

Faculty of Computer Science

Fall Semester 2020

CS334

Compiler Design Project Fall 2020

**Done by:**

|  |  |
| --- | --- |
| **Name** | **ID** |
| Ahmed Mohamed Abdelfattah Issa | 191947 |
| Ali Mohamed Hobeldein | 192231 |
| Mohamed Khaled El Sharkawy | 170879 |

**Submitted to:**

**Dr. Zeinab Abd El Halim.**

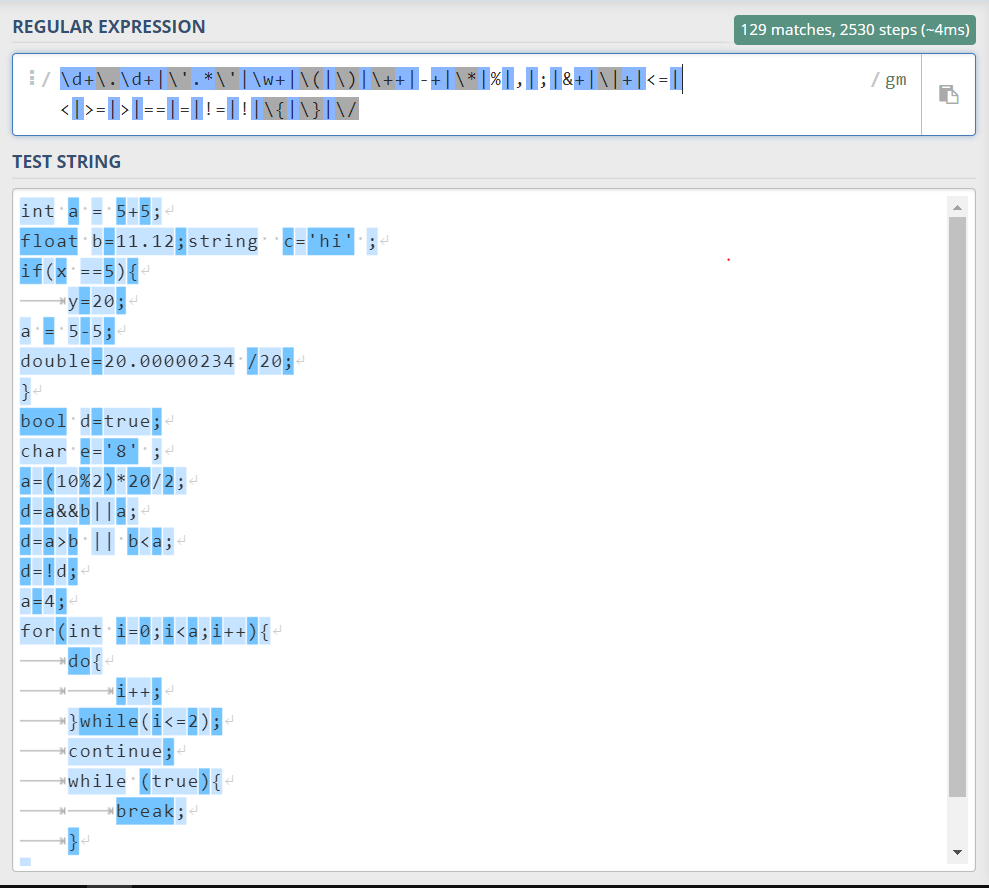
|  |  |
| --- | --- |
| |  | | --- | |  | |

**Dr. Rahma Hussein.**

**Phase 1:**

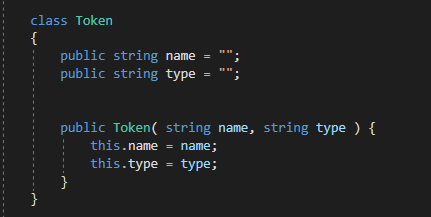
This phase is the process of converting a sequence of characters (Code Block) input by user into a sequence of tokens.

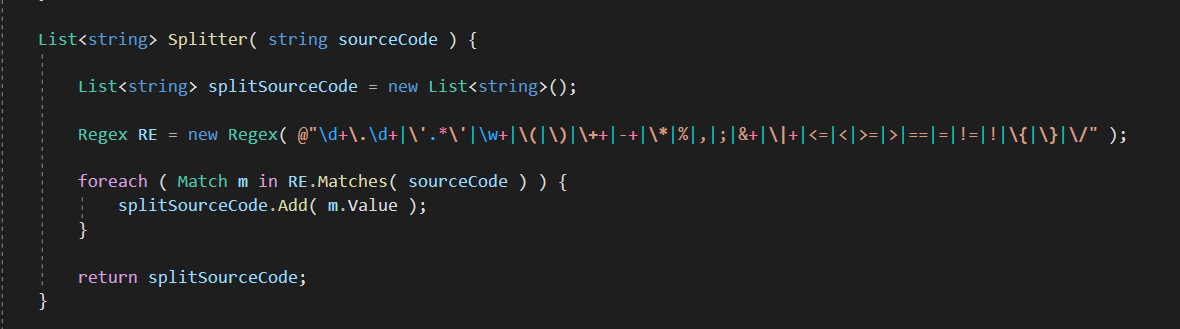
Instead of DFA or split, we have decided to use regular expression match method because it allows you to specify the tokens you are looking for like the image and it is very simple.



Splitter : return list of matches strings.

Create class Token to store name and type of token that we will use later.



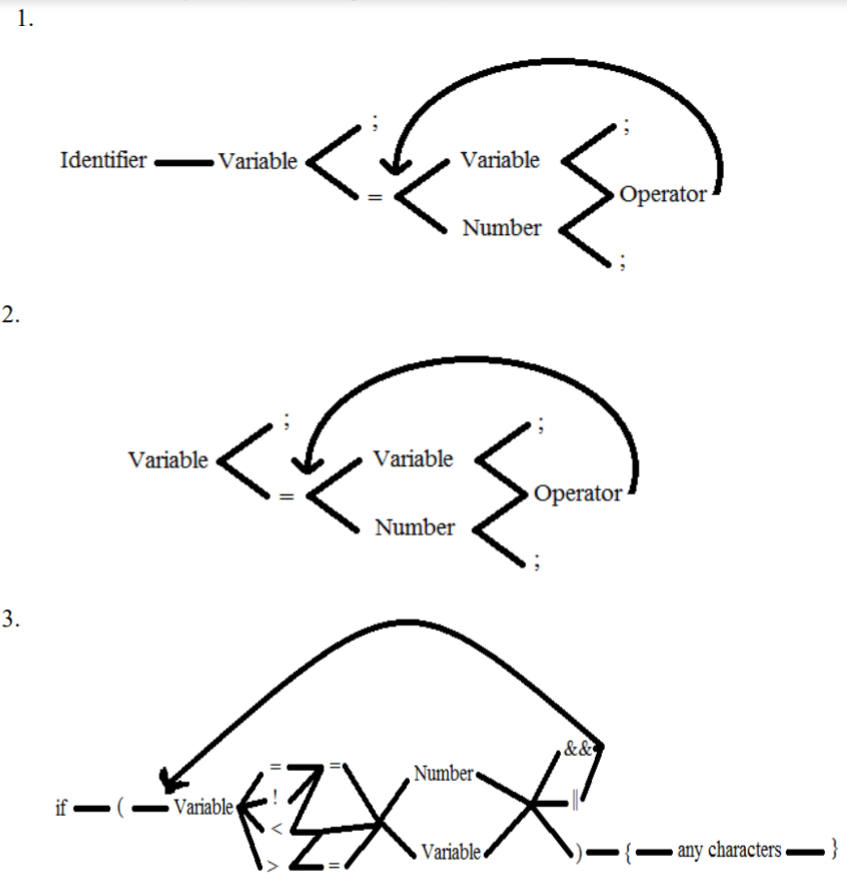


Scanner: take the list of matches strings, get the type of each string and store it in the List of Nodes (Token).

**Phase 2:**

In this project phase, a semantic analyzer is developed so that errors in the code block could be identified, the SemanticAnalyzer() function takes the list of tokens identified by the Scanner, and checks if the tokens sequence follow any of the three rules provided in the project description using Top-Down Parsing, then it return a list of the errors found.

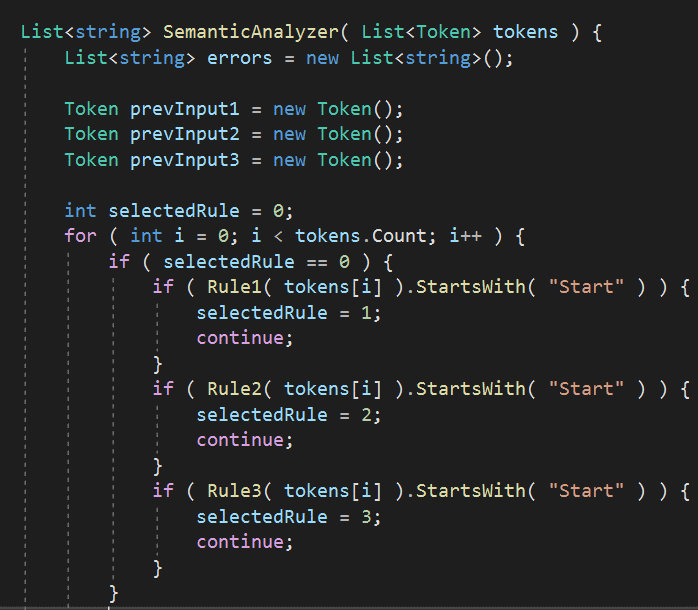
The three rules:

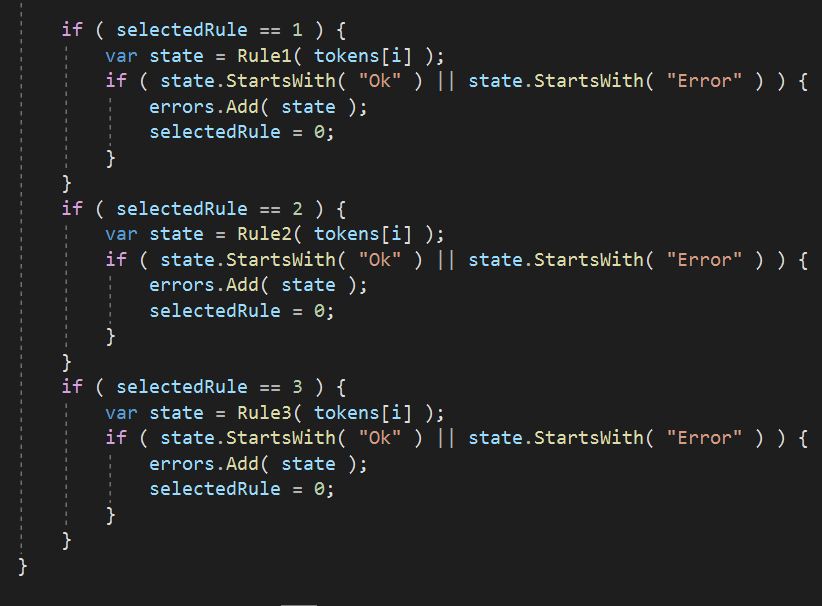


An overview of the SemanticAnalyzer() function with collapsed code:

****

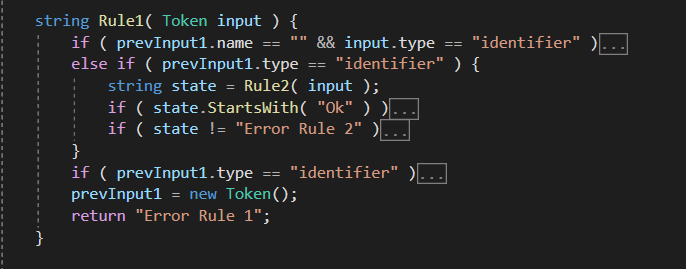
The SemanticAnalyzer() has a main “for” loop that iterates over the given list of tokens, it test which of the three rules will start accepting the i-th token, once a rule starts accepting, this rule becomes the selected rule, it will remain selected until it either returns an error or returns an “Ok”, which means the last sequence of tokens did follow the rule correctly:

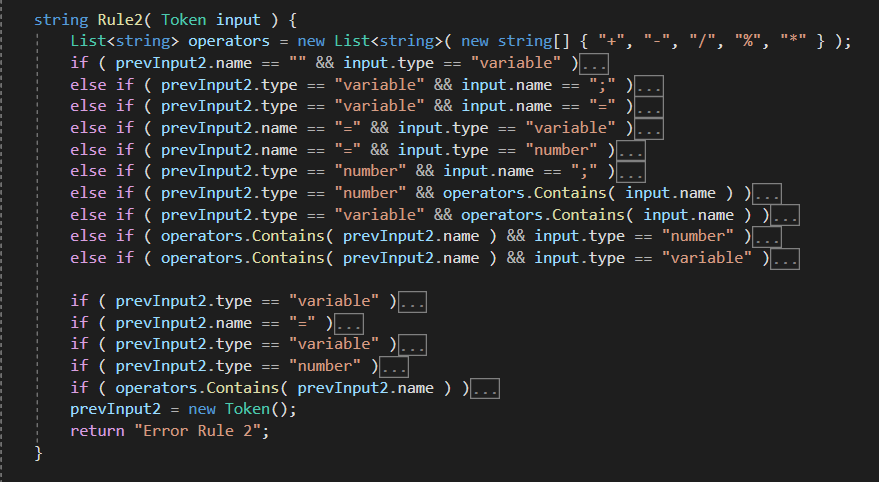


****

Each rule takes a token from each loop iteration and stores the previous token it has seen inside prevInput. Inside each rule, the upper half contains the conditions that begin, ends and continues the application of that rule, the lower half catches the errors and returns what the rule expected instead.

Role 1,2,3:

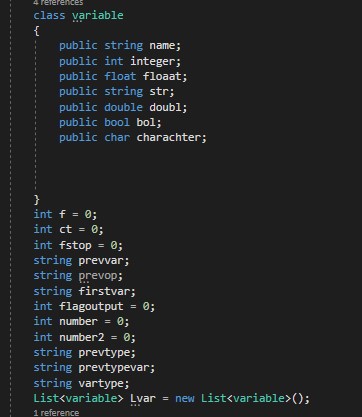
****

****

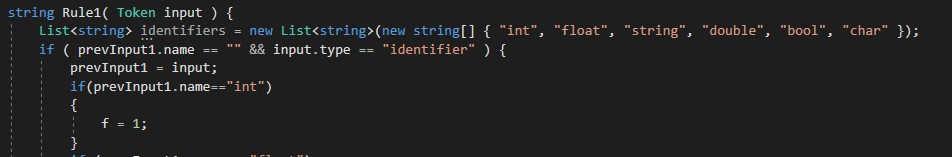
**Phase 3:**

In this phase we save the variable value in the memory and every variable shows on the data grid view.

Our class variable and list of class variables.

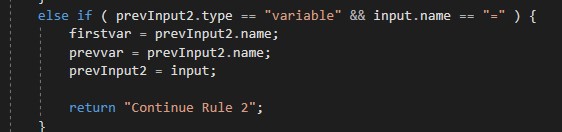


First in rule1 function we put a flag (f=1) to know that the identifier is integer

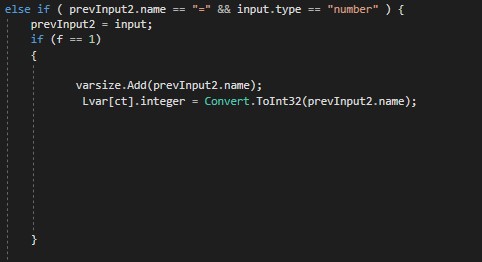
****

In rule 2 function We made first var to catch the variable before “=”

prevvar to catch the first variable after “=”



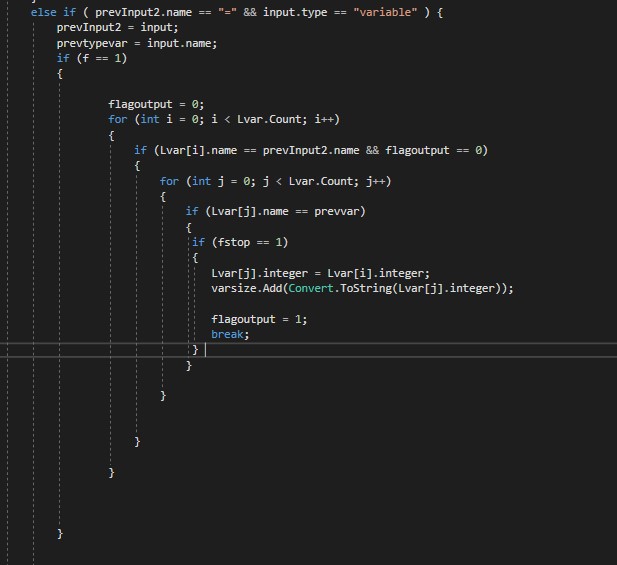
In this condition the variable becomes any integer value

****

Ex: int x=3;

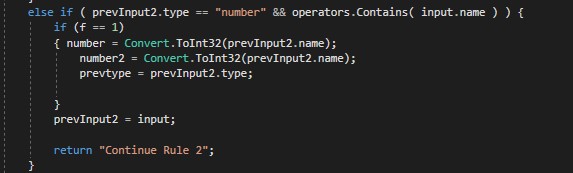
In this condition integer variable becomes another integer variable

Ex: int x= y;

****

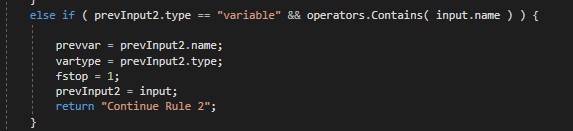
In this condition we catch the number before the operator.

Number becomes this number.

****

In this condition we catch the variable before the operator.

Prevvar becomes this variable.

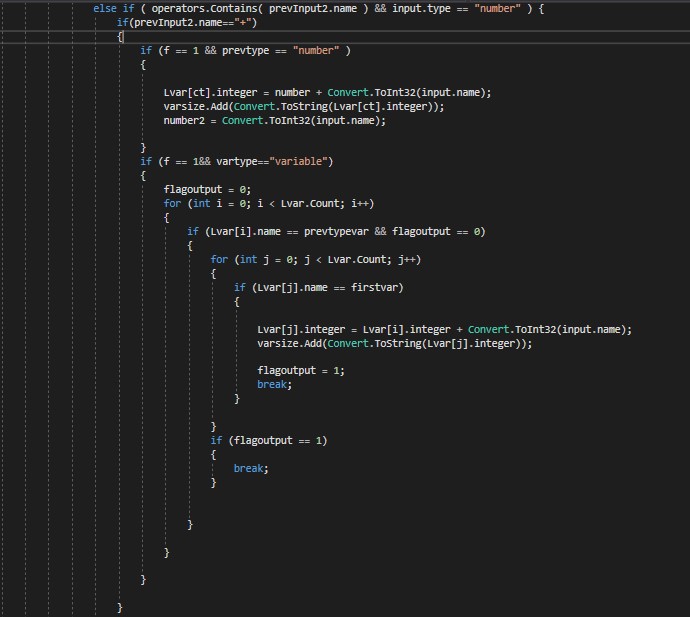
****

In this condition we take the values of “+” operator when

Num+ num

And

Integer variable + num

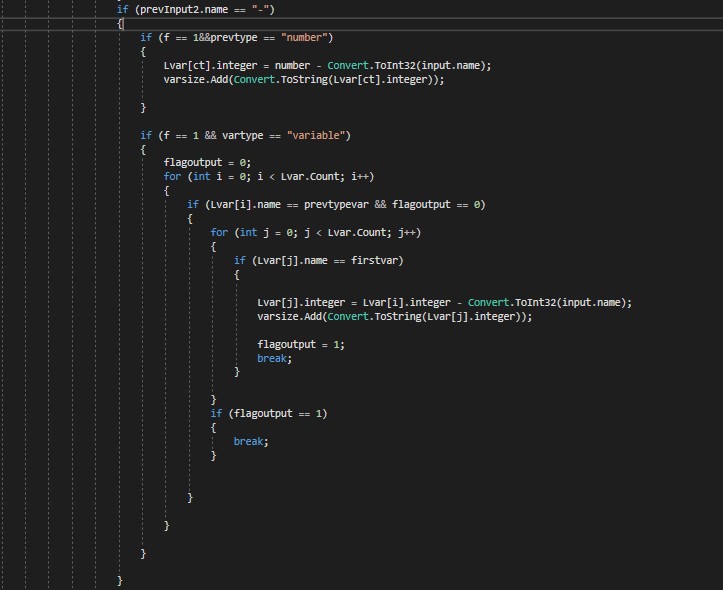
****

In this condition we take the values of “-” operator when

Num-num

And

Integer variable-num

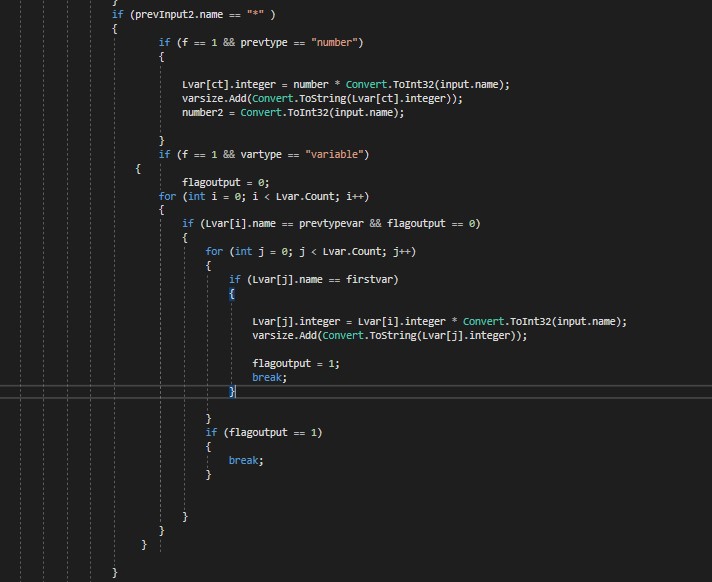
****

In this condition we take the values of “\*” operator when

Num\*num

And

Integer variable\*num

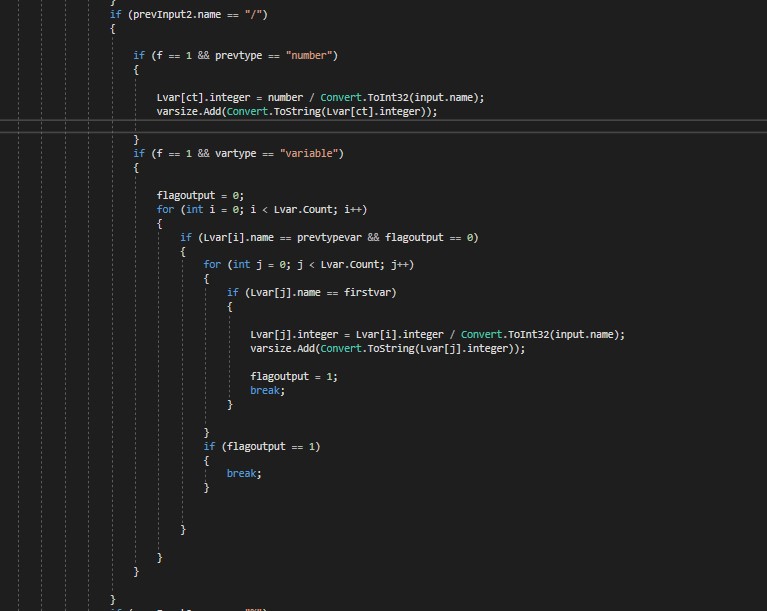


In this condition we take the values of “/” operator when

Num/num

And

Integer variable/num

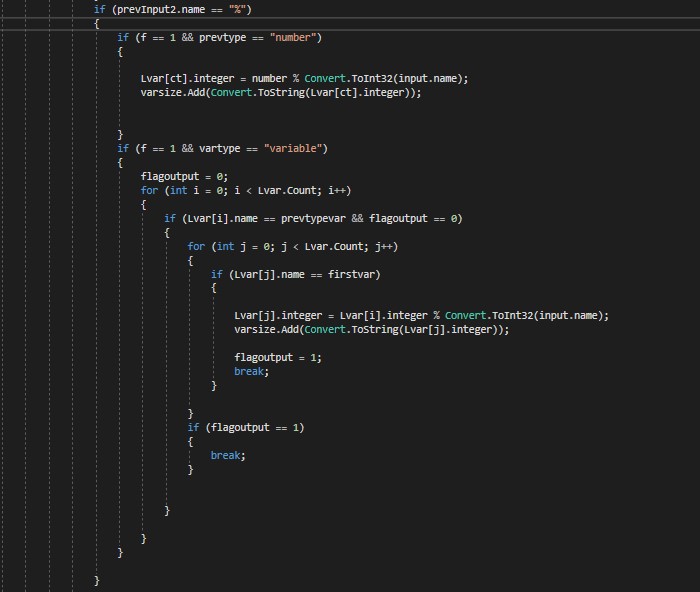
****

In this condition we take the values of “%” operator when

Num % num

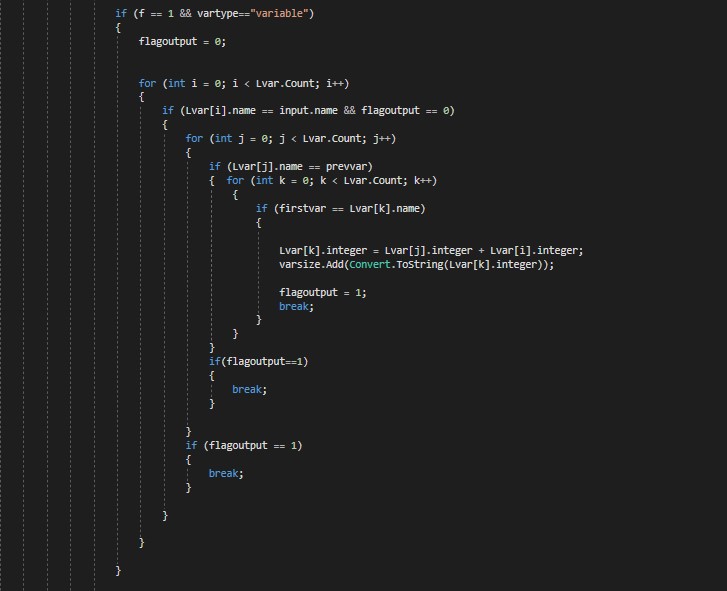
And

Integer variable % num

****

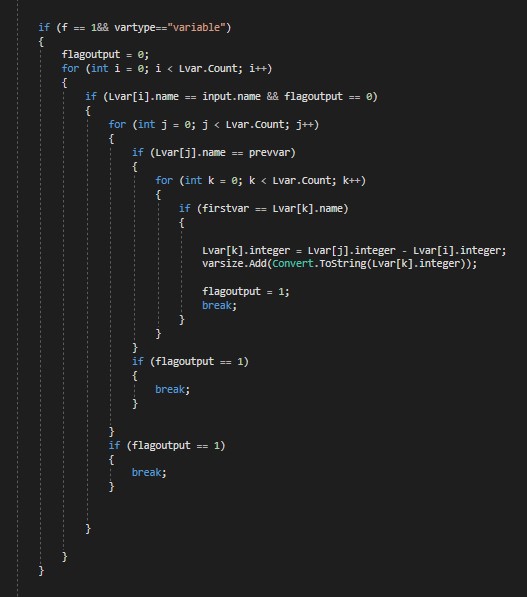
In this condition we take the values of “+” operator when

Integer variable+ Integer variable

****

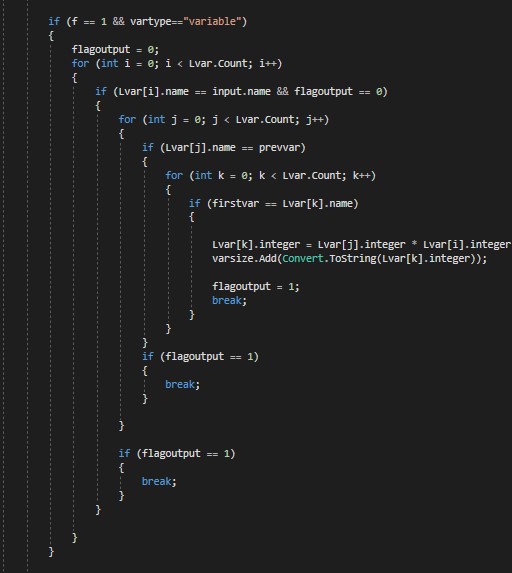
In this condition we take the values of “-” operator when

Integer variable- Integer variable

****

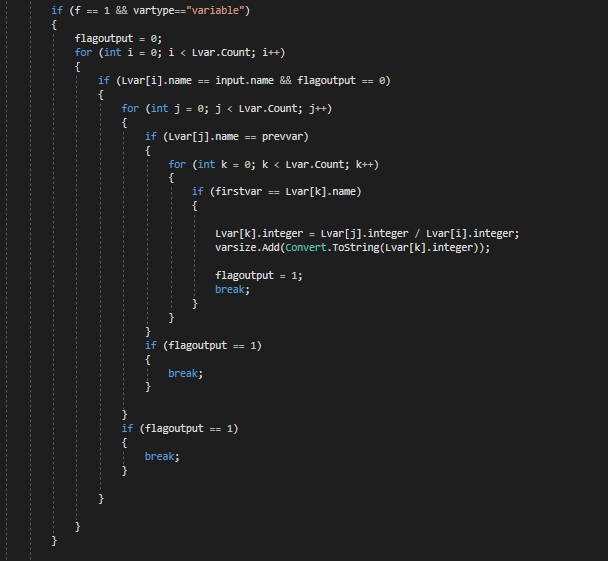
In this condition we take the values of “\*” operator when

Integer variable\*Integer variable

****

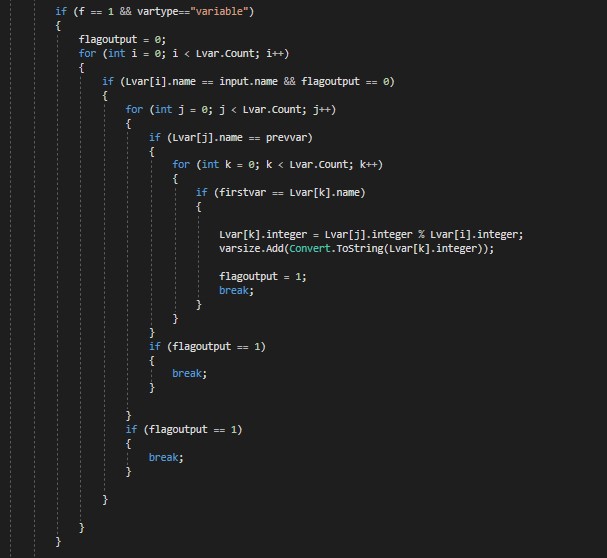
In this condition we take the values of “/” operator when

Integer variable/Integer variable

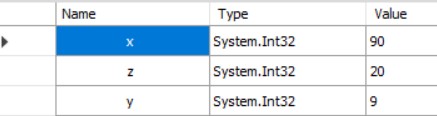
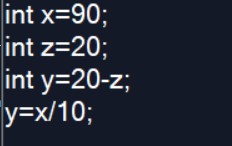
****

In this condition we take the values of “%” operator when

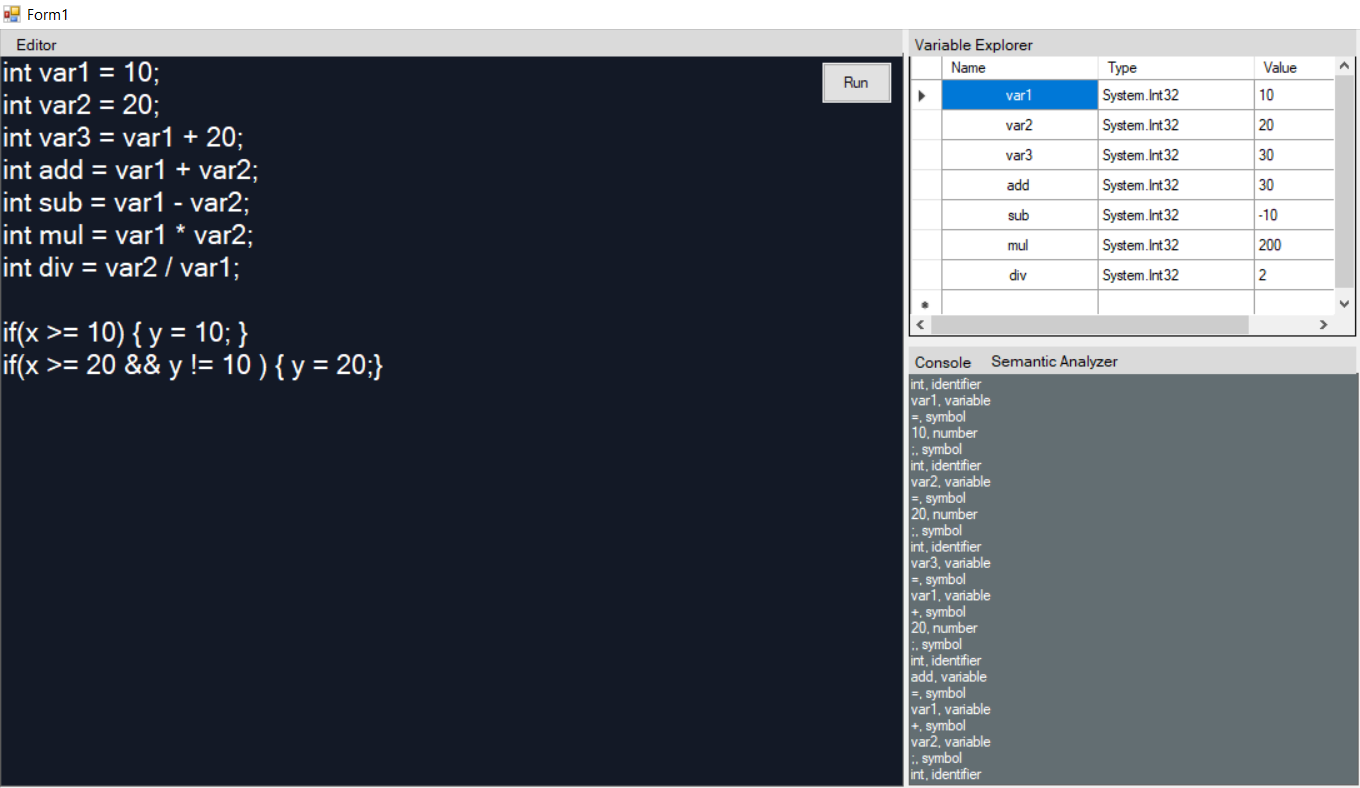
Integer variable% Integer variable

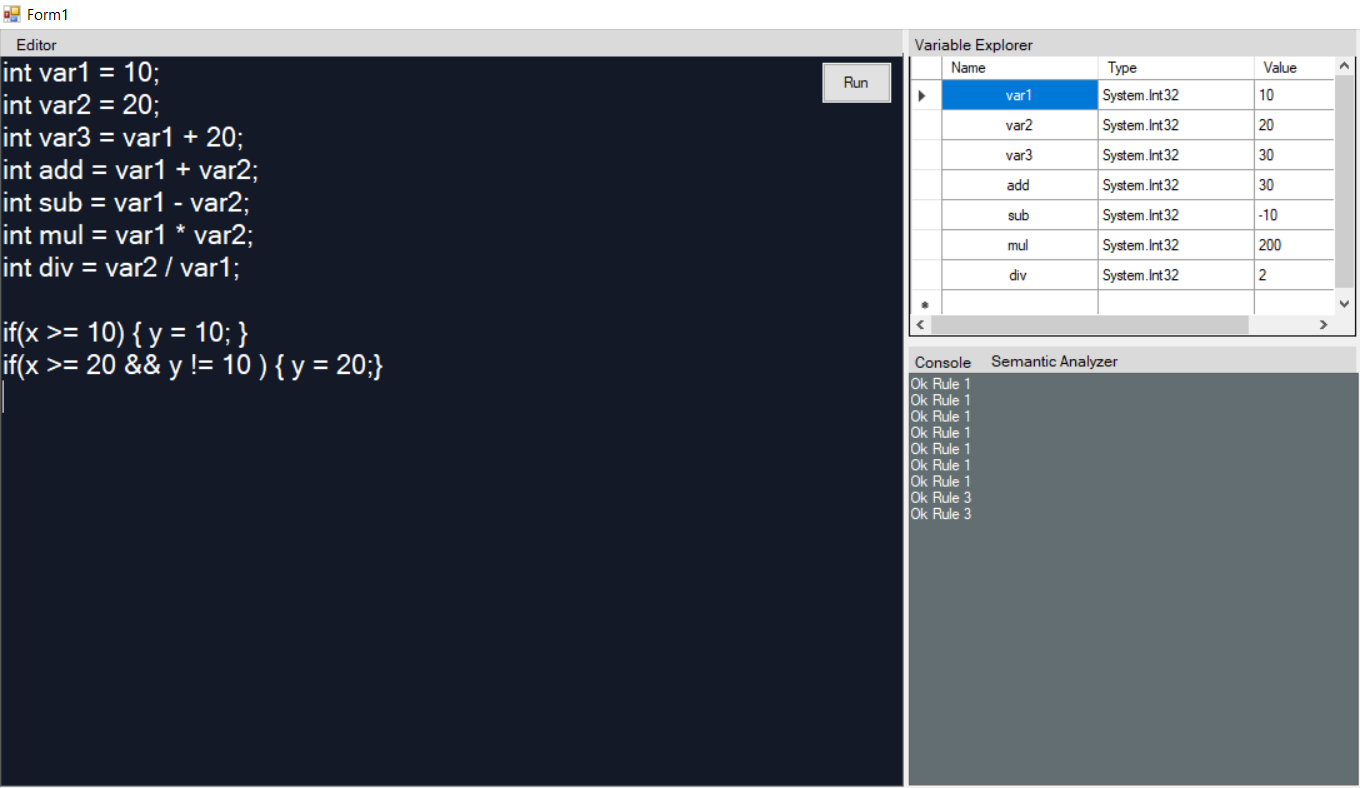
****

Test Case :



**User Interface:**





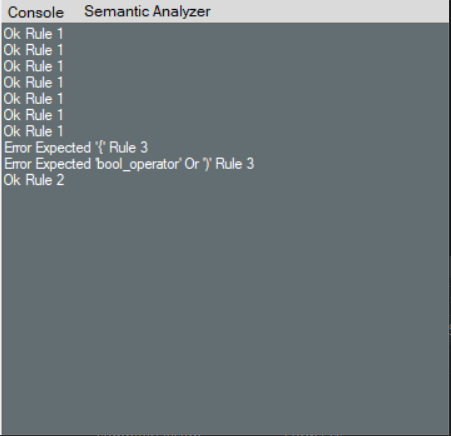
**Output :**

Console : int, identifier var1, variable =, symbol 10, number ;, symbol int, identifier var2, variable =, symbol 20, number ;, symbol int, identifier var3, variable =, symbol var1, variable +, symbol 20, number ;, symbol int, identifier add, variable =, symbol var1, variable +, symbol var2, variable ;, symbol int, identifier sub, variable =, symbol var1, variable -, symbol var2, variable ;, symbol int, identifier mul, variable =, symbol var1, variable \*, symbol var2, variable ;, symbol int, identifier div, variable =, symbol var2, variable /, symbol var1, variable ;, symbol if, reserved word (, symbol

**Semantic Analyzer**

Checking the lines with roles

Ok Rule 1 Ok Rule 1 Ok Rule 1 Ok Rule 1 Ok Rule 1 Ok Rule 1 Ok Rule 1 Ok Rule 3 Ok Rule 3

If we replace

if(x >= 10) {y = 10; }

if(x >= 20 && y != 10) { y = 20;}

with

if(x >= 10) y = 10; }

if(x >= 20 && y != 10 { y = 20;}

the analyzer output :

**Memory:**

The memory appears in data grid view with variable type and value.

