Program Testing and Analysis

—Final Exam—

Department of Computer Science
University of Stuttgart

Winter semester 2020/21, March 3, 2021

Note: The solutions provided here may not be the only valid solutions.

Part 1 [4 points]

1. Which of the following statements is true? (Only one statement is true.)

   - [ ] In operational semantics, a program is blocked if the program reaches an `if` statement.
   - [x] In operational semantics, a program is divergent if the evaluation sequence is infinite.
   - [ ] In operational semantics, axioms and rules specify the inputs a program is executed with.
   - [ ] In operational semantics, the evaluation sequence is always bounded by the number of statements in the program.
   - [ ] In operational semantics, every transition increases the number of entries in the store.

2. Which of the following statements is true? (Only one statement is true.)

   - [x] Feedback-directed random testing incrementally builds test cases while avoiding illegal and redundant method sequences.
   - [ ] Feedback-directed random testing provides feedback to developers about the effectiveness of manually written tests.
   - [ ] Feedback-directed random testing relies on developers to provide feedback about the usefulness of generated tests.
   - [ ] Feedback-directed random testing systematically exercises all paths through a program.
   - [ ] Feedback-directed random testing reduces a given program to facilitate debugging an already known crash.

3. Which of the following statements is true? (Only one statement is true.)

   - [ ] A backward data flow analysis starts at the entry node of a control flow graph.
   - [ ] The domain of a backward data flow analysis is the empty set.
   - [x] A forward data flow analysis starts at the entry node of a control flow graph.
   - [ ] The domain of a forward data flow analysis is the empty set.
   - [ ] The meet operator of a data flow analysis describes the initial state of the analysis.

4. Which of the following statements is true? (Only one statement is true.)

   - [ ] Symbolic execution tries to exercise all paths through a program dependency graph.
   - [x] Symbolic execution tries to exercise all paths through an execution tree.
   - [ ] Symbolic execution tries to exercise all paths through a tree of interleavings.
   - [ ] Symbolic execution tries to exercise all paths through a call graph.
   - [ ] Symbolic execution tries to exercise all paths through an abstract syntax tree.

Part 2 [10 points]

Consider the following SIMP program:

```
while !b-2 < 5 do (if !a != 7 then b := !b+2 else skip)
```

1. Give the semantics of the program as a sequence of transitions of the abstract machine for SIMP that was introduced in the lecture. For your reference, the appendix provides the transition rules (copied from Fernandez' book).

   You only have to give the first eight transitions, as well as the final configuration of the abstract machine. Use the following template to present your solution. We provide two lines for each configuration. The template starts with the initial configuration.

   You may use the following abbreviations:

Abbreviation	Meaning
B	`if !a != 7 then b := !b+2 else skip`
C	`!b-2 < 5`
s	$\{a \mapsto 4, b \mapsto 4\}$

$\langle$`while !b-2 < 5 do (if !a != 7 then b := !b+2 else skip)` $\circ\ nil, nil, \{a \mapsto 4, b \mapsto 4\}\rangle$

$\rightarrow \langle$`!b-2 < 5` $\circ$ `while` $\circ\ nil,$ `!b-2 < 5` $\circ$ `if !a != 7 then b := !b+2 else skip` $\circ\ nil, \{a \mapsto 4, b \mapsto 4\}\rangle$

$\rightarrow \langle$`!b-2` $\circ$ `5` $\circ$ `<` $\circ$ `while` $\circ\ nil,$ `!b-2 < 5` $\circ$ `if !a != 7 then b := !b+2 else skip` $\circ\ nil, \{a \mapsto 4, b \mapsto 4\}\rangle$

$\rightarrow \langle$`!b` $\circ$ `2` $\circ$ `-` $\circ$ `5` $\circ$ `<` $\circ$ `while` $\circ\ nil,$ `!b-2 < 5` $\circ$ `if !a != 7 then b := !b+2 else skip` $\circ\ nil, \{a \mapsto 4, b \mapsto 4\}\rangle$

$\rightarrow \langle$`2` $\circ$ `-` $\circ$ `5` $\circ$ `<` $\circ$ `while` $\circ\ nil, 4 \circ$ `!b-2 < 5` $\circ$ `if !a != 7 then b := !b+2 else skip` $\circ\ nil, \{a \mapsto 4, b \mapsto 4\}\rangle$

$\rightarrow \langle$`-` $\circ$ `5` $\circ$ `<` $\circ$ `while` $\circ\ nil, 2 \circ 4 \circ$ `!b-2 < 5` $\circ$ `if !a != 7 then b := !b+2 else skip` $\circ\ nil, \{a \mapsto 4, a \mapsto 4\}\rangle$

$\rightarrow \langle$`5` $\circ$ `<` $\circ$ `while` $\circ\ nil, 2 \circ$ `!b-2 < 5` $\circ$ `if !a != 7 then b := !b+2 else skip` $\circ\ nil, \{a \mapsto 4, b \mapsto 4\}\rangle$

$\rightarrow \langle$`<` $\circ$ `while` $\circ\ nil, 5 \circ 2 \circ$ `!b-2 < 5` $\circ$ `if !a != 7 then b := !b+2 else skip` $\circ\ nil, \{a \mapsto 4, b \mapsto 4\}\rangle$

$\rightarrow \langle$`while` $\circ\ nil, True \circ$ `!b-2 < 5` $\circ$ `if !a != 7 then b := !b+2 else skip` $\circ\ nil, \{a \mapsto 4, b \mapsto 4\}\rangle$

$\rightarrow^* \langle nil, nil, \{a \mapsto 4, b \mapsto 8\}\rangle$

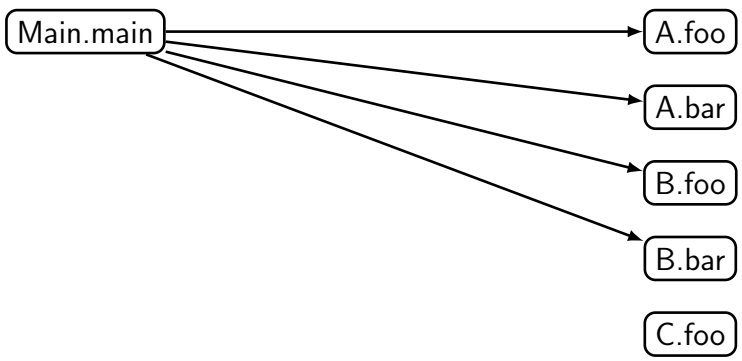
2. Does the program terminate successfully?
   - [x] Yes.
   - [ ] No.

Part 3 [11 points]

Consider the following Java program:

```
1  class CallGraph {
2      static class Main {
3          public static void main(String[] args) {
4              A x;
5              if (args.length == 2)
6                  x = new A();
7              else
8                  x = new B();
9              B y = new B();
10             x = y;
11             if (x.foo() == 0) {
12                 x.bar();
13             }
14         }
15     }
16
17     static class A {
18         int foo() {
19             return 5;
20         }
21
22         void bar() {
23         }
24     }
25
26     static class B extends A {
27         int foo() {
28             return 3;
29         }
30
31         void bar () {
32         }
33     }
34
35     static class C extends A {
36         int foo() {
37             return 7;
38         }
39     }
40 }
```

1. Use the "Rapid Type Analysis" (RTA) approach to compute a call graph of the program. Use the following template to give your solution.



2. Explain why there is or why there is not (depending on your solution above) an edge between `Main.main` and `C.foo`.

   Solution:
   RTA checks which classes are instantiated in the analyzed program and considers only their methods as possible call targets. In the given program, class `C` is never instantiated, and hence, there cannot be a call to `C.foo` according to RTA.

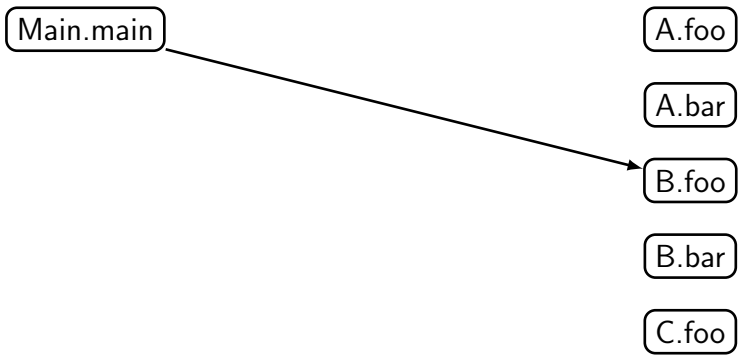
3. Now suppose we use "Declared-Type Analysis" (DTA) to compute the call graph.

   - Provide the type propagation graph computed by the analysis, including the type(s) associated with each node.

   - Does DTA change the call graph compared to the graph computed with RTA? If it changes, explain how and why; otherwise, explain why not.

     Solution:
     No, DTA does not change the call graph for this program. According the type propagation graph given above, variables with statically declared type `A` may hold objects of types `A` and `B`. The calls at lines 11 and 12 have `x` as their base object, i.e., a variable of statically declared type `A`. Hence, DTA concludes that the call at line 11 can be dispatched to `A.foo` or `B.foo`, and that the call at line 12 can be dispatched to `A.bar` and `B.bar`, which matches the call graph produced by RTA.

4. Give an ideal call graph for the above program, where "ideal" means that it contains exactly those edges that may occur in an execution of the program. Use the following template to provide your solution.



5. Explain why there is or why there is not (depending on your solution above) an edge between `Main.main` and `B.bar`.

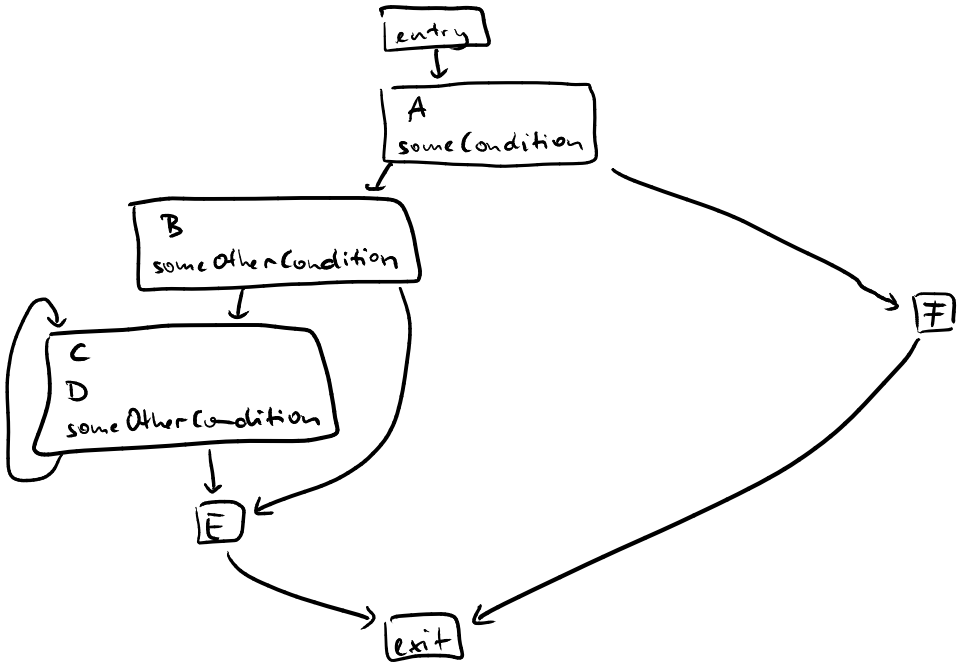
   Solution:
   The only call to a `bar` method is at line 12. However, this line will not be reached in any execution of the program because the conditional check at line 11 always evalutes to `false`. The reason is that the `B.foo` method called at line 11 always returns `3`, or more generally, that all `foo` methods always return a non-zero value.

Part 4 [13 points]

Suppose we generate inputs for the following piece of code using the AFL fuzzer (note: `A`, `B`, etc. denote individual statements, whose details are irrelevant):

```
1  A
2  if (someCondition) {
3    B
4    while (someOtherCondition) {
5      C
6      D
7    }
8    E
9  } else {
10   F
11 }
```

1. Draw a control flow graph of the code, where nodes are basic blocks. Your graph should have entry and exit nodes.



2. Assume AFL has already generated five inputs, which trigger executions that cover the following lines:

Input	Covered lines	Coverage as measured by AFL
Input 1	1, 2, 3, 4, 8	AB, BE
Input 2	1, 2, 3, 4, 5, 6, 4, 8	AB, BCD, CDE
Input 3	1, 2, 3, 4, 5, 6, 4, 5, 6, 4, 8	AB, BCD, CDCD, CDE
Input 4	1, 2, 3, 4, 5, 6, 4, 5, 6, 4, 5, 6, 4, 8	AB, BCD, CDCD, CDE
Input 5	1, 2, 10	AF

   Indicate which coverage information AFL tracks for these inputs by providing the covered edges. Use the last column of the table to provide your answers and provide a short explanation of your notation below. (Here and in the following question, assume that the "edge hit count" refinement discussed in the lecture is not considered, but only whether edges are covered or not.)

   Explanation of notation:

   Solution:
   Each entry in the last column means an edge in the control flow graph that is covered by an execution. For example, "AB" stands for the edge between the basic block that contains `A` and the basic block that contains `B`, whereas "CDCD" stands for the edge between the basic block that contains both `C` and `D` back into itself.

3. Which of the inputs will be retained for further mutation? Explain your answer.

   Solution: AFL will retain input 1, input 2, input 5, as well as either input 3 or input 4. (Explanation: Inputs 3 and 4 cover the same edges, whereas all other inputs cover some edge not covered by the other inputs.)

4. Suppose that one of the inputs to mutate is the following (represented as a sequence of bits):

   1011 0001 1001 1011 1111 0011 0110 0001

   Show two mutations that AFL might create and explain how they are created.

   - Mutation 1:
     Solution:
     1111 0001 1001 1011 1111 0011 0110 0001
     (created by a bit flip in the first byte)
   - Mutation 2:
     Solution:
     1011 0001 0000 0000 1001 1011 1111 0011 0110 0001
     (created by inserting the known interesting integer zero as an 8-bit value)

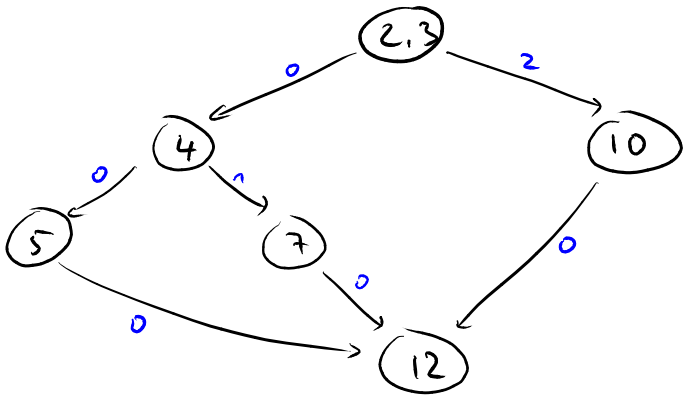
Part 5 [14 points]

Consider the following JavaScript code:

```
1  function profileMe(foo) {
2    var bar = foo;
3    if (bar > 0) {
4      if (bar == 12) {
5        console.log("just");
6      } else {
7        console.log("some");
8      }
9    } else {
10     console.log("strings");
11   }
12   return bar;
13 }
```

The following applies Ball-Larus path profiling.

1. Give the control flow graph of the function. Abbreviate statements with their line number.



2. Compute for each node in the graph the $NumPaths$, and assign to each edge in the graph an integer, so that the sum of integers along a path yields a unique number for each path. Follow the Ball-Larus algorithm presented in the lecture. Use the following table for $NumPaths$. Provide the integers of edges by adding them to the above control flow graph.

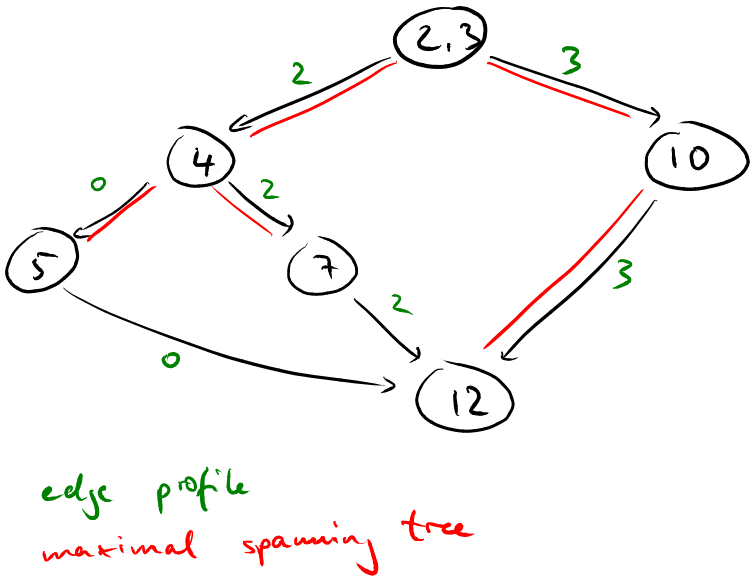
Solution:

Node n	$NumPaths(n)$
2,3	3
4	2
5	1
7	1
10	1
12	1

3. The second step of the Ball-Larus algorithm assigns addition operations to edges of the control flow graph. To determine where to add addition operations, suppose the function is called as follows:

```
1 f(0);
2 f(-1);
3 f(-2);
4 f(7);
5 f(14);
```

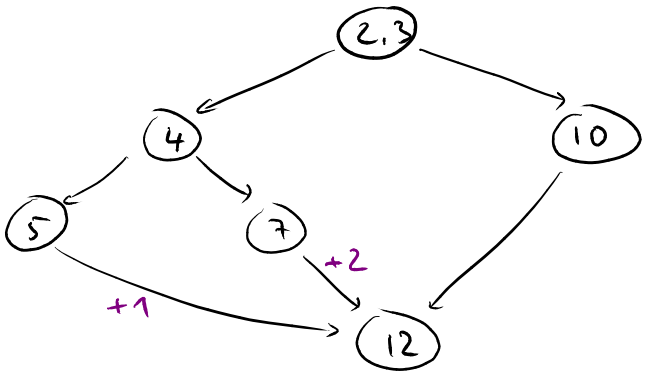
Based on these calls, provide the edge profile by reproducing the control flow graph while indicating how often each edge has been executed:



4. In this graph, show a spanning tree that maximizes the overall cost of edges that are part of the spanning tree. Based on the spanning tree, indicate the edges at which to perform addition operations. The overall goal is to minimize the number of addition operations for the given edge profile.

   (a) Reproduce the graph and indicate where to perform the addition operations by adding the operations to the respective edges:

   Solution:



(b) Based on the above solution, provide the path encodings in the following table:

Solution:

Path	Encoding
2,3 → 10 → 12	0
2,3 → 4 → 5 → 12	1
2,3 → 4 → 7 → 12	2

Part 6 [8 points]

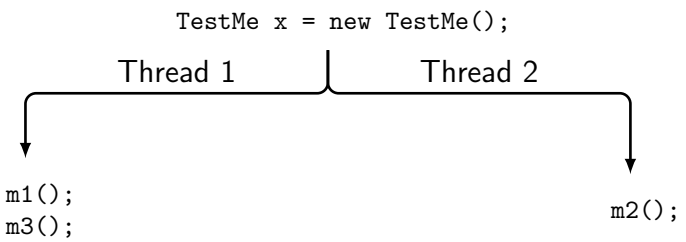
Consider the following Java class:

```
1  class TestMe {
2      int[] arr = { 0, 1, 2, 3 };
3      int index = 1;
4
5      void m1() {
6          synchronized (this) {
7              index = index + 3;
8              index = index - 2;
9          }
10     }
11
12     int m2() {
13         return this.arr[index];
14     }
15
16     void m3() {
17         synchronized (this) {
18             index = index - 1;
19         }
20     }
21 }
```

1. Even though the class is using some synchronization, it is not thread-safe. Explain why this class fails to be thread-safe.

   Solution:
   The methods of the class all access the `index` field, and even though `m1` and `m3` protect their accesses by synchronizing on `this`, `m2` fails to synchronize its access to `index`. As a result, concurrent calls to `m2` and any of the other two methods will lead to a data race, where `m2` reads `index` while the other method is writing into `index` without synchronizing the order of these concurrent accesses. The data race can lead to concurrent behavior that cannot occur in any linearization of the concurrent usage of the class, as illustrated in the example below.

2. Suppose the ConTeGe tool discussed in the lecture generates tests for the class. Provide a test that could expose a thread safety bug in the class. Use the following template to provide your solution, where each line should be filled by one method call.



3. Show an execution of the above test that causes an exception by providing the sequence of line numbers of the class that get executed in an order that is (i) possible with the test provided above and (ii) ends in an `ArrayIndexOutOfBoundsException` that does not happen in any linearization of the test.

   Solution:
   After the prefix has initialized the `TestMe` object, thread 1 executes `m1`. During the execution of `m1`, line 6 acquires the lock on `this`, and then line 7 increases `index` to `4`. Now, thread 2 starts to execute `m2`, which at line 13 reads value `4` of `index` and tries to look up this index in `this.arr`. Because the array has only four elements, the index is out of bounds, resulting in an `ArrayIndexOutOfBoundsException`. This concurrent behavior is not possible in any of the method-level linearizations of the test, because line 13 cannot be interleaved between lines 7 and 8 in a linearized execution.

(This page intentionally left blank.)

Appendix

You may remove the pages of the appendix to allow for easier reading.

For Part 2: Transition rules of the abstract machine for SIMP (copied from Fernandez' book).

1. Evaluation of Expressions:

$$
\begin{array}{rcll}
\langle n \cdot c, r, m\rangle & \to & \langle c, n \cdot r, m\rangle & \\
\langle b \cdot c, r, m\rangle & \to & \langle c, b \cdot r, m\rangle & \\
 & & & \\
\langle \neg B \cdot c, r, m\rangle & \to & \langle B \cdot \neg \cdot c, r, m\rangle & \\
\langle (B_1 \ \wedge \ B_2) \cdot c, r, m\rangle & \to & \langle B_1 \cdot B_2 \cdot \wedge \cdot c, r, m\rangle & \\
\langle \neg \cdot c, b \cdot r, m\rangle & \to & \langle c, b' \cdot r, m\rangle & \text{if } b' = not\ b \\
\langle \wedge \cdot c, b_2 \cdot b_1 \cdot r, m\rangle & \to & \langle c, b \cdot r, m\rangle & \text{if } b_1\ and\ b_2 = b \\
 & & & \\
\langle (E_1\ op\ E_2) \cdot c, r, m\rangle & \to & \langle E_1 \cdot E_2 \cdot op \cdot c, r, m\rangle & \\
\langle (E_1\ bop\ E_2) \cdot c, r, m\rangle & \to & \langle E_1 \cdot E_2 \cdot bop \cdot c, r, m\rangle & \\
\langle op \cdot c, n_2 \cdot n_1 \cdot r, m\rangle & \to & \langle c, n \cdot r, m\rangle & \text{if } n_1\ op\ n_2 = n \\
\langle bop \cdot c, n_2 \cdot n_1 \cdot r, m\rangle & \to & \langle c, b \cdot r, m\rangle & \text{if } n_1\ bop\ n_2 = b \\
 & & & \\
\langle !l \cdot c, r, m\rangle & \to & \langle c, n \cdot r, m\rangle & \text{if } m(l) = n
\end{array}
$$

2. Evaluation of Commands:

$$
\begin{array}{rcl}
\langle skip \cdot c, r, m\rangle & \to & \langle c, r, m\rangle \\
 & & \\
\langle (l := E) \cdot c, r, m\rangle & \to & \langle E \cdot := \cdot c, l \cdot r, m\rangle \\
\langle := \cdot c, n \cdot l \cdot r, m\rangle & \to & \langle c, r, m[l \mapsto n]\rangle \\
 & & \\
\langle (C_1; C_2) \cdot c, r, m\rangle & \to & \langle C_1 \cdot C_2 \cdot c, r, m\rangle \\
 & & \\
\langle (if\ B\ then\ C_1\ else\ C_2) \cdot c, r, m\rangle & \to & \langle B \cdot if \cdot c, C_1 \cdot C_2 \cdot r, m\rangle \\
\langle if \cdot c, True \cdot C_1 \cdot C_2 \cdot r, m\rangle & \to & \langle C_1 \cdot c, r, m\rangle \\
\langle if \cdot c, False \cdot C_1 \cdot C_2 \cdot r, m\rangle & \to & \langle C_2 \cdot c, r, m\rangle \\
 & & \\
\langle (while\ B\ do\ C) \cdot c, r, m\rangle & \to & \langle B \cdot while \cdot c, B \cdot C \cdot r, m\rangle \\
\langle while \cdot c, True \cdot B \cdot C \cdot r, m\rangle & \to & \langle C \cdot (while\ B\ do\ C) \cdot c, r, m\rangle \\
\langle while \cdot c, False \cdot B \cdot C \cdot r, m\rangle & \to & \langle c, r, m\rangle
\end{array}
$$