**Phase three**

**Java Byte Code Generation**

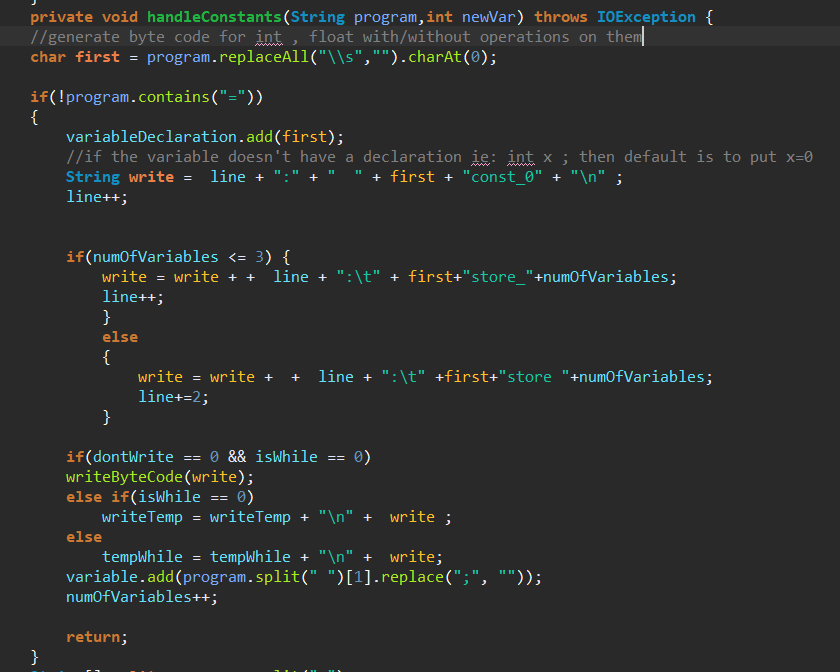
**-Description :**

We implemented this phase using our own parser generator implemented in phase two , instead of bison –Bonus Part –

This phase wont run unless phase two has no errors i.e : No semantic errors in the program.

**-Functions explained :**

1)—**handleConstants** – This function handles the cases of declaration of a new variable , appling operation on it



The above handles the case of int x; or float x;

We set the default of any new variable as (0) –Initialize the variable- so that there wont be any problems faced if we don’t initialize the variable

First we check if the line contains (=) – no initialization – then we add it’s type whether it is an int or float to an arrayList called **variableDeclaration** and also add the new variable to another arrayList holding the names of each variable called **variable**

The variable **numOfVariables** is just a counter to count the number of variables found in the program , this helps in writing the bytecode as when storing , loading the variables they should be placed in certain places.

The variables **line** is also a counter but it counts the bytes that is used by bytecode .

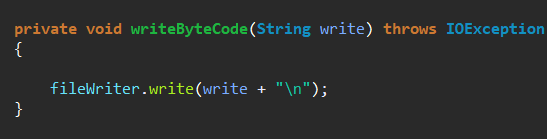
In bytecode there are default store mnemonics either in int(i) or float(f)

Store\_0 Store\_1 Store\_2 Store\_3 -🡪 Take only 1 byte

Store 4~more 🡪 Take 2 bytes , one byte for opcode one byte for the number , as it is not build in

That’s why the code checks if numOfVariables is <=3 to see which mnemonic to use.

dontWrite & isWhile are both variables used when (if) or (while) is present , they will be explained later.



If there is no (if) nor (while) then we write directly to the output file .

**2) Primitive types with declaration :**



The rest of –handleConstant—function , checks if the declaration of a certain constant is an operation or a number

It first splits on (=) then String check is the number/Operation after the (=)

The int called newVar is used as a flag to check if this variable was already declared or not , if not then we have to get it’s primitive type (int or float) and it’s index.—To load and store it—

The **try catch** is used to check if the declaration is a number or an operation

b-1) **handleNum**

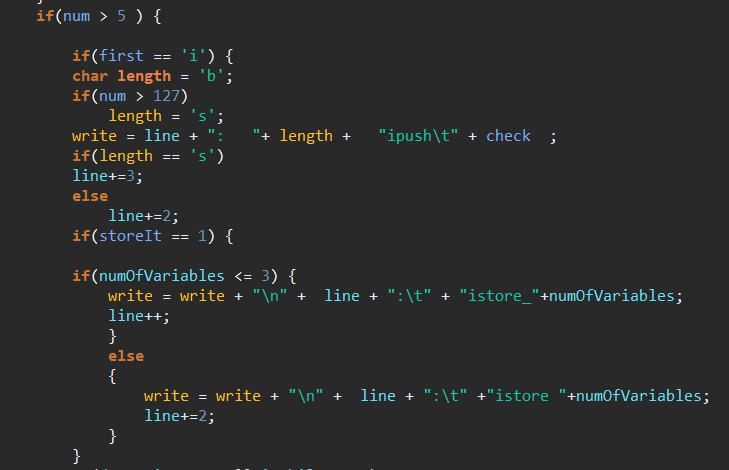
There are certain cases for a number to be loaded to the stack in bytecode ,here it shows if it is a negative number.



If num == -1 then the bytecode mnemonic is iconst\_m1

Then we check if we want to store it or not yet , if yes we do the same as we did before and that is checking the numOfVariables and choosing the right mnemonic to use and how many bytes .

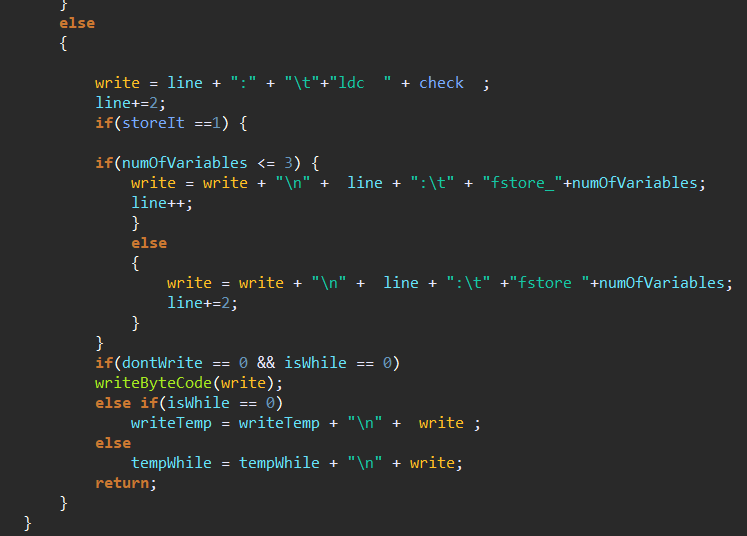
And check the (dontWrite and isWhile) that will be explained later.



If a number is greater than 5 then we need to use different mnemonics for int and float , the above shows the case of int

If the num is short , meaning the num > 127 then we will use sipush num , this takes 3 bytes , 1 for opcode 2 for the num

If it is less then it is a byte 🡪 bipush num , this takes 2 bytes , 1 for opcode 1 for num.



This is the same as the previous but if the declaration is float we use 🡪 Idc num with 2 bytes

**b-2)--- handleOp –**



First we check which operation is performed and let String op be the bytecode mnemonic – Knowing that int and float are similar but have different beginnings

So char first indicates if it is int or float (iadd or fadd… etc)

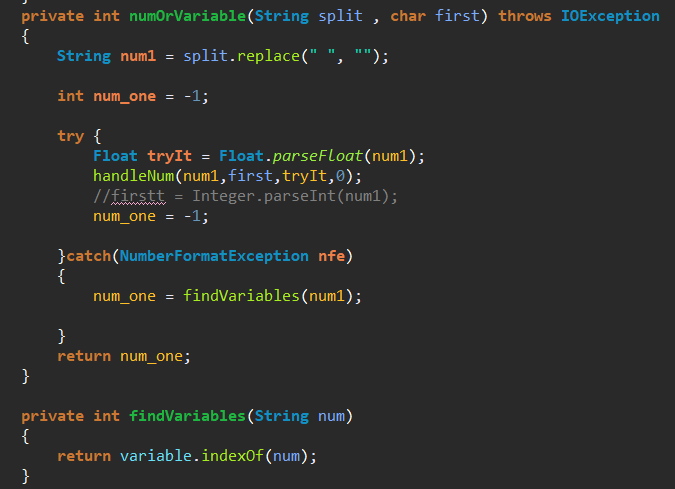
num\_one 🡪 is the number or variable we operate on

num\_two 🡪 is the second variable or number we operate on

In the following figure it shows how this is done by try..catch ,

if it is a num it handles it using the function explained **handleNum** ,setting the storeIt =0

Else it is a variable and we get the index of this variable in order to load it.



**Continue function –Handle Op--**

Then if the num\_one is a number it is already loaded in the stack waiting for num\_two if it is a variable then it has two choices



If it’s index is from 0-3

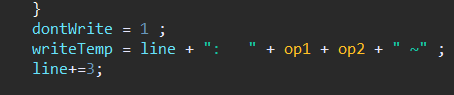
Then use load\_0 load\_1.. with only one byte

Else we use load 4~more with 2 bytes.

Same goes for num\_two.

**3) If Condition.. else :**

There are two variables used



Int **dontWrite** , String **writeTemp**

**dontWrite** will be set to 1 when the program starts an if condition

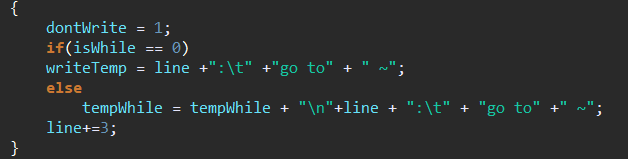
and is set to 0 when the if conditions finishes (when it finds ‘}’ )

**writeTemp** is used to store the bytecode of the if till it finishes , this is used

to ensure back tracing where we set the number that should point to the

end of the if with (~) then replace it when we finish reading the if

condition.



If there is an **else** the bytecode go to ~ also has the (~) symbol till we reach the

end of this condition.

The following figure shows the function that handles the if , in bytecode the standard if is used in two ways

Either to compare with zero Or compare with number/variable

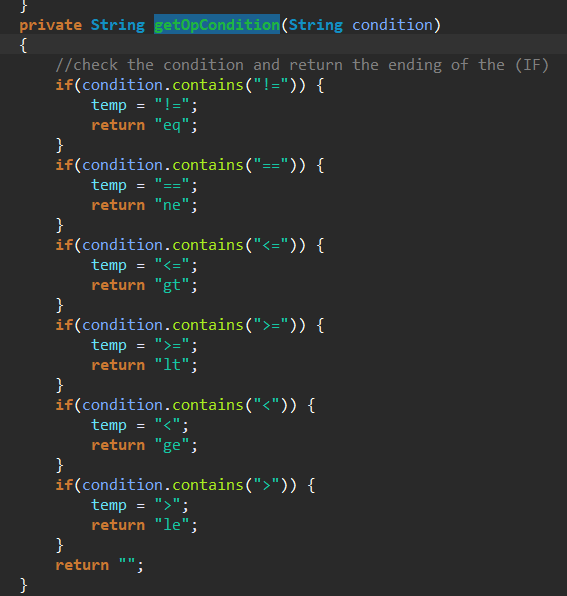
The bytecode mnemonics are similar but work differently

So the code first checks if the condition is compared to a zero then the (if) is the start of the mnemonic else the start is (if\_icmp)

Then var1 is always a variable so we get the variable’s index and load it , then if we compare with something other than zero so we need to get the variable or number as we did in previously.



This gets the ending of the bytecode mnemonic



Ex: ifeq , if\_icmpeq

Ifne , if\_icmpne

**Handling the back tracing in if condition :**

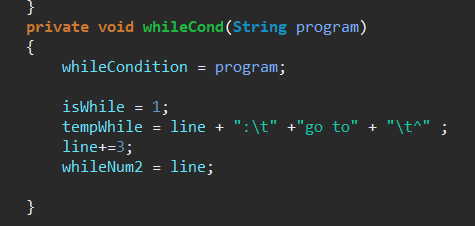


When dontWrite is 1 , meaning there was an if condition or else then we need to check first the next line to be read if there is an (else) so the (~) should be replaced by the line number after the (go to ) statement

Which is t=line+3;

Else it is replaced by the current line that will be written.

**3)While :**

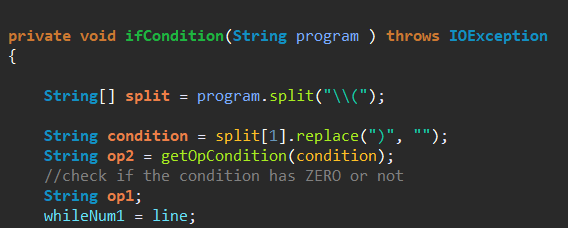


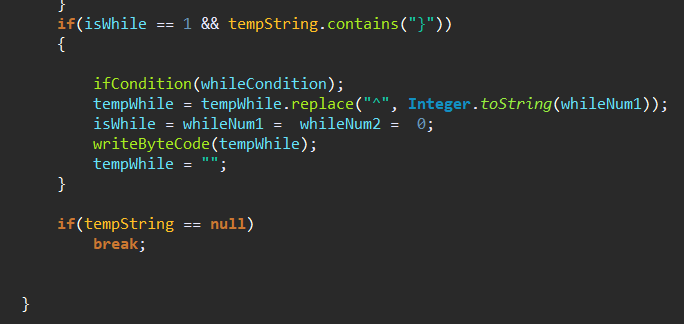
Three variables are used for the while:

**String tempWhile ; 🡪 used to store the bytecode that is inside the while loop till it reaches ( } ).**

**Int isWhile; 🡪** used as a flag to help in writing the bytecode in the **tempWhile** string other than the file.

**Int whileNum1 , whileNum2; 🡪**Two numbers are used in the back tracing both will be substituted at the **end.**

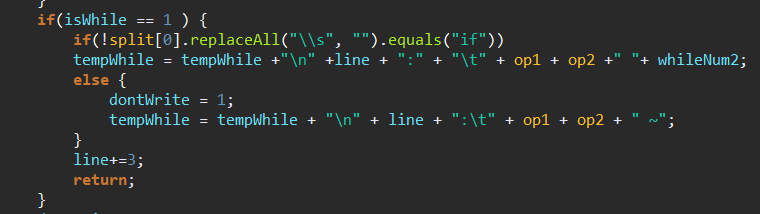
**Back tracing in while:**



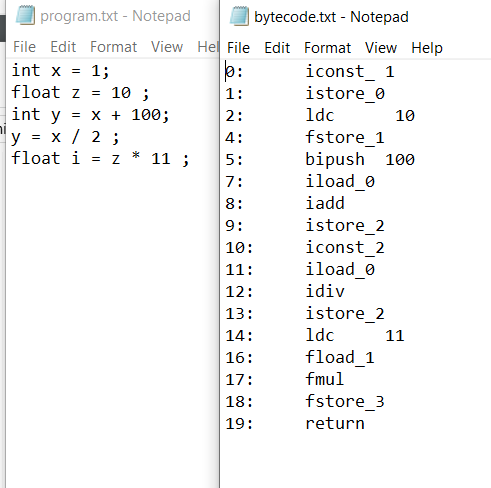
The int whileNum1 is gonna be the line before writing the if condition of the while loop i.e: in bytecode the while loop’s bytecode ends with it’s condition

Ex:

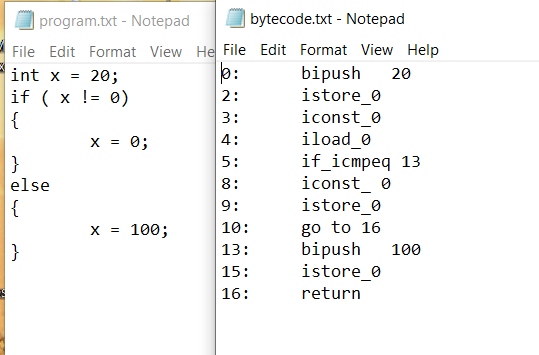
While(x != 0) 🡪 the (x != 0) will be considered an if statement and the number it follows is the **beginning** of the while loop.—whileNum2 --

****

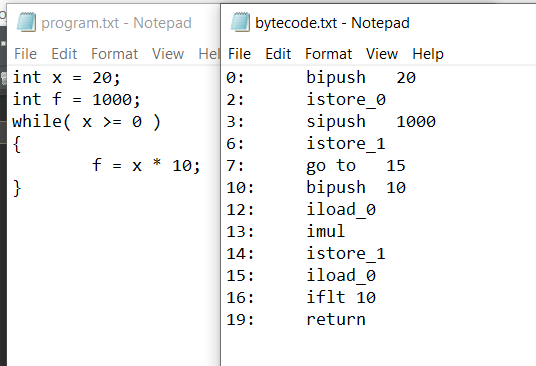
**Sample Runs :**



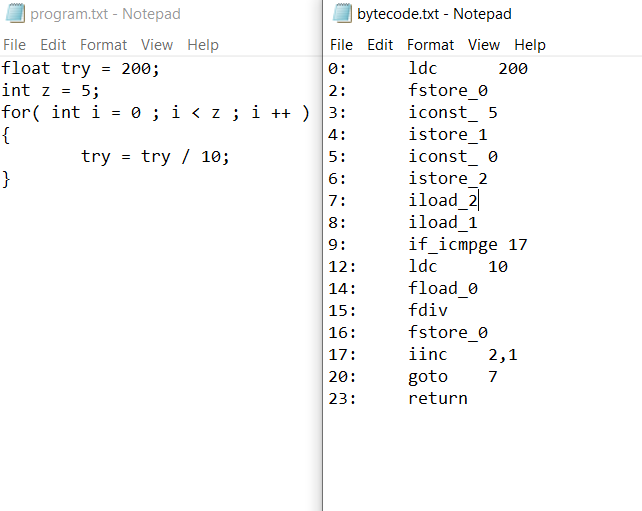
**Sample (1)**



**Sample (2)**



**Sample(3)**



**Sample(4)**