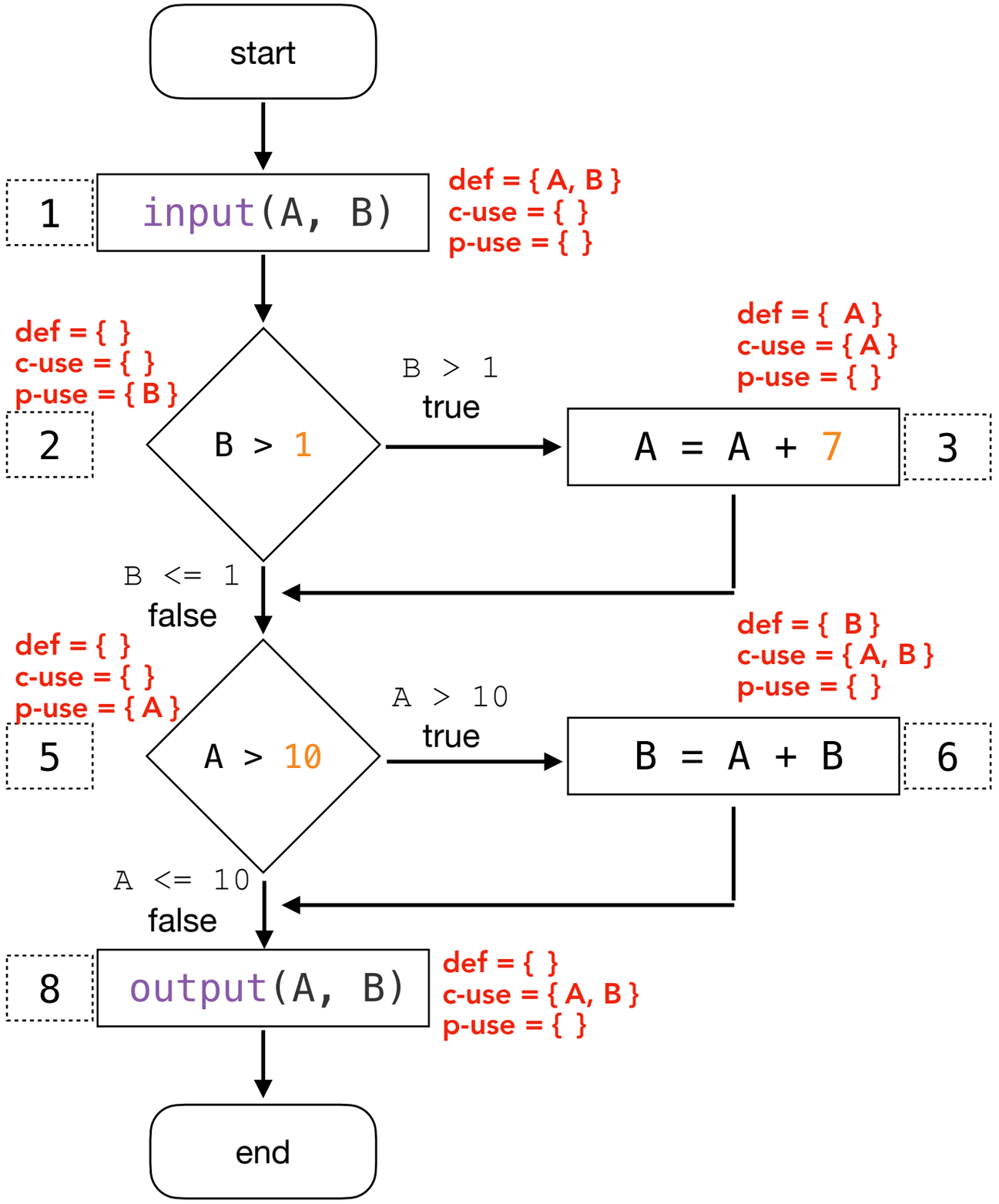
Variable A

pair id	def	use	*path*
1	1	3	<1,2,3>
2	1	6	<1,2,5,6>
3	1	8	<1,2,5,6,8>
4	1	8	<1,2,5,8>
5	1	(5,T)	<1,2,5,6>
6	1	(5,F)	<1,2,5,8>
7	3	3	<3,3>
8	3	6	<3,5,6>
9	3	8	<3,5,6,8>
10	3	8	<3,5,8>
11	3	(5,T)	<3,5,6>
12	3	(5,F)	<3,5,8>

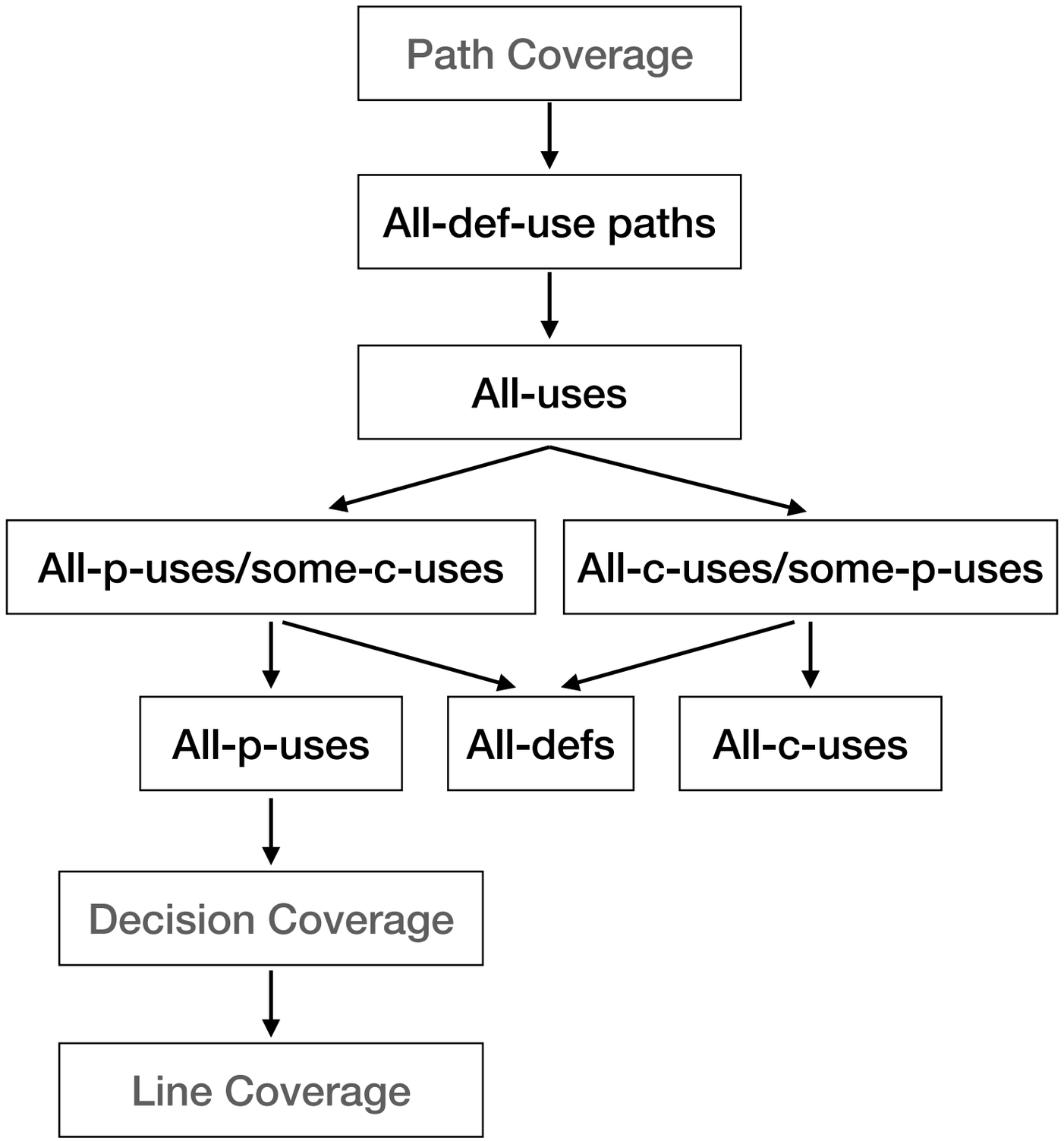


Variable B

pair id	def	use	*path*
1	1	6	<1,2,3,5,6>
2	1	6	<1,2,5,6>
3	1	8	<1,2,3,5,8>
4	1	8	<1,2,5,8>
5	1	(2,T)	<1,2,3>
6	1	(2,F)	<1,2,5>
7	6	6	<6,6>
8	6	8	<6,8>



Criteria subsumption



Dataflow

- As with most white box testing methods, the data flow approach is most effective at the unit level of testing. When code becomes more complex and there are more variables to consider it becomes more time consuming for the tester to analyze data flow roles, identify paths, and design the tests. Other problems with data flow oriented testing occur in the handling of dynamically bound variables such as pointers.

- Finally, there are no commercially available tools that provide strong support for data flow testing, such as those that support control-flow based testing. In the latter case, tools that determine the degree of coverage, and which portions of the code are yet uncovered, are of particular importance. An example of a prototype tool for data flow testing has been described in here https://link.springer.com/chapter/10.1007/978-3-642-28038-2_17 and here https://ieeexplore.ieee.org/document/6698890.

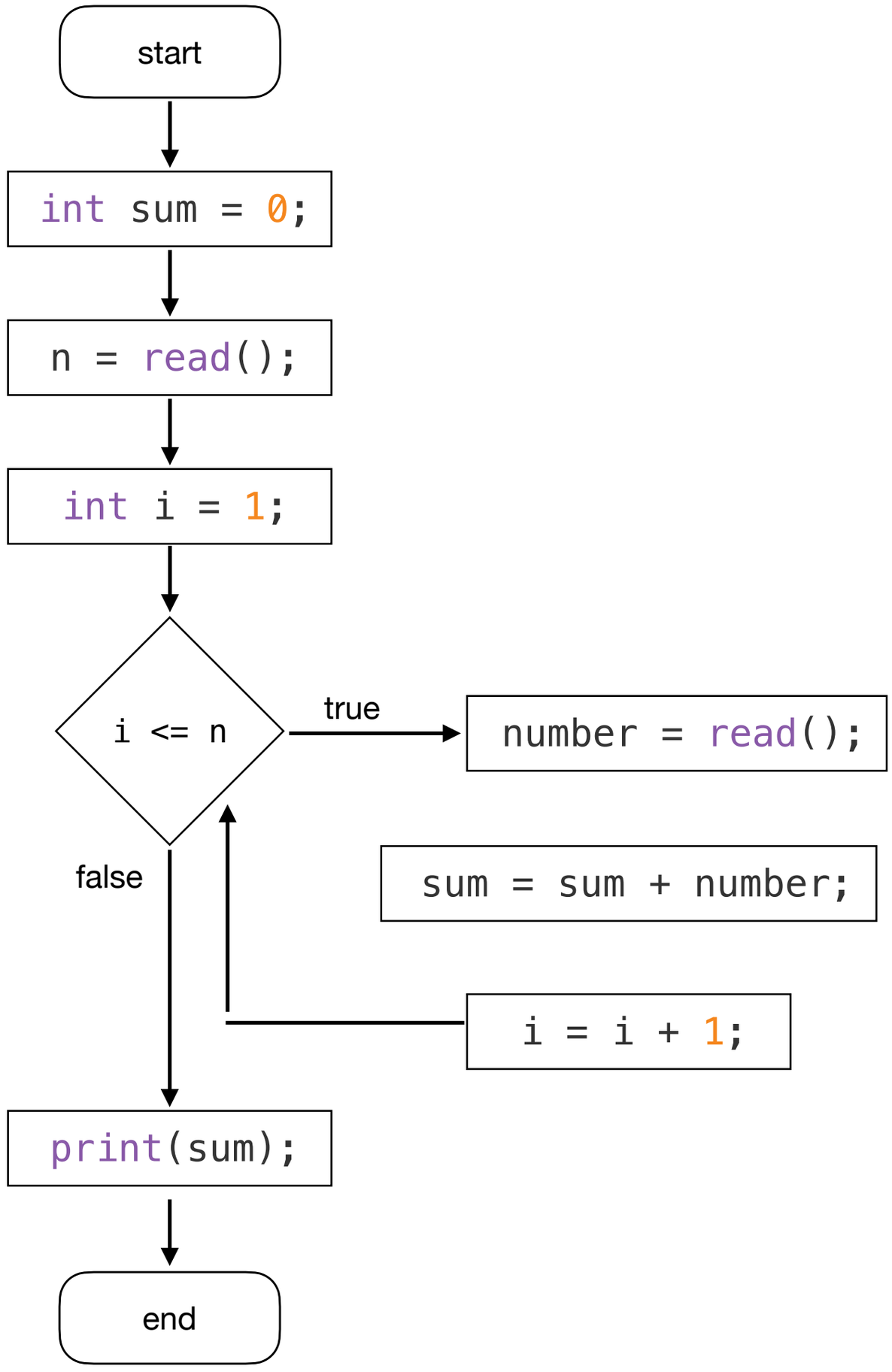
References

- Ting Su, Ke Wu, Ke Wu , Weikai Miao, Weikai Miao , Geguang Pu, Geguang Pu , Jifeng He, Jifeng He , Yuting Chen, Yuting Chen , Zhendong Su, Zhendong Su. A Survey on Data-Flow Testing. ACM Computing Survey, 2017. https://dl.acm.org/doi/10.1145/3020266
- Sandra Rapps and Elaine J. Weyuker. Selecting Software Test Data Using Data Flow Information. IEEE Transactions on Software Engineering, 11(4), April 1985, pp. 367-375. http://ieeexplore.ieee.org/stamp/stamp.jsp? tp=&arnumber=1702019
- P. Frankl and E. Weyuker. An Applicable Family of Data Flow Testing Criteria. IEE Transaction on software eng., vol.14, no.10, October 1988.
- E. Weyuker. The evaluation of Program-based software test data adequacy criteria. Communication of the ACM, vol.31, no.6, June 1988.
- Janvi Badlaney Rohit Ghatol Romit Jadhwani. An Introduction to Data-Flow Testing, NCSU CSC TR-2006-22, 2006, https://people.eecs.ku.edu/~saiedian/Teaching/Fa09/814/Lectures/intro-df-testing-1.pdf.
- Software Testing: A Craftsman's Approach.2nd CRC publication, 2002.
- Gordon Fraser and José Miguel Rojas; Software Testing, 2019. ISBN 978-3-030-00262-6.
- Chapter 6, 12, and 13 of the Software Testing and Analysis: Process, Principles, and Techniques. Mauro Pezzè, Michal Young, 1st edition, Wiley, 2007.
- Chapter 5 of the Practical software testing a process-oriented approach. Ilene Burnstein, 2002.
- Chapter 4 of the Foundations of software testing: ISTQB certification. Graham, Dorothy, Erik Van Veenendaal, and Isabel Evans, Cengage Learning EMEA, 2008.
- Zhu, H., Hall, P. A., & May, J. H. (1997). Software unit test coverage and adequacy. ACM computing surveys.
- CS/CE/SE 6367 Software Testing, Validation and Verification at The University of Texas, Dallas.

Appendix

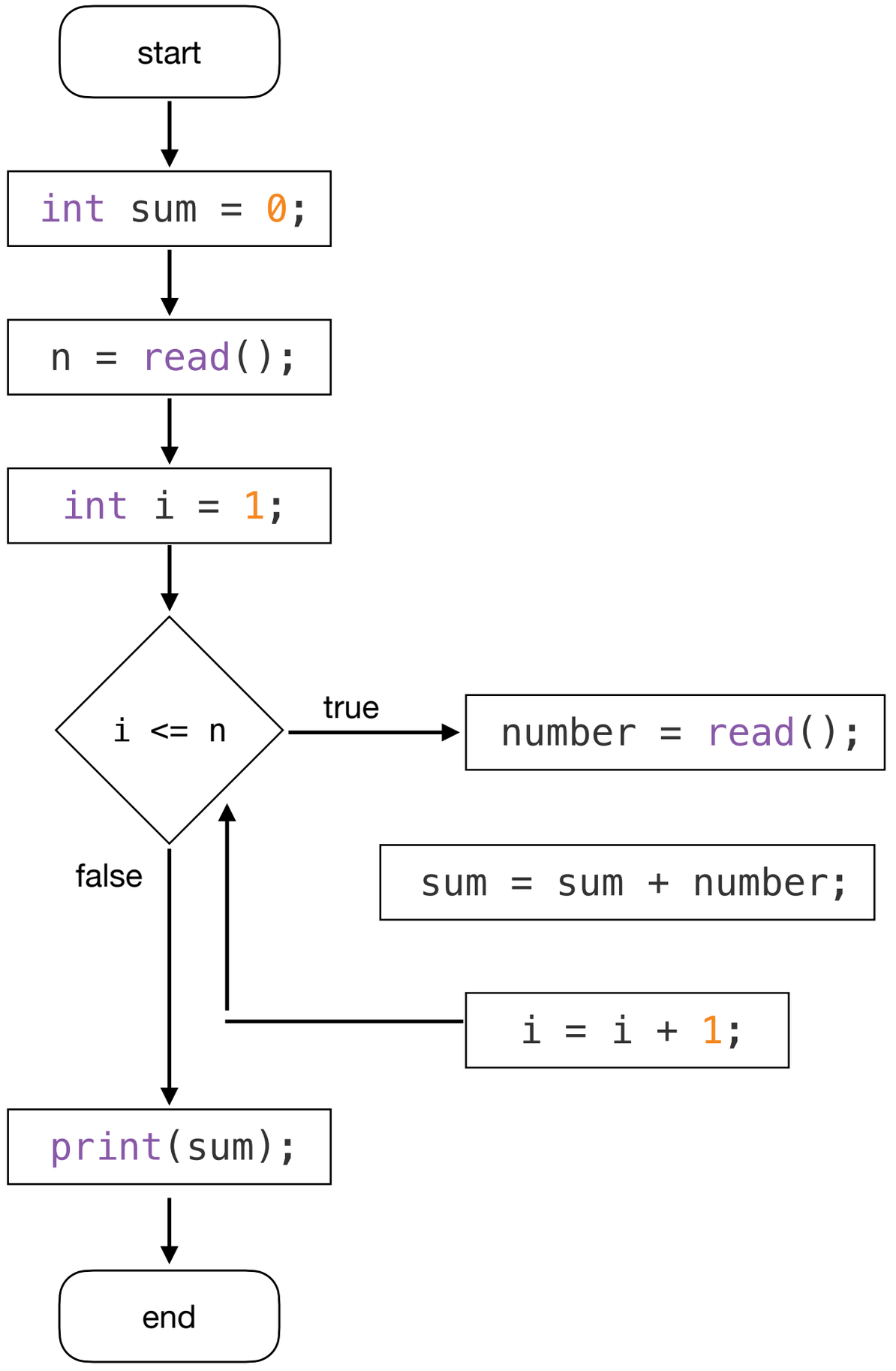
```
public int sum_of_n_numbers() {
    int sum = 0;
    n = read();
    int i = 1;
    while (i <= n) {
        number = read();
        sum = sum + number;
        i = i + 1;
    }
    print(sum);
}
```

```
   public int sum_of_n_numbers() {
1      int sum = 0;
2      n = read();
3      int i = 1;
4      while (i <= n) {
5          number = read();
6          sum = sum + number;
7          i = i + 1;
8      }
9      print(sum);
   }
```

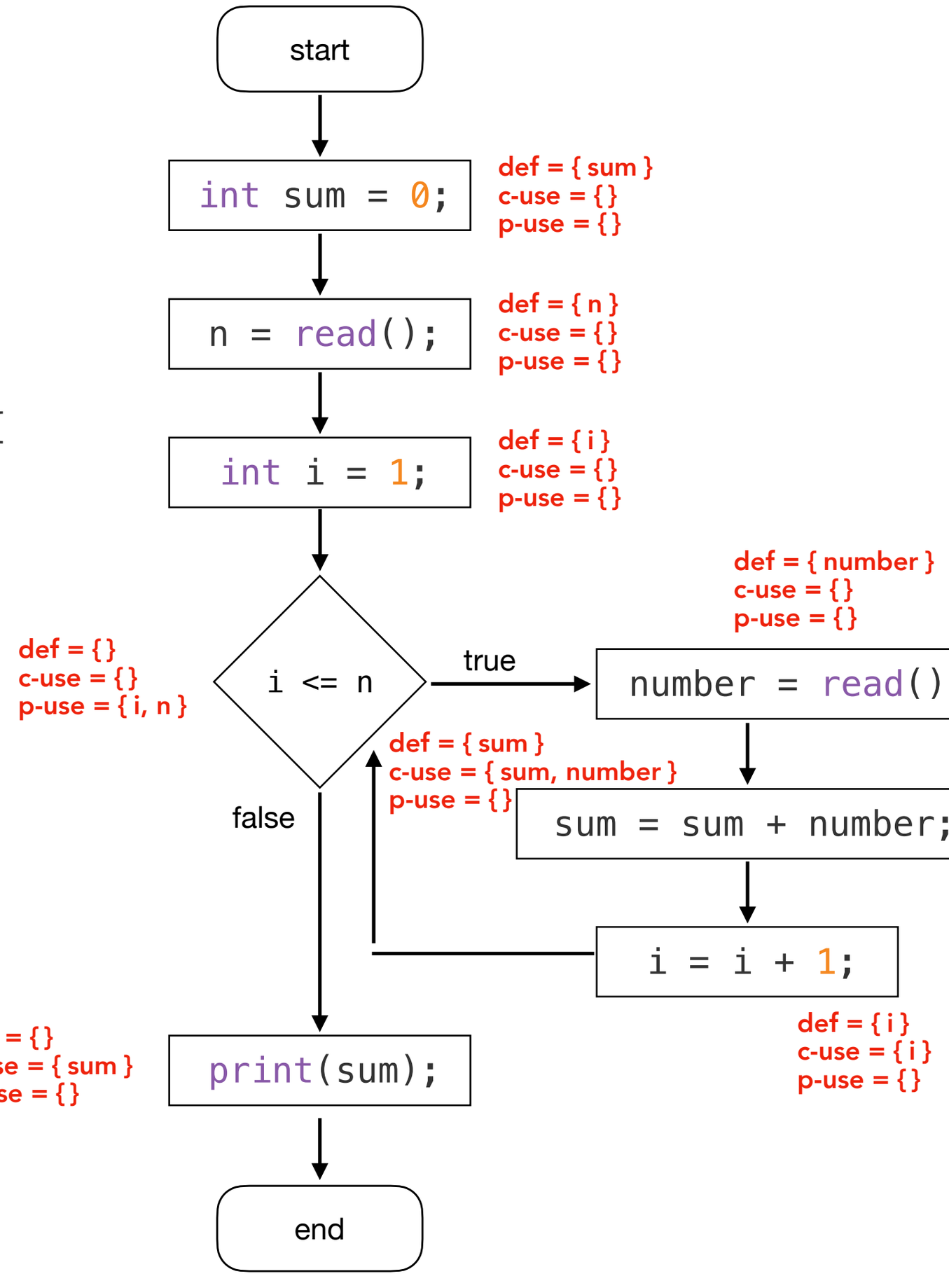


```
   public int sum_of_n_numbers() {
1      int sum = 0;
2      n = read();
3      int i = 1;
4      while (i <= n) {
5          number = read();
6          sum = sum + number;
7          i = i + 1;
8      }
9      print(sum);
   }
```

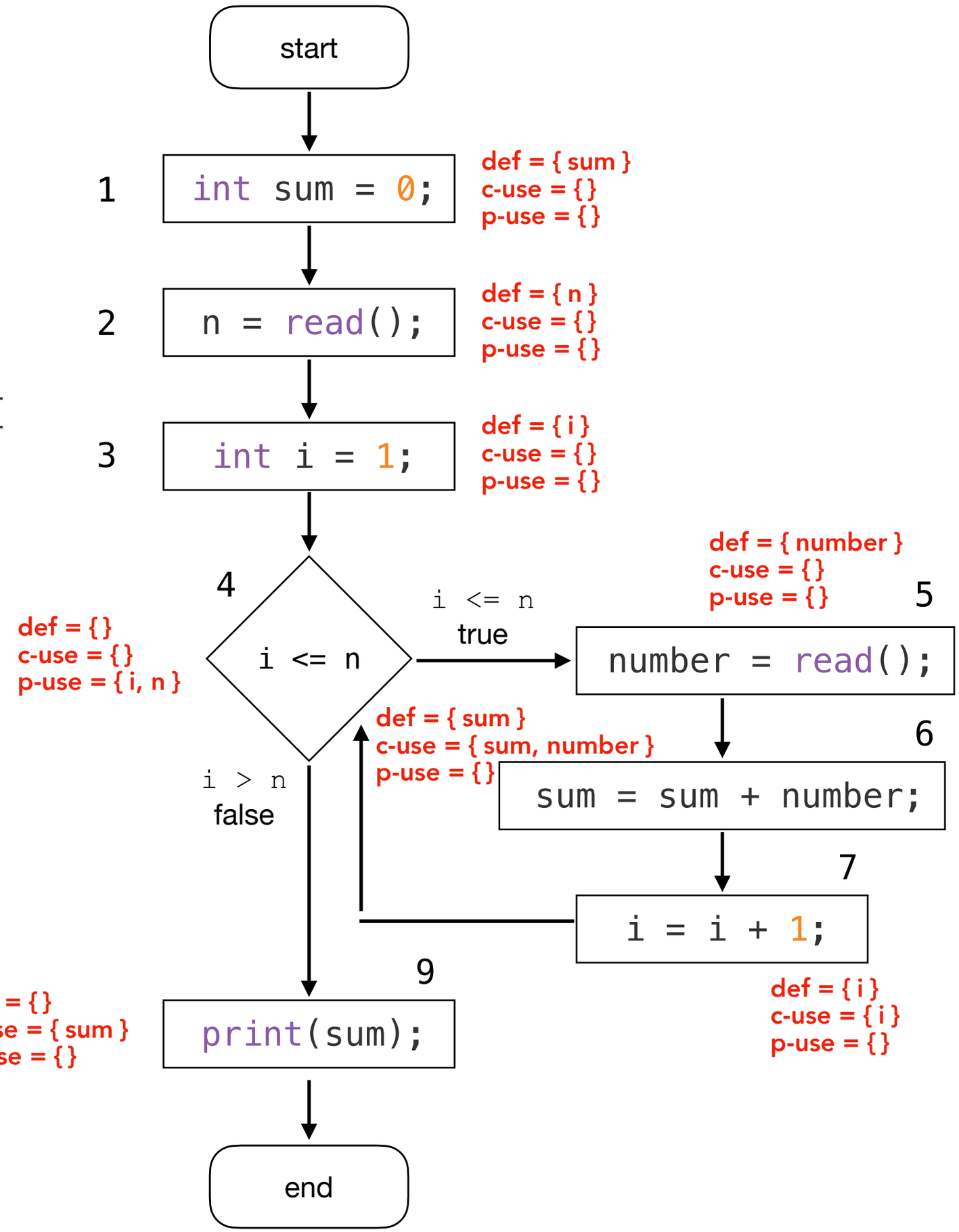
The variables of interest are **sum**, **i**, **n**, and **number**. Since the goal is to satisfy the all def-use criteria we will need to tabulate the def-use occurrences for each of these variables.



```
   public int sum_of_n_numbers() {
1      int sum = 0;
2      n = read();
3      int i = 1;
4      while (i <= n) {
5          number = read();
6          sum = sum + number;
7          i = i + 1;
8      }
9      print(sum);
   }
```



```
   public int sum_of_n_numbers() {
1      int sum = 0;
2      n = read();
3      int i = 1;
4      while (i <= n) {
5          number = read();
6          sum = sum + number;
7          i = i + 1;
8      }
9      print(sum);
   }
```



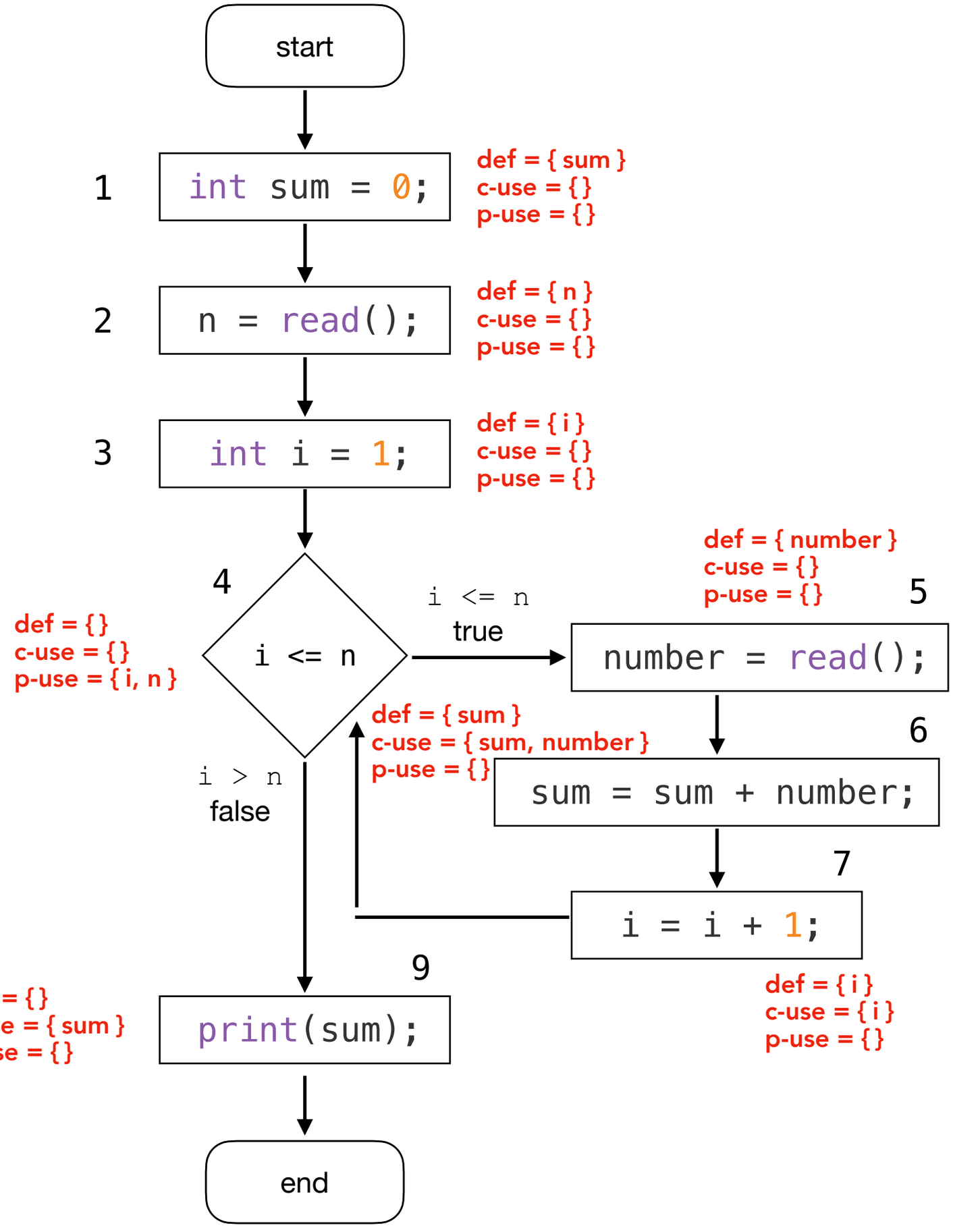
each def-use pair is assigned an identifier

var	id	def	use	*path*

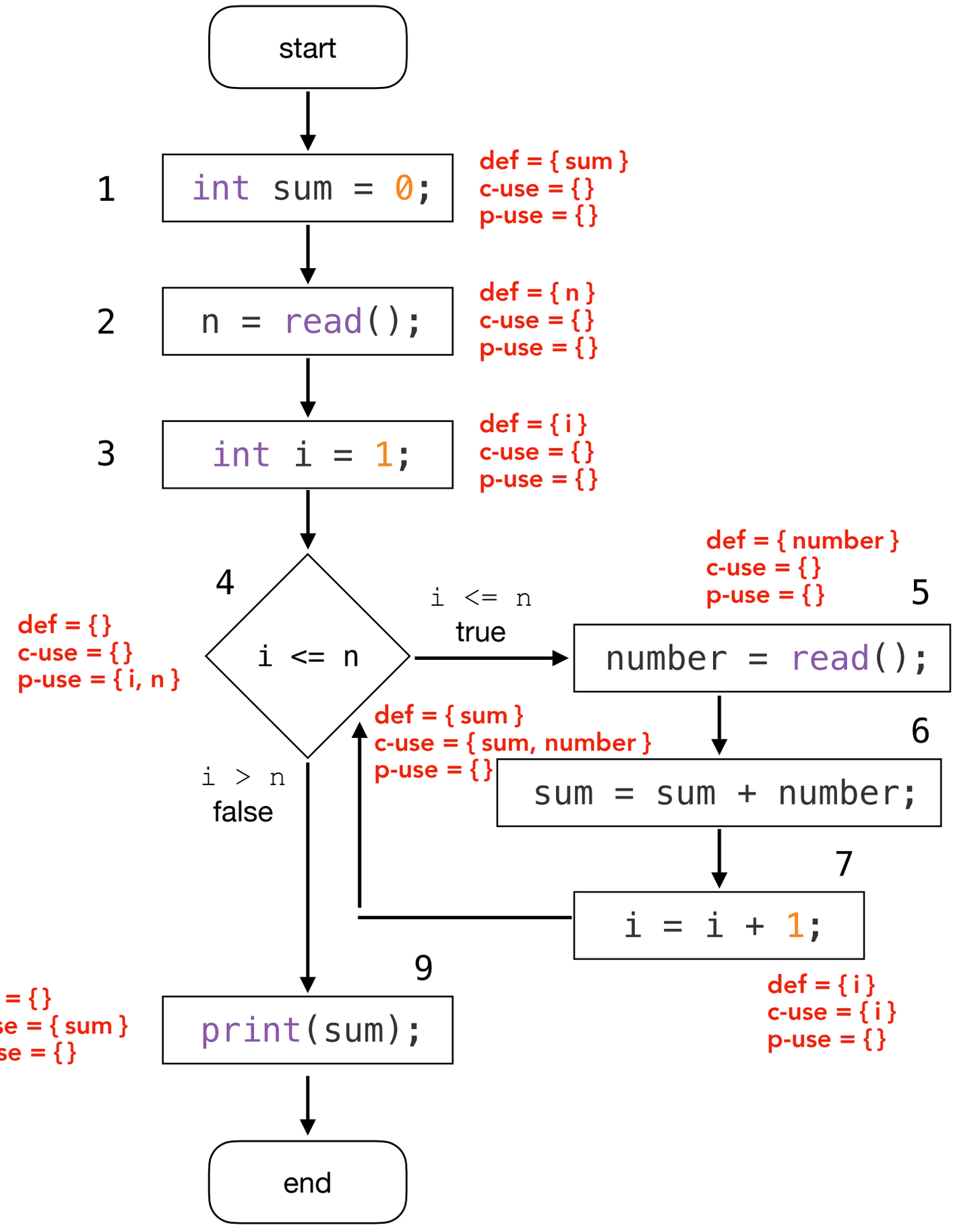
variable name

full path

Line numbers are used to show occurrence of the *def* or *use*.

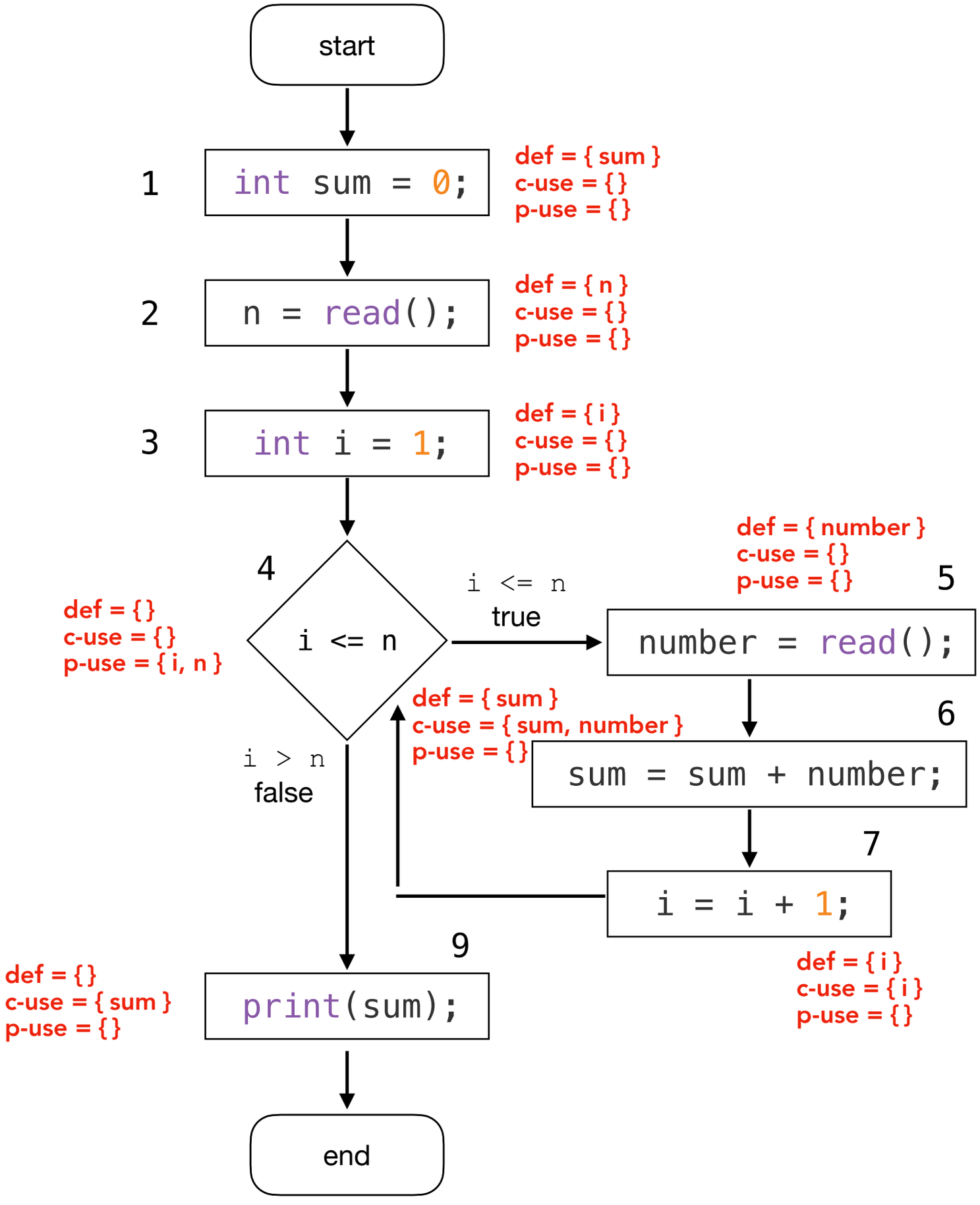


var	id	def	use	*path*
n = 0	1	2	4	<2,3,4>



var	id	def	use	*path*
n = 0	1	2	4	<2,3,4>

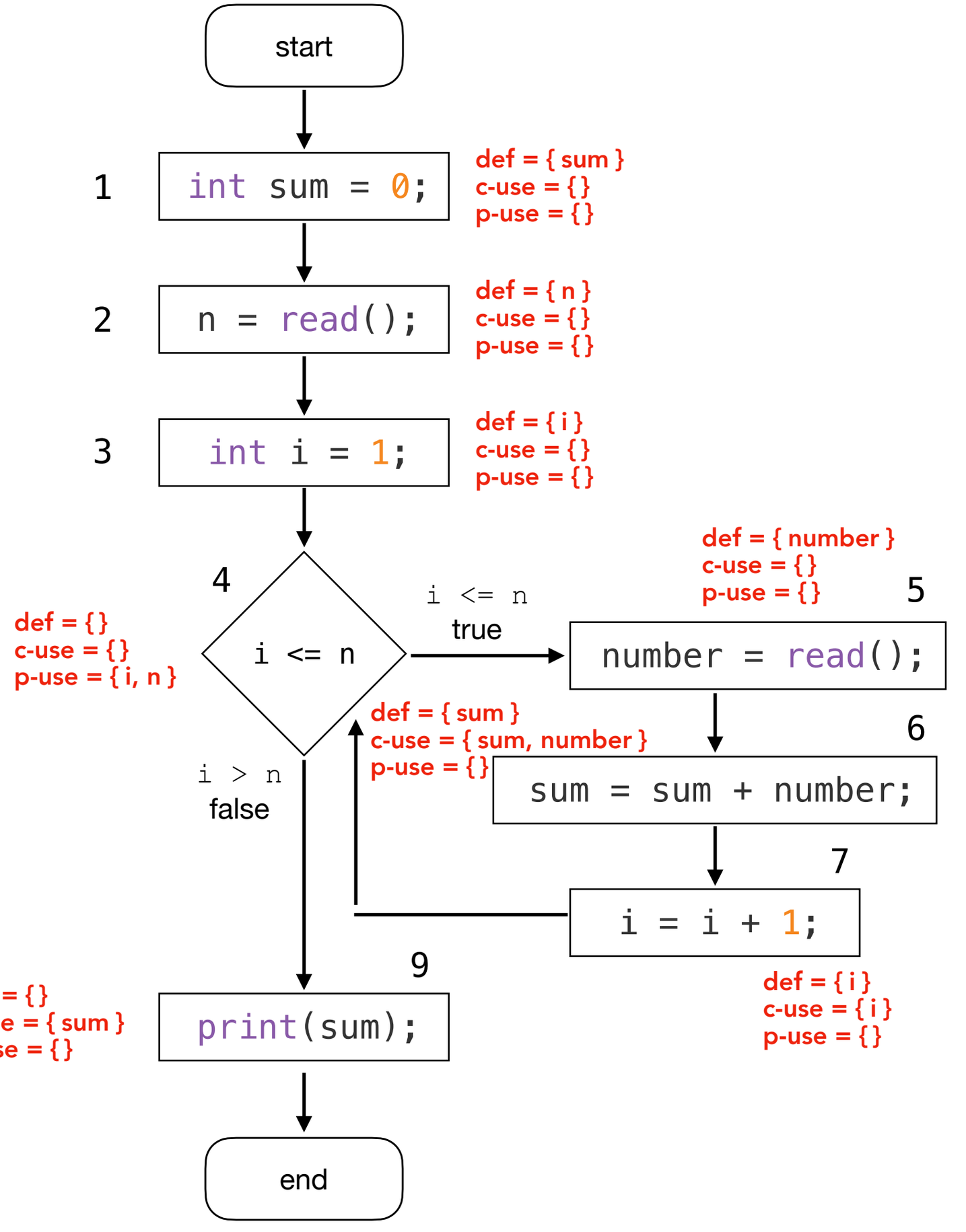
var	id	def	use	*path*
number	1	5	6	<5,6>



var	id	def	use	*path*
n = 0	1	2	4	<2,3,4>

var	id	def	use	*path*
number	1	5	6	<5,6>

var	id	def	use	*path*
sum	1	1	6	<1,2,3,4,5,6>
sum	2	1	9	<1,2,3,4,9>
sum	3	6	6	<6,6>
sum	4	6	9	<6,7,4,9>

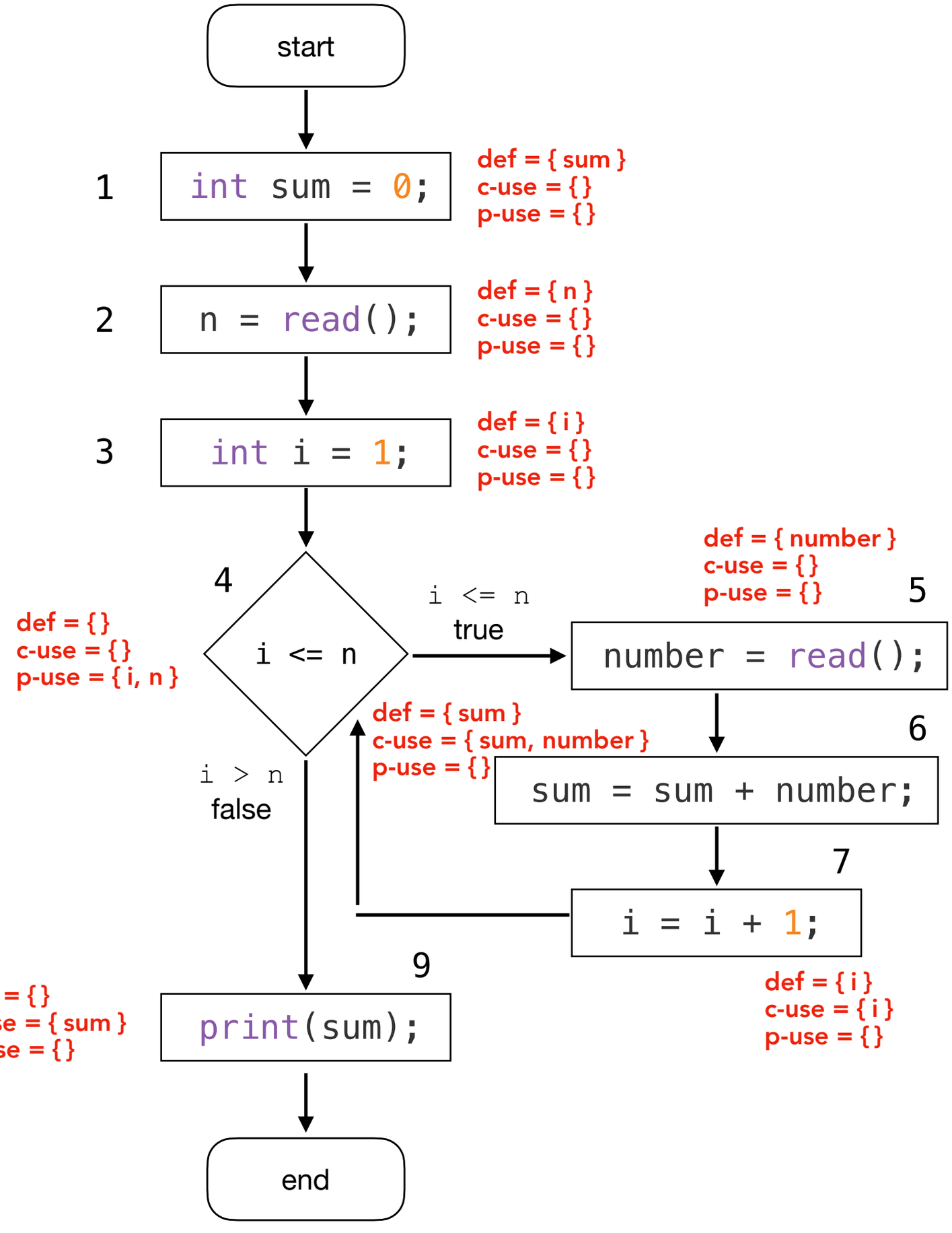


var	id	def	use	*path*
n = 0	1	2	4	<2,3,4>

var	id	def	use	*path*
number	1	5	6	<5,6>

var	id	def	use	*path*
sum	1	1	6	<1,2,3,4,5,6>
sum	2	1	9	<1,2,3,4,9>
sum	3	6	6	<6,6>
sum	4	6	9	<6,7,4,9>

var	id	def	use	*path*
i	1	3	4	<3,4>
i	2	3	7	<3,4,5,6,7>
i	3	7	7	<7,7>
i	4	7	4	<7,4>

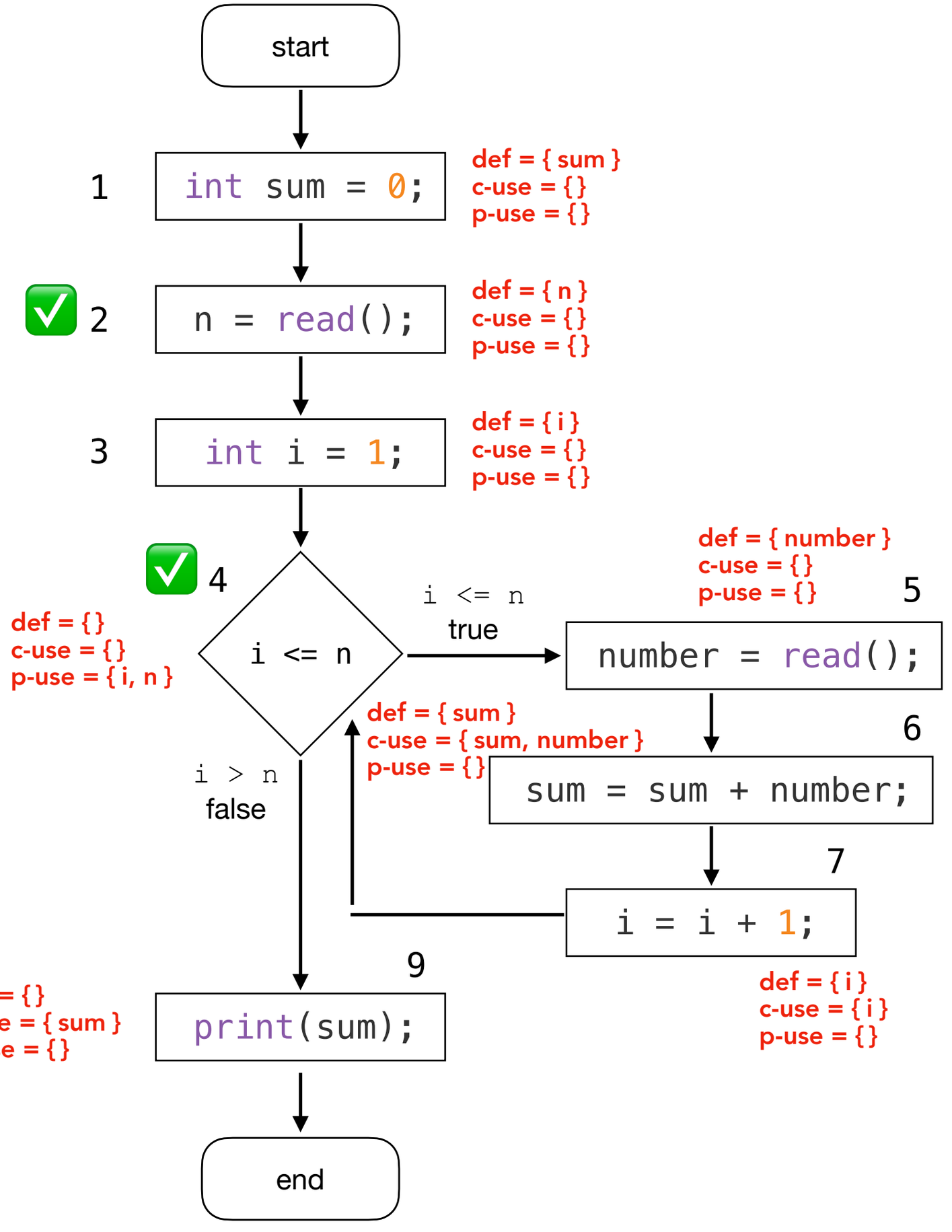


var	id	def	use	*path*
n = 0	1	2	4	<2,3,4>

var	id	def	use	*path*
number	1	5	6	<5,6>

var	id	def	use	*path*
sum	1	1	6	<1,2,3,4,5,6>
sum	2	1	9	<1,2,3,4,9>
sum	3	6	6	<6,6>
sum	4	6	9	<6,7,4,9>

var	id	def	use	*path*
i	1	3	4	<3,4>
i	2	3	7	<3,4,5,6,7>
i	3	7	7	<7,7>
i	4	7	4	<7,4>

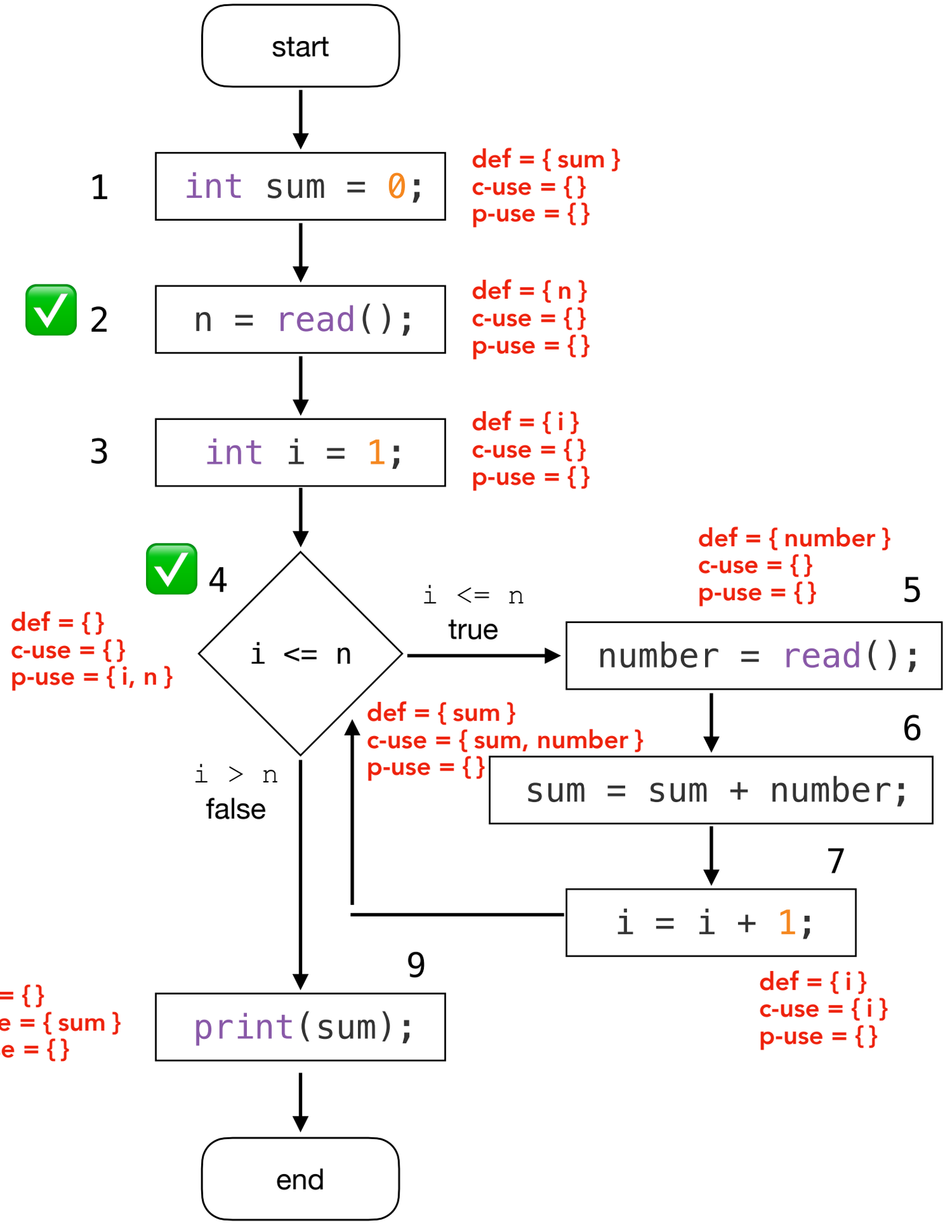


var	id	def	use	*path*
✅ n = 0	1	2	4	<2,3,4>

var	id	def	use	*path*
number	1	5	6	<5,6>

var	id	def	use	*path*
sum	1	1	6	<1,2,3,4,5,6>
sum	2	1	9	<1,2,3,4,9>
sum	3	6	6	<6,6>
sum	4	6	9	<6,7,4,9>

var	id	def	use	*path*
i	1	3	4	<3,4>
i	2	3	7	<3,4,5,6,7>
i	3	7	7	<7,7>
i	4	7	4	<7,4>

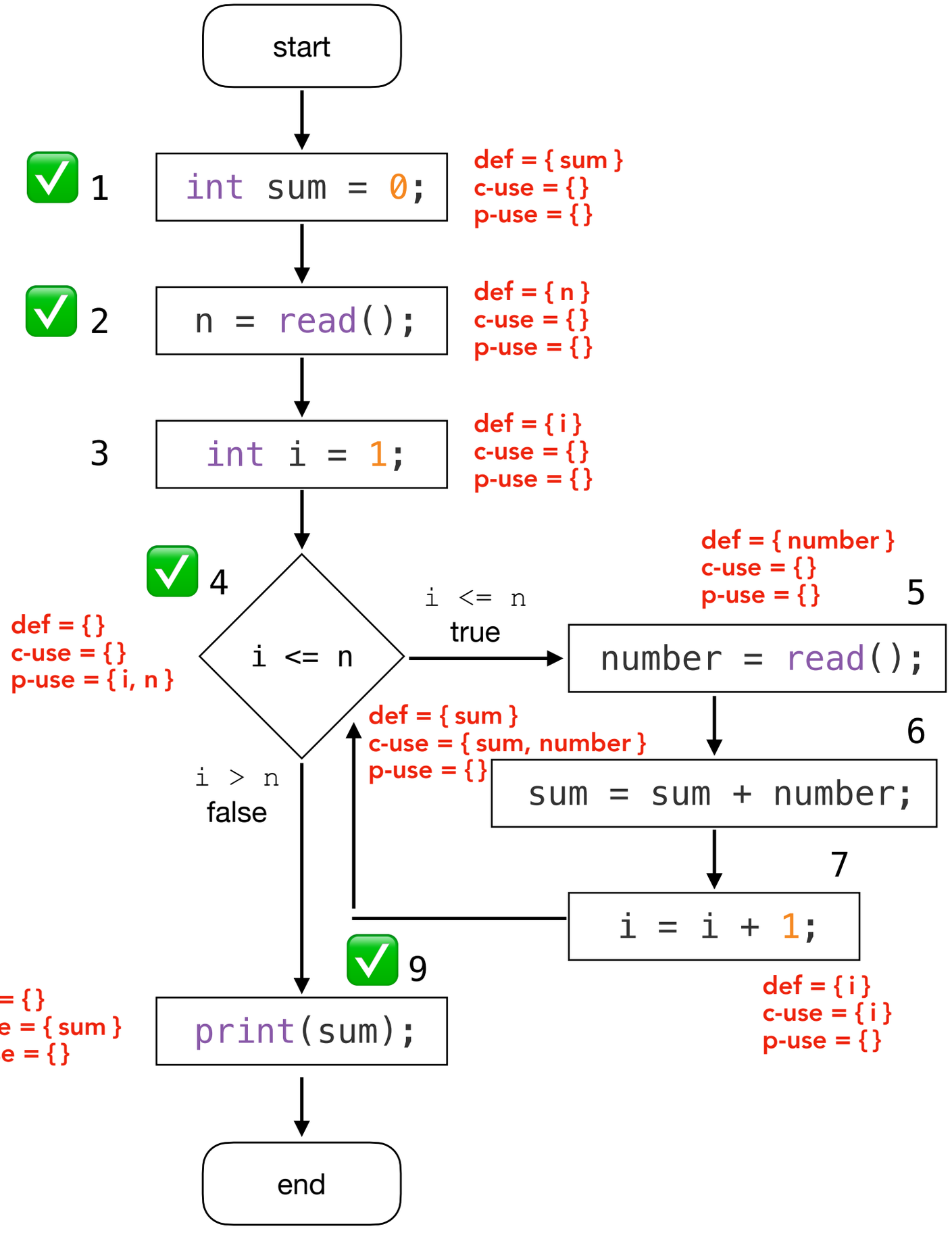


var	id	def	use	*path*
n = 0	1	2	4	<2,3,4>

var	id	def	use	*path*
number	1	5	6	<5,6>

var	id	def	use	*path*
sum	1	1	6	<1,2,3,4,5,6>
sum	2	1	9	<1,2,3,4,9>
sum	3	6	6	<6,6>
sum	4	6	9	<6,7,4,9>

var	id	def	use	*path*
i	1	3	4	<3,4>
i	2	3	7	<3,4,5,6,7>
i	3	7	7	<7,7>
i	4	7	4	<7,4>

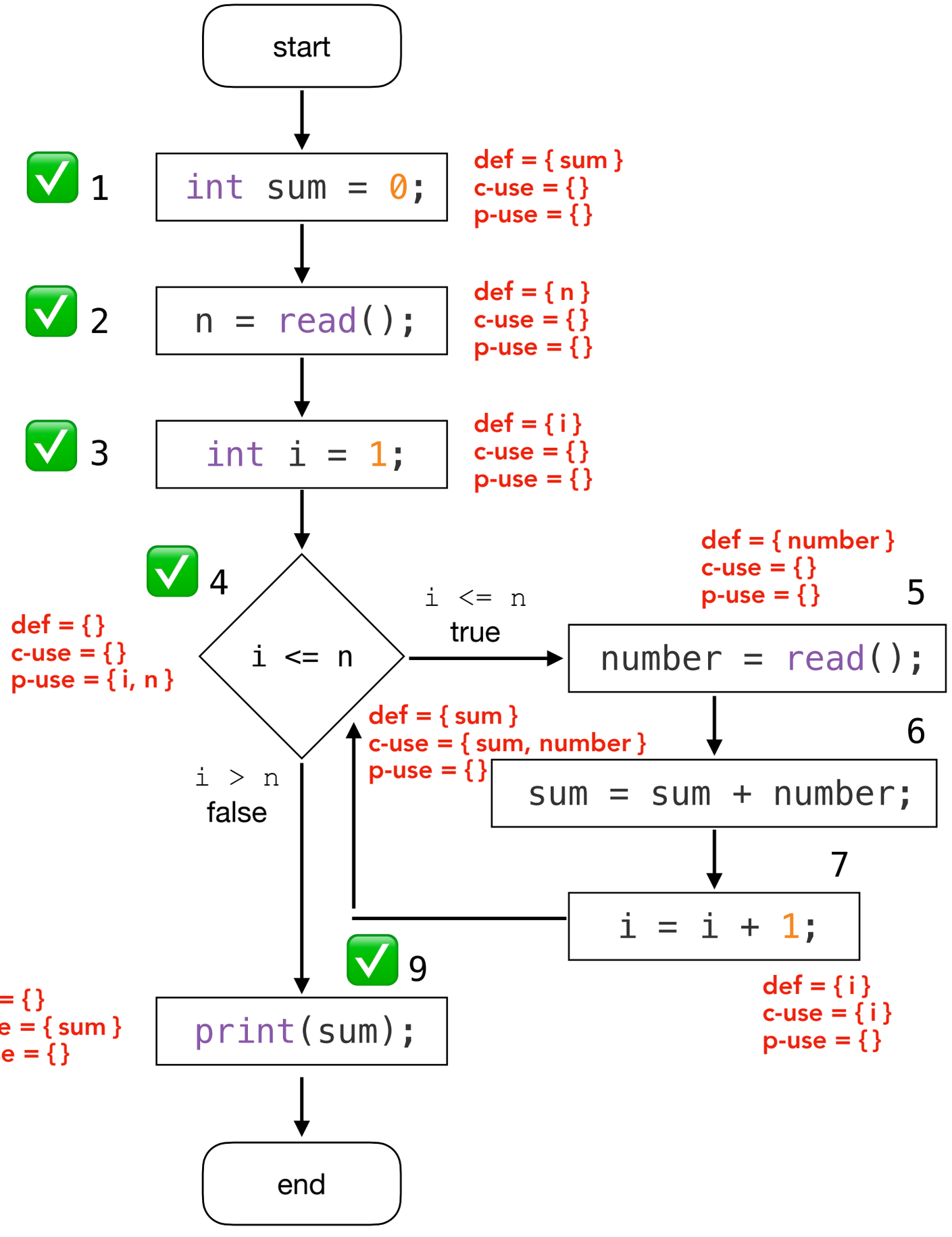


var	id	def	use	*path*
n = 0	1	2	4	<2,3,4>

var	id	def	use	*path*
number	1	5	6	<5,6>

var	id	def	use	*path*
sum	1	1	6	<1,2,3,4,5,6>
sum	2	1	9	<1,2,3,4,9>
sum	3	6	6	<6,6>
sum	4	6	9	<6,7,4,9>

var	id	def	use	*path*
i	1	3	4	<3,4>
i	2	3	7	<3,4,5,6,7>
i	3	7	7	<7,7>
i	4	7	4	<7,4>

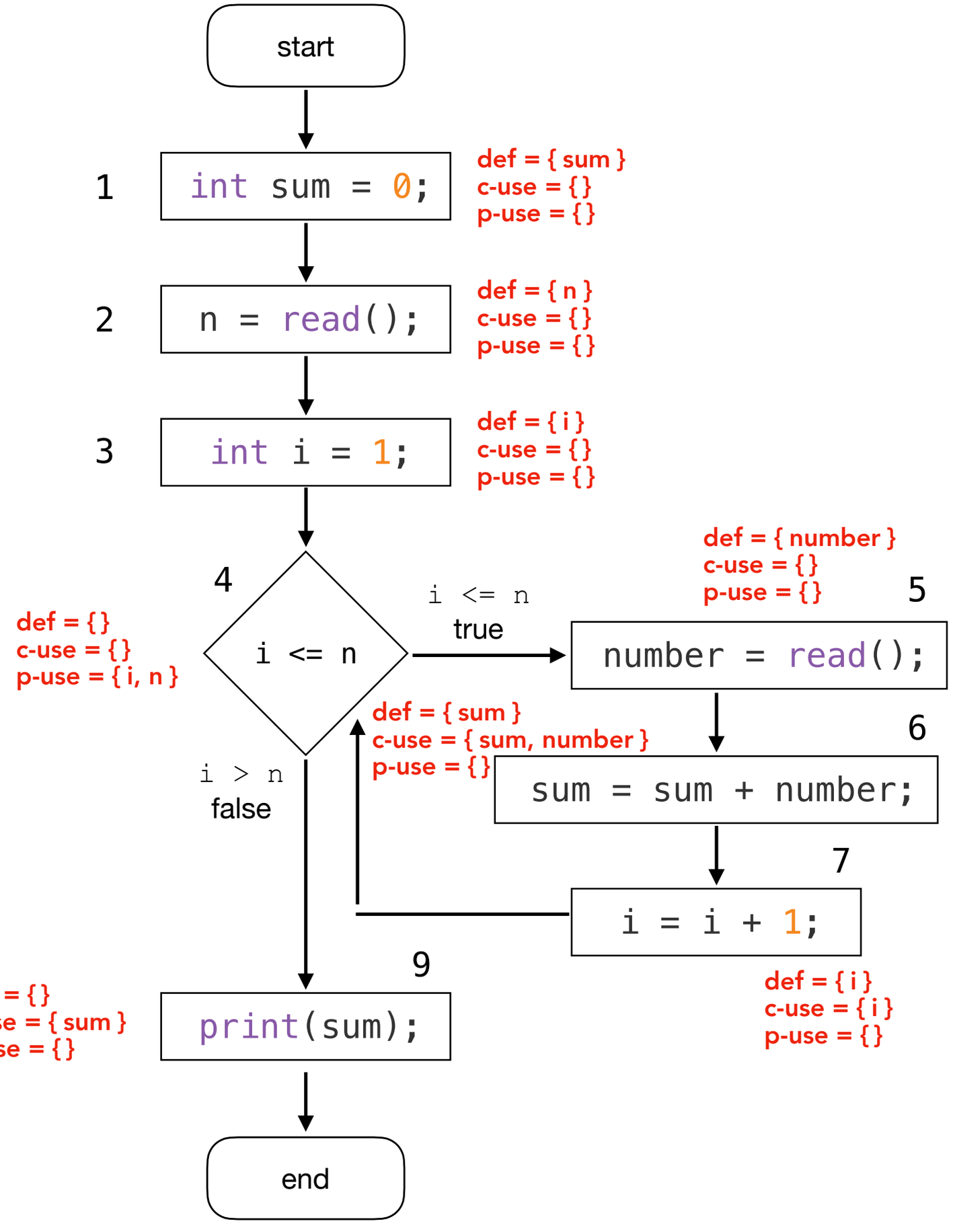


var	id	def	use	*path*
n = 1	1	2	4	<2,3,4>

var	id	def	use	*path*
number	1	5	6	<5,6>

var	id	def	use	*path*
sum	1	1	6	<1,2,3,4,5,6>
sum	2	1	9	<1,2,3,4,9>
sum	3	6	6	<6,6>
sum	4	6	9	<6,7,4,9>

var	id	def	use	*path*
i	1	3	4	<3,4>
i	2	3	7	<3,4,5,6,7>
i	3	7	7	<7,7>
i	4	7	4	<7,4>

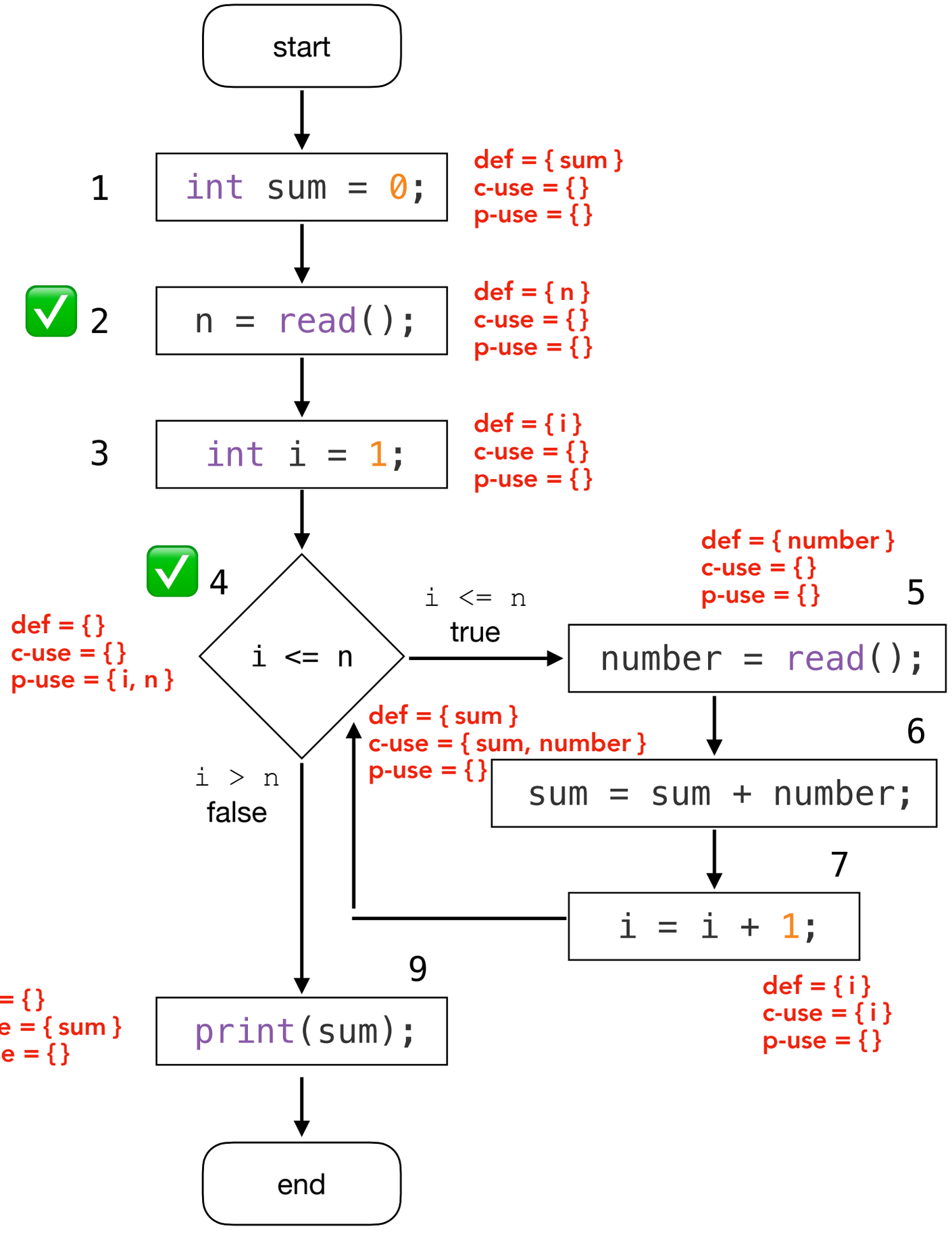


var	id	def	use	*path*
n = 1	1	2	4	<2,3,4>

var	id	def	use	*path*
number	1	5	6	<5,6>

var	id	def	use	*path*
sum	1	1	6	<1,2,3,4,5,6>
sum	2	1	9	<1,2,3,4,9>
sum	3	6	6	<6,6>
sum	4	6	9	<6,7,4,9>

var	id	def	use	*path*
i	1	3	4	<3,4>
i	2	3	7	<3,4,5,6,7>
i	3	7	7	<7,7>
i	4	7	4	<7,4>

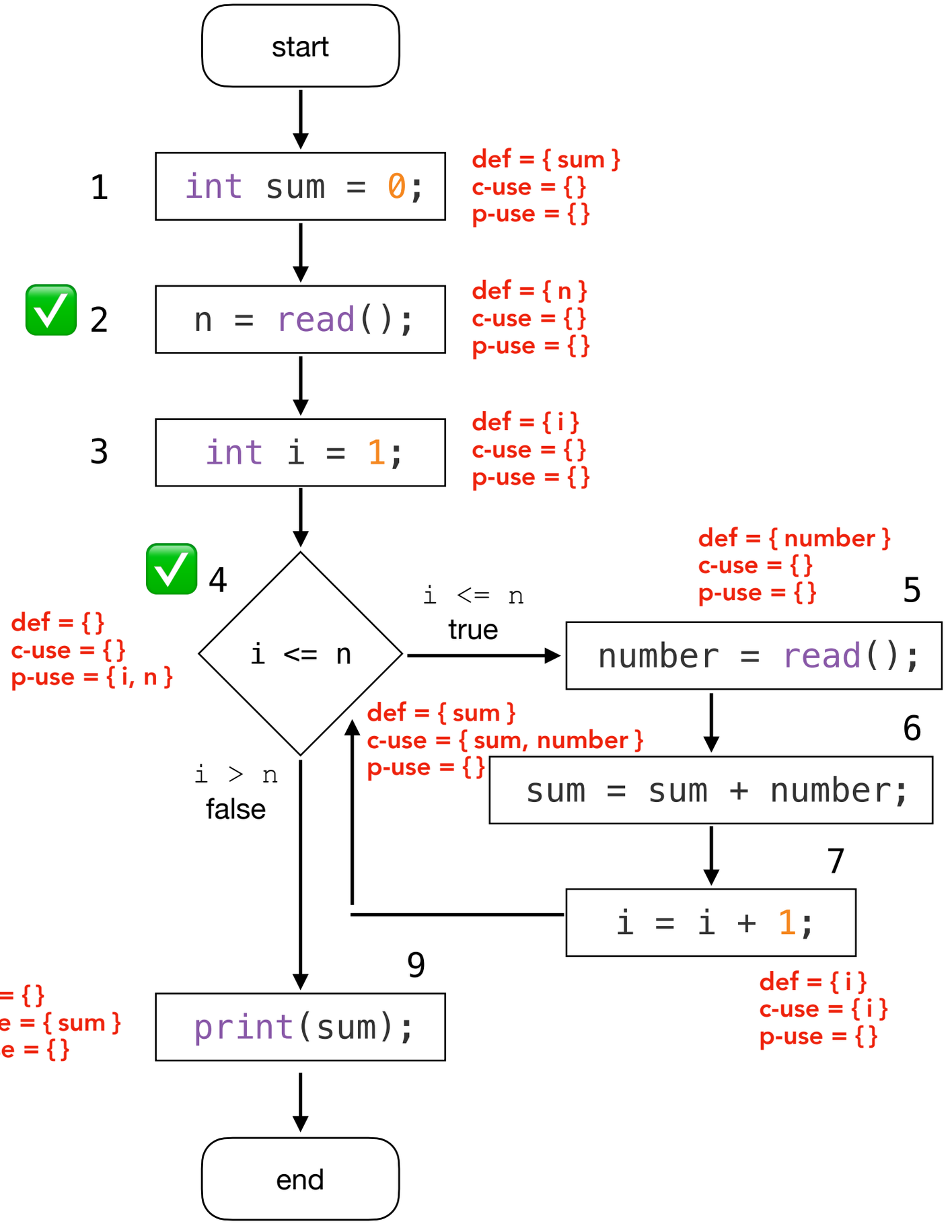


var	id	def	use	*path*
✅ n = 1	1	2	4	<2,3,4>

var	id	def	use	*path*
number	1	5	6	<5,6>

var	id	def	use	*path*
sum	1	1	6	<1,2,3,4,5,6>
sum	2	1	9	<1,2,3,4,9>
sum	3	6	6	<6,6>
sum	4	6	9	<6,7,4,9>

var	id	def	use	*path*
i	1	3	4	<3,4>
i	2	3	7	<3,4,5,6,7>
i	3	7	7	<7,7>
i	4	7	4	<7,4>

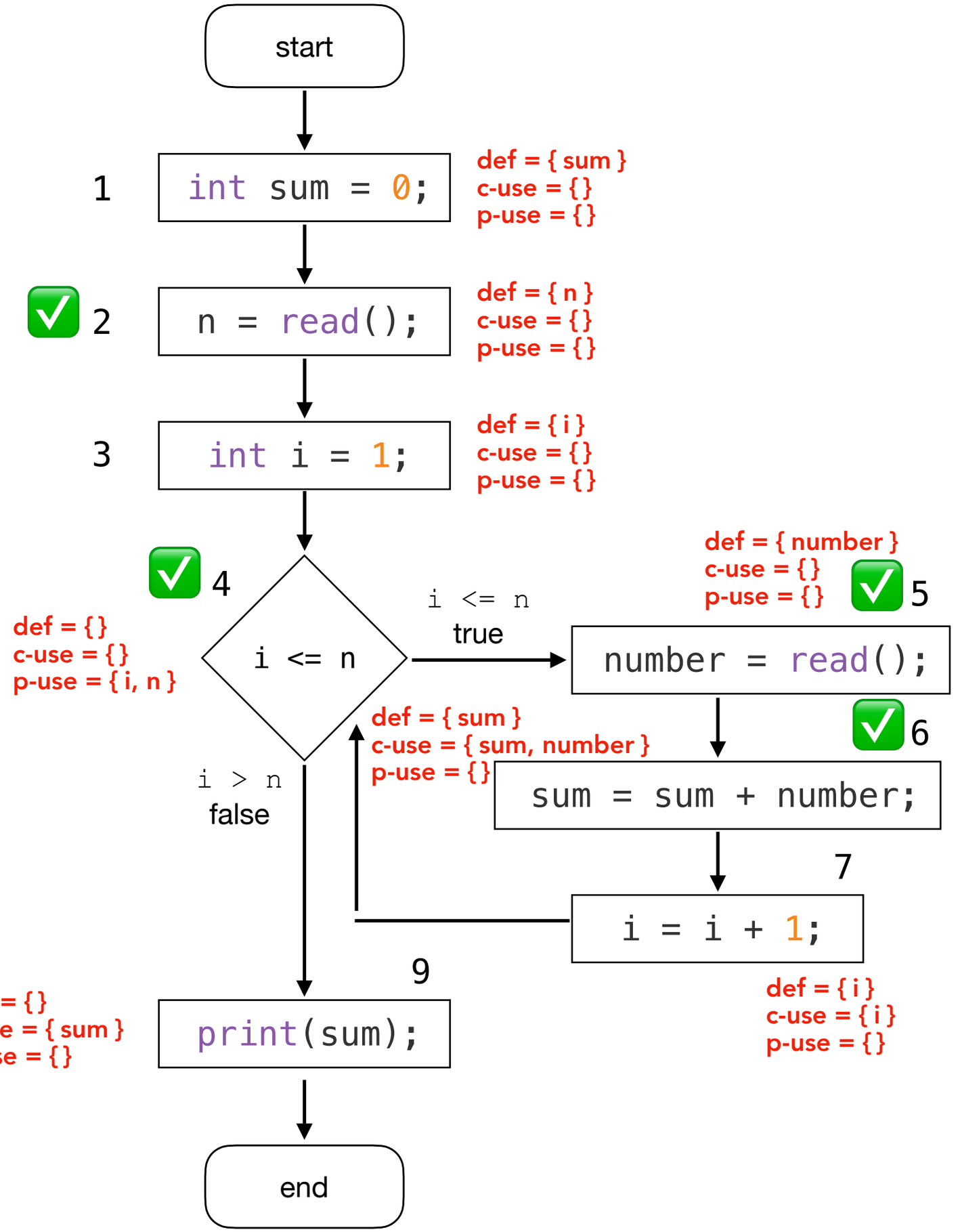


var	id	def	use	*path*
n = 1	1	2	4	<2,3,4>

var	id	def	use	*path*
number	1	5	6	<5,6>

var	id	def	use	*path*
sum	1	1	6	<1,2,3,4,5,6>
sum	2	1	9	<1,2,3,4,9>
sum	3	6	6	<6,6>
sum	4	6	9	<6,7,4,9>

var	id	def	use	*path*
i	1	3	4	<3,4>
i	2	3	7	<3,4,5,6,7>
i	3	7	7	<7,7>
i	4	7	4	<7,4>

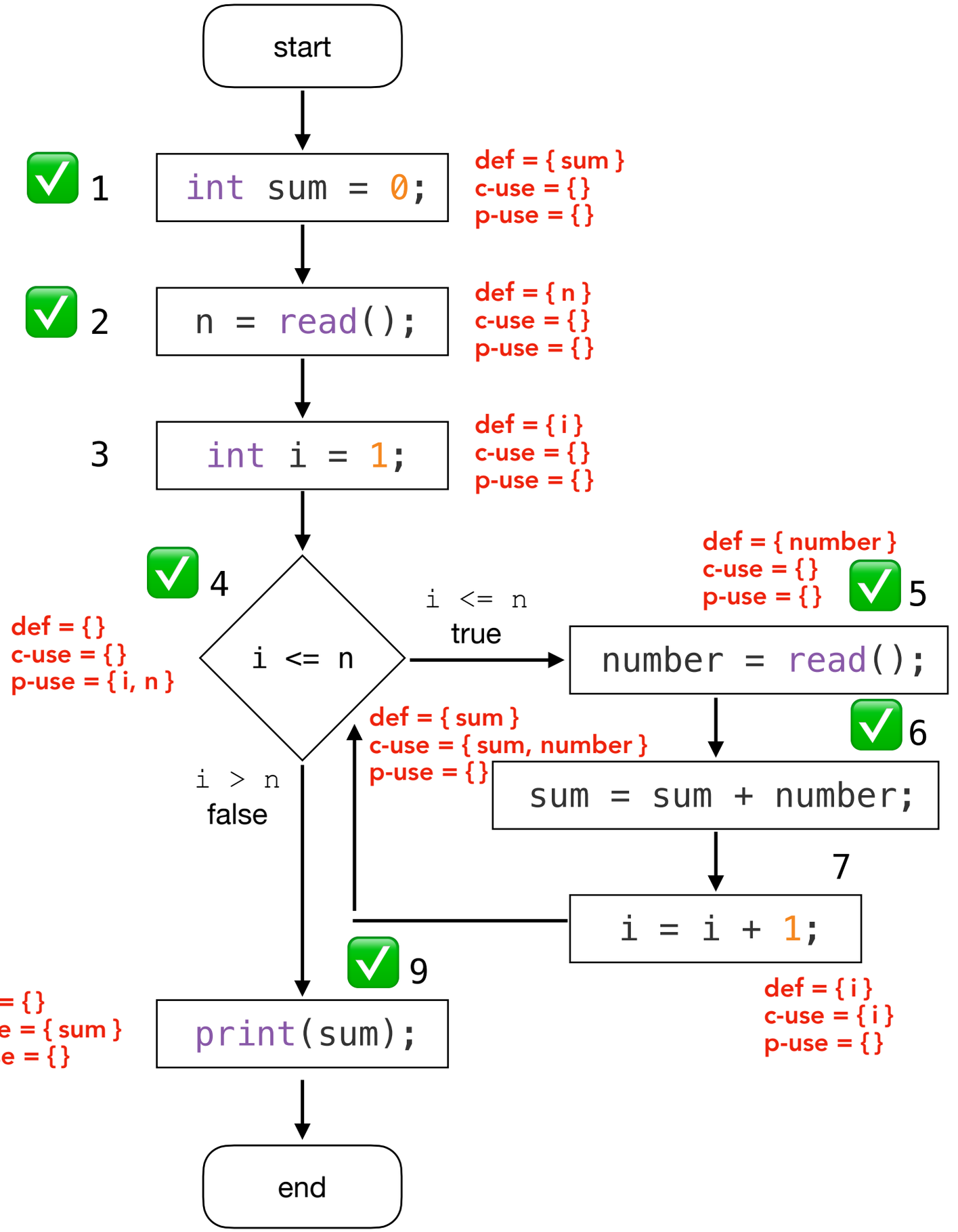


var	id	def	use	*path*
n = 1	1	2	4	<2,3,4>

var	id	def	use	*path*
number	1	5	6	<5,6>

var	id	def	use	*path*
sum	1	1	6	<1,2,3,4,5,6>
sum	2	1	9	<1,2,3,4,9>
sum	3	6	6	<6,6>
sum	4	6	9	<6,7,4,9>

var	id	def	use	*path*
i	1	3	4	<3,4>
i	2	3	7	<3,4,5,6,7>
i	3	7	7	<7,7>
i	4	7	4	<7,4>



	var	id	def	use	*path*
✅	n = 1	1	2	4	<2,3,4>

	var	id	def	use	*path*
✅	number	1	5	6	<5,6>

	var	id	def	use	*path*
✅	sum	1	1	6	<1,2,3,4,5,6>
	sum	2	1	9	<1,2,3,4,9>
✅	sum	3	6	6	<6,6>
✅	sum	4	6	9	<6,7,4,9>

	var	id	def	use	*path*
✅	i	1	3	4	<3,4>
✅	i	2	3	7	<3,4,5,6,7>
✅	i	3	7	7	<7,7>
✅	i	4	7	4	<7,4>