1 Intermediate Representation

(Prepared by Sonu Mehta)
Here begins the code optimization part. The compiler pipe line begins with source code which is unoptimized, having specifications of the program logic by human programmer.On the other end of the pipeline, we have assembly code which actually gets executed on machine. The question we want to address here is: where to do the optimization? Here are three different options along with their pros and cons:

- **Optimize at source code level** : For e.g, we can run optimization on the abstract syntax tree(AST) of the programming language. But then, we will have to deal with a lot of features which are very specific to a programming language. For e.g., C has *for* and *while* loops but optimization does not care whether loops are written using *for* or *while*, it just needs to know that there is a loop. Also, sometimes we need information about the architecture to do architecture specific optimizations but if we do the optimization at source code level, we do not have visibility into the architecture.

- **Optimize at assembly language**: This level has much less features, usually nothing specific to any programming language. But here the optimization space is very limited because most of the decisions are already taken by the time we reach the assembly code.

- **Optimize the intermediate representation(IR)** : Most compilers have intermediate representation(IR). The AST is translated to IR and then optimized IR is converted to assembly code. What is a good intermediate representation? The IR should be simple enough, should not have unnecessary program features(e.g., *for* vs *while* loop differentiation). Also we want to make minimum commitments when designing IR so that we have large optimization space. Essentially there is a tradeoff in designing the IR:
  i) IR too close to assembly - there is smaller optimization space
  ii) IR too close to source - loose architecture specification optimizations.

More Software Engineering advantages of IR:
We can have n number of source code languages on the source code side and similarly, we can have m number of instruction set architectures(ISA) like x86 etc on the architecture side. If we write one IR that can serve both different source code languages and architectures, then we just need to write optimizers once for that IR. We won't have to write different optimizers for each language or each ISA. But then there is one more issue here: The types of constructs needed to support C(lower level language than python) program are different from the type of constructs needed to support python program. IR design can be considered more art than science and is constantly evolving.

In reality, its difficult to design one IR, so there are multiple IRs -(in the range of 3-5). Each IR has its own level of abstraction. Each level will have different optimizations, because different types of optimizations require different abstractions. This can be explained with an example: Consider *Tensorflow* :
1. In *Tensorflow*, source program is represented as graph.
2. Nodes are operators. Edges represent data flow.
3. Datatypes are tensors(multi-Dimensional arrays).

Some example optimizations are:
1. **Matrix level rewrites** : $A * A^{-1} = I$. (For this, we need to know that A is a matrix and $*$ is

multiplication operation). Such operations are easier closer to the source code because by the time we reach assembly code, we lose track of this high level information.

2. **Parallelization operation**: This needs dependency information which is in the form of matrices, so better to do this optimization at higher level than lower level.

3.**Locality** : There is fixed size architecture, we need to organize data so that the cache is maximum utilized. Optimizations like these are done in higher level IRs.

4. **Register Allocation** : These need to be done at lower level of abstractions, so these are done at lower level IRs.

2 Three Address Code

2.1 IR for Assembly Code Generation

This is low level IR and is meant for assembly code generation. Typically this IR is present in compiler tool chains of almost all languages. IR resembles high-level assembly. Some properties for such IR are:

- They have register names similar to assembly language. Unlike assembly code, they have unlimited numbers of registers.
- They have control structures like conditional and unconditional branches(similar to assembly). They are lower level than if-else(in C) etc.
- They have opcodes. They have higher level opcodes, something like doing vector addition. These opcodes are very similar to the ones in assembly which make their conversion to assembly opcodes easier. Sometimes IR opcodes could be lower level as well. For eg., some assembly opcodes can do multiple operations (*add*, *multiply*) in one instruction which might not be the case with the IR opcode. These need to be kept into mind while converting from IR to assembly code. An opcode is higher level if its translation to assembly code results in more than 1 instruction. If more than one opcodes translate to one instruction in assembly code, then it is considered lower level opcode.

2.2 IR Characteristics

Each instruction is of the form

$x = y$ op z (binary operation)

or

$x =$ op y (unary operation)

where y and z are registers or constants, can be scalars or vectors. This type of IR is called *Three-Address Code*. This is because, for each operation, there are at max 3 addresses (x, y, z). Addresses are registers or memory addresses.

2.3 Three Address Code Property

Expression $x + y * z$ translates to

$t1 = y * z$

$t2 = x + t1$

Property : Each sub-expression has a name which makes compilation easier later on from the perspective of optimization. Assembly operations such as *load* and *store* are also supported.

2.4 IR Code Generation

This is very similar to assembly code generation.

Define function $igen(e, t)$ as a function which

(INPUT) : takes an expression e and a value t. t represents the register name in which the value of expression should be computed.

(OUTPUT) : $igen(e, t)$ generates code to compute the value of expression e in register t. (Currently assuming that there are unlimited number of registers)

2.4.1 IR CodeGen Example

$$
\begin{aligned}
igen(e_1 + e2, t) = \\
& igen(e_1, t_1) \\
& igen(e_2, t_2) \\
& t = t_1 + t_2
\end{aligned}
$$

t_1 and t_2 are fresh registers.

2.4.2 Three Address Code IR

LLVM (Low level Virtual Machine) is the one of most popular *Three Address Code* IRs.

<u>**Types** in LLVM are</u> :
(integer) $i1$, $i8$, $i32$.. ($i1$ stands for 1-bit integer and so on..)
(float)
(pointers) i32*, i32**..
(vector types)$\langle 4 * i32 \rangle$
(struct types) $\{i32, i16\}$

The size of the integer is not consequential to optimization, so we make this decision by the time we reach IR.

<u>**Risc-like opcodes**</u> : *LLVM* has opcodes like add, load, store, call, ret, branch, conditional branch... Unlike assembly code, these opcodes can have unbounded number of arguments.

3 Optimization Overview

Most intermediate representations are organised as *control flow graphs (CFG)* over *basic blocks*.

3.1 Basic Blocks

A basic block is a single entry, single exit, straight line code segment. More formally, it is a maximal sequence of instructions with no labels (except at the first instruction) and no jumps (except at the last instruction).

In the following figure, (a) is a basic block. (b) is not a basic block because it has multiple exits; (c) is not a basic block because it has multiple entries; (d) is not a basic block because it does not represent a straight line code.

1.png

Figure 1: Examples and Counterexamples for basic block

Consider the following example of a single basic block:

1. L:
2. $t := 2 * x$
3. $w := t + x$
4. if w > 3 goto L'

- Is it ok to change (3) to $w := 3 * x$? It is ok. If addition of two numbers is more expensive as compared to multiplication of a number with a constant, the above change can be considered an optimization.
- Is it ok to change (4) to if x > 1 goto L' ? It is not ok. Because here x is a finite bounded integer and due to overflow, it is possible to have $3 * x > 3$ and not x > 1.
- Is it ok to remove (2) ? Depends on whether variable t is used later in the code or not.

3.2 Control Flow Graph

A control flow graph is a directed graph with

- basic blocks as nodes
- edge from block A to block B if the execution can pass from the last instruction in A to the first instruction in B, for example
  - the last instruction in A is: jump Lb
  - the last instruction in A is: if id1 = id2 then goto Lb
  - execution can fall-through from block A to block B

Consider the following example of a CFG:

`images/CFG Example.png`

Figure 2: CFG Example

The body of a method (or procedure) can be represented as a control-flow graph. There is one initial node (entry node). All ”return” nodes are terminal.

3.3 Optimization

The optimizations are performed on the control flow graph of the intermediate representation of the code to improve program's resource utilization:

- Execution Time
- Code Size
- Memory Usage
- Frequency of Disk I/O operations (or network operations)
- Power Consumption (Not same as energy)

It is important to remember that optimization should not alter the meaning of the program. It should not alter what the program computes.

For example, the following optimization changes the meaning of the program and thus, it is not a valid optimization.

w := 3 * x if w > 3 goto L

cannot be converted to: if x > 1 goto L

3.4 Typical Granularity of Optimization

1. **Local Optimizations**
   - applied to a basic block in isolation
   - easiest to implement
2. **Global Optimizations**
   - applied to a CFG (method body) in isolation while crossing the boundaries of basic blocks
3. **Inter-Procedural Optimizations**
   - applied across method (CFG) boundaries
   - difficult to implement but usually most effective

3.5 Economics of the Optimization

Optimizations are more of an art rather than science. The current state of the art methods are based on the concept of "Maximum benefit for minimum cost" where cost can denote the development and integration costs of the optimization.

- Some optimizations are hard to implement
- Some optimizations require large compilation time
- Some optimizations have low payoff (the benefits) and it is often difficult to quantify payoff.

4 Global Optimizations

(Prepared by Aditya Senthilnathan)

Global optimisations go beyond the scope of a single basic block. It needs reasoning of the whole control flow graph which may be within the body of a method.

4.1 Global Constant Propagation

- This is similar to local constant propagation but now we are reasoning in a global level (across several basic blocks within the body of a method)
- In this optimisation, we basically ask the question "When can we replace the use of a variable **x** with a constant **k**

  Ans: If on **every path** to the use of x, if the **last assignment** to x is **x:=k**, then we can do this transformation/optimisation.

The algorithm for global constant propagation requires a class of algorithms which are called "Global Dataflow Analysis". This kind of reasoning (with paths, etc.) is required for many different types of optimisation problems in compilers. And so, it helps to create a common framework in which we can implement the same logic but with different kinds of properties so that there is reuse of the same idea.
Let's identify some traits that global optimisations tasks share:

- <u>Optimisation depends on knowing property X at a certain program point P</u>

  Eg. In global constant propagation, we were interested in knowing if x:=k is the last assignment on all possible paths, at the program point of the last basic block (where x is used) in our example graph.
- <u>Proving X at P requires knowledge of entire program</u>

  Eg. In our global constant propagation example, to be able to say x:=k is the last assignment to x on all possible paths that can reach program point P, we need to know the characteristics of the entire program. As opposed to local optimistaion where you only need info about the concerned basic block. Here, even though transformation is only made in a basic block, it relies on a property that requires knowledge of the entire program.
- <u>It's OK to be conservative</u>

  We can either say,

  X is definitely true

  or say,

  We don't know if X is true

  We want to get "X is definitely true" as much as possible because only then can we apply the transformation but if we get the conservative answer "We don't know if X is true", then we don't apply the transformation to preserve program correctness. Knowing if X is definitely not true is useless because in this case we still don't apply the transformation and so this use case is clubbed with "We don't know if X is true".

5 DFA Values

(Prepared by Aditya Senthilnathan)
Recall that for global constant propagation, we need the following:

Optimisation: Replace a use of x by a constant k
Constraint: On every path to the use of x, the last assignment to x should be x:=k

We need to do this for every variable in the program. But first let's do this for a single variable x and then we can generalise it to more variables.

5.1 Dataflow Analysis Values

These are abstract values that are supposed to capture the set or category of concrete values that x (a variable) may take.

At each program point, we associate one of the following values with X (a variable),

Value	Explanation
$\top$ (top)	This value represents either that • This statement never executes • or we have not executed or seen it yet
$\perp$ (bottom)	This value represents either that • We can't say that X is a constant • X is definitely not a constant
C (constant)	This value represents that X is a constant C

Question: What are we trying to do here? What do these values mean?

Answer: We are trying to figure out what are the possible values of X (a variable) at every program point. The values we are talking about essentially tell us if the program reached here what is the value of X or all the possible values of X at this program point.
Now we will discuss each of the values in a little more detail,

- $\top$ (top) - This value represents that this statement never executes i.e the program/control never reaches here. This might be because this part of the program is never reachable from the beginning of the program. So in this case we just say that the value is $\top$ here because the statement never executes and therefore it is not meaningful to say what this value is.
- C (constant) - It could be any constant. Eg. 0,1,2,-2,etc. This value says that whenever this program point is reached, irrespective of the path taken, we are sure that X will be equal to constant value C at this point.

- <u>$\perp$ (bottom)</u> - This value essentially says that we don't know if X is a constant or not at this point. We are not sure of it's value. This might be happening because it's possible that on different paths X could take different values or on some path X is not a constant, etc. Even if we know for sure that X is not a constant, we still assign $\perp$ to X. This is a conservative assignment. Technically, even if we (DFA algorithm) assign $\perp$ to X at every program point conservatively, then this solution is also correct but it is trivially true and is not of much use to us.

Note: If we're still in the middle of execution of the DFA algorithm and at a particular point, we have value $\top$ for one of the variables in the program then it could also mean that the DFA algorithm hasn't seen this point yet or hasn't reached this point yet. The above discussion and interpretation of $\top$ only applies if a variable has $\top$ value after the DFA algorithm has finished executing

5.2 Using DFA values for transformation

Given global constant information ($\top$, $\perp$, C)

- Inspect x = ? (either $\top$ or $\perp$ or C) at the program point that just precedes the statement which you want to examine (the statement that uses x)
- If x = C, then replace the use of x with C

For some examples of how to assign DFA values see the lecture video

6 Dataflow Analysis Algorithm (DFA)

(Prepared by Aditya Senthilnathan)
In this section, we will now try to answer the question "How do we identify what the DFA value should be at every program point?"
We will only analyse one particular variable x for now. This can easily be generalised for arbitrary number of variables and this is left to the reader.

6.1 Insight

DFA of a complex and large program can be expressed as a combination of simple rules relating the change in information between adjacent statements.

We're just going to see

- What the value of x is just before this statement
- What is the current statement
- Based on the current statement, we then use rules (to be discussed) to compute what should be the value after the statement

One way to think about this is that we are "pushing" or "transferring" information from one statement to the next. It's almost just like program execution with the caveat that we are not dealing with concrete values anymore but rather abstract DFA values.

6.2 Function for "Constant Information"

We now define a function $C(x, s, in/out)$ where,

- x - variable for which we want to compute abstract values
- s - statement s in the program
- in/out - whether it is input to the statement or output to the statement i.e just before the statement or just after the statement.
  - $C(x, s, in)$ = Value of x just before s
  - $C(x, s, out)$ = Value of x just after s i.e just after statement s is executed

The idea is to calculate the value of this function $C(x, s, in/out)$ for every x and every s (both in and out)

6.3 Transfer Function

We need to define how information is transferred from one statement to another i.e how are we pushing information. For this we need to define a transfer function.

Transfer Function: transfers information from one statement to another

We are going to define rules for the following two cases:

- **Case 1:** There are one or more predecessor points with edges to the current program point. In this case, we will see how to combine information/values flowing in from the predecessor points to assign a value for the $C()$ function in the current program point
- **Case 2:** For a given statement s, we see how the information/value is modified as it flows through the statement

6.3.1 Transfer function for Global Constant Propagation

We will define the function by doing an exhaustive case analysis. We can do this kind of finite description of the function with an exhaustive case analysis because there are only a limited number of values and cases to consider.

$C(x, s, p)$ can take only one of 3 different unique values during and after execution of the DFA algorithm i.e $\top, \bot, C$. Also once the algorithm picks a constant value for x at a particular program point, the constant value is fixed and it cannot vary after that. Eg. If the DFA algorithm picks 3, it cannot assign 4 later on. It can only change to either $\top$ or $\bot$ later on in the execution.

Rules for the Transfer function:

1. If $C(x, p, out) = \bot$ then $\forall s \in$ successors(p), $C(x, s, in) = \bot$
2. If $C(x, s, in) = C$ and s does not modify x then, $C(x, s, out) = C$
3. If $C(p_i, x, out) = \bot$ for any i, then $C(x, s, in) = \bot$

4. If $C(p_i, x, out) = c$ and $C(p_j, x, out) = d$ and $c \neq d, i \neq j$ for some i, j then, $C(s, x, in) = \bot$
5. If $C(p_i, x, out) = c$ or $C(p_i, x, out) = \top$ for all i, then $C(s, x, in) = c$
6. If $C(p, x_i, out) = \top$ for all i, then $C(s, x, in) = \top$
7. If $C(s, x, in) = \top$, then $C(s, x, out) = \top$
8. If rule 7 does not apply i.e $C(s, x, in) \neq \top$, then $C(x := c, x, out) = c$ [where c is a constant]
9. If rule 7 does not apply i.e $C(s, x, in) \neq \top$, then $C(x := f(...), x, out) = \bot$
10. $C(y := ..., x, out) = C(y := ..., x, in)$ where $y \neq x$

We now have 10 rules relating $C(s, x, in/out)$ to the value of this function at a predecessor point. These rules are exhaustive.

But so far we have just stated the rules. How do we get the actual values?

Think of this as a system of equations. We are interested in solving and identifying values for this C function such that all 10 rules are satisfied.

How do we solve this system of equations?

Note that there is a trivial solution,

$$C(s, x, in/out) = \bot, \text{ for all statements s}$$

All 10 rules are satisfied by this solution but this solution is useless to us because it doesn't allow us to perform any transformations/optimizations. We're not interested in just any solution. We are interested in the "most precise" solution (for some notion of precision). For eg. Saying that $x = 4$ at some program point is more precise than just saying $x = \bot$. Similarly, saying that $x = \top$ is more precise than saying $x = \bot$ because the fact that this program point is unreachable is still useful information because now we can consider doing dead code elimination in this part of the program.

To solve this system of equations in a more precise way, we use a Fixed Point Iteration Algorithm.

Algorithm:

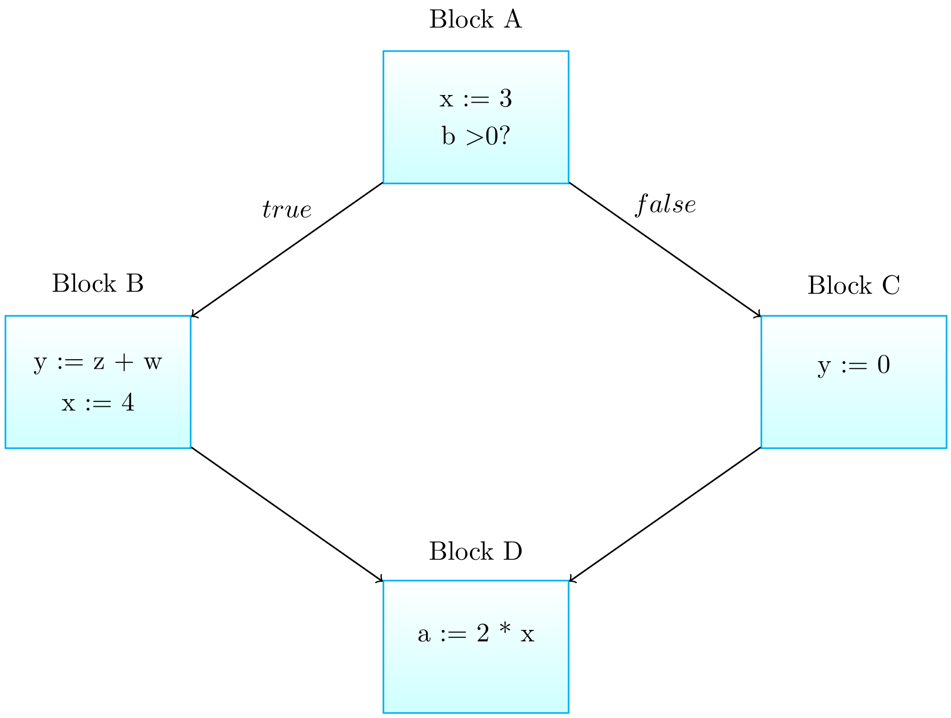
- At every entry point s to the program, set $C(s, x, in) = \bot$
- At every other statement s (which is non-entry), set $C(s, x, in) = \top$
- For every statement s, $C(s, x, out) = \top$
- Repeat the following until all program points satisfy rules 1-10:
  - Pick a statement s not satisfying one or more rules in rules 1-10 and update the corresponding $C()$ function value using the appropriate rule

There is a guarantee that this algorithm will converge. It will also give us the most precise answer. In the worst case, the solution it gives will be $\bot$ at all program points.

7 DFA Example

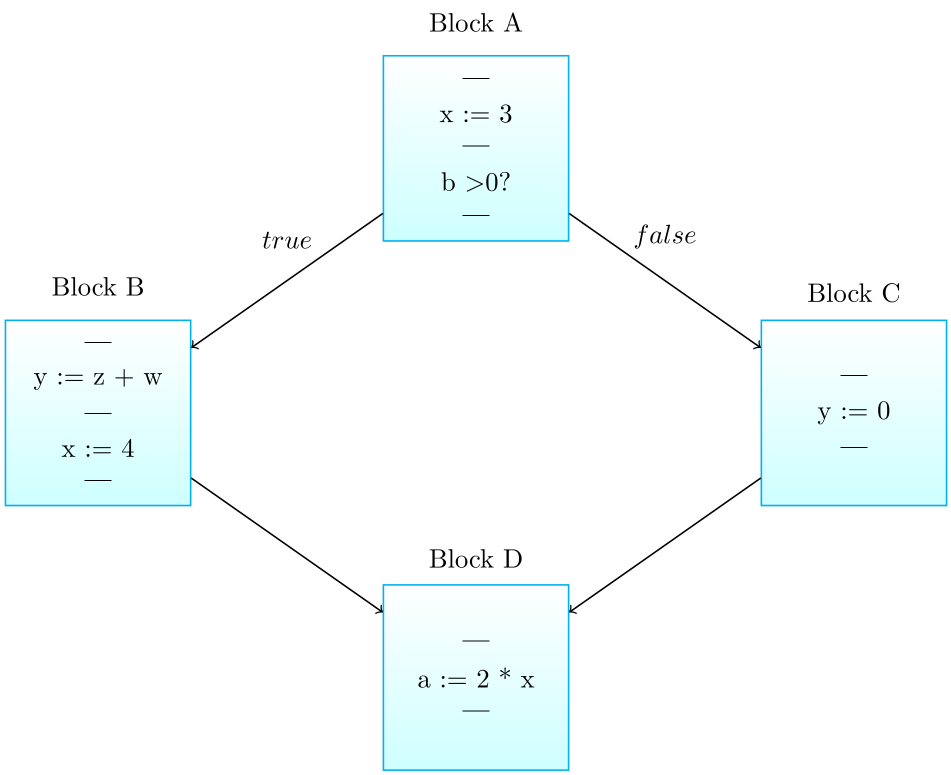
We will apply the fixed point algorithm to the variable “x” in the following piece of code.

Figure 3: Example



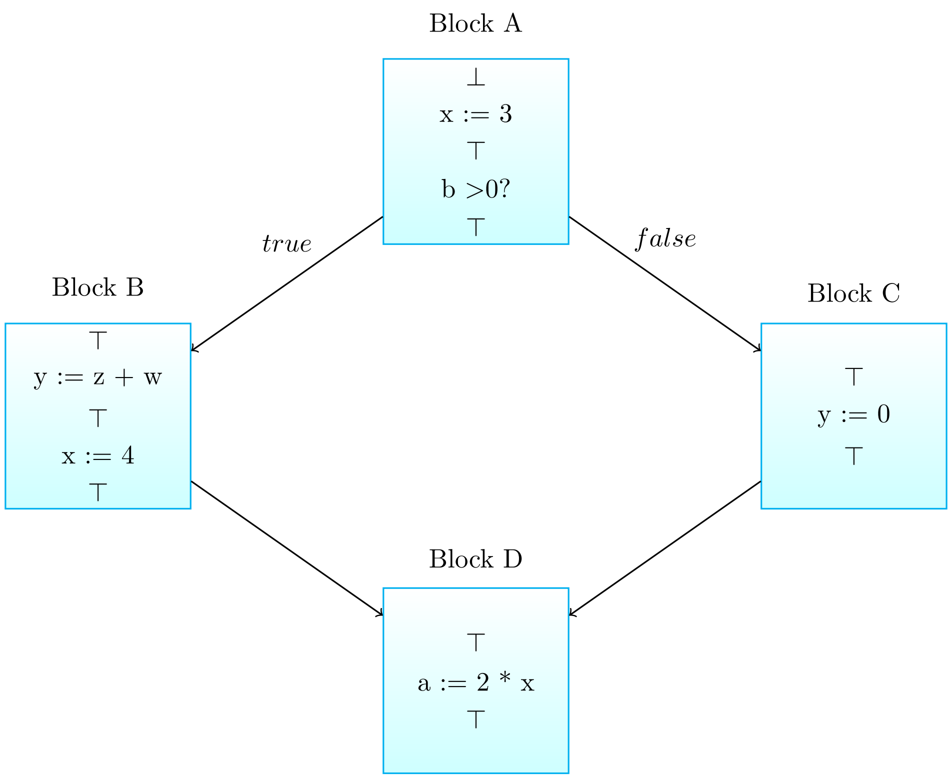
Next figure shows all the program points in our example. Program points are all the points that appear before and after every statement. In case of only one predecessor, the "out of the predecessor" is merged with the "in of the successor" while working on this piece of code. "—" marks the program points in the example we are working on.

Figure 4: Program Points



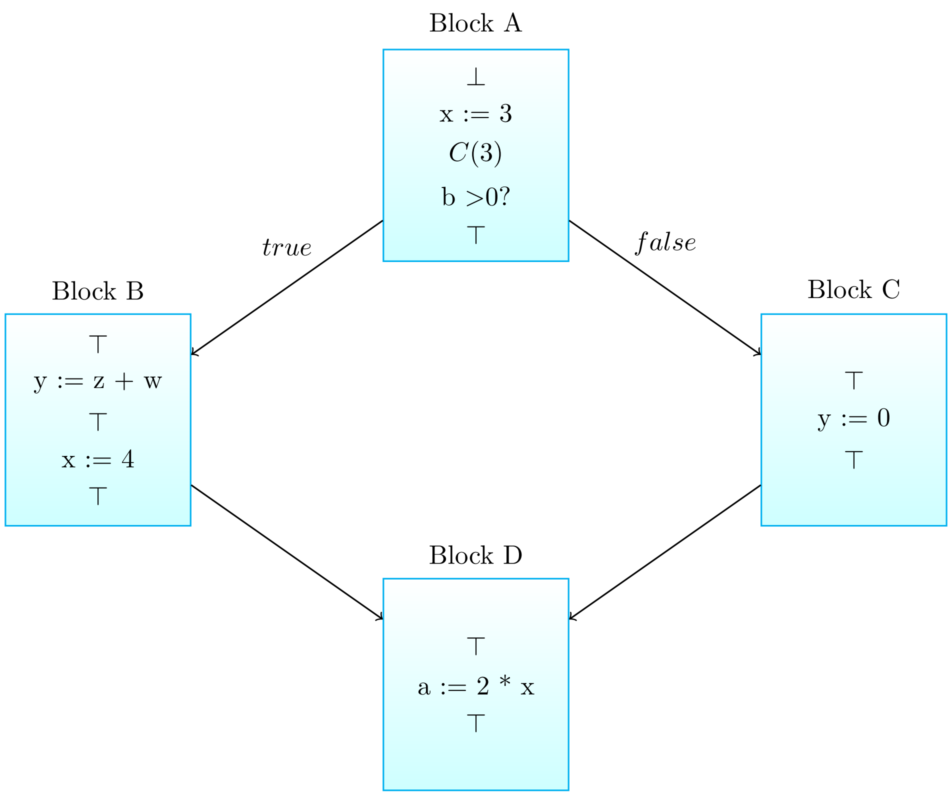
Algorithm starts by initializing every value to $\top$(top) except the "in of entry statement" which is initialized to $\perp$(bottom).

Figure 5: Initialization



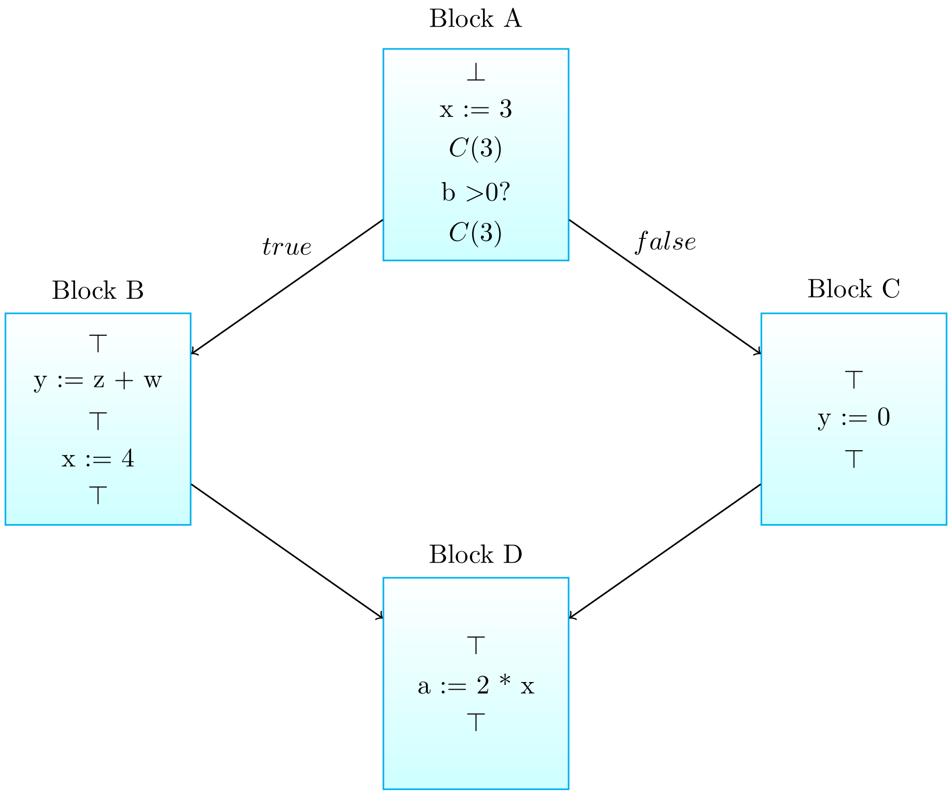
Now the algorithm will pick the statement which does not follow the transfer function rules, which is the “x := 3” statement in this example, and update it. The “out of the statement” should be Constant 3 according to the transfer function rule. After updating the “out of the statement”, we get this graph.

Figure 6: “x := 3” statement update



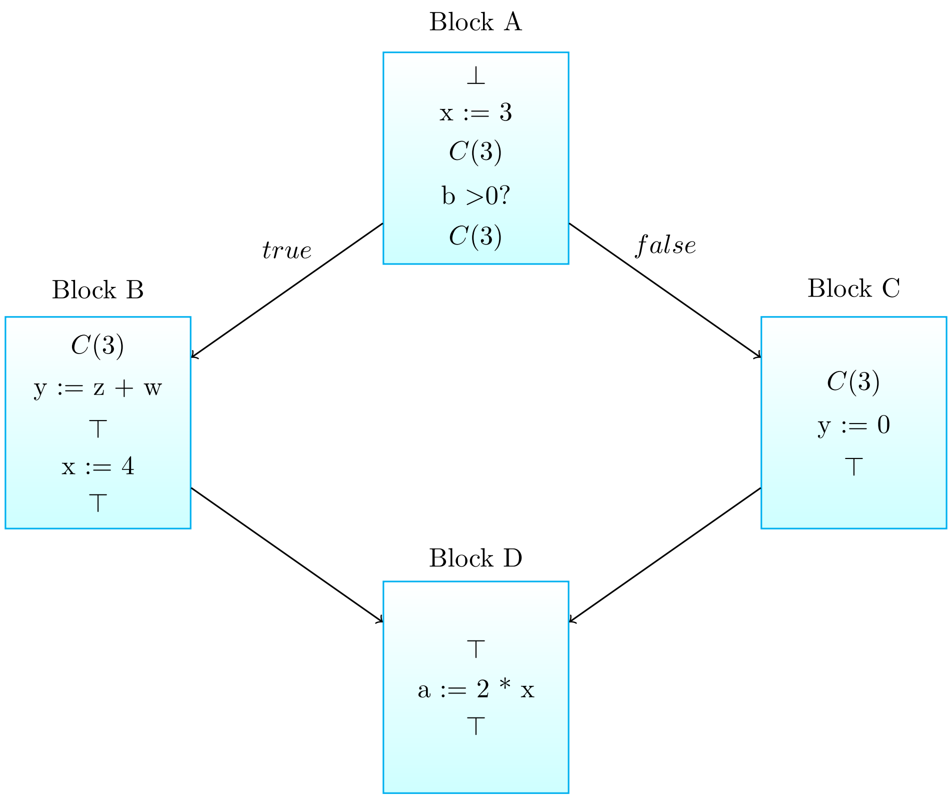
The algorithm repeats this process until all program points satisfy the transfer function rules.

Figure 7: "b >0?" statement update



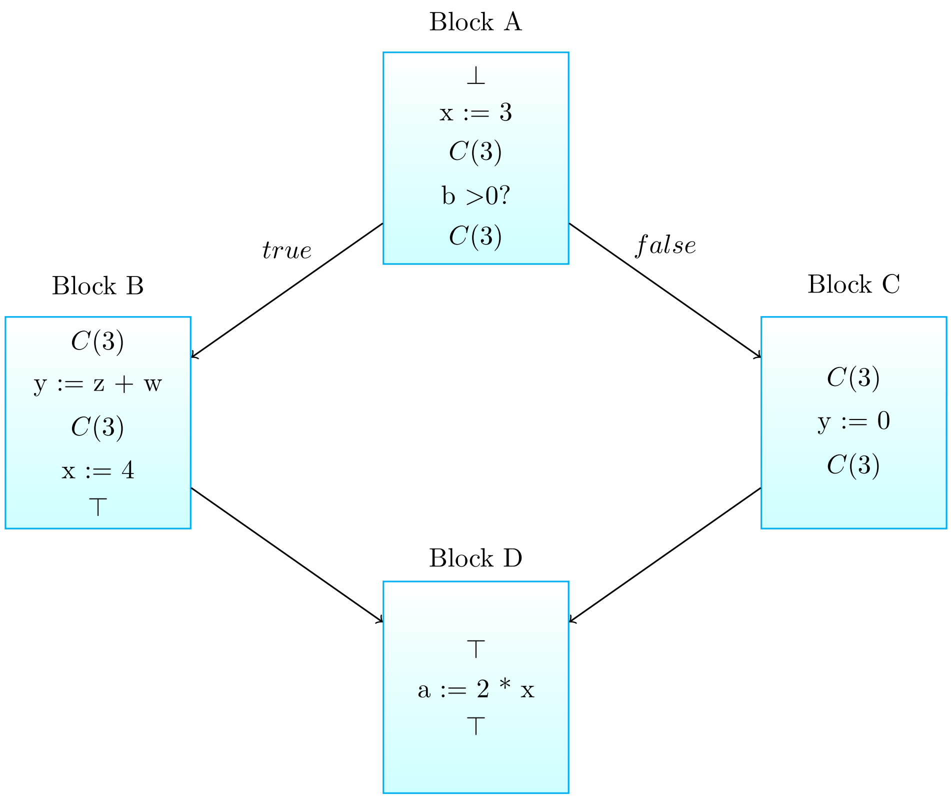
At this stage, there are two program points which do not satisfy the transfer function rules. The entry point of Block B and Block C do not match with their predecessors. We can update them.

Figure 8: "Successors of Block B and Block C" update



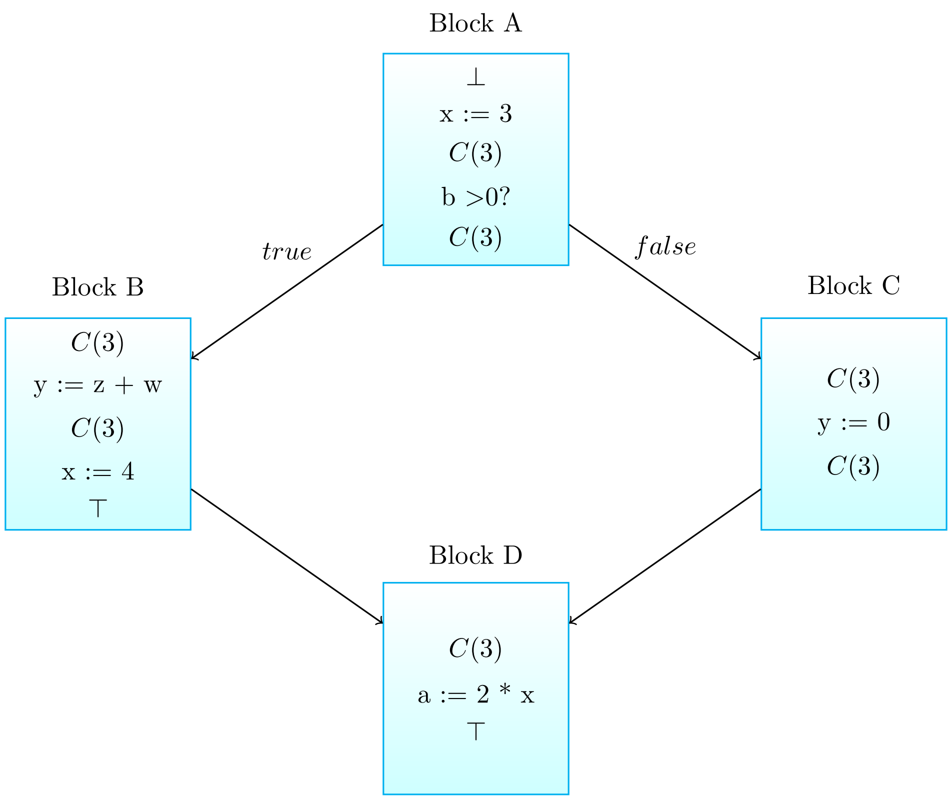
Again there are two different statements that do not satisfy the transfer function rules, we can update them.

Figure 9: "y := z + w" and "y := 0" statements update



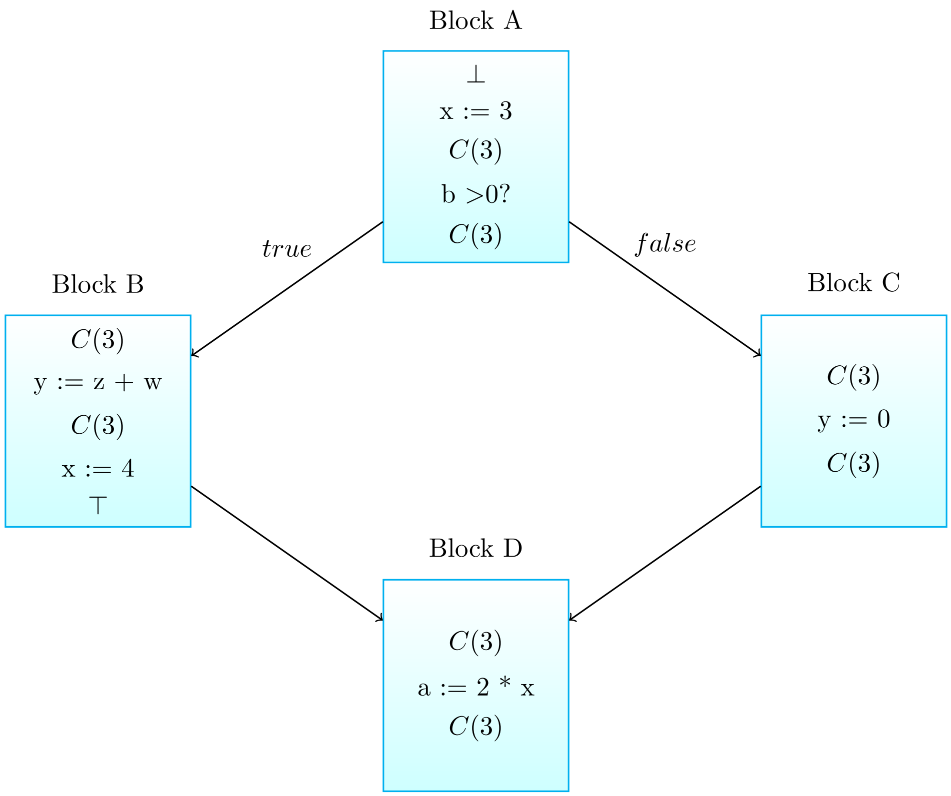
At this stage, we can either update "x := 4" statement, or we can update the successor of Block D. Irrespective of which one we pick, algorithm will return the same answer. In this example, we will update the successor of Block D. One predecessor is a constant, and other predecessor of Block D is $\top$. Using the 3rd transfer function rule, we can update the value of successor to the same constant.

Figure 10: "Successor of Block D" update



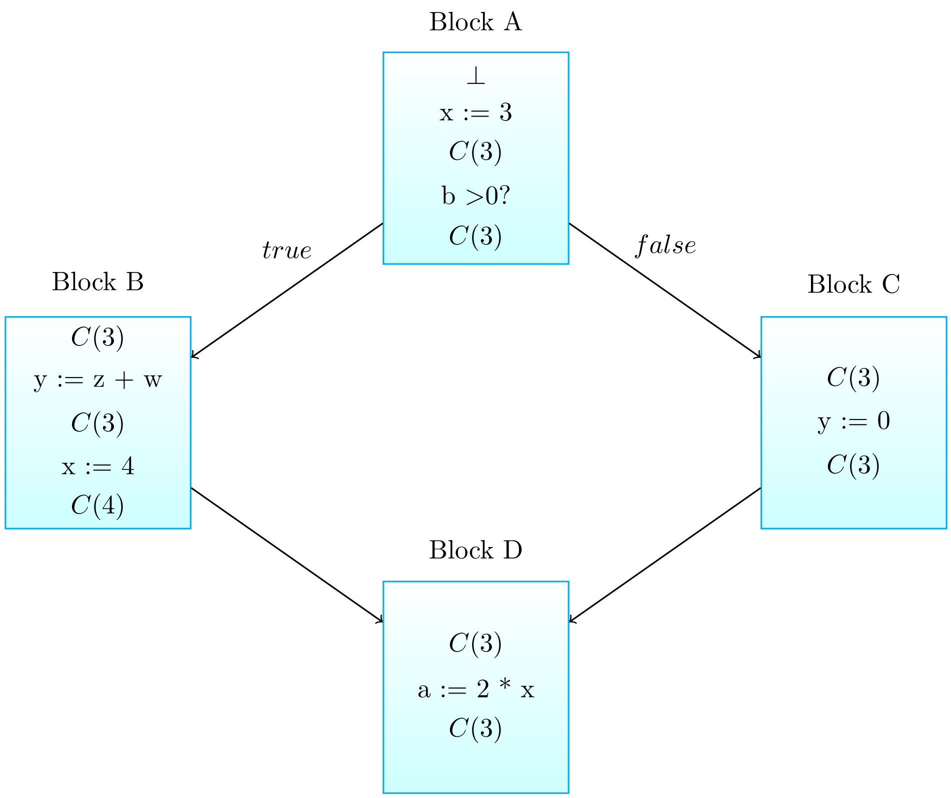
Updating “a := 2 * x” statement in Block D.

Figure 11: “a := 2 * x” update



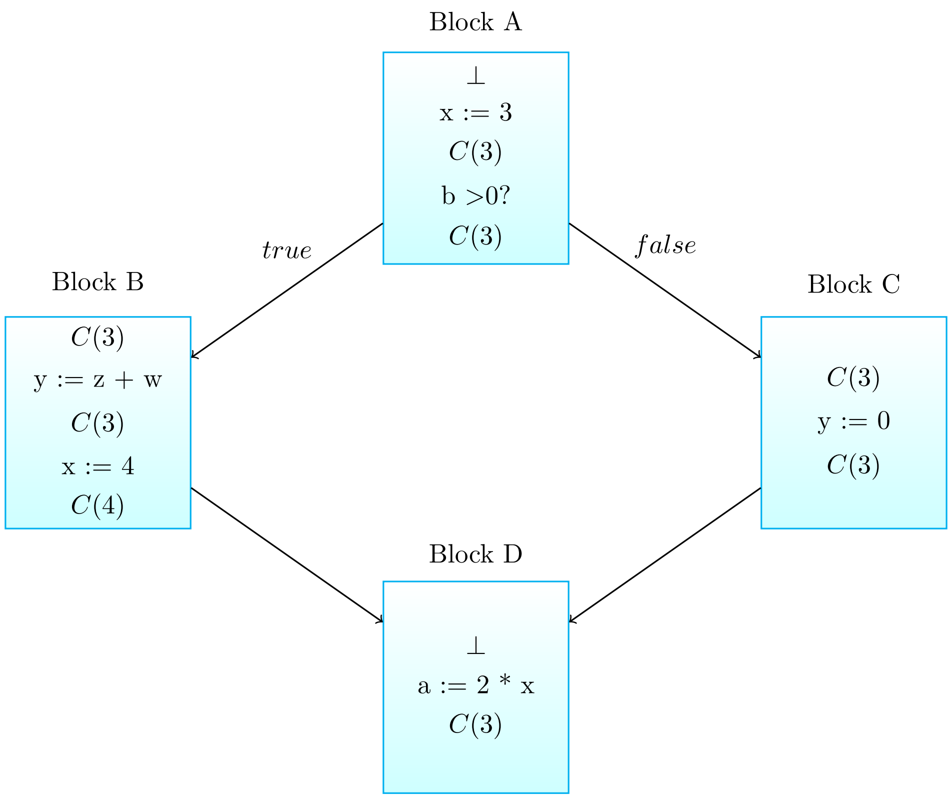
There is only one statement, “x:=4” in Block B, that does not satisfy the transfer function rules. After updating it, we get this graph.

Figure 12: “x := 4” statement update



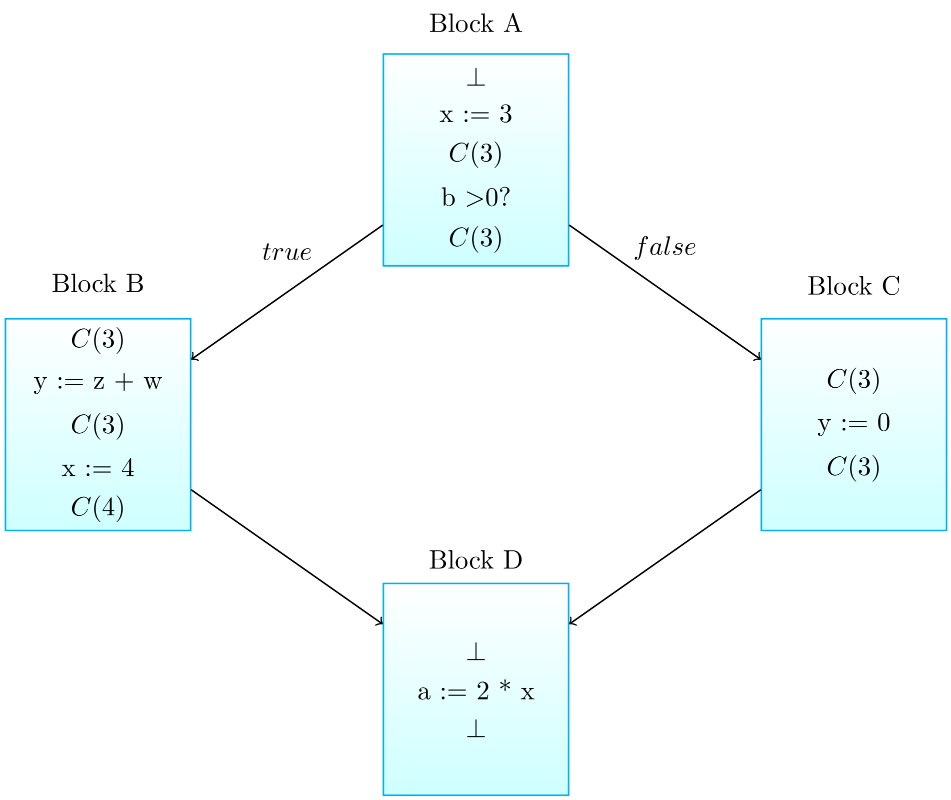
Because of the previous update, the successor of the Block D does not satisfy the transfer function rules. According to the 2nd transfer function rule, it should have the value ⊥.

Figure 13: "Successor of Block D" update



Updating the “a = 2 * x” statement in Block D, we get this graph.

Figure 14: “a := 2 * x” statement update



At this stage, there is no program point that violates any of the 8 transfer function rules, so the algorithm terminates by returning this **Fixed point solution** to us.

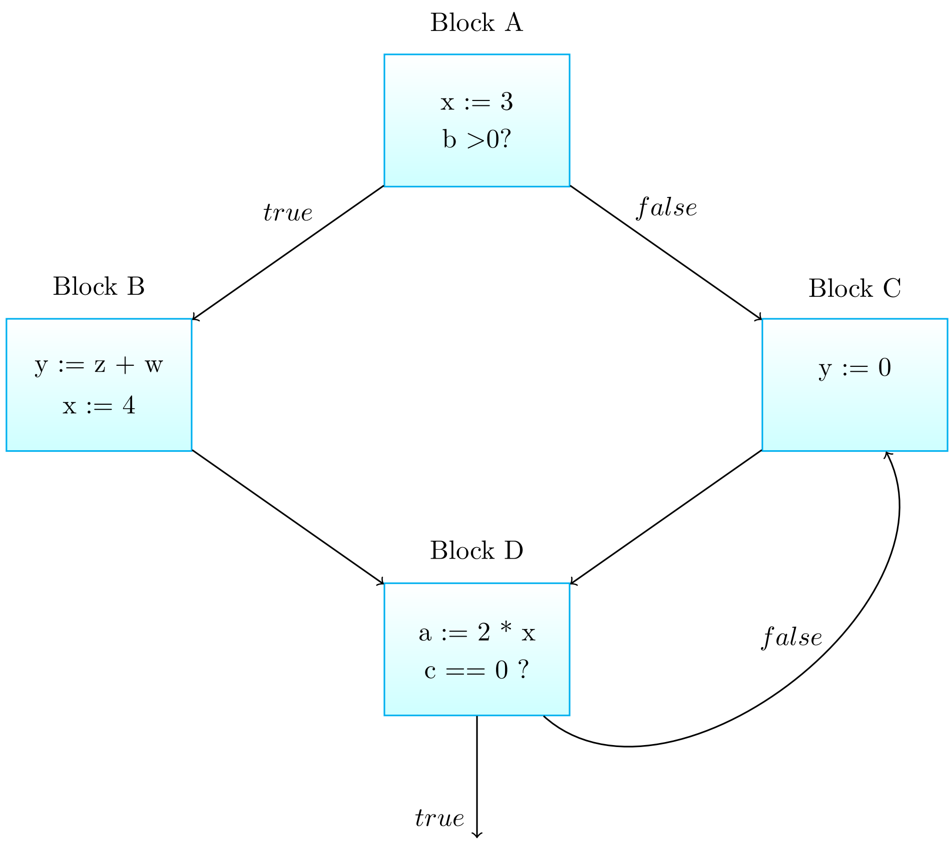
7.1 Some Guarantees

- The fixed point solution will satisfy all the transfer function rules at each program point.
- The algorithm is guaranteed to converge, irrespective of whether the program has loops or not.
- Whenever there are multiple options to choose for a statement S to be updated, irrespective of which one we choose, we always get the same answer.
- The fixed point solution is going to have some precision guarantee, by some definition of precision, for the system of equations we are trying to solve.

8 DFA Loop Example

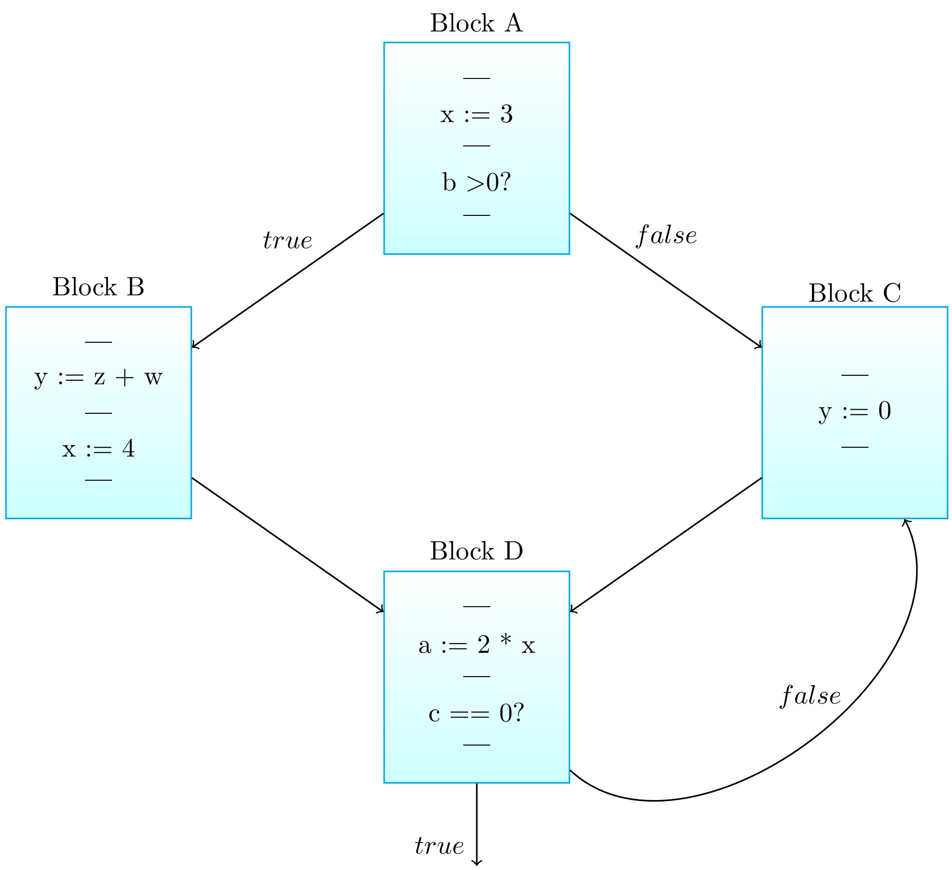
In this section, we will look at an example of DFA for a program with loops. We will apply the DFA algorithm on the variable "x" in the following piece of code.

Figure 15: Example



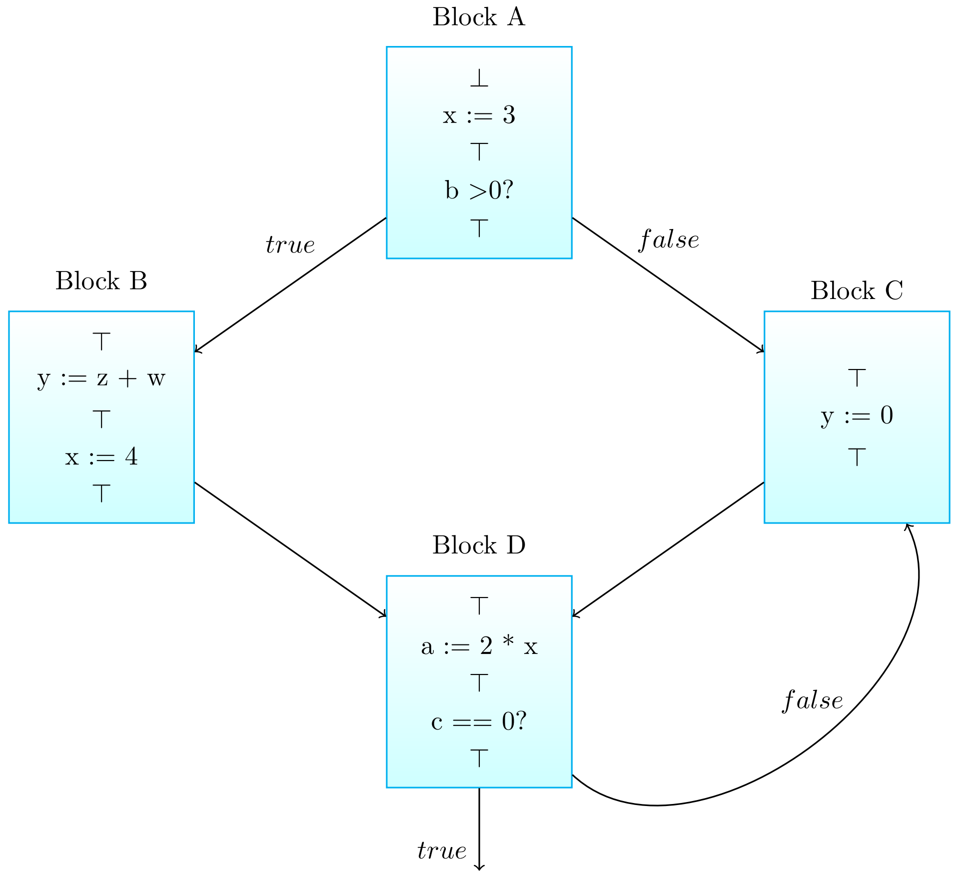
Next figure shows all the program points in our example. Program points are all the points that appear before and after every statement. In case of only one predecessor, the "out of the predecessor" is merged with the "in of the successor" while working on this piece of code. "—" marks the program points in the example we are working on.

Figure 16: Program Points



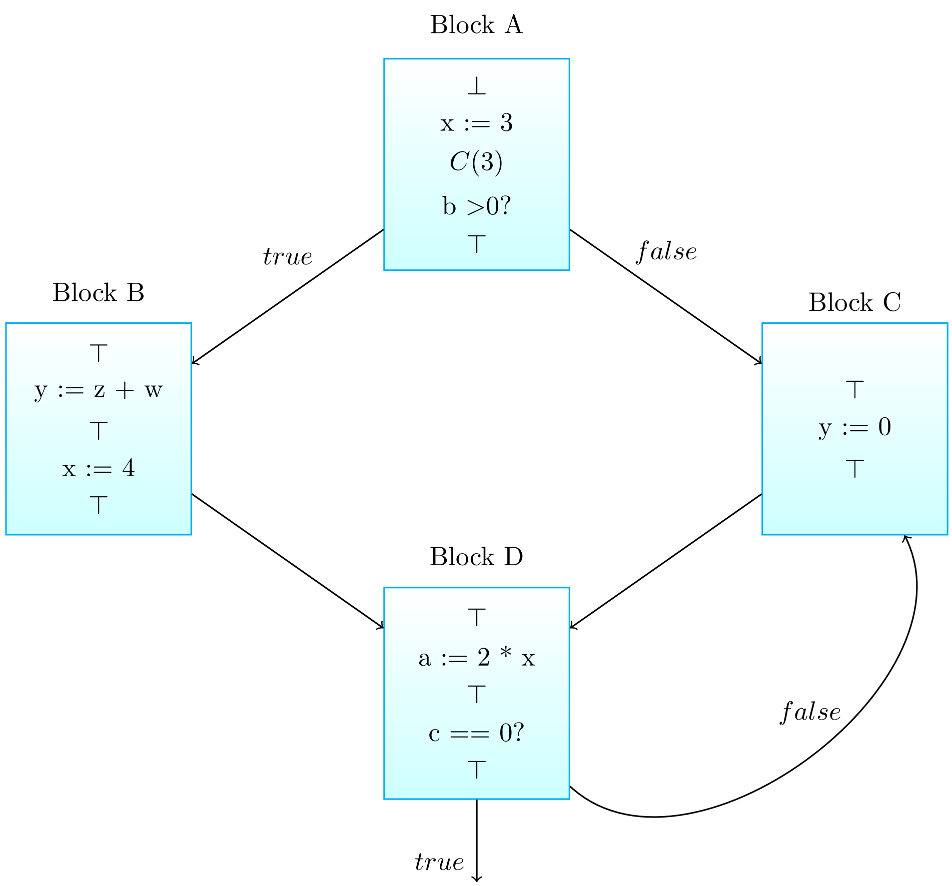
Next step is to initialize all the program points. To do that, we initialize all the program points to value $\top$ except the entry points of the program which are initialized to $\bot$

Figure 17: Initialization



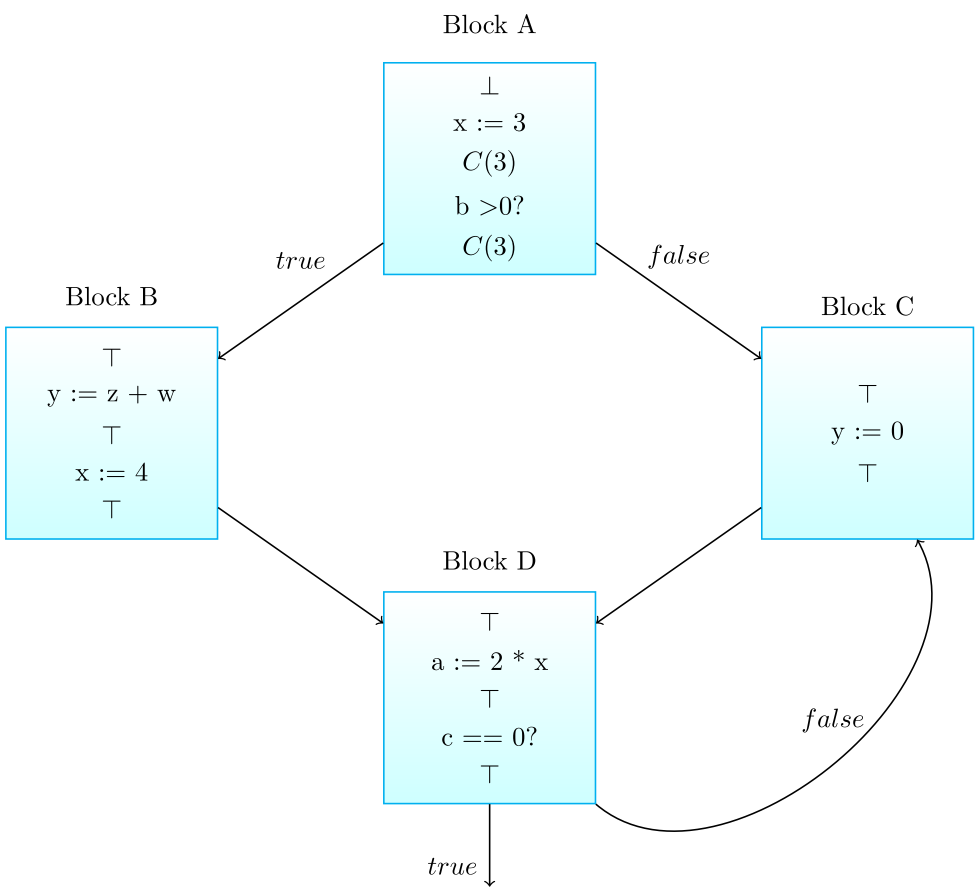
In the next step, we identify the statements which do not follow the transfer function rules and start updating them. At this stage, only the statement "x := 3" does not follow the transfer function rule, so we update it.

Figure 18: "x := 3" statement update



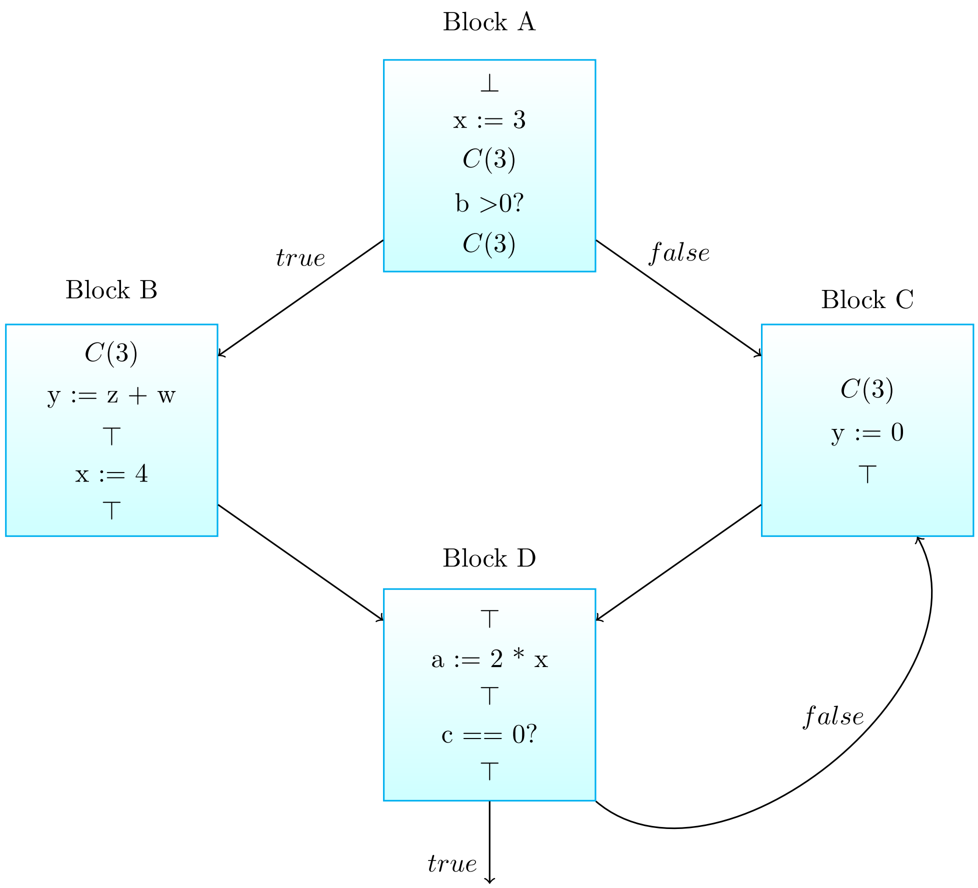
The process of identifying the statement that violates the transfer function rules and updating it, is repeated until every program point satisfies them. At this stage, the “b >0?” statement violates the transfer function rules, so we update it.

Figure 19: “b >0?” statement update



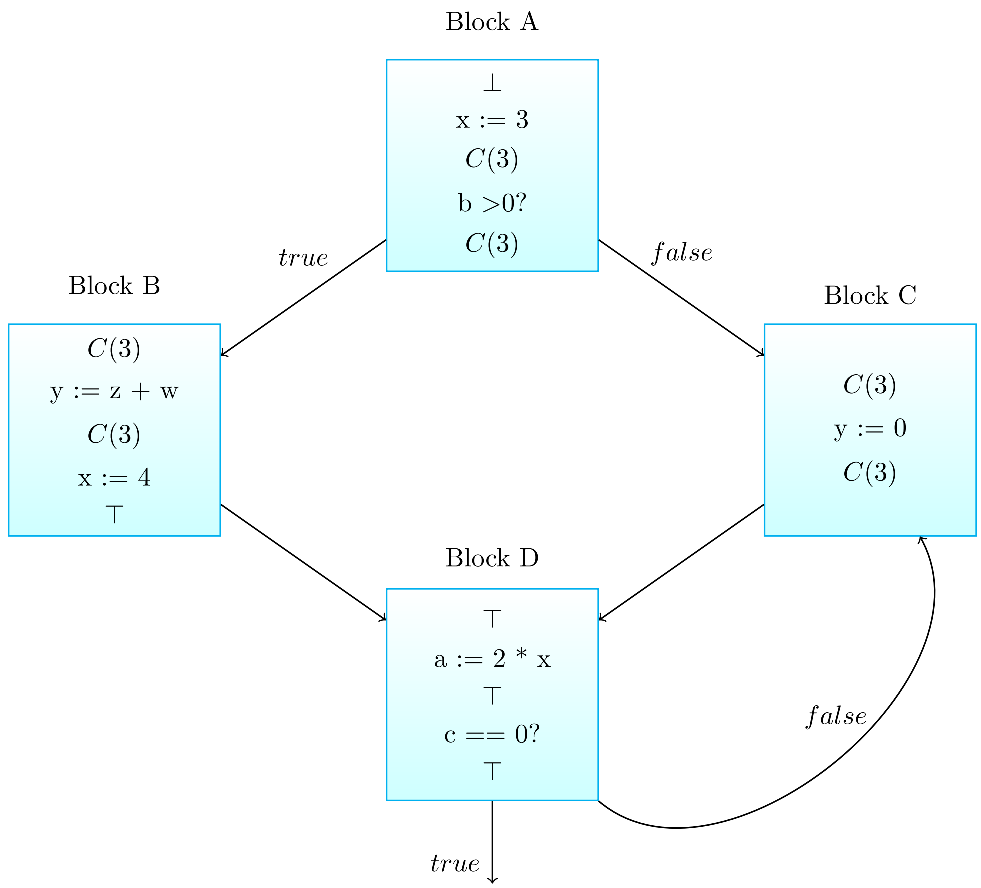
At this stage, the successors of Block B and Block C do not satisfy the transfer function rules, so we can update them.

Figure 20: "Successors of Block B and Block C" update



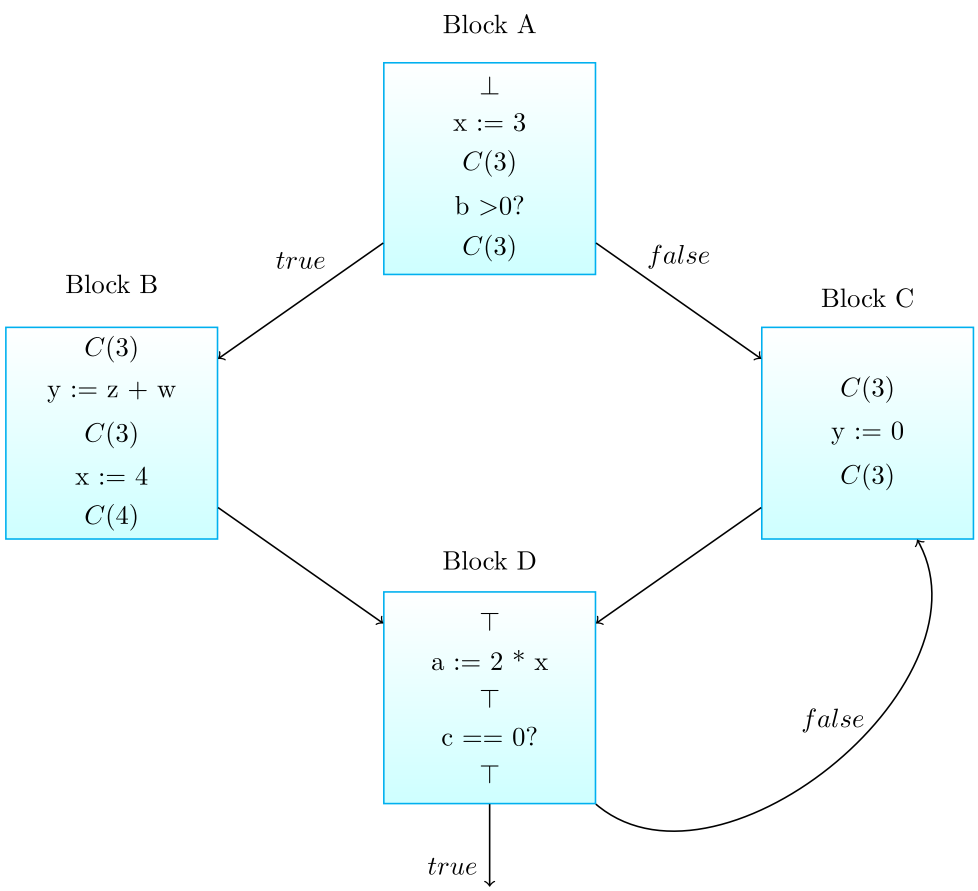
There are again two different statements "y := z + w" and "y:=0" that violate the transfer function rules, and we will update them.

Figure 21: "y := z + w" and "y := 0" statements update



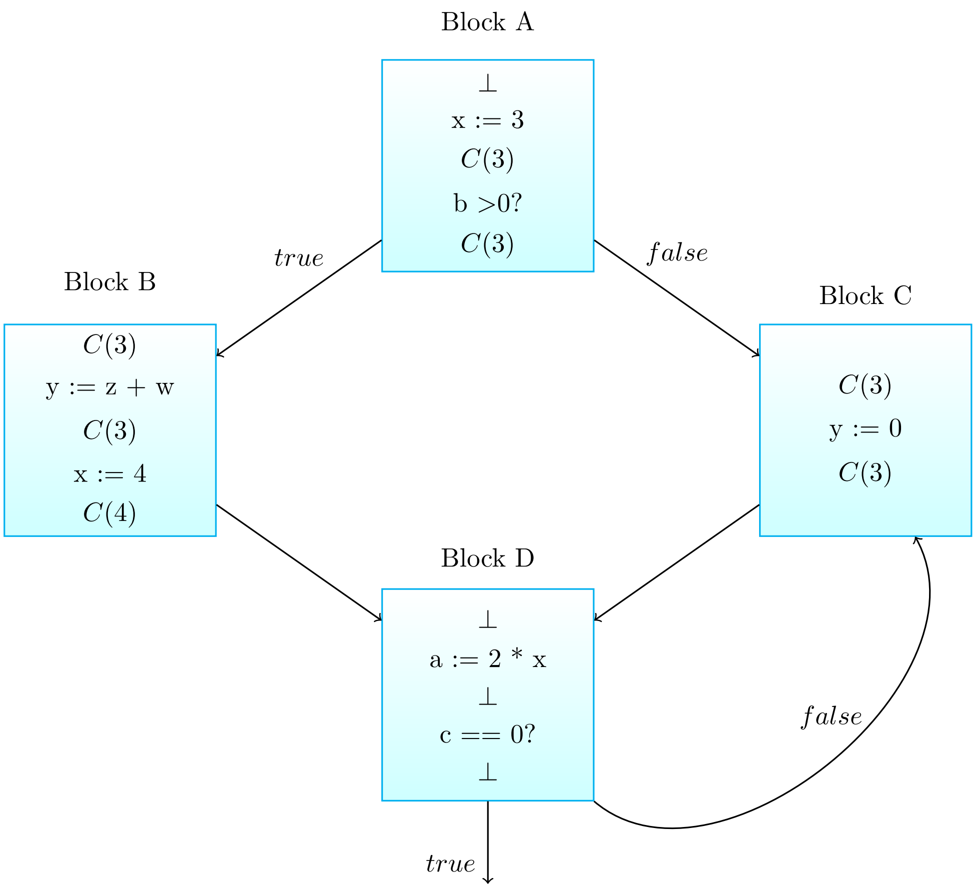
We have again two different statements that we can update at this stage, and we can choose to update anyone that we like, since the algorithm is guaranteed to return the same solution. In this example, we will update the “x := 4” statement.

Figure 22: “x := 4” statement update



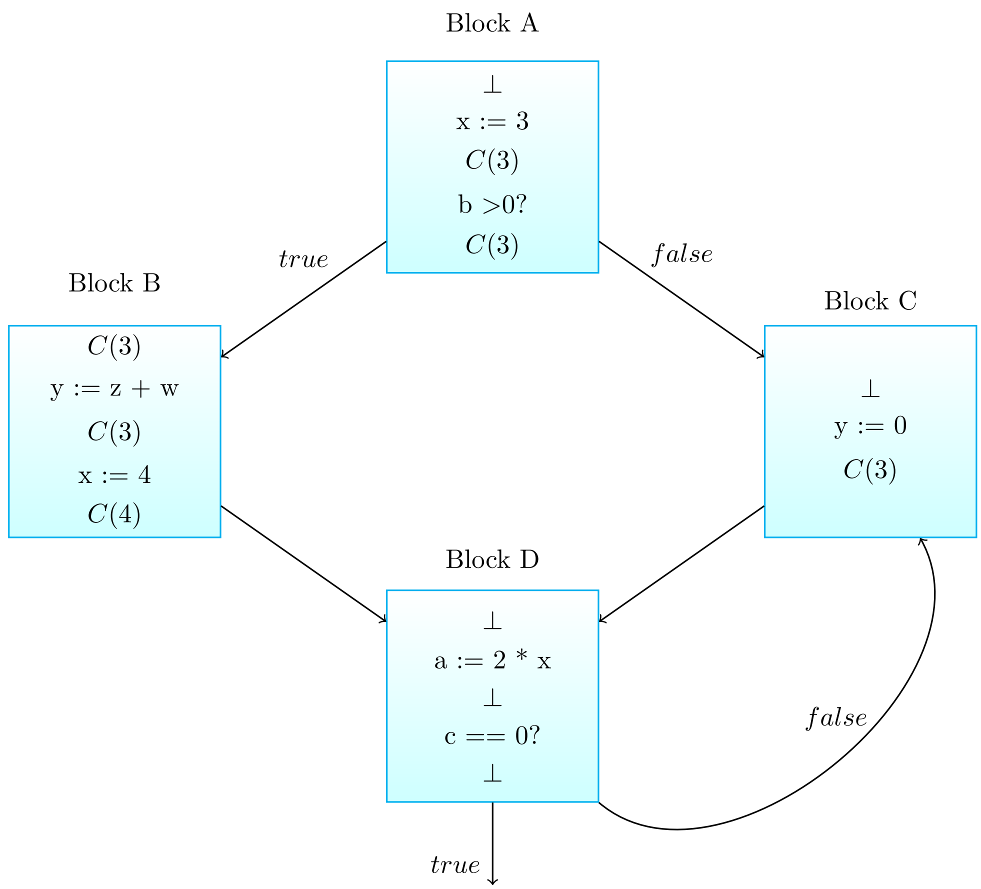
Now we will do the same thing for Block D, update the successor for Block D and then update its statements one by one. At the end, we will reach this stage.

Figure 23: “Block D” update



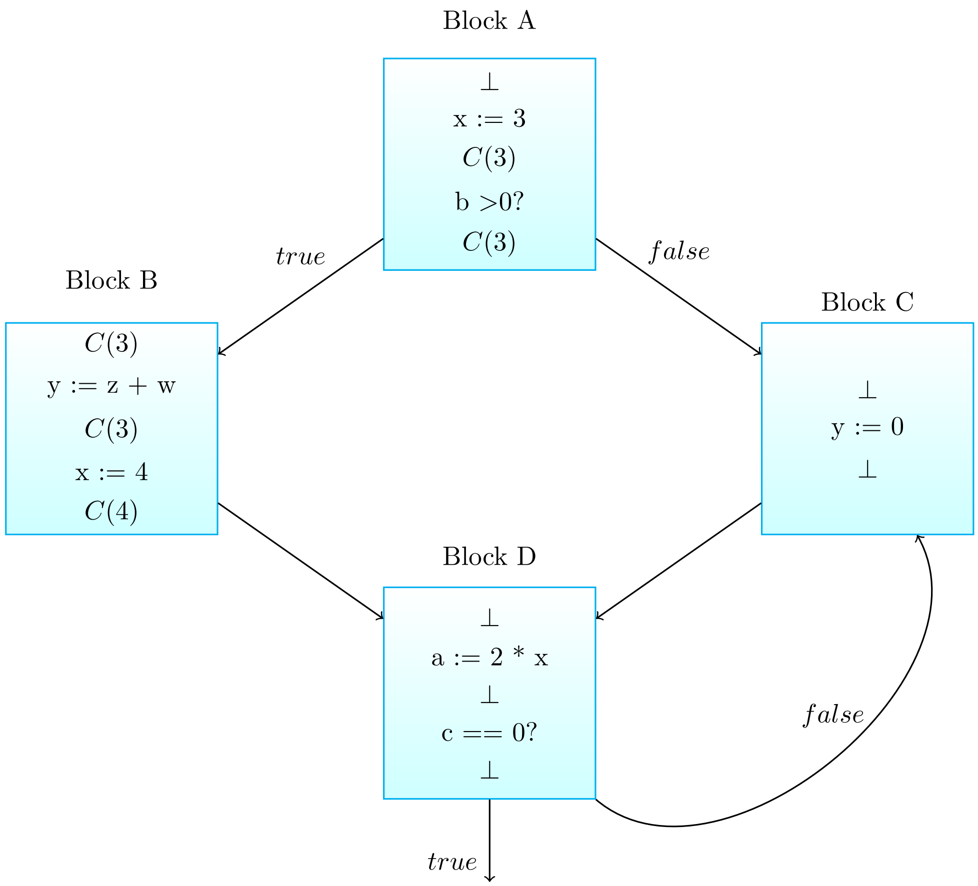
At this stage, we haven't reached the fixed point solution. The successor of Block C violates the transfer function rules. Earlier, it was acceptable for the algorithm to update the value to C(3) because the predecessor entering from Block D was $\top$, which is now updated to $\perp$. So, we will have to update this program point.

Figure 24: "Successor of Block C" update



Updating the “y := 0” statement, we reach this stage.

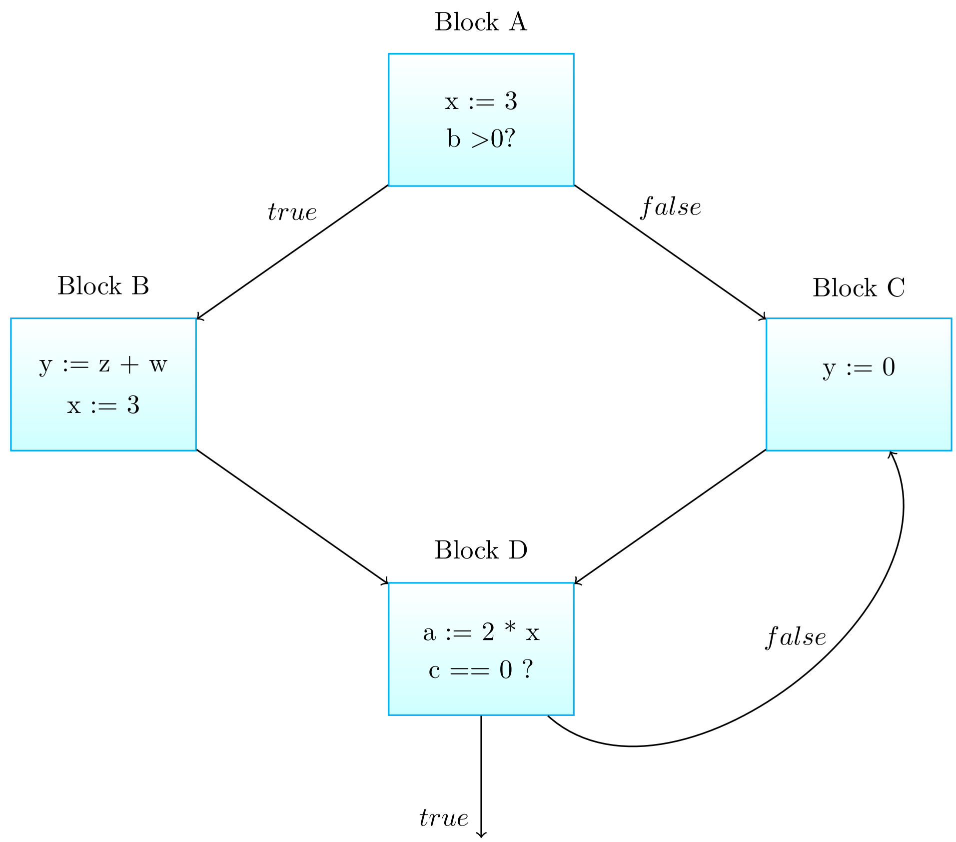
Figure 25: “y := 0” statemnt update



All the program points at this stage satisfy the transfer function rules, so this is a **Fixed Point Solution**.

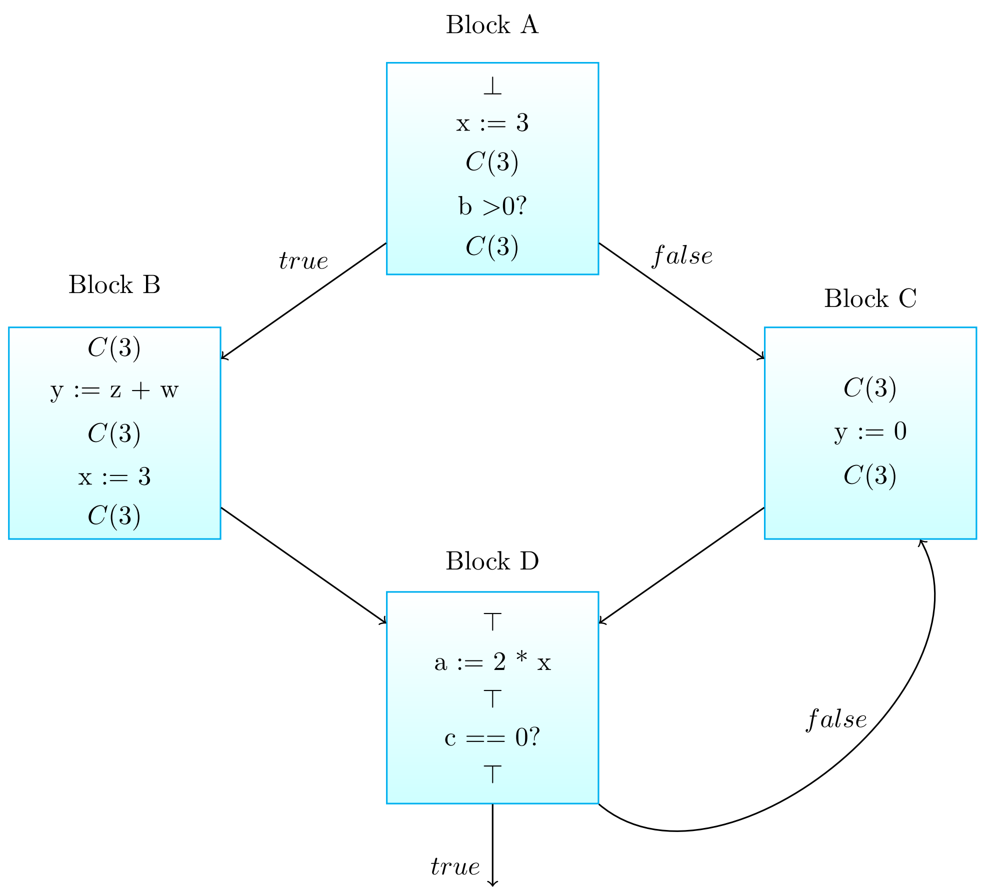
To show how the algorithm could have worked differently, we will change the "x := 4" statement to "x := 3" in Block C of our same example. This is our new example now.

Figure 26: Example



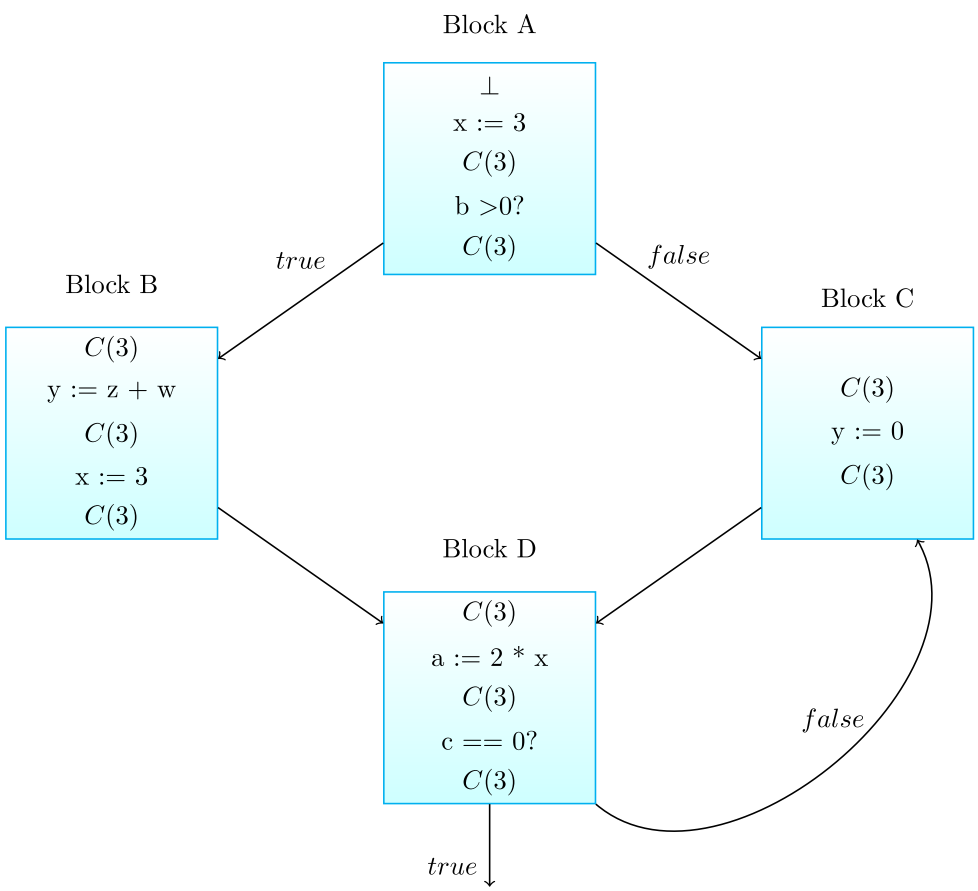
Algorithm will again start by initializing everything to $\top$, except the entry statement which is initialized to $\perp$. We will follow the same procedure and eventually reach this point after updating Blocks A, B and C.

Figure 27: "Blocks A, B and C" update



This time the successor of Block D will be updated to constant 3 instead of ⊥, because both the predecessors have the same value of constant 3. Updating the successor, and the following instructions in Block D, we reach this stage.

Figure 28: “Block D” update



Question : Have we reached the fixed point now?

Answer : It turns out that, in this case, we have reached the **Fixed Point Solution**. Both the predecessors of Block C, coming from Block A and Block D, have the value of Constant 3, so there is no need to update it.

9 DFA Value Orderings

In this section, we are going to discuss the convergence property of the DFA algorithm. And we will start by defining some arguments for it.

9.1 Orderings

Idea: Simplify the presentation of analysis by ordering the abstract values. We will create an arbitrary operator, “$<$” (less than), which orders them in the following manner.

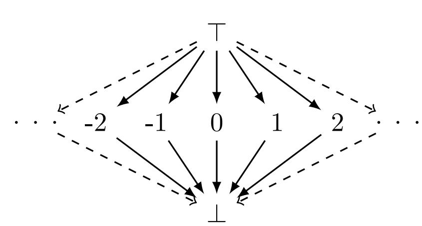
$$\perp\text{(bottom)} < \text{C(constant)} < \top\text{(top)}$$

This operator is a transitive which means that $\perp$(bottom) $<\top$(top) is also true.
We are going to use this operator to reason about the convergence properties of the DFA algorithm.

9.2 Partial Ordering

This ordering that we defined is a partial ordering, and not a total ordering. It means that among all the possible values, it is not necessary that all values are comparable. There are some values that are comparable, and some values that are not.
Another way to represent the partial ordering is by using a vertical representation instead of a horizontal representation, and using the directed arrow to represent the “$<$” (less than) operator. The arrow from $\top$ to C indicates that C $<\top$.

C is not a value, but a place holder for any constant. A better way to represent the ordering would be in the following manner.



Notice that the figure does not have any relation between 0 and 1, or 1 and 2, or between any two constants, in general. That is why it is called a partial ordering, because only some values are related by “$<$” operator.

Greatest Value: The value which is not less than any other value. In this case, it is $\top$(top).
Least Value: The value which is not greater than any other value.In this case, it is $\perp$(bottom).

9.3 Greatest Lower Bound

To build the concept of greatest lower bound, we introduce "$\leq$"(less than equal to) operator which in addition to all the properties of "$<$"(less than) operator, is also reflexive, *i.e.*, x $\leq$ x $\forall$ x.
The **greatest lower bound(glb)** of $x_1, x_2, ..., x_n$ is the greatest value that is lower than(by "$\leq$" operator) x_i, $\forall i$ s.t. $1 \leq i \leq n$.
Here are some examples of glb:

1. glb($\top$,1) = 1
2. glb($\top$,$\perp$) = $\perp$
3. glb(2,$\perp$) = $\perp$
4. glb(1,2,$\top$) = $\perp$
5. glb(1,2) = $\perp$

Observation: Glb can be used to replace the first four transfer function rules which defined the relation between the "out of the predecessors" and "in of the successor".

$$C(s, x, in) = \text{glb}\{C(p, x, out) \mid \text{p is a predecessor of s}\}$$

9.4 Convergence Argument

The DFA algorithm repeats itself until nothing changes. The only reason for it to not converge is, if it keeps changing forever. There are two reasons for it to keep changing forever.

1. Algorithm can keep taking a different value from the infinite set of values, for the same variable and at the same program point.
2. Algorithm can keep oscillating between the finite number of same values, for the same variable at the same program point.

Using glb, we will prove that the fix point iteration always converges. Here are the convergence arguments:

1. Values start at $\top$ and can only decrease.
2. $\top$ can change to C, and C to $\perp$.
   It cannot happen that $\top$ changes to some constant C, let's say 1 and in a different iteration it changes to a different constant, let's say 0. This is because one of the predecessors would still have the value 1, and if any other predecessor changes its value to a different constant, we are going to take the glb of predecessors which only allows the value to decrease in the order mentioned in this argument. This intuitive argument can be proved more formally by using induction on the Control Flow Graph of the program.
   Since this argument does not allow the value to go upwards, i.e., from C to $\top$, or from $\perp$ to C, or from $\perp$ to $\top$, oscillation is not possible among these values.
3. Thus, C(s, x, in/out) can change at most twice, $\forall$ s and x.

9.5 Worst Case Execution Time

9.5.1 For One variable

At least one value changes in each iteration, and at every program point a value can change twice therefore:
Number of steps of Fixed Point Iteration Algorithm $\leq$ (Number of C(s, x, in/out) statements) * 2
There can be at most two C(s, x, in/out) statements per each statement therefore:
Number of steps of Fixed Point Iteration Algorithm $\leq$ (Number of program statements) * 4

9.5.2 For all variables

There are two options to deal with all the variables.

1. Run the algorithm separately for each variable.
   To calculate the worst execution time, we multiply the number of variables to worst execution time for one variable. The number of variables would be less than the number of program statements, assuming Three Address Code. Therefore, the worst case would be quadratic in size of the program.

2. Keep track of all variables simultaneously.
   This can be done by modifying the transfer function such that it looks at every variable simultaneously and updates multiple values in one step. It improves the performance because looking at a set of variables is usually cheaper that looking at each of them separately.
   It also provides more information while updating the values. For example, if we have a statement like "x := y + z", and we have the information that y and z are constants at this point, then we can also infer that x is a constant. Had we dealt with the variables separately, this would have not been possible. In another words, the analysis would have been weaker. If we do things separately, we may end up with less precise solution in some situations, for some type of analysis.