Command-Line-Based Simple Program Interpreter

Project Report

Group 9

COMP2021 Object-Oriented Programming
(Fall 2022)

LIU Minghao

ZHANG Tinayi

Contents

1 Introduction

This document describes the design and implementation of an interpreter for SIMPLE programs by group 9. The project is part of the course COMP2021 Object-Oriented Programming at PolyU.

2 A Command-Line-Based Interpreter for Simple Programs

This section will introduce the design of all aspects of the Command-Line-Based Interpreter for Simple Programs and the principles of implementing all the requirements and bonus of the project.

2.1 Design

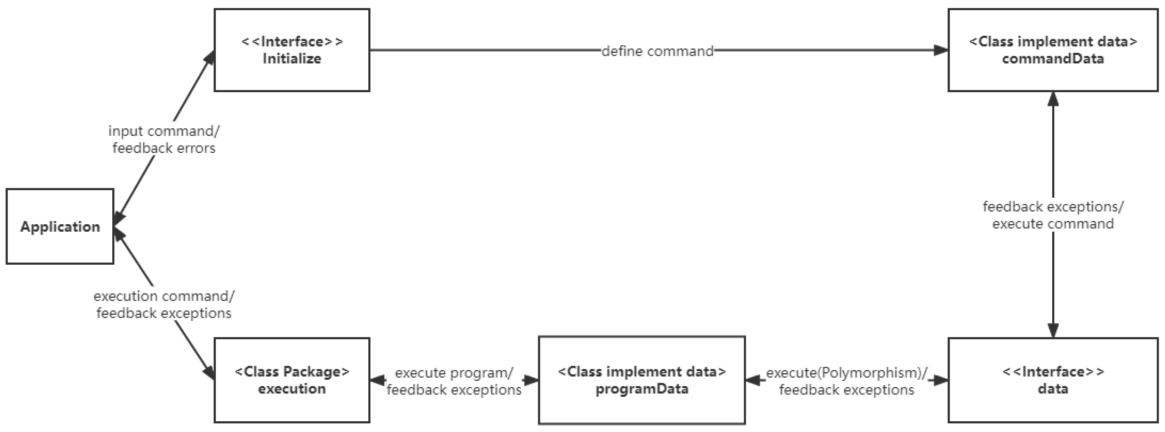


Figure 1: Overall Design

In general, this Command-Line-Based Simple Program Interpreter is based on the MVC design template (See Figure 1). The user will run the application and input the command. If there are some errors in the command statement, the **initialize** will return the errors and this line of command will be ignored. Otherwise, the command data will be stored in corrsponding **commandData (eg. assignData, binexprData)** which implements the interface **data**. If the command user input is an execution command (eg. execute, debug) in the package **execution**, it will execute the programData and use dynamic binding(Polymorphism) to execute the **data** directly and feedback exceptions.

2.1.1 Data Structure

- HashMap< key, value >

  A HashMap is used to store the data which needs to be found through a key directly.

- Vector

  A vector is used to store the variables, break-points, code and instruments, in case that its length can be changed in time.

2.1.2 Interface

1. data
   - MAX_INT_VAL and MIN_INT_VAL

     Set them to -99999 and 99999 respectively. They will be used to check if an int value is out of range.

   - inner class ele

     It stores variable name and value of it by two private fields, varName and val. And by using getVarName() and getVal(), the fields can be accessed. Besides, a function setVal(String x) is used to assign a new value of the variable.

   - String exe(String programName, boolean isDebugging)

     This abstract method should be implemented by its implementers. When this method is called, it will execute the corresponding (dynamic binding) command and return the result. And the parameter programName is used to get the **programData** which stores all the data of this program by **Memory**.

- String getLab()

  Should be implemented and return the label of the corresponding command.

- void store(Vector<String> v)

  Should be implemented and store this line of command to vector <u>v</u>.

2. initialize
   - unNameSet

     Store all Simple program keywords

   - HashMap<String, data> Memory

     Memory stores all the legal commands. And set the key as the label of command for faster query. And set <u>data</u> as the value which contains all the information of one command.

   - boolean error(String[] s)

     Should be implemented and it checks the errors in the command statement.

   - void define(String[] s)

     Should be implemented. It defines the **commandData** and puts the information into **Memory**.

   - Other check-error functions

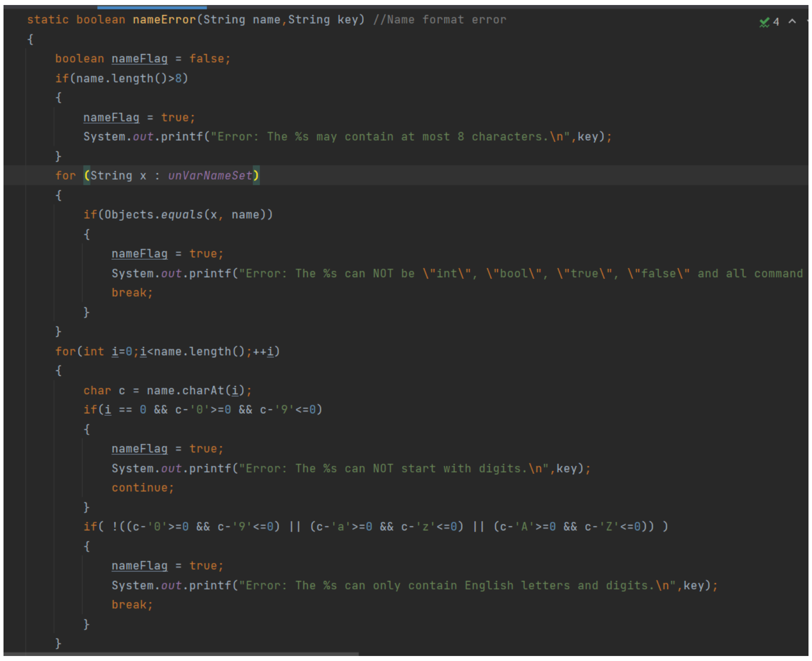
     See 2.1.3 for details. These functions are put in this interface because they will be called frequently by the classes that implement it.

2.1.3 Error Feedback

The errors appeared during **initialize** are the <u>invalid command format, naming convention errors and repeat defined or undefined errors</u>. And during **execution**, it will return <u>type errors, variable repeated definition errors and undefined variable errors</u>.

- Errors during initialize (<interface> initialize)
    1. boolean nameError(String name, String key)

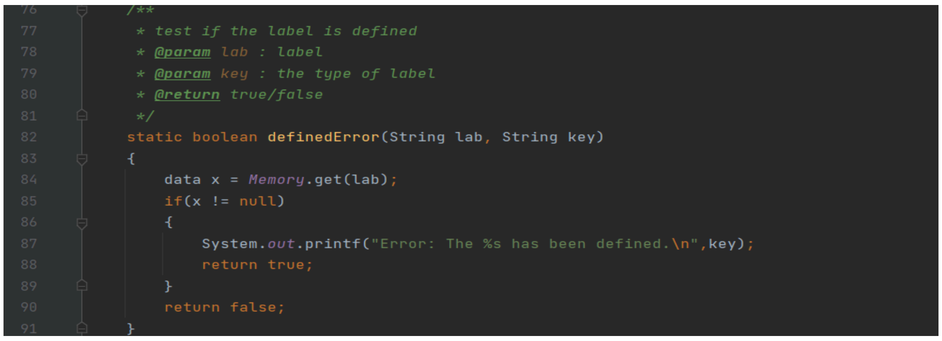
        The parameter name is the test target and key is the type (eg. label, expName, varName) of this name. It can test if the length of name is longer than 8 to make sure it has at most 8 characters. And beginning with digits is not allowed. Besides, it will return an error if name not only contains English letters and digits. There is an unNameSet containing all the keywords in Simple program. The set is used to check if the name is illegal.



```java
static boolean nameError(String name,String key) //Name format error
{
    boolean nameFlag = false;
    if(name.length()>8)
    {
        nameFlag = true;
        System.out.printf("Error: The %s may contain at most 8 characters.\n",key);
    }
    for (String x : unVarNameSet)
    {
        if(Objects.equals(x, name))
        {
            nameFlag = true;
            System.out.printf("Error: The %s can NOT be \"int\", \"bool\", \"true\", \"false\" and all command
            break;
        }
    }
    for(int i=0;i<name.length();++i)
    {
        char c = name.charAt(i);
        if(i == 0 && c-'0'>=0 && c-'9'<=0)
        {
            nameFlag = true;
            System.out.printf("Error: The %s can NOT start with digits.\n",key);
            continue;
        }
        if( !((c-'0'>=0 && c-'9'<=0) || (c-'a'>=0 && c-'z'<=0) || (c-'A'>=0 && c-'Z'<=0)) )
        {
            nameFlag = true;
            System.out.printf("Error: The %s can only contain English letters and digits.\n",key);
            break;
        }
    }
```

    2. boolean definedError(String lab, String key)

        The parameter lab is the input label and key is the label type. This function will check if this label has been defined by using a HashMap which stores all the defined commands.



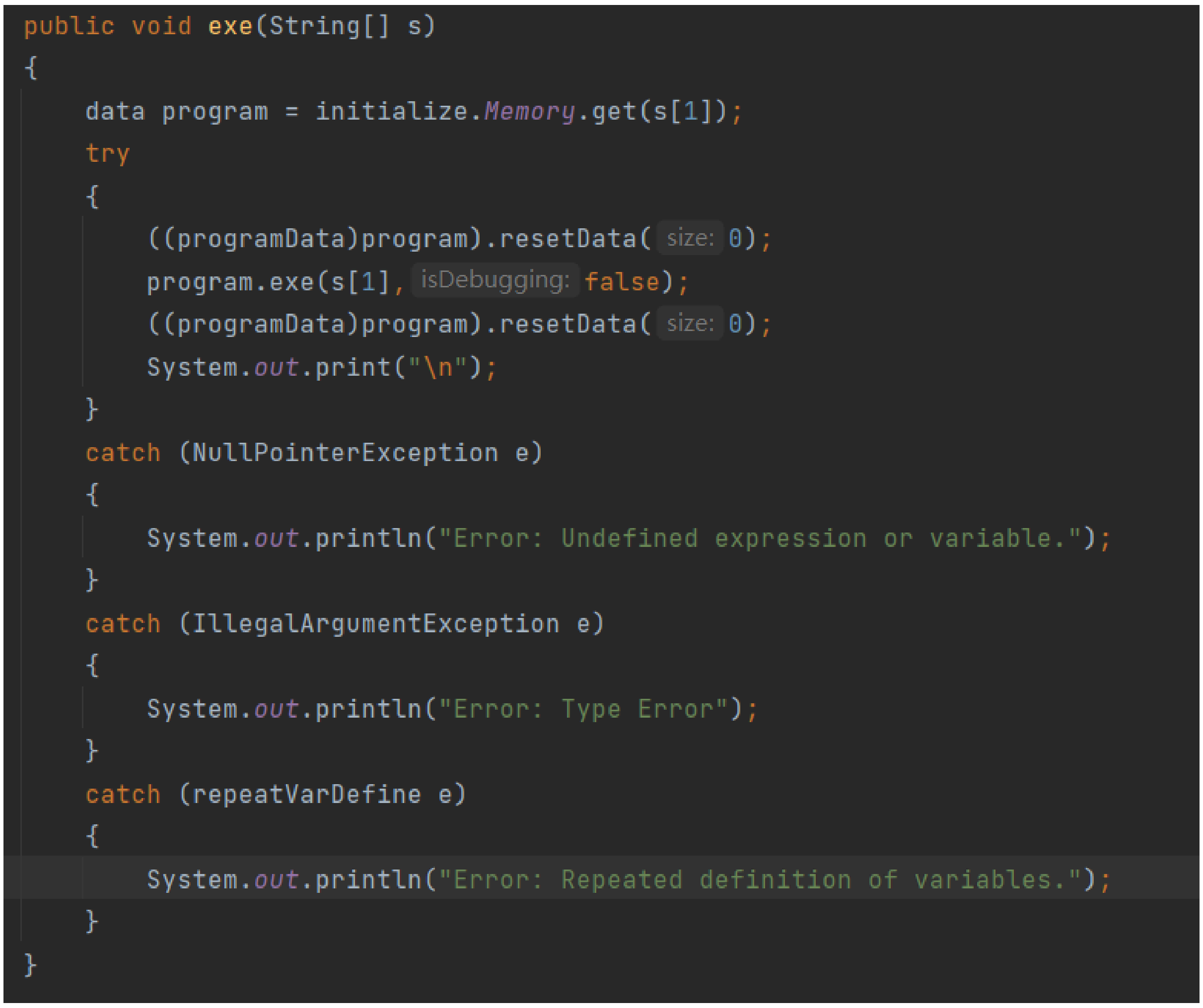
```java
/**
 * test if the label is defined
 * @param lab : label
 * @param key : the type of label
 * @return true/false
 */
static boolean definedError(String lab, String key)
{
    data x = Memory.get(lab);
    if(x != null)
    {
        System.out.printf("Error: The %s has been defined.\n",key);
        return true;
    }
    return false;
}
```

3. boolean undefinedError(String lab, String key)

   It’s similar to the last one. But it checks if the label is undefined.

- Errors during execution

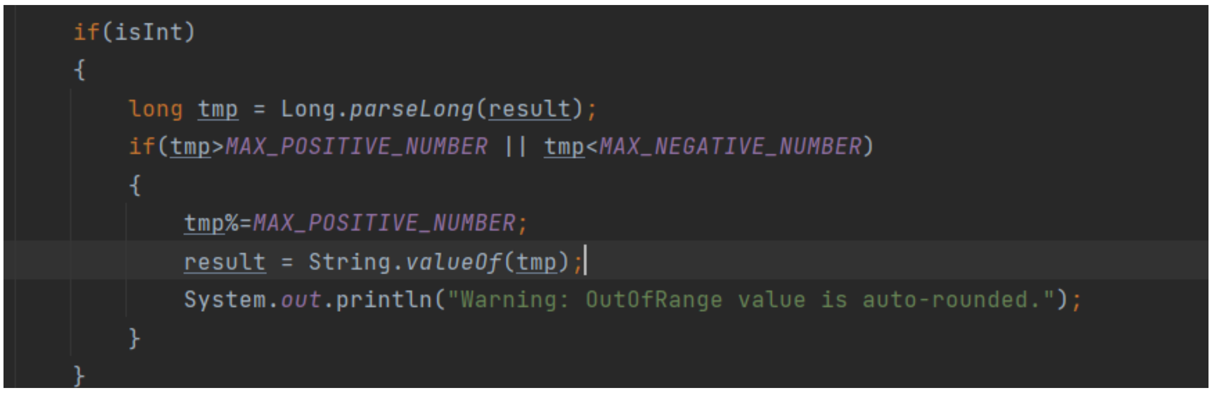
  This part lists the errors found during execution. The errors are checked by throwing exceptions in exe method. And for pointing out specific errors, I defined my own exceptions (eg. repeatVarDefine, tillBreakPoint) in which tillBreakPoint exception is only used to stop the program.



```java
public void exe(String[] s)
{
    data program = initialize.Memory.get(s[1]);
    try
    {
        ((programData)program).resetData( size: 0);
        program.exe(s[1], isDebugging: false);
        ((programData)program).resetData( size: 0);
        System.out.print("\n");
    }
    catch (NullPointerException e)
    {
        System.out.println("Error: Undefined expression or variable.");
    }
    catch (IllegalArgumentException e)
    {
        System.out.println("Error: Type Error");
    }
    catch (repeatVarDefine e)
    {
        System.out.println("Error: Repeated definition of variables.");
    }
}
```

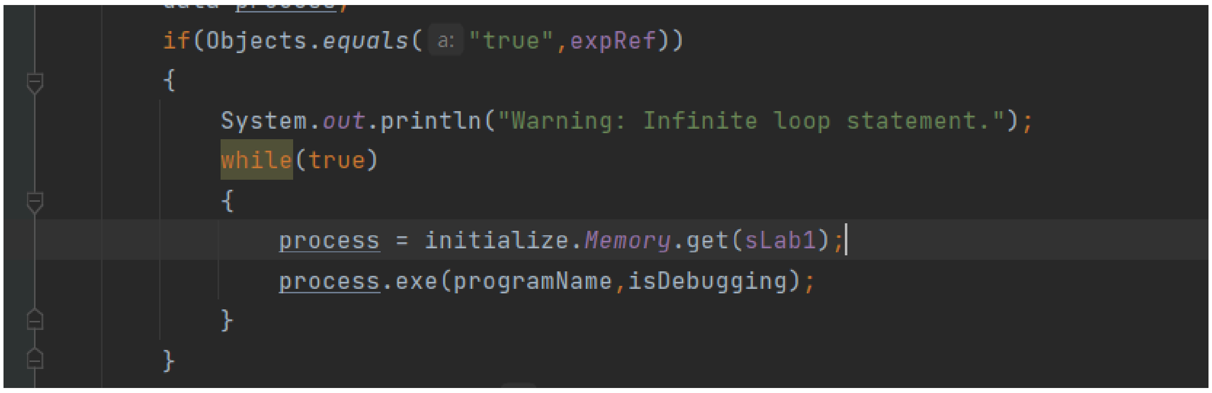
2.1.4 Warning Feedback

- Int value out of range [-99999,99999] and it will be auto-rounded by mod 99999



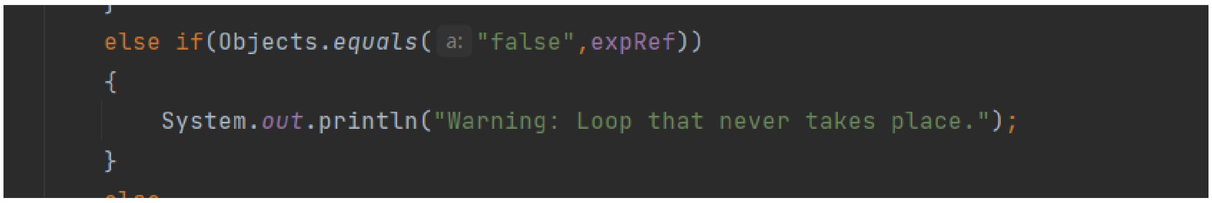
```
if(isInt)
{
    long tmp = Long.parseLong(result);
    if(tmp>MAX_POSITIVE_NUMBER || tmp<MAX_NEGATIVE_NUMBER)
    {
        tmp%=MAX_POSITIVE_NUMBER;
        result = String.valueOf(tmp);
        System.out.println("Warning: OutOfRange value is auto-rounded.");
    }
}
```

- Infinite loop statement eg. while true



```
if(Objects.equals( a: "true",expRef))
{
    System.out.println("Warning: Infinite loop statement.");
    while(true)
    {
        process = initialize.Memory.get(sLab1);
        process.exe(programName,isDebugging);
    }
}
```

- Loop that never takes place eg. while false



```
else if(Objects.equals( a: "false",expRef))
{
    System.out.println("Warning: Loop that never takes place.");
}
```

2.2 Requirements

2.2.1 REQ1 vardef lab typ varName expRef

1. **The requirement is implemented.**
2. **Implementation detials**
   - **Define Command (class vardef implements initialize)**

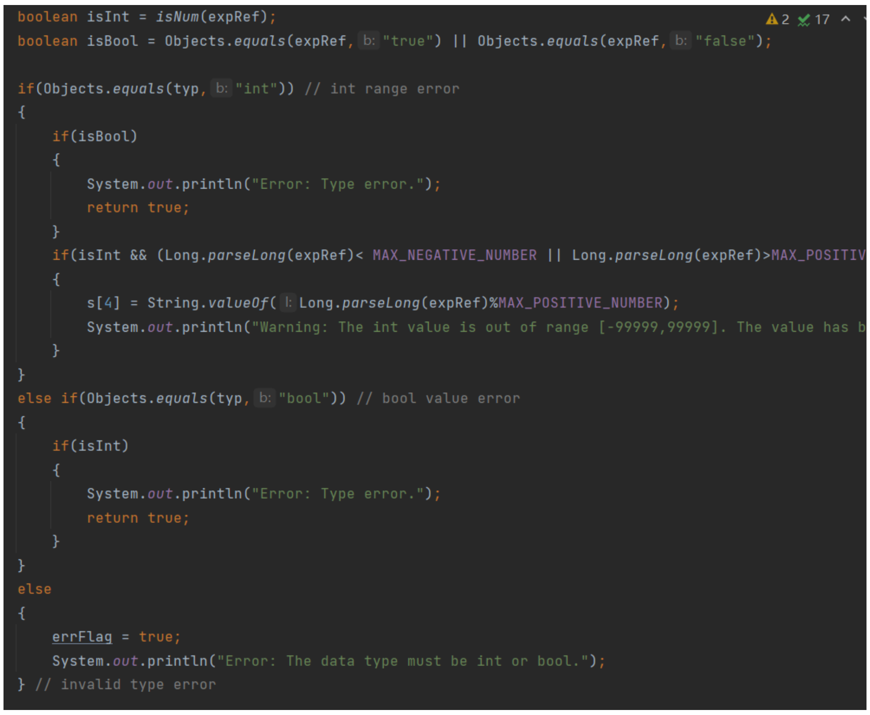
     If there are no **errors during initialize**. Define a new instance of **vardefData** and put it in **Memory** with the varName as the key and **vardefData** as the value.

   - **Store Command (class vardefData implements data)**

     Define lab, typ, varName and expRef in private fields and use a constructor to initialize their values. This can store all the data in the command.

3. **Error conditions**
   - **Errors during initialize**

     It will check the length of the command string array split by space to make sure the command format is right. And call the following method in **initialize: nameError** to check if lab and varName are legal, **definedError** to make sure lab and varName are undefined before, **undefinedError** to make sure expRef has been defined if it's not a number or "true" or "false". Besides, it will check if there is an invalid type. And test if the type of the variable and the input data have the same type. When the int value is out of range, it will give a warning and auto-round it.



```
boolean isInt = isNum(expRef);
boolean isBool = Objects.equals(expRef, b: "true") || Objects.equals(expRef, b: "false");

if(Objects.equals(typ, b: "int")) // int range error
{
    if(isBool)
    {
        System.out.println("Error: Type error.");
        return true;
    }
    if(isInt && (Long.parseLong(expRef)< MAX_NEGATIVE_NUMBER || Long.parseLong(expRef)>MAX_POSITIV
    {
        s[4] = String.valueOf( l: Long.parseLong(expRef)%MAX_POSITIVE_NUMBER);
        System.out.println("Warning: The int value is out of range [-99999,99999]. The value has b
    }
}
else if(Objects.equals(typ, b: "bool")) // bool value error
{
    if(isInt)
    {
        System.out.println("Error: Type error.");
        return true;
    }
}
else
{
    errFlag = true;
    System.out.println("Error: The data type must be int or bool.");
} // invalid type error
```

2.2.2 REQ2 binexpr expName expRef1 bop expRef2

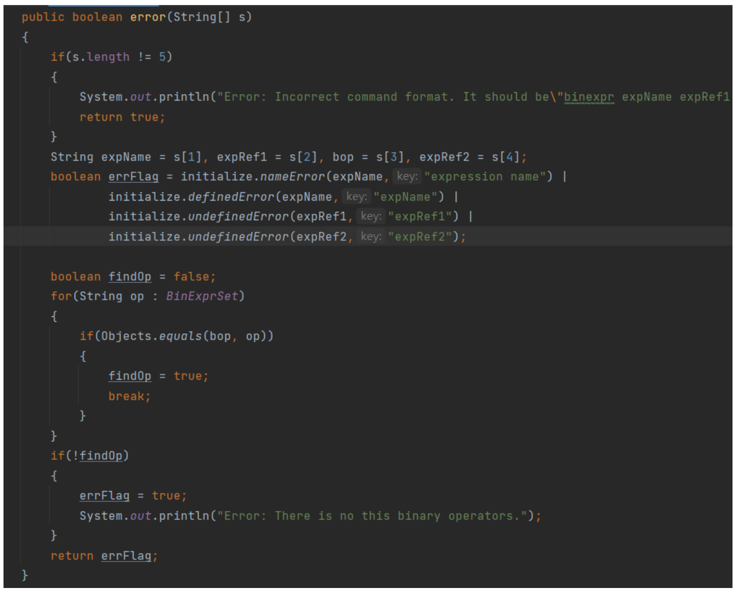
1. **The requirement is implemented.**
2. **Implementation detials**
    - **Define Command (class binexpr implements initialize)**

      If there are no **errors during initialize**. Define a new instance of **binexprData** and put it in **Memory** with expName as the key and **binexprData** as the value.

    - **Store Command (class binexprData implements data)**

      Define expName, expRef1, bop and expRef2 in private fields and use a constructor to initialize their values. This can store all the data in the command.
3. **Error conditions**
    - **Errors during initialize**

      It will check the length of the command string array split by space to make sure the command format is right. And call the following method in **initialize:** **nameError** to check if expName is legal, **definedError** to make sure expName is undefined before, **undefinedError** to make sure expRef1 and expRef2 have been defined if they are not a number or “true” or “false”. In addition, there is a BinExprSet storing all the legal binary operators. It will be used to test if the input operator is illegal.



```
public boolean error(String[] s)
{
    if(s.length != 5)
    {
        System.out.println("Error: Incorrect command format. It should be\"binexpr expName expRef1
        return true;
    }
    String expName = s[1], expRef1 = s[2], bop = s[3], expRef2 = s[4];
    boolean errFlag = initialize.nameError(expName, key: "expression name") |
            initialize.definedError(expName, key: "expName") |
            initialize.undefinedError(expRef1, key: "expRef1") |
            initialize.undefinedError(expRef2, key: "expRef2");

    boolean findOp = false;
    for(String op : BinExprSet)
    {
        if(Objects.equals(bop, op))
        {
            findOp = true;
            break;
        }
    }
    if(!findOp)
    {
        errFlag = true;
        System.out.println("Error: There is no this binary operators.");
    }
    return errFlag;
}
```

2.2.3 REQ3 unexpr expName uop expRef1

1. **The requirement is implemented.**
2. **Implementation detials**
    - **Define Command (class unexpr implements initialize)**

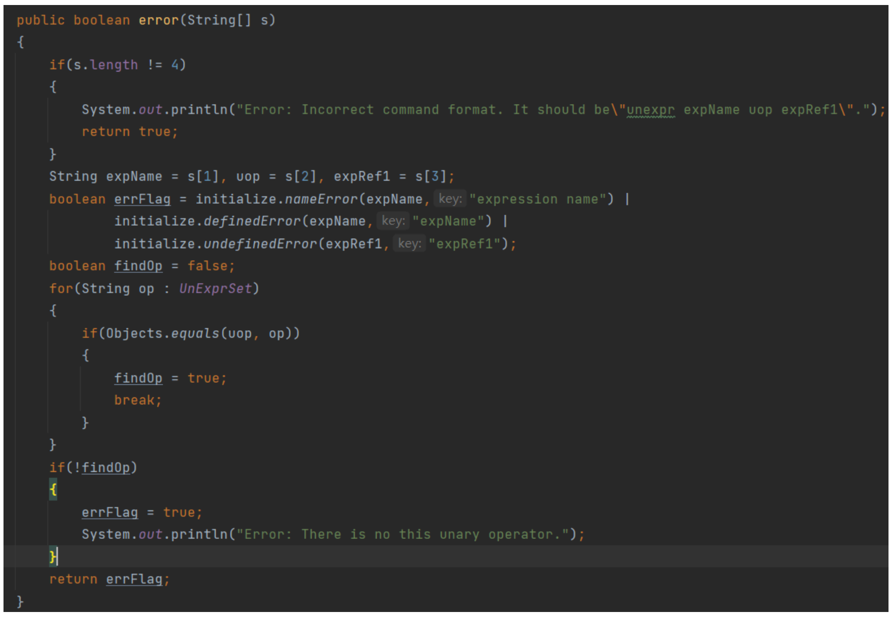
      If there are no **errors during initialize**. Define a new instance of **unexprData** and put it in **Memory** with expName as the key and **unexprData** as the value.

    - **Store Command (class unexprData implements data)**

      Define expName, uop and expRef1 in private fields and use a constructor to initialize their values. This can store all the data in the command.

3. **Error conditions**
    - **Errors during initialize**

      It will check the length of the command string array split by space to make sure the command format is right. And call the following method in **initialize:** **nameError** to check if expName is legal, **definedError** to make sure expName is undefined before, **undefinedError** to make sure expRef1 has been defined if it’s not a number or “true” or “false”. Besides, there is a UnExprSet storing all the legal unary operators. It will be used to test if the input operator is illegal.



```
public boolean error(String[] s)
{
    if(s.length != 4)
    {
        System.out.println("Error: Incorrect command format. It should be\"unexpr expName uop expRef1\".");
        return true;
    }
    String expName = s[1], uop = s[2], expRef1 = s[3];
    boolean errFlag = initialize.nameError(expName, key: "expression name") |
            initialize.definedError(expName, key: "expName") |
            initialize.undefinedError(expRef1, key: "expRef1");
    boolean findOp = false;
    for(String op : UnExprSet)
    {
        if(Objects.equals(uop, op))
        {
            findOp = true;
            break;
        }
    }
    if(!findOp)
    {
        errFlag = true;
        System.out.println("Error: There is no this unary operator.");
    }
    return errFlag;
}
```

2.2.4 REQ4 assign lab varName expRef

1. **The requirement is implemented.**
2. **Implementation detials**
    - **Define Command (class assign implements initialize)**

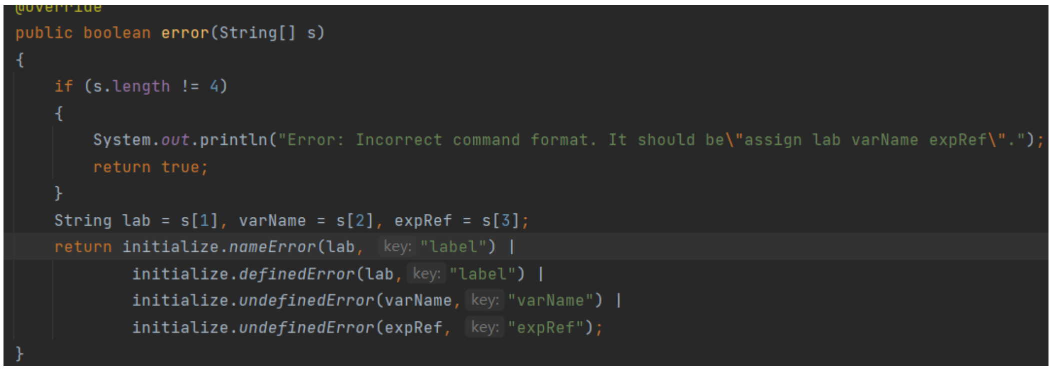
      If there are no **errors during initialize**. Define a new instance of **assignData** and put it in **Memory** with lab as the key and **assignData** as the value.

    - **Store Command (class assignData implements data)**

      Define lab, varName and expRef in private fields and use a constructor to initialize their values. This can store all the data in the command.

3. **Error conditions**
    - **Errors during initialize**

      It will check the length of the command string array split by space to make sure the command format is right. And call the following method in **initialize:** **nameError** to check if lab is legal, **definedError** to make sure lab is undefined before, **undefinedError** to make sure varName and expRef have been defined if they are not a number or “true” or “false”.



```
@Override
public boolean error(String[] s)
{
    if (s.length != 4)
    {
        System.out.println("Error: Incorrect command format. It should be\"assign lab varName expRef\".");
        return true;
    }
    String lab = s[1], varName = s[2], expRef = s[3];
    return initialize.nameError(lab,  key: "label") ||
            initialize.definedError(lab, key: "label") ||
            initialize.undefinedError(varName, key: "varName") ||
            initialize.undefinedError(expRef,  key: "expRef");
}
```

2.2.5 REQ5 print lab expRef

1. **The requirement is implemented.**
2. **Implementation detials**
    - **Define Command (class print implements initialize)**

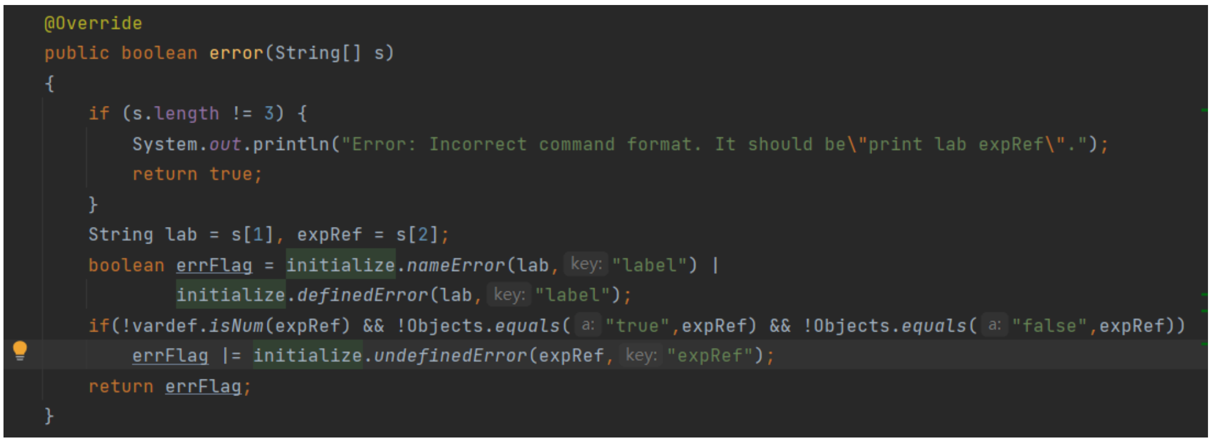
      If there are no **errors during initialize**. Define a new instance of **printData** and put it in **Memory** with lab as the key and **printData** as the value.

    - **Store Command (class printData implements data)**

      Define lab and expRef in private fields and use a constructor to initialize their values. This can store all the data in the command.

3. **Error conditions**
    - **Errors during initialize**

      It will check the length of the command string array split by space to make sure the command format is right. And call the following method in **initialize: nameError** to check if lab is legal, **definedError** to make sure lab is undefined before, **undefinedError** to make sure expRef has been defined if it’s not a number or “true” or “false”. Besides, it will check if the expRef is a number or “true” or “false”. If it’s none of them, it will call the check-error functions with <u>expRef</u> as the parameter to test if the name is legal and undefined.



```
@Override
public boolean error(String[] s)
{
    if (s.length != 3) {
        System.out.println("Error: Incorrect command format. It should be\"print lab expRef\".");
        return true;
    }
    String lab = s[1], expRef = s[2];
    boolean errFlag = initialize.nameError(lab, key: "label") |
            initialize.definedError(lab, key: "label");
    if(!vardef.isNum(expRef) && !Objects.equals( a: "true",expRef) && !Objects.equals( a: "false",expRef))
        errFlag |= initialize.undefinedError(expRef, key: "expRef");
    return errFlag;
}
```

2.2.6 REQ6 skip lab

1. **The requirement is implemented.**
2. **Implementation detials**
   - **Define Command (class skip implements initialize)**

     If there are no **errors during initialize**. Define a new instance of **printData** and put it in **Memory** with <u>lab</u> as the key and **printData** as the value.

   - **Store Command (class skipData implements data)**

     Define <u>lab</u> in private fields and use a constructor to initialize it.

3. **Error conditions**
   - **Errors during initialize**

     It will check the length of the command string array split by space to make sure the command format is right. And call the following method in **initialize: nameError** to check if <u>lab</u> is legal, **definedError** to make sure <u>lab</u> is undefined before.

2.2.7 REQ7 block lab statementLab1 ... statementLabn

1. **The requirement is implemented.**
2. **Implementation detials**

- **Define Command (class block implements initialize)**

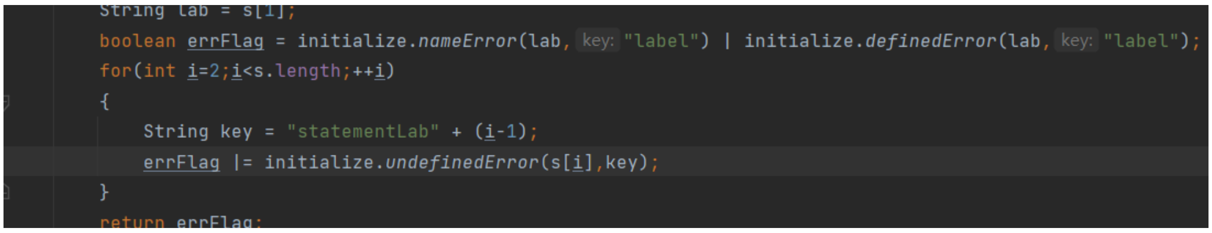
  If there are no **errors during initialize**. Define a new instance of **blockData** and put it in **Memory** with lab as the key and **blockData** as the value.

- **Store Command (class blockData implements data)**

  Define lab and a string list which include all statementLab in private fields and use a constructor to initialize their values. This can store all the data in the command.

3. **Error conditions**
   - **Errors during initialize**

     It does not check the length of the array of command strings separated by spaces because the number of statementLabs is not limited, which is what makes it different from other commands. And call the following method in **initialize: nameError** to check if lab is legal, **definedError** to make sure lab is undefined before. Also for every statementLabs, call the following method in **initialize: definedError** to make sure lab is defined before.



```
String lab = s[1];
boolean errFlag = initialize.nameError(lab, key: "label") | initialize.definedError(lab, key: "label");
for(int i=2;i<s.length;++i)
{
    String key = "statementLab" + (i-1);
    errFlag |= initialize.undefinedError(s[i],key);
}
return errFlag;
```

2.2.8 REQ8 if lab expRef statementLab1 statementLab2

1. **The requirement is implemented.**
2. **Implementation detials**
   - **Define Command (class condition implements initialize)**

     If there are no **errors during initialize**. Define a new instance of **conditionData** and put it in **Memory** with lab as the key and **conditionData** as the value.

   - **Store Command (class conditionData implements data)**

Define lab, expRef, sLab1 and sLab2 in private fields and use a constructor to initialize their values. This can store all the data in the command.

3. **Error conditions**
   - **Errors during initialize**

     It will check the length of the command string array split by space to make sure the command format is right. And call the following method in **initialize: nameError** to check if lab, expRef, sLab1 and sLab2 is legal, **definedError** to make sure lab is undefined before and **undefinedError** to make sure sLab1 and sLab2 is defined before. Also invoke **undefinedError** to make sure expRef has been defined if it's not a number or "true" or "false".

2.2.9 REQ9 while lab expRef statementLab1

1. **The requirement is implemented.**
2. **Implementation detials**
   - **Define Command (class loop implements initialize)**

     If there are no **errors during initialize**. Define a new instance of **loopData** and put it in **Memory** with lab as the key and **loopData** as the value.

   - **Store Command (class loopData implements data)**

     Define lab, expRef and sLab1 in private fields and use a constructor to initialize their values. This can store all the data in the command.

3. **Error conditions**
   - **Errors during initialize**

     It will check the length of the command string array split by space to make sure the command format is right. And call the following method in **initialize: nameError** to check if lab, expRef and sLab1 is legal, **definedError** to make sure lab is undefined before and **undefinedError** to make sure sLab1 is defined before. Also invoke **undefinedError** to make sure expRef has been defined if it's not a number or "true" or "false".

2.2.10 REQ10 program programName statementLab

1. **The requirement is implemented.**
2. **Implementation detials**
    - **Define Command (class program implements initialize)**

      If there are no **errors during initialize**. Define a new instance of **programData** and put it in **Memory** with programName as the key and **programData** as the value.

    - **Store Command (class programData implements data)**

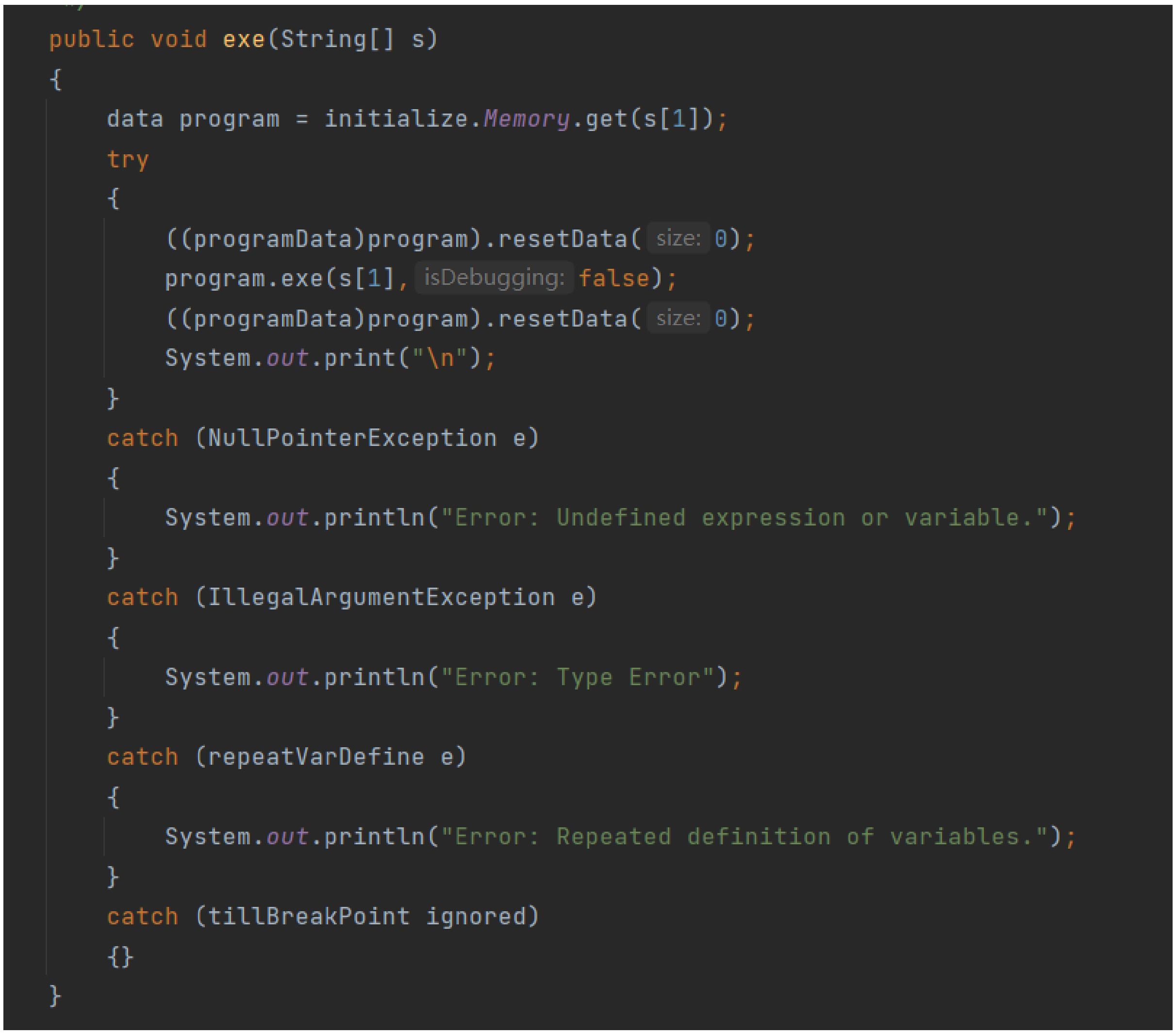
      Define programName and statementLab in private fields and use a constructor to initialize their values. This can store all the data in the command.

3. **Error conditions**
    - **Errors during initialize**

      It will check the length of the command string array split by space to make sure the command format is right. And call the following method in **initialize:** **nameError** to check if programName is legal, and **undefinedError** to make sure statementLab is defined before.

2.2.11 REQ11 execute programName

- **The requirement is implemented.**
- **Implementation detials**

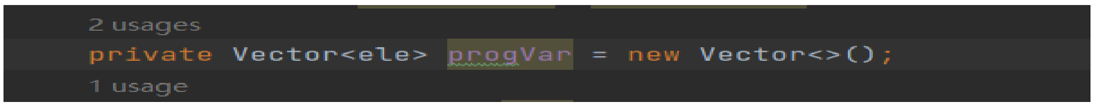


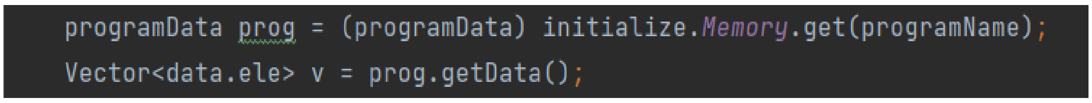
```java
public void exe(String[] s)
{
    data program = initialize.Memory.get(s[1]);
    try
    {
        ((programData)program).resetData( size: 0);
        program.exe(s[1], isDebugging: false);
        ((programData)program).resetData( size: 0);
        System.out.print("\n");
    }
    catch (NullPointerException e)
    {
        System.out.println("Error: Undefined expression or variable.");
    }
    catch (IllegalArgumentException e)
    {
        System.out.println("Error: Type Error");
    }
    catch (repeatVarDefine e)
    {
        System.out.println("Error: Repeated definition of variables.");
    }
    catch (tillBreakPoint ignored)
    {}
}
```

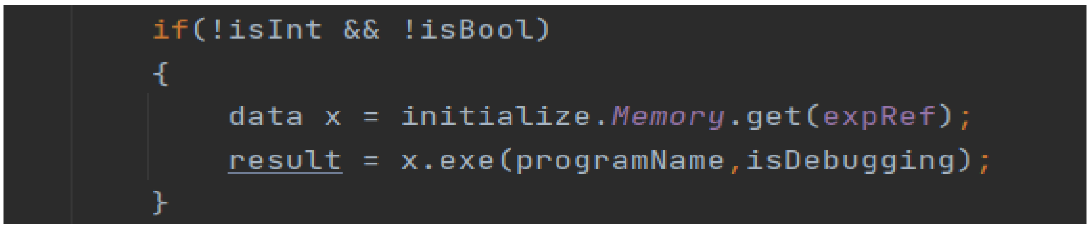
It will get the **programData** from **Memory** by using programName as the key. And run the exe method in this **programData**. By using the dynamic binding of **data** interface, when we meet a label(if it's not a number or "true" or "false", it will be a label or varName or expName), we can directly run its exe method to execute it and get the result. The whole execution process is implemented recursively. Besides, as for the overall design of exe method, please check 2.1.2.1. Before or after each **execution**, we reset the variable date vector to clear all the variables in it. The following are the specific exe methods for corresponding commands.

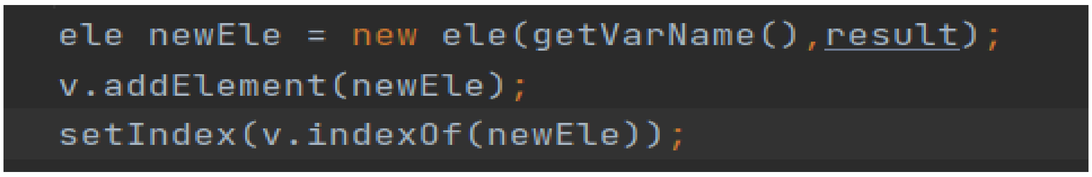
- **execution of vardefData.exe**

  There is a private field index storing the index of this variable in the vector which stores all the variables in this program. And the initial value of it is -1. So, if the value of index equals to -1, a new variable should be defined. It is worth emphasizing that only variables that are put into the vector, which is progVar (See details in REQ10), are defined. To define a variable, a newEle(an instance of **data.ele**) which stores the varName and its val is defined and put in progVar. If the expRef is a label, we should execute the **data** with this label first. The exe method has a return String value. If some command(eg. skip) is no need to return a value, it will return an empty String "".


```
2 usages
private Vector<ele> progVar = new Vector<>();
1 usage
```

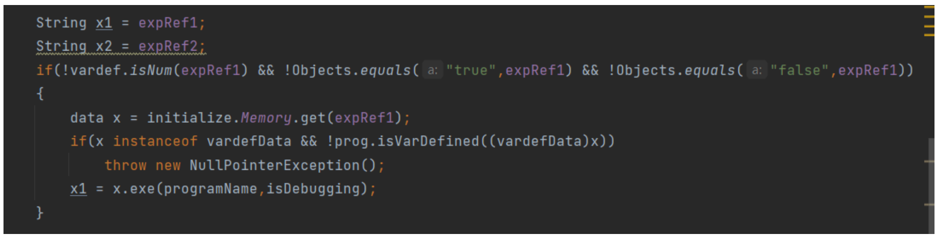

```
programData prog = (programData) initialize.Memory.get(programName);
Vector<data.ele> v = prog.getData();
```


```
if(!isInt && !isBool)
{
    data x = initialize.Memory.get(expRef);
    result = x.exe(programName,isDebugging);
}
```

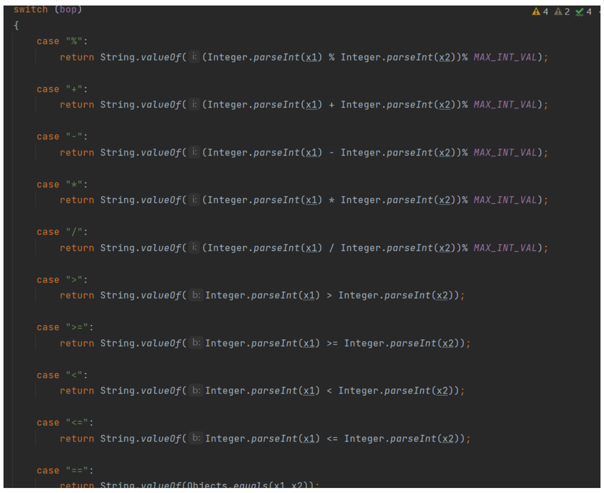

```
ele newEle = new ele(getVarName(),result);
v.addElement(newEle);
setIndex(v.indexOf(newEle));
```

- **execution of binexprData.exe**

If expRef1 is not a number or "true" or "false", it's a label and we should execute the **data** with expRef1 as a key in **Memory** first. As for expRef2 it's the same. After those results are obtained, the binary operator will be used to get the result of this command.



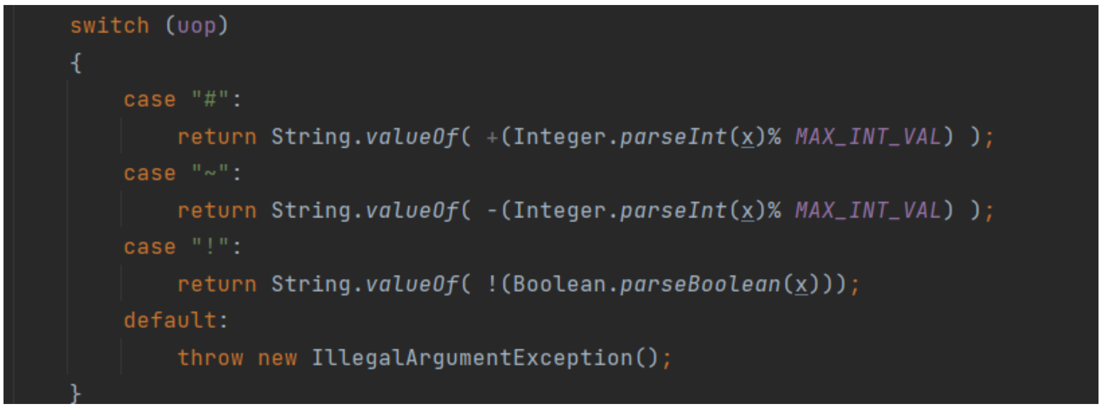
```
String x1 = expRef1;
String x2 = expRef2;
if(!vardef.isNum(expRef1) && !Objects.equals( a: "true",expRef1) && !Objects.equals( a: "false",expRef1))
{
    data x = initialize.Memory.get(expRef1);
    if(x instanceof vardefData && !prog.isVarDefined((vardefData)x))
        throw new NullPointerException();
    x1 = x.exe(programName,isDebugging);
}
```



```
switch (bop)
{
    case "%":
        return String.valueOf( i: (Integer.parseInt(x1) % Integer.parseInt(x2))% MAX_INT_VAL);

    case "+":
        return String.valueOf( i: (Integer.parseInt(x1) + Integer.parseInt(x2))% MAX_INT_VAL);

    case "-":
        return String.valueOf( i: (Integer.parseInt(x1) - Integer.parseInt(x2))% MAX_INT_VAL);

    case "*":
        return String.valueOf( i: (Integer.parseInt(x1) * Integer.parseInt(x2))% MAX_INT_VAL);

    case "/":
        return String.valueOf( i: (Integer.parseInt(x1) / Integer.parseInt(x2))% MAX_INT_VAL);

    case ">":
        return String.valueOf( b: Integer.parseInt(x1) > Integer.parseInt(x2));

    case ">=":
        return String.valueOf( b: Integer.parseInt(x1) >= Integer.parseInt(x2));

    case "<":
        return String.valueOf( b: Integer.parseInt(x1) < Integer.parseInt(x2));

    case "<=":
        return String.valueOf( b: Integer.parseInt(x1) <= Integer.parseInt(x2));

    case "==":
        return String.valueOf(Objects.equals(x1,x2));
```

- **execution of unexprData.exe**

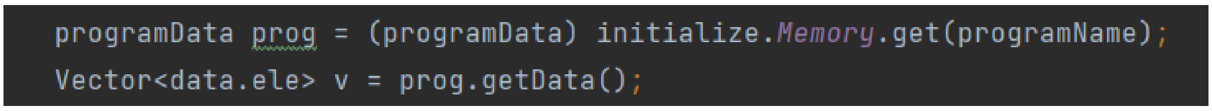
  This is similar to the last one.



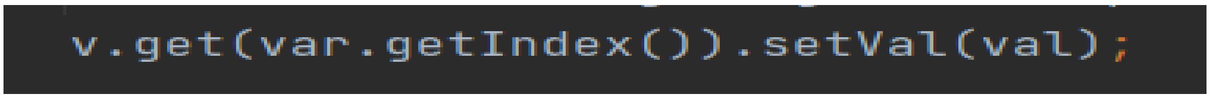
```
switch (uop)
{
    case "#":
        return String.valueOf( +(Integer.parseInt(x)% MAX_INT_VAL) );
    case "~":
        return String.valueOf( -(Integer.parseInt(x)% MAX_INT_VAL) );
    case "!":
        return String.valueOf( !(Boolean.parseBoolean(x)));
    default:
        throw new IllegalArgumentException();
}
```

- **assignData.exe**

  Firstly, **progVar** can be obtained by programName by **Memory**. The instance of **vardefData** which contains the data of this variable can be obtained by getIndex() and index it from **progVar**. If expRef is number or “true” or “false” and the type is the same as the variable, it will be set to the value of the variable and auto-round it if available. Otherwise, the value of expRef should be obtained by executing it first.



```
programData prog = (programData) initialize.Memory.get(programName);
Vector<data.ele> v = prog.getData();
```



```
v.get(var.getIndex()).setVal(val);
```

- **execution of printData.exe**

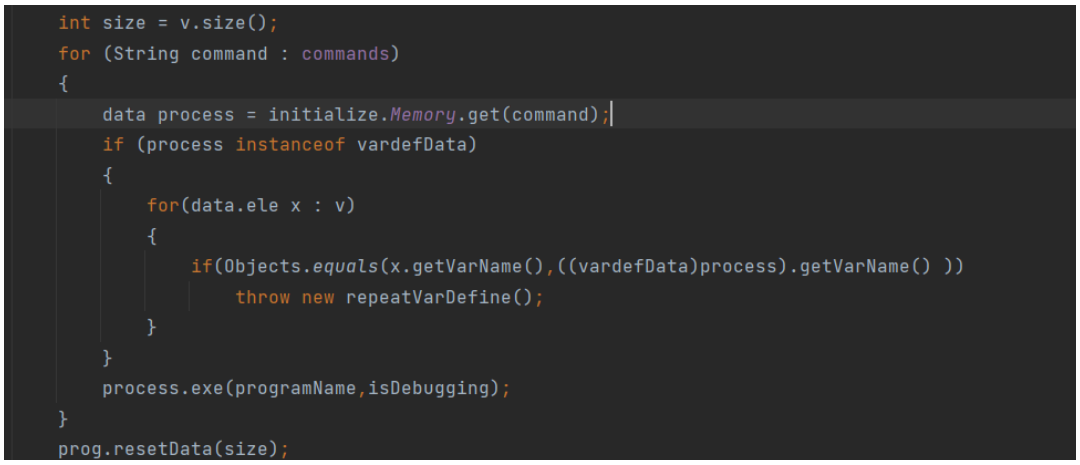
  The **commandData** with expRef should be executed first, if it's not a number or "true" or "false". Then print it.

- **execution of skipData.exe**

  Just skip and return an empty String "".

- **execution of blockData.exe**

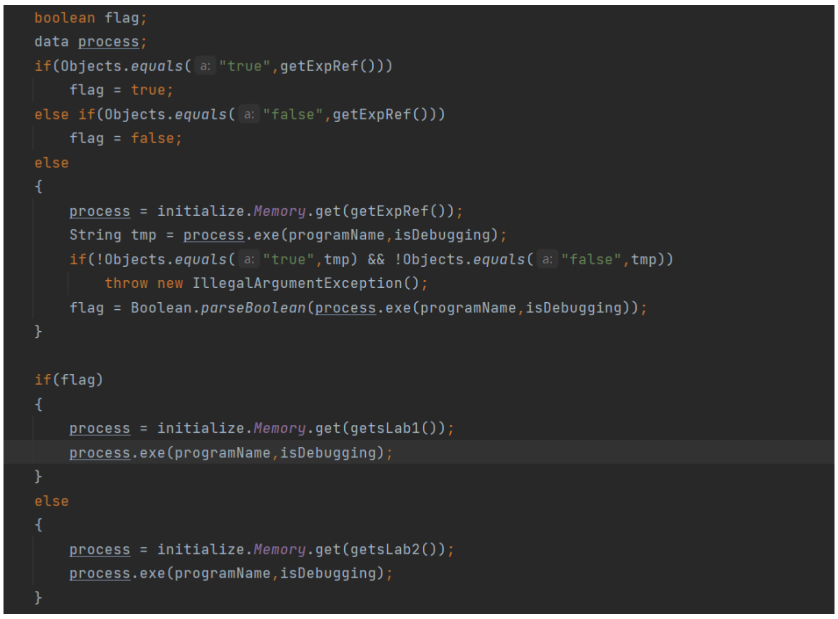
  In this method, all the labels are executed in order by recursion. When you quit this block, the variable defined in this block will be removed. These variables are the local variables in this block.



```
int size = v.size();
for (String command : commands)
{
    data process = initialize.Memory.get(command);
    if (process instanceof vardefData)
    {
        for(data.ele x : v)
        {
            if(Objects.equals(x.getVarName(),((vardefData)process).getVarName() ))
                throw new repeatVarDefine();
        }
    }
    process.exe(programName,isDebugging);
}
prog.resetData(size);
```

- **execution of conditionData.exe**

If expRef is not “true” or “false”, the **commandData** with it should be executed first. The String return value will be changed to boolean type by Boolean.parseBoolean(). If it’s true, the **commandData** with statementLab1 will be executed. Otherwise, the **commandData** with statementLab2 will be executed.

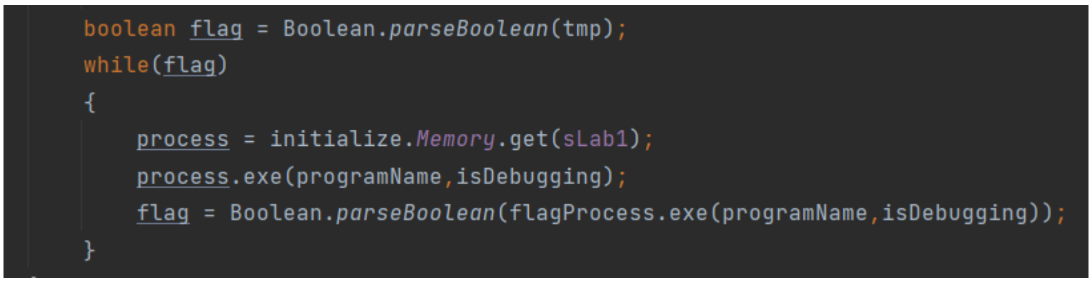


```
boolean flag;
data process;
if(Objects.equals( a: "true",getExpRef()))
    flag = true;
else if(Objects.equals( a: "false",getExpRef()))
    flag = false;
else
{
    process = initialize.Memory.get(getExpRef());
    String tmp = process.exe(programName,isDebugging);
    if(!Objects.equals( a: "true",tmp) && !Objects.equals( a: "false",tmp))
        throw new IllegalArgumentException();
    flag = Boolean.parseBoolean(process.exe(programName,isDebugging));
}

if(flag)
{
    process = initialize.Memory.get(getsLab1());
    process.exe(programName,isDebugging);
}
else
{
    process = initialize.Memory.get(getsLab2());
    process.exe(programName,isDebugging);
}
```

\

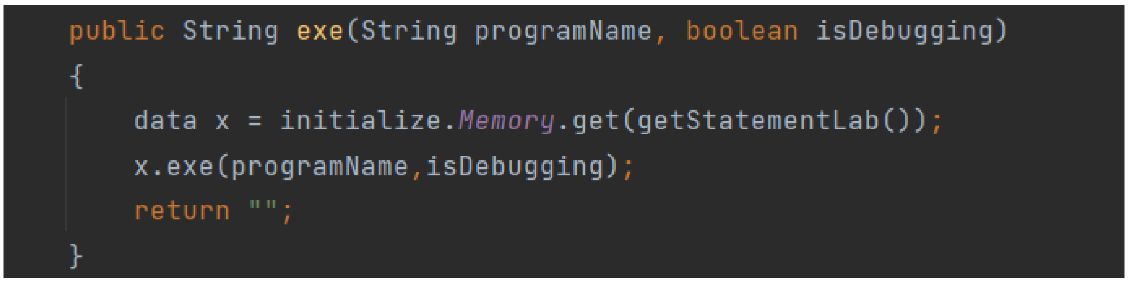
- **execution of loopData.exe**

If expRef is not “true” or “false”, the **commandData** with it should be executed first. The String return value will be changed to boolean type by Boolean.parseBoolean(). According to its value, the **commandData** with the label of statementLab1 is executed.



```
boolean flag = Boolean.parseBoolean(tmp);
while(flag)
{
    process = initialize.Memory.get(sLab1);
    process.exe(programName,isDebugging);
    flag = Boolean.parseBoolean(flagProcess.exe(programName,isDebugging));
}
```

- **execution of programData.exe**



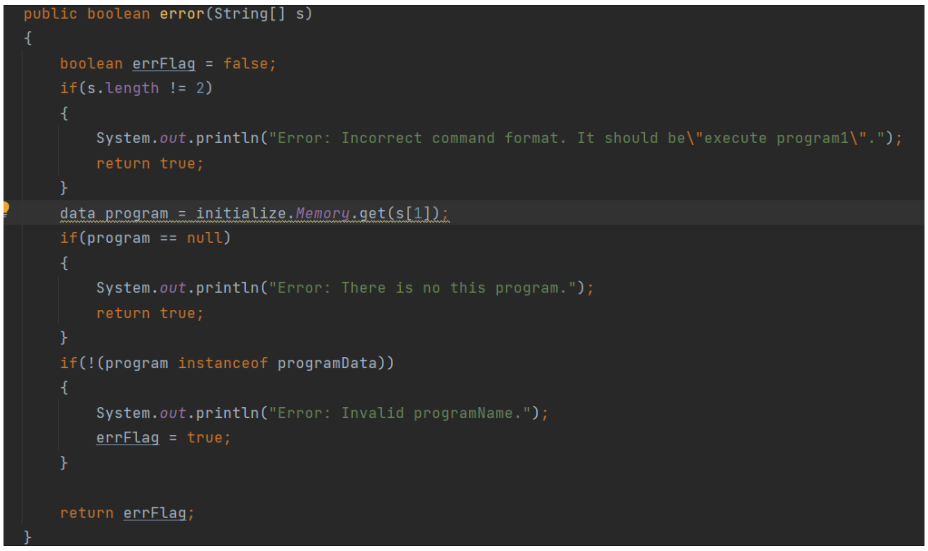
```java
public String exe(String programName, boolean isDebugging)
{
    data x = initialize.Memory.get(getStatementLab());
    x.exe(programName,isDebugging);
    return "";
}
```

Obtain the **commandData** with statemenLab through **Memory** and execute it.

1. **Error conditions**

   - **Errors during initialize**

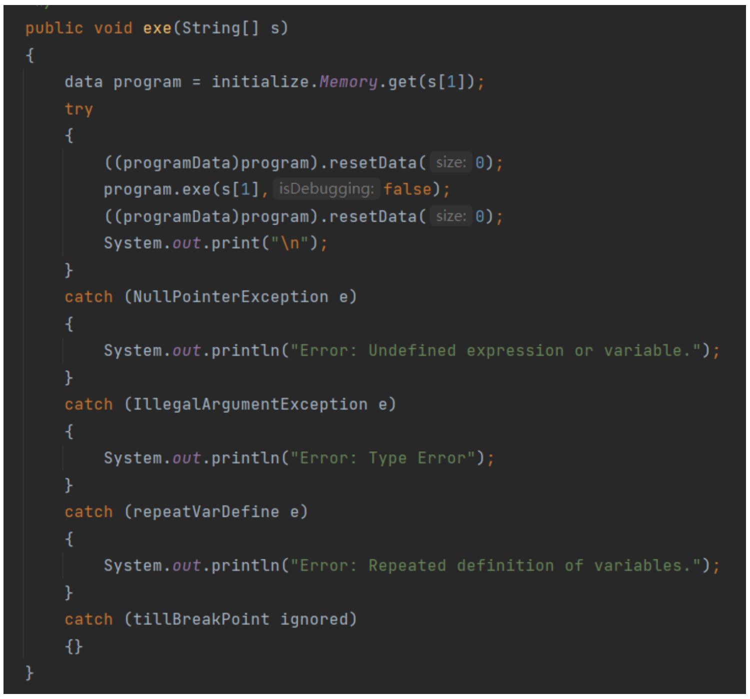
     It will check the length of the command string array split by space to make sure the command format is right. Through programName as a key, the **data(dynamic binding)** can be obtained through **Memory**. Then check if the implementation of **data** exists and if it's an instance of **programData**.



```java
public boolean error(String[] s)
{
    boolean errFlag = false;
    if(s.length != 2)
    {
        System.out.println("Error: Incorrect command format. It should be\"execute program1\".");
        return true;
    }
    data program = initialize.Memory.get(s[1]);
    if(program == null)
    {
        System.out.println("Error: There is no this program.");
        return true;
    }
    if(!(program instanceof programData))
    {
        System.out.println("Error: Invalid programName.");
        errFlag = true;
    }

    return errFlag;
}
```

   - **Errors during execution**

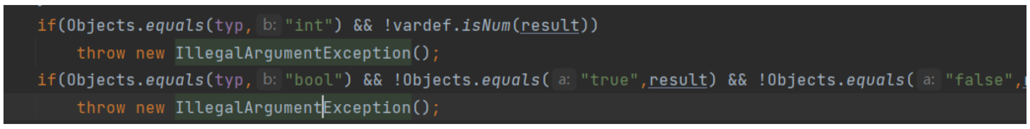
     See the overall design of this part from 2.1.3. The following are the specific errors corresponding to certain commands.



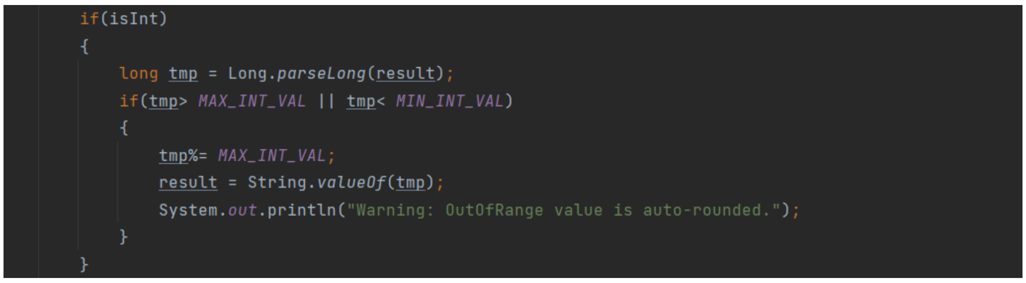
```java
public void exe(String[] s)
{
    data program = initialize.Memory.get(s[1]);
    try
    {
        ((programData)program).resetData( size: 0);
        program.exe(s[1], isDebugging: false);
        ((programData)program).resetData( size: 0);
        System.out.print("\n");
    }
    catch (NullPointerException e)
    {
        System.out.println("Error: Undefined expression or variable.");
    }
    catch (IllegalArgumentException e)
    {
        System.out.println("Error: Type Error");
    }
    catch (repeatVarDefine e)
    {
        System.out.println("Error: Repeated definition of variables.");
    }
    catch (tillBreakPoint ignored)
    {}
}
```

- **errors in vardefData.exe**

  If the type of the variable is different with the value of expRef, it will throw IllegalArgumentException which causes type error. If the int value is out of range, it will give a warning and round it.



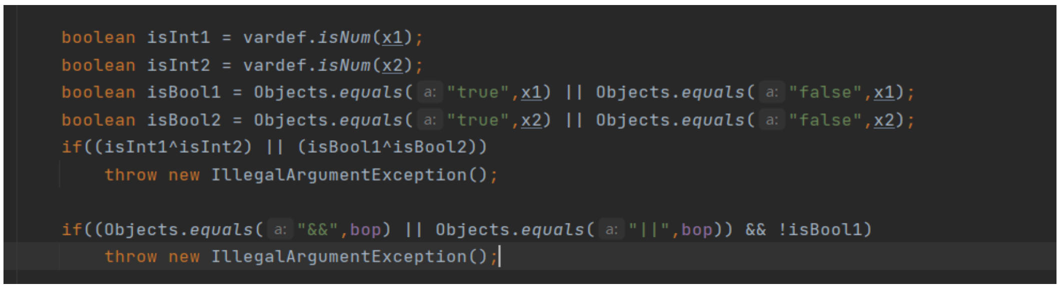
```java
if(Objects.equals(typ, b: "int") && !vardef.isNum(result))
    throw new IllegalArgumentException();
if(Objects.equals(typ, b: "bool") && !Objects.equals( a: "true",result) && !Objects.equals( a: "false",
    throw new IllegalArgumentException();
```



```java
if(isInt)
{
    long tmp = Long.parseLong(result);
    if(tmp> MAX_INT_VAL || tmp< MIN_INT_VAL)
    {
        tmp%= MAX_INT_VAL;
        result = String.valueOf(tmp);
        System.out.println("Warning: OutOfRange value is auto-rounded.");
    }
}
```

- **errors in binexprData.exe**

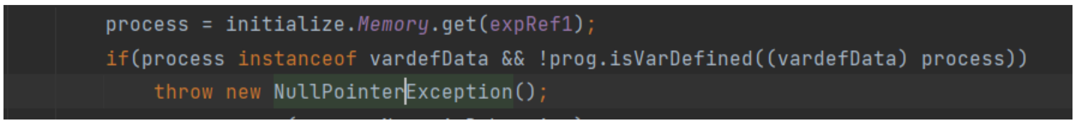
  If expRef1 or expRef2 is an instance of vardefData, but it's not defined, a NullPointerException will be thrown. If the data type of expRef1 and expRef2 are different, we throw IllegalArgumentException. Besides, if the binary operator does not match with expRef1 and expRef2, a IllegalArgumentException will also be thrown.



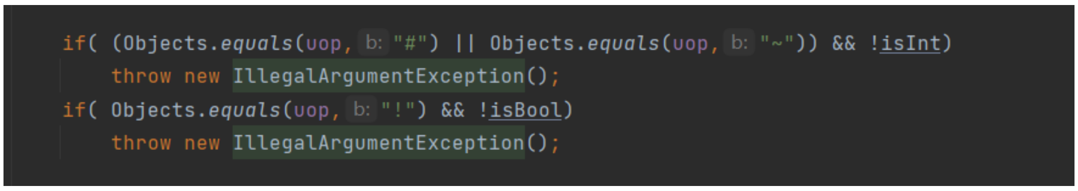
```
boolean isInt1 = vardef.isNum(x1);
boolean isInt2 = vardef.isNum(x2);
boolean isBool1 = Objects.equals( a: "true",x1) || Objects.equals( a: "false",x1);
boolean isBool2 = Objects.equals( a: "true",x2) || Objects.equals( a: "false",x2);
if((isInt1^isInt2) || (isBool1^isBool2))
    throw new IllegalArgumentException();

if((Objects.equals( a: "&&",bop) || Objects.equals( a: "||",bop)) && !isBool1)
    throw new IllegalArgumentException();
```

- **errors in unexprData.exe**

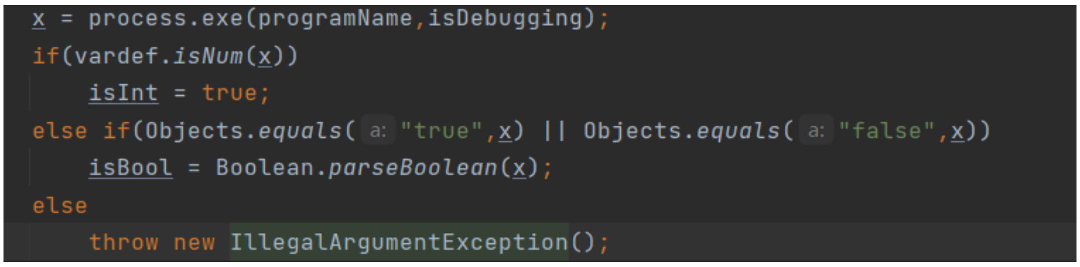
If expRef1 is an instance of vardefData, but it's not defined, a NullPointerException will be thrown. If the value of expRef1 is neither a number nor boolean, an IllegalArgumentException is thrown. And if the data type of expRef1 does not match the unary operator, an IllegalArgumentException will be thrown.



```
process = initialize.Memory.get(expRef1);
if(process instanceof vardefData && !prog.isVarDefined((vardefData) process))
    throw new NullPointerException();
```



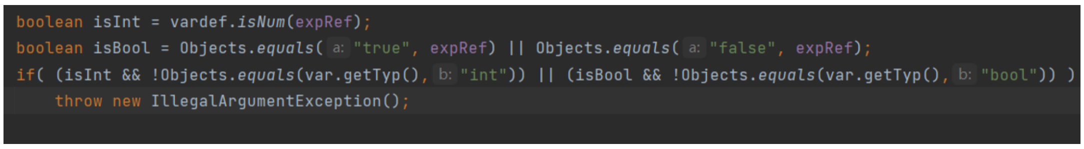
```
if( (Objects.equals(uop, b: "#") || Objects.equals(uop, b: "~")) && !isInt)
    throw new IllegalArgumentException();
if( Objects.equals(uop, b: "!") && !isBool)
    throw new IllegalArgumentException();
```



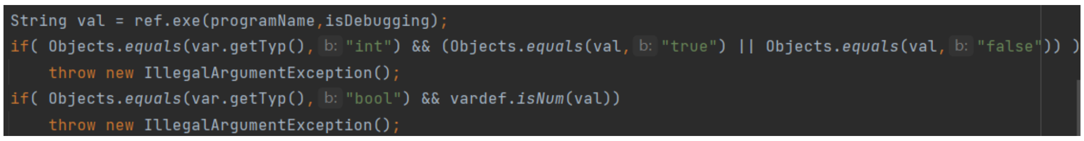
```
x = process.exe(programName,isDebugging);
if(vardef.isNum(x))
    isInt = true;
else if(Objects.equals( a: "true",x) || Objects.equals( a: "false",x))
    isBool = Boolean.parseBoolean(x);
else
    throw new IllegalArgumentException();
```

- **errors in assignData.exe**

If the type of the value of expRef does not match the type of this variable, an IllegalArgumentException will be thrown.



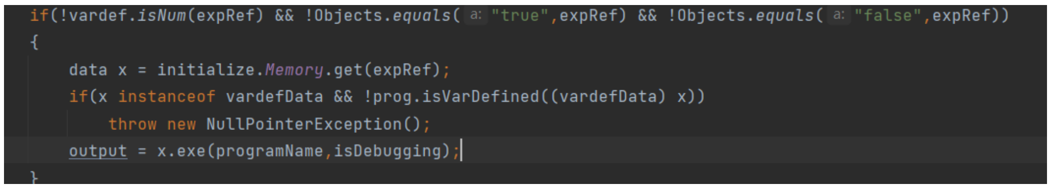
```
boolean isInt = vardef.isNum(expRef);
boolean isBool = Objects.equals( "true", expRef) || Objects.equals( "false", expRef);
if( (isInt && !Objects.equals(var.getTyp(), "int")) || (isBool && !Objects.equals(var.getTyp(), "bool")) )
    throw new IllegalArgumentException();
```



```
String val = ref.exe(programName,isDebugging);
if( Objects.equals(var.getTyp(), "int") && (Objects.equals(val, "true") || Objects.equals(val, "false")) )
    throw new IllegalArgumentException();
if( Objects.equals(var.getTyp(), "bool") && vardef.isNum(val))
    throw new IllegalArgumentException();
```

- **errors in printData.exe**

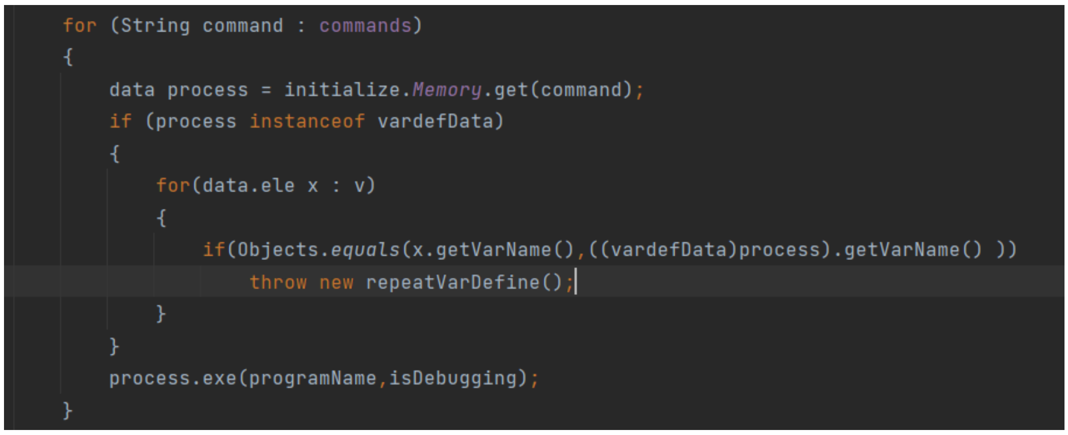
If expRef is an instance of vardefData, but it's not defined, a NullPointerException will be thrown.



```
if(!vardef.isNum(expRef) && !Objects.equals( "true",expRef) && !Objects.equals( "false",expRef))
{
    data x = initialize.Memory.get(expRef);
    if(x instanceof vardefData && !prog.isVarDefined((vardefData) x))
        throw new NullPointerException();
    output = x.exe(programName,isDebugging);
}
```

- **errors in blockData.exe**

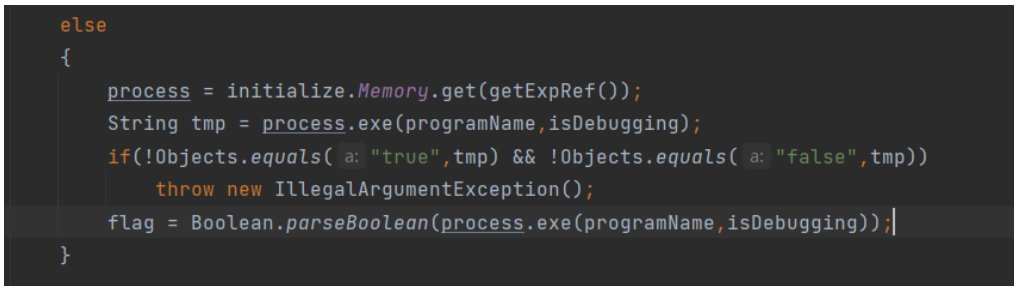
If there is a variable which is needed to define in the block, but it has been defined outside, a repeatVarDefine exception will be thrown.



```
for (String command : commands)
{
    data process = initialize.Memory.get(command);
    if (process instanceof vardefData)
    {
        for(data.ele x : v)
        {
            if(Objects.equals(x.getVarName(),((vardefData)process).getVarName() ))
                throw new repeatVarDefine();
        }
    }
    process.exe(programName,isDebugging);
}
```

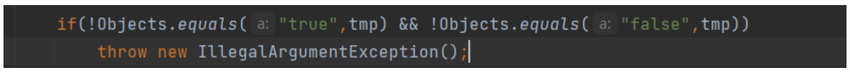
- **errors in conditionData.exe**

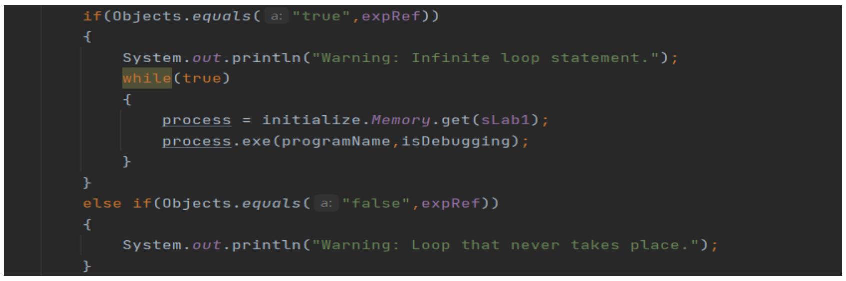
If the value of expRef is neither “true” or “false”, it will throw an IllegalArgumentException.


```
else
{
    process = initialize.Memory.get(getExpRef());
    String tmp = process.exe(programName,isDebugging);
    if(!Objects.equals( a: "true",tmp) && !Objects.equals( a: "false",tmp))
        throw new IllegalArgumentException();
    flag = Boolean.parseBoolean(process.exe(programName,isDebugging));
}
```

- **errors in loopData.exe**

If the value of expRef is neither “true” or “false”, it will throw an IllegalArgumentException. If the value of expRef is always “true” or “false”, it will give a warning.

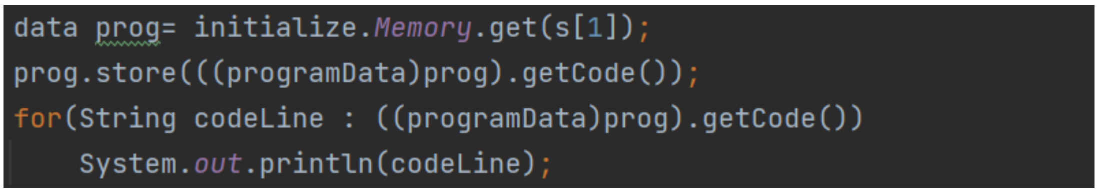

```
    if(!Objects.equals( a: "true",tmp) && !Objects.equals( a: "false",tmp))
        throw new IllegalArgumentException();
```


```
if(Objects.equals( a: "true",expRef))
{
    System.out.println("Warning: Infinite loop statement.");
    while(true)
    {
        process = initialize.Memory.get(sLab1);
        process.exe(programName,isDebugging);
    }
}
else if(Objects.equals( a: "false",expRef))
{
    System.out.println("Warning: Loop that never takes place.");
}
```

2.2.12 REQ12 list simple programs

1. **The requirement is implemented.**
2. **Implementation detials**

It will get **programData** from **Memory** by using programName as a key, store the code in a new vector name prog, and print out the code in the vector line by line by iteration with the **System.out.println()** method.


```
data prog= initialize.Memory.get(s[1]);
prog.store(((programData)prog).getCode());
for(String codeLine : ((programData)prog).getCode())
    System.out.println(codeLine);
```

3. **Error conditions**

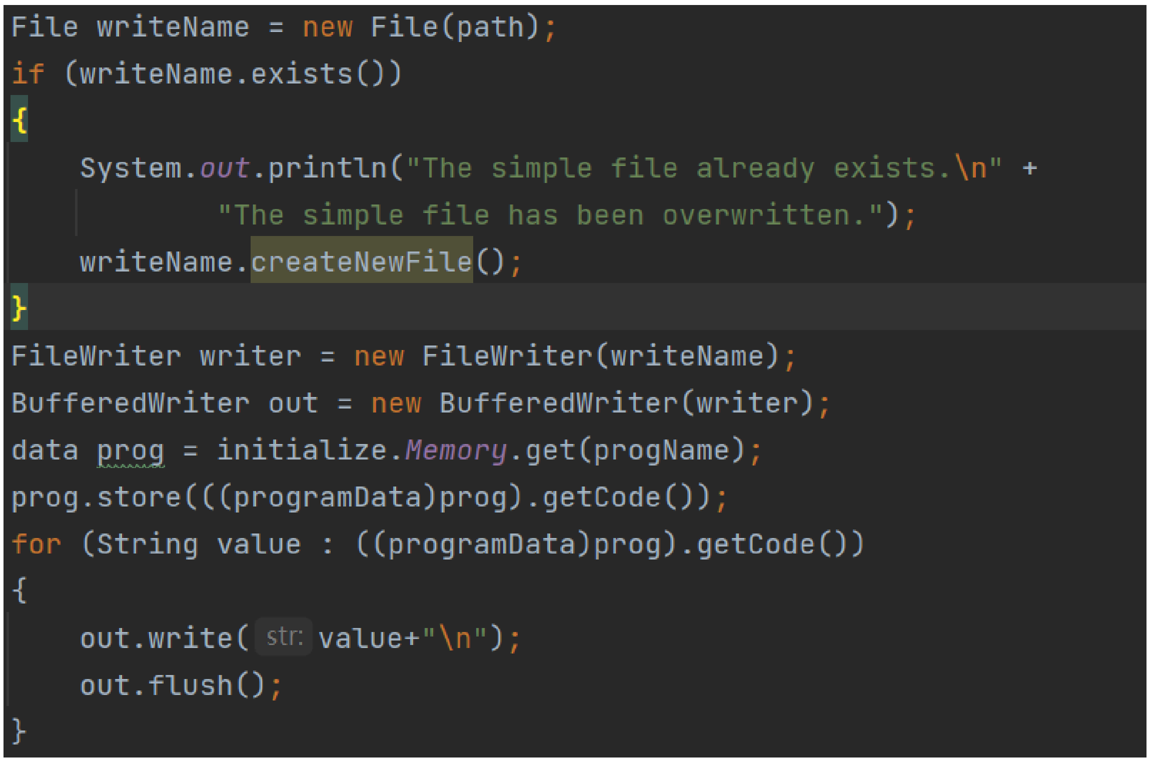
   - **Errors in enter command**

     It will check the length of the command string array split by space to make sure the command format is right. And call the **Memory** in **initialize** to find whether the programName has been created and whether it is an instance of **programData.**

2.2.13 REQ13 store simple programs

1. **The requirement is implemented.**
2. **Implementation detials**

Set the name of the file to the path entered by the user. If the file already exists, the program will tell the user and overwrite the contents of the original file. It will get the **programData** from **Memory** by using programName as a key, store the code in memory as a new vector name prog, import **java.io.WriterFile** and **java.io.BufferedWriter** to write the code in the vector to the file line by line by iteration, and use **flush()** method to refresh the buffer.



```
File writeName = new File(path);
if (writeName.exists())
{
    System.out.println("The simple file already exists.\n" +
            "The simple file has been overwritten.");
    writeName.createNewFile();
}
FileWriter writer = new FileWriter(writeName);
BufferedWriter out = new BufferedWriter(writer);
data prog = initialize.Memory.get(progName);
prog.store(((programData)prog).getCode());
for (String value : ((programData)prog).getCode())
{
    out.write( str: value+"\n");
    out.flush();
}
```

3. **Error conditions**
   - **Errors in enter command**

     It will check the length of the command string array split by space to make sure the command format is right. And call the **Memory** in **initialize** to find whether the programName has been created and whether it is an instance of **programData.** Also check whether the path is legal or not and the suffix of the file is ".simple".

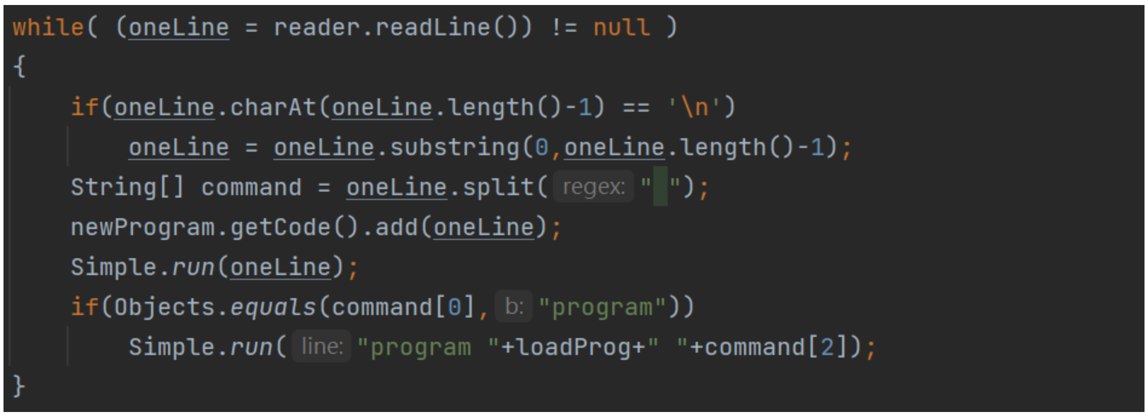
   - **Errors in storeProgram.exe**

     If an IOException error occurs during operation, the system will catch the error and throw.

2.2.14 REQ14 load simple programs

1. **The requirement is implemented.**
2. **Implementation detials**

Finds the path entered by the user in the computer's local files. The program will import java.io.FileReader and java.io.BufferedReader and write the code in the simple file iteratively line by line with the programName entered by the user to store it as a new program and store it in memory together with the named program in the file.



```
while( (oneLine = reader.readLine()) != null )
{
    if(oneLine.charAt(oneLine.length()-1) == '\n')
        oneLine = oneLine.substring(0,oneLine.length()-1);
    String[] command = oneLine.split( regex: " ");
    newProgram.getCode().add(oneLine);
    Simple.run(oneLine);
    if(Objects.equals(command[0], b: "program"))
        Simple.run( line: "program "+loadProg+" "+command[2]);
}
```

3. **Error conditions**
   - **Errors in enter command**

     It will check the length of the command string array split by space to make sure the command format is right. And call **Memory** in **initialize** to find out if the programName has been created, and return failure if it has. Also check whether the loadPath is legal or not and the suffix of the file is ".simple".

   - **Errors in loadProgram.exe**

     It will check if the file exists by the loadPath entered by the user, if not the file returns failure, and check if there is no program command in the file, if not it means that the program cannot be read.

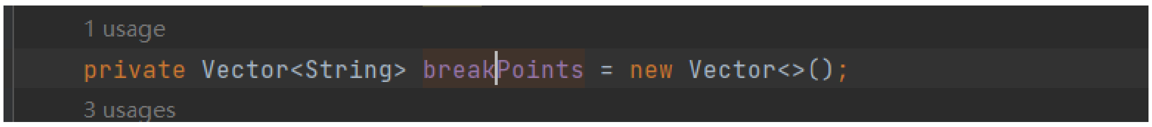
2.2.15 REQ15 quit

1. **The requirement is implemented.**
2. **Implementation detials**

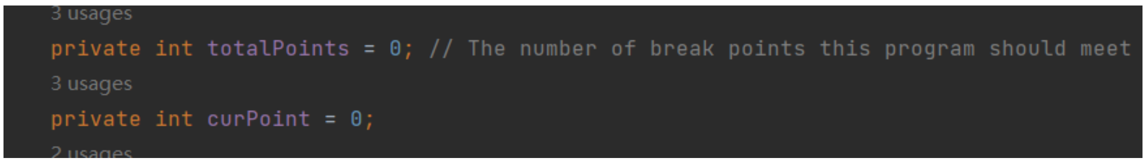
If the user enters “quit”, the Command-Line-Based Simple Program Interpreter will terminate (See details in 3 User Manual).

2.2.16.1 BON1.1 togglebreakpoint programName statementLab

1. **The requirement is implemented.**
2. **Implementation detials**
   - **Data Structure**



```
1 usage
private Vector<String> breakPoints = new Vector<>();
3 usages
```

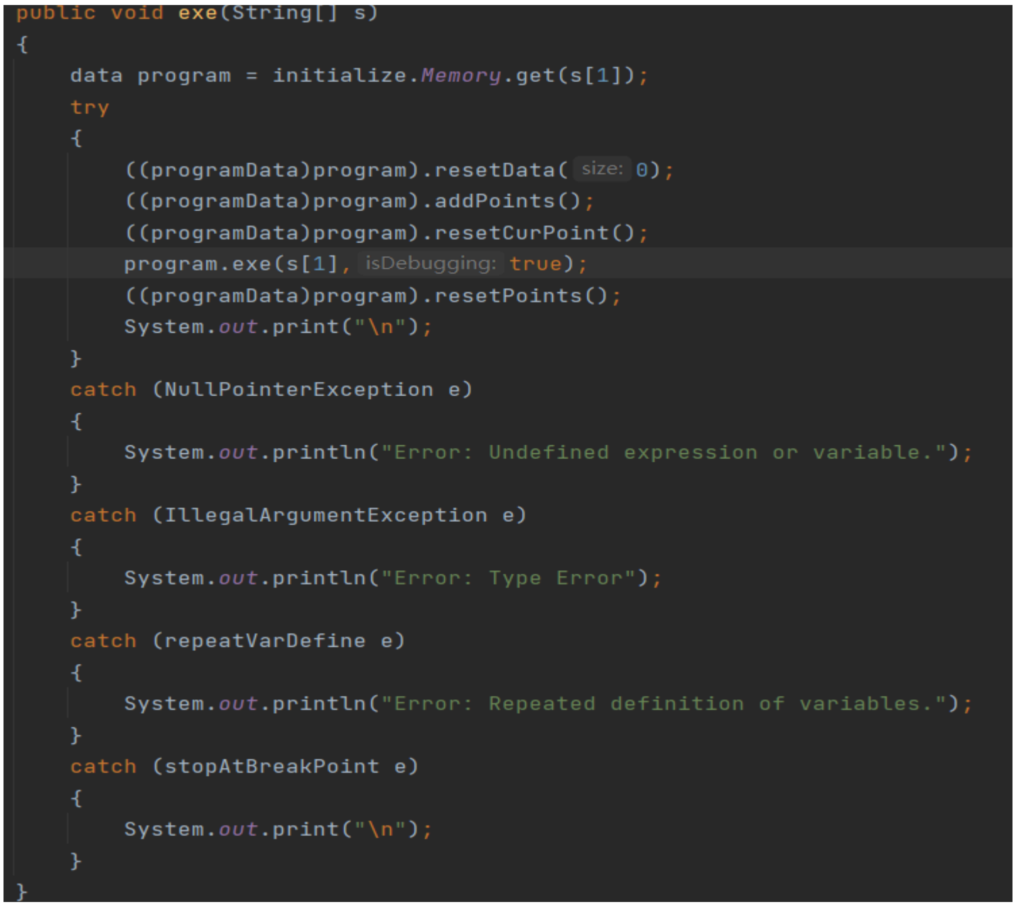


```
3 usages
private int totalPoints = 0; // The number of break points this program should meet
3 usages
private int curPoint = 0;
2 usages
```

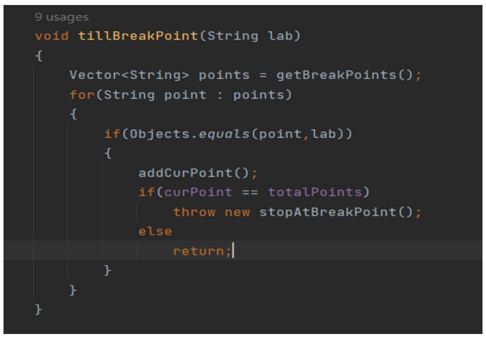
These are private fields in **programData**. For each instance of **programData**, there is a vector breakPoints storing the labels which are set to be break-points. The totalPoints and curPoint are used to record how many break-points should be met in debugging this program and how many break-points have been met right now. The following implementation will show how they work in debugging/continue debugging.

- **Implementation**

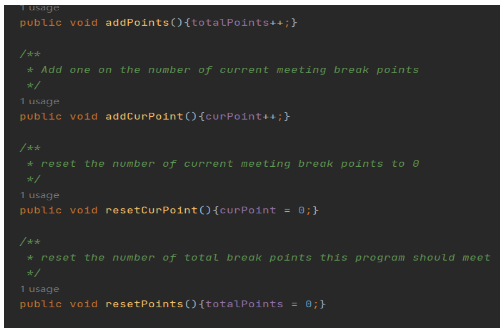
If there are no errors in the command statement, the exe method in **debugs** class will be invoked. The parameter inDebugging is set to true and in each command it will test if this command is a break-point. It's easy to figure out *the times you input the debug command equal to the number of break-points the program will meet*. So we only need to record how many times the debug command is called and the break-points the program has met now. If the current met number of break-points equals the total number we should meet, a stopAtBreakPoint exception will be thrown. Why do we use this exception to end the program? Because only the debugging completes the whole program, the totalPoints will be reset to zero. In this condition, there is no exception thrown and the recursion can come back to execute the resetPoints() function. Otherwise, if an exception is thrown, it will not be reset.



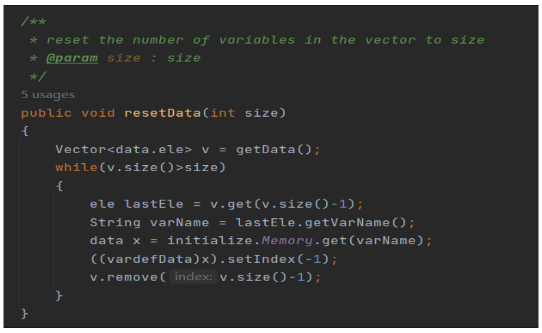
```java
public void exe(String[] s)
{
    data program = initialize.Memory.get(s[1]);
    try
    {
        ((programData)program).resetData( size: 0);
        ((programData)program).addPoints();
        ((programData)program).resetCurPoint();
        program.exe(s[1], isDebugging: true);
        ((programData)program).resetPoints();
        System.out.print("\n");
    }
    catch (NullPointerException e)
    {
        System.out.println("Error: Undefined expression or variable.");
    }
    catch (IllegalArgumentException e)
    {
        System.out.println("Error: Type Error");
    }
    catch (repeatVarDefine e)
    {
        System.out.println("Error: Repeated definition of variables.");
    }
    catch (stopAtBreakPoint e)
    {
        System.out.print("\n");
    }
}
```



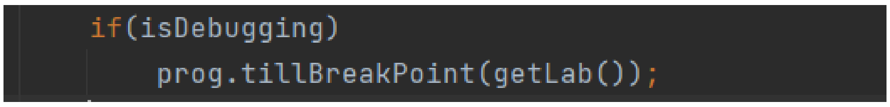
```java
9 usages
void tillBreakPoint(String lab)
{
    Vector<String> points = getBreakPoints();
    for(String point : points)
    {
        if(Objects.equals(point,lab))
        {
            addCurPoint();
            if(curPoint == totalPoints)
                throw new stopAtBreakPoint();
            else
                return;
        }
    }
}
```



```java
1 usage
public void addPoints(){totalPoints++;}

/**
 * Add one on the number of current meeting break points
 */
1 usage
public void addCurPoint(){curPoint++;}

/**
 * reset the number of current meeting break points to 0
 */
1 usage
public void resetCurPoint(){curPoint = 0;}

/**
 * reset the number of total break points this program should meet
 */
1 usage
public void resetPoints(){totalPoints = 0;}
```



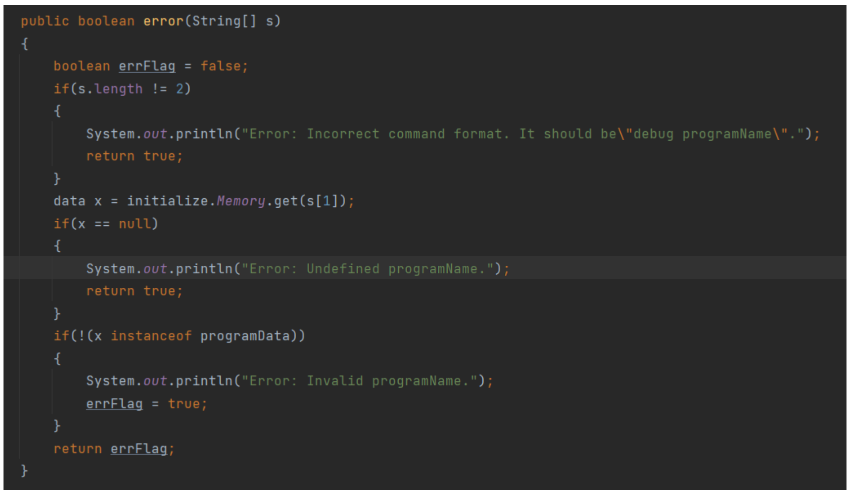
```java
/**
 * reset the number of variables in the vector to size
 * @param size : size
 */
5 usages
public void resetData(int size)
{
    Vector<data.ele> v = getData();
    while(v.size()>size)
    {
        ele lastEle = v.get(v.size()-1);
        String varName = lastEle.getVarName();
        data x = initialize.Memory.get(varName);
        ((vardefData)x).setIndex(-1);
        v.remove( index: v.size()-1);
    }
}
```



```java
    if(isDebugging)
        prog.tillBreakPoint(getLab());
```

3. **Error conditions**
    - **Errors in enter command**

      It will check the length of the command string array split by space to make sure the command format is right. It will check if the programName has been defined and if the command with this label is an instance of **programData** to make sure it's a program instead of other data with the same name.

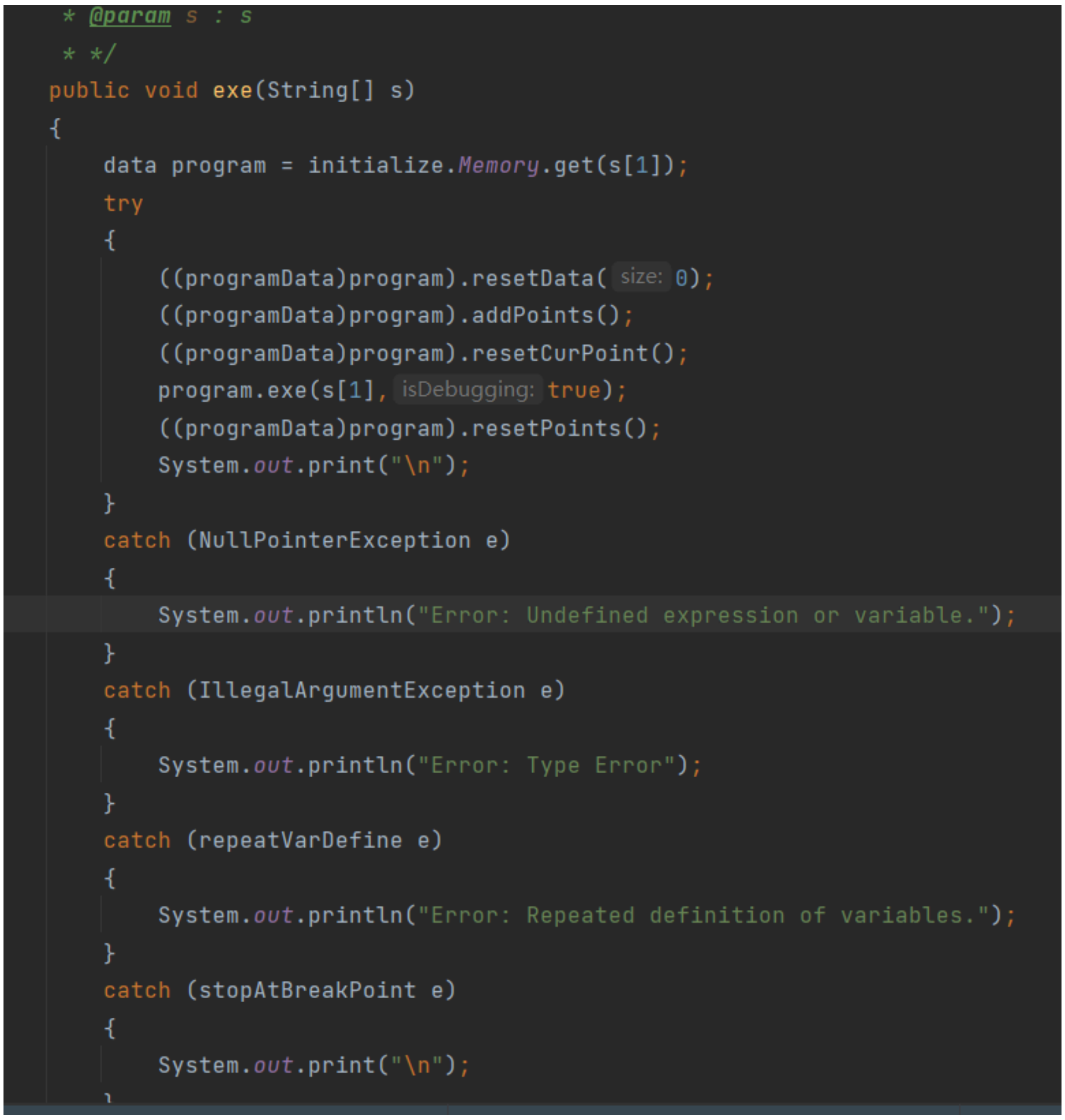


```
public boolean error(String[] s)
{
    boolean errFlag = false;
    if(s.length != 2)
    {
        System.out.println("Error: Incorrect command format. It should be\"debug programName\".");
        return true;
    }
    data x = initialize.Memory.get(s[1]);
    if(x == null)
    {
        System.out.println("Error: Undefined programName.");
        return true;
    }
    if(!(x instanceof programData))
    {
        System.out.println("Error: Invalid programName.");
        errFlag = true;
    }
    return errFlag;
}
```

2.2.16.2 BON1.2 debug programName

1. **The requirement is implemented.**
2. **Implementation detials**

   This debugging is similar to **execute**. The boolean parameter will be set to true. And when executing **commandData**, it will test if the command is a break-point. However, after debugging, it will not reset the variable vector for the implementation of **inspect** command. It will reset it before debugging.



```
 * @param s : s
 * */
public void exe(String[] s)
{
    data program = initialize.Memory.get(s[1]);
    try
    {
        ((programData)program).resetData( size: 0);
        ((programData)program).addPoints();
        ((programData)program).resetCurPoint();
        program.exe(s[1], isDebugging: true);
        ((programData)program).resetPoints();
        System.out.print("\n");
    }
    catch (NullPointerException e)
    {
        System.out.println("Error: Undefined expression or variable.");
    }
    catch (IllegalArgumentException e)
    {
        System.out.println("Error: Type Error");
    }
    catch (repeatVarDefine e)
    {
        System.out.println("Error: Repeated definition of variables.");
    }
    catch (stopAtBreakPoint e)
    {
        System.out.print("\n");
```

3. Error conditions

- **Errors in enter command**

  It will check the length of the command string array split by space to make sure the command format is right. It will check if the programName has been defined and if the command with this label is an instance of **programData** to make sure it's a program instead of other data with the same name.

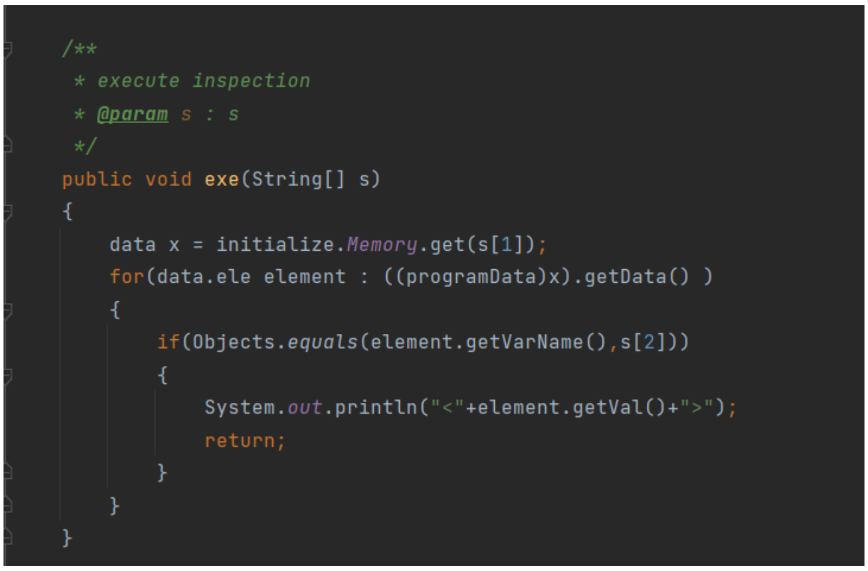
- **Errors during execution**

  It's the same as **execute** commands.

2.2.16.3 BON1.3 inspect programName varName

1. **The requirement is implemented.**
2. **Implementation detials**

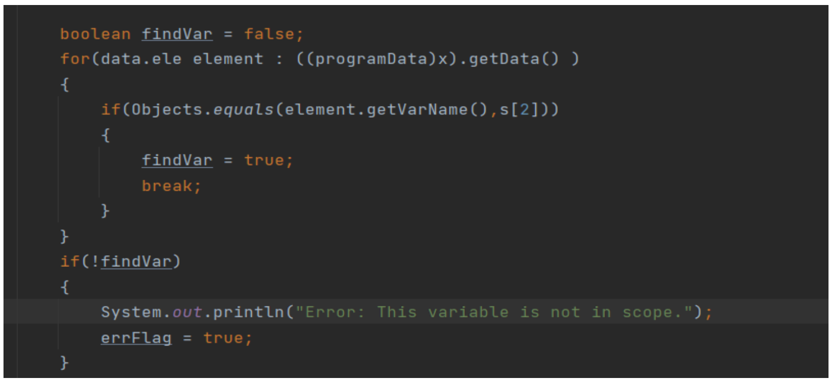
It has been mentioned that the variable vector will not be reset after each debugging. So we only need to find the target variable from it.



```
/**
 * execute inspection
 * @param s : s
 */
public void exe(String[] s)
{
    data x = initialize.Memory.get(s[1]);
    for(data.ele element : ((programData)x).getData() )
    {
        if(Objects.equals(element.getVarName(),s[2]))
        {
            System.out.println("<"+element.getVal()+">");
            return;
        }
    }
}
```

3. **Error conditions**
   - **Errors in enter command**

It will check the length of the command string array split by space to make sure the command format is right. It will check if the programName has been defined and if the command with this label is an instance of **programData** to make sure it's a program instead of other data with the same name. Besides, it will also check if the target variable is in scope.

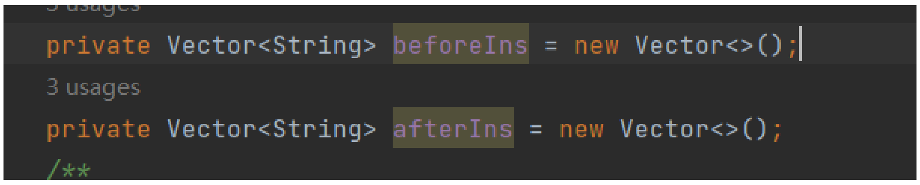


```
boolean findVar = false;
for(data.ele element : ((programData)x).getData() )
{
    if(Objects.equals(element.getVarName(),s[2]))
    {
        findVar = true;
        break;
    }
}
if(!findVar)
{
    System.out.println("Error: This variable is not in scope.");
    errFlag = true;
}
```

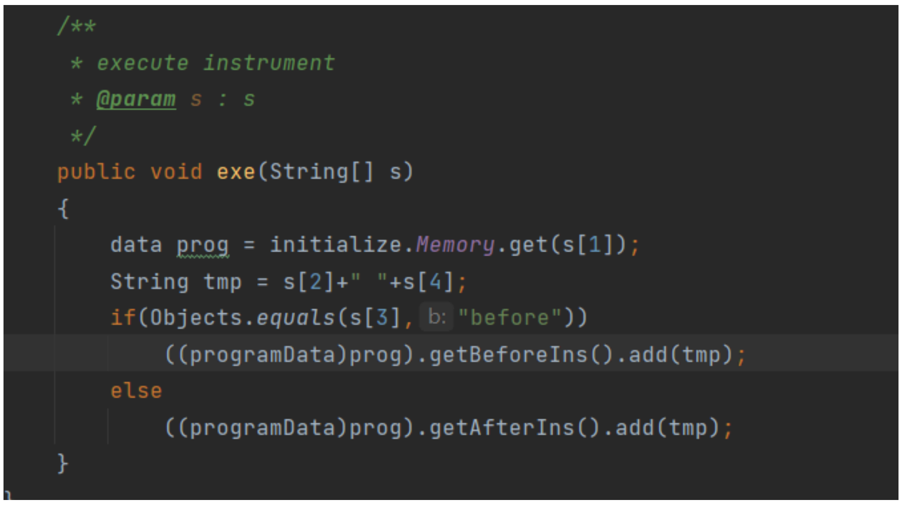
2.2.17 BON2 instrument simple programs

1. **The requirement is implemented.**
2. **Implementation detials**

In each instance of **programData**, there are two vectors, beforeIns and afterIns, to store the corresponding labels. When the command is executed, the statementLab and expRef will be stored in the corresponding vector.



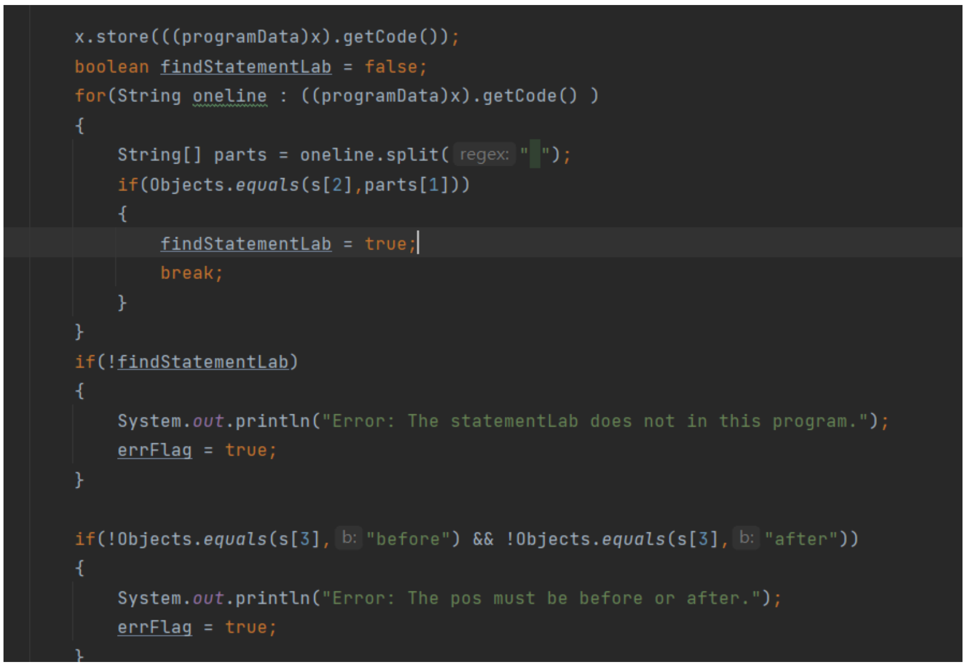
```
private Vector<String> beforeIns = new Vector<>();
3 usages
private Vector<String> afterIns = new Vector<>();
/**
```



```
/**
 * execute instrument
 * @param s : s
 */
public void exe(String[] s)
{
    data prog = initialize.Memory.get(s[1]);
    String tmp = s[2]+" "+s[4];
    if(Objects.equals(s[3], b: "before"))
        ((programData)prog).getBeforeIns().add(tmp);
    else
        ((programData)prog).getAfterIns().add(tmp);
}
```

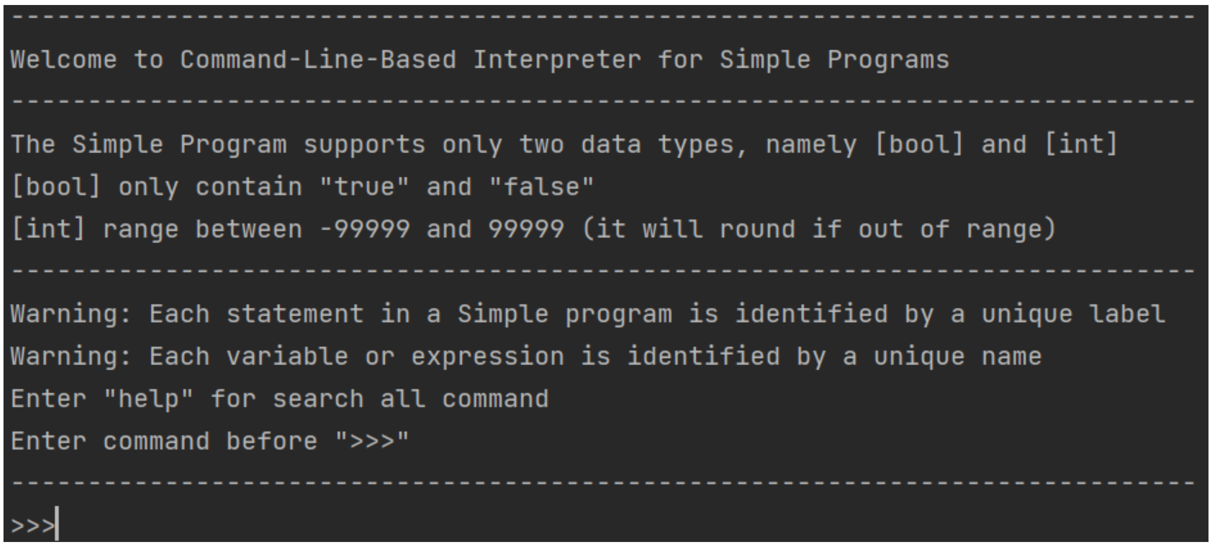
3. **3. Error conditions**
   - **Errors in enter command**

It will check the length of the command string array split by space to make sure the command format is right. It will check if the programName has been defined and if the command with this label is an instance of **programData** to make sure it's a program instead of other data with the same name. Besides, it will check if the statementLab exists and make sure that the pos is "before" or "after".


```
x.store(((programData)x).getCode());
boolean findStatementLab = false;
for(String oneline : ((programData)x).getCode() )
{
    String[] parts = oneline.split( regex: " ");
    if(Objects.equals(s[2],parts[1]))
    {
        findStatementLab = true;
        break;
    }
}
if(!findStatementLab)
{
    System.out.println("Error: The statementLab does not in this program.");
    errFlag = true;
}

if(!Objects.equals(s[3], b: "before") && !Objects.equals(s[3], b: "after"))
{
    System.out.println("Error: The pos must be before or after.");
    errFlag = true;
```

3 User Manual

Run the application class and it will launch the menu class. In the menu, the user will see a welcome message and some instructions and warnings for this Interpreter. The user will see the line "Enter "help" for search all commands and examples of commands", which means that typing "help" in the Interpreter " in the Interpreter to get a description of all command statements. The user will see the line "Enter command before >>>", which means that when


```
----------------------------------------------------------------------
Welcome to Command-Line-Based Interpreter for Simple Programs
----------------------------------------------------------------------
The Simple Program supports only two data types, namely [bool] and [int]
[bool] only contain "true" and "false"
[int] range between -99999 and 99999 (it will round if out of range)
----------------------------------------------------------------------
Warning: Each statement in a Simple program is identified by a unique label
Warning: Each variable or expression is identified by a unique name
Enter "help" for search all command
Enter command before ">>>"
----------------------------------------------------------------------
>>>
```

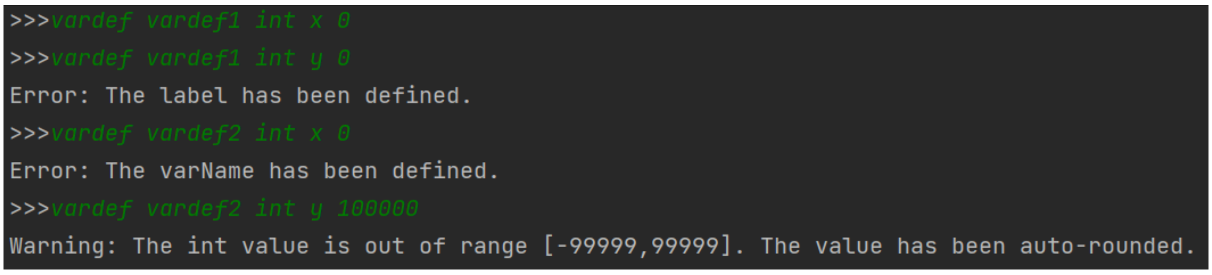
the user sees ">>>", he can enter the command to run the simple program.

Figure 2 : Welcome Page

To define the variable, input the command:

vardef [lab] [typ] [varName] [expRef]

If a variable is defined unsuccessfully, the interpreter will print the error (See Figure 3 for some Error or Warning).



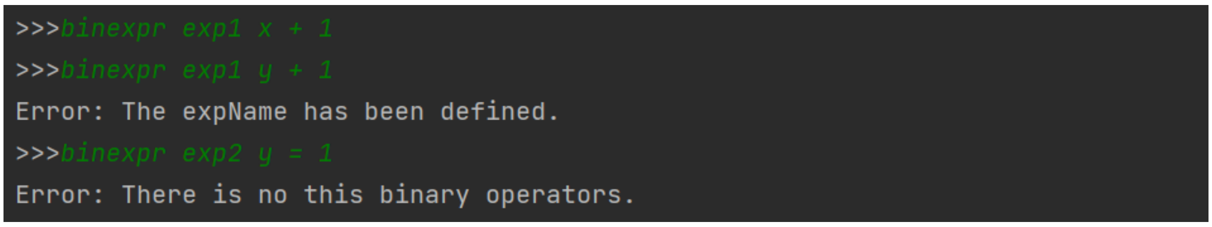
```
>>>vardef vardef1 int x 0
>>>vardef vardef1 int y 0
Error: The label has been defined.
>>>vardef vardef2 int x 0
Error: The varName has been defined.
>>>vardef vardef2 int y 100000
Warning: The int value is out of range [-99999,99999]. The value has been auto-rounded.
```

Figure 3 : variable define

To define the binary expression, input the command:

binexpr [expName] [expRef1] [bop] [expRef2]

You need to define **[expRef1]** first. If a binary expression is defined unsuccessfully, the interpreter will print the error (See Figure 4).



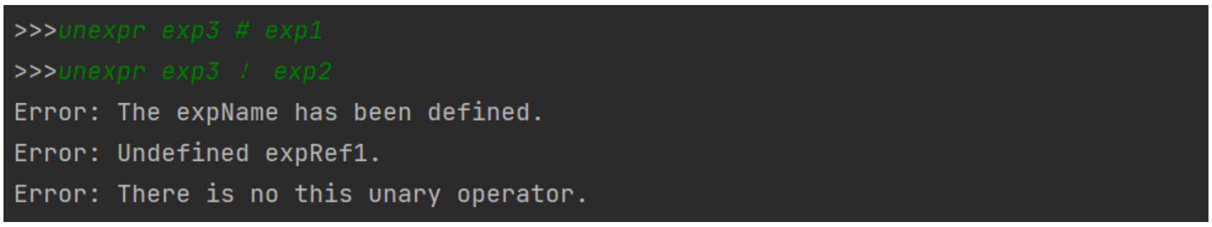
```
>>>binexpr exp1 x + 1
>>>binexpr exp1 y + 1
Error: The expName has been defined.
>>>binexpr exp2 y = 1
Error: There is no this binary operators.
```

Figure 4 : binary expression define

To define the unary expression, input the command:

unexpr [expName] [uop] [expRef1]

You need to define**[expRef1]** first. If a unary expression is defined unsuccessfully, the interpreter will print the error (See Figure 5).



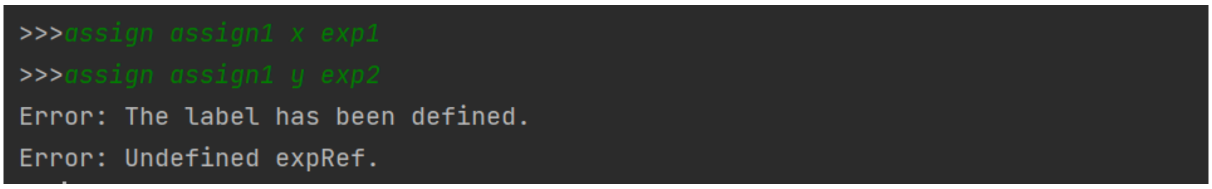
```
>>>unexpr exp3 # exp1
>>>unexpr exp3 ! exp2
Error: The expName has been defined.
Error: Undefined expRef1.
Error: There is no this unary operator.
```

Figure 5 : unary expression define

To define the assignment statements, input the command:

assign [lab] [varName] [expRef]

You need to define **[varName]** and **[expRef]** first. If an assignment statement is defined unsuccessfully, the interpreter will print the error (See Figure 6).



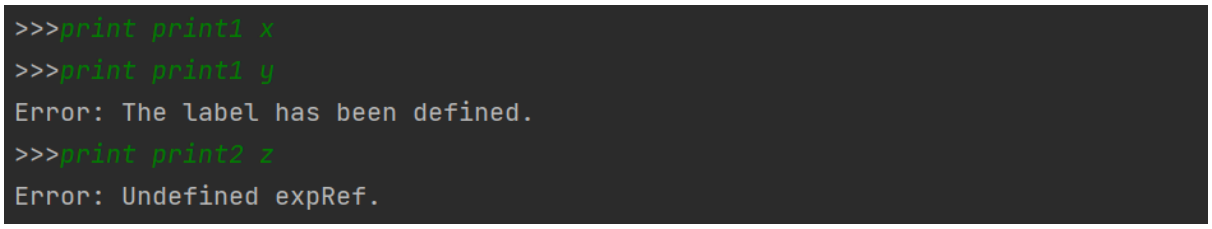
```
>>>assign assign1 x exp1
>>>assign assign1 y exp2
Error: The label has been defined.
Error: Undefined expRef.
```

Figure 6 : assignment statements define

To define the print statements, input the command:

print [lab] [expRef]

You need to define **[expRef]** first. If a print statement is defined unsuccessfully, the interpreter will print the error (See Figure 7).



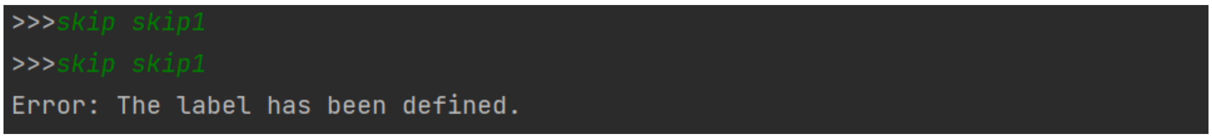
```
>>>print print1 x
>>>print print1 y
Error: The label has been defined.
>>>print print2 z
Error: Undefined expRef.
```

Figure 7 : print statements define

To define the skip statements, input the command:

skip [lab]

If a skip statement is defined unsuccessfully, the interpreter will print the error (See Figure 8).



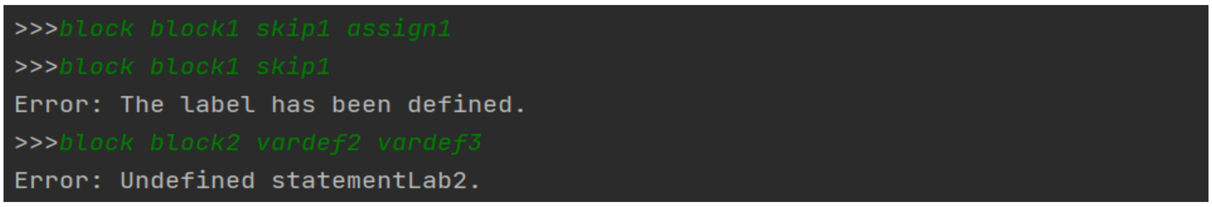
```
>>>skip skip1
>>>skip skip1
Error: The label has been defined.
```

Figure 8 : skip statements define

To define the block statements, input the command:

block [lab] [statementLab1] ... [statementLabn]

You need to define **[statementLab1]** to **[statementLabn]** first. If a block statement is defined unsuccessfully, the interpreter will print the error (See Figure 9).



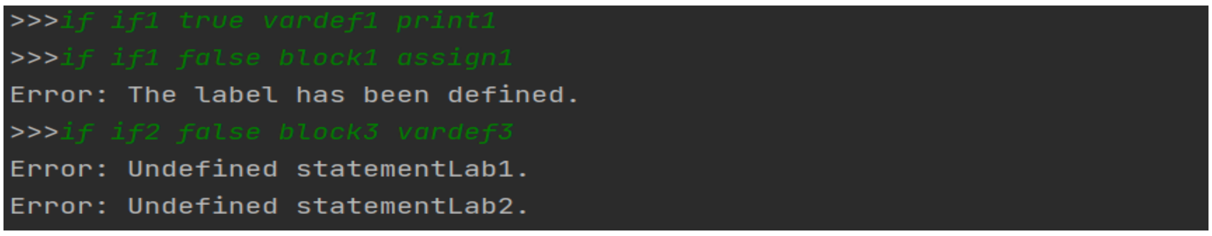
```
>>>block block1 skip1 assign1
>>>block block1 skip1
Error: The label has been defined.
>>>block block2 vardef2 vardef3
Error: Undefined statementLab2.
```

Figure 9 : block statements define

To define the conditional statements, input the command:

if [lab] [expRef] [statementLab1] [statementLab2]

You need to define **[statementLab1]** and **[statementLab2]** first. If **[expRef]** is not true or false, you should also define it first**.** If a loop statement is defined unsuccessfully, the interpreter will print the error (See Figure 10).



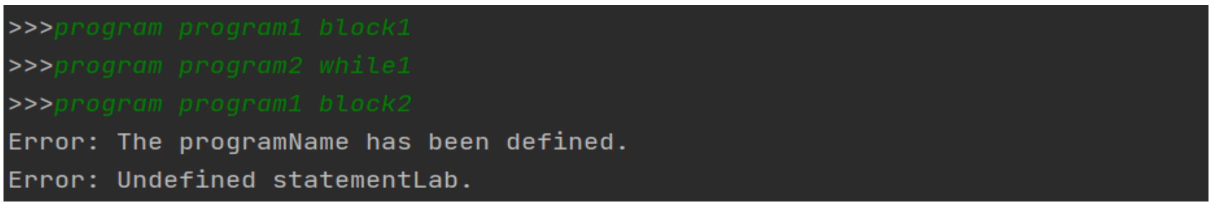
```
>>>if if1 true vardef1 print1
>>>if if1 false block1 assign1
Error: The label has been defined.
>>>if if2 false block3 vardef3
Error: Undefined statementLab1.
Error: Undefined statementLab2.
```

Figure 10 : conditional statements define

To define the Simple programs, input the command:

program [programName] [statementLab]

You need to define **[statementLab]** first. If a Simple program is defined unsuccessfully, the interpreter will print the error (See Figure 11).



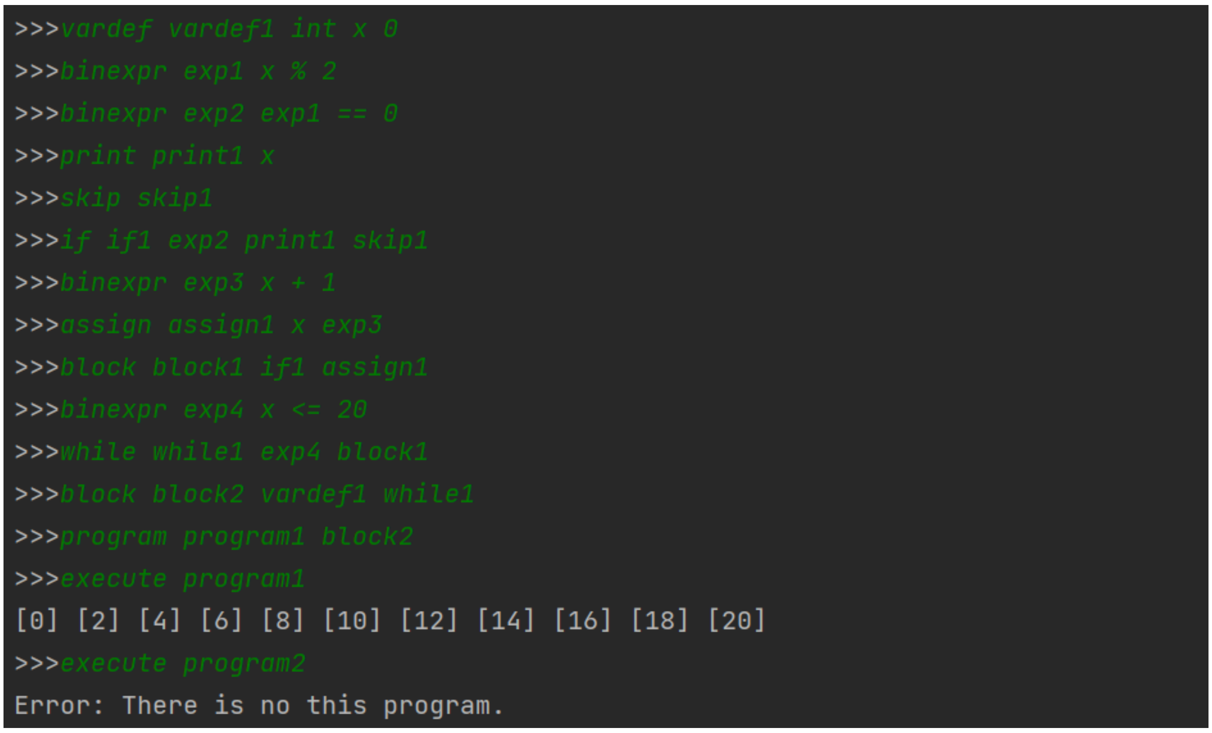
```
>>>program program1 block1
>>>program program2 while1
>>>program program1 block2
Error: The programName has been defined.
Error: Undefined statementLab.
```

Figure 11 : Simple programs define

To execute the Simple programs, input the command:

execute [programName]

You need to define **[programName]** first. If a Simple program is executed unsuccessfully, the interpreter will print the error (See Figure 12).



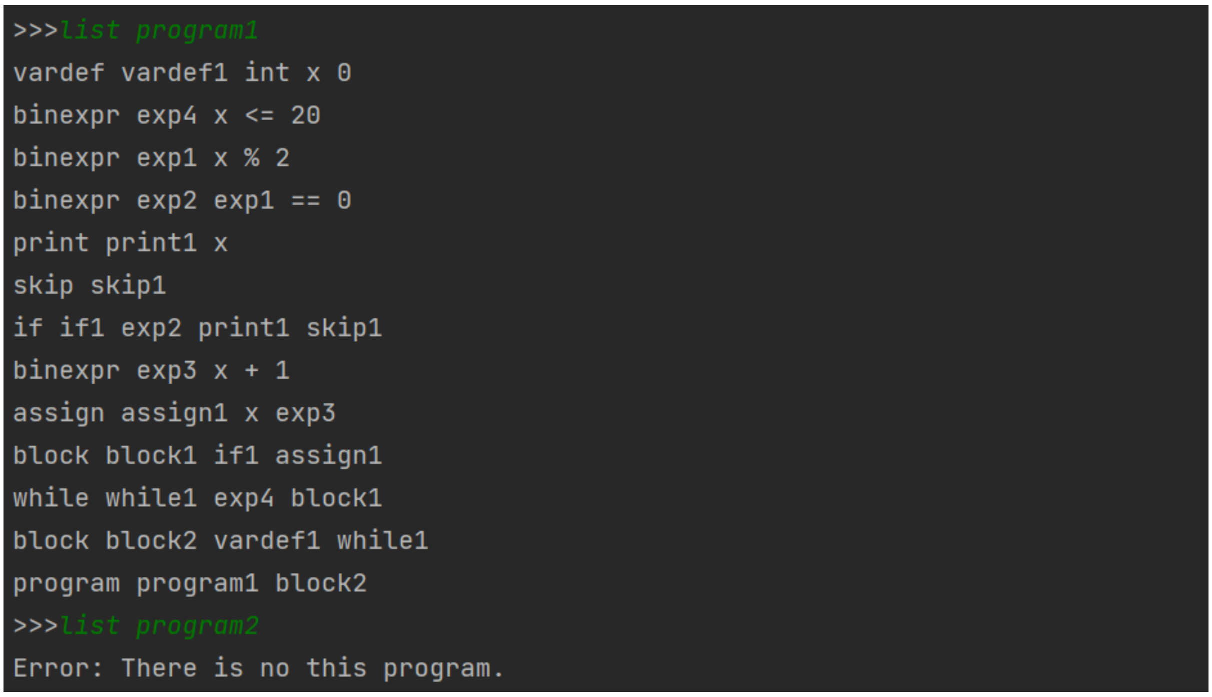
```
>>>vardef vardef1 int x 0
>>>binexpr exp1 x % 2
>>>binexpr exp2 exp1 == 0
>>>print print1 x
>>>skip skip1
>>>if if1 exp2 print1 skip1
>>>binexpr exp3 x + 1
>>>assign assign1 x exp3
>>>block block1 if1 assign1
>>>binexpr exp4 x <= 20
>>>while while1 exp4 block1
>>>block block2 vardef1 while1
>>>program program1 block2
>>>execute program1
[0] [2] [4] [6] [8] [10] [12] [14] [16] [18] [20]
>>>execute program2
Error: There is no this program.
```

Figure 12 : execute Simple programs

To list the Simple programs, input the command:

list [programName]

You need to define **[programName]** first. If a Simple program is listed unsuccessfully, the interpreter will print the error (See Figure 13).



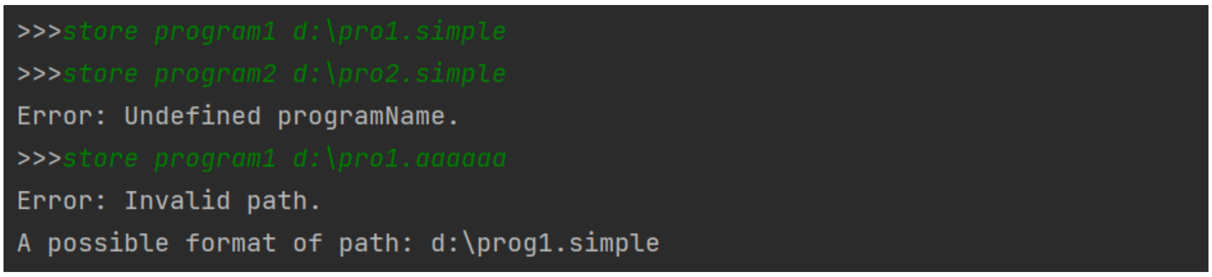
```
>>>list program1
vardef vardef1 int x 0
binexpr exp4 x <= 20
binexpr exp1 x % 2
binexpr exp2 exp1 == 0
print print1 x
skip skip1
if if1 exp2 print1 skip1
binexpr exp3 x + 1
assign assign1 x exp3
block block1 if1 assign1
while while1 exp4 block1
block block2 vardef1 while1
program program1 block2
>>>list program2
Error: There is no this program.
```

Figure 13 : list Simple programs

To store the Simple programs, input the command:

store [programName] [path]

You need to define **[programName]** first. If a Simple program is stored unsuccessfully, the interpreter will print the error (See Figure 14).



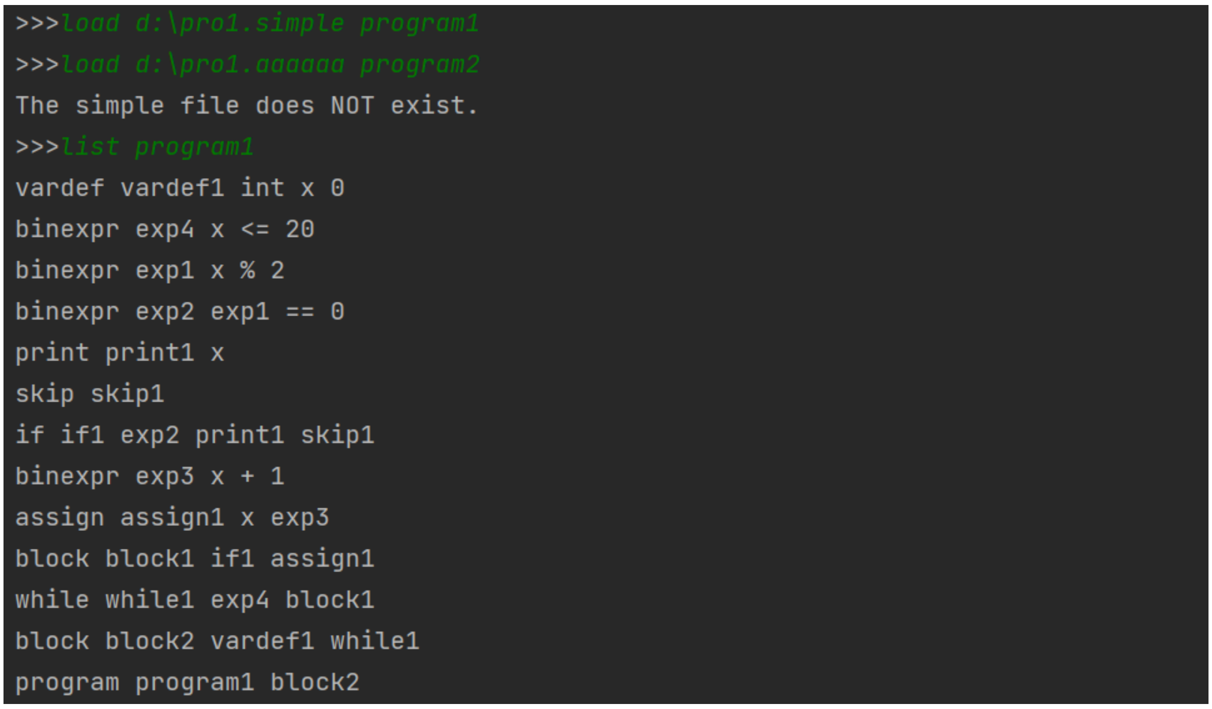
```
>>>store program1 d:\pro1.simple
>>>store program2 d:\pro2.simple
Error: Undefined programName.
>>>store program1 d:\pro1.aaaaaa
Error: Invalid path.
A possible format of path: d:\prog1.simple
```

Figure 14 : store Simple programs

To load the Simple programs, input the command:

load [path] [programName]

You should not define **[programName]** first. If a Simple program is loaded unsuccessfully, the interpreter will print the error (See Figure 15).



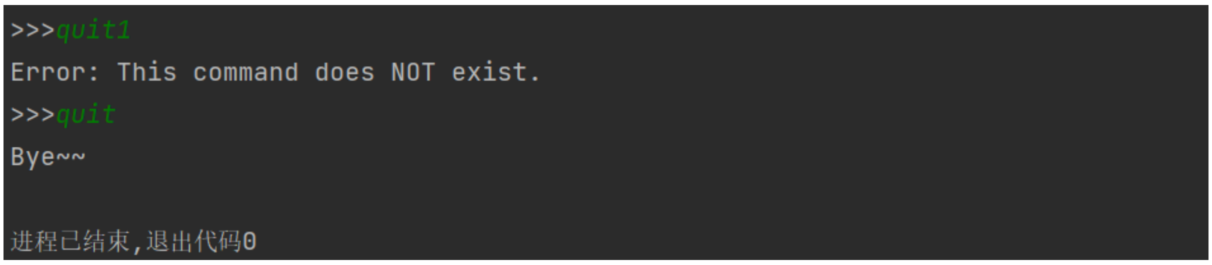
```
>>>load d:\pro1.simple program1
>>>load d:\pro1.aaaaaa program2
The simple file does NOT exist.
>>>list program1
vardef vardef1 int x 0
binexpr exp4 x <= 20
binexpr exp1 x % 2
binexpr exp2 exp1 == 0
print print1 x
skip skip1
if if1 exp2 print1 skip1
binexpr exp3 x + 1
assign assign1 x exp3
block block1 if1 assign1
while while1 exp4 block1
block block2 vardef1 while1
program program1 block2
```

Figure 15 : load Simple programs

To quit the interpreter , input the command:

quit

If the interpreter quit unsuccessfully, the interpreter will print the error (See Figure 16).



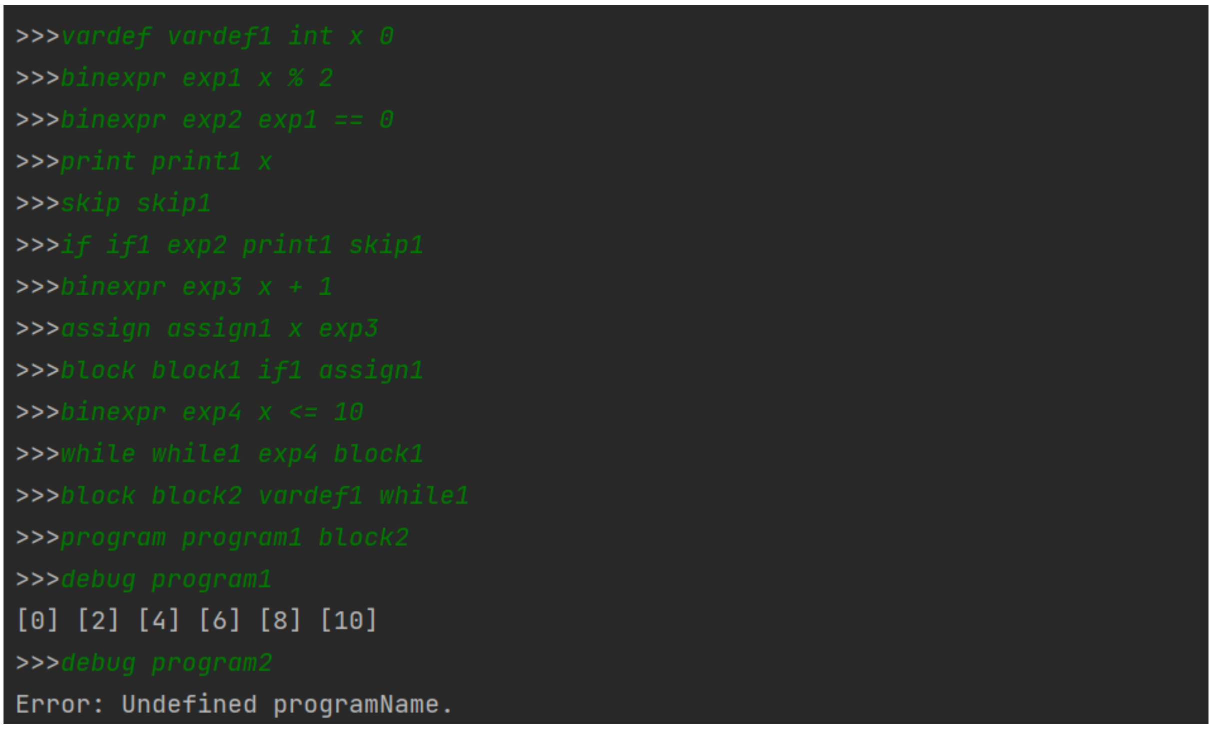
```
>>>quit1
Error: This command does NOT exist.
>>>quit
Bye~~

进程已结束,退出代码0
```

Figure 16 : quit

To debug the Simple programs, input the command:

debug [programName]

You should define **[programName]** first. If there is no breakpoint, the debug has the same role as execute. If a Simple program is loaded unsuccessfully, the interpreter will print the error (See Figure 17).



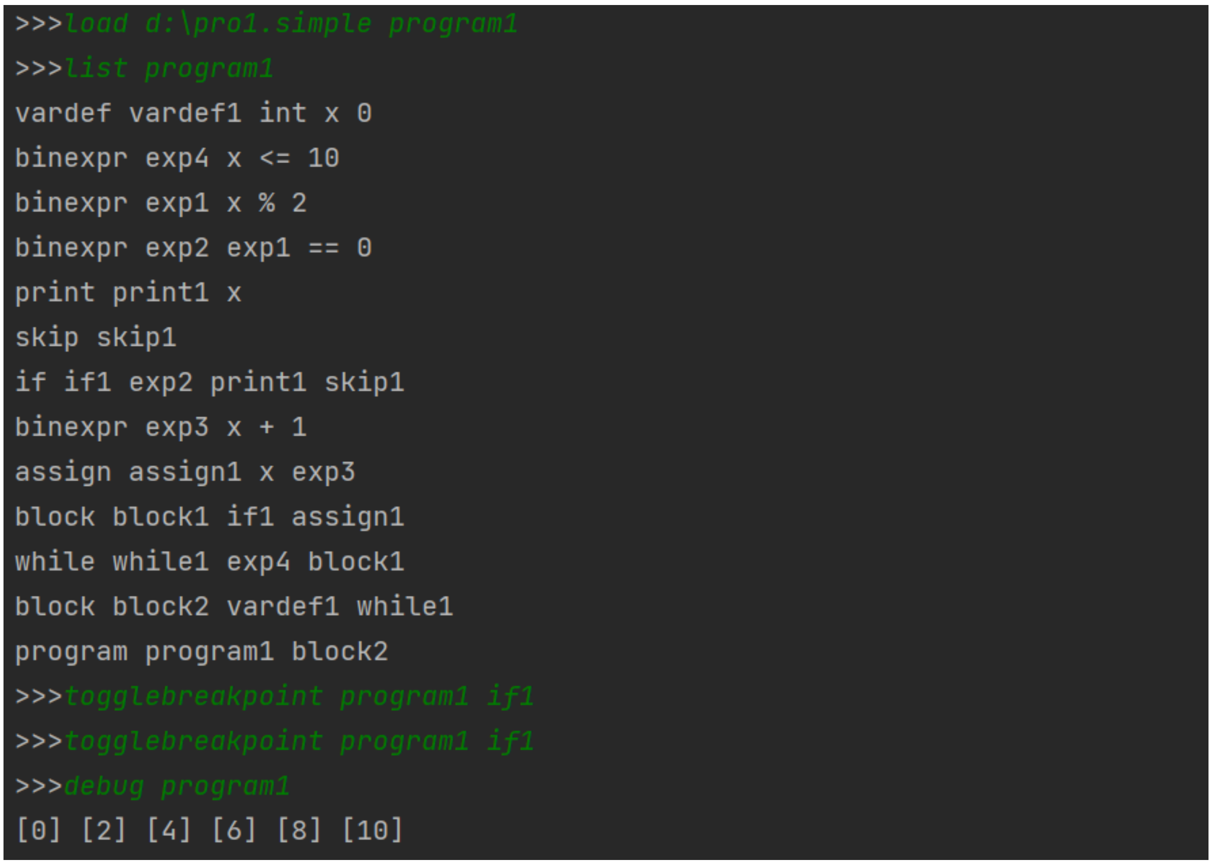
```
>>>vardef vardef1 int x 0
>>>binexpr exp1 x % 2
>>>binexpr exp2 exp1 == 0
>>>print print1 x
>>>skip skip1
>>>if if1 exp2 print1 skip1
>>>binexpr exp3 x + 1
>>>assign assign1 x exp3
>>>block block1 if1 assign1
>>>binexpr exp4 x <= 10
>>>while while1 exp4 block1
>>>block block2 vardef1 while1
>>>program program1 block2
>>>debug program1
[0] [2] [4] [6] [8] [10]
>>>debug program2
Error: Undefined programName.
```

Figure 17 : debug Simple programs

To set/remove breakpoint, input the command:

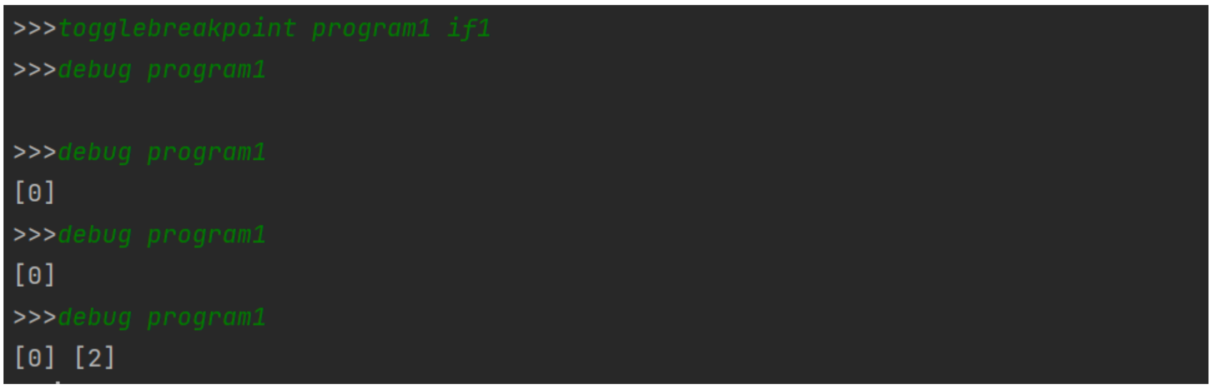
togglebreakpoint [programName] [statementLab]

You should define **[programName]** and **[statementLab]** first. If no breakpoint is specified in the togglebreakpoint command, a breakpoint will be set up according to the command line, otherwise the breakpoint at that position will be canceled(See Figure 18 & 19). If a Simple program is loaded unsuccessfully, the interpreter will print the error (See Figure 20 for some Error type).



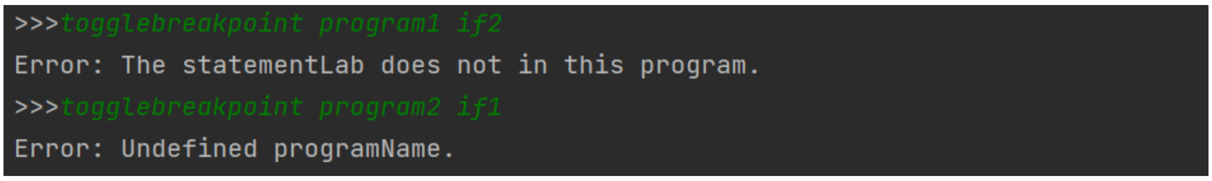
```
>>>load d:\pro1.simple program1
>>>list program1
vardef vardef1 int x 0
binexpr exp4 x <= 10
binexpr exp1 x % 2
binexpr exp2 exp1 == 0
print print1 x
skip skip1
if if1 exp2 print1 skip1
binexpr exp3 x + 1
assign assign1 x exp3
block block1 if1 assign1
while while1 exp4 block1
block block2 vardef1 while1
program program1 block2
>>>togglebreakpoint program1 if1
>>>togglebreakpoint program1 if1
>>>debug program1
[0] [2] [4] [6] [8] [10]
```

Figure 18 : remove breakpoint



```
>>>togglebreakpoint program1 if1
>>>debug program1

>>>debug program1
[0]
>>>debug program1
[0]
>>>debug program1
[0] [2]
```

Figure 19 : set breakpoint



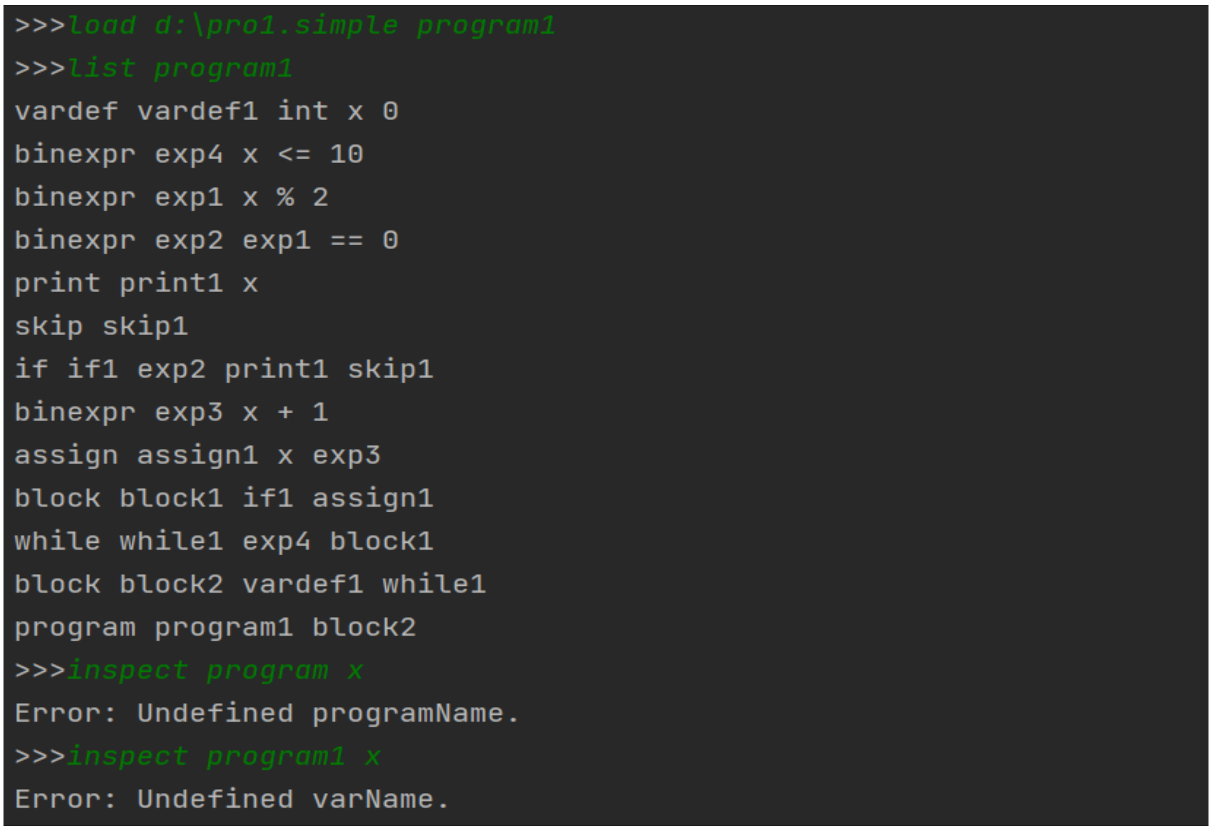
```
>>>togglebreakpoint program1 if2
Error: The statementLab does not in this program.
>>>togglebreakpoint program2 if1
Error: Undefined programName.
```

Figure 20 : toggle breakpoint error

To inspect variable, input the command:

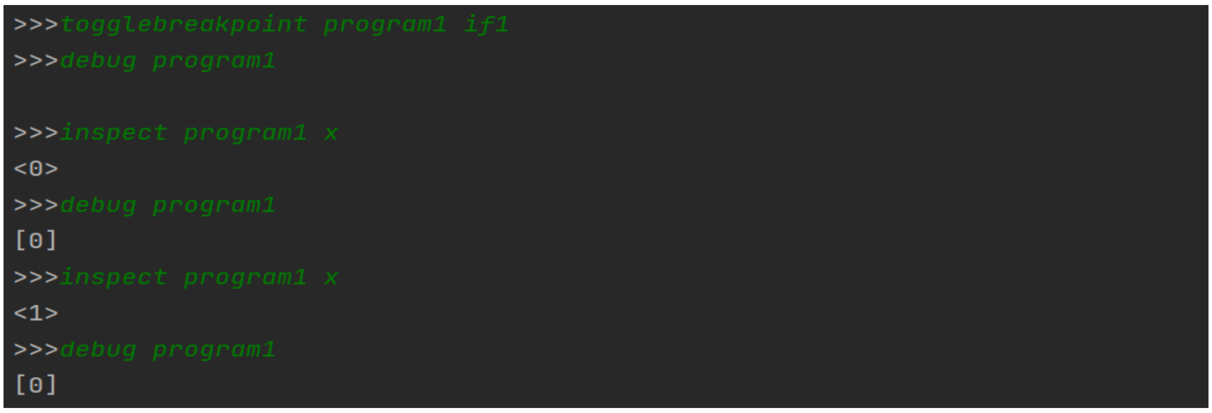
inspect [programName] [varName]

You should define **[programName]** and **[varName]** first. If no breakpoint is specified in the Simple program and no debug is executed, the inspect statement will not be executed and will only take effect after the debug is started(See Figure 21 & 22). If the variable is not in the scope, it will If a variable is inspected unsuccessfully, the interpreter will print the error (See Figure 21 for some Error type).



```
>>>load d:\pro1.simple program1
>>>list program1
vardef vardef1 int x 0
binexpr exp4 x <= 10
binexpr exp1 x % 2
binexpr exp2 exp1 == 0
print print1 x
skip skip1
if if1 exp2 print1 skip1
binexpr exp3 x + 1
assign assign1 x exp3
block block1 if1 assign1
while while1 exp4 block1
block block2 vardef1 while1
program program1 block2
>>>inspect program x
Error: Undefined programName.
>>>inspect program1 x
Error: Undefined varName.
```

Figure 21 : unsuccessfully inspect x



```
>>>togglebreakpoint program1 if1
>>>debug program1

>>>inspect program1 x
<0>
>>>debug program1
[0]
>>>inspect program1 x
<1>
>>>debug program1
[0]
```

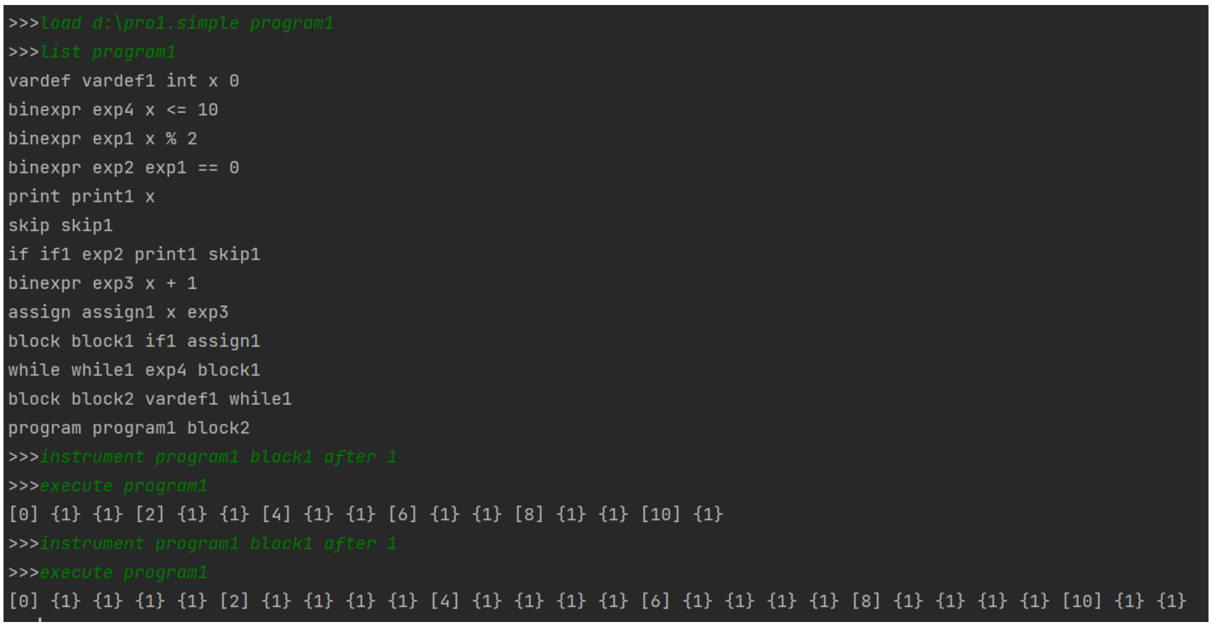
Figure 22 : successfully inspect x

To instrument expression reference, input the command:

instrument [programName] [statementLab] [pos] [expRef]

You should define **[programName]** and **[statementLab]** first. For the same statement pointed to by command, the instrument command does not replace, means that multiple instrument commands can be used to print out multiple strings at a single statement location

(See Figure 23).

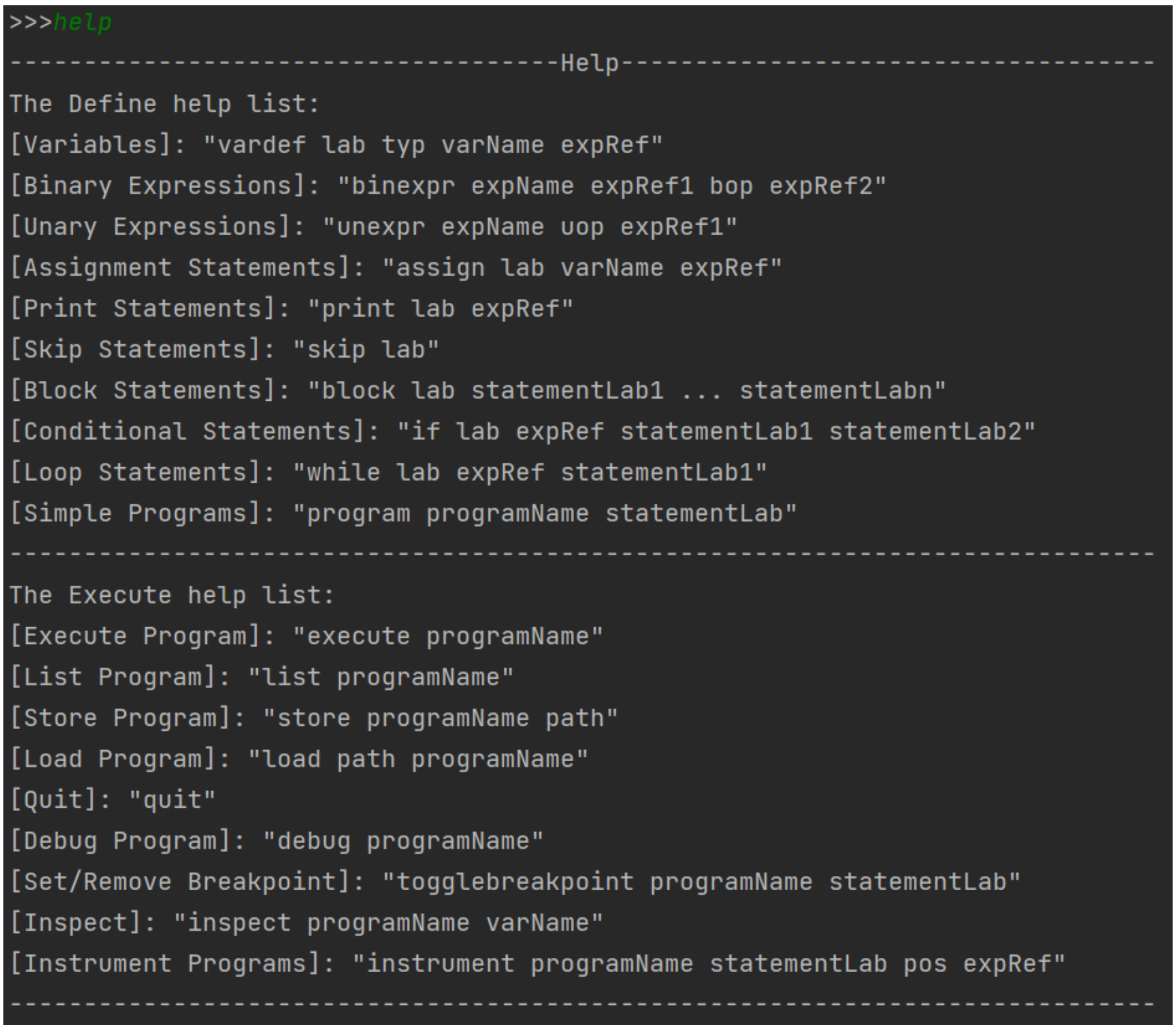


```
>>>load d:\pro1.simple program1
>>>list program1
vardef vardef1 int x 0
binexpr exp4 x <= 10
binexpr exp1 x % 2
binexpr exp2 exp1 == 0
print print1 x
skip skip1
if if1 exp2 print1 skip1
binexpr exp3 x + 1
assign assign1 x exp3
block block1 if1 assign1
while while1 exp4 block1
block block2 vardef1 while1
program program1 block2
>>>instrument program1 block1 after 1
>>>execute program1
[0] {1} {1} [2] {1} {1} [4] {1} {1} [6] {1} {1} [8] {1} {1} [10] {1}
>>>instrument program1 block1 after 1
>>>execute program1
[0] {1} {1} {1} {1} [2] {1} {1} {1} {1} [4] {1} {1} {1} {1} [6] {1} {1} {1} {1} [8] {1} {1} {1} {1} [10] {1} {1}
```

Figure 23 : instrument expression reference

To get a description of all command statements(See Figure 24), input the command:

help



```
>>>help
-----------------------------------Help-----------------------------------
The Define help list:
[Variables]: "vardef lab typ varName expRef"
[Binary Expressions]: "binexpr expName expRef1 bop expRef2"
[Unary Expressions]: "unexpr expName uop expRef1"
[Assignment Statements]: "assign lab varName expRef"
[Print Statements]: "print lab expRef"
[Skip Statements]: "skip lab"
[Block Statements]: "block lab statementLab1 ... statementLabn"
[Conditional Statements]: "if lab expRef statementLab1 statementLab2"
[Loop Statements]: "while lab expRef statementLab1"
[Simple Programs]: "program programName statementLab"
--------------------------------------------------------------------------
The Execute help list:
[Execute Program]: "execute programName"
[List Program]: "list programName"
[Store Program]: "store programName path"
[Load Program]: "load path programName"
[Quit]: "quit"
[Debug Program]: "debug programName"
[Set/Remove Breakpoint]: "togglebreakpoint programName statementLab"
[Inspect]: "inspect programName varName"
[Instrument Programs]: "instrument programName statementLab pos expRef"
--------------------------------------------------------------------------
```

Figure 24 : Help Page

End of Command-Line-Based Simple Program Interpreter project report

COMP2021 Group 9